ReCoSoC 2010

KIT SCIENTIFIC REPORTS 7551

Proceedings of the

5th International Workshop on Reconfigurable Communication-centric Systems on Chip 2010 – ReCoSoC'10

May 17-19, 2010
Karlsruhe, Germany

KIT Scientific Publishing

Michael Hübner, Loïc Lagadec, Oliver Sander, Jürgen Becker (eds.)

Proceedings of the 5th International Workshop on Reconfigurable Communication-centric Systems on Chip 2010 – ReCoSoC'10

May 17-19, 2010
Karlsruhe, Germany

Karlsruhe Institute of Technology

KIT SCIENTIFIC REPORTS 7551

Proceedings of the 5th International Workshop on Reconfigurable Communication-centric Systems on Chip 2010 – ReCoSoC‘10

May 17-19, 2010
Karlsruhe, Germany

Michael Hübner
Loïc Lagadec
Oliver Sander
Jürgen Becker
(eds.)

KIT Scientific Publishing

Report-Nr. KIT-SR 7551

Umschlagsbild:
Wikimedia Commons. Fotograf: Meph666

Impressum

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

KIT – Universität des Landes Baden-Württemberg und nationales
Forschungszentrum in der Helmholtz-Gemeinschaft

KIT Scientific Publishing 2010
Print on Demand

ISSN 1869-9669
ISBN 978-3-86644-515-4

ReCoSoC'10 Reconfigurable Communication-centric Systems on Chip

Michael Hübner, Karlsruhe Institute of Technology, Karlsruhe, Germany
Loïc Lagadec, Université de Bretagne Occidentale, Lab-STICC, Brest, FRANCE

The fifth edition of the Reconfigurable Communication-centric Systems-on-Chip workshop (ReCoSoC 2010) was held in Karlsruhe, Germany from May 17th to May 19th, 2010.
ReCoSoC is intended to be a periodic annual meeting to expose and discuss gathered expertise as well as state of the art research around SoC related topics through plenary invited papers and posters. Similarly to the event in 2008 and the years before, several keynotes given by internationally renowned speakers as well as special events like e.g. tutorials underline the high quality of the program.

ReCoSoC is a 3-day long event which endeavors to encourage scientific exchanges and collaborations. This year again ReCoSoC perpetuates its original principles: thanks to the high sponsoring obtained from our partners registration fees will remain low.

Goals:
ReCoSoC aims to provide a prospective view of tomorrow's challenges in the multi-billion transistor era, taking into account the emerging techniques and architectures exploring the synergy between flexible on-chip communication and system reconfigurability.

The topics of interest include:

- Embedded Reconfigurability in all its forms
- On-chip communication architectures
- Multi-Processor Systems-on-Chips
- System & SoC design methods
- Asynchronous design techniques
- Low-power design methods
- Middleware and OS support for reconfiguration and communication
- New paradigms of computation including bio-inspired approaches

A special thank goes to the local staff, especially to the local chair Oliver Sander, Mrs. Hirzler, Mrs. Bernhard and Mrs. Daum who enabled a professional organization before and while the conference. Thanks to Gabriel Marchesan, the web-page of the conference was always up to date and perfectly organized. Furthermore Prof. Becker supported the conference through his group at the Institute for Information Processing Technology. We also thank the International Department for offering the providing the „Hector Lecture Room“ for the conference.

Michael Hübner
Loïc Lagadec
ReCoSoC 2010 Program Co-Chairs

Program Committee

Jürgen Becker	Karlsruhe Institute of Technology	Germany
Pascal Benoit	LIRMM, Montpellier	France
Koen Bertels	TU Delft	Netherlands
Christophe Bobda	University of Potsdam	Germany
Lars Braun	Karlsruhe Institute of Technology	Germany
René Cumplido	INAO	México
Debatosh Debnath	Oakland University	USA
Jean-Luc Dekeyser	University of Lille	France
Didier Demigny	ENSSAT, Lannion	France
Peeter Ellervee	Tallinna Tehnikaülikool	Estonia
Christian Gamrat	CEA	France
Georgi Gaydadjiev	TU Delft	Netherlands
Manfred Glesner	TU Darmstadt	Germany
Diana Goehringer	Fraunhofer IOSB	Germany
Jim Harkin	University of Ulster	Northern Ireland
Andreas Herkersdorf	Technische Universität München	Germany
Thomas Hollstein	TU Darmstadt	Germany
Michael Hübner	Karlsruhe Institute of Technology	Germany
Leandro Indrusiak	University of York	UK
Loic Lagadec	Université de Bretagne Occidentale	France
Heiner Litz	Universität Heidelberg	Germany
Patrick Lysaght	Xilinx Inc.	USA
Fearghal Morgan	NUI Galway	Ireland
Johnny Öberg	KTH	Sweden
Ian O'connor	LEOM, Lyon	France
Katarina Paulsson	Ericsson	Sweden
J.-L. Plosila	University of Turku	Finland
Bernard Pottier	University of Bretagne Occidentale	France
Ricardo Reis	UFRGS	Brazil
Michel Robert	LIRMM, Montpellier	France
Alfredo Rosado	Universitat de Valencia	Spain
Eduardo Sanchez	EPFL	Switzerland
Oliver Sander	Karlsruhe Institute of Technology	Germany
Gilles Sassatelli	LIRMM, Montpellier	France
Tiberiu Seceleanu	University of Turku	Finland
Dirk Stroobandt	Universiteit Gent	Belgium
Lionel Torres	LIRMM, Montpellier	France
Francois Verdier	Université de Cergy-Pontoise	France
Nikos Voros	Technological Educational Institute of Mesolonghi	Greece
Hans-Joachim Wunderlich	Universität Stuttgart	Germany
Peter Zipf	TU Darmstadt	Germany

Table of Contents

A Self-adaptive communication protocol allowing fine tuning between flexibility and performance in Homogeneous MPSoC systems

Remi Busseuil, Gabriel Marchesan Almeida, Sameer Varyani, Pascal Benoit, Gilles Sassatelli
Laboratoire d'Informatique,
de Robotique et de Microelectronique
de Montpellier (LIRMM)
Montpellier, France
Email: firstname.lastname@lirmm.fr

***Abstract*—MPSoC have become a popular design style for embedded systems that permit devising tradeoffs between performance, flexibility and reusability. While most MPSoCs are heterogeneous for achieving a better power efficiency, homogeneous systems made of regular arrangements of a unique instance of a given processor open interesting perspectives in the area of on-line adaptation.**

Among these techniques, task migration appears very promising as it allows performing load balancing at run-time for achieving a better resources utilization. Bringing such a technique into practice requires devising appropriate solutions in order to meet quality of service requirements. This paper puts focus on a novel technique that tackles the difficult problem of inter-task communication during the transient phase of task migration. The proposed adaptive communication scheme is inspired from TCP/IP protocols and shows acceptable performance overhead while providing communication reliability at the same time.

I. Introduction

Thanks to the technology shrinking techniques, an integrated circuit can include an exponentially increasing number of transistors. This trend plays an important role at the economic level, although the price per transistor is rapidly dropping the NRE (Non-Recurring Engineering) costs, and fixed manufacturing costs increase significantly. This pushes the profitability threshold to higher production volumes opening a new market for flexible circuits which can be reused for several product lines or generations, and scalable systems which can be designed more rapidly in order to decrease the Time-to-Market. Moreover, at a technological point of view, current variability issues could be compensated by more flexible and scalable designs. In this context, Multiprocessor Systems-on-Chips (MPSoCs) are becoming an increasingly popular solution that combines flexibility of software along with potentially significant speedups.

These complex systems usually integrate a few mid-range microprocessors for which an application is usually statically mapped at design-time. Those applications however tend to increase in complexity and often exhibit time-changing workload which makes mapping decisions suboptimal in a number of scenarios. These facts challenge the design techniques and methods that have been used for decades and push the community to research new approaches for achieving system adaptability and scalability. The most promising development tends to be homogeneous structures based on a Network on Chip (NoC), with distributed identical nodes containing both computing capabilities and memory. Such systems allow using advanced techniques that permit optimizing online application mapping. Among these techniques, this paper puts focus on dynamic load balancing based on task migration.

Migrating processing tasks at runtime while ensuring real-time constraints are met require devising precise deterministic protocols to guarantee application consistency. In this paper, we propose an adaptive communication protocol, based on TCP and UDP models, which ensure determinism of critical migration mechanisms while providing enhanced useful services such as port opening.

This paper is organized as follows: section 2 presents related works in the field of communication inside distributed structures and examples of Network on Chip protocols. Section 3 introduces the platform developed regarding scalability, adaptability and reuse issues used to test our communication protocol, named HS-Scale. Section 4 describes how to ensure determinism and reliability with our protocol facing the problems raised by this type of platform. Finally, section 5 shows some results concerning the performance of our protocol. Section 6 draws some conclusions on the presented work and gives some perspectives about other upcoming challenges of the area.

II. State-of-the-art

In this section we will discuss the communication trends inside the new emerging MPSoC architecture, *i.e.* based on NoC, with distributed memory architecture, and a message passing communication model. In this context, we will then see different task migration techniques. Finally, we will present an overview of existing NoC communication protocols.

A. Communication inside distributed memory architecture

Nowadays, distributed memory structures tend to become the most attracted solution to achieve scalability and reuse challenges in new emerging architectures. To ensure coherency between the hardware and the software in such systems, the Message Passing Model of computation is the most commonly used - except in some marginal architectures like COMA [1] or ccNUMA [2] MPSoC. This model of computation is based on explicit communication between tasks. Here communications among tasks take place through messages and are implemented with functions allowing reading and writing to communication channels. Synchronizations between tasks are explicit and made by blocking reading or writing primitives. CORBA, DCOM, SOAP, and MPI are examples of message passing models. Message Passing Interface (MPI) is the most popular implementation of the message passing model, and, to the best of our knowledge, embedded implementations exist only for this model [3].

One of the other hot issues in today computing architecture is load balancing and resource usage optimization. Indeed, in nowadays MPSoC, task migration techniques have been mainly studied, to reduce hotspot and increase the overall resource usage. Next paragraph gives a brief overview of some task migration techniques.

B. *Task migration*

For shared memory systems such as today's multi-core computers, task migration is facilitated by the fact that no data or code has to be moved across physical memories: since all processors are entitled to access any location in the shared memory, migrating a task comes down to electing a different processor for execution. But in the case of multiprocessor-distributed memory/message passing architectures, both process code and state have to be migrated from a processor private memory to another, and synchronizations must be performed using exchanged messages.

Task migration has also been explored for decreasing communication overhead or power consumption [4]. In [5], authors present a migration case study for MPSoCs that relies on the μClinux operating system and a check pointing mechanism. The system uses the MPARM framework [6], and although several memories are used, the whole system supports data coherency through a shared memory view of the system. In [7] authors present an architecture aiming at supporting task migration for a distributed memory multiprocessor-embedded system. The developed system is based on a number of 32-bit RISC processors without memory management unit (MMU). The used solution relies on the so-called *task replicas* technique; tasks that may undergo a migration are present on every processor of the system. Whenever a migration is triggered, the corresponding task is respectively inhibited from the initial processor and activated in the target processor.

Although the efficiency of these techniques has been proven, none of these papers mention the communication issues due to task migration. Among those task migration specific protocols, we can cite: localization of a task, set up of the communication or maintenance of communication channel during task move.

C. *Network on Chip communication*

Network communication has been widely studied for many years, mainly in the context of cluster of PCs and High Performance Computing. However, Network on Chip communication, even if a lot of concepts can be taken from those studies, differs in many manners from traditional Network. A NoC, for example, has rarely to be dynamically expanded, so it does not need a *live connection* service. However, to provide reuse and scalability, NoC protocol should be made for an arbitrary number of nodes. [8] illustrates some techniques concerning on-chip communication, like energy-efficient protocols, or lightweight encapsulation. As in standard Network, reordering and adaptive routing is often provided, to avoid saturation of a node, like in [9]. At an architectural level, regular structure like 2D mesh [10] is the most frequently used, but other structures exist, like [11] which uses an octagon for example. This last article proposes both packet switching, where each packet is redirected individually, and circuit switching, where a unique channel is opened during the whole communication process.

As for task migration, task placement can increase communication throughput. [12] proposes a circuit-switch NoC statically programmed at compile time to optimize the overall network bandwidth. A lightweight NoC architecture called HERMES, based on a X then Y routing is proposed in [10]. It provides simple packet switching network with unique predictable routing for each packet from the same sender and receiver. Hence, neither reordering nor acknowledgment is necessary.

One particularity of these protocols is to be hardware dependent: the structure of the NoC will influence significantly the software communication policy. We will see in the next paragraph the platform used to develop our new protocol.

III. HS-Scale (Hardware and Software Scalable platform)

The key motivations of our approach being scalability and self-adaptability, the system presented in this paper is built around a distributed memory/message passing system that provides efficient support for task migration. The decision-making policy that controls tasks processes is also fully distributed for scalability reasons. This system therefore aims at achieving continuous, transparent and decentralized runtime task placement on an array of processors for optimizing application mapping according to various potentially time changing criteria.

A. *System overview*

The architecture is made of a homogeneous array of PE (Processing Elements) communicating through a packet switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each NPU, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System which runs on each NPU. The structural view of the NPU is depicted in Figure 1.

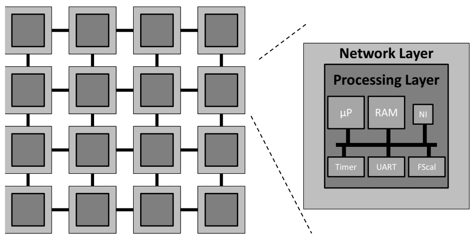


Fig. 1. HS-Scale structural description

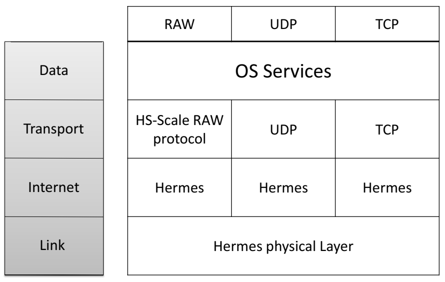


Fig. 2. HS-Scale available communication protocols

The NPU is built of two main layers, the network layer and the processing layer. The Network layer is essentially a compact routing engine (XY routing). The processing layer is based on a simple and compact RISC microprocessor, its static memory and a few peripherals (a timer, an interrupt controller, an UART and a frequency scaler) as shown in Figure 1. A multitasking microkernel implements the support for time multiplexed execution of multiple tasks [13].

The communication framework of HS-Scale is derived from the Hermes Network-on-chip [10]. The lightweight operating system we use was designed for our specific needs, inspired by the RTOS of Steve Roads [14]. Despite being small (35 KB), this kernel does preemptive switching between tasks and also provides them with a set of communication primitives that are presented later. The OS is capable of dynamic task loading and dynamic task migration.

B. Self-adaptive mechanisms

The platform is entitled to take decisions that relate to application implementation through task placement. These decisions are taken in a fully decentralized fashion as each NPU is endowed with equivalent decisional capabilities. Each NPU monitors a number of metrics that drive an application-specific mapping policy; based on these information a NPU may decide to push or attract tasks which results in respectively parallelizing or serializing the corresponding tasks execution, as several tasks running onto the same NPU are executed in a time-sliced manner.

Mapping decisions are specified on an application-specific basis in a dedicated operating system service. Although the policy may be focused on a single metric, composite policies are possible. Three metrics are available to the remapping policy for taking mapping decisions:

- NPU workload
- FIFO queues filling level
- Task distance

NPU workload is measured as the amount of time used to process the user tasks - i.e. excluding the idle task and the communication tasks. Task distance corresponds to the number of hop a packet needs to go through during a communication between two tasks. As the Network structure is a 2D mesh, this measure can be computed as the Manhattan distance between the two nodes hosting the tasks.

Several migration policies have been developed, like the use of a static threshold to start a migration. In this case, the task is migrated to the neighbor with the best value of the selected metric - for example the neighbor with the lowest cpu workload. For more information about the task migration policies, we invite the reader to read our previous paper [15].

C. Communication system

The Network of HS-Scale is based on the NoC HERMES [10]. It provides a low area overhead packet-switching network thanks to a simple *X then Y* routing algorithm. Only two fields are needed to encapsulate a HERMES packet: one for the sender and receiver addresses, and the second for the number of 32-bits words inside the packet.

However, such protocol is too simple to provide high level services usually used in real-time multi-task operating systems. In the standard usually used Internet encapsulation model, 4 layers of functionality are provided: the link layer, the Internet layer, the transport layer and the application layer [16]. The HERMES protocol supply link layer and Internet layer functionality. Transport and data layers are so implemented in software - as OS services - using the concept of TCP. To keep compatibility between HERMES and IP, XY addresses of HERMES have been statically mapped to IP addresses. Transport layer was adapted to provide the same services as TCP and UDP: the notion of ports have been raised, redirection and retransmission have been included in TCP. A checksum has been optionally made to provide reliability in non reliable networks. As this network can be considered as reliable, this feature has not been used, but it makes the protocol more generic in term of hardware possible platform.

Figure 2 shows the different protocols implanted in HS-Scale. A RAW protocol, simply using Hermes routing layer and with packets directly given to the OS has been made to provide a base rate of the bandwidth. When a task needs a communication channel, it uses an unused port which will become the identity of the communication channel. When the communication is closed, the port is marked as unused again.

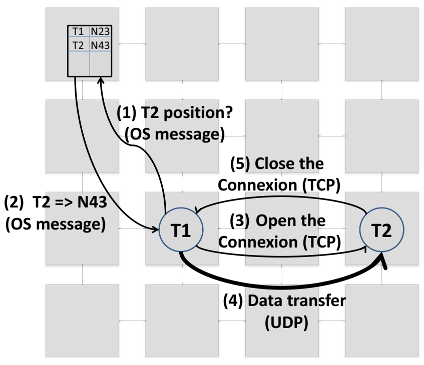

T1 N23
T2 N43
(1) T2 position? (OS message)
(2) T2 => N43 (OS message)
(5) Close the Connexion (TCP)
(3) Open the Connexion (TCP)
(4) Data transfer (UDP)
T1
T2

Fig. 3. *Inter-node* task communication protocol

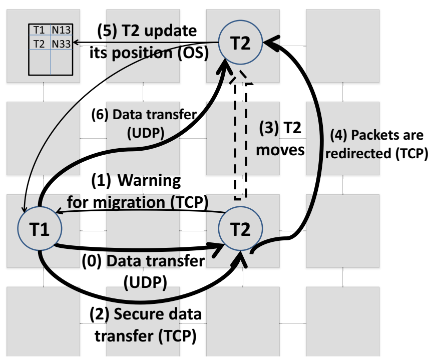

T1 N13
T2 N33
(5) T2 update its position (OS)
(6) Data transfer (UDP)
(3) T2 moves
(4) Packets are redirected (TCP)
(1) Warning for migration (TCP)
(0) Data transfer (UDP)
(2) Secure data transfer (TCP)
T1
T2
T2

Fig. 4. Communication protocol during task migration

IV. Communication issues in self-adaptive homogeneous platforms

A. Different types of communication

A homogeneous and regular platform like HS-Scale has an application domain more generic than heterogeneous, application specific platform. Thus, the communication has to face numerous issues to provide both genericity and performance. We can distinguish different types of communication in HS-Scale:

- Communication between tasks *intra-node*, i.e. inside the same node
- Communication between tasks *inter-node*, i.e. between two different nodes
- Service messages, provided by the Operating system of a node to another one
- Exception messages, provided by a task to an Operating system to request a service

For *intra-node* communication between tasks, we use simple FIFO queue handled by the operating system of the node. The operating system provides some basic primitives of MPI (Message Passing Interface), so tasks can easily communicate.

For *inter-node* communication between tasks, that protocol is more complicated. Figure 3 represents the concept of the protocol. First, we need to open a connection between the two tasks: a service message is sent to the master node to find the receiver task. As RAW protocol does not provide any opening or close of connection, the TCP *three-steps* handshake protocol is used. After opening, we need to transfer the information with the best performance we have, with no need of Quality of Service: so the UDP protocol is used. At the end of the transmission, the TCP *four-steps* closing protocol is used. This protocol is transparent for the task itself: the Operating System is handling it, and only the MPI services are provided. The Operating System has to check whether the receiving task is on the same node or abroad.

Service messages and Exception messages need Quality of Service to ensure reliability of the Network. For this purpose, TCP protocol is used to send messages.

B. Task Migration

Communication during task migration main issue can be expressed as follows: how to keep the maximum performance during the transfer of a task from a NPU to another. Without any particular protocol, the simplest way to ensure no dropping of packets is to close the communication during task migration. Although this method can be considered as reliable, the opening and closing protocol plus the loss of the connection will bring a big overhead in term of performance.

To ensure no loss of packets during migration, we have to focus our attention on two points: first, we need to ensure the reception and the order of the packets, so that no packets are dropped. Second, the packets have to be redirected, in case they go to the wrong node. Those features are again part of the TCP protocol, so the idea is also here to use TCP to ensure the reliability of the system.

Figure 4 shows the different steps of the communication during a task migration - here, the receiver. Before the migration, the task which wants to migrate send a message to the tasks communicating with itself, warning them that it wants to communicate and so switch the communication in TCP (action 1 in the figure). The migration can begin when the task receives at least one TCP packet from every sender tasks (2). If packets arrive during migration, they are redirected to the new NPU which receives the task (3): if the task is ready, the packet is consumed, if not, the packet is stored in a fifo, and dropped when the fifo is full. When the migration is complete, the task sends a message to the tasks communicating with it to update its position and to switch in UDP again. As TCP provides reordering, if TCP packets arrive after because of redirection, the task can reorder the packets.

V. Performance of the auto-adaptive protocol

A. Maximum bandwidth achievable

The first purpose that we need to focus on in our communication protocol is the bandwidth. Indeed, the bandwidth has to be tuned to fit the purpose of the chip. Figure 5 shows the bandwidth for the two protocols available, comparing them with a raw transfer as described before. The comparison is made with different size of packets and with different amount of data. A timer in each node measures the time to transfer, to

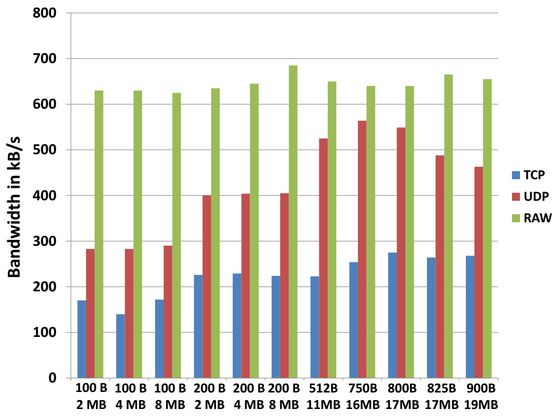


Fig. 5. TCP, UDP and RAW Bandwidth versus packet size and amount of data transferred

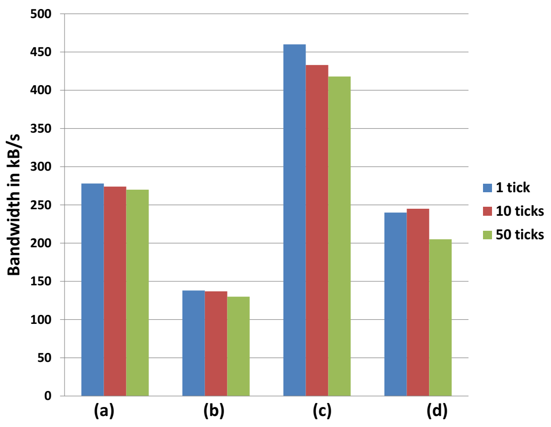


Fig. 6. TCP and UDP Bandwidth when the CPU is idle or in load

compute the bandwidth. The RAW communication can achieve an average bandwidth more than 600 kB/s, when the UDP can achieve 550 kB/s at best, and TCP 300 kB/s. The performance delta between RAW and UDP, less than 10%, proves the efficiency of our UDP like protocol. The TCP protocol, however, has a bandwidth 50% smaller than RAW communication, but this protocol has been developed to be used on precise, few communication process like OS messages, opening or closing of a connection, or task migration.

The second point raised by figure 5 is the bandwidth variations compared to the segmentation of the data and the amount of data transmitted. If in RAW, the variations are quite low, they can go up to 45% for the UDP and TCP protocols. The low bandwidth obtained with really small packets (100 or 200 bytes) can be explained by the fact that encapsulation in UDP and TCP is huge, around 50 bytes, which makes the payload of each packets really small. In the contrary, for really huge packets, hardware and software capability of a NPU is too low to register the whole packet in one tick, which makes the operating system run a rescheduling, and so it lowers the bandwidth.

However, the optimum size, around 750 bytes per packets, is really dependent of the platform. In hardware first, the computation capability of the CPU to deal with a packet will influence the overall processing time. Time to process a packet will vary in function of frequency, CPU architecture or Processing Unit design - with the addition of a dedicated Network encoder/decoder for example. But in software too, the optimum can greatly vary. Indeed, this optimum depends on the average time between two rescheduling of the communication procedure. Depending on CPU charge, the communication procedure rescheduling will proceed less often, and so the optimum will change. Hence, next paragraph will show results about bandwidth variations with a CPU in charge.

B. *Performance in charge*

As the CPU is both used for communication decoding and computation, it is interesting to see the influence of CPU charge on the bandwidth. Figure 6 shows the bandwidth of UDP/TCP communication when - for (a) in TCP and (c) in UDP - the CPU is in a *communication only* mode, i.e. there is no other task running on the CPU, and when - for (b) in TCP and (d) in UDP - the CPU is in *heavy loaded* mode, i.e. when the CPU runs a mjpeg decoder based on 3 interdependent tasks. The packet size is fixed to 750 bytes, and the measures are made on a 16MB transfer. We can see that the CPU load can lower the communication bandwidth down to 50% of its nominal value. This reduction can be explained by the heavy computation needed to process TCP or UDP packet with a general purpose CPU like those used in HS-Scale. Even so, the software implementation of TCP encapsulation and decapsulation, responsible of these results, is not a bad choice here: as this protocol is considered to be used for special purpose communication, which can be considered as negligible in terms of data transferred compared to stream inter-process communication, the throughput is not a critical issue. In this case, this implementation seems more appropriate than a hardware one, which would consume area.

The second point stressed by figure 6 is the time between two rescheduling of the communication protocol. The Operating System in HS-Scale uses simple round robin rescheduling with fixed size time slots called ticks: the rescheduling procedure runs after each tick. As the communication procedure is considered as a task, the time to reschedule will vary in function of the load of the CPU and of the *sleeping time* parameter. This parameter is the number of ticks between the last proceeding of the communication task and its insertion in the round robin queue. It can be tuned from 1 tick, which correspond to each task being rescheduled only once between two communication runs, to any positive numbers.

Figure 7 illustrates this principle of rescheduling. In (a) and (b), the CPU is in *communication only* mode: in this case, the rescheduling should be as often as possible, to avoid empty slots, like in (b). In (c) and (d), the CPU is in *heavy loaded* mode: in this case, the rescheduling time can influence the computation time, but also the bandwidth. In term of computation time, the ratio of time spent on the

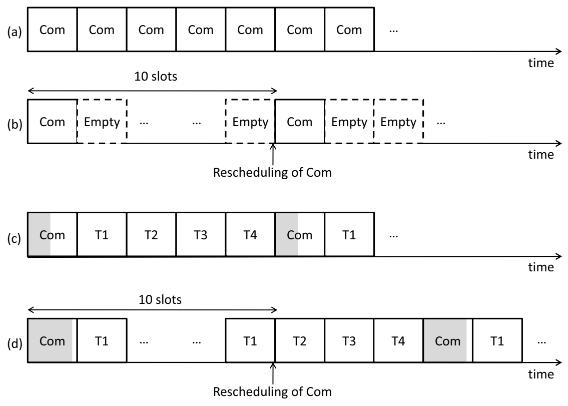


Fig. 7. Communication protocol, rescheduling issues

communication task is, on the average:

$$P = \frac{1}{(N-1)+T}$$

where N is the number of ticks and T the number of tasks. In term of bandwidth also, the variations are no more monotonous versus the number of ticks: if we have a situation where the time spent to receive - in gray in fig. 7 (c) and (d) - is smaller than a timeslot, we can have a situation, like illustrated in fig. 7, where the average time spent to receive is bigger with few ticks than without. This situation can be observed in figure 6 (d), where the bandwidth is higher with 10 ticks than with 1 tick. We can conclude that the rescheduling time can be an issue in this kind of architecture, and that it has to be tuned accurately depending on the application. If the CPU has a variable load, a dynamically variable time to reschedule the communication, depending on the receiving speed and the number of tasks, is certainly the best solution.

VI. Conclusion

Future new MPSoC architectures will have to ensure flexibility and adaptability to face the new challenges raised by technology shrinking and computing requirement. For this purpose, array of identical software independent nodes linked by a NoC seems to be the most promising solution: it can ensure generic computation with high performance thanks to a good load balancing strategy.

This article describes an adaptive communication protocol purposely made to face dynamic task migration issues in such homogeneous structures. As every node is independent, it has to deal with the asynchronicity of each node, with providing sequential behavior of critical codes. For this purpose, this protocol has been built using inspiration of TCP and UDP features. Thanks to their historically proved reliability, scenarios of secure communication channel creation and conservation during task migration has been displayed. Finally, performance issues show the interest of such protocol with a really small overhead for UDP-like transaction.

References

[1] F. Dahlgren and J. Torrellas, *Cache-only memory architectures*, Computer, vol. 32, no. 6, pp. 7279, 1999.
[2] P. Stenstrm, T. Joe, and A. Gupta, *Comparative performance evaluation of cache-coherent numa and coma architectures*, ISCAS conference, 1992.
[3] M. Saldana and P. Chow, *TDM-MPI: an MPI implementation for multiple processors across multiple FPGAs*, Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2006.
[4] S. Carta, M. Pittau, A. Acquaviva, et al., *Multi-processor operating system emulation framework with thermal feedback for systems-on-chip*, Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI), 2007.
[5] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, *Supporting task migration in multi-processor systems-onchip: a feasibility study*, Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2006.
[6] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, *MPARM: exploring the multi-processor SoC design space with systemC*, Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 2005.
[7] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva, *Impact of task migration on streaming multimedia for embedded multiprocessors: a quantitative evaluation*, Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2007.
[8] V. Raghunathan, M. B. Srivastava and R. K. Gupta, *A survey of techniques for energy efficient on-chip communication*, DAC Conference, 2003.
[9] P Guerrier and A. Greiner, *A Generic Architecture for On-Chip Packet-Switched Interconnections*, DATE Conference, 2000.
[10] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Mller and Luciano Ost, *HERMES: an infrastructure for low area overhead packet-switching networks on chip*, IEEE VLSI Journal, 2004
[11] Faraydon Karim, Anh Nguyen and Sujit Dey, *An Interconnect Architecture for Networking Systems on Chips*, IEEE micro, 2002
[12] Jian Liang, Andrew Laffely, Sriram Srinivasan, and Russell Tessier, *An architecture and compiler for scalable on-chip communication*, IEEE VLSISoC, 2004
[13] Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert, *An AdaptiveMessage Passing MPSoC Framework*, International Journal of Reconfigurable Computing (IJRC) journal, 2009
[14] Steve Rhoads, *Plasma Most MIPS I(TM)*, http://www.opencores.org/project,plasma
[15] G. Marchesan Almeida, N. Saint-Jean, S. Varyani, G. Sassatelli, P. Benoit and L. Torres, *Exploration of Task Migration Policies on the HS-Scale System*, Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC08), 2008
[16] Vinton Cerf, Yogen Dalal and Carl Sunshine, *SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM*, 1974

Instruction Set Simulator for MPSoCs based on NoCs and MIPS Processors

Leandro Möller[1], André Rodrigues[1], Fernando Moraes[2], Leandro Soares Indrusiak[3], Manfred Glesner[1]

[1] Darmstadt University of Technology - Institute of Microelectronic Systems - Darmstadt, Germany
[2] Faculty of Informatics - Catholic University of Rio Grande do Sul - Porto Alegre, Brazil
[3] Department of Computer Science - University of York - York, United Kingdom
Email: moller@mes.tu-darmstadt.de

Abstract

Even though Multiprocessor System-on-Chip (MPSoC) is a hot topic for a decade, Instruction Set Simulators (ISSs) for it are still scarce. Data exchange among processors and synchronization directives are some of the most required characteristics that ISSs for MPSoCs should supply to really make use of the processing power provided by the parallel execution of processors. In this work a framework for instantiating ISSs compatible with the MIPS processor is presented. Communication among different ISS instances is implemented by message passing, which is actually performed by packets being exchanged over a NoC. The NoC, the ISS and the framework that controls the co-simulation between them are all implemented in Java. Both ISS and the framework are free open-source tools implemented by third parties and available on the internet.

1. Introduction

Multiprocessor systems have become a standard in the computer industry since the release of the Intel Pentium D in 2005 [1]. Since then, processor manufacturers have focused in multi-core architectures to raise the processing power, favoring a larger number of cores instead of trying to achieve higher clock speeds, avoiding also the complexity of superscalar pipelines. While executing several small applications in parallel have a significant improve in performance with actual multiprocessor systems, a unique complex application needs a careful development to use wisely this processing power. It is not simply to write the application code with multiple threads, but each thread has to be really executing in the same time as the other threads, instead of paused in a *wait* directive.

While communication infrastructures based on bus have been sufficient for multiprocessor systems so far, the increase of number of cores and data transfer associated will demand a more complex on-chip interconnection. For this purpose Networks-on-Chip (NoCs) have arisen as a scalable solution to future increase of number of cores. The use of a NoC represents no direct changes to the developer of the complex application, but it counts when the execution time of the complex application is being analyzed.

The design space exploration of the scenario presented in the previous paragraphs and the tools to aid the development of complex applications are the goal of this work. A MIPS-like processor was connected to the HERMES NoC and presented in [2]. In [2] the debug of complex applications are implemented based on print directives. The work presented here improves the debugability by providing an Instruction Set Simulator (ISS) for the MIPS processor while considering the communication time and traffic under simulation in the NoC.

The ISS used in this work is the MARS ISS, developed by the Missouri State University [3]. This ISS was connected the RENATO NoC model [4], which is an actor-oriented model based on the HERMES NoC. The simulation environment used to control both the simulation of the NoC and the ISS is the Ptolemy II [5], developed by the EECS department at UC Berkley.

The rest of this work is divided as follows. Section 2 presents other ISSs targeting MPSoC architectures. A background about the tools and basic information required to understand this work is presented in Section 3. Section 4 presents how the communication among ISSs takes place. Section 5 presents timing delays of the system and Section 6 concludes this work.

2. Related Works

In this section different MPSoCs that have tools for debugging their embedded software are presented. Table 1 summarizes the most important information of these works and adds the work proposed in this paper. As presented in Table 1, all works use SystemC as simulation engine and memory mapped techniques to communicate with other processors, except the work proposed on this paper that uses the Ptolemy II simulation engine and the message passing technique to communicate with other processors.

MPARM [6] uses ARM processors connected through AMBA bus to compose the MPSoC. Multiprocessor applications are debugged with the SWARM ISS, which is developed in C++ and was wrapped to communicate with the MPSoC simulated in SystemC. The platform allows booting multiple parallel uClinux kernels on independent processors.

STARSoC [7] uses OpenRisc1200 processors connected through Wishbone bus. Debugging is implemented with the

OR1Ksim ISS, which is implemented in C language. The OR1Ksim also allows to be remote operated using GDB. Operating System is not yet supported.

HVP [8] supports several processors and therefore several ISSs. The work presented MPSoCs that contain ARM9 processors using ARM's ISS and in-house VLIW and RISC processors debugged by the LisaTek ISS. The ARM processors execute a lightweight operating system (name was not disclosed). The communication among processors was reported to be AMBA among ARM processors and SimpleBus among the in-house processors used.

SoClib [9] is a project developed jointly by 11 laboratories and 6 industrial companies. It contains simulation models for processor cores, interconnect and bus controllers, embedded and external memory controllers, or peripheral and I/O controllers. The MPSoC accepts the following processor cores: MIPS-32, PowerPC-405, Sparc-V8, Microblaze, Nios-II, ST-231, ARM-7tdmi and ARM-966. The GDB client/server protocol has been implemented to interface with these processors. The following operating systems are supported: DNA/OS, MutekH, NetBSD, eCos and RTEMS. Several bus and NoCs with different topologies wrapped with the VCI communication standard were ported and presented at www.soclib.fr.

The proposed work is based on a MIPS-like processor, implemented in hardware by the Plasma processor available for free at Opencores [10] and implemented by MARS [3] when simulating the processor as an ISS. All previous works used SystemC as simulation environment; this work uses Ptolemy II [5]. This work also differs from the others because it exchanges data between processors by using the native protocol of the NoC, therefore no extra translation is needed before sending and receiving packets.

Table 1 – MPSoCs that have tools for debugging embedded software.

Work ID	Simulation engine	Processor	Communication	Data exchange	ISS	OS
MPARM	SystemC	ARM	Bus (Amba)	Memory	SWARM	uClinux
STARSoC	SystemC	OpenRisc 1200	Bus (Wishbone)	Memory	OR1Ksim	No
HVP	SystemC	Several	Bus (several)	Memory	Several	Yes
SoClib	SystemC	Several	Bus / NoC (several)	Memory	GDB	several
Proposed	Ptolemy II	Plasma (MIPS)	NoC (Hermes)	Message	MARS	No

3. Background

This session reviews the required infrastructure to build our MPSoC simulation environment.

3.1. Ptolemy II

Ptolemy II [5] is a framework developed by the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley and it is implemented in Java. The key concept behind Ptolemy II is the use of well-defined models of computation to manage the interactions between various actors and components. In this work only the Discrete Event (DE) model of computation was used, but others are available on Ptolemy II.

In DE, the communication between actors is modeled as tokens being sent across connections. The sent token and its timestamp constitute an event. When an actor receives an event, it is activated and a reaction might occur, which may change the internal state of the actor and / or generate new events, which might in its turn generate other reactions. The events are processed chronologically [5].

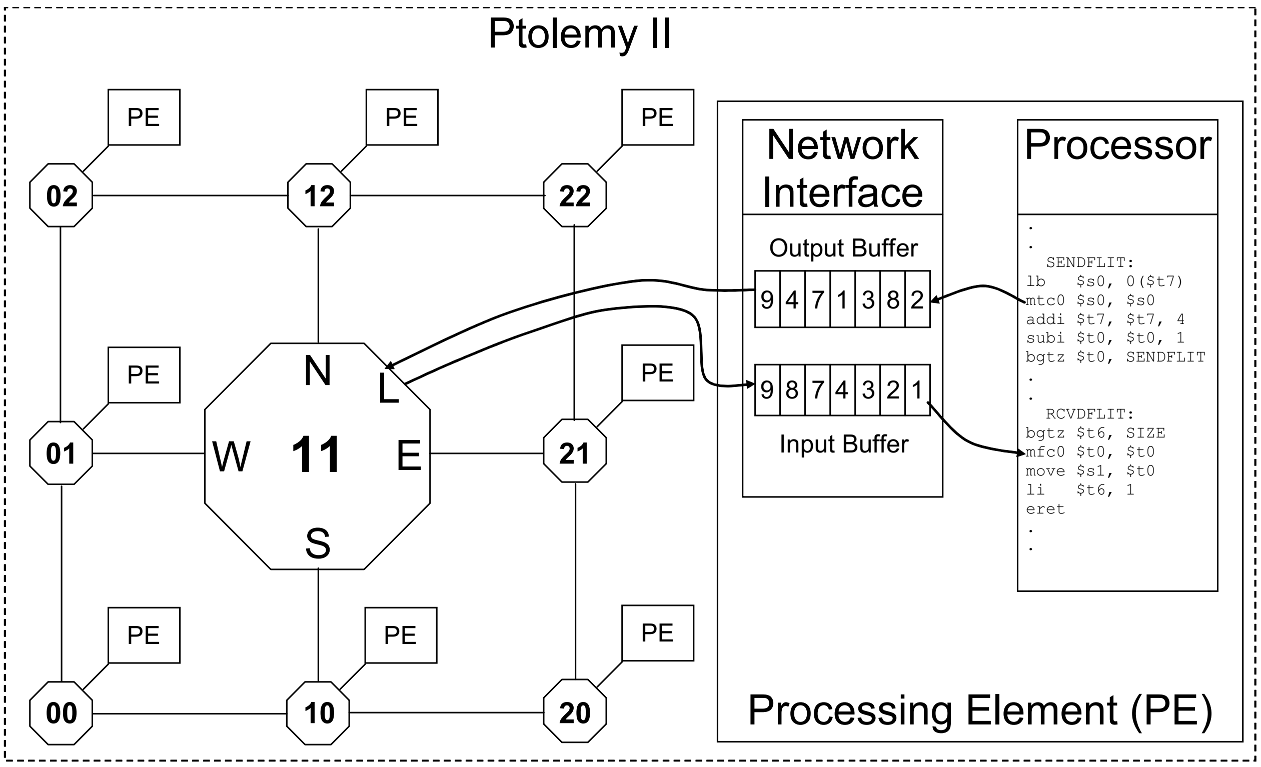
3.2. MARS ISS

MARS [3] is a MIPS Instruction Set Simulator (ISS). This means that MARS simulates the execution of programs written in the MIPS assembly language. MARS can be executed by command line or Graphical User Interface. MARS was developed by Peter Sanderson and Kenneth Vollmar, from the Missouri State University, and is written entirely in Java and distributed in an executable Jar file. MARS can simulate 155 basic instructions from the MIPS-32 instruction set, as well as about 370 pseudo-instructions or instruction variations, 17 syscall functions for console and file I/O and 21 syscalls for other uses.

3.3. RENATO NoC

RENATO NoC [4] was developed using the Ptolemy II framework and its behavior and timing constraints are based on the HERMES NoC. The basic element of the NoC is a five bi-directional port router, which is connected to 4 other neighbor routers and to a local IP core, following a MESH topology. The router employs a XY routing algorithm, round-robin arbitration algorithm and input buffers at each input port.

The RENATO NoC model can be connected to a debugging tool called NoCScope [11]. NoCScope provides improved observability of RENATO routers and overall resources in use. Seven scopes are currently available, allowing the user to see information about hot spots, power consumption, buffer occupation, input traffic, output traffic, end-to-end and point-to-point communications.



```
.
.
  SENDFLIT:
lb   $s0, 0($t7)
mtc0 $s0, $s0
addi $t7, $t7, 4
subi $t0, $t0, 1
bgtz $t0, SENDFLIT
.
.
  RCVDFLIT:
bgtz $t6, SIZE
mfc0 $t0, $t0
move $s1, $t0
li   $t6, 1
eret
.
.
```

Figure 1 – Block diagram of the proposed multiprocessor ISS.

4. Communication among processors

This section presents how the MARS ISS was connected to the RENATO NoC to allow the creation of a multiprocessor ISS. Figure 1 shows a block diagram of the system that will be used in the next subsections to guide the explanation of each component.

4.1. Processor to NI

In the current version of this work, each processor executes the MIPS assembly code of one task of the application. Communication between tasks happens by exchanging packets. In order to send a packet to another task, the header of the packet and the packet data need to be first stored in the data memory of the processor. The header of the packet is composed by the address of the target router where the processor is connected and the number of data flits this packet contains. After that, the send packet subroutine is called.

The send packet subroutine first reads the size flit of the packet stored in the memory to a register and reads to another register the output buffer size available in the NI. If there is enough space available in the NI to store the packet, the subroutine proceeds sending the packet flit by flit to the NI. The process of "reading" a flit from the NI uses the instruction "move from coprocessor 0" (*mfc0*), while the process of "sending" a flit to the NI uses the instruction "move to coprocessor 0" (*mtc0*). Thus, from the point of view of the processor, coprocessor 0 is now the NI.

4.2. NI to NoC

With the packet stored in the NI output buffer, the NI sends the packet flit by flit to the input local port of the router where this NI is connected. This happens following the flow control protocol in use by the NoC and using the timing delays set on the NoC model being executed by Ptolemy.

4.3. NoC to NI

When packets are being received from the NoC into the NI, a different buffer (input buffer) is used, thus allowing parallel sending and receiving of packets. The receiving of packets also occur following the flow control in use by the NoC and using the timing delays set on the NoC model.

4.4. NI to processor

As soon as the flits of the packet arrive in the input buffer of the NI, the NI launches a specific interruption to the processor meaning that a new packet has arrived. The MARS ISS, which was executing its task, saves its context and receives the interruption in the form of a Java exception. The standard routine for handling exceptions is called. By the ID of the specific exception, the exact exception is found out to be the "new message from network exception". The specific subroutine of this exception is launched. This subroutine mainly reads the complete packet from the NI using the "move from coprocessor 0" (*mfc0*) instruction to read each flit of the packet. After the complete packet was read from the NI and stored in the processor's memory, the processor's context is restored and it can now continues with its execution possibly using the data that was received.

5. Synchronization

The straightforward solution in Java to connect more than one MARS ISS to the NoC is to create a new MARS instance object for every new MARS instantiated in the NoC. However, this alternative failed due to the fact that MARS has been programmed using several static classes, attributes and methods. All of its main resources, such as the memory and the register bank, are declared as static. Therefore, if one tries to run more than one instance of MARS concurrently inside a single Java Virtual Machine (JVM), all the running instances will share the same resources, which will lead to unexpected behavior.

One possible workaround for this problem is to run each MARS instance in a different JVM. Java does not directly share memory between multiple VMs, so by running each MARS in a different JVM, one is safely isolating each instance of MARS. One problem with this approach is that the exchange of messages between different JVMs is only possible by using APIs such as Java Remote Method Invocation (RMI) and sockets, which would greatly increase the complexity of the system.

Another solution would be to reprogram MARS to remove the problematic static attributes and make them unique for each instance. However, this solution was also not optimal, considering the large number of static members declared in MARS and that every new future version of MARS would also require these modifications.

A better solution is to instantiate isolated ClassLoaders, one for each instance of MARS to be loaded. This works because a static element in Java is unique only in the context of a ClassLoader, therefore the static elements will not interfere with the other instances of MARS called by other ClassLoaders. By using this approach, the task of exchanging messages between the MARS instance and its corresponding NI also becomes trivial, and can be done simply by injecting a NI object when instantiating MARS.

A side effect of this solution is that each MARS instance and the NoC are considered as different threads by Java, and this would require extra algorithms based on *wait* and *notify* directives to maintain the time constraints followed by the NoC. As the main goal of this work is not provide good latency figures to the multiprocessor system application under simulation, we proceeded without the extra algorithms, aiming a faster simulation. Figure 2 presents a printout of the most important events occurred during the transfer of a packet composed by 2 header flits and 10 payload flits from MARS #1 to MARS #2. MARS #1 is connected to router 00 as illustrated in Figure 1 and MARS #2 is connected to router 21. No extra traffic is currently occupying the NoC.

```
3002 MARS #1 sending target  flit    (21) to NI #1
3002 MARS #1 sending size    flit    (10) to NI #1
3002 MARS #1 sending payload flit #0 (9) to NI #1
3003 MARS #1 sending payload flit #1 (9) to NI #1
3003 MARS #1 sending payload flit #2 (4) to NI #1
3003 MARS #1 sending payload flit #3 (7) to NI #1
3003 MARS #1 sending payload flit #4 (1) to NI #1
3003 MARS #1 sending payload flit #5 (3) to NI #1
3003 MARS #1 sending payload flit #6 (8) to NI #1
3003 MARS #1 sending payload flit #7 (2) to NI #1
3003 MARS #1 sending payload flit #8 (6) to NI #1
3086 MARS #1 sending payload flit #9 (5) to NI #1
3087 NI #1   sending target  flit    (21) to NoC
3089 NI #1   sending size    flit    (10) to NoC
3091 NI #1   sending payload flit #0 (9) to NoC
3093 NI #1   sending payload flit #1 (9) to NoC
3095 NI #1   sending payload flit #2 (4) to NoC
3097 NI #1   sending payload flit #3 (7) to NoC
3099 NI #1   sending payload flit #4 (1) to NoC
3101 NI #1   sending payload flit #5 (3) to NoC
3103 NI #1   sending payload flit #6 (8) to NoC
3105 NI #1   sending payload flit #7 (2) to NoC
3107 NI #1   sending payload flit #8 (6) to NoC
3109 NI #1   sending payload flit #9 (5) to NoC
3112 NoC     sending target  flit    (21) to NI #2
3116 NoC     sending size    flit    (10) to NI #2
3120 NoC     sending payload flit #0 (9) to NI #2
3120 NI #2   sending payload flit #0 (9) to MARS #2
3124 Noc     sending payload flit #1 (9) to NI #2
3128 Noc     sending payload flit #2 (4) to NI #2
3132 Noc     sending payload flit #3 (7) to NI #2
3136 Noc     sending payload flit #4 (1) to NI #2
3140 Noc     sending payload flit #5 (3) to NI #2
3144 Noc     sending payload flit #6 (8) to NI #2
3148 Noc     sending payload flit #7 (2) to NI #2
3152 Noc     sending payload flit #8 (6) to NI #2
3156 Noc     sending payload flit #9 (5) to NI #2
3166 NI #2   sending payload flit #1 (9) to MARS #2
3170 NI #2   sending payload flit #2 (4) to MARS #2
3172 NI #2   sending payload flit #3 (7) to MARS #2
3174 NI #2   sending payload flit #4 (1) to MARS #2
3175 NI #2   sending payload flit #5 (3) to MARS #2
3177 NI #2   sending payload flit #6 (8) to MARS #2
3178 NI #2   sending payload flit #7 (2) to MARS #2
3180 NI #2   sending payload flit #8 (6) to MARS #2
3181 NI #2   sending payload flit #9 (5) to MARS #2
```

Figure 2 – Timing delays of the most important events during the transfer of a packet between two processors.

All the following comments presented in this paragraph refer to Figure 2. Between times 3002 and 3086 MARS #1 sends the packet to the NI connected to it (NI #1), exactly as explained in Section 4.1. Eleven of the twelve flits of the packet were sent in the first 2 simulation cycles, and the last flit of the packet at time 3086. This strange behavior implies the following results: (1) MARS #1 thread was executed two times concurrently to Ptolemy thread, between times 3002-3003 and 3086; (2) MARS thread can be faster enough to execute at least 11 *mtc0* instructions in a row during 2 simulation cycles of Ptolemy; (3) MARS thread was not called again during 83 simulation cycles (3086-3003). Between times 3087 and 3109 each flit of the packet was sent constantly every 2 simulation cycles from NI #1 to the NoC, exactly as explained in Section 4.2. This behavior is equal to the real HERMES NoC that needs 2 clock cycles to transfer a flit using handshake flow control. Between time 3112 and 3156 all the flits from the packet were delivered from the NoC to NI #2 as explained in Section 4.3. However, due to some technical difficulties in the current version, it was not possible to deliver each flit every 2 simulation cycles, but 4 simulation cycles in this case. At time 3120 it is possible to see that NI #2 delivered the first payload flit immediately to MARS #2. Between times 3166 and 3181 the rest of the payload flits were delivered to MARS #2 as described in Section 4.4. Here again it is possible to see that the data transfer did not follow a constant pattern, similar to one the occurred between times 3002 and 3086. This unpredictable behavior is a side effect of running multiple threads with no proper synchronization.

6. Conclusion and Future Work

This work presented an ISS for multiprocessor systems based on the MIPS processor. In this work the RENATO NoC model was connected to two instances of the MARS ISS and as result applications based on more then one processor can be easily debugged with the presented approach. The most important contribution of this work is the NI, which allows both systems to communicate, thus creating a more realistic multiprocessing system model composed by computation and communication.

Initial figures regarding latency between processors' communication through the NoC were measured and we report to be insufficient in the current version. In order to have a good latency figure we must: (1) back annotate the timing delays of each assembly instruction from a real MIPS processor to MARS; (2) add extra synchronization logic to mimic the timing delays between processor and NI. In the current version of this work we guarantee only the NoC timing delays as presented in [4]. Future works will be related to steps 1 and 2.

References

[1] Intel Corporation. Intel Pentium D (Smithfield) Processor. Available at: http://ark.intel.com/ProductCollection.aspx?codeName=5788.

[2] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. "HeMPS - A Framework for NoC-Based MPSoC Generation". In: ISCAS'09, 2009, pp. 1345-1348.

[3] Vollmar, D. and Sanderson, D. "A MIPS assembly language simulator designed for education". Journal of Computing Sciences in Colleges, vol. 21(1), Oct. 2005, pp. 95-101.

[4] Indrusiak, L.S.; Ost, L.; Möller, L.; Moraes, F.; Glesner, M. Applying UML Interactions and Actor-Oriented Simulation to the Design Space Exploration of Network-on-Chip Interconnects. In: ISVLSI'08, 2008, pp. 491-494.

[5] Eker, J.; Janneck, J.; Lee, E.; Liu, J.; Liu, X.; Ludvig, J.; Neuendorffer, S.; Sachs, S.; Xiong, Y. "Taming Heterogeneity - The Ptolemy Approach". Proceedings of the IEEE, vol. 91 (2), Jan. 2003, pp. 127-144.

[6] Benini, L.; Bertozzi, D.; Bogliolo, A.; Menichelli, F.; Olivieri, M. "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC". The Journal of VLSI Signal Processing, vol. 41 (2), Sep. 2005, pp. 169-182.

[7] Boukhechem, S.; Bourennane, E. "SystemC Transaction-Level Modeling of an MPSoC Platform Based on an Open Source ISS by Using Interprocess Communication". International Journal of Reconfigurable Computing, vol. 2008, Article ID 902653, 2008, 10 p.

[8] Ceng, J.; Sheng, W.; Castrillon, J.; Stulova, A.; Leupers, R.; Ascheid, G.; Meyr, H. "A high-level virtual platform for early MPSoC software development". In: CODES+ISSS'09, 2009, pp. 11-20.

[9] Pouillon, N.; Becoulet, A.; Mello, A.; Pecheux, F.; Greiner, A. "A Generic Instruction Set Simulator API for Timed and Untimed Simulation and Debug of MP2-SoCs". In: RSP'09, 2009, pp. 116-122.

[10] Opencores. Available at: http://www.opencores.org.

[11] Möller, L.; Indrusiak, L.S.; Glesner, M. "NoCScope: A Graphical Interface to Improve Networks-on-Chip Monitoring and Design Space Exploration". In: IDT'09, 2009.

Impact of Task Distribution, Processor Configurations and Dynamic Clock Frequency Scaling on the Power Consumption of FPGA-based Multiprocessors

Diana Goehringer, Jonathan Obie
Fraunhofer IOSB
Ettlingen, Germany
{diana.goehringer, jonathan.obie}@iosb.fraunhofer.de

Michael Huebner, Juergen Becker
ITIV, Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{michael.huebner, becker}@kit.edu

Abstract— **As only the currently required functionality on a dynamic reconfigurable FPGA-based system is active, a good performance per power ratio can be achieved. To find such a good performance per power ratio for a given application is a difficult task, as it requires not only knowledge of the behavior of the application, but also knowledge of the underlying hardware architecture and its influences on the performance and the static and dynamic power consumption. Is it for example better to use two processors running at half the clock frequency than a single processor? The main contributions of this paper are: the description of a tool flow to measure the power consumption for multiprocessor systems in Xilinx FPGAs, a novel runtime adaptive architecture for analyzing the performance per power tradeoff and for dynamic clock frequency scaling based-on the inter-processor communication. Furthermore, we use three different application scenarios to show the influence of the clock frequency, different processor configurations and different application partitions onto the static and dynamic power consumption as well as onto the overall system performance.**

***Keywords-** Power Consumption, Multiprocessor System-on-Chip (MPSoC), Dynamic Frequency Scaling, Task Distribution, Application Partitioning, Dynamic and Partial Reconfiguration, FPGA.*

I. Introduction

Parameterizable function blocks used in FPGA-based system development, open a huge design space, which can only hardly be managed by the user. Examples for this are arithmetic blocks like divider, adder, soft IP-multiplier, which are adjustable in terms of bitwidth and parallelism. Additional to arithmetic blocks, also soft-IP processor cores provide a variety of parameters, which can be adapted to the requirements of the application to be realized with the system. Especially, Xilinx offers with the MicroBlaze Soft-IP RISC processor [1] a variety of options for characterizing the core individually. These options are amongst others the use of cache memory and its size, the use of an arithmetic unit, a memory management unit and the number of pipeline stages. Furthermore, the tools offer to deploy up to two processor cores as multiprocessor on one FPGA. Every option now can be adjusted to find an optimal parameterization of the processor core in relation to the target application. For example, a specific cache size can speed up the application tremendously, but also the optimal partition of functions onto the two cores has a strong impact on the speed and power consumption of the system. The examples show the huge design space, if only one parameter is used. It is obvious, that the usage of multiple parameters for system adjustment leads to a multidimensional optimization problem, which is not or at least very hardly manageable by the designer. In order to gain experience regarding the impact of processor parameterization in relation to specific application scenario, it is beneficial to evaluate e.g. the performance and power-consumption of an FPGA-based system and normalize the results to a standard design with a default set of parameter. The result of such an investigation is a first step for developing standard guidelines for designers and an approach for an abstraction of the design space in FPGA-based system design. This paper presents first results of a parameterizable multiprocessor system on a Xilinx Virtex-4 FPGA, where the parameterization of the processor is evaluated in terms of power consumption and performance. Moreover, the varying partition of the different application scenarios is evaluated in terms of power consumption for a fixed performance. For this purpose, a tool flow for analyzing the power consumption through generating the value change dump (VCD) file from the post place and route simulation will be introduced. The presented flow enables to generate the most accurate power consumption estimation from this level of abstraction. A further output of the presented work is an overview of the impact of parameterization to the power consumption. The results can be used as a basic guideline for designers, who want to optimize their system performance and power consumption.

The paper is organized in the following manner: In Section II related work is presented. Section III describes the power estimation tool flow used in this approach. The novel system architecture used for analyzing the performance and the power consumption of the different applications is presented in Section IV. The application scenarios are described in Section V. In Section VI the application integration and the results for performance and power consumption are given. Finally, the paper is closed by presenting the conclusions and future work in Section VII.

II. Related Work

Optimization of the dynamic and static power consumption is very important, especially for embedded systems, because they often use batteries as a power source.

Therefore, many researchers like for example Meintanis et al [2] explored the power consumption of Xilinx Virtex-II Pro, Xilinx Spartan-3 and Altera Cyclone-II FPGAs. They estimated the power consumption at design-time using the commercial tools provided by Xilinx and Altera. They further explored the differences between the measured and estimated power consumption for these FPGAs. Becker et al. [3] explored the difference between measured and estimated power consumption for the Xilinx Virtex-2000E FPGA. Furthermore, they explored the behavior of the power consumption, when using dynamic reconfiguration to exchange the FPGA-system at runtime.

Other works focus on the development of own tools and models for efficient power estimation at design-time for FPGA-based systems. Poon et al. [4] present a power model to estimate the dynamic, short circuit and leakage power of island-style FPGA architectures. This power model has been integrated into the VPR CAD flow. It uses the transition density signal model [5] to determine signal activities within the FPGA. Weiss et al. [6] present an approach for design-time power estimation for the Xilinx Virtex FPGA. This estimation method works well for control flow oriented applications but not so well for combinatorial logic. Degalahal et al. [7] present a methodology to estimate dynamic power consumption for FPGA-based system. They applied this methodology to explore the power consumption of the Xilinx Spartan-3 device and to compare the estimated results with the measured power consumption.

All these approaches focus either on the proposal of a new estimation model or tool for estimating the power consumption at design-time or they compare their own or commercial estimation models and tools with the real measured power consumption. The focus of the investigations presented in this paper is to show the impact of parameterization of IP-cores, here specifically the MicroBlaze soft processor, which differs from the approaches mentioned above where the topic is more on tool development for power estimation.

The novelty of our approach is to focus on the requirements of the target application and to propose a design guideline for system developers of processor-based FPGA systems. This means, providing guidance in how to design a system to achieve a good tradeoff between performance and power consumption for a target application. To develop such a guideline the impact of the frequency, different processor configurations and the task distribution in a processor-based design is investigated in this paper for different application scenarios. To the best of our knowledge, similar work has not done before.

III. Tool Flow for Power Measurement

Xilinx provides two kinds of tools for power consumption estimation: Xilinx Power Estimator (XPE) [8] and Xilinx Power Analyzer (XPower) [9].

The XPE tool is based on an excel spreadsheet. It receives information about the number and types of used resources via the report generated by the mapping process (MAP) of the Xilinx tool flow. Alternatively, the user can manually set the values for the number and type of used resources. The frequencies used within the design have to be manually set by the user. The advantage of this method is that results are obtained very fast. The disadvantage is that the results are not very accurate, especially for the dynamic power consumption. This is, because the different toggling rates of the signals are not taking into account. Also, the results are not as accurate, because they are based on the MAP report, and not on the post place and route (PAR) report, which resembles the system used for generating the bitstream.

The XPower tool estimates the dynamic and static power consumption for submodules, different subcategories and the whole system based on the results of a post place and route (PAR) simulation. This makes the estimation results much more accurate compared to the XPE tool, because the final placed and routed system is considered for the power estimation. But even more important, due to the simulation of the PAR system with real input data, the toggling rates of the signals can be extracted and used within the power estimation. For estimating the power consumption with the XPower tool the following input files are required:

- Native Circuit Description (NCD) file, which specifies the design resources
- Physical Constraint File (PCF), which specifies the design constraints
- Value Change Dump (VCD) file, which specifies the simulated activity rates of the signals

The NCD and the PCF file are obtained after the PAR phase of the Xilinx implementation tool flow. The VCD file is generated by doing a simulation of the PAR design with the ModelSim simulator.

Due to the higher accuracy the XPower tool was used here. As we wanted to estimate the power consumption for systems with one or two MicroBlaze processors, the hardware and the software executables of the different system were designed within the Xilinx Platform Studio (XPS)[10]. Figure 1. shows the flow diagram for doing power estimation with XPower for an XPS system.

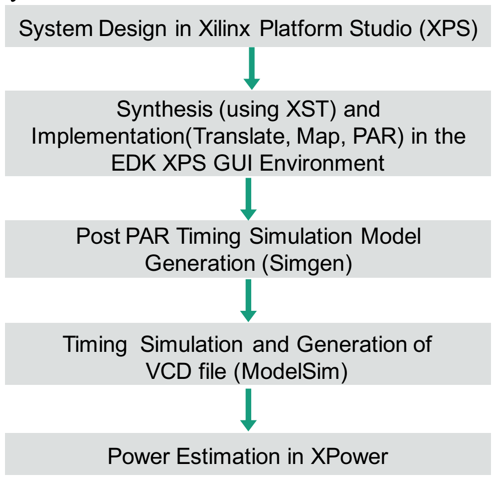

System Design in Xilinx Platform Studio (XPS)
Synthesis (using XST) and Implementation(Translate, Map, PAR) in the EDK XPS GUI Environment
Post PAR Timing Simulation Model Generation (Simgen)
Timing Simulation and Generation of VCD file (ModelSim)
Power Estimation in XPower

Figure 1. Flow Diagram of the EDK XPower Flow

After the system has been designed and implemented within the XPS environment, the Simgen [10] tool is used to generate the post PAR timing simulation model of the system. This simulation model is the used to simulate the behavior of the system with the ModelSim simulator and to generate the VCD file. In the last step XPower is used to read in the VCD, the NCD and the PCF files of the design and to estimate the dynamic and static power consumption.Care has to be taken, because in a normal Xilinx implementation flow the software executables are integrated into the memories of the processors after the bitstream has been generated. When using XPower and the post PAR simulation, the memories of the processor have to be initialized in an earlier step. This means, into the post PAR simulation model, otherwise the simulated system behavior and the VCD file would not be accurate.

IV. Novel System Architecture

The system structure of the dual-processor system is shown in Figure 2. Three new components have been designed and implemented: the Virtual-IO, the Bridge and the Reconfigurable Clock Unit. All three components have been integrated into a library for the XPS tool. Therefore, they can be inserted and parameterized using the graphical user interface (GUI) of the XPS tool, which makes them easy reusable within other XPS designs.

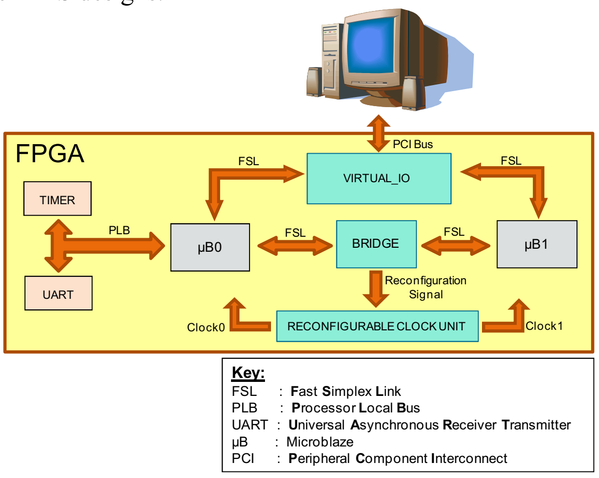

FPGA
PCI Bus
FSL
VIRTUAL_IO
FSL
TIMER
PLB
µB0
FSL
BRIDGE
FSL
µB1
UART
Reconfiguration Signal
Clock0
RECONFIGURABLE CLOCK UNIT
Clock1
Key:
FSL : Fast Simplex Link
PLB : Processor Local Bus
UART : Universal Asynchronous Receiver Transmitter
µB : Microblaze
PCI : Peripheral Component Interconnect

Figure 2. Dual processor design with three new components: Virtual-I/O, Bridge and Reconfigurable Clock Unit

The Virtual-IO receives data from the host PC and sends results back to the host PC via the PCI-bus. The Virtual-IO communicates via the Fast Simplex Links (FSLs) [11] with two MicroBlaze processors (µB0 and µB1). µB0 communicates with the user via the UART interface. It also has a timer, which is used to measure the performance of the overall system. The two processors communicate with each other via FSLs over the Bridge component. Depending on the fill level of the FIFOs within the Bridge reconfiguration signals are send to the Reconfigurable Clock Unit. The Reconfigurable Clock Unit reconfigures the clocks of the two processors based on the reconfiguration signals issued by the Bridge. For the uni-processor system, which is used for comparison, the Bridge, the Reconfigurable Clock Unit, µB1 and their connections were removed as shown in Figure 3.

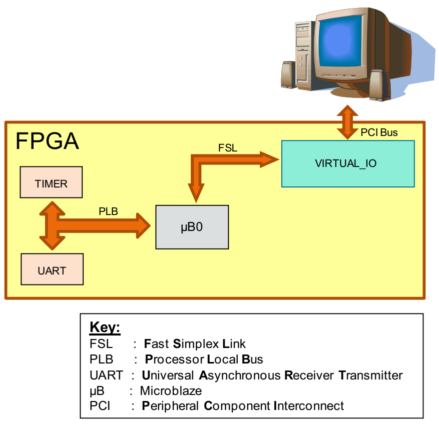

FPGA
PCI Bus
FSL
VIRTUAL_IO
TIMER
PLB
µB0
UART
Key:
FSL : Fast Simplex Link
PLB : Processor Local Bus
UART : Universal Asynchronous Receiver Transmitter
µB : Microblaze
PCI : Peripheral Component Interconnect

Figure 3. Uni-processor system

The following subsections explain the new components and their features more in detail.

A. *Virtual-IO*

The Virtual-IO component is used to communicate with the host PC via the PCI-bus. It provides an input and an output port to the PCI-bus and one input and one output port for each MicroBlaze processor. It consists of two FIFOs, one for the incoming and one for the outgoing data of the PCI-bus. Each FIFO is controlled via a Finite State Machine (FSM), as it is shown in Figure 4.

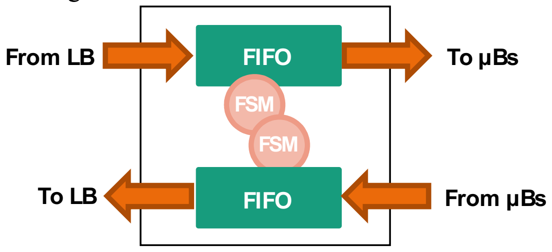


Figure 4. Virtual-IO component

The Virtual-IO is a wrapper around 6 different modules. The first module is Virtual-IO 1, which sends data first to µB0 and then to µB1. It then receives the calculated results in the same order. The second module is Virtual-IO 2, which sends data only to µB0. Results are only received over µB1. Therefore, µB0 sends its results to µB1, which then sends the results of µB0 together with its own results back to the Virtual-IO 2. The third module is Virtual-IO 3, which sends first data to µB0. Afterwards, it sends in parallel to both processors µB0 and µB1 the same data. Finally, it sends some data only to µB1. After the execution of the processors, first µB0 and then µB1 send their results back to the Virtual-IO 3. The fourth module is Virtual-IO 4, which is only connected to one of the processors, e.g. µB0. Due to this, this module is used in all uni-processor designs. For a dual-processor design it sends data to µB0, which then forwards parts of the data to µB1. After execution µB1 sends its results back to µB0, which forwards the results of the execution of the two processors to the Virtual-IO 4. The fifth module is Virtual-IO 5, which sends the same data to both processors in parallel, but receives the results only via µB0. The sixth module is Virtual-IO 6. It is very similar to

Virtual-IO 5. The only difference is that it receives the calculation results from µB1 instead of µB0.

The modules can be selected in the XPS GUI via the parameters of the Virtual-IO component. Other parameters that can be set by the user are: the number of input and output words for each processor separately, the number of common input words and the size of the image (only for image processing applications).

B. Bridge

The Bridge module is used for the inter-processor communication. It consists of two asynchronous FIFOs controlled by FSMs, to support a communication via the two different clock domains of the processors, as shown in Figure 5. This Bridge component controls the fill level of the two FIFOs. If one FIFO is to nearly full, it is assumed that the processor, which reads from this FIFO, is too slow. As a result, a reconfiguration signal to increase the clock rate of this processor is send to the Reconfiguration Clock Unit.

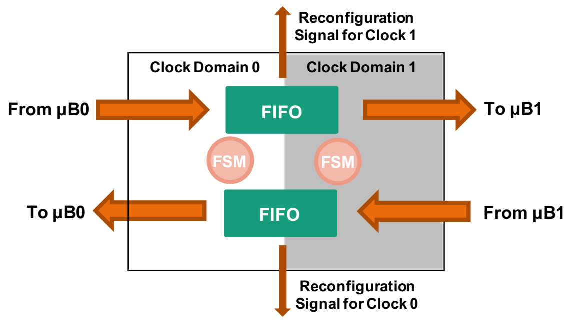


Figure 5. Internal structure of the Bridge

C. Reconfigurable Clock Unit

The internal structure of the Reconfigurable Clock Unit is shown in Figure 6. It consists of two Digital Clock Managers (DCMs) [12], two Clock Buffer Multiplexer primitives (BUFGMUXes) [13] and the Logic component, which controls the reconfiguration of the DCMs.

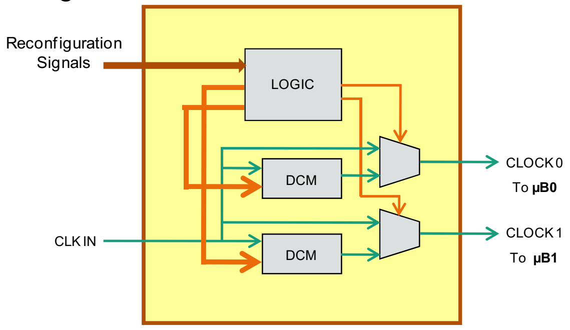


Figure 6. Internal structure of the Reconfigurable Clock Unit

The Logic component receives the reconfiguration signals of the Bridge component. It then starts the reconfiguration of the DCM primitive for the slower processor. For the reconfiguration the specific ports provided by Xilinx for dynamic reconfiguration of the Virtx-4 DCM primitive are used. During the reconfiguration process the DCM has to be kept in a reset state for a minimum of 200 ms. During this time interval the outputs of this DCM are not stable and cannot be used. Instead of stalling the corresponding processor, the BUFGMUX primitive is used to provide CLK_IN, the original input clock of the two DCM, to the processor, whose DCM is under reconfiguration. The BUFGMUX is a special clock multiplexer primitive, which assures, that no glitches occur, when switching to a different clock. After the configuration of the DCM is finished, the BUFGMUX is used to switch back to the DCM clock. An alternative would be to stall the processor, while its clock is being reconfigured. Because 200 ms are quite a long time, especially for image processing applications, where each 40 ms a new input frame is received from a camera; this would result in a loss of input data.

To prevent an oscillation, the controller logic will stop increasing the clock frequency, if 125 MHz for this MicroBlaze have been reached, which is the maximum frequency supported by the MicroBlaze and its peripherals, or if its clock frequency has been increased for three consecutive times. If the reconfiguration signal is still asserted meaning the processor is still too slow, then the DCM of the faster processor is reconfigured to provide a slower clock to the faster processor.

Alternatively, instead of dynamically reconfiguring the DCM, different output ports of a DCM could be used to generate different clocks. Using several BUFGMUXes the different clocks could be selected. The advantage is a faster switch between different clocks and the drawback is that not as many different clocks are possible as when dynamic reconfiguration is used. This will be investigated in future work.

V. APPLICATION SCENARIOS

Three different applications scenarios were used to explore the impact of the processor configurations, the task distribution and the dynamic clock frequency scaling on the power consumption of FPGA-based processor systems. The three different algorithms are described in detail in the next subsections. The first algorithm is the well known sorting algorithm called Quicksort [14]. It consists of a lot of branches and comparisons. The second algorithm is an image processing algorithm called Normalized Squared Correlation (NCC), which consists of many arithmetic operations, e.g. multiply and divide. The third algorithm is a variation of a bioinformatic algorithm called DIALIGN [15], which consists of many comparisons and additions and subtractions. These algorithms with their different algorithm requirements, e.g. branches, comparators, multiply & divide, add & subtract, were used to provide a user guideline of designing a system with a good performance per power tradeoff for a specific application. By comparing the algorithm requirements of new applications with the three example algorithms, the system configurations of the most similar example algorithm is chosen as a starting system. Such a guideline to limit the design space is very important to save time and achieve a higher time-to-market, because the simulation and the power estimation with XPower are very time-consuming. Also, the bitstream generation to measure the performance of the application on the target hardware architecture is time-consuming. These long design times can be shorten by starting with an appropriate design, e.g. the right processor configurations, a good task distribution and a well selected execution frequency.

A. *Sorting Algorithm: Quicksort*

Quicksort [14] is a well known sorting algorithm with a divide and conquer strategy. It sorts a list by recursively partitioning the list around a pivot and sorting the resulting sublists. It has an average complexity of Θ (n log n).

B. *Image Processing Algorithm: Normalized Squared Correlation*

2D Squared Normalized Correlation (NCC) is often used to identify an object within an image. The evaluated expression is shown in equation (1).

$$C(p) = \frac{\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(\left(\mathrm{A}_{\mathrm{p}}(i,j)-\overline{\mathrm{A}_{\mathrm{p}}}\right)*\left(\mathrm{T}(i,j)-\overline{\mathrm{T}}\right)\right)\right)^2}{\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(\mathrm{A}_{\mathrm{p}}(i,j)-\overline{\mathrm{A}_{\mathrm{p}}}\right)^2\right)*\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(\mathrm{T}(i,j)-\overline{\mathrm{T}}\right)^2\right)} \quad (1)$$

T : Template image with n Rows and m Columns

A_p : Sub window of the search region wit h n Rows and m Columns

$\overline{T}$: Mean of T

$\overline{A_p}$: Mean of A_p

This algorithm uses a template T of the object to be searched for and moves this template over the search region A of the image. A_p, the subwindow of the search region at point p with the same size as T, is then correlated with T. The result of this expression is stored at point p in the result image C. The more similar A_p and T are the higher is the result of the correlation. If they are equal, the result is 1. The object is then detected at the location with the highest value.

C. *Bioinformatic Algorithm: DIALIGN*

DIALIGN [15] is a bioinformatics algorithm, which is used for comparison of the alignment of two genomic sequences. It produces the alignment with the highest number of similar elements and therefore the highest score as shown in Figure 7.

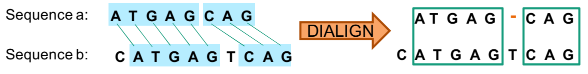


Figure 7. Alignment of two sequences a and b with DIALIGN.

VI. Integration and Results

For the power consumption estimation and the performance measurement a Xilinx Virtex-4 FX 100 FPGA was used. The performance was measured on the corresponding FPGA Board from Alpha Data [16]. As measuring the exact power consumption of the FPGA on this board is not possible, it was estimated at design-time using the XPower tool flow as described in Section III. The impact of the clock frequency, the configuration of the processor and the task distribution onto the power consumption and the performance of the system has been explored and the results are presented in the following subsections. For each exploration some parameters had to be kept fixed to assure a fair comparison. For the exploration of the impact of the clock frequency, the algorithm and the processor configuration have been kept fixed. For the exploration of the impact of the configuration of the processor the clock frequency were kept fixed. Finally, for the exploration of the task distribution, the processor configuration and the performance were kept fixed to lower the overall system power consumption while maintaining the performance similar to the performance achieved with a reference uni-processor design running at 100 MHz, which is a standard frequency for Virtex-4 based MicroBlaze systems.

A. *Impact of the clock frequency*

First of all the impact of the variation of the clock frequency onto the power consumption was explored for a uni-processor system, which executes the NCC algorithm on one MicroBlaze. The MicroBlaze was configured to use a 5-stage pipeline and no arithmetic unit. The results for the dynamic and quiescent power consumption for the core and the other components as well as the total power consumption of the system are given in TABLE I. The quiescent power consumption is also called static power consumption in the following, because it represents the power consumption of the user configured FPGA without any switching activity.

The impact of the clock frequency onto the static - and the dynamic power consumption is presented in Figure 8. and Figure 9. respectively. As can be seen the static power consumption increases by around 0,24 mW / MHz, while the dynamic power consumption increases by around 3,26 mW / MHz.

Out of this results the impact onto the total power consumption, which is around 3,5 mW / MHz. The impact on the total power consumption as well as on the performance is shown in Figure 10.

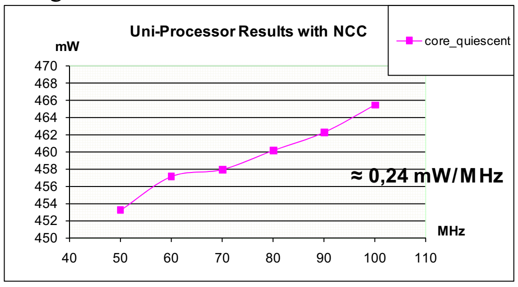


Figure 8. Impact of the clock frequency onto the static power consumption of a uni-processor design.

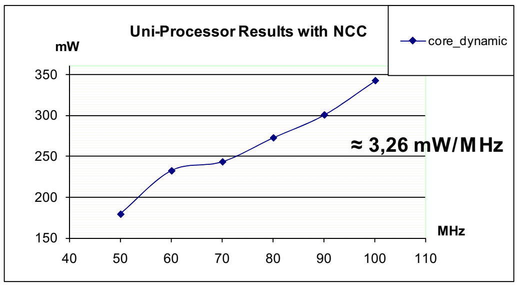


Figure 9. Impact of the clock frequency onto the dynamic power consumption of a uni-processor design.

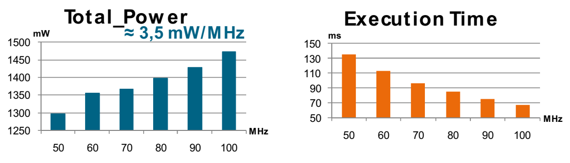


Figure 10. Impact of the clock frequency onto the total power consumption and onto the execution time of a uni-processor design executing the NCC algorithm.

TABLE I. IMPACT OF THE VARIATION OF THE CLOCK FREQUENCY ONTO THE POWER CONSUMPTION

Clk Freq. (MHz)	$P_{CoreDynamic}$(mW)	$P_{OthersDynamic}$(mW)	$P_{CoreQuiescent}$(mW)	$P_{OthersQuiescent}$(mW)	P_{Total}(mW)	P_{Total}(%)
50	180	26	453	641	1298	- 11,9
60	232	26	457	641	1355	- 8,0
70	243	26	458	641	1367	- 7,2
80	273	26	460	641	1398	- 5,1
90	301	26	462	641	1428	- 3,1
100	343	26	465	641	1473	NA

B. *Impact of the processor configurations*

For exploration, a uni-processor design consisting of a single MicroBlaze running at 100 MHz was used. The results were compared against a reference configuration, which was a MicroBlaze with a 5-stage pipeline and no arithmetic unit (integer divider and barrel shifter). The following configurations were explored:

i. adding an arithmetic unit (AU)
ii. reduction of the pipeline to 3-stages (RP)
iii. combination of i and iii (AU+RP)

The impact onto the power consumption and the performance was explored for all three algorithms. The impact is very different for the different applications, due to the different algorithm requirements, as mentioned in Section V and its subsections.

Figure 11. and TABLE II. show the impact of the different configurations for the Quicksort algorithm. Due to the multiple branches in the algorithm a reduction of the pipeline stages is very beneficial in terms of execution time and power consumption. The impact of the addition of the arithmetic unit only provides a minimal improvement in terms of performance, but with a stronger degradation of the power consumption. Depending on the performance and power consumption constraints, either the system with the AU + RP or the RP system would be chosen.

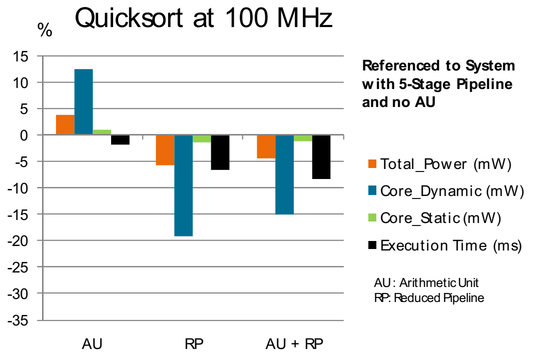


Figure 11. Impact of the MicroBlaze configurations for the Quicksort algorithm.

Figure 12. and TABLE III. show the impact of the different configurations for the NCC algorithm. As this algorithm requires many arithmetic operations, the addition of an AU improves the overall execution time, while the reduction of the pipeline stages results in a strong degradation (over 50%). This degradation is due to the reason that the execution of arithmetic operations take more clock cycles, if the pipeline is reduced. Therefore, for this and similar algorithms a system with an AU and a 5-stage pipeline would be optimal from a performance perspective. If the power consumption needs to be reduced and some performance degradation is acceptable, than the reference system or the AU+RP system would be a good choice.

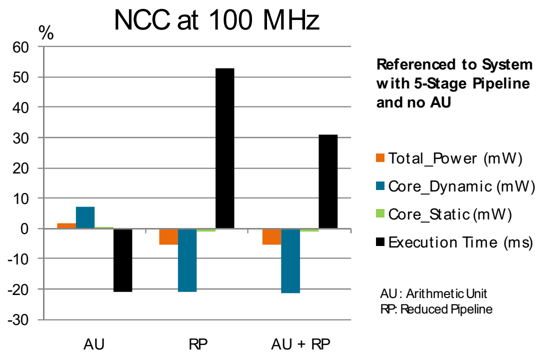


Figure 12. Impact of the MicroBlaze configurations for the NCC algorithm.

In Figure 13. and TABLE IV. the impact onto the performance and power consumption of the three different processor configurations compared to the reference system are presented for the DIALIGN algorithm. Adding an AU improves the execution time only a little bit, while increasing the overall power consumption compared to the reference design. The reduction of the pipeline to 3-stages improves the total power consumption by 6,8%, but worsening the execution time by 25%. The combination of AU+RP shows nearly the same impact as the RP system. Therefore, the reference system is the best choice, if performance is the most important factor. If on the other hand the power consumption is more important, than the RP system would be a good choice for these kinds of algorithms.

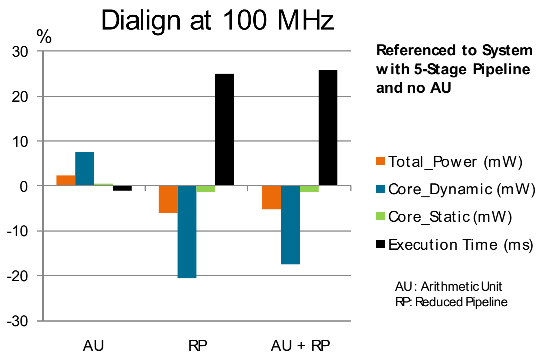


Figure 13. Impact of the MicroBlaze configurations for the DIALIGN algorithm

TABLE II. IMPACT OF THE MICROBLAZE CONFIGURATIONS FOR THE QUICKSORT ALGORITHM AT 100 MHZ

µB Parameter	$P_{CoreDynamic}$(mW)	$P_{OthersDynamic}$(mW)	$P_{CoreStatic}$(mW)	$P_{OthersStatic}$(mW)	P_{Total}(mW)	P_{Total}(%)	Time (ms)
Default	438,33	26,13	472,91	639,56	1576,93	NA	18,42
Arithmetic Unit	493,26	26,14	477,20	639,58	1636,18	+ 3,76	18,10
3-stage Pipeline	354,23	26,13	466,41	639,56	1486,33	- 5,75	17,21
Both	372,84	26,13	467,84	639,58	1506,39	- 4,47	16,89

TABLE III. IMPACT OF THE MICROBLAZE CONFIGURATIONS FOR THE NCC ALGORITHM AT 100 MHZ

µB Parameter	$P_{CoreDynamic}$(mW)	$P_{OthersDynamic}$(mW)	$P_{CoreStatic}$(mW)	$P_{OthersStatic}$(mW)	P_{Total}(mW)	P_{Total}(%)	Time (ms)
Default	341,68	26,09	465,44	639,57	1472,78	NA	67,74
Arithmetic Unit	366,28	26,13	467,33	639,57	1499,31	+ 1,80	53,62
3-stage Pipeline	269,63	26,10	459,97	639,57	1395,27	- 5,26	103,64
Both	269,40	26,12	459,95	639,58	1395,05	- 5,28	88,84

TABLE IV. IMPACT OF THE MICROBLAZE CONFIGURATIONS FOR THE DIALIGN ALGORITHM AT 100 MHZ

µB Parameter	$P_{CoreDynamic}$(mW)	$P_{OthersDynamic}$(mW)	$P_{CoreStatic}$(mW)	$P_{OthersStatic}$(mW)	P_{Total}(mW)	P_{Total}(%)	Time (µs)
Default	431,67	26,06	472,38	639,58	1569,69	NA	786,48
Arithmetic Unit	464,39	26,06	474,93	639,59	1604,97	+ 2,25	777,64
3-stage Pipeline	343,01	26,05	465,54	639,59	1474,19	- 6,08	982,85
Both	355,88	26,06	466,53	639,58	1488,05	- 5,20	988,05

TABLE V. QUICKSORT POWER CONSUMPTION

	Uni-Processor (100MHz)	Dual_2 (80/50 MHz)	Dual_5 (95 MHz)
Execution Time - ms	18,42	18,80	19,27
Core (dyn/stat)_Power - mW	438,33 / 472,91	295,89 / 461,95	384,34 / 468,72
Total Power - mW	1576,93	1475,56	1570,79
Total Power - %	NA	- 6,43	- 0,39

TABLE VI. NCC POWER CONSUMPTION

	Uni-Processor (100MHz)	Dual_3 (54 MHz)	Dual_2 (87,5/50 MHz)
Execution Time - ms	67,74	67,28	67,62
Core (dyn/stat)_Power - mW	341,68 / 465,44	297,39 / 462,07	322,32 / 463,96
Total Power - mW	1472,78	1477,20	1504,02
Total Power - %	NA	+ 0,30	+ 2,12

TABLE VII. DIALIGN POWER CONSUMPTION

	Uni-Processor (100MHz)	Dual_5 (50 MHz)	Dual_6 (50 MHz)
Execution Time - ms	30,21	30,16	30,16
Core (dyn/stat)_Power - mW	431,67 / 472,38	440,80 / 473,09	352,45 / 466,27
Total Power - mW	1569,69	1631,62	1536,44
Total Power - %	NA	+ 3,95	- 2,12

C. Impact of the task distribution and the frequency scaling

To measure the impact onto the power consumption the algorithms were partitioned onto two MicroBlaze processors. The frequency for the two processors was chosen in such a way, that the execution time of the dual-processor design was as similar as possible to the reference system consisting of a single MicroBlaze running at 100 MHz. For all systems the configurations of the processors were fixed to a 5-stage pipeline and no arithmetic unit.

TABLE V. shows the results for distributing the Quicksort algorithm on two processors instead of one. Two partitions were done. The first one is called Dual_2 (80/50 MHz), which means, that the Virtual-IO 2 was used and µB0 was running at 80 MHz while µB1 was running at 50 MHz. The algorithm was so partitioned that µB0 receives the whole data to be sorted. It then divides the data into two parts and sends the second part to µB1. Both then sort their partition. µB0 forwards its sorted part of the list to µB1, which sends the final combined sorted list via the Virtual-IO 2 to the host PC. With this partition the overall power consumption could be reduced by 6,43% compared to the single processor reference system.

The second partition called Dual_5 (95 MHz) uses the Virtual-IO 5 to send incoming data to both processors running at 95 MHz. µB0 searches the list for elements smaller and µB1 searches the list for elements bigger than the pivot. When one has found an element the position of this element is send to the other processor. Both processor then update their lists by swapping the own found element with the one the other processor has found. At the end both processors have as a result a searched list. µB0 then sends its resulting list back to the host PC via the Virtual-IO 5. The power consumption of this version is nearly the same as the reference system, while the total execution time increases.

TABLE VI. shows the result for the partitioning of the NCC algorithm onto two processors. The first partitioning uses the Virtual-IO 3 to partition the incoming image into two overlapping tiles, one for each processor. The overlapping part is send to both processors simultaneously. As the NCC is a window-based image processing algorithm, the boarder pixels between the two tiles are needed by both processors. Each of the processors runs at 54 MHz, which results in a similar execution time, and also in a similar total power consumption as the reference design.

The second partition called Dual_2 (87,5 /50 MHz) uses Virtual-IO 2 to send the whole image to µB0. µB0 runs at 87, 5 MHz and calculates the complete numerator and the denominator. Then it forwards both to µB1, which does the division and sends the results back to the Virtual-IO 2. µB1 runs at 50 MHz. While the execution time is nearly the same, the overall power consumption is increased slightly by 2,12%.

TABLE VII. shows the result for executing the DIALIGN Algorithm with two processors. Two partitions were done. The first one is called Dual_5 (50 MHz) and uses Virtual-IO 5 to send the incoming sequences to both processors running at 50 MHz. Each processor calculates half of the resulting score matrix. µB0 calculates on a row-based fashion all values above the main diagonal. µB1 calculates on a column-based fashion all values below the main diagonal. The scores on the main diagonal are calculated by both processors. After µB0 has finished calculating one row and µB1 one column respectively, they exchange the first score nearest to the main diagonal, as this score is needed by both processors for calculating the next row/column respectively. While the execution time is nearly the same, the overall power consumption is increased by 3,95%.

The second partition is called Dual_6 (50 MHz). It uses the Virtual-IO 6 to send the sequences to the processors, which run both at 50 MHz. Here a systolic array approach is used for executing the DIALIGN algorithm. µB1 then sends the final alignment and the score back to the host PC. With this partition the overall power consumption could be reduced by 2,12% compared to the single processor reference system.

VII. Conclusions and Outlook

This paper reports the research and evaluation of different microprocessor parameterization, application and data partitioning on a dual-processor system. The results of the experiments show the impact of the different parameterization on the power consumption and performance in relation to a set of selected applications. Depending on the application type it can be seen that different parameter configurations, e.g. configuration of the processors and their frequencies, but also a good application partitioning, are essential for achieving an efficient tradeoff between performance and power constraints. The results can be used to guide developers what parameter set suits to a certain application scenario. The vision is that more application scenarios will be analyzed in order to provide a broad overview of the parameter impact. It is envisioned to extend existing hardware benchmarks from different application domains in terms of a parameterization guideline also for further FPGA series from Xilinx.

Furthermore, the paper provides a tutorial for the estimation of the power consumption on a high level of abstraction, but with a high accuracy through post place and route simulation. Therefore, other research in this area can be done and exchanged in the community.

Acknowledgment

The authors would like to thank Prof. Alba Cristina M. A. de Melo and Jan Mendonca Correa for providing us with their C code implementation of the DIALIGN algorithm.

References

[1] "Xilinx MicroBlaze Reference Guide", UG081 (v7.0), September 15, 2006, available at: http://www.xilinx.com.

[2] D. Meintanis, I. Papaefstathiou: "Power Consumption Estimations vs Measurements for FPGA-based Security Cores"; International Conference on Reconfigurable Computing and FPGAs 2008 (ReConFig 2008), Cancun, Mexico, December 2008.

[3] J. Becker, M. Huebner, M. Ullmann: "Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations"; In Proc. of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03), Sao Paulo, Brazil, September 2003.

[4] K. Poon, A. Yan, S.J.E. Wilton: "A Flexible Power Model for FPGAs"; In Proc. of 12th International Conference on Field-Programmable Logic and Applications (FPL 2002), September 2002.

[5] F. Najm: "Transition density: a new measure of activity in digital circuits"; IEEE Transactions on Computer-Aided Design, vol. 12, no. 2, pp. 310-323, February 1993.

[6] K. Weiss, C. Oetker, I. Katchan, T. Steckstor, W. Rosenstiel: "Power estimation approach for SRAM-based FPGAs"; In Proc. of International Symposium on Field Programmable Gate Arrays (FPGA'00), pp. 195-202, Monterey, CA, USA, 2000.

[7] V. Degalahal, T. Tuan; "Methodology for high level estimation of FPGA power consumption"; In Proc. of ASP–DAC 2005 Conference, Shanghai, January 2005.

[8] "Xilinx Power Estimator User Guide", UG440 (v3.0), June 24, 2009, available at: http://www.xilinx.com.

[9] "Development System Reference Guide", v9.2i, Chapter 10 XPower, available at: http://www.xilinx.com.

[10] "Embedded System Tools Reference Manual", Embedded Development Kit, EDK 9.2i, UG111 (v9.2i), September 05, 2007, Chapter 3, available at: http://www.xilinx.com.

[11] "Fast Simplex Link (FSL) Bus (v2.00a)"; DS449 Dec. 1, 2005, available at http://www.xilinx.com.

[12] "Virtex-4 FPGA Configuration User Guide", UG071 (v1.11), June 9, 2009, available at: http://www.xilinx.com.

[13] "Virtex-4 FPGA User Guide", UG070 (v2.6), December 1, 2008, available at: http://www.xilinx.com.

[14] C. A. R. Hoare: "Quicksort"; Computer Journal, vol. 5, 1, 10–15 (1962).

[15] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, R. P. Jacobi: "A Hardware Accelerator for Fast Retrieval of DIALIGN Biological Sequence Alginments in Linear Space"; IEEE Transactions on Computers, vol. 59, no. 6, pp. 808-821, 2010.

[16] Alpha-Data: http://www.alpha-data.com

Novel Approach for Modeling Very Dynamic and Flexible Real Time Applications

Ismail Ktata[1,2], Fakhreddine Ghaffari[1], Bertrand Granado[1] and Mohamed Abid[2]

[1]*ETIS Laboratory, CNRS UMR8051, University of Cergy-Pontoise, ENSEA, 6 avenue du Ponceau F95000 Cergy-Pontoise, France*
[2]*Computer & Embedded Systems Laboratory (CES), National School of Engineers of Sfax, (ENIS), B.P.W. 3038 Sfax, Tunisia*
[1]Email: {firstname.name}@ensea.fr
[2]Email: {firstname.name}@enis.rnu.tn

Abstract

Modeling techniques are used to solve a variety of practical problems related to processing and scheduling in several domains like manufacturing and embedded systems. In such flexible and dynamic environments, there are complex constraints and a variety of unexpected disruptions. Hence, scheduling remains a complex and time-consuming process and the need for accurate models for optimizing this process is increasing. This paper deals with dynamically reconfigurable architectures which are designed to support complex and flexible applications. It focuses on defining a solution approach for modeling such applications where there is significant uncertainty in the duration, resource requirements and number of tasks to be executed.

Keywords: ***Dynamically Reconfigurable Architecture, uncertainty, scheduling, modeling methodologies, DFG.***

I. Introduction

Today, integrated silicon applications are more and more complex. Moreover, in spite of its performance, ASICs development is still long and very expensive, and provides inefficient solutions for many applications which are composed of several heterogeneous tasks with different characteristics. In addition, the growing complexity of real-time applications today presents important challenges, in great part due to their dynamic behavior and uncertainties which could happen at runtime [1]. To overcome these problems, designers tend to use dynamically reconfigurable architectures (DRA). The development of the latter opens new horizons in the field of architecture design. Indeed, the DRAs are well suited to deal with the dynamism of new applications and allow better compromise between cost, flexibility and performance [2]. In particular, fine grained dynamically reconfigurable architectures (FGDRA), as a kind of DRAs, can be adapted to any application more optimally than coarse grain DRAs. This feature makes them today an interesting solution when it comes to handle computational tasks in a highly constrained context. However, this type of architecture makes the applications design very complex [3], especially with the lack of suitable and efficient tools. This complexity could be abstracted at some level in two ways: at design time by providing design tools and at run time by providing an operating system that abstracts the lower level of the system [4]. Moreover, such architecture requires the presence of an appropriate operating system that could manage new tasks at run time and under different constraints. This operating system, and to effectively manage dynamic applications, has to be able to respond rapidly to events. This can be achieved by providing a suitable scheduling approach and dedicated services like hardware preemption that decreases configurations and contexts transfer times. To realize an efficient schedule of an application, this operating system needs to know the behavior of this application, in particular the part where the dynamicity can be exploited on a DRA.

In this paper, we are interested in the modeling of applications that could be executed on dynamically reconfigurable architecture. This kind of applications is characterized, in addition to its real-time constraints, by several types of flexibility. The purpose is to improve the performance of the modeling techniques which facilitates the job to design an efficient scheduling approach.

The remainder of this paper is structured as follows: in Section 2, brief review is given about the context and the related work on modeling techniques used in different domains. Section 3 describes our new technique modeling applications targeted to DRA. The Section 4 reports a description of the proposed modeling method and comparisons with other models, while the last Section draws conclusions.

II. Context and problem definition

Today, embedded systems are more and more used in several domains: automobiles, robots, planes, satellites, boats, industrial control systems, etc. An important feature of these systems is to be reactive. A reactive system is a system that continuously reacts to its environment at a rate imposed by this environment itself. It receives inputs from the environment, responds to these stimuli by making a number of operations and produces the outputs used by the environment, known as reactions. Dynamically reconfigurable architectures are an interesting solution for this type of applications. Due to that emerging range of applications with dynamic behavior, dynamic scheduling for reconfigurable system-on-chip (RSoC) platforms has become an important field of research [2].

A. *Problematics*

This paper deals with the constraint-based scheduling for real-time applications executed on FGDRA. In particular, we focus on two major problems:

- The modeling of the application that should exhibit its dynamical aspects and must allow the expression of its constraints, in particular real-time constraints.
- The run-time performance of the scheduling algorithm that must be reasonable in term of overhead for a typical application.

The different components of a scheduling problem are the tasks, the potential constraints, the resources and the objective function. Tasks execution must be programmed to optimize a specific objective with the consideration of several criteria. Many resolution strategies have been proposed in literature [5]. Usually these methods assume that processing times can be modeled with deterministic values. They use predictive schedule that gives an explicit idea of what should be done. Unfortunately, in real environments, the probability of a pre-computed schedule to be executed exactly as planned is low [6]. This is because of not only variations, but also because of a lot of data that are only previsions or estimations. It is then necessary to deal with uncertainty or flexibility in the process data. Hence, for an effective resolution, we need to make a significant reformulation of the problem and the solving methods in order to facilitate the incorporation of this uncertainty and imprecision in scheduling [7].

Uncertainty in scheduling may arise from many sources [8]:

- The release time of tasks can be variable, even unexpected.
- New unexpected tasks may occur.
- Cancellation or modification of existing tasks.
- The execution order of tasks on resources can be changed.
- Resources may become unavailable.
- Tasks assignments: if a task could be done on different resources (identical or not), the choice of this resource can be changed. This flexibility is necessary if such a resource becomes unusable or less usable than others.
- The ability to change execution mode: this mode includes the approval or disapproval of preemption, whenever a task could be resumed or not, the overlap between tasks, changing the range of a job, taking into account whether or not a time of preparation, changing the number of resources needed for a task, etc.

We are considering real cases where some variations could occur and some data may change over the forecast. The model has to be few sensible to data uncertainties and variations, and be flexible to be adaptable to the possible disturbances.

B. *Scheduling under uncertainty*

In general, there are two main approaches dealing with uncertainty in a scheduling environment according to phases in which uncertainties are taken into account [8]:

- Proactive scheduling approach aims to build a robust baseline schedule that is protected as much as possible against disruptions during schedule execution. It takes into account uncertainties only in design phase (off-line). Hence, it constructs predictive schedule based on statistical and estimated values for all parameters, thus implicitly assuming that this schedule will be executed exactly as planned. However, this could become infeasible during the execution due to the dynamic environment, where unexpected events continually occur. Therefore, in this case, a reactive approach may be more appropriate.
- Instead of anticipating future uncertainties, reactive scheduling takes decisions in real-time when some unexpected events occur. A reference deterministic scheduling, determined off-line, is sometimes used and re-optimized. In general, reactive methods may be more appropriate for high degrees of uncertainty, or when information about the uncertainty is not available.

A combination of the advantages of both precedent approaches is called proactive-reactive scheduling. This hybrid method implies a combination of a proactive strategy for generating a protected baseline schedule with a reactive strategy to resolve the schedule infeasibilities caused by the disturbances that occur during schedule execution. Hence, this scheduling/rescheduling method permits to take into account uncertainties all over the execution process and ensures better performance [9] [10]. For rescheduling, the literature provided two main strategies: schedule repair and complete rescheduling. The first strategy is most used as it takes less time and preserves the system stability [11].

Scheduling techniques are quite different depending on the nature of the problem and the type of disturbance considered: resources failure, the duration of the variation and the fact that new tasks can occur, etc. The mainly used methods are dispatching rules, heuristics, metaheuristics and artificial intelligence techniques [12]. In [13] authors considered a scheduling problem where some tasks (called "uncertain tasks") may need to be repeated several times to satisfy the design criteria. They used an optimization methodology based on stochastic dynamic programming. In [14] and [15] scheduling problem with uncertain resource availabilities was encountered. Authors used proactive-reactive strategies and heuristic techniques. Another uncertainty case, which is uncertain tasks duration, had been studied in [16] and [17]. Authors discuss properties of robust schedules, and develop exact and heuristic solution approaches.

C. *Scheduling model*

To study a system we need models to describe it, including significant system characteristics of geometry, information and dynamism. The latter is a crucial system characteristic as it permits to represent how a system behaves and changes states over time. Moreover, dynamic modeling can cover different domains from the very general to the very specific [18]. Model types have different presentations, as shown in figure 1: some are text-based using symbols while others have associated diagrams.

- Graphical models use a diagram technique with named symbols that represent process and lines that connect the symbols and represent relationships and various other graphical notations to represent constraints (figure 1(a), (b) (d)).
- Textual models typically use standardized keywords accompanied by parameters (figure 1(c)).

In addition, some models have static form, whereas others have natural dynamics during model execution as in figure 1 (a). The solid circle (a token) moves through the network and represents the executional behavior of the application.

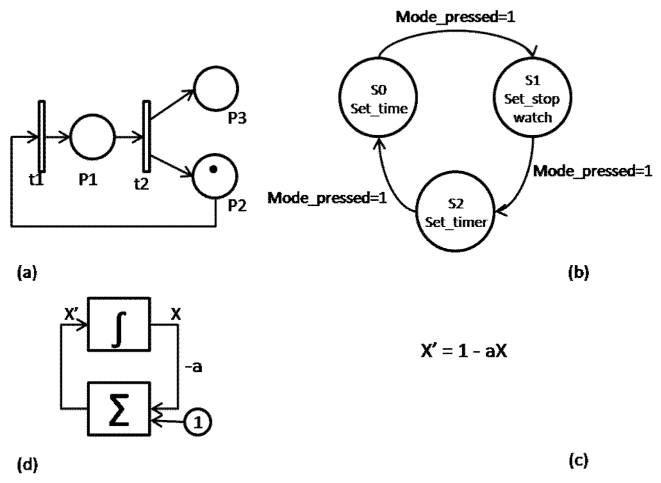


Figure 1. Four types of dynamic system models. (a) Petri net. (b) Finite state machine. (c) Ordinary differential equation. (d) Functional block model.

In the domain of embedded systems, a large number of modeling languages have been proposed [19], [20], [21], including extensions to finite state machines, data flow graphs, communicating processes, and Petri nets, among others. In this section we present main models of computation for real-time applications reported in the literature.

- Finite State Machines

The classical Finite State Machine (FSM) representation is probably the most well-known model used for describing control systems. However, one of the disadvantages of FSMs is the exponential growth of the number of states that have to be explicitly captured in the model as the system complexity increases making the model increasingly difficult to visualize and analyze [22]. For dynamic systems, the FSM representation is not appropriate because the only way to model such kind of systems is to create all the states that represent the dynamic behavior of the application. It is then unthinkable to use it as the number of states could be prohibitive.

- Data-Flow Graph

A data-flow graph (DFG) is a set of compute nodes connected by directed links representing the flow of data. It is very popular for modeling data-dominated systems. It is represented by a directed graph whose nodes describe the processing and the arcs represent the partial order followed by the data. However, the conventional model is inadequate for representing the control unit of systems [23]. It provides no information about the ordering of processes. It is therefore inappropriate to model dynamic applications.

- Petri Net

Petri net (PN) is a modeling formalism which combines a well-defined mathematical theory with a graphical representation of the dynamic behavior of systems [18]. Petri Net is a 5-tuple PN = (P, T, F, W, Mo) where: P is a finite set of places which represent the status of the system before or after the execution of a transition. T is a finite set of transitions which represent tasks. F is a set of arcs (flow relation). W: F ➔ {1, 2, 3, ... } is a weight function. M_0 is the initial marking. However, though Petri net is well-established for the design of static systems, it lacks support for dynamically modifiable systems [24]. In fact, the PN structure presents only the static properties of a system while the dynamic one results from PN execution which requires the use of tokens or markings (denoted by dots) associated with places [25]. The conventional model suffers from good specification of complex systems like lack of the notion of time which is an essential factor in embedded applications and lack of hierarchical composition [26]. Therefore, several formalisms have independently been proposed in different contexts in order to overcome the problems cited above, such as introducing the concepts of hierarchy, time, and valued tokens. Timed PNs are those with places or transitions that have time durations in their activities. Stochastic PNs include the ability to model randomness in a situation, and also allow for time as an element in the PN. Colored PNs allow the user and designer to witness the changes in places and transitions through the application of color-specific tokens, and movement through the system can be represented through the changes in colors [18].

Most of the methodologies mentioned provide no sufficient support for systems which include variable dynamic features. Dynamic creation of tasks for instance is not supported by most of the systems above mentioned [26]. In [26], authors proposed an extension of high-level Petri net model [27] in order to capture dynamically modifiable embedded systems. They coupled that model with graph transformation techniques and used a double pushout approach which consists of the replacement of a Petri net by another Petri net after firing of transitions. This approach allows modeling dynamic tasks creation but not variable execution time nor variable number of needed resources.

III. Proposed method

Before beginning to describe our modeling method, we define the constraints that typically appear in dynamic systems. In our case, we consider a firm real-time context. In fact, for actually developed applications, especially multimedia and control applications, tardiness or deadline violations results only in degradation of Quality of Service (QoS) without affecting good processing. In this context hardware tasks are characterized by the following parameters:

- Execution time (Ci),
- Deadline (Di),
- Periodicity (Pi),
- Precedence constraints among tasks,
- Tasks could be preempted,
- Used resources of the DRA.

In real world, all characteristics may change: tasks (the release time, deadline and execution time), as well as the availability of resources. There are several types of changes like uncertainty, unexpected changes and values variation. For our study, we will consider three cases of dynamic scheduling problems for dynamic applications:

(a) The number of tasks is not fixed. It may change from iteration to another.

(b) The tasks execution time may change too.

(c) The number of needed resources for tasks execution is variable. In addition, the number of available resources may decrease after a failure occurs.

For those cases, the goal is to develop a robust scheduling method that is little sensible to data uncertainties and variations between theory and practice. In addition, the schedule has to be flexible to be adaptable to the possible disturbances. Therefore, we consider a proactive-reactive approach. The proactive technique (a reference schedule computed offline for the static part) is used to facilitate the execution of the reactive strategy online, so that scheduling decisions have a better quality and produced in a shorter time. To represent all those constraints in the same model and to be adequate for the adopted scheduling approach, our graph will be composed of two forms of nodes. The first type of nodes refers to static tasks which are known in advance and whose execution is permanent during the whole application of the lifecycle execution. The second type is for dynamic tasks which could be executed in a cycle and not in others and with a variable number of needed resources. Therefore, the first step is to separate the two parts of the application (static and dynamic). Then, a priority-based schedule is established offline for static part. Such a schedule serves very important functions. The first is to allocate cells of the hardware DRA to the different hardware tasks. The second is to serve as a basis for planning tasks that may occur during execution. The basic idea is to sort static tasks from the graph and schedule them according to a priority scheme, while respecting their precedence constraints (topological order). Tasks are executed at the earliest possible time. If there is equality between some tasks, task which has maximum execution time will have priority to be launched before the others. At runtime, the objective is to generate a new schedule that deviates from the original schedule as little as possible so that the repair operations will be mostly simple. Therefore, the online scheduler must take into account the next tasks to be performed with their dynamic aspects (different duration, more or less instances to execute, more or less number of needed cells for execution). It has to prefetch configurations context of new tasks in the columns that are available for executing and to find the possible way to integrate them in the current schedule without effect on performance. This schedule repair must rely on rapid algorithms based on a simple scheduling technique so that it can perform online execution with no overhead. It consists in finding a suitable partitioning of N tasks, forming the application, to be executed on M target resources of the hardware devices. In addition, tasks execution order has to meet the real time constraints. This scheduling problem is known to be NP-hard [28] [29]. In this context, heuristics are schedule repair methods which offer fast and reasonably good solution but do not guarantee to find an optimal schedule.

We make use of the example shown in figure 3 in order to illustrate the different definitions corresponding to our model. For this example, the set {T1, T2, T3, T4, T5, T6, T7, T8} represents the static tasks which are always executed in each period and {T9, T10, T11} are dynamic tasks which may be executed in some period but not in others and whose number of resource requirements is variable. Each task is represented with time characteristics: Ci for the execution time and Di for the deadline.

The first dynamic feature considered in this model is tasks with variable execution time. To represent that case, we are inspired by the Program Evaluation and Review Technique (PERT) [30], which is a network model that allows randomness in activity execution times. For each task, PERT indicates a start date and end time at the earliest and latest. The chart identifies the critical path which determines the minimum duration of the project. Tasks are represented by arcs with an associated number presenting the tasks duration. Between arcs, we find circles marking events of beginning or end of tasks (figure 2). In this model, we replace the release time feature by the execution time as it would be changed over execution, and we keep the deadline as it will be useful to minimize the makespan (or the length of the schedule). In figure 3, tasks with variable execution time are {T1, T7, T8}. This will be noticed by the use of asterisk.

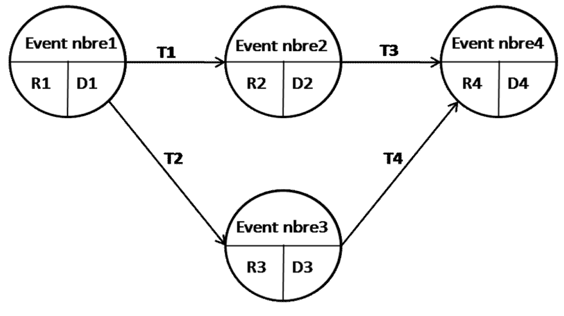


Figure 2. PERT graph

To be executed, hardware tasks need a minimum number of resources. The percentage of this minimal number is indicated in labels over tasks nodes. This case is frequently found on some computer vision applications where a first task is detecting objects and then a particular processing is applied on each detected object. To execute multiple instances of this process, the minimum needed number of resources will be multiplied by the number of instances. In figure 3, task T9 will be executed n times. T9 is represented with circled node and the minimum needed number of resources indicated in the label will be multiplied by n. If the integer number n=0 then task will not occur. The instance number is unexpected from the static phase and decision will be taken online. The number n will depend on the previous executions and the actual input data to be processed (such as keypoints in the robotic vision application). Thus, n will be recalculated, after each period, based on its previous values. For more details, we take an example of execution. In each iteration the scheduler will have two values of n: n_p which is a predicted value of n to be used by the scheduler for the next period, and n_r which is the real value of n for the executed task. At t=0, let $n= n_p =0$ (no prediction to execute T9). In the first execution, the application needs $n= n_r =10$ instances of T9. So, for the second iteration, the scheduler will predict to execute $n= n_p =10$ instances of T9, while, in the real execution, $n= n_r =6$. For the next execution, n_p could be: the maximum of precedent values, the highest values of n, the average of real last values ($n= n_p =8$), or a Gaussian like probability which is a typical realistic distribution, etc. The choice will depend on the application. For the example of a moving robot which need to predict the direction and the presence or absence of obstacles, it will be preferred to take the last real values with different weights. In our model, the number n is represented above the arcs. The arcs represent the dependencies between tasks. For static (permanent) tasks, we represent arcs with solid lines, while unpredictable dependencies are represented by dashed lines.

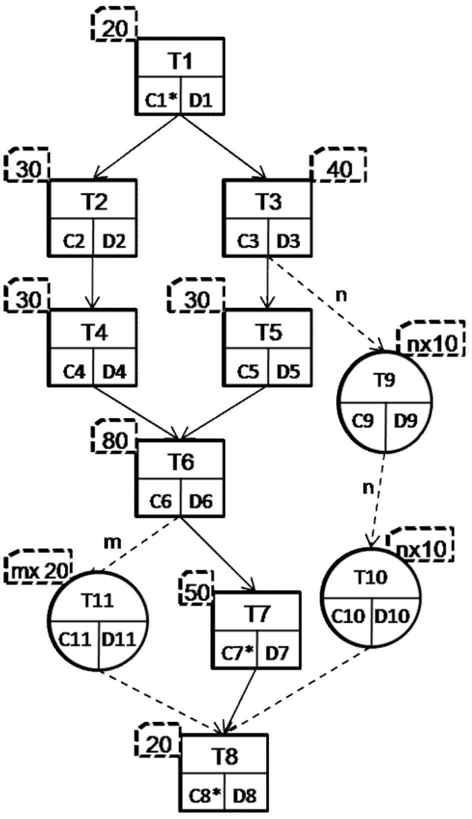


Figure 3. New model for dynamic application

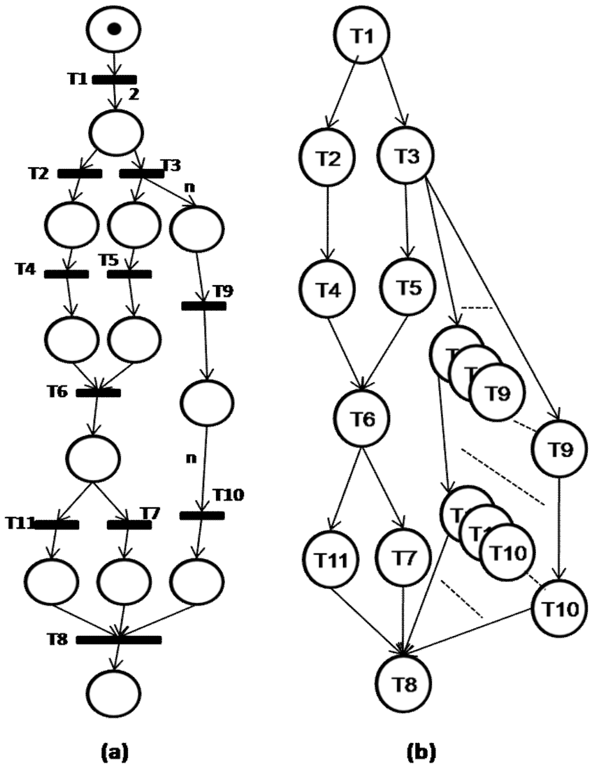


Figure 4. (a) A Petri net representation of the example of figure 3. (b) A DFG representation of the example of figure 3.

On the reconfigurable device, resource failure may occur which will affect the resource availability. Thus, execution may occur or not depending on the available resources compared with needed ones. A variable, which is initialized as the total number of resources, will indicate the amount of remaining resources. From the ready list, algorithm determines the tasks that can be executed on the reconfigurable device. Tasks with the higher priorities will be placed first until the area of device is fully occupied.

IV. Comparison of models

When we compare with other models (section 2-C), the proposed technique presents several advantages. For unpredicted number of occurring tasks, the data flow graph does not contain information about the number of instances. During execution of the application, every task represented by the nodes of DFG is executed once in each iteration [31]. Only when all nodes have finished their executions, a new iteration can start. So to model this, we need to represent *n* nodes of the same task, which increases the size of the model (see figure 4(b)). In our model, information about number of instances of a same task to be executed is noted by the circle form of the task and the indicated number above its arc. For PN model, (see figure 4(a)), arcs could be labeled with their weights where a k-weighted arc can be interpreted as the set of k parallel arcs [32]. But, from its definition (section 2-C), weights are positive integers, so it cannot present a fictive arc with non firing transition representing a task that may not be executed in some iterations. In figure 4(a), if T9 and T10 are not executed, then *n* should be null, which is impossible from PN definition. In addition, to fire T8 all input places should have at least one token, which will be not possible if T10 or T11 was not executed (fired).

Timing information is useful for determining minimum application completion time, latest starting time for an activity which will not delay the system, and so on. We have inspired from the PERT chart to explicitly represent useful time features that are, in our case, execution time and deadline. However, Petri net does not provide any of this type of information. The only important aspect of time is the partial ordering of transitions. For example, it presents variable tasks duration with a set of consequent transitions for each task which will complicate the model (see figure 5). The addition of timing information might provide a powerful new feature for Petri nets but may not be possible in a manner consistent with the basic philosophy of Petri nets Research [33].

For resource representation, PN models this feature by an added place with a fixed number of tokens. To begin execution, a task removes a token from the resource place and gives it back in the end of its execution. However, this model is inadequate in our case since the number of available resources may change over the execution.

Therefore, the use of conventional modeling methods is not effective in our case. In fact, with PN and DFG model (figure 4), there is no distinction between static tasks and dynamic ones (that may not be executed), nor an explicit notion of time (as variable execution time of some tasks).

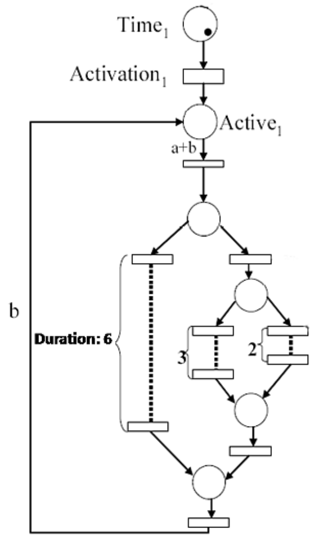


Figure 5. PN model for variable tasks duration

The main advantage of the proposed method is the possibility to present several dynamic features of real time applications with the minimum of nodes and thus in a simple formalism. Indeed, from the first sight of the model, we can bring out three main characteristics of this model:

- The distinction between the static execution (which is presented by the squared nodes) and the dynamic execution i.e. the tasks whose execution and number of instances is uncertain (presented by the circled nodes),
- The tasks whose execution time is variable (presented by the asterisk),
- The percentage, for each task, of needed resources for its execution. In each iteration, and depending on the available resources, schedule is able to decide which ready tasks could be executed on the device.

Those informations will be useful further for the scheduler. We consider a 1D area model as it is a commonly used reconfigurable resource model, but even 2D area model could be considered. In that considered 1D model, tasks can be allocated anywhere along the horizontal device dimension; the vertical dimension is fixed and spans the total height of the hardware task area (see figure 6).

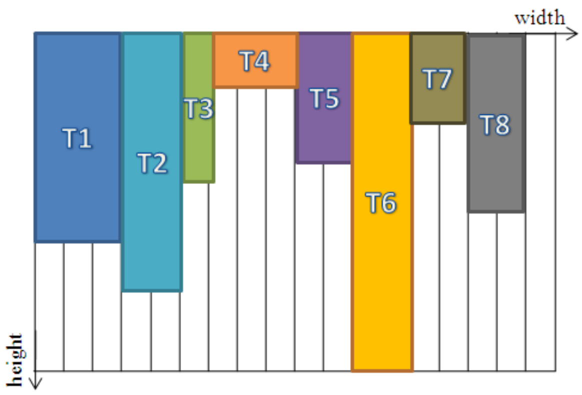


Figure 6. 1D area model of a reconfigurable device

Based on the proposed tasks model, statically defined tasks represented by squared nodes, will be scheduled and placed on the reconfigurable device during the proactive phase. Whereas dynamically defined tasks, which are represented by circled nodes, will be scheduled online in a reactive phase. This online decision will take into account the variable parameters and try to fit into the set of already guaranteed tasks, to delay or to reject these new dynamic tasks.

V. Conclusion

This paper presented a particular modeling problem dealing with the implementation of dynamic and flexible applications on dynamically reconfigurable architecture. The purpose is to consider most of the dynamic features supported by the architecture and to present them in an easy and efficient method. For that point we have proposed a model, based on some features of existing modeling techniques, and which is more dedicated to dynamic real time applications. The main advantage of our specification model is the possibility to obtain more exact scheduling characteristics from the representation. Those characteristics include the distinction between static and dynamic occurring tasks, bring out tasks whose execution time may change over the time and determine the number of resources needed for executing hardware tasks. So, we will be able either to take decisions or not. As a reconfigurable architecture, we target OLLAF [4] (Operating system enabled Low LAtency Fgdra), an original FGDRA specifically designed to enhance the efficiency of an RTOS services necessary to manage such architecture. Future works will consist in integrating our scheduling approach among the services of an RTOS taking into account the new possibilities offered by OLLAF.

VI. References

[1] C.Steiger, H.Walder, M.Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks", Computers, IEEE Transactions on Volume 53, Issue 11, Nov. 2004 Page(s): 1393 - 1407.

[2] J. Noguera, R.M. Badia, "Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling", ACM Transactions on Embedded Computing Systems, Volume 3, Issue 2 (May 2004) pp. 385-406.

[3] A. Mtibaa, B. Ouni and M. Abid, "An efficient list scheduling algorithm for time placement problem", Computers and Electrical Engineering 33 (2007) 285–298.

[4] S. Garcia, B. Granado, "OLLAF: a Fine Grained Dynamically Reconfigurable Architecture for OS Support", EURASIP Journal on Embedded Systems, October 2009.

[5] P. Brucker & S. Knust, "Complex Scheduling", Springer Berlin Heidelberg, 2006.

[6] V. T'kindt and J.-C. Billaut, "Multicriteria Scheduling: Theory, Models and Algorithms", Springer-Verlag (Heidelberg), second edition (2006).

[7] N. González, R. Vela Camino, I. González Rodríguez, "Comparative Study of Meta-heuristics for Solving Flow Shop Scheduling Problem Under Fuzziness", Second International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2007, Spain, June 18-21, 2007, Proceedings Part I : 548-557.

[8] A. J. Davenport and J. C. Beck, "A Survey of Techniques for Scheduling with Uncertainty", accessible on-line at http://tidel.mie.utoronto.ca/publications.php on February 2006, 2000.

[9] H.Aytug, M.A.Lawley, K.McKay, S.Mohan, R.Uzsoy, "Executing production schedules in the face of uncertainties: A review and some future directions", European Journal of Operational Research 161, 2005, p86-110.

[10] W.Herroelen and R.Leus, "Project scheduling under uncertainty: Survey and research potentials", European Journal of Operational Research, Vol. 165(2) (2005) 289--306.

[11] G.E.Vieira, J.W.Hermann, and E.Lin, "Rescheduling manufacturing systems: a framework of strategies, policies and methods", Journal of Scheduling, 6 (1), 36-92 (2003).

[12] D. Ouelhadj, and S. Petrovic, "A survey of dynamic scheduling in manufacturing systems", Journal of Scheduling, 2008.

[13] Peter B. Luh, Feng Liu and Bryan Moser, "Scheduling of design projects with uncertain number of iterations", European Journal of Operational Research, 1999, vol. 113, issue 3, pages 575-592.

[14] O.Lambrechts, E.Demeulemeester, W.Herroelen, "Proactive and reactive strategies for resource-constrained project scheduling with uncertain resource availabilities", Journal of scheduling 2008, vol.11, no.2, pp. 121-136.

[15] S.Liu, K.L.Yung, W.H.Ip, "Genetic Local Search for Resource-Constrained Project Scheduling under Uncertainty", International Journal of Information and Management Sciences 2007, VOL 18; NUMB 4, pages 347-364.

[16] M. Turnquist and L. Nozick, "Allocating time and resources in project management under uncertainty", Proceedings of the 36th Annual Hawaii International Conference on System Sciences, Island of Hawaii, January 2003.

[17] J. Christopher Beck, Nic Wilson, "Proactive Algorithms for Job Shop Scheduling with Probabilistic Durations", Journal of Artificial Intelligence Research 28 (2007) 183–232.

[18] P.A Fishwick, "Handbook of dynamic system modeling", Chapman & Hall/CRC Computer and Information Science Series 2007.

[19] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vicentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis", Proc. IEEE, 85(3):366–390, March 1997.

[20] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich, "Models of computation for embedded system design". In A. A. Jerraya and J. Mermet, editors, System-Level Synthesis, pages 45–102, Dordrecht, 1999. Kluwer

[21] A. Jantsch. "Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation". Morgan Kaufmann, San Francisco, CA, 2003.

[22] L. Alejandro Cortes, "Verification and Scheduling Techniques for Real-Time Embedded Systems", Ph. D. Thesis No. 920, Dept. of Computer and Information Science, Linköping University, March 2005.

[23] L. Alejandro Cortés, P. Eles and Z. Peng, "A Survey on Hardware/Software Codesign Representation Models", SAVE Project, Dept. of Computer and Information Science, Linköping University, Linköping, June 1999.

[24] Carsten Rust, Franz J. Rammig: "A Petri Net Based Approach for the Design of Dynamically Modifiable Embedded Systems". DIPES 2004: 257-266.

[25] M. Tavana, "Dynamic process modelling using Petri nets with applications to nuclear power plant emergency management", Int. J. Simulation and Process Modelling (2008), Vol. 4, No. 2, pp.130–138.

[26] Franz-Josef Rammig, Carsten Rust, "Modeling of Dynamically Modifiable Embedded Real-Time Systems", WORDS Fall 2003: 28-34.

[27] E. Badouel and J. Oliver. "Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes". In Proc. of a workshop within the 19th Int'l Conf. on Applications and Theory of Petri Nets, 1998.

[28] Michael Garey and David Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness", Freeman, 1979.

[29] Z.A. Mann, A. Orbán, "Optimization problems in system-level synthesis". Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications, Tokyo-Japan (2003).

[30] F. Chauvet, J.-M. Proth, "The PERT Problem with Alternatives: Modelisation and Optimisation", Report N° RR-3651 (1999) SAGEP (INRIA Lorraine) France.

[31] Oliver Sinnen. "Task Scheduling for Parallel Systems" (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience, 2007.

[32] Tadao Murata, "Petri nets: Properties, Analysis and Applications", Proceedings of the IEEE, 77(4):541-574, April 1989.

[33] James L. Peterson, "Petri net theory and the modeling of systems", Prentice Hall PTR, 1981.

New Three-level Resource Management for Off-line Placement of Hardware Tasks on Reconfigurable Devices

Ikbel Belaid, Fabrice Muller

University of Nice Sophia-Antipolis
LEAT-CNRS, France
e-mail: belaid@unice.fr, fmuller@unice.fr

Maher Benjemaa

Research Unit ReDCAD
National engineering school of Sfax
Tunisia
e-mail: Maher.Benjemaa@enis.rnu.tn

***Abstract*—The FPGA devices are widely used in reconfigurable computing systems, these devices can achieve high flexibility to adapt themselves to various applications by reconfiguring dynamically portions of the dedicated resources. This adaptation with the application requirements reaches better performance and efficient resource utilization. However, the run-time partial reconfiguration brings more complex partitioning of the FPGA reconfigurable area. This issue implies that efficient task placement algorithm is required. Many on-line and off-line algorithms designed for such partially reconfigurable devices have been proposed to provide efficient hardware task placement. In these previous proposals, the quality of hardware task placement is measured by the resource wastage and task rejection and major of these research works disregard the configuration overhead. Moreover, these algorithms optimize these criteria separately and do not satisfy all goals. These considerations can not reflect the overall situation of placement quality. In this paper, we have interested in off-line placement of hardware tasks in partially reconfigurable devices and we propose a novel three-level resource management for hardware task placement. The proposed off-line resource management is based on mixed integer programming formulation and enhances placement quality which is measured by the rate of task rejection, resource utilization and configuration overhead.**

Keywords-hardware task placement; mixed integer programming; run-time reconfiguration

I. INTRODUCTION

The FPGA devices get faster and larger due to the high density of their heterogeneous resources. Consequently, the number and the complexity of modules to load on them increases, hence better performance can be achieved by exploiting FPGAs in reconfigurable computing systems. Furthermore, the ability to reconfigure the FPGA partially as it is running speeds up the applications in reconfigurable systems. The technique of run-time partial reconfiguration also improves the performance of hardware task scheduling. For hardware tasks, placement and scheduling are strongly linked; the scheduler decision should be taken in accordance with the ability of placer to allocate free resources in reconfigurable hardware device to the elected task. While some proposed techniques increase the performance of scheduling as well as of application, these techniques suffer from placement problems: resource wastage, task rejection and configuration overheads. In this paper, under FOSFOR project[1], we focus mainly on an efficient management of hardware tasks on reconfigurable hardware devices by taking advantage from the run-time reconfiguration. Nowadays, heterogeneous SRAM-based FPGAs are the most prominent reconfigurable hardware devices. In this work, we target the recent Xilinx column-based FPGAs to optimize the quality of hardware task placement basing on mixed integer programming formulation and by using powerful solvers which rely on the complete non-exhaustive resolution method called Branch and Bound. Experiments are conducted on an application of heterogeneous tasks and an improvement in placement quality was shown by 30 % as an average rate of resource utilization which achieves up 27 % of resource gain comparing to static design. In the worst case, the resulted configuration overhead is 10 % of the total running time and we discarded the issue of task rejection.

The rest of the paper is organized as follows: the next section reviews some related work of hardware task placement. Section 3 details our three-level off-line strategy of resource management on FPGA. Section 4 depicts the formulation of placement problem as mixed integer programming. Section 5 describes the obtained results and the evaluation of placement quality. Concluding remarks and future works are presented in section 6.

II. RELATED WORK

Placement problem consists of two main functions: i) *the partitioning* that handles the free resource space to identify the potential sites for hardware task execution and ii) *the fitting* that selects the feasible placement solution. Many research groups investigated on the placement of hardware tasks on FPGAs. Current strategies dealing with task placement are divided into two categories: off-line placement and on-line placement.

[1]FOSFOR (Flexible Operating System FOr Reconfigurable platforms) is French national program (ANR) targeting the most evolved technologies. Its main objective is to design a real time operating system distributed on hardware and software execution units which offers required flexibility to application tasks through the mechanisms of dynamic reconfiguration and homogeneous Hw/Sw OS services.

A. *On-line Methods for Hardware Task Placement*

The main reference is [1], Bazargan et al. propose on-line scenario and introduces two partitioning techniques: Keeping All Maximal Empty Rectangles and Keeping Non-overlapping Empty Rectangles. Both techniques manage free resource space to search the empty holes. Nevertheless, the fitting is conducted by bin-packing rules. In [2], Walder et al. deals with 2D on-line placement by relying on efficient partitioning algorithms that enhance the Bazargan's partitioning such as On-The-Fly partitioning. The Walder's partitioner delays split decision instead of using the Bazargan's heuristics for decision of split and uses hash matrix data structure that finds a feasible placement in constant time. Ahmadinia et al. presents in [3] a new method of on-line placement by managing the occupied space on the device and by fitting the tasks on the sites by means of Nearest Possible Position method that reduces the communication cost. Some metaheuristics are adopted to resolve the hardware task placement such as [4] that employs an on-line task rearrangement by using genetic algorithm approach. In [4], when a new arriving task could not be placed immediately by first-fit strategy, the proposed approach combining two genetic algorithms allows the task rotation and tries to rearrange a subset of tasks executing on the FPGA to allow the processing of the pending task sooner.

B. *Off-line Methods for Hardware Task Placement*

In the off-line scenario for hardware task placement, [1] defines 3D templates in time and space dimensions and uses simulated annealing and greedy research heuristics. By considering the placement of hardware tasks as rectangular items on hardware device as rectangular unit, several approaches for resolving the two-dimensional packing problem are proposed. For example, in [5], the off-line approximate heuristics: Next-Fit Decreasing Height, First-Fit Decreasing Height and Best-Fit Decreasing Height are presented as strip packing approaches based on packing items by levels. Lodi et al. propose also in [6] and [7] different off-line approaches to resolve hardware task placement as 2D bin-packing problem for instance Floor-Ceiling algorithm and Knapsack packing algorithm. The Knapsack packing algorithm proposed in [7] initializes each level by the tallest unpacked item and completes it by packing tasks as the associated Knapsack problem that maximizes the total area within level. In [8], as bin packing problem, an off-line approach is proposed by Fekete et al. through a graph-theoretical characterization of the packing of a set of items into a single bin. Tasks are presented as three-dimensional boxes and the feasible packing is decided by the orthogonal packing problem within a given container. Their approach considers packing classes, precedence constraints and the edge orientation to solve the packing problem. Similarly, in [9], Teich et al. defines the task placement as more-dimensional packing problem. From a set of tasks modeled as 3D polytopes with two spatial dimensions and the time of computation and basing on packing classes as well as on a fixed scheduling, they search a feasible placement on a fixed-size chip to accommodate a set of tasks. The resolution is performed by Branch and Bound technique to optimality of dynamic hardware reconfiguration. By optimizing the total execution time and the resource utilization, the method of placement in [10] proposed by Danne and Stuehmeier consists of two phases. The first phase is the recursive bi-partitioning by means of slicing tree that defines the relative position of each hardware task towards the other hardware task placement and finds the appropriate room in the reconfigurable device for each hardware task according to task's resources and inter-task communication. The second phase uses the obtained room topology to achieve the sizing that computes the possible sizes for each room.

Major of the existing strategies provide a non-guarantee system as they suffer from task rejection and resource wastage. Major of the proposed methods of placement are applicable only in homogeneous devices and addresses near-identical and non-preemptive hardware tasks. As we have full knowledge about the set of hardware tasks and the features of the reconfigurable device, in this work, we present a realistic three-level resource management solution as a new strategy to perform off-line placement of hardware tasks in heterogeneous FPGA.

III. THREE-LEVEL OFF-LINE RESOURCE MANAGEMENT

In our three-level resource management, we are based on features of hardware tasks and reconfigurable hardware device. We use Xilinx's Virtex FPGA as a reference for the hardware reconfigurable device to lead our hardware resource management study.

A. *Terminology*

We define few terms which are used to describe the three-level resource management. Throughout the paper, the number of tasks is presented by *NT*, *NZ* is the number of Reconfigurable Zones (RZ), *NR* is the number of Reconfigurable Physical Blocs (RPB) specific for a given RZ and *NP* is the number of Reconfigurable Bloc (RB) types in the chosen technology. We consider two levels of abstraction.

1) Application Level: Each hardware task (T_i) is featured by the worst case execution time (C_i), the period (P_i) and a set of predefined preemption points ($Preemp_{i,l}$) specified by the designer according to the known states in the task behavior and to the data dependency between these states. The number of preemption points of T_i is denoted by $NbrPreemp_i$. In addition, tasks are presented by a set of reconfigurable resources called Reconfigurable Blocs (RB_k). The RBs are the required physical resources in the reconfigurable hardware device to achieve task execution and they define the RB-model of the hardware task as expressed by (1). The determination of the RB-model of hardware tasks is well detailed in our work in [11]. The RBs are the smallest reconfigurable units in the hardware device. They are determined according to the available reconfigurable resources in the device and they match closely its reconfiguration granularity. Each type of RB is characterized by specified cost $RBCost_k$ which is defined according to its frequency in the device, its power consumption and its functionality.

$$T_i\_RB = \{\alpha_{i,k}\, RB_k\}, \alpha_{i,k} \text{ is natural},$$
$$1 \leq i \leq NT \text{ and } 1 \leq k \leq NP \tag{1}$$

2) *Physical Level*

a) *Reconfigurable Zones (RZ):* RZs are virtual blocs customized to model the classes of hardware tasks. RZs separate hardware tasks from their execution units on the reconfigurable device. They are determined through the RB-model of hardware tasks during step 1 of the first level of resource management. Hence, each RZ (RZ_j) is depicted by its RB-model as described by (2).

$$RZ_j\_RB = \{\beta_{j,k}\, RB_k\}, \beta_{j,k} \text{ is natural},$$
$$1 \leq j \leq NZ \text{ and } 1 \leq k \leq NP \tag{2}$$

b) *Reconfigurable Physical Blocs (RPB):* During the placement, the RB-model of RZs are fitted on RPBs partitioned on reconfigurable hardware device. RPBs are 2D physical blocs representing the physical locations of RZs within the reconfigurable area. Each RPB is characterized by four fixed coordinates and is depicted by its RB-model as presented by (3). As RZs are abstractions of hardware task classes, the RPBs are the execution units where the tasks could be placed.

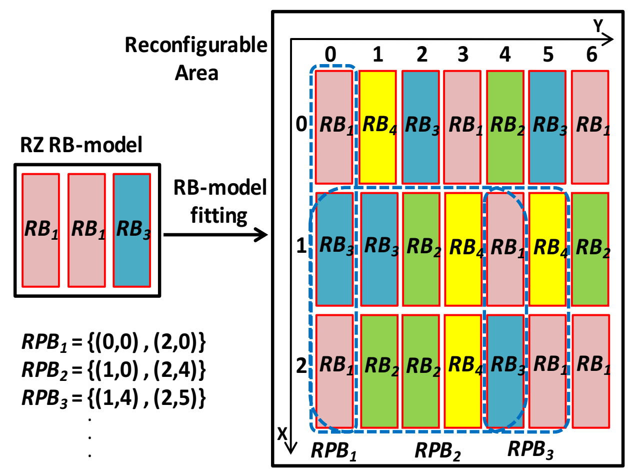


Figure 1. Example of RPBs for RZ.

$$RPB_p\_RB = \{\gamma_{p,k}\, RB_k\}, \gamma_{p,k} \text{ is natural},$$
$$1 \leq p \leq NR \text{ and } 1 \leq k \leq NP \tag{3}$$

Fig. 1 illustrates an example of potential RPBs partitioned on reconfigurable area of the hardware device for fitting RZ requiring two RB_1 and one RB_3.

The management of hardware resources in reconfigurable hardware device to perform off-line placement of hardware tasks consists of three levels described by the three following sections.

B. *Level 1: Off-line Flow of Hardware Task Classification*

Level 1 takes the application tasks as input and provides the types and the instances of RZs. It consists of three steps.

1) *Step 1: RZ Types Search:* It gathers tasks sharing the same types of RBs under the same type of RZ by taking the maximum number of each RB type between tasks. Step 1 is achieved by Algorithm 1.

Algorithm 1. RZ types search or hardware task search.

```
RZ-reference = 0        // references of RZs types
List-RZ                 // list of RZs types
n                       // natural
For all tasks $T_i$ Do     // $T_i\_RB = \{X_{i,k}\, RB_k\}$
    RZ=Create new RZ ($X_{i,k}$)   //$RZ = \{X_{i,k}\, RB_k\}$
    If ((RZ-reference ≠ 0) and (∃ n, 1≤n≤ RZ-reference/ ∀ k (($X_{i,k} \neq 0$ and $Z_{n,k} \neq 0$)
    or ($X_{i,k} = 0$ and $Z_{n,k} = 0$))) then
        // this test checks whether the new created RZ type already exists in list -RZ
          For all k Do
              $Z_{n,k}$= max ($X_{i,k}$, $Z_{n,k}$)          // update RBs number of $RZ_n$
    Else
             Increment RZ-reference
             $RZ_{RZ\text{-reference}}$ = RZ              // $RZ_{RZ\text{-reference}} = \{X_{i,k}\, RB_k\}$
             Insert(list-RZ, $RZ_{RZ\text{-reference}}$)
    END If
END For
```

The maximum number of RZ types is the number of hardware tasks. At the end of step 1, we obtain the tasks classes (RZ_j).

2) *Step 2: Classification of Hardware Tasks:* Step 2 starts by computing cost D between tasks and each RZ type resulting from step 1. Costs D represent the differences on RBs between tasks and RZs, consequently, they express the resource wastage when task is mapped to the RZ. Based on RB-models of task T_i and RZ RZ_j, cost D is computed as follows according to two cases.
We define by (4)

$$d_{i,j,k} = \alpha_{i,k} - \beta_{j,k}$$
$$1 \leq i \leq NT, 1 \leq j \leq NZ \text{ and } 1 \leq k \leq NP \tag{4}$$

Case 1: $\forall\, k,\ d_{i,j,k} \leq 0$, RZ_j contains a sufficient number of each type of RB (RB_k) required by T_i, cost D is equal to the sum of differences in the number of each RB type between T_i and RZ_j weighted by $RBCost_k$ as expressed in (5).

$$D(T_i, RZ_j) = \sum_{1 \leq k \leq NP} RBCost_k \times |d_{i,j,k}| \tag{5}$$

Case 2: $\exists k, d_{i,j,k} > 0$, the number of RBs required by T_i exceeds the number of RBs in the RZ_j or T_i needs RB_k which is not included in RZ_j. In this case cost D is infinite (see (6)).

$$D(T_i, RZ_j) = \infty \tag{6}$$

Step 2 assigns each task to the RZ giving the lowest cost D and by using (7), computes the workload of each RZ according to this assignment.

$$Load\_RZ_j = \sum_{i \ in \ RZ_j} \left(C_i/P_i + NbrPreemp_i \times Overhead_j/P_i \right) \tag{7}$$

$Overhead_j$ denotes the configuration overhead of RZ_j on the target technology. This overhead is computed by conducting the whole Xilinx partial reconfiguration flow from the floorplanning of RZ_j on the device up to partial bitstream creation and by taking into account the configuration frequency (*frequency*) and the width of the selected configuration port (*port width*) as expressed by (8).

$$Overhead_j = bitstream \ size/frequency \times port \ width \tag{8}$$

3) Step 3: Decision of Increasing the Number of RZs

This step is performed when an overload (>100%) is detected within some RZs after step 2. Step 3 lightens the overload in RZs by migrating some execution sections (Exe_i) defined by the predefined preemption points of its assigned tasks to the non-overloaded RZs giving finite D with them. When the overload persists, step 3 increments the number of overloaded RZ till covering its overload. Step 3 is conducted by means of Algorithm 2.
As generic placement, our off-line placement includes the main functions of placement: partitioning and fitting fulfilled by the two following levels of resource management.

C. Level 2: Partitioning of RPBs on the Target Device

In this level, for each RZ resulting from level 1, level 2 searches all its potential physical sites partitioned on the device which are RPBs. During RPB partitioning, we must take into account the heterogeneity of the target device. In fact, the RPBs must contain all the types of RBs required by the RZ and the number of RBs in RPBs must be greater than or equal to the number of RBs in RZs.

Algorithm 2. Decision of increasing the number of RZs.

```
$Load_m$ : the load (%) of overloaded $RZ_m$
$Load_n$ : the load (%) of non-overloaded $RZ_n$
$Load_{n,i}$: the load (%) of non-overloaded $RZ_n$ after adding a section of execution of $T_i$
$Section_i$: the list of combinations of execution sections of task $T_i$
$Exe_i$: the execution section of $T_i$
p,q,r,j,i,l: naturals
$L1$ = {loads of overloaded $RZ_j$}
$L2$ = {loads of non-overloaded $RZ_j$}
$L3$: list of tasks

Sort $L1$ in descending order
Sort $L2$ in ascending order, in case of equality ,
Sort $L2$ in ascending order according to configuration overhead
For p = 1 to size of $L1$ Do  // Browsing overloaded RZs $RZ_m$
  $RZ_m = L1(p)$
  $Load_m = load(RZ_m)$
  q = 1
  While (q ≤ size of $L2$ and $Load_m$>100) Do  // Browsing non-overloaded RZs $RZ_n$
      $RZ_n = L2(q)$
      $Load_n = load(RZ_n)$
      // Search {$Ti$ } from $RZ_m$ to migrate to $RZ_n$
      If ($\exists\{T_i\}$ assigned to $RZ_m / D(T_i , RZ_n) \neq \infty$)  then
        Sort {$T_i$} in ascending order according to $D(T_i , RZ_n)$ in $L3$
         r = 1
         While ((r ≤ size of $L3$ ) and ($Load_m$ > 100)) Do // Browsing {$Ti$ }
            $T_i = L3(r)$
            l = 1
            While (l <= size($Section_i$) and $Load_m$ >100) Do
            // Checking the possibility of relocation of  the sections of
            $T_i$  by respecting the load of $RZ_n$
              select the first execution section $Exe_i$ and discard it from $Section_i$
              $Load_{n,i} = Load_n + Exe_i/P_i + Overhead_n/P_i$
              If ($Load_{n,i}$ <= 100) then   // Migration of $Exe_i$ from $RZ_m$  to $RZ_n$  is accepted
                 $Load_m = Load_m - Exe_i/P_i - Overhead_m/P_i$  // Removing $Exe_i$ from $RZ_m$
                 $Load_n = Load_{n,i}$  // Migration of $Exe_i$ to $RZ_n$
              End If
              l++
            End While
            r++
         End While
      End If
      q++
 End While
 If ($Load_m$ > 100) then
   New $RZ_m$ * ( $\lceil Load_m/100 \rceil$ - 1)  // Adding new $RZ_m$
    Reinitialize the load  of {$RZ_n$}when it does not affect the number of added $RZ_m$
 End If
End For
```

D. Level 3: Two-level Fitting

Level 3 consists of two independent sub-levels. The first one ensures the fitting of RZs on the most suitable non-overlapped RPBs in terms of resource efficiency. The second sub-level performs the mapping of tasks to RZs according to their preemption points by avoiding the RZ overload and by guaranteeing the achievement of task execution. Task mapping is based on run-time partial reconfiguration and promotes solutions of lowest cost D and reducing overheads.

As proved in our work [12], the placement problem is NP-complete problem. Its search space grows exponentially with the number of tasks and RZs. Level 2 and level 3 depict the principle functions of off-line placement of hardware tasks and level 1 is a pre-placement analysis. Placement problem is a combinatory optimization problem, it uses discrete solution set, chooses the best solution out of all possible combinations and aims the optimization of multi-criteria function. Consequently, level 2 and level 3 are formulated as mixed integer programming in the following section as it uses some binary and natural variables.

IV. MIXED INTEGER PROGRAMMING FOR HARDWARE TASK PLACEMENT

Level 2 and level 3 are modeled by the quadruplet (V,X,C,F).

Constants (V)

NT, *NZ*, *NP*
Task features, RZ features, RB features
Device features: *width*, *height*, *Device_RB*

Variables (X)

RPB_j features for each RZ_j
(X_j,Y_j): The coordinates of the upper left vertex of RPB_j
$WRPB_j$: The abscissa of the upper right vertex of RPB_j
$HRPB_j$: The ordinate of the bottom left vertex of RPB_j
Task preemption points
$PreempUnicity_{j,i,l}$: Boolean variable controls whether the mapping of $Preemp_{i,l}$ of T_i is performed on RZ_j
$SumPreemp_{j,i}$: The sum of preemption points of T_i performed within RZ_j is expressed by (9).

$$SumPreemp_{j,i} = \sum_{\substack{PreempUnicity_{j,i,l} \neq 0 \\ 1 \le l \le NbrPreemp_i}} 1 \qquad (9)$$
$$1 \le i \le NT \text{ and } 1 \le j \le NZ$$

$Occupation_{j,i}$: The mapping of preemption points of tasks to RZs produces the occupation rates of tasks T_i in RZ_j which are computed as in (10).

$$Occupation_{j,i} = \begin{cases} PreempDiff_{j,i} + \left(C_i - Preemp_{i,NbrPreempi}\right), & \\ \qquad if\ PreempUnicity_{j,i,NbrPreempi} \neq 0 & \\ PreempDiff_{j,i}, & otherwise \end{cases} \qquad (10)$$

$$PreempDiff_{j,i} = \sum_{\substack{1 \le l < NbrPreemp_i \\ PreempUnicity_{j,i,l} \neq 0}} \left(Preemp_{i,l+1} - Preemp_{i,l}\right)$$

AverageLoad: After preemption point mapping, the average of RZ workloads is calculated by (11).

$$AverageLoad = \frac{\sum_{\substack{1 \le i \le NT \\ 1 \le j \le NZ}} \left(Occupation_{j,i}/P_i + Overhead_j \times SumPreemp_{j,i}/P_i\right)}{NZ} \qquad (11)$$

Constraints (C)

RPB coordinates domain (CP1): the values of RPB coordinates are limited by the width and the height of the device.

$$1 \le X_j, WRPB_j \le width, 1 \le Y_j, HRPB_j \le height \qquad (12)$$

Heterogeneity Constraint (CP2): During RPBs partitioning and RZs fitting, this constraint claims that the number of RBs in RPBs is greater or equal to those in RZs as formulated by (13).

$$\beta_{j,k} \le \sum_{\substack{X_j \le m \le WRPB_j \\ Y_j \le n \le HRPB_j}} \sum_{device\_RB_{[m][n]} = RB_k} 1, \qquad (13)$$
$$1 \le j \le NZ \text{ and } 1 \le k \le NP$$

Non-overlapping between RPBs (CP3): As expressed by (14), this constraint restricts the fitting of RZs on non-overlapped RPBs.

$$X_q > WRPB_j \ or\ X_j > WRPB_q$$
$$or\ Y_q > HRPB_j \ or\ Y_j > HRPB_q, \qquad (14)$$
$$\forall j \neq q, 1 \le j, q \le NZ$$

Non-overload in RZs (CM1): During fitting tasks T_i on RZ_j, each RZ_j must not be overloaded (see (15)).

$$\sum_{1 \le i \le NT} \left(\frac{Occupation_{j,i}}{P_i} + \frac{SumPreemp_{j,i} \times Overhead_j}{P_i}\right) \le 100 \qquad (15)$$

***Infeasibility of mapping for preemption points (CM2)*:** This constraint repeals the mapping of preemption points of tasks to RZ_j giving infinite cost *D* (see (16)).

$$PreempUnicity_{j,i,l} = 0 \ when\ D(T_i, RZ_j) = \infty \qquad (16)$$
$$\forall\ 1 \le i \le NT, 1 \le l \le NbrPreemp_i \text{ and } 1 \le j \le NZ$$

***Uniqueness of preemption points (CM3)*:** As explained by (17), each preemption point of task T_i must be mapped to unique RZ_j.

$$\sum_{1 \le j \le NZ} PreempUnicity_{j,i,l} = 1 \qquad (17)$$
$$\forall\ 1 \le i \le NT \text{ and } 1 \le l \le NbrPreemp_i$$

This constraint guarantees also the achievement of task execution and discards the problem of task rejection.

Minimization Objective Function

During our resolution, we considered two sub-problems: the partitioning of RPBs on the device resolved simultaneously with the fitting of RZs on the selected RPBs and the mapping of tasks to their appropriate RZs. The selection of the best solutions for both sub problems is guided by the following objective function.

$$F = PlaceFunction + MappingFunction$$

PlaceFunction focuses on the sub-problem of fitting RZs on the most suitable RPBs partitioned on the device. By respecting the heterogeneity and the non-overlapping constraints, *PlaceFunction* promotes the fitting of RZs on RPBs that strictly contain the number and type of RBs required by RZs. As expressed by (18), *PlaceFunction* evaluates the resources efficiency of the RZ fitting on the selected RPBs on the heterogeneous device.

$$PlaceFunction = \sum_{\substack{1\le j\le NZ\\1\le k\le NP}} RBCost_k \times \left(\gamma_{j,k} - \beta_{j,k}\right) \tag{18}$$

MappingFunction deals with the fitting of preemption points of hardware tasks on the RZs by respecting the three last constraints and by optimizing the three following criteria of measuring mapping quality.

$$MappingFunction = Map1 + Map2 + Map3$$

By means of (19), *Map*1 targets the full exploitation of RZs by approaching their workloads to 100%. *Map*1 aims also the load balancing of RZs by minimizing the variance of their workloads towards the *AverageLoad*.

$$\begin{aligned} &Map1 = \\ &\sum_{1\le j\le NZ}\left(100 - Load\_RZ_j\right) + \sum_{1\le j\le NZ}\left(Load\_RZ_j - AverageLoad\right)^2/NZ \\ &Load\_RZ_j = \sum_{1\le i\le NT} \frac{Occupation_{j,i}}{P_i} + \frac{SumPreemp_{j,i}\times Overhead_j}{P_i} \end{aligned} \tag{19}$$

In (20), *Map*2 computes the overhead resulting from task mapping. *Map*2 takes into account all the possible preemption points, even the successive ones within the same task, in order to obtain the worst case overhead. In fact, the scheduler could preempt a task on these successive preemption points in the same RZ in favor of a higher priority task. Minimizing *Map*2 promotes the solutions of mapping tasks to RZs providing the lowest overhead.

$$Map2 = \sum_{\substack{1\le i\le NT\\1\le j\le NZ}} SumPreemp_{j,i} \times Overhead_j \tag{20}$$

The goal of *Map*3 expressed by (21) is to map tasks with high occupation rate to the RZs providing the lowest cost *D*. The benefit of *Map*3 is the optimization of resource use since cost *D* considers the weight of each resource in terms of its frequency on the device and the importance of its functionality. As cost *D* reveals the resource wastage when task is mapped to RZ, minimizing *Map*3 ensures the use optimization of costly resources by mapping tasks with low occupation rates to RZs including costly resources.

$$\begin{aligned} Map3 = \sum_{\substack{1\le i\le NT\\1\le j\le NZ}} &\Big(D\left(T_i, RZ_j\right)^2 \times {Occupation_{j,i}}^2/4 \\ &- D\left(T_i, RZ_j\right) \times Occupation_{j,i}\Big) \end{aligned} \tag{21}$$

V. RESULTS AND PLACEMENT QUALITY EVALUATION

A. Proposed Application

The experiments in this section deal with the effect of three-level off-line resource management on an application composed of seven heterogeneous tasks. As shown in Fig. 2, our application contains varied-size tasks with heterogeneous resources which are considered the main functions in the current real-time applications. The application consists of microcontroller (T48) that guides the remainder part of the application and ensures the hardware task configuration as well as the data flow synchronization. MDCT task computes the modified discrete cosine transform which is the main function in JPEG compression. JPEG task performs hardware compression of 24 frames per second by using the data provided by MDCT task. AES (Advanced Encryption Standard) encrypts the resulted information from JPEG task by processing blocs of 128 bits with 256bit-key.VGA task drives VGA monitors and can display one picture on the screen either of chars or color waveforms or color grid. We did not consider the communication latency between the microcontroller and the other hardware tasks and we focused only on finding efficient placement for the hardware tasks.

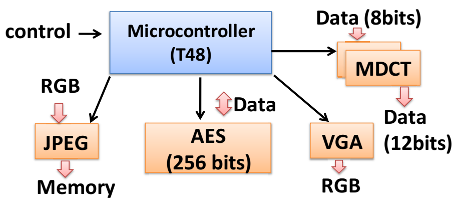


Figure 2. Hardware tasks of the application.

At design time, we synthesized the hardware resources of these hardware tasks by means of ISE 11.3 Xilinx tool and we choose Xilinx Virtex 5 SX50 as reconfigurable hardware device. In Virtex 5 technology [13], there are four main resource types: CLBL, CLBM, BRAM and DSP. By considering the reconfiguration granularity, the RBs in Virtex 5 are vertical stacks composed of the same type of resources: RB_1 (20 CLBMs), RB_2 (20 CLBLs), RB_3 (4 BRAMs) and RB_4 (8 DSPs). We have assigned 20, 80, 192 and 340 as *RBCost* respectively for RB_1, RB_2, RB_3 and RB_4. Virtex 5 FPGA and the hardware tasks are modeled with their RB-models. Configuration overheads are determined by considering that each task defines an RZ. The partial reconfiguration flow dedicated by PlanAhead 11.3 Xilinx tool enables the floorplanning of hardware tasks on the chosen device to create their bitstreams independently for estimating configuration overheads. We rely on parallel 8bit-width configuration port and use 100 MHz as the configuration clock frequency. Preemption points are determined arbitrarily according to the granularity of hardware tasks and their C_i. For all tasks, we consider that the first preemption point is equal to 0 µs. The features of hardware tasks and their instances are presented in TABLE I.

B. Obtained Results

The pre-placement analysis performed by level 1 produce the set of RZ types according to the RBs requirements in hardware tasks. Thus, following to step 1 of level 1, the RB-models of the obtained RZ types are indicated on the RZs line in TABLE I.

TABLE I. FEATURES OF HARDWARE TASKS

	MDCT	AES	VGA	T48	JPEG
Instances	T_1,T_2	T_3	T_4	T_5	T_6
RB-model (µs)	2 RB_1, 12 RB_2, 3 RB_3, 0 RB_4	4 RB_1, 7 RB_2, 1 RB_3, 1 RB_4	2 RB_1, 4 RB_2, 1 RB_3, 0 RB_4	5 RB_1, 4 RB_2, 0 RB_3, 0 RB_4	8 RB_1, 12 RB_2, 0 RB_3, 2 RB_4
WCET (µs)	40552	44540	2500	20000	350000
Period (µs)	416666	200000	10000	50000	416666
Overhead (µs)	3215	1980	681	721	2968
Preemption Points (µs)	10000, 20000, 30000	30000, 40000	1650, 2000	5000, 10000, 15000	200000, 300000
RZs	RZ_1	RZ_2	RZ_1	RZ_3	RZ_4

WCET: Worst Case Execution Time of the task, **Period**: the period of the task which is equal to the deadline, **Overhead**: configuration overhead of the task in Virtex 5 SX50 with parallel 8bit-width port (100 MHz), **Preemption points**: points in time taken from WCET predefined by the designer, **RZs**: the assigned RZ for the task in step 1 of level 1.

In this application, RZ_1 is created by MDCT type (T_1 and T_2) and VGA type (T_4). As explained by step 1, when different types of tasks construct the RZ type, the maximum number of each RB type must be taken between these tasks. In the case that the maximum numbers of distinct RB types are produced by different tasks, the whole partial reconfiguration flow for this RZ type must be performed to recompute its configuration overhead. For RZ_1, the RB-model and configuration overhead are taken form MDCT type as it gives the maximum number of RBs. TABLE II describes the obtained RZ types, their workloads (%) and the costs D between tasks and RZs. The workloads of the obtained RZs are computed by assigning to each RZ the tasks giving lowest cost D with them presented by the bold numbers in TABLE II. An overload in RZ_2 is detected after step 2 of level 1 and is due to the execution times of T_3 and T_4 as well as to the configuration overhead of RZ_2. Step 3 of level 1 resolves this overload in RZ_2 by migrating the first execution section marked by the first preemption point (0 µs) and the second preemption point (1650 µs) of T_4 to RZ_1. RZ_1 is the unique RZ type that could accept T_4 as it is non-overloaded and it gives finite cost D (1024) with it. Step 3 decides this task relocation instead of adding another RZ_2 to resolve efficiently its overload.

TABLE II. STEP 1 AND STEP 2 RESULTS

	MDCT {T_1,T_2}	AES {T_3}	VGA {T_4}	T48 {T_5}	JPEG {T_6}
***RZ_1* (26%)**	**0**	∞	1024	∞	∞
***RZ_2* (110%)**	∞	**0**	**620**	∞	∞
***RZ_3* (46%)**	∞	∞	∞	**0**	∞
***RZ_4* (86%)**	∞	∞	∞	1380	**0**

The resolution of the level 2 and level 3 of our off-line resource management are resolved by means of powerful solvers dedicated by AIMMS environment (www.aimms.com) that relies on the Branch and Bound [14] method which guarantees the optimal solution. We have considered two independent sub-problems. The first one ensures the partitioning of RPBs on the device for all the RZs provided by level 1 combined with the fitting of RZs on the most suitable RPBs by respecting the constraints *CP1*, *CP2* and *CP3* and by optimizing the objective expressed by *PlaceFunction*. This first sub-problem was modeled as mixed integer linear program and was resolved after 3 minutes by CPU of 2 GHz with 2 GB of RAM. Nevertheless, the second sub-problem consists on mapping the tasks to the most appropriate RZs according to their predefined preemption points by satisfying the constraints *CM1*, *CM2* and *CM3* and by promoting the solution that optimizes the objectives expressed by *Map1*, *Map2* and *Map3*. The task mapping sub-problem was formulated as mixed integer non-linear program and was resolved after 1 second.

For the first sub-problem, TABLE III shows the RZ fitting on the selected RPBs defined by their coordinates.

TABLE III. RPBS FOR RZ FITTING

	X_i	$WRPB_i$	Y_i	$HRPB_i$
RPB_1	25	28	1	6
RPB_2	34	45	1	2
RPB_3	1	3	1	4
RPB_4	34	45	3	5

In TABLE IV, the costs (Δ) expressing the differences in RB_k between RZs and their associated RPBs obtained after resolution depict the resource efficiency. The obtained results of RZ fitting provide an averaged resource utilization of 30% of the available RBs in the heterogeneous device. This resource utilization achieves up 27 % of resource gain comparing to static design. The static design is obtained by fitting each instance of each hardware task on its RPB without using the concept of dynamic partial reconfiguration.

TABLE IV. RESOURCE EFFICIENCY

	RB_1	RB_2	RB_3	RB_4	Δ
RPB_1	6	12	6	0	4 RB_1, 3 RB_3
RPB_2	12	8	2	2	8 RB_1, 1 RB_2, 1 RB_3, 1 RB_4
RPB_3	8	4	0	0	2 RB_1
RPB_4	18	12	3	3	10 RB_1, 3 RB_3, 1 RB_4

Figure 3 shows the floorplanning of RPBs in Virtex 5 SX50 according to the obtained RPB coordinates.

The mapping of task preemption points are detailed in TABLE V. $T_{i,x}$ depicts the x-th execution section of T_i. Tasks T_1, T_2, T_3, T_5 and T_6 are efficiently mapped to their optimal RZs by optimizing the objectives expressed by (20) and (21) of reducing overheads and the use of costly resources. For T_4, the analytic resolution assigns more execution sections of this task to its optimal RZ RZ_2 (80%) than RZ_1 (20%). The two first sections of T_4 ($T_{4,1}$,$T_{4,2}$)

marked by the its preemption point (0 µs) and its third preemption point (2000 µs) are mapped to RZ_2. The last execution section of T_4 ($T_{4,3}$) is fitted on RZ_1. On the same RZ, the tasks are scheduled by respecting their deadlines and are preempted on their predefined preemption points. Moreover, we considered that execution sections, delimited by the preemption points within task, are independent. Effectively, there is need neither to exchange data nor to send synchronization resource between these execution sections.

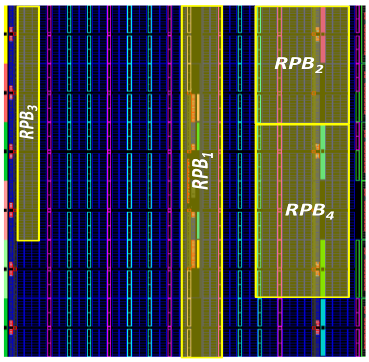


Figure 3. RZ fitting on Virtex 5 SX50.

TABLE V. MAPPING OF PREEMPTION POINTS

RZ₁	$T_{1,1}$,$T_{1,2}$,$T_{1,3}$,$T_{1,4}$	100% of T_1
	$T_{2,1}$,$T_{2,2}$,$T_{2,3}$,$T_{2,4}$	100% of T_2
	$T_{4,3}$	20 % of T_4
RZ₂	$T_{3,1}$,$T_{3,2}$,$T_{3,3}$	100% of T_3
	$T_{4,1}$,$T_{4,2}$	80% of T_4
RZ₃	$T_{5,1}$,$T_{5,2}$,$T_{5,3}$,$T_{5,4}$	100% of T_5
RZ₄	$T_{6,1}$,$T_{6,2}$,$T_{6,3}$	100% of T_6

After mapping of preemption points of tasks to RZs fitted on the reconfigurable device, the analytic resolution produces 50623 µs of total configuration overhead which represents 10 % of total running time. To optimize the objective of load balancing and full exploitation of RZs expressed by (19) the Branch and Bound method converges to an average workload of 70%. The problem of task rejection is discarded as the mapping resolution guarantees execution unit (RZ) for all execution sections of tasks as expressed by *CM3* in (17).

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new three-level resource management targeting the enhancement of placement quality for off-line placement of hardware tasks. By adopting run-time partial reconfiguration and mixed integer programming, we improved the quality of placement in terms of resource efficiency, configuration overhead and the exploitation of RZs. The problem of task rejection is discarded. Future work targets the directed acyclic graphs and involves adding precedence constraints as well as deadline and periodicity constraints to achieve an off-line mapping/scheduling of hardware tasks on the reconfigurable hardware devices.

REFERENCES

[1] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast Template Placement for Reconfigurable Computing Systems,” *IEEE Design and Test, Special Issue on Reconfigurable Computing,* vol. 17, pp. 68–83, January. 2000.

[2] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," *International Parallel and Distributed Processing Symposium (IPDPS'03)*, pp. 178, April 2003.

[3] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A New Approach for On-line Placement on Reconfigurable Devices," *International Parallel and Distributed Processing Symposium (IPDPS'04)*, p. 134, April 2004.

[4] H. ElGindy, M. Middendorf, H. Schmeck, and B.Schmidt, "Task rearrangement on partially reconfigurable FPGAs with restricted buffer," *Field Programmable Logic and Applications*, pp. 379-388, August 2000.

[5] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey," *European Journal of Operational Research*, Vol 141, pp. 241-252, March 2001.

[6] A. Lodi, S. Martello, and D. Vigo, "Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem," *Meta-heuristics : advances and trends in local search paradigms for optimization*, pp. 125-139, July 1997.

[7] A. Lodi, S. Martello, and D. Vigo, "Heuristic and metaheuristic approaches for a class of two-dimenional bin packing problems," *INFORMS journal on computing*, Vol 11, pp. 345-357, 1999.

[8] S.P. Fekete, E. Kohler, and J. Teich "Optimal FPGA module placement with temporal precedence constraints," *Design Automation and Test in Europe*, pp. 658–665, March 2001.

[9] J. Teich, S.P. Fekete, and J. Schepers, "Optimization of dynamic hardware reconfiguration," *The journal of supercomputing*, Vol 19, pp. 57–75, 2001.

[10] K. Danne, S. Stuehmeier, "Off-line placement of tasks onto reconfigurable hardware considering geometrical task variants," *International Federation for Information Processing*, 2005.

[11] I. Belaid, F. Muller, M. Benjemaa, "Off-line placement of hardware tasks on FPGA," *19th International Conference on Field Programmable Logic and Application (FPL'09)*, pp. 591-595, September 2009.

[12] I. Belaid, F. Muller, M. Benjemaa, "Off-line placement of reconfigurable zones and off-line mapping of hardware tasks on FPGA", *Design and Architectures for Signal and Image Processing*, September 2009.

[13] "Virtex-5 FPGA Configuration User Guide," *Xilinx white paper*, August 2009.

[14] J. Clausen, "Branch and Bound algorithms-principles and examples", March 1999.

Exploration of Heterogeneous FPGA Architectures

Umer Farooq, Husain Parvez, Zied Marrakchi and Habib Mehrez
LIP6, Université Pierre et Marie Curie
4, Place Jussieu, 75005 Paris, France
Email: umer.farooq@lip6.fr

***Abstract*—Heterogeneous FPGAs are commonly used in industry and academia due to their superior area, speed and power benefits as compared to their homogeneous counterparts. The layout of these heterogeneous FPGAs are optimized by placing hard-blocks in distinct columns. However, communication between hard-blocks that are placed in different columns require additional routing resources; thus overall FPGA area increases. This problem is further aggravated when the number of different types of hard-blocks, placed in distinct columns, increase in an FPGA. This work compares the effect of different floor-plannings on the area of island-style FPGA architectures. A tree-based architecture is also presented; unlike island-style architectures, the floor-planning of heterogeneous tree-based architectures does not affect its routing requirements. Different FPGA architectures are evaluated for three sets of benchmark circuits, which are categorized according to their inter-block communication trend. The island-style column-based floor-planning is found to be 36%, 23% and 10% larger than a near-ideal non-column-based floor-planning for three sets of benchmarks. Column-based floor-planning is also found to be 18%, 21% and 40% larger than the tree-based FPGA architecture for the same benchmarks.**

I. Introduction

During the recent past, embedded hard-blocks (HBs) in FPGAs (i.e. heterogenous FPGAs) have become increasingly popular due to their ability to implement complex applications more efficiently as compared to homogeneous FPGAs. Previous research [1][2][3][4] has shown that embedded HBs in FPGAs have resulted in significant area and speed improvements. The work in [5] shows that the use of HBs in FPGAs reduces the gap between ASIC and FPGA in terms of area, speed and power consumption. Some of the commercial FPGA vendors like Xilinx [6] and Altera [7] are also using HBs (e.g. multipliers, RAMs and DSP blocks). This trend has resulted in the creation of domain-specific FPGAs. Domain-specific FPGAs are a trade-off between specialization and flexibility. In domain-specific FPGAs, if an application design, that is implemented on an FPGA, uses an embedded hard-block, area, speed and power improvements are achieved. However, if embedded-blocks remain unused, precious logic and routing resources are wasted. On the other hand, a homogeneous FPGA has no such problem but can result in higher area, lower speed and more power consumption for the implementation of same design.

Almost all the work cited above considers island-style FPGAs as the reference architecture where HBs are placed in fixed columns; these columns of HBs are interspersed evenly among columns of configurable logic blocks (CLBs). The main advantage of island-style, column-based heterogeneous FPGA lies in its simple and compact layout generation. When tile-based layout for an FPGA is required, the floor-planning of similar type blocks in a column simplifies the layout generation. The complete width of the entire column, having same type of blocks, can be adjusted appropriately to generate a very compact layout. However, the column-based floor-planning of FPGA architectures limits each column to support only one type of HB. Due to this limitation, the architecture is bound to have at least one separate column for each type of HB even if the application or a group of applications that is being mapped on it uses only one block of that particular type. This can eventually result in the loss of precious logic and routing resources. This loss can become even more severe with the increase in number of types of blocks that are required to be supported by the architecture.

This work generates FPGAs using different floor-planning techniques and then compares them to column-based floor-planning. Mainly six floor-planning techniques are explored, four of which are column-based and two are non column-based. Though, the column-based techniques are advantageous in terms of easy and compact layout generation but this advantage could be overshadowed by the poor resource utilization. On the other hand non-column-based floor-planning can give better resource utilization, but at the expense of a difficult layout generation. Also, compact layout generation is not feasible, because HB dimension need to be the multiple of smallest block (usually a CLB); eventually some area loss.

This work also compares a tree-based heterogenous FPGA architecture [8] with different floor-planning techniques of mesh-based heterogeneous FPGA architecture. Contrary to mesh-based heterogenous FPGA, routability of a tree-based FPGA is independent of its floor-planing and the number of types of HBs required to be supported by the architecture. So, tree-based heterogenous FPGA can be advantageous as compared to mesh-based FPGA. Two different techniques are explored for tree-based FPGA architecture. First technique respects the symmetry of hierarchy, which is one of the characteristics of tree-based architectures. However, in order to provide only the required amount of HBs, second technique does not respect the symmetry of hierarchy.

This paper presents only area results and power and timing comparisons are not considered in this work. The remainder of the paper is organized as follows: section II gives a brief overview of two FPGA architectures. Section III gives a brief overview of exploration environments of two FPGA architectures. Section IV presents the exploration flow. Section V

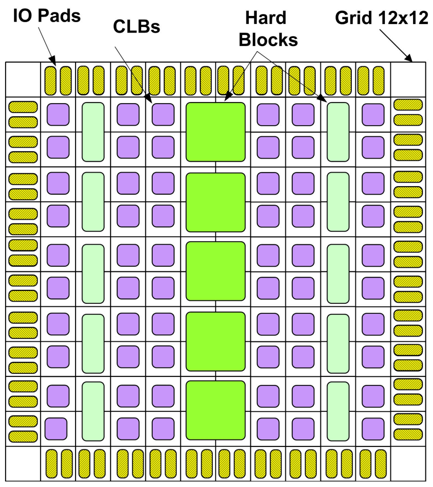


Fig. 1. **Mesh-based Heterogeneous FPGA**

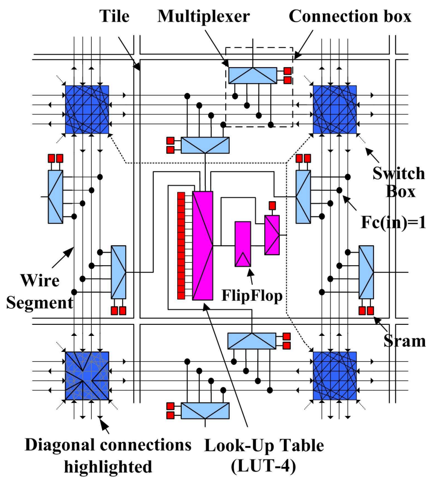


Fig. 2. **Detailed Interconnect of a CLB with its Neighboring Channels**

presents experimental results and section VI finally concludes this paper.

II. Reference FPGA Architectures

This section gives a brief overview of the two heterogeneous FPGA architectures that are used in this work.

A. Mesh-based Heterogenous FPGA Architecture

First of all, mesh-based heterogeneous FPGA architecture is presented. A mesh-based heterogeneous FPGA architecture contains CLBs, I/Os and HBs that are arranged on a two dimensional grid. In order to incorporate HBs in a mesh-based FPGA, the size of HBs is quantized with size of the smallest block of the architecture i.e. CLB. The width and height of an HB is a multiple of the smallest block in the architecture. An example of such FPGA is shown in Figure 1. In mesh-based FPGA, input and output pads are arranged at the periphery of the architecture. The position of different blocks in the architecture depends on the used floor-planning technique. A block (referred as CLB or HB) is surrounded by a uniform length, single driver, unidirectional routing network [9]. The input and output pins of a block connect with the neighboring routing channel. In the case where HBs span multiple tiles, horizontal and vertical routing channels are allowed to pass through them. An FPGA tile showing the detailed connection of a CLB with its neighboring routing network is shown in Figure 2. A unidirectional disjoint switch box connects different routing tracks together. The connectivity of the routing channel with the input and output pins of a block, abbreviated as Fcin and Fcout, is set to be 1. The channel width is varied according to the netlist requirement but remains a multiple of 2 [9].

B. Tree-based Heterogeneous FPGA Architecture

A tree-based architecture is a hierarchical architecture having unidirectional interconnect. A generalized example of a tree-based architecture is shown in Figure 3. A tree-based architecture exploits the locality of connections that is inherent in most of the application designs. In this architecture, CLBs, I/Os and HBs are partitioned into a multilevel clustered structure where each cluster contains sub clusters and switch blocks allow to connect external signals to sub-clusters. Tree-based architecture contains two unidirectional, single length interconnect networks: a downward network and an upward network as shown in Figure 4. Downward network is based on butterfly fat tree topology and allows to connect signals coming from other clusters to its sub-clusters through a switch block. The upward network is based on hierarchy and it allows to connect sub-cluster outputs to other sub-clusters in the same cluster and to clusters in other levels of hierarchy. Figure 3 shows a three-level, arity-4, tree-based architecture. In a heterogenous tree-based architecture, CLBs and I/Os are normally placed at the bottom of hierarchy whereas HBs can be placed at any level of hierarchy to meet the best design fit. For example, in Figure 3 HBs are placed at level 2 of hierarchy. In a tree-based architecture, CLBs and HBs communicate with each other using switch blocks that are further divided into downward and upward mini switch boxes (DMSBs & UMSBs). These DMSBs and UMSBs are unidirectional full cross bars that connect signals coming into the cluster to its sub-clusters and signals going out of a cluster to the other clusters of hierarchy. A tree-based cluster showing the detailed connection of a CLB with its neighboring CLBs is shown in Figure 4. It can be seen from the figure that DMSBs are responsible for downward interconnect and UMSBs are responsible for upward interconnect and they are combined together to form the switch block of a cluster. The number of signals entering into and leaving from the cluster can be varied depending upon the netlist requirement. However they are kept uniform over all the clusters of a level.

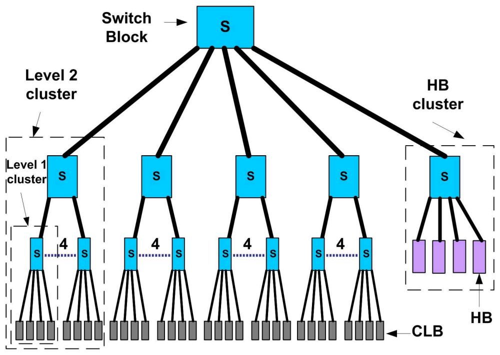


Fig. 3. **Tree-based Heterogeneous FPGA**

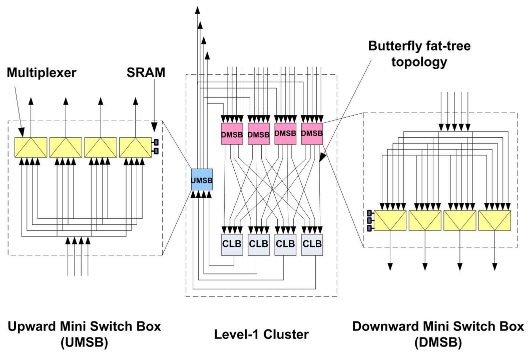

Multiplexer
SRAM
Butterfly fat-tree topology
DMSB
UMSB
CLB
Upward Mini Switch Box (UMSB)
Level-1 Cluster
Downward Mini Switch Box (DMSB)

Fig. 4. **Detailed Interconnect of Level-1 Cluster of Tree-based FPGA**

C. *Characteristics of Mesh-based and Tree-based Heterogeneous FPGAs*

Both tree-based and mesh-based heterogeneous FPGAs have particular characteristics that are mainly dependant on the basic interconnect structure and the arrangement of different blocks in the architecture. For example, the major advantage of a tree-based heterogeneous FPGA is its predictable routing and its independence of the types and position of blocks supported by the architecture. In a tree-based architecture, number of paths required to reach a destination are limited and hence the number of switches crossed by a signal to reach from a source to a destination do not vary greatly. It can be seen from Figure 3 that any CLB can reach a HB by traversing four switches. Unlike tree-based FPGAs, routability of mesh-based FPGA is greatly dependant upon the position of different blocks on the architecture. In mesh-based FPGAs, routability is not predictable and number of paths available to reach a destination are almost unlimited. Hence the number of switches crossed to reach a destination vary with respect to the position of blocks in the architecture. For example, any CLB in the left most column of Figure 1 crosses at least eight switches to reach a HB in the second last column of the architecture. However, this number of switches is reduced to only one if that CLB is placed beside the HB of the second last column of architecture. So, floor-planning plays a very important role in column-based island-style heterogeneous FPGAs. This problem further aggravates for the communication of different HBs placed in different columns.

III. Exploration Environments

In this section, the exploration environments of two FPGA architectures are presented. We also describe different floor-planning techniques that are explored using these exploration environments. Floor-planning techniques can have major implications on the area of a mesh-based FPGA. If a tile-based layout is required for an FPGA, the floor-planning of similar type of blocks in columns can help optimize the tile area of a block. The complete width of a column can be adjusted according to the layout requirements of similar blocks placed in a column. On the other hand, if blocks of different types are placed in a column, the width of a column cannot be fully optimized; as the columns widths can only be reduced to maximum width of any tile in that column. There will remain some unused area in smaller tiles. Such a problem does not arise if a tile-based layout is not required. In such a case, an FPGA hardware netlist can be laid out using an ASIC design flow.

A. *Exploration Environment of Mesh-based FPGA*

This work uses the mesh-based architecture exploration environment presented earlier in [10]. This work further improves this environment by implementing Range Limiter [11] and column-move operation for heterogeneous architectures. An FPGA architecture is initially defined using an architecture description file. BLOCKS of different sizes are defined, and later mapped on a grid of equally sized SLOTS, called as a SLOT-GRID. Each BLOCK occupies one or more SLOTS. The type of the BLOCK and its input and output PINS are used to find the size of a BLOCK. In a LUT-4 based FPGA, a CLB occupies one slot and 18x18 multiplier occupies 4x4 slots. Input and output PINS of a BLOCK are defined, and CLASS numbers are assigned to them. PINS with the same CLASS number are considered equivalent; thus a NET targeting a receiver PIN of a BLOCK can be routed to any of the PINS of the BLOCK belonging to the same CLASS. Once the architecture of FPGA is defined, the benchmark circuit is placed on the architecture.

Placer Operations: A simulated annealing based [12] PLACER is used to perform different operations. The PLACER either **(i)** moves an instance from one BLOCK to another, **(ii)** moves a BLOCK from one SLOT position to another, **(iii)** rotates a BLOCK at its own axis, or **(iv)** moves a complete column of BLOCKS from one SLOT position to another. After each operation the placement cost is recomputed for all the disturbed nets. Depending on the cost value and the annealing temperature the operation is accepted or rejected. Multiple netlists can be placed together to get a single architecture floor-planning for all the netlists. For multiple netlist placement, each BLOCK allows multiple instances to be mapped onto it, but multiple instances of the same netlist can not be mapped on a single block.

Placer performs move and rotate operations on a “source”

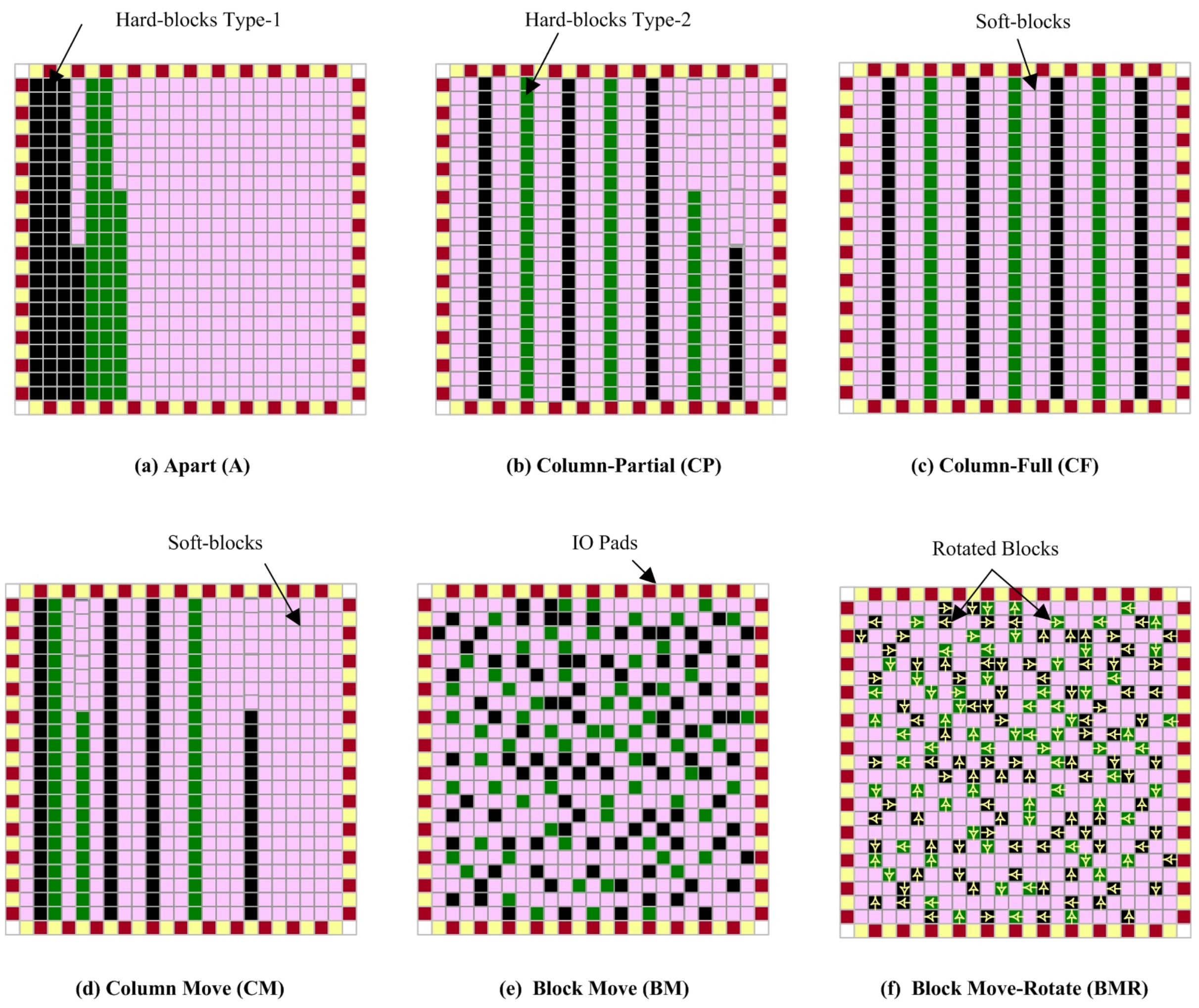


Fig. 5. **Floor-Planning Techniques**

and a "destination" block. When a source is to be moved from one slot position to another, any random SLOT is selected as its destination. The rectangular window starting from this destination slot and having same size and shape as that of source is called destination window whereas the window occupied by the source is called source window. Normally, source contains one block whereas destination window can contain one or more blocks. Once the source and destination windows are selected, the move operation is performed if **(i)** the destination window does not exceed the boundaries of SLOT-GRID, **(ii)** destination window does not contain any block that exceeds the boundary of destination window and **(iii)** destination window does not overlap (partially or fully) source window. However, if these conditions are not met, the procedure continues until a valid destination window is not found. When a block is to be rotated, same source position becomes its destination position and block is rotated around its own axis. The block rotate operation becomes important when pins of a block have different classes. In such a case the size of bounding box varies depending upon the position and direction of the pins. Multiples of $90°$ rotation are performed for square blocks while multiples of $180°$ are performed for rectangular blocks. A $90°$ rotation for rectangular blocks requires both move and rotate operations; it is left for future work.

By using different PLACER operations, six floor-planning technique are explored. The detail of these floor planning techniques is as follows:

Floor-Planning Techniques: **(i)** Hard-blocks are placed in fixed columns, apart from the CLBs as shown in Figure 5 (a). Such kind of floor-planning technique can be beneficial for data path circuits as described by [13]. It can be seen from the figure that if all HBs of a type are placed and still there is space available in the column then in order to avoid wastage of resources, CLBs are placed in the remaining place of column. **(ii)** Columns of HBs are evenly distributed among columns of CLBs, as shown in Figure 5 (b). **(iii)** Columns of HBs are evenly distributed among CLBs. Contrary to first and second techniques, whole column contains only one type of blocks. This technique is normally used in commercial architectures and is shown in 5 (c). **(iv)** The HBs are placed in columns but columns are not fixed, rather they are allowed to move through the column-move operation of PLACER. This technique is shown in Figure 5 (d). **(v)** In this technique HBs are not restricted to remain in columns; and they are allowed to move through block move operation as shown in Figure 5 (e). **(vi)** The blocks are allowed to move and rotate through block move and rotate operations. This floor-planning technique is shown in Figure 5 (f).

B. Exploration Environment of a Tree-based FPGA

A tree-based FPGA is defined using an architecture description file. The architecture description file contains different architectural parameters along with the definition of different BLOCKS used by the architecture. Once the architecture is defined, PARTITIONER partitions the netlist using a top-down recursive partitioning approach. First, top level clusters are constructed, and then each cluster is partitioned into sub-clusters, until the bottom of hierarchy is reached [14]. The main objective of PARTITIONER is to reduce communication between the clusters by absorbing maximum communication inside the clusters. Two simple techniques are explored for tree-based FPGA. **(i)** A generalized example of first technique is shown in Figure 3. This technique is referred as symmetric (SYM). In this technique HBs can be placed at any level which gives the best design fit. However in this technique the symmetry of hierarchy is respected which can eventually results in wastage of HBs. For example in Figure 3, it can be seen that this architecture supports 4 HBs of a certain type (because it is an arity 4 architecture). But, for a netlist requiring only two HBs other two HBs will remain unused and will be wasted. **(ii)** This technique is same as SYM except that in this technique the symmetry of hierarchy for HBs is not respected and only that number of HBs are used that are needed. This technique is referred as asymmetric (ASYM).

IV. Experimental Flow

This section discusses different types of benchmarks, the software flow and the experimental methodology used to explore two FPGA architectures.

A. Benchmark Selection

Generally in academia and industry, the quality of an FPGA architecture is measured by mapping a certain set of benchmarks on it. Thus the selection of benchmarks plays a very important role in the exploration of heterogeneous FPGAs. This work puts special emphasis on the selection of benchmark circuits, as different circuits can give different results for different architecture floor-planning techniques. This work categorizes the benchmark circuits by the trend of communication between different blocks of the benchmark. So, three sets of benchmarks are assembled having distinct trend of inter-block communication. These benchmarks are shown in Tables I, II and III respectively. These benchmarks are obtained from three different sources. For example the benchmarks shown in Table I are the designs developed at Université Pierre et Marie Curie. The benchmarks shown in Table II are obtained from `http://www.opencores.org/`

TABLE I
DSP Benchmarks SET I

Circuit Name	Inputs	Outputs	CLBs (LUT4)	Mult (8x8)	Slansky (16+16)	Sff (8)	Sub (8-8)	Smux (32:16)
FIR	9	16	32	4	3	4	-	-
FFT	48	64	94	4	3	-	6	-
ADAC	18	16	47	-	-	2	-	1
DCU	35	16	34	1	1	4	2	2

TABLE II
Open Core Benchmarks Set II

Circuit Name	No of Inputs	No of Outputs	No of LUTs	No of Multipliers (16x16)	No of Adders (20+20)
cf_fir_3_8_8	42	18	159	4	3
cf_fir_7_16_16	146	35	638	8	14
cfft16x8	20	40	1511	-	26
cordic_p2r	18	32	803	-	43
cordi_r2p	34	40	1328	-	52
fm	9	12	1308	1	19
fm_receiver	10	12	910	1	20
lms	18	16	940	10	11
reed_solomon	138	128	537	16	16

TABLE III
Open Core Benchmarks Set III

Circuit Name	No of Inputs	No of Outputs	No of LUTs	No of Multipliers (18x18)
cf_fir_3_8_8	42	22	214	4
diffeq_f_systemC	66	99	1532	4
diffeq_paj_convert	12	101	738	5
fir_scu	10	27	1366	17
iir1	33	30	632	5
iir	28	15	392	5
rs_decoder_1	13	20	1553	13
rs_decoder_2	21	20	2960	9

and benchmarks shown in Table III are obtained from `http://www.eecg.utoronto.ca/vpr/`. The communication between different blocks of a benchmark can be mainly divided into the following four categories:
CLB-CLB: CLBs communicate with CLBs.
CLB-HB: CLBs communicate with HBs and vice versa.
HB-HB: HBs communicate with other HBs.
IO-CLB/HB: I/O blocks communicate with CLBs and HBs.

In the SET I benchmarks, the major percentage of total communication is between HBs (i.e. HB-HB) and only a small part of total communication is covered by the communication CLB-CLB or CLB-HB. Similarly, in SET II the major percentage of total communication is between HBs and CLBs where either HBs are source and CLBs are destination or vice versa. In SET III, major percentage of total communication is covered by CLB-CLB and only a small part of total communication is covered by CLB-CLB or CLB-HB. Normally the percentage of IO-CLB/HB is a very small part of the total communication for all the three sets of benchmarks.

B. Software Flow

The software flow used to place and route different benchmarks (netlists) on the two heterogeneous FPGAs is shown in Figure 6. The input to the software flow is a VST file

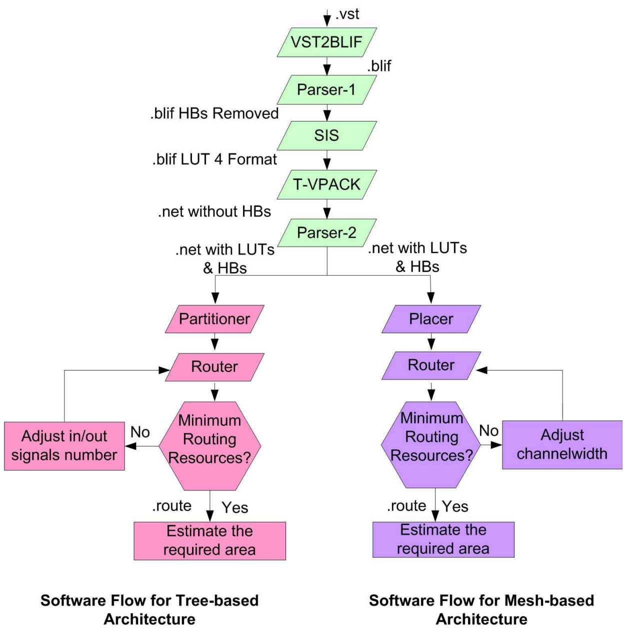

.vst
VST2BLIF
.blif
Parser-1
.blif HBs Removed
SIS
.blif LUT 4 Format
T-VPACK
.net without HBs
Parser-2
.net with LUTs & HBs
.net with LUTs & HBs
Partitioner
Router
Adjust in/out signals number
No
Minimum Routing Resources?
.route
Yes
Estimate the required area
Placer
Router
Minimum Routing Resources?
No
Adjust channelwidth
.route
Yes
Estimate the required area
Software Flow for Tree-based Architecture
Software Flow for Mesh-based Architecture

Fig. 6. **Software Flow**

(structured vhdl). This file is converted into BLIF format [15] using a modified version of VST2BLIF tool. The BLIF file is then passed through PARSER-1 which removes HBs from the file and adds temporary inputs and outputs to the file to preserve the dependance between HBs and rest of the netlist. The output of PARSER-1 is then passed through SIS [16] that synthesizes the blif file into LUT format which is later passed through T-VPACK [17] which packs and converts it into .NET format. Finally the netlist is passed through PARSER-2 that adds previously removed HBs and also removes temporary inputs and outputs. The final netlist in .NET format contains CLBs, HBs and I/O instances that are connected to each other via NETS. Once the netlist is obtained in .NET format, it is placed and routed separately on the two architectures. The benchmarks shown in Table III do not follow this flow and they are obtained directly in the synthesized blif format with HBs. These benchmarks are passed through T-VPACK for conversion into .NET format. Once the conversion to .NET format is done, they follow the same flow as other two sets of benchmarks.

After obtaining the netlists in the .NET format, the netlists are mapped on the FPGA architecture. For tree-based architecture, the netlist is first partitioned using a software module called PARTITIONER. This module partitions CLBs, HBs and I/Os into different clusters in such a way that the inter-cluster communication is minimized. By minimizing inter-cluster communication we obtain a depopulated global interconnect network and hence reduced area. PARTITIONER is based on hMetis [14]; hMetis generates a good solution in a short time because of its multi-phase refinement approach. Once partitioning is done, placement file is generated that contains positions of different blocks on the architecture. This placement file along with netlist file is then passed to another software module called ROUTER. The ROUTER uses a pathfinder algorithm [18]. Pathfinder uses a negotiation based iterative approach to route all the NETS in the netlist. In order to optimize the FPGA architecture, a binary search algorithm is used. This algorithm determines the minimum number of signals required to route a netlist on FPGA. Once the optimization is over, area of the architecture is estimated using an area model which is based on symbolic standard cell library SXLIB [19]. The area of FPGA is estimated by combining areas of CLBs, HBs, multiplexors of downward and upward interconnect, and all associated programming bits.

For mesh-based architecture, the netlist file is passed to a software module called PLACER that uses simulated annealing algorithm [12] [11] to place CLBs, HBs and I/Os on their respective blocks in FPGA. The bounding box (BBX) of a NET is a minimum rectangle that contains the driver instance and all receiving instances of a NET. The PLACER optimizes the sum of half-perimeters of the bounding boxes of all NETS. It moves an instance randomly from one block position to another; the BBX cost is updated. Depending on cost value and annealing temperature, the operation is accepted or rejected. After placement, a software module named ROUTER routes the netlist on the architecture. The router uses a pathfinder algorithm [18] to route the netlist using FPGA routing resources. In order to optimize the FPGA resources, a binary search algorithm similar to the one used for tree-based FPGA is used to determine the smallest channel width required to route a netlist. Once this optimization process is over, area is estimated in the same manner as for tree-based architecture.

C. *Experimental Methodology*

In this work, experiments are performed individually for each netlist. The architecture definition, floor-planning, placement, routing and architecture optimization is performed individually for each netlist. Although, such an approach is not applicable to real FPGAs, as their architecture, floor-planning and routing resources are already defined. In order to make our results more comparable to real FPGAs, we even generated and optimized a single maximum FPGA architecture for each group of netlist. However, it was noticed that the floor-planning and routing resources were mainly decided by the largest netlist in each group. Thus the results for three sets of netlists corresponded roughly to the results for three largest netlists in each of the three sets of netlists. So, to get more realistic results, the architecture and floor-planning is optimized individually for each netlist; later average of all netlists gives more realistic results.

V. Experimental Results

Experiments are performed for six different types of floor-planning techniques of mesh-based FPGA and two techniques of tree-based FPGA (described in section III). Experimental results obtained for three sets of benchmarks are shown in Figures 7, 8 and 9. In these figures, the results for benchmarks 1 to 4, 5 to 13 and 14 to 21 correspond to SET I, SET II and

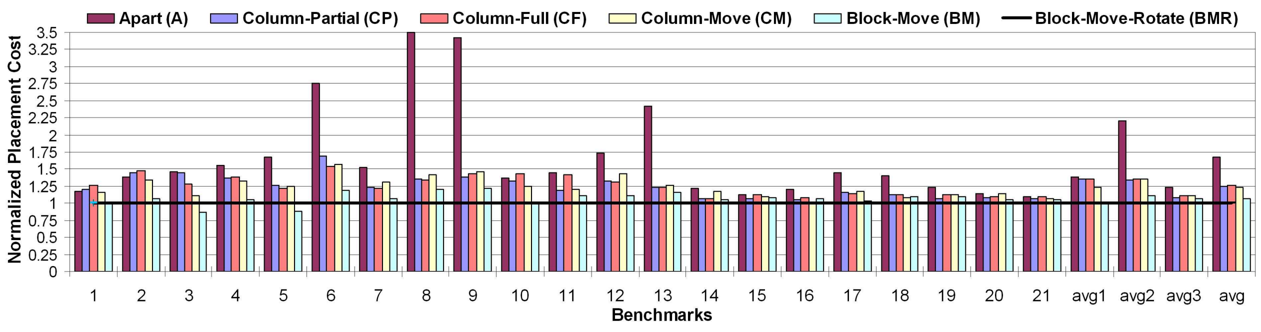


Fig. 7. **Placement Cost Normalized to BMR Floor-Planning**

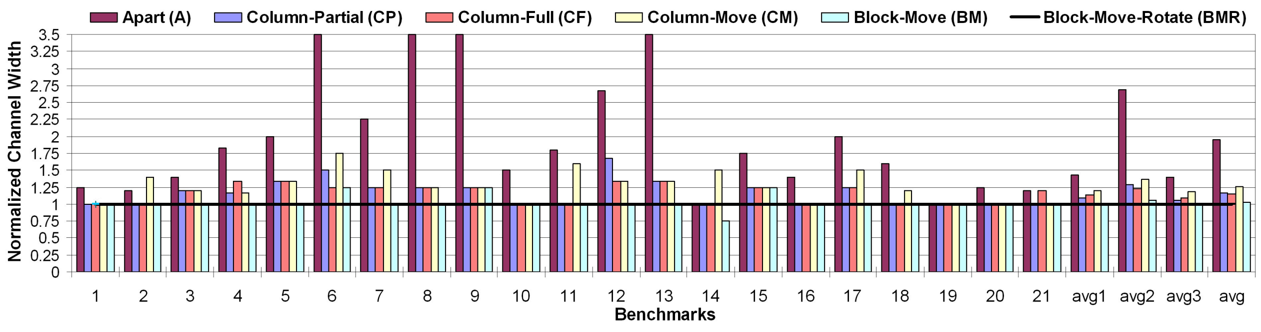


Fig. 8. **Channel Width Normalized to BMR Floor-Planning**

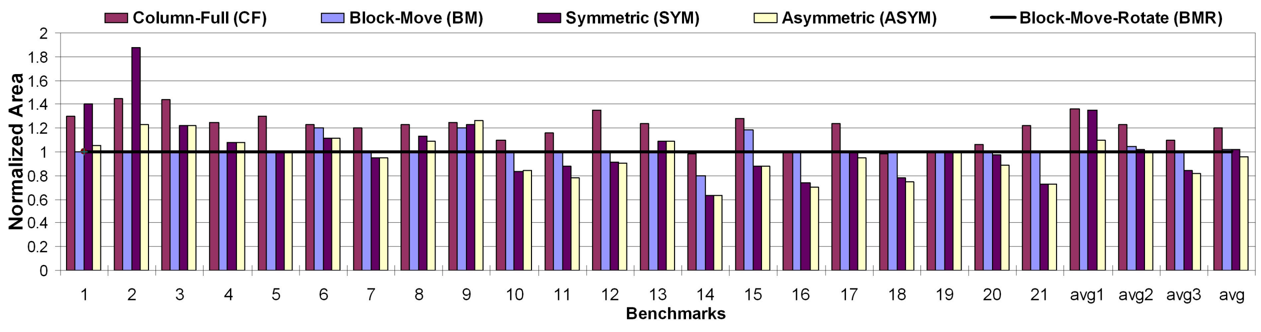


Fig. 9. **Area Results Normalized to BMR Floor-Planning**

SET III respectively. The avg1, avg2 and avg3 in the Figures 7, 8 and 9 correspond to the geometric average of these results for SET I, SET II and SET III respectively. The avg corresponds to the average of all netlists.

Figure 7 shows the placement cost for different floor-planning techniques of mesh-based FPGA, normalized against the placement cost of BMR floor-planning technique. Placement cost is the sum of half perimeters of bounding boxes of all the NETS in a netlist. It can be seen from the figure that, on average, BMR gives equal or better results as compared to other techniques for all three sets of benchmarks. On average, CF gives 35%, 35% and 11% more placement cost than BMR, for SET I, SET II and SET III benchmark circuits respectively. Figure 8 shows channel-width requirements for different floor-planning technique, normalized against BMR. Decrease in placement cost does not always give channel width advantages. However, channel-width gain are achieved in many benchmarks. On average, CF requires 13%, 22% and 9% more channel width than BMR for SET I, SET II and SET III respectively. The increase in channel width increases the overall area of the architecture, as shown in Figure 9. In this figure, the area results of CF, BM floor-planning techniques of mesh-based FPGA and, SYM and ASYM techniques of tree-based FPGA are normalized against the area results of BMR floor-planning technique of mesh-based FPGA. For the sake of clarity, the results for A, CP and CM floor-planning techniques are not presented. On average, CF requires 36%, 23% and 10% more area than BMR for SET I, SET II and SET III respectively. For SET I benchmark circuits, SYM requires 35% more area than BMR, and ASYM requires 10% more area than BMR. However for SET II benchmark circuits, on average BMR is almost equal to SYM and ASYM. For SET III benchmark circuits BMR is worse than SYM and ASYM by 14% and 18% respectively.

The results show that BMR technique produces least placement cost, smallest channel width and hence smallest area for mesh-based heterogeneous FPGA. However, BMR floor-planning technique is dependant upon target netlists to be mapped upon FPGA. Such an approach is not suitable for FPGA production; as floor-planning need to be fixed before mapping application designs on them. Moreover, the hardware layout of BMR might be un-optimized. In this work, the BMR floor-planning serves as a near ideal floor-planning with which other floor-planning techniques are compared. It can also be noted that results of CF compared to BMR vary depending upon the set of benchmarks that are used. For SET I benchmark circuits, where the types of blocks for each benchmark are two or more than two and communication is dominated by HB-HB type of communication, CF produces worse results than the other two sets of benchmarks. This is because columns of different HBs are separated by columns of CLBs and HBs need extra routing resources to communicate with other HBs. However in BMR there is no such limitation; HBs communicating with each other can always be placed close to each other. For other two sets the gap between CF and BMR is relatively less. The reduced HB-HB communication in SET II and SET III benchmark circuits is the major cause of reduction in the gap between CF and BMR. However 23% and 10% area difference for SET II and SET III is due to the placement algorithm. In CF, the simulated annealing placement algorithm is restricted to place hard-block instances of a netlist at predefined positions. This restriction for the placer reduces the quality of placement solution. Decreased placement quality requires more routing resources to route the netlist; thus more area is required.

For tree-based FPGA, ASYM produces best results in terms of area and it is better than the best technique of mesh-based FPGA (i.e. BMR) by an average of 5% for a total of 21 benchmarks. The major advantage of a heterogeneous tree-based FPGA is that the maximum number of switches required to route a connection between CLB-HB or HB-HB remain relatively constant. However, the netlist that contains more HB-HB communication (such as SET I), the constant switch requirement does not mean minimum switch requirement. The architecture floor-planning of tree-based FPGA does not effect the switch requirement of the architecture. However, the floor-planning of mesh-based FPGA causes drastic impact on the switching requirement of the architecture.

VI. Conclusion

This paper has explored two heterogeneous FPGA architectures. Different mesh-based floor-plannings are compared. The floor-plannings of mesh-based FPGA influence the routing network requirement of the architecture. The column-fixed floor-planning is on average 36%, 23% and 10% more area consuming than the block-move-rotate floor-planning for three sets of netlists. The tree-based architecture is independant of its floor-planning, however its layout is relatively less scalable than the mesh architecture. The column-fixed floor-planning is on average 18%, 21% and 40% larger than the tree-based FPGA architectures for the same benchmark. These area gains will decrease due to layout inefficiencies of tree-based architecture. Hardware layout efforts are required to maintain the area benefits on tree-based FPGAs. A mesh containing smaller trees architecture can be designed to resolve scalability issues of tree architecture.

References

[1] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert, “Embedded floating-point units in FPGAs,” in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 2006, pp. 12–20.

[2] C. Ho, P. Leong, W. Luk, S. Wilton, and S. Lopez-Buedo, “Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs,” in *Proc. FCCM*, 2006, pp. 35–44.

[3] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, “A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[4] K. Underwood and K. Hemmert, “Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance,” in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004.*, 2004, pp. 219–228.

[5] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 2006, pp. 21–30.

[6] Xilinx, “Xilinx,” *http://www.xilinx.com*, 2010.

[7] Altera, “Altera,” *http://www.altera.com*, 2010.

[8] Z. Marrakchi, U. Farooq, and H. Mehrez, “Comparison of a tree-based and mesh-based coarse-grained fpga architecture,” in *ICM ’09*, 2009.

[9] G. Lemieux, E. Lee, M. Tom, , and A. Yu, “Directional and single-driver wires in fpga interconnect,” in *IEEE Conference on FPT*, 2004, pp. 41–48.

[10] H. Parvez, Z. Marrakchi, U. Farooq, and H. Mehrez, “A New Coarse-Grained FPGA Architecture Exploration Environment,” in *Field-Programmable Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 285–288.

[11] V. Betz and J. Rose, “VPR: A New Packing Placement and Routing Tool for FPGA research,” *International Workshop on FPGA*, pp. 213–22, 1997.

[12] C. C. Skiścim and B. L. Golden, “Optimization by simulated annealing: A preliminary computational study for the tsp,” in *WSC ’83: Proceedings of the 15th conference on Winter Simulation*. Piscataway, NJ, USA: IEEE Press, 1983, pp. 523–535.

[13] D. Cherepacha and D. Lewis, “DP-FPGA: An FPGA architecture optimized for datapaths,” *VLSI Design*, vol. 4, no. 4, pp. 329–343, 1996.

[14] G.Karypis and V.Kumar, “Multilevel k-way hypergraph partitioning,” 1999.

[15] “Berkeley logic synthesis and verification group,university of california, berkeley. berkeley logic interchange format (blif), http://vlsi.colorado.edu/vis/blif.ps.”

[16] E. M. Sentovich and al, “Sis: A system for sequential circuit analysis,” *Tech. Report No. UCB/ERL M92/41, University of California, Berkeley*, 1992.

[17] A. Marquardt, V. Betz, and J. Rose, “Using cluster based logic blocks and timing-driven packing to improve fpga speed and density,” in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 1999, pp. 39–46.

[18] L. McMurchie and C. Ebeling, “Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs,” in *Proc.FPGA’95*, 1995.

[19] A. Greiner and F. Pecheux, “Alliance: A complete set of cad tools for teaching vlsi design,” *3rd Eurochip Workshop*, 1992.

DYNAMIC ONLINE RECONFIGURATION OF DIGITAL CLOCK MANAGERS ON XILINX VIRTEX-II/VIRTEX-II-PRO FPGAS: A CASE STUDY OF DISTRIBUTED POWER MANAGEMENT

Christian Schuck, Bastian Haetzer, Jürgen Becker

Institut für Technik der Informationsverarbeitung - ITIV
Karlsruher Institut für Technologie (KIT)
Vincenz-Prießnitz-Strasse 1 76131 Karlsruhe

ABSTRACT

Xilinx Virtex-II family FPGAs support an advanced low skew clock distribution network with numerous global clock nets to support high speed mixed frequency designs. Digital Clock Managers in combination with Global Clock Buffers are already in place to generate the desired frequency and to drive the clock networks with different sources respectively. Currently almost all designs run at a fixed clock frequency determined statically during design time. Such systems cannot take the full advantage of partial and dynamic self reconfiguration. Therefore, this paper introduces a new methodology that allows the implemented hardware to dynamically self adopt the clock frequency during runtime by dynamically reconfiguring the Digital Clock Managers. Inspired by nature self adoption is done completely decentralized. Figures for reconfiguration performance and power savings will be given. Further, the tradeoffs for reconfiguration effort using this method will be evaluated. Results show the high potential and importance of the distributed DFS method with little additional overhead.

1. INTRODUCTION

Xilinx Virtex FPGAs have been designed with high performance applications in mind. They feature several dedicated Digital Clock Managers (DCMs) and Digital Clock Buffers for solving high speed clock distribution problems. Multiple clock nets are supported to enable highly heterogeneous mixed frequency designs. Usually all clock frequencies for the single clock nets and the parameters for the DCMs are determined during design time through static timing analysis. Targeting maximum performance these parameters strongly depend on the longest combinatorial path (critical path) between two storage elements of the design unit they are driving. For minimum power the required throughput of the design unit determines the lower boundary of the possible clock frequency. In both cases non adjusted clock frequencies lead to high waste of either processing power or energy [9][11].

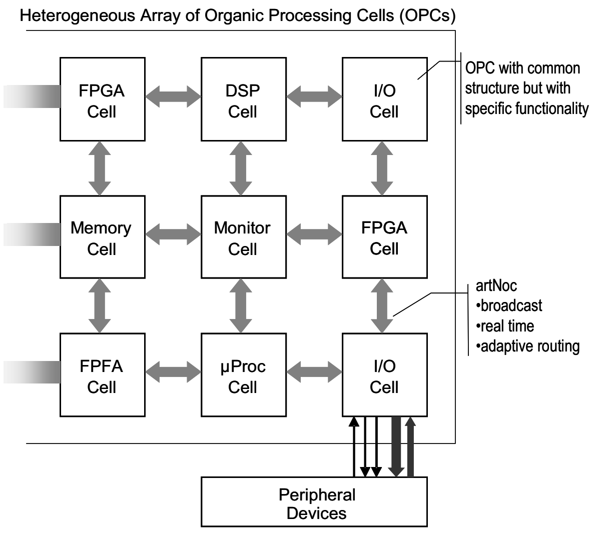


Fig. 1. DodOrg organic hardware architecture

Considering the feature of partial and dynamic self-reconfiguration of Xilinx Virtex FPGAs during runtime a high dynamic and flexibility arises. Static analysis methods are no longer able to sufficiently determine an adjusted clock frequency during design time. At the same time a new partial module is reconfigured onto the FPGA grid, its critical path changes and in turn the clock frequency has to be adjusted as well during runtime to fit the new critical path. On the other side the throughput requirement of the application or the environmental conditions may change over time making an adjustment of clock frequency necessary.

Therefore, a new paradigm of system design is necessary to efficiently utilize the available processing power of future chip generations. To address this issue in [1] the Digital on Demand Computing Organism (DodOrg) was proposed, which is derived from a biological organism. Decentralisation of all system instances is the key feature to reach the desired goals of self-organisation, self-adoption, self-healing in short the self-x features. Hence the hardware

architecture of the DodOrg system consist of many heterogeneous so called Organic Processing Cells (OPCs), that communicate through the artNoC [3] router network as shown in Figure 1. In general, all OPCs are made of the same blueprint. On the one side they contain a highly adaptive heterogeneous data path and on the other side several common units that are responsible for the organic behaviour of the architecture. Among them a Power-Management-Unit is able to perform dynamic frequency scaling (DFS) on OPC level. Therefore, it can control and adjust performance and power consumption of the cell according to the actual computational demands of the application and the critical path of the cells data path. DFS has a high potential, as it decreases the dynamic power consumption by decreasing the switching activity of flip flops, gates in the fan-out of flip flops and the clock trees. Therefore, the cell's clock domain is decoupled by the network interface and can operate independently from the artNoC and the other OPCs of the organic chip.
In [5] we presented a prototype implementation of the DodOrg architecture on a Virtex FPGA, where it is possible to dynamically change the cells data path through 2D-partial and dynamic reconfiguration. Therefore, a novel IP core, the artNoC-ICAP-Interface was developed in order to perform fast 2 dimensional self- reconfiguration and provide a virtual decentralisation of the internal FPGA configuration access port (ICAP). This paper enhances the methodology by enabling the partial and dynamic self-reconfiguration of the Virtex DCMs, which is inherently not given, through the artNoC-ICAP-Interface. Therefore, the desired self adoption with respect to a fine grained power management could be achieved.
The rest of the paper is organized as follows. Section 2 reviews several other proposals for DFS on FPGAs while section 3 summarizes important aspects of the Xilinx Virtex II FPGA clock net infrastructure. Section 4 describes the details of our approach to dynamically reconfigure the DCMs during runtime before section 5 shows reconfiguration performance and power saving figures. Finally, section 6 concludes the work and gives an outlook to future work.

2. RELATED WORK

Recently, several work has been published dealing with power management and especially clock management on FPGAs. All authors agree that there is a high potential for using DFS method in both ASIC and FPGA designs [7] [11].
In [9] the authors show that even because of FPGA process variations and because of changing environmental conditions (hot, normal, cold temperature) dynamically clocking designs can lead to a speed improvement of up to 86% compared to using a fixed, statically estimated clock during design time. The authors use an external programmable clock generator that is controlled by a host PC. However, in order to enable the system to self adapt its clock frequency on chip solutions are required.
In [7] the authors proposed an online solution for clock gating. They propose a feedback multiplexer with control logic in front of the registers. So it is possible to keep the register value and to prevent the combinatorial logic behind the register to toggle. But simultaneously they highlight that clock gating on FPGAs could have a much higher power saving efficiency if it would be possible to completely gate the FPGA clock tree. To overcome this drawback in [8] the authors provide an architectural block that is able to perform DFS. However this approach leads to low speed designs and clock skew problems as it is necessary to insert user logic into the clock network.
We show that on Xilinx Virtex-II no additional user logic is necessary to efficiently and reliably perform a fine grained self adaptive DFS. All advantages of the high speed clock distribution network could be maintained.

3. XILINX CLOCK ARCHITECTURE

This section gives a brief overview over the Xilinx Virtex-II clock architecture as our work makes extensive use of the provided features.

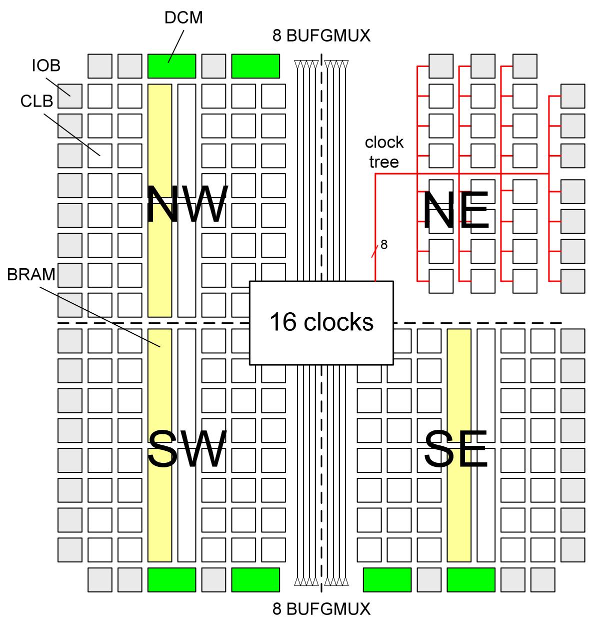


Fig. 2. Xilinx Virtex Clock distribution network

3.1. Clock Network Grid

Besides normal routing resources Xilinx Virtex-II FPGAs have a dedicated low skew high speed clock distribution network [4][6]. They feature 16 global clock buffers (BUFGMUX, see section 3.3) and support up to 16 global clock domains (Figure 2). The FPGA grid is partitioned into 4 quadrants (NW, SE, SW, and SE) with up to 8 clocks per quadrant. Eight clock buffers are in the middle of the top edge and eight are in the middle of the bottom edge. In principle any of these 16 clock buffer outputs can be used in any quadrant as long as opposite clock buffers on the top and on the bottom are not used in the same quadrant, i.e. there is no conflict [6]. In addition, device dependant, up to 12 DCMs are available. They can be used to drive clock buffers with different clock frequencies. In the following important features of the DCMs and clock buffers will be summarized.

3.2. Digital Clock Managers

Besides others, frequency synthesis is an important feature of the DCMs. Therefore, 2 main different programmable outputs are available. CLKDV provides an output frequency that is a fraction (÷1.5, ÷2, ÷2.5… ÷7, ÷7.5, ÷8, ÷9… ÷16) of the input frequency CLKIN.
CLKFX is able to produce an output frequency that is synthesised by combination of a specified integer multiplier M ∊ {2…32} and a specified integer divisor D ∊ {1…32} by calculation CLKFX = M÷D*CLKIN.

3.3. Global Clock Buffer

Global Clock Buffers have three different functionalities. In addition to pure clock distribution, they can also be configured as a global clock buffer with a clock enable (BUFGCE). Then the clock can be stopped at any time at the clock buffer output.
Further clock buffers can be configured to act as a “glitch-free” synchronous 2:1 multiplexer (BUFGMUX). These multiplexers are capable of switching between two clock sources at any time, by using the select input that can be driven by user logic. No particular phase relations between the two clocks are needed. For example as shown in Figure 3 they can be configured to switch between two DCM clock CLKFX outputs.
As we will see in the next section our design makes use of this feature.

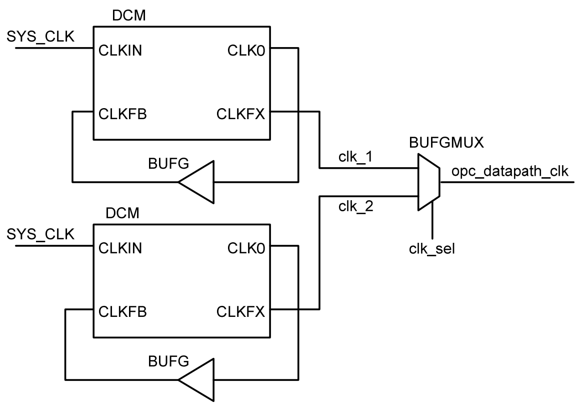


Fig. 3. Example BUFGMUX /DCM configuration

4. ORGANIC SYSTEM ARCHITECTURE

Compared to µC ASIC solutions SRAM based FPGAs like Virtex-II consume a multiple of power. This is due to the fine grained flexibility and adaptability and the involved overhead. By just using these features during design time to create a static design, most of the potential remains unused. Instead dynamic and partial online self-reconfiguration during runtime is a promising approach to exploit the full potential and even to close the energy gap. Therefore, in [5] we proposed to implement the OPC based organic computing organisms on a Virtex-II Pro FPGA as shown in Figure 4.
This paper focuses on the power related issues of the cell based DodOrg architecture on the FPGA prototype. Important aspects to reach the desired goal of a fine grained, decentralized self adaptive power management will be discussed in the subsequent sub-sections.

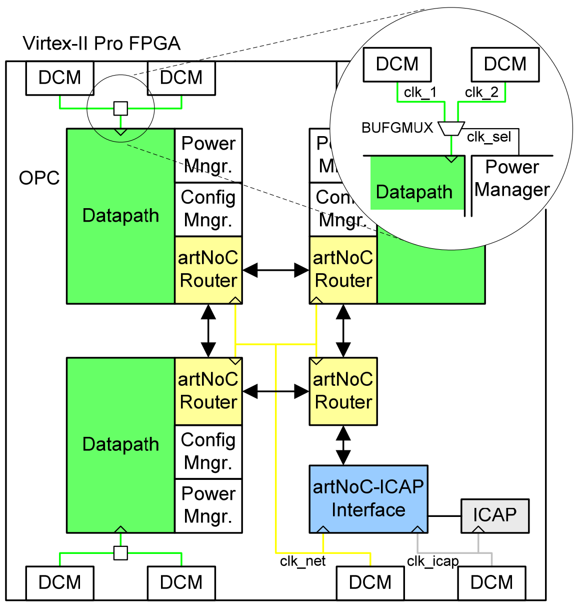


Fig. 4. DodOrg FPGA Floorplan / Clock Architecture

4.1. Clock Partitioning

Depending on the size of the device several OPCs are mapped onto a single FPGA (Figure 4). The clock net of the highly adaptive data path (DP) of every OPC is connected to a BUFGMUX that is driven by a pair of DCMs. Inside every OPC has its own power management unit (PMU) that is connected to the select input of the BUFGMUX. So it can quickly choose between the two DCM clock sources. The DP-clock is decoupled from the artNoC clock by using a dual ported dual clock FIFO buffer. Further the PMU is connected to the artNoC. Thus it is able to exchange power related information with the other PMUs. Beyond that it has access to the artNoC-ICAP-

Interface. Therefore, during runtime every PMU can dynamically adapt the DCM CLKFX output clock frequency through partial self-reconfiguration by using the features of the artNoC-ICAP-Interface.

4.2. artNoC-ICAP-Interface

The artNoC-ICAP-Interface is a small and lightweight IP, which on the one side acts as a wrapper around the native hardware ICAP and on the other side connects to the artNoC network. It provides a virtual decentralisation of the ICAP as well as an abstraction of the physical reconfiguration memory. Its main purpose is to perform the Readback-Modify-Writeback (RMW) method in hardware. Therefore, a fast and true 2 dimensional reconfiguration of all FPGA resources is possible, i.e. reconfiguration is no longer restricted to columns. Due to it's partitioning into two clock domains, one clock domain for the artNoC controller side and one clock domain for the ICAP side, maximal reconfiguration performance could be achieved [5].
As every bit within the reconfiguration memory can be reconfigured independently the configuration of the DCMs can be altered as well. However, a special procedure is necessary that is described in the next paragraph.

4.3. DCM Reconfiguration Details

During reconfiguration of DCMs it is important that a glitchless switching from one clock frequency to another can be guaranteed. In general, after initial setup CLKDV and CLKFX outputs are only enabled when a stable clock is established. After that, the DCM is locked to the configured frequency, as long as the jitters of the input clock CLKIN stays in a given tolerance range [6]. For our scenario we assume that the input clock is stable.
If we change the DCM configuration (D, M) in configuration memory to switch from one clock frequency to a different frequency while the DCM is locked, it loses the lock and no stable output, i.e. no output can be guaranteed. Therefore, to ensure a consistent locking to the new frequency the following steps have to be performed:
Stop the DCM by writing a zero configuration (D= 0, M = 0)
Write the new configuration ($D = D_{new}, M = M_{new}$).
To simplify the handling of the DCM reconfiguration this two step procedure is internally executed by the artNoC-ICAP-Interface. It therefore features a special DCM addressing mode, for an easy access to the DCM configuration. Figure 5 shows a plot of a DCM reconfiguration procedure performed by the artNoC-ICAP-Interface. The plots were recorded by a 4 channel digital oscilloscope with all important signals routed to FPGA pins. Figure 5 a) shows the ICAP enable signal that is asserted by the artNoC-ICAP-Interface during ICAP read

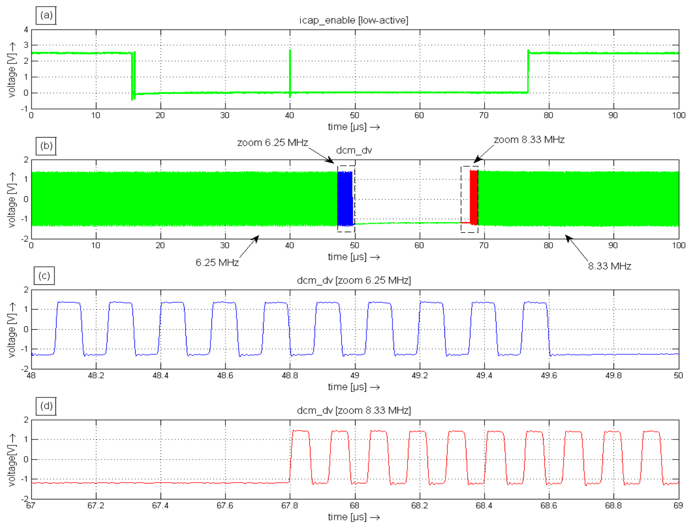


Fig. 5 DCM reconfiguration performance

and write operation. It is an indicator for the overall duration of the reconfiguration procedure. It strongly depends on the device size or rather on the configuration frame length. In this case a Virtex-II Pro XC2VP30 device was used with a frame length of 824 Byte. For reconfiguration of the DCM just a single configuration frame has to be processed. From the beginning of the icap_enable low phase to the spike in Figure 5a) the configuration frame is read back from the configuration memory. Then, the ICAP is configured to write mode and the zero configuration to shut off the DCM is written followed by a dummy frame to flush the ICAP input register. As soon as the writing of the dummy frame is finished the DCM stops. Figure 5 c) shows a zoom of the DCM_CLKFX output (Figure 5 b)) at this point in time. We see that the DCM_CLKFX was running at 6.25 MHz and stops without any jitter or spikes. Immediately after the dummy frame, the read back frame which has been merged with the new DCM parameters is written back to the ICAP followed by a second dummy frame. As soon as the dummy frame is processed the DCM_CLKFX output runs with the new frequency in this case 8.33 MHz. Figure 5 d) shows a zoom of this point in time. Again no glitches or spikes occur. The overall processing time for a complete DCM reconfiguration in this case is 60,7 µsec. In general the reconfiguration time for a different Virtex-II family device is given by:

$$t_{DCM} \sim \text{frame length [Byte]} * 2 / 67{,}7 \text{ [Byte/µs]} + \text{frame\_length [Byte]} * 4 / 90{,}5 \text{ [Byte/µs]}$$

(@ ICAP_CLK = 100 MHz)
The two summands in the formula are resulting from the fact that ICAP has different throughputs for reading and writing reconfiguration data [5].
Therefore, this procedure presents a save method to dynamically reconfigure DCMs during runtime. However, even if self-adaptive decentralized DFS can be realized with the presented method two main drawbacks are obvious:

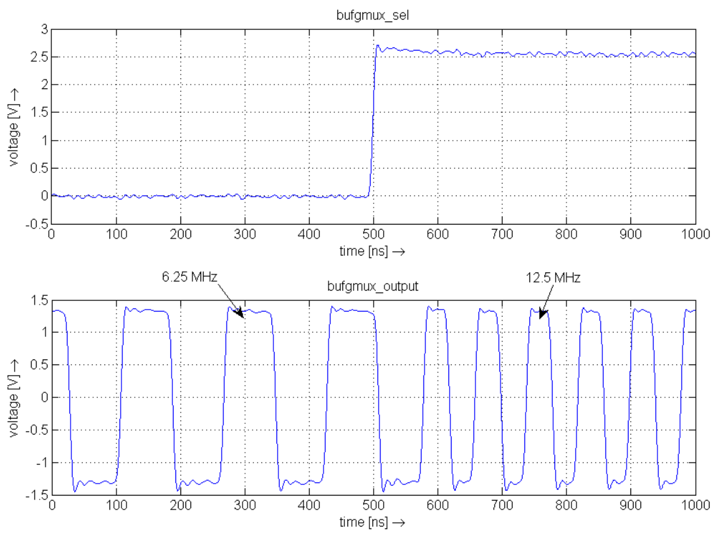


Fig. 6 BUGMUX clock switching

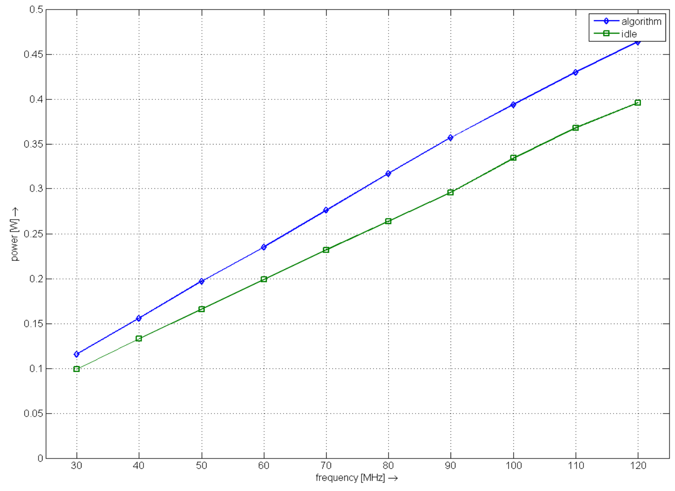
- Relatively long setup delay until the new frequency is valid (in this example: 60,7µs).
- Interruption of clock frequency during reconfiguration (in this example: 18,2µs)

This means that the method is appropriate for reaching long term or intermediate term power management goals, i.e. a new data path is configured and the clock frequency is adapted to its critical path and then stays constant until a new data path is required. But if a frequent and immediate switching is necessary, e.g. when data arrives in burst and between burst the OPC wants to toggle between shut off (f_{DP} = 0Hz) and maximal performance ($f_{DP} = f_{max}$) the method needs to be extended.

In this case a setup consisting of two DCMs and a BUFGMUX, as shown in Figure 3 can be chosen. The select input of the BUFGMUX is connected to the PMU of the OPC. Therefore, it is able to toggle between two frequencies immediately without any delay as shown in Figure 6. Further the interruption of clock frequency during reconfiguration can be hidden. By a combination of both techniques a broad spectrum of different clock frequencies as well as an immediate uninterrupted switching is available.

5. RESULTS

In the preceding section results for reconfiguration times and tradeoffs have already been presented. This section evaluates the potential of power savings and performance enhancements in the context of module based partial online reconfiguration. Especially, the overhead in terms of area and power consumption introduced by the approach (PM, artNoC-ICAP-Interface, DCM) is taken into account.



5.1. Test Setup Power Measurement

We calculated the power consumption by measuring the voltage drop over an external shunt resistor (0.4 Ohm) on the FPGA core voltage (FGPA_VINT). As a test system the Xilinx XUP board with a Virtex-II Pro (XC2VP30) device was used. For all measurements the board source clock of 100 MHz was used as an input clock to the design.

To isolate the portions of power consumption, as shown in Table 1, several distinct designs have been synthesised.

For DCM measurement an array of toggle flip-flops at 100 MHz with and without a DCM in the clock tree have been recorded and the difference of both values has been taken. For extracting ICAP power consumption a system consisting of PM, artNoC-ICAP-Interface and ICAP instance and a second identical system but without ICAP instance have been implemented. After activation the PM sends bursts of two complete alternating configuration

	"Passive" Power [mW]	"Active" Power [mW]
static_offset	-	11
DCM	-	37
PM	<1	<1
artNoC-ICAP-IF	<1	9
ICAP	69	76

Tab. 1 Component Power Consumption

frames targeting the same frame in configuration memory. The ratio of toggling bits between the two frames is 80% and is considered to be representative for a partial reconfiguration. Therefore, before PM activation the "passive" power and after activation the "active" power could be measured. Again the difference in power consumption of the two systems was taken to extract ICAP portion. The other components were measured with the same methodology. Therefore, e.g. all components necessary to implement the approach presented in section

4.3 with two DCMs + BUFGMUX consume 196mW when active, i.e. 180mW when passive. But it has to be considered that artNoC-ICAP-Interface as well as ICAP is also used for partial 2D reconfiguration.

5.2. Area and Resource Utilization

Resource	Number	Percentage
Slices	364	2%
Slice FlipFlops	177	0%
4 input LUTs	664	2%
BRAMs	1	0%
MULT18x18s	2	1%

Tab. 2 Resource Utilization artNoC-ICAP-Interface

The resource requirement for the artNoC-ICAP-Interface with DCM reconfiguration mode is shown in Table 2.

5.3. Power Performance Evaluation

To put the previous power figures into a context we determined the power consumption of a Microblaze soft core processor at different clock frequencies as shown in Figure 7. As we can see there is a high potential for power savings (for example the difference in power consumption in idle state between 100 MHz and 50 MHz is 170mW).
The overhead (ICAP+artNoC-ICAP-IF) for DCM reconfiguration in a static design is in the range of a MB operating at 20 MHz. As expected, we see that there is a linear dependency between clock frequency and power consumption. Therefore, the energy consumed per clock-cycle: $E= P/f_{clk}$; $f_{clk}= c*P$ is constant for all clock frequencies. This means, in terms of power savings for a static data path there is no point for using reconfiguration of DCMs. A setup of DCM_{fmax} and BUFGCE to toggle between $f=f_{max}$ and $f=0$ is most appropriate. In terms of performance, DCM reconfiguration can be used to evaluate maximum clock frequency during runtime.
In turn, in a dynamic scenario, where the data path and therefore also the critical path changes, DCM reconfiguration is necessary to achieve maximum module performance. It also comes without any additional overhead as ICAP + artNoC-ICAP-IF +DCM are already needed for reconfiguration. The capability of DCM reconfiguration together with BUFGMUX provides the basis for fine-grained short or long term power management strategies.

6. SUMMARY AND FUTURE WORK

In this paper we have presented a novel methodology to dynamically reconfigure Digital Clock Managers on Xilinx Virtex-II devices through ICAP. On one side optimal performance of partial modules and on the other side the goal of uniform power consumption can be achieved without external hardware. Our measurements show that power consumed by the components of the proposed hardware framework, especially the DCMs itself, is not negligible and has to be counterweighted. With DCM reconfiguration times in the range of 60µs long term power management goals can be reached. We also provide figures for reconfiguration times as well resource utilization.
Future work is targeting towards the examination of the system level power saving effect resulting from distributed power management with multiple PM and multiple clock domains.

7. REFERENCES

[1] Jürgen Becker et. al. “Digital On-Demand Computing Organism for Real-Time Systems”.In Wolfgang Karl et. al, *Workshop Proceedings of the 19th International Conference on Architecture of Computing Systems(ARCS’06)*

[2] U.Brinkschulte, M.Pacher and A. von Renteln. „An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware”. Springer, 2007

[3] C. Schuck, S. Lamparth and J. Becker “artNoC- A novel Multi-Functional Router Architecture for Organic Computing” International Conference on Field Programmable Logic and Applications 2007, FPL 2007

[4] Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet; DS083 (v4.7) November 5, 2007

[5] C. Schuck; B. Haetzer, and J. Becker „An interface for a dezentralized 2D-reconfiguration on Xilinx Virtex-FPGAs for organic computing” Proc. Reconfigurable Communication-centric SoCs, 2008. ReCoSoC 2008, ISBN: 978-84-691-3603-4

[6] Xilinx Virtex-II Pro and Virtex-II Pro X FPGA User Guide, UG012 (V4.2) 5 November 2007

[7] Y. Zhang, J. Roivainen, and A. Mämmelä „Clock-Gating in FPGAs: A Novel and Comparative Evaluation” Proc. 9th Euromicro Conference on Digital System Design (DSD’06)

[8] I. Brynjolfson, and Z. Zilic “Dynamic Clock Management for Low Power Applications in FPGAs” Proc. Custom Integrated Circuits Conference 2000

[9] J.A. Bower, W. Luk, O. Mencer, M.J. Flynn, M. Morf “Dynamic clock-frequencies for FPGAs” Microprocessors and Microsystems, Volume 30, Issue 6, 4 September 2006, Pages 388-397, Special Issue on FPGA’s

[10] B. Fechner “Dynamic delay-fault injection for reconfigurable hardware” Proc. 19th International Parallel and Distributed Processing Symposium, 2005. IPDPS 2005,

[11] I. Brynjolfson, and Z. Zilic “FPGA Clock Management for Low Power” Proc. International Symposium on FPGAs, 2000

[12] M. Huebner, C. Schuck, and J. Becker “Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs” 20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, Volume , Issue , 25-29 April 2006.

Practical Resource Constraints for Online Synthesis

Stefan Döbrich
Chair for Embedded Systems
University of Technology
Dresden, Germany
stefan.doebrich@inf.tu-dresden.de

Christian Hochberger
Chair for Embedded Systems
University of Technology
Dresden, Germany
christian.hochberger@inf.tu-dresden.de

***Abstract*—Future chip technologies will change the way we deal with hardware design. First of all, logic resources will be available in vast amount. Furthermore, engineering specialized designs for particular applications will no longer be the general approach as the non recurring expenses will grow tremendously. Reconfigurable logic in the form of FPGAs and CGRAs has often been promoted as a solution to these problems. We believe that online synthesis that takes place during the execution of an application is one way to broaden the applicability of reconfigurable architectures as no expert knowledge of synthesis and technologies is required. In this paper we show that even a moderate amount of reconfigurable resources is sufficient to speed up applications considerably.**

***Index Terms*—online synthesis, adaptive computing, reconfigurable architecture, CGRA, AMIDAR**

I. Introduction

Following the road of Moore's law, the number of transistors on a chip doubles every 24 months. After being valid for more than 40 years, the end of Moore's law has been forecast many times, but technological advances have kept the progress intact.

Further shrinking of the feature size of traditionally manufactured chips will lead to exponentially increased mask costs. This makes it prohibitively expensive to produce small quantities of chips for a particular design. Also, the question comes up, how to make use of the vast amounts of resources without building individual chip designs for each application.

Reconfigurable logic in different granularities has been proposed to solve both problems [16]. It allows us to build large quantities of chips and yet use them individually. Field programmable gate arrays (FPGAs) are in use for this purpose for more than two decades. Yet, it requires much expert knowledge to implement applications or part of them on an FPGA. Also, reconfiguring FPGAs takes a lot of time due to the large amount of configuration information.

Coarse Grain Reconfigurable Arrays (CGRAs) try to solve this last problem by working on word level instead of bit level. The amount of configuration information is dramatically reduced and also the programming of such architectures can be considered more *software style*. The problem with CGRAs is typically the tool situation. Currently available tools require an adaptation of the source code and typically have very high runtime so that they need to be run by experts and only for very few selected applications.

Our approach tries to make the reconfigurable resources available for all applications in the embedded systems domain. Thus, synthesis of accelerating circuits takes place during the applications execution. No hand crafted adaptation of the source code shall be required, although it is clear that manual fine-tuning of the code can lead to better results.

In this contribution we want to show that even a relatively small amount of reconfigurable resources is sufficient for a substantial application acceleration.

The remainder of this paper is organized as follows. In section II we will give an overview of related work. In section III we will present the model of our processor which allows an easy integration of synthesized functional units at runtime. In section IV we will detail how we figure out the performance sensitive parts of the application by means of profiling. Section V explains our online synthesis approach. Results for some benchmark applications are presented in section VI. Finally, we give a short conclusion and an outlook onto future work.

II. Related Work

Reconfigurable logic for application improvement has been used for more than two decades. A speedup of 1000 and more could be achieved consistently during this period. Examples range from the CEPRA-1X for cellular automata simulation [10] to implementations of the BLAST algorithm [15]. Unfortunately, these speedups require highly specialized HW architectures and domain specific modelling languages.

A survey of the early approaches using reconfigurable logic for application speed up can be found in [3].

Static transformation from high level languages like C into fine grain reconfigurable logic is still the research focus of a number of academic and commercial research groups. Only very few of them support the full programming language [11].

Efficient static transformation from high level languages into CGRAs is also investigated by several groups. The DRESC [14] tool chain targeting the ADRES [13][17] architecture is one of the most advanced tools. Yet, it requires hand written annotations to the source code and in some cases even some hand crafted rewriting of the source code. Also, the compilation times easily get into the range of days.

The RISPP architecture [1] lies between static and dynamic approaches. Here, a set of candidate instructions are evaluated at compile time. These candidates are implemented dynamically at runtime by varying sets of so called atoms. Thus, alternative design points are chosen depending on the actual execution characteristics.

Dynamic transformation from software to hardware has been investigated already by other researchers. Warp processors dynamically transform assembly instruction sequences into fine grain reconfigurable logic[12]. Furthermore, dynamic synthesis of Java bytecode has been evaluated [2]. Nonetheless, this approach is only capable of synthesizing combinational hardware.

The token distribution principle of AMIDAR processors has some similarities with Transport Triggered Architectures[4]. Yet, in TTAs an application is transformed directly into a set of tokens. This leads to a very high memory overhead and makes an analysis of the executed code extremely difficult.

III. The AMIDAR processing model

In this section, we will give an overview of the AMIDAR processor model. We describe the basic principles of operation, which includes the architecture of an AMIDAR processor in general, as well as specifics of its components. Furthermore, we discuss the applicability of the AMIDAR model to different instruction sets. Finally, several mechanisms of the model, that allow the processor to adapt to the requirements of a given application at runtime are shown.

A. Overview

An AMIDAR processor consists of three main parts. A set of functional units, a token distribution network and a communication structure. Two functional units, which are common to all AMIDAR implementations are the code memory and the token generator. As its name tells, the code memory holds the applications code. The token generator controls the other components of the processor by means of tokens. Therefore, it translates each instruction into a set of tokens, which is distributed to the functional units over the token distribution network. The tokens tell the functional units what to do with input data and where to send the results. Specific AMIDAR implementations may allow the combination of the code memory and the token generator as a single functional unit. This would allow the utilization of side effects like instruction folding. Functional units can have a very wide range of meanings: ALUs, register files, data memory, etc. Data is passed between functional units over the communication structure. This data can have various meanings: instructions, address information, or application data. Figure 1 sketches the abstract structure of an AMIDAR processor.

B. Principle of Operation

Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor explicit pipelining are used to execute instructions. Instead, instructions are broken down to a set of tokens which are distributed to a set of functional units. These tokens are 5-tuples, where a token is defined as $T = \{UID, OP, TAG, DP, INC\}$. It carries the information about the type of operation (OP) that shall be executed by the functional unit with the specified id (UID). Furthermore, the version information of the input data (TAG) that shall be

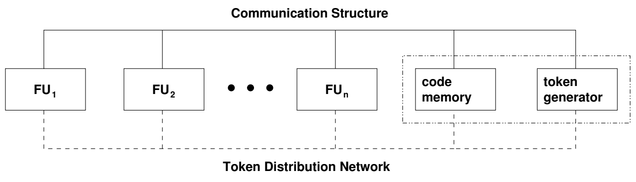


Fig. 1. Abstract Model of an AMIDAR Processor

processed and the destination port of the result (DP) are part of the token. Finally, every token contains a tag increment flag (INC). By default, the result of an operation is tagged equally to the input data. In case the TAG-flag is set, the output tag is increased by one. The token generator can be built such that every functional unit which shall receive a token is able to receive it in one clock cycle. A functional unit begins the execution of a specific token as soon as the data ports received the data with the corresponding tag. Tokens which do not require input data can be executed immediately. Once the appropriately tagged data is available, the operation starts. Upon completion of an operation the result is sent to the destination that was denoted in the token. An instruction is completed, when all of its tokens are executed. To keep the processor executing instructions, one of the tokens must trigger the sending of a new instruction to the token generator. A more detailed explanation of the model can be found in [7].

C. Applicability

In order to apply the presented model to an instruction set, a composition of microinstructions has to be defined for each instruction. Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimal.

The great advantage of this model is that the execution of an instruction depends on the token sequence, and not on the timing of the functional units. Hence, functional units can be replaced at runtime with other versions of different characterizations. The same holds for the communication structure, which can be adapted to the requirements of the running application. Intermediate virtual assembly languages like Java bytecode, LLVM bitcode or the .NET common intermediate language are good candidates for instruction sets.

The structure of a minimum implementation of an AMIDAR based Java processor is sketched in figure 2. Firstly, it contains the mandatory functional units token generator and code memory. In case of a Java machine, the code memory holds all class files and interfaces, as well as their corresponding constant pools and attributes. Additionally, the processor contains several memory functional units. These units realize the operand stack, the object heap, the local variable memory and the method stack. In order to process arithmetic operations, the processor shall contain at least one ALU functional unit. Nonetheless, it is possible to separate integer and floating point operations into two disjoint functional units, which improves the throughput. Furthermore, the processor contains

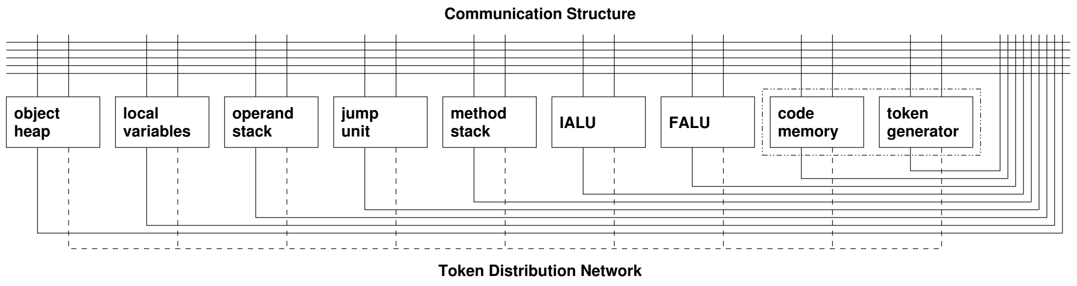


Fig. 2. Model of a Java (non)Virtual Machine on AMIDAR Basis

a jump unit which processes all conditional jumps. Therefore, the condition is evaluated, and the resulting jump offset is transfered to the code memory.

The range of functional unit implementations and communication structures is especially wide if the instruction set has a very high abstraction level and/or basic operations are sufficiently complex. Finally, the data driven approach makes it possible to easily integrate new functional units and create new instructions to use these functional units.

D. Adaptivity in the AMIDAR Model

The AMIDAR model exposes different types of adaptivity. All adaptive operations covered by the model are intended to dynamically respond to the running applications behavior. Therefore, we identified adaptive operations that adapt the communication structure to the actual interaction scheme between functional units. As a functional unit may be the bottleneck of the processor, we included similar adaptive operations for functional units. The following subsections will give an overview of the adaptive operations provided by the AMIDAR model. Most of the currently available reconfigurable devices do not fully support the described adaptive operations (e.g. addition or removal of bus structures). Yet, the model itself contains these possibilities, and so may benefit from future hardware designs. In previous work [7] we have given a detailed overview of these adaptive operations, so this paper provides a short overview only.

E. Adaptive Communication Structures

The communication structure can minimize the bus conflicts that occur during the data transports between functional units. In order to react to the communication characteristics of any given application, functional units may be connected/disconnected to/from a bus structure. This can happen as part of an evasion to another bus structure in case of congestion, as well as the creation of a completely new interconnection. Furthermore, bus structures may be split or folded with the objective of a more effective communication. In [9] we have shown how to identify the conflicting bus taps and we have also shown a heuristics to modify the bus structure to minimize the conflicts.

F. Adaptive Functional Units

Three different categories of adaptive operations may be applied to functional units. Firstly, variations of a specific functional unit regarding chip size, latency or throughput may be available. The most appropriate implementation is chosen dynamically at runtime and may change throughout the lifetime of the application. Secondly, the number of instances of a specific functional unit may be increased or decreased dynamically. This is an alternative way to respond to changing throughput requirements. Finally, dynamically synthesized functional units may be added to the processors datapath. It is possible to identify heavily utilized instruction sequences of an application at runtime (see section IV). Suitable code sequences can be transformed into functional units by means of online synthesis. These functional units would replace the software execution of the related code.

G. Synthesizing Functional Units in AMIDAR

AMIDAR processors need to include some reconfigurable fabric in order to allow the dynamic synthesis and inclusion of functional units. Since fine grained logic (like FPGAs) requires large amount of configuration data to be computed and also since the fine grain structure is neither required nor helpful for the implementation of most code sequences, we focus on CGRAs for the inclusion into AMIDAR processors.

The model includes many features to support the integration of synthesized functional units into the running application. It allows bulk data transfers from and to data memories, as well as the synchronization of the token generator with operations that take multiple clock cycles. Finally, synthesized functional units are able to inject tokens in order to influence the data transport required for the computation of a code sequence.

IV. Runtime Application Profiling

A major task in synthesizing hardware functional units for AMIDAR processors is runtime application profiling. This allows the identification of candidate instruction sequences for hardware acceleration. Plausible candidates are the runtime critical parts of the current application.

In previous work [8] we have shown a profiling algorithm and corresponding hardware implementation which generates detailed information about every executed loop structure. Those profiles contain the total number of executed instructions inside the affected loop, the loops start program counter, its end program counter and the total number of executions of this loop. The profiling circuitry is also capable to profile nested loops, not only simple ones.

A profiled loop structure becomes a synthesis candidate in case its number of executed instructions surmounts a given threshold. The size of this threshold can be configured dynamically for each application.

Furthermore, an instruction sequence has to match specific constraints in order to be synthesized. Currently, we are not capable of synthesizing code sequences containing the following instruction types, as our synthesis algorithm has not evolved to this point yet.

- memory allocation operations
- exception handling
- thread synchronization
- some special instructions, e.g. `lookupswitch`
- access operations to multi-dimensional arrays
- method invocation operations

From this group only access to multi-dimensional arrays and method invocations are important from performance aspect.

Multi-dimensional arrays do actually occur in compute kernels. Access operations on these arrays are possible in principle in the AMIDAR model. Yet, multi-dimensional arrays are organized as arrays of arrays in Java. Thus, access operations need to be broken down into a set of stages (one for each dimension), which is not yet supported by our synthesis algorithm. Nevertheless, a manual rewrite of the code is possible to map multi-dimensional arrays to one dimension.

Similarly, method inlining can be used to enable the synthesis of code sequences that contain method invocations. Techniques for the method inlining are known from JIT compilers that preserve the polymorphism of the called method. Yet, these techniques require the abortion of the execution of the HW under some conditions, which is not yet supported by our synthesis algorithm.

V. Online Synthesis of Application Specific Functional Units

The captured data of the profiling unit is evaluated periodically. In case an instruction sequence exceeds the given runtime threshold the synthesis is triggered, and runs as a low priority process concurrently to the application. Thus, it only occurs if spare computing time remains in the system, and also cannot interfere with the running application.

A. Synthesis Algorithm

An overview of the synthesis steps is given in figure 3. The parts of the figure drawn in grey are not yet implemented.

Firstly, an instruction graph of the given sequence is created. In case an unsupported instruction is detected the synthesis is aborted. Furthermore, a marker of a previously synthesized functional unit may be found. If this is the case it is necessary to restore the original instruction information and then proceed with the synthesis. This may happen if an inner loop has been mapped to hardware before, and then the wrapping loop shall be synthesized as well.

Afterwards, all nodes of the graph are scanned for their number of predecessors. In case a node has more than one predecessor it is necessary to introduce specific Φ-nodes to the graph. These structures occur at the entry of loops or in typical if-else-structures. Furthermore, the graph is annotated with branching information. This will allow the identification of the actually executed branch and the selection of the valid data when merging two or more branches by multiplexers. For if-else-structures, this approach reflects a speculative execution of the alternative branches. The condition of the if-statement is used to control the selection of one set of result values. Loop entry points are treated differently, as no overlapping or software pipelining of loop kernels is employed.

In the next step the graph is annotated with a virtual stack. This stack does not contain specific data, but the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the predecessor of an instruction may not be the producer of its input.

Afterwards an analysis of access operations to local variables, arrays and objects takes place. This aims at loading data into the functional unit and storing it back to its appropriate memory after its execution. Therefore, a list of data that has to be loaded and a list of data that has to be stored is created.

The next step transforms the instruction graph into a hardware circuit. This representation fits precisely into our simulation. All arithmetic or logic operations are transformed into their abstract hardware equivalent. The introduced Φ-nodes are transfered to multiplexer structures. The annotated branching information helps to connect the different branches correctly and to determine the appropriate control signal. Furthermore, registers and memory structures are introduced. Registers hold values at the beginning and the end of branches in order to synchronize different branches. Localization of memory accesses is an important measure to improve the performance of potential applications. In general, SFUs could also access the heap to read or write array elements, but this access would incur an overhead of several clocks. The memory structures are connected to the consumer/producer components of their corresponding arrays or objects. A datapath equivalent to the instruction sequence is the result of this step.

Execution of consecutive loop kernels is strictly separated. Thus, all variables and object fields altered in the loop kernel are stored in registers at the beginning of each loop iteration.

Arrays and objects may be accessed from different branches that are executed in parallel. Thus, it is necessary to synchronize access to the affected memory regions. Furthermore, only valid results may be stored into arrays or objects. This is realized by special enable signals for all write operations. The access synchronization is realized through a controller synthesis. This step takes the created datapath and all information about timing and dependency of array and object access operations as input. The synthesis algorithm has a generic interface which allows to work with different scheduling algorithms. Currently we have implemented a modified ASAP scheduling which can handle resource constraints and list scheduling. The result of this step is a finite state machine (FSM) which controls the datapath and synchronizes all array and object access operations. Also the FSM takes care of the

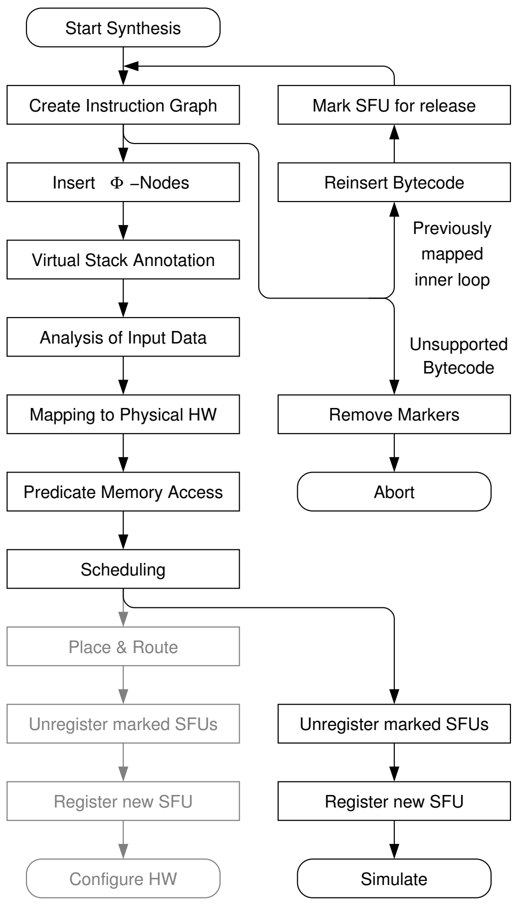


Fig. 3. Overview of synthesis steps

appropriate execution of simple and nested loops.

As mentioned above, we do not have a full hardware implementation yet. Thus, placement and routing for the CGRA are not required. We use a cycle accurate simulation of the abstract datapath created in the previous steps.

In case the synthesis has been successful, the new functional unit needs to be integrated into the processor. If marker instructions of previously synthesized FUs were found, the original instruction sequence has to be restored. Furthermore, the affected SFUs have to be unregistered from the processor and the hardware used by them has to be released.

B. *Functional Unit Integration*

The integration of the synthesized functional unit (SFU) into the running application consist of three major steps. (1) a token set has to be generated which allows the token generator to use the SFU. (2) the SFU has to be integrated into the existing circuit and (3) the synthesized code sequence has to be patched in order to access the SFU.

The token set consist of three parts: (1) the tokens that transport input data to the SFU. These tokens are sent to the appropriate data sources (e.g. object heap). (2) the tokens that control the operation of the SFU, i.e. that start the operation (which happens once the input data is available) and emit the results. (3) the token set that stores the results of the SFU operation in the corresponding memory.

In a next step it is necessary to make the SFU accessible to the other processor components. This requires to register it in the bus arbiter and to update the token generator with the computed token sets. The token set will be triggered by a reserved bytecode instruction.

Finally, the original bytecode sequence has to be replaced by the reserved bytecode instruction. To allow multiple SFUs to co-exist, the reserved bytecode carries the ID of the targeted SFU. Patching of the bytecode sequence is done in such a way that the token generator can continue the execution at the first instruction after the transformed bytecode sequence. Also, it must be possible to restore the original sequence in case a embracing loop nesting level shall be synthesized.

Now, the sequence is not processed in software anymore but by a hardware SFU. Thus, it is necessary to adjust the profiling data of the affected code sequence.

In [5], we have given further information and a more detailed description of the integration process.

VI. Evaluation

In previous research [6] we have evaluated the potential speedup of a simplistic online synthesis with unlimited ressources. To be more realistic, int this work we assume a CGRA with a limited number of processing elements, and a single shared memory for all arrays and objects. The scheduling itself has been calculated by a longest path list scheduling. The following data-set was gained for every benchmark:

- its runtime, and therewith the gained speedup
- the number of states of the controlling state machine
- the number of different contexts regarding the CGRA
- the number of complex operations in those contexts

The reference value for all measurements is the plain software execution of the benchmarks. Note: The mean execution time of a bytecode in our processor is 3-4 clock cycles. This is in the same order as JIT-compiled code on IA32 machines.

A. *Benchmark Applications*

We have chosen applications of four different domains as benchmarks, in order to test our synthesis algorithm.

The first group contains the cryptographic block ciphers Rijndael, Twofish, Serpent and RC6. We evaluated the round key generation out of a 256 bit master key, as well as the encryption of a 16 byte data block.

Another typical group of algorithms used in the security domain are hash algorithms and message digests. We chose the Message Digest 5 (MD5), and two versions of the Secure Hash Algorithm (SHA-1 and SHA-256) as representatives, and evaluated the processing of sixteen 32bit words.

Thirdly, we chose the Sobel convolution operator, a grayscale filter and a contrast filter as representatives of image processing kernels. These three filters operate on a dedicated pixel of an image, or on a pixel and its neighbours. Thus, we measured the appliance of these filters onto a single pixel.

TABLE I
RUNTIME ACCELERATION OF BENCHMARK APPLICATIONS

	Configuration	Rijndael		Twofish		RC6		Serpent	
Round Key Generation		Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
	plain software	17760	-	525276	-	61723	-	44276	-
	4 operators	4602	3.86	43224	12.15	3725	16.57	6335	6.99
	8 operators	4284	4.15	35130	14.95	3459	17.84	6245	7.09
	12 operators	4337	4.09	34280	15.32	3459	17.84	6230	7.11
	16 operators	4337	4.09	34112	15.40	3459	17.84	6230	7.11

	Configuration	Rijndael		Twofish		RC6		Serpent	
Single Block Encryption		Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
	plain software	21389	-	12864	-	17371	-	34855	-
	4 operators	6230	3.43	8506	1.51	2852	6.09	3278	10.63
	8 operators	6181	3.46	8452	1.52	2810	6.18	3273	10.65
	12 operators	6167	3.47	8452	1.52	2768	6.28	3273	10.65
	16 operators	6167	3.47	8452	1.52	2768	6.28	3273	10.65

	Configuration	SHA-1		SHA-256		MD5	
Hash & Digest Algorithms		Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
	plain software	23948	-	47471	-	11986	-
	4 operators	4561	5.25	3619	13.12	1485	8.07
	8 operators	4561	5.25	3484	13.63	1485	8.07
	12 operators	4561	5.25	3484	13.63	1485	8.07
	16 operators	4561	5.25	3484	13.63	1485	8.07

	Configuration	Sobel Filter		Grayscale Filter		Contrast Filter	
Image Processing		Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
	plain software	6930	-	236	-	608	-
	4 operators	1110	6.24	59	4.00	90	6.76
	8 operators	1110	6.24	59	4.00	90	6.76
	12 operators	1110	6.24	59	4.00	90	6.76
	16 operators	1110	6.24	59	4.00	90	6.76

	Configuration	JPEG–Encoder		Color Space Transformation		2-D Forward DCT		Quantization	
JPEG Encoding		Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
	plain software	17368663	-	3436078	-	23054	-	7454	-
	4 operators	4737944	3.67	323805	10.61	2743	8.40	1816	4.10
	8 operators	4645468	3.74	292889	11.73	2572	8.96	1816	4.10
	12 operators	4620290	3.76	277431	12.39	2545	9.06	1816	4.10
	16 operators	4612561	3.77	269702	12.74	2545	9.06	1816	4.10

Finally, we evaluated a complete application and encoded a given 160x48x24 bitmap into a JPEG image. The computation kernels of this application are the color space transformation, 2-D forward DCT and quantization. We did not downsample the chroma parts of the image.

B. Runtime Acceleration

Except from the contrast and grayscale filter, all applications contained either method invocations or access to multidimensional arrays. As we mentioned above, the synthesis does not support these instruction types yet. In order to show the potential of our algorithm we inlined the affected methods and flattened the multidimensional arrays to one dimension.

The subsequent evaluations have shown sophisticated results. Speedups between 3.5 and 12.5 were achieved for most kernels. The encryption of the Twofish cipher is an outlier, being caused by a large communication overhead. Analogous, several applications, e.g. SHA-256, gained better results originating from a benefiting communication/computation ratio. The JPEG encoding application as a whole has gained a speedup of 3.77, which fits into the overall picture. The runtime results for all benchmarks are shown in table I.

C. Schedule Complexity

In a next step, we evaluated the complexity of the controlling units that were created by the synthesis. Therefore we measured the size of the finite state machines, that are controlling every synthesized functional unit. Every state is related to a specific configuration of the reconfigurable array. In the worst case, all of those contexts would be different. Thus, the size of a controlling state machine is the upper bound for the number of different contexts.

Afterwards, we created a configuration profile for every context, which reflects every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of remaining elements is a lower bound for the number of contexts that are necessary to drive the functional unit. The effective number of necessary configurations lies between those two bounds, as it depends on the place-and-route results of the affected operations.

The context informations for the benchmarks are presented in table II. It shows the size of the controlling finite state machine (States), and the number of actually different contexts (Contexts) for every of our benchmarks. It shows, that only three of eighteen state machines on an array with 16 processing

TABLE II
COMPLEXITY OF THE SCHEDULES OF BENCHMARK APPLICATIONS

Round Key Generation	Configuration	Rijndael		Twofish		RC6		Serpent	
		States	Contexts	States	Contexts	States	Contexts	States	Contexts
	4 operators	57	42	230	110	48	21	124	37
	8 operators	55	31	148	113	44	20	106	43
	12 operators	55	31	130	91	44	20	103	42
	16 operators	55	31	122	83	44	20	103	42

Single Block Encryption	Configuration	Rijndael		Twofish		RC6		Serpent	
		States	Contexts	States	Contexts	States	Contexts	States	Contexts
	4 operators	78	37	46	33	25	17	153	54
	8 operators	71	31	40	26	23	20	152	54
	12 operators	69	26	40	22	23	19	152	54
	16 operators	69	23	40	20	23	19	152	54

Hash & Digest Algorithms	Configuration	SHA-1		SHA-256		MD5	
		States	Contexts	States	Contexts	States	Contexts
	4 operators	138	29	107	28	531	20
	8 operators	138	29	92	31	531	20
	12 operators	138	29	92	31	531	20
	16 operators	138	29	92	30	531	20

Image Processing	Configuration	Sobel Filter		Grayscale Filter		Contrast Filter	
		States	Contexts	States	Contexts	States	Contexts
	4 operators	17	13	13	9	56	18
	8 operators	17	13	13	9	56	18
	12 operators	17	13	13	9	56	18
	16 operators	17	13	13	9	56	18

JPEG Encoding	Configuration	JPEG–Encoder		Color Space Transformation		2-D Forward DCT		Quantization	
		States	Contexts	States	Contexts	States	Contexts	States	Contexts
	4 operators	132	64	22	17	89	43	16	11
	8 operators	109	61	18	15	70	42	16	11
	12 operators	110	60	17	15	67	39	16	11
	16 operators	105	55	17	14	67	36	16	11

elements consist of more than 128 states. Furthermore, the bigger part of the state machines contains a significant number of identical states regarding the executed operations. Thus, the actual number of contexts is well below the number of states.

D. Resource Utilization

Another characteristic of the synthesized control units, is the distribution of multi-cycle operations like multiplication or division (complex operations) within the created contexts.

Table III shows the aggregate distribution of complex operations within the schedules. It shows a total number of 1887 contexts for all of our benchmarks, as we scheduled them for a reconfigurable array with four operators. Furthermore, it can be seen that a large set of 1265 contexts did not contain any complex operation. Furthermore, the bigger part of the remaining contexts utilized only one or two complex operations, which sums up to 1725 contexts utilizing two or less complex operations. Hence, only 165 contexts used more than two complex operators.

Entirely, it can be seen, that the 1-quantile covers more than 84% of all contexts, regardless of the reconfigurable arrays size. Furthermore, the 2-quantile contains more than 91% of the contexts. Thus, it is reasonable to reduce the complexity of the reconfigurable array, as a full-fledged homogeneous array structure may not be necessary. Hence, the chipsize of the array would shrink. Nonetheless, this would also decrease the gained speedup. The following subsection shows the influence of such a limitation on the runtime and speedup, with the help of small modifications to the constraints of our measurements.

E. Exemplified Resource Limitation

The results in the preceding subsections suggest the use of a heterogeneous array, as more than 90% of the contexts that were created by our synthesis algorithm used two or less complex operators. This array would provide a full-fledged functionality on a small number of processing elements. All other operators could be cut down to combinational functions. We extended our implemented scheduling algorithms to show the influence of such a limitation, by confining the number of complex processing elements inside the array.

The Twofish benchmark, the 2-D forward DCT of the JPEG encoding, and the JPEG encoding itself have been chosen to show the influence of the described resource limitations regarding a reconfigurable array with 16 processing elements, as they utilized the largest numbers of complex operators.

Limiting the number of full-fledged processing elements to four did not result in a noteworthy drop of the speedup. However, a further confinement to two complex operators inside the array delivered noteworthy results. The achieved speedups dropped 4.5% (JPEG-Encoding), 12.1% (2-D forward DCT) and 10.7% (Twofish round key generation). The results of the resource limited benchmarks are shown in table IV.

TABLE III
OVERALL UTILIZATION OF COMPLEX PROCESSING ELEMENTS IN SYNTHESIZED FUNCTIONAL UNITS

Configuration	Contexts	0		≤ 1		≤ 2		> 2	
4 operators	1887	1265	67%	1591	84%	1725	91%	162	9%
8 operators	1722	1206	70%	1471	85%	1563	91%	159	9%
12 operators	1699	1211	71%	1474	87%	1557	92%	142	8%
16 operators	1684	1219	72%	1476	88%	1556	92%	130	8%

TABLE IV
INFLUENCE OF RESOURCE CONSTRAINTS ON SPEEDUP OF JPEG-ENCODING AND SELECTED APPLICATION KERNELS

Complex Operators	JPEG-Encoding		2-D Forward DCT		Twofish Round Key Generation	
	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
without synthesis	17368663	-	23054	-	525276	-
unrestricted	4612561	3.77	2545	9.06	34112	15.40
4 operators	4694575	3.70	2644	8.72	34726	15.13
2 operators	4819664	3.60	2897	7.96	38230	13.74

VII. CONCLUSION

In this article we have shown a online-synthesis algorithm for AMIDAR processors. The displayed approach targets maximum simplicity and runtime efficiency of all used algorithms.

It is capable of synthesizing functional units fully automated at runtime regarding given resource constraints. The target technology for our algorithm is a coarse grain reconfigurable array. Initially, we assumed a reconfigurable fabric with homogeneously formed processing elements and one single shared memory for all objects and arrays. Furthermore, we used list scheduling as scheduling algorithm.

We evaluated our algorithm by examining four groups of benchmark applications. On average across all benchmarks, a speedup of 7.78 was achieved.

Comparing the runtime of the benchmarks, regarding the underlying reconfigurable fabrics size, shows notably results. An array of eight processing elements delivers the maximum speedup for most benchmarks. The improvements gained through the use of a larger array are negligible. Thus, the saturation of the speedup was achieved with a surprisingly moderate hardware effort.

Furthermore, we displayed the complexity of the synthesized finite state machines. This evaluation showed, that most of our benchmarks could be driven by less than 128 states, and that more than 90% of these corresponding contexts contained two or less complex operations.

Hence, we constrained the number of complex processing elements inside our array, to show the influence of such a limitation onto the speedup. A limit of four complex operations per context nearly did not affect the speedup, while a limit of two complex operations decreased the speedup of the evaluated benchmarks by approximately 5% - 12%.

VIII. FUTURE WORK

As the full potential of our synthesis algorithm has not been reached, future work will concentrate on improving it in multiple ways. This contains the implementation of access to multidimensional arrays and inlining of invoked methods at synthesis time. Additionally, we will explore the effects of instruction chaining in synthesized functional units. Furthermore, we are planning to overlap the transfer of data to a synthesized functional unit and its execution. Also, we will introduce an abstract description layer to our synthesis. This will allow easier optimization of the algorithm itself and will open up the synthesis for a larger number of instruction sets.

REFERENCES

[1] L. Bauer, M. Shafique, S. Kramer, and J. Henkel. RISPP: Rotating instruction set processing platform. In *DAC*, pages 791–796, 2007.

[2] A. C. S. Beck and L. Carro. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *DAC*, pages 732–737, 2005.

[3] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[4] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[5] S. Döbrich and C. Hochberger. Towards dynamic software/hardware transformation in AMIDAR processors. *it - Information Technology*, pages 311–316, 2008.

[6] S. Döbrich and C. Hochberger. Effects of simplistic online synthesis in AMIDAR processors. In *ReConFig*, pages 433–438, 2009.

[7] S. Gatzka and C. Hochberger. A new general model for adaptive processors. In *ERSA*, pages 52–62, 2004.

[8] S. Gatzka and C. Hochberger. Hardware based online profiling in AMIDAR processors. In *IPDPS*, page 144b, 2005.

[9] S. Gatzka and C. Hochberger. The organic features of the AMIDAR class of processors. In *ARCS*, pages 154–166, 2005.

[10] C. Hochberger, R. Hoffmann, K.-P. Völkmann, and S. Waldschmidt. The cellular processor architecture CEPRA-1X and its conguration by CDL. In *IPDPS*, pages 898–905, 2000.

[11] A. Koch and N. Kasprzyk. High-level-language compilation for reconfigurable computers. In *ReCoSoC*, pages 1–8, 2005.

[12] R. L. Lysecky and F. Vahid. Design and implementation of a microblaze-based WARP processor. *ACM Trans. Embedded Comput. Syst.*, 8(3):1–22, 2009.

[13] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *FPL*, pages 61–70, 2003.

[14] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE*, pages 10296–10301, 2003.

[15] E. Sotiriades and A. Dollas. A general reconfigurable architecture for the BLAST algorithm. *J. VLSI Signal Process. Syst.*, 48(3):189–208, 2007.

[16] S. Vassiliadis and D. Soudris, editors. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007.

[17] K. Wu, A. Kanstein, J. Madsen, and M. Berekovic. MT-ADRES: Multithreading on coarse-grained reconfigurable architecture. In *ARC*, pages 26–38, 2007.

ISRC: a runtime system for heterogeneous reconfigurable architectures

Florian Thoma, Jürgen Becker
Institute for Information Processing Technology
Karlsruhe Institute of Technology
Germany
Email: {florian.thoma, juergen.becker}@kit.edu

***Abstract*—By definition, runtime systems bridge the gap between the application and the operation system layer on a processor based hardware. Novel paradigms like reconfigurable computing and heterogeneous multi-core system on chips require additional services provided by the runtime system in comparison to the traditional approaches which are well established in single- or homogeneous multi-core systems. Especially the different characteristics of the target architectures provided in a multi-core System-on-Chip can be exploited for a power and performance efficient task execution. Furthermore, the algorithm for application task scheduling for that kind of hardware architecture has to consider varying timing of the task realizations on the different hardware and additionally, dynamic effects if runtime reconfiguration is used for loading functional blocks on the hardware modules on demand. The Intelligent Services for Reconfigurable Computing (ISRC) approach shows the feasibility to handle the complex system environment of a heterogeneous hardware architecture and presents first results with the MORPHEUS chip, consisting of a processor, coarse-, medium- and fine-grained reconfigurable hardware and a complex communication infrastructure.**

I. Introduction

During the last years the field of Reconfigurable Computing (RC) has expanded and the architectures have gone beyond purely FPGA based systems. At the same time the usage scenarios have extended to other application domains which have higher requirements in terms of adaptivity to user input or other environmental influences. This dynamic system behavior impedes the static partition, allocation and scheduling of the application during design time. Hence the necessity to make as many decisions as possible at runtime. Partitioning and retargeting to different engines are not feasible during runtime but an alternative is to provide multiple implementations for each partition at design time. Scheduling and allocation of these alternatives can then be done by a runtime system depending on system state.

In this paper we present our intelligent services for reconfigurable computing (ISRC) as a implementation of this approach. The subsequent section II places this work in relation to current work in the field of reconfigurable computing. The following section III provides an overview of the MORPHEUS SoC architecture. The corresponding MORPHEUS toolchain is presented in section IV. The main part of the paper which describes the runtime system is formed by section V. An exemplary usage scenario is described in section VI. The conclusion of this and an outlook on future work are given in section VII.

II. Related Work

Reconfigurable Computing is an active and dynamic field of research. Systems like Berkeley Emulation Engine 2 (BEE2) [1] and Erlangen Slot Machine (ESM) [2] are examples of homogeneous partial reconfigurable systems using FPGAs. MORPHEUS [3] is the first truly heterogeneous reconfigurable computing architecture by exploiting the strengths of several different reconfigurable architectures within a Configurable-System-on-Chip (CSoC). During the last years it has become apparent that the use of RC-specific operating systems is required for efficient utilization of the inherent computing power by application designers [4], [5], [6]. Hthreads [7], [8] covers only FPGA targets without use of reconfiguration whereas its successor [9] is focused on heterogeneous many-core processors. In the field of Heterogeneous Computing [10], [11], [12] there has been work done on scheduling but since these machines are typically still processor-based it does not take reconfiguration overhead which is inherent in RC into account. There has been work [13] on scheduling between μP and FPGA but it focuses on the area optimization on the FPGA side. ReconOS [14] shows a way to integrate hardware tasks into a real-time operating system but does not handle scheduling and reconfiguration of tasks as the underlying hardware model is static.

III. System Architecture

The main components of the MORPHEUS System-on-Chip (SoC) are three different heterogeneous reconfigurable engines (HREs), which enable high flexibility for application design and an ARM9 embedded RISC processor, which is responsible for triggering data, control and configuration transfers between all resources in the system [15]. Resources are memory units, IO peripherals, and several HREs each residing in its own clock domain with a programmable clock frequency (see figure 1). All system modules are interconnected via multilayer AMBA buses and/or a Network-on-Chip (NoC). All data transfers between HREs, on- and off-chip memories may be either DNA (Direct Network Access) triggered, DMA triggered or managed directly by the ARM.

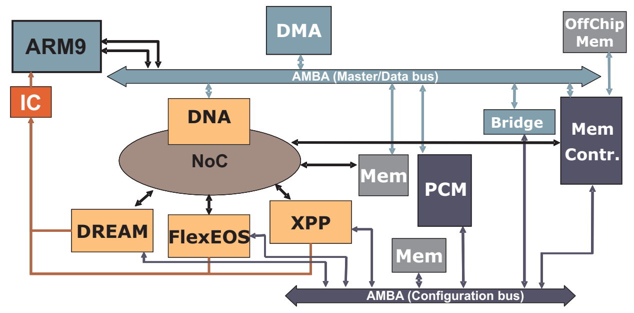


Fig. 1. MORPHEUS architecture

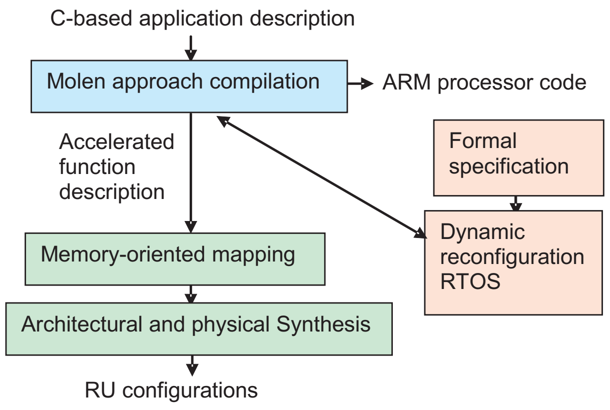

C-based application description
Molen approach compilation
ARM processor code
Accelerated function description
Formal specification
Dynamic reconfiguration RTOS
Memory-oriented mapping
Architectural and physical Synthesis
RU configurations

Fig. 2. Integrated design flow

The DREAM core is a medium-grained reconfigurable array consisting of 4-bit oriented ALUs, where up to four configurations may be kept concurrently in shadow registers. This component is mostly targeting instruction level parallelism, which can be automatically extracted from a C-subset language called Griffy-C.

The FlexEOS is a lookup table based fine-grain reconfigurable device – also known as embedded Field Programmable Gate Array (eFPGA). As any FPGA, it is capable to map arbitrary logic up to a certain complexity if the register and memory resources are matching the specifics of the implemented logic.

XPP-III is a data processing architecture based on a hierarchical array of coarse-grained, adaptive computing elements called Processing Array Elements (PAEs) and a packet-oriented communication network. An XPP-III core contains a rectangular array of ALU-PAEs and RAM-PAEs for data-flow processing. Crossbars tightly couple the array with a column of Function-PAEs (FNC-PAEs) for control-flow oriented code. More regular streaming algorithms like filters or transforms are efficiently implemented on the data flow part of the XPP-III array. Flow graphs of arbitrary shape can be directly mapped to ALUs and routing connections, resulting in a parallel, pipelined implementation. Events enable also conditional operation and loops. One of the strengths of the XPP array originates from fast dynamic runtime reconfiguration.

The integration of the ST NoC with the various reconfigurable components requires an innovative interconnect infrastructure [16]. Processing cores and storage units are connected through the spidergon topology [17] that promises to deliver optimal cost/performance trade-off for multi core designs. In this proprietary topology, the IP blocks are arranged in a sort of ring where each is connected to its clockwise and its counter-clockwise neighbours as in a polygonal structure. In addition, each IP block is also connected directly to its diagonal counterpart in the network, which allows the routing algorithm to minimize the number of nodes that a data packet has to traverse before reaching its destination.

The MORPHEUS SoC features three levels of memory. Each HRE has several dual-clock Data Exchange Buffers (DEB) for local storage and communication with the NoC. The buffers can be accessed as normal RAM or as FIFO. The second level is the on-chip SRAM used by the ARM core and the HREs. The third level is provided by external SRAM.

The Predictive Configuration Manager (PCM) [18] manages the reconfiguration overhead. By caching and prefetching configurations, it minimizes the reconfiguration latencies and offers a unified interface for heterogeneous engines as well as high-level services for the runtime system (see section V).

IV. TOOLCHAIN

The objectives of the Morpheus toolchain are to satisfy embedded computing requirements within the context of Reconfigurable Computing architecture solutions. These objectives are portability (avoiding platform adherence), computing performance efficiency, flexibility and application programming productivity. The toolchain (figure 2) combines the large set of following technologies that are necessary to obtain an integrated design flow from high level application programming (such as C language and graphical interfaces) to hardware and software implementation: compilation, real-time operating system (RTOS), data parallel reorganization, architectural and physical synthesis, formal specification. The toolchain entry point for the application programmer follows an enhanced version of the Molen paradigm [19]. According to this paradigm, the programmer describes its application in a classical C-language program and just annotates the functions identified as preferably implemented on a HRE. The compiler then generates code for the host processor of the system, statically optimizing the scheduling of configurations and executions of the accelerated function on the HRE. The compiler also generates a Configuration Call Graph that will be used as a basis for a more precise dynamic scheduling (see section V). Concerning the accelerated function implementation, the toolchain offers a high level graphical capture. The accelerated function is supposed to be of data-streaming computing type. The graphical interface thus offers a way to best express the parallelism inherent to the function. The benefit of such integrated toolchain is to permit a fast and thus efficient design space exploration: partitioning trials (identification of accelerated parts) and allocation trials (implementation unit choice) with quick feedback since implementation results are obtained with the toolchain itself.

The spatial design part of the toolchain concerns the accelerated function implementation on a HRE from high level specification to retargetable implementation synthesis [20]. This includes data movements management since the HRE are defined to work on the Main Memory Data Structure (MMDS): data circulates on a loop from MMDS to local memories, then back to MMDS. This data movement management is produced by DMA engines that transfer data to local memories according to predefined address patterns.

The design flow starts with the high level specification of an accelerated function required by Molen. This specification (usually in C language) is currently translated manually into an array transformation formalism [21] through the interactive SPEAR [22] graphical interface as entry point. This graphical view contains processes (elementary functions) connected together to form the accelerated function. The SPEAR tool generates automatically the appropriate communication processes between the computing processes. A mechanism implements the communication processes and manages specific data arrays manipulation, communication and scheduling. SPEAR addresses the programming of the DMA unit through parameters definition. The synthesis serves to finalize the design with the implementation on the chip. In order to flow down from this high level specification to the architecture and then physical levels we use a common high level synthesis intermediate description Control Data Flow Graph (CDFG) defined for our specific purpose. The global CDFG is the representation of the complete application mapped and scheduled on a particular HRE. This CDFG includes the connectors mentioned above and some elementary CDFG representing the computation processes written in C language. The front-end analysis module of CASCADE [23] elaborates those CDFG.

An important challenge is the adaptation of the computation kernels to different reconfigurable units. The adaptation will result in a similar graph carrying nodes denoting hardware primitives known to exist in a target architecture (low level CDFG). Reconfigurable architectures are specified using a generic model extrapolated from fine grain FPGAs (Madeo [24]) and an extensible set of classes for resources (LUT, operators, memories, ...). Process based structure is allowed, with additional access to local memories.

V. Runtime System

This section describes the mechanisms used to control the dynamic reconfiguration aspects of the MORPEUS system. The base is formed by a RTOS and is topped by an allocation and scheduling system for reconfigurable operations.

A. Real-time operating system

ECos was chosen for this project as the base for hardware abstraction layer (HAL) and RTOS core as a compromise between the rich set of features of a Linux kernel and the simplicity of minimalist kernels like TinyOS or μC/OS-II. It is highly configurable and offers a choice between its own API

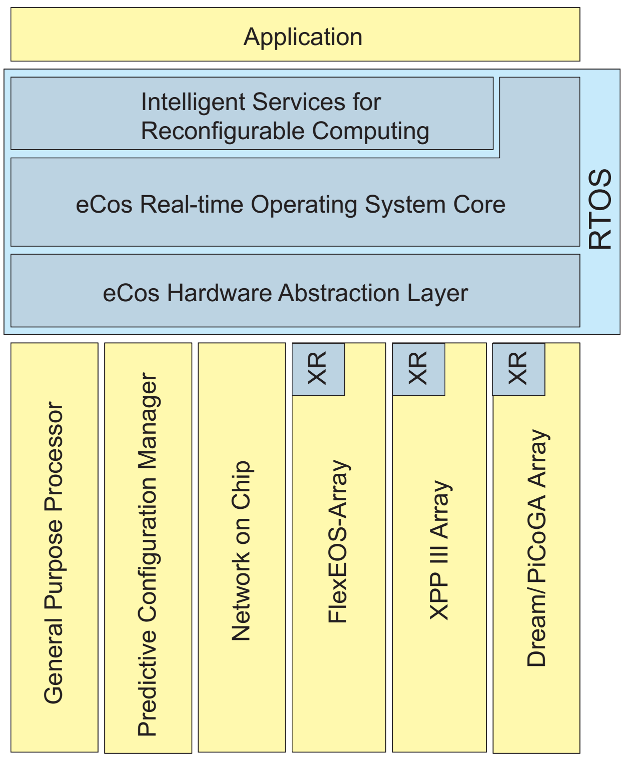

Application
Intelligent Services for Reconfigurable Computing
eCos Real-time Operating System Core
eCos Hardware Abstraction Layer
RTOS
General Purpose Processor
Predictive Configuration Manager
Network on Chip
XR
FlexEOS-Array
XR
XPP III Array
XR
Dream/PiCoGA Array

Fig. 3. Structure of the runtime system

and others like POSIX and μITRON. The RTOS has a layered structure which is shown in figure 3. The bottom layer is the HAL which provides a more uniformed access to the reconfigurable hardware and the system infrastructure. For the transfer of the parameters, it provides virtual exchange registers (XR) for the compiler which are mapped to the parameter registers in the HREs. It also provides the basis for a pipeline service between the HREs. The middle layer is the RTOS Core. It provides the basic operating system services including multithreading that are not related to dynamic reconfiguration and is based on an already existing eCos RTOS. The top layer is formed by the dynamic reconfiguration framework called Intelligent Services for Reconfigurable Computing (ISRC). It provides the services for the configuration and execution of operations on the HREs.

B. Intelligent services for reconfigurable computing

1) ISRC overview: The dynamic control of the reconfiguration units is performed by the ISRC layer on top of the RTOS and the PCM, which is a HW-implemented unit supporting the RTOS and is described in section III. ISRC performs the following actions with information received from the PCM and the application / compiler:

- Allocation decision (choice of the implementation on the various reconfigurable units FlexEOS, XPP, DREAM). If there are functionally equivalent implementations of the reconfigured operation it allows to make a choice at runtime depending on the platform / application status

- Priority calculation of pending operations
- Task execution status management
- Resource request to the PCM for fine dynamic scheduling

Information needed from the compiler is contained in the Configuration Call Graph (CCG). Other information used by the RTOS consists of:

- Result of execution branching decisions
- Synchronization points
- Task parameters

The configuration is handled by ISRC and there is no direct communication between the application and the PCM. The communication between the application and the reconfigurable units is also only indirect and handled by ISRC.

The control of an operation is handled with the help of the HRE status register by writing commands like start execution or stop in this register. The operation is updating its status in the register whenever a state change occurs and indicates the change by an interrupt. The dynamic scheduler not only schedules the different threads but also computes the priorities of pending and near-future operations and schedules them for configuration and execution on the HREs. The priorities are communicated to the PCM to allow the speculative prefetching of configurations. The PCM is commanded by the scheduler to configure the reconfigurable units. The allocation of the heterogeneous reconfigurable engine (HRE) to the different operations is closely linked to the scheduling to consider the configuration time / execution time trade-off if the preferred heterogeneous reconfigurable engine is already in use. As it is preferable (for implementation and control reasons) that the HREs do not perform memory access to the system memory, the operating system controls the DMA-controller of the system. This allows feeding the HREs with data and transferring the result back to the system memory. The information about the memory arrays are provided by the application. At the same time, the runtime system ensures the data consistency by configuring the NoC to use available buffers. The NoC is then controlling the data flow and the buffer switching on its own.

2) ISRC inputs: Input for the RTOS consists of the configuration call graph (CCG), the bitstream library and the implementation / configuration matrix. The bitstream library contains the available operations, their implementations and the properties of these implementations like throughput, delay, size and power. The implementation / configuration matrix contains the list of the implementations used within a configuration and also indicates which configuration contains which implementation. Output of the RTOS are the prefetching priorities and configuration commands for the configuration manager, execution commands for the operations on HREs and control information for the NoC and the DMA-controller.

3) Controlling HREs: The MORPHEUS SoC contains three vastly different reconfigurable units. The FlexEOS from M2000 is a fine-grained embedded FPGA. The DREAM from ARCES is a middle-granular reconfigurable unit with very fast context switches. The XPP from PACT is coarse-granular array of processing elements optimized for pipelining algorithms. With the difference in architecture comes a big difference in the configuration and execution control mechanisms. This has to be transparent for the user of design tools higher up the tool-flow. The difference of the configuration mechanisms is handled by a dedicated PCM unit. The services for reconfigurable computing sit on top of this to provide a uniform interface for the design tools. The control of the HREs extends the SET / EXECUTE concept of the Molen compiler. The extensions have been necessary with regard to concurrent running threads competing for resources. In the original approach by Molen the instruction set of the processor is extended with special machine instructions. These instructions are replaced and extended by operating system calls.

- SET: the compiler notifies the operating system as soon as possible about the next pending operation to configure. The operating system uses this information to prepare configuration. This allocation is done dynamically by ISRC during runtime using the alternative implementations of the operation, which are provided by the designer, on different reconfigurable units, depending on available resources, execution priority and configuration alternatives. The designer can also provide several implementations on the same heterogeneous reconfigurable engine, which are optimized for different criteria. The SET system call is non-blocking and returns immediately.
- EXECUTE: the compiler demands the execution of an operation. The scheduler of ISRC decides then according to current state of the system and the scheduling policy when to execute this operation. The EXECUTE system call is non-blocking and returns immediately.
- BREAK: wait for the completion of a operation for synchronization. The parallel running of operations which are mapped on the HREs with the code on the general purpose processor leads to synchronization issues. Therefore the RTOS provides a BREAK system call which waits for the completion of the referenced operations. The compiler has to assure data consistency between parallel operations by using this system call. For sequential consistency, there is always a need for a break, corresponding to an EXECUTE system call before its data is further processed. It follows that the BREAK system call has to be a blocking operation which returns when all indicated operations are completed.
- MOVTX and MOVFX: transfer data from the ARM processor to a specific exchange register of the HRE and reverse. These instructions are non-blocking and return immediately.
- RELEASE: the configured operation is no longer needed and can be discarded. If an operation is no longer used, for example the calling loop has been left, the allocated resources have to be freed by the compiler with the RELEASE system call. The release system call is non-blocking and returns immediately.

The usage of these system calls can be best demonstrated with the following code example

```
#pragma MOLEN_FUNCTION 1
void func1(int para1, int para2){...}
int p1,p2;
...
func1(p1,p2);
...
```

which is replaced by the compiler with the following code

```
...
SET(1)
...
MOVTX(1,0,&p1)
MOVTX(1,1,&p2)
EXEC(1)
...
BREAK(1)
MOVFX(1,0,&p1)
MOVFX(1,1,&p2)
...
RELEASE(1)
...
```

4) Dynamic scheduling and allocation: The MORPHEUS platform is intended for many conceivable applications. The range goes from purely static data stream processing to highly dynamic reactive control systems. These control systems react on changes of the environment like user interaction, requested quality of service, radio signal strength or battery level in mobile applications. These changes can significantly change the execution path of the application and the priority of the various operations. Static scheduling and allocation is not sufficient for such reactive systems. The topics of scheduling and allocation are tightly related on reconfigurable systems. For this reason the runtime system provides a combined dynamic scheduler and allocator. It determines during runtime schedule and allocation of the operations requested within one thread and in parallel threads. The combined scheduler / allocator determines which alternative implementations, which can include a fall back software implementation, of the requested operations are available and their associated costs. The usage of configuration call graphs (see section 3.4) provides knowledge about the structure of the application. All this information is used to update the schedule of pending operations with the goal to improve metrics like overall throughput and latency. When reasonable the scheduler moves operations from an over-loaded HRE to another HRE between consecutive calls. A requirement for the usefulness of dynamic allocation is that the application designer offers a choice of implementations for as many operations as possible. The user has a choice between various scheduling and allocation strategies.

a) First Come First Serve: This is the most basic scheduling methodology. The only criterion for scheduling is the sequence of requests. When a task is finished the next operation from the queue is examined if the needed HRE is available, otherwise the scheduler waits till the needed HRE is freed. When the operation is available for several HREs they are all tried in sequence of falling throughput.

b) First Come First Free: Instead of one global queue this methodology uses one queue per HRE. At request time the operation is added to each HRE queue where it is available. As soon as a HRE is finished it is used by the next operation in its queue and the operation is removed from all other queues. This method maximizes the overall capacity utilization of each HRE but can result in frequent reconfigurations.

c) Shortest Configuration First: The delay for configuration depends on two factors. First one is the size of the configuration bitstream. It also depends on transfer speed of the bitstream to the HRE which again depends on the used memory of the system. The off-chip sram and flash memories are significantly slower than the on-chip configuration sram managed by the Predictive Configuration Manager. The schedule requests the prefetching state from the PCM and calculates a configuration time. An available HRE is then configured for the operation with the shortest configuration time. This methodology minimizes idle times due to preferring recurring and short operations.

d) Maximum Throughput First: This methodology uses the operation with the highest throughput waiting for a specific HRE. This maximizes overall data throughput but can harm responsiveness.

e) Minimal Delay First: This methodology uses the operation with the lowest delay waiting for a specific HRE. This maximizes responsiveness but can harm data throughput.

f) Weighted Score: All the above methods for scheduling and allocation focus on one criterion while completely ignoring the influence of the other factors for the performance of the system. With the exception of First Come First Serve and First Come First Free they also suffer from the chance that operations starve and never get processed. The solution used here is to compute a score for each operation by weighted sum of all criteria including a waiting-time. The formula for the score can be computed as

$$\begin{aligned} P = &K_C \sum_{i=0}^{3} \frac{ConfigurationSpeed_i}{BitstreamSize_i} + K_T Throughput \\ &+ K_D \frac{1}{Delay} + K_W WaitingTime \end{aligned}$$

The weighting factors K_C, K_T, K_D and K_W are adjustable and allow tailoring of the scheduling to the specific needs of an application. The Σ sum is necessary to consider the different characteristics of the different levels of the configuration memory hierarchy. The index 0 refers to the configuration exchange buffer (CEB) and 1 refers to the on-chip configuration memory. The external configuration memory is divided between sram (index 2) and flash memory (index 3). The equation can be easily extended for other storages as hard disk drives or network storage. The values of $BitstreamSize_0$ and $BitstreamSize_1$ depend on the current prefetching state and can be determined by polling the PCM. Throughput

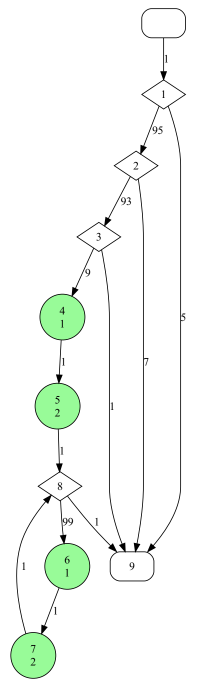


Fig. 4. Configuration Call Graph

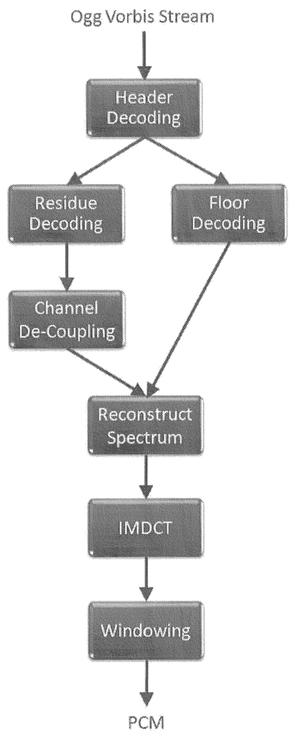


Fig. 5. Ogg Vorbis data flow

and delay are two dimensions of performance measurement. $WaitingTime$ is increased every time an operation is not scheduled to execute and is used to prevent starvation.

5) Configuration Call Graphs: The dynamic allocation is improved by using foresight of coming operations. The runtime system uses for this purpose the configuration call graphs provided by the Molen compiler. The compiler provides one CCG per thread with an example in figure 4. It contains the sequence, including branches and loops, of the operations and their configuration. During runtime the scheduler traces the running of the different application threads through their configuration call graphs to have a global estimate about which heterogeneous reconfigurable engine is going to be available in the near future for use and the next pending operations. This probability information is communicated to the predictive configuration manager which uses it for prefetching configuration bitstreams from external memory to the on-chip configuration memory which results in a significant reduction of reconfiguration overhead. The PCM feeds back information about the prefetching state of the bitstreams to the runtime system. The scheduler uses the prefetching state for allocation decisions, e.g. it can favour a slower implementation which is already in the on-chip configuration memory against a faster implementation which at first has to be loaded from external memory.

6) DMA / DNA: The DMA and DNA provide communication mechanism between various HREs and memory and between HREs for data transfers. The dynamic allocation of operations can make it necessary to change the source or destination address of data transfers which make it essential to handle the communication by the operating system. It provides an API with a unified interface for transferring data with the NoC or the DMA for application programmers or upstream tools like SPEAR. The linking of a transfer to an operation allows to migrate the transfer automatically to the new HRE.

VI. Scenario and Example

It is planned to demonstrate the potential of ISRC with an implementation of the Ogg Vorbis [25] audio codec. Base implementation and reference benchmark are provided by the integer-only Tremor implementation of said codec. The data flow of the decoder can be seen in figure 5.

Profiling has shown that the inverse modified cosine transformation (IMDCT) is the most computation intensive kernel of the decoding algorithm. Implementation for FPGA [26], XPP [27] and PiCoGA are already done which leaves the implementation of the other kernels and adaptation for the MORPHEUS toolchain to be done.

VII. Conclusion and Future Work

In this paper we introduced the intelligent services for reconfigurable computing as a runtime system for heterogeneous reconfigurable systems. We explained how the system calls for controlling the HREs relate to the Molen compiler. Finally we presented a selection of algorithms available for dynamic scheduling and allocation and showed the advantages of the Weighted Score algorithm over other algorithms.

The next steps is the completion of the application example and benchmarking of this application. Additionally the benchmark base could be extended with a broader selection of applications. It would also be interesting to show the versatility of ISRC by porting it to other reconfigurable architectures or RTOS kernels.

Acknowledgment

The authors would like to thank the European union for their support of this work in the scope of the FP6 project MORPHEUS [28].

References

[1] C. Chang, J. Wawrzynek, and R. Brodersen, “BEE2: A high-end reconfigurable computing system,” *IEEE Design & Test*, pp. 114–125, 2005.

[2] J. Angermaier, D. Göhringer, M. Majer, J. Teich, S. Fekete, and J. van der Veen, “The Erlangen slot machine — a platform for interdisciplinary research in dynamically reconfigurable computing,” *it – Information Technology*, vol. 49, pp. 143–149, 2007.

[3] F. Thoma, M. Kuhnle, P. Bonnot, E. M. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. D. Muller-Glaser, and J. Becker, “Morpheus: Heterogeneous reconfigurable computing,” *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 409–414, 27-29 Aug. 2007.

[4] H. Walder and M. Platzner, “Reconfigurable hardware operating systems: From design concepts to realizations,” in *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. Citeseer, 2003, pp. 284–287.

[5] C. Steiger, H. Walder, M. Platzner *et al.*, “Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks,” *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[6] H. So and R. Brodersen, “Improving usability of FPGA-based reconfigurable computers through operating system support,” in *Field Programmable Logic and Applications, 2006. FPL ’06. International Conference on*. Citeseer, 2006.

[7] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, “Hthreads: A computational model for reconfigurable devices,” in *Field Programmable Logic and Applications, 2006. FPL’06. International Conference on*, 2006, pp. 1–4.

[8] J. Agron, W. Peck, E. Anderson, D. Andrews, R. Sass, F. Baijot, and J. Stevens, “Run-time services for hybrid cpu/fpga systems on chip,” in *In Proceedings of the 27th IEEE International Real-Time Systems Symposium, Rio De Janeiro*, 2006.

[9] J. Agron and D. Andrews, “Building Heterogeneous Reconfigurable Systems Using Threads,” in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[10] M. Maheswaran and H. Siegel, “A dynamic matching and scheduling algorithm for heterogeneous computing systems,” *Proceedings of the Seventh Heterogeneous Computing Workshop*, p. 57, 1998.

[11] H. Siegel and S. Ali, “Techniques for mapping tasks to machines in heterogeneous computing systems,” *Journal of Systems Architecture*, vol. 46, no. 8, pp. 627–639, 2000.

[12] T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen *et al.*, “A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems,” in *Proceedings of the 8th Heterogeneous Computing Workshop (HCW’99)*. IEEE Computer Society Washington, DC, USA, 1999, pp. 15–29.

[13] P. Saha and T. El-Ghazawi, “Extending Embedded Computing Scheduling Algorithms for Reconfigurable Computing Systems,” *Programmable Logic, 2007. SPL’07. 2007 3rd Southern Conference on*, pp. 87–92, 2007.

[14] E. Lübbers and M. Platzner, “ReconOS: An RTOS supporting Hard- and Software Threads,” in *17th International Conference on Field Programmable Logic and Applications (FPL), Amsterdam, Netherlands*, 2007.

[15] D. Rossi, F. Campi, A. Deledda, S. Spolzino, and S. Pucillo, “A heterogeneous digital signal processor implementation for dynamically reconfigurable computing,” in *Custom Integrated Circuits Conference, 2009. CICC ’09. IEEE 13-16 Sept. 2009*, 2009, pp. 641–644.

[16] M. Kuehnle, M. Huebner, J. Becker, A. Deledda, C. Mucci, F. Ries, A. M. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia, T. DeMarco, and F. Campi, “An interconnect strategy for a heterogeneous, reconfigurable soc,” *Design & Test of Computers, IEEE*, vol. 25, no. 5, pp. 442–451, Sept.-Oct. 2008.

[17] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, “Spidergon: a novel on-chip communication network,” *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, p. 15, 2004.

[18] S. Chevobbe and S. Guyetant, “Reducing Reconfiguration Overheads in Heterogeneous Multi-core RSoCs with Predictive Configuration Management,” in *ReCoSoC 2008, Barcelona*, 2008.

[19] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, “The MOLEN polymorphic processor,” *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, 2004.

[20] B. Pottier, T. Goubier, and J. Boukhobza, “An integrated platform for heterogeneous reconfigurable computing (*invited paper*),” in *Proc. ERSA’07*, Jun. 2007.

[21] A. Demeure and Y. Del Gallo, “An array approach for signal processing design,” *Sophia-Antipolis conference on Micro-Electronics (SAME), France, October*, 1998.

[22] E. Lenormand and G. Edelin, “An industrial perspective: Pragmatic high end signal processing design environment at thales,” *Proceedings of the 3rd International Samos Workshop on Synthesis, Architectures, modelling, and Simulation*, 2003.

[23] http://www.criticalblue.com.

[24] L. Lagadec, B. Pottier, and O. Villellas-Guillen, *An LUT-Based high level synthesis framework for reconfigurable architectures*. Dekker, Nov. 2003, pp. 19–39.

[25] http://www.vorbis.com.

[26] R. Koenig, A. Thomas, M. Kuehnle, J. Becker, E. Crocoll, and M. Siegel, “A mixed-signal system-on-chip audio decoder design for education,” *RCEducation 2007, Porto Alegre, Brasil*, 2007.

[27] R. Koenig, T. Stripf, and J. Becker, “A novel recursive algorithm for bit-efficient realization of arbitrary length inverse modified cosine transforms,” in *Design, Automation and Test in Europe, 2008. DATE ’08*, March 2008, pp. 604–609.

[28] http://www.morpheus ist.org.

A Self-Checking HW Journal for a Fault Tolerant Processor Architecture

Mohsin AMIN, Camille DIOU, Fabrice MONTEIRO, Abbas RAMAZANI, Abbas DANDACHE
LICM Laboratory, University Paul Verlaine – Metz, 7 rue Marconi, 57070-Metz, France
{amin, diou, ramazani.a,}@univ-metz.fr, {fabrice.monteiro, abbas.dandache}@ieee.org

***Abstract*—In this paper we are presenting a *Dependable Journal* that filters the error(s) from entering into the main memory. It provisionally stores the data before sending to the main memory. The possible errors provoked inside the *Journal* are detected and corrected using *Hamming codes* while the error produced in processor are detected and corrected using *Concurrent Checking Mechanism (CCM)* and *Rollback Mechanism* respectively. In case of error detection in the processor, it rollbacks and re-executes from the last sure state. In this way only Validated Data (*VD*) is written in the main memory.**

The *VHDL-RTL* of the journalized processor has been developed from which it is evident that the depth of the journal has an important impact on the overall area of the processor. So, the depth of the journal has been varied to find the optimal processor area vs. depth of the journal.

I. Introduction

With ever decreasing trend of devices size, they are becoming more sensitive to the effects of radiation [1]. Soft errors, produced due to single event strikes are the most commonly occurring errors both in ASIC and FPGA implementations.

Error Detection and Correction (EDC) have been used to overcome the problem of Single Event Upsets (SEUs) caused by external radiations in [2]. In the near future, the small size and high frequency trend will further increase the failure rate in the modern systems [3] because we have reached the performance bottleneck. By changing such parameters we cannot further increase the performance.

That's why we need to develop architectures having reasonably good performances. Moreover, it should be dependable so that the processed data can be trusted. In this context Dependable Multi-Processor System on Chip (MPSoC) seems to be a convincing approach. We are in the phase of developing an MPSoC while keeping in mind future additional design constraints of power consumption and time-to-market [4].

To do so we have chosen the canonical stack processor [5], has been made fault tolerant by adding some extra *HW* in [6], respecting the constrains of dependable system in [7], [8]. The reasons of choosing and designing of stack processor can be found in the previous work [9]. This work is an extension of our previous work presented in [9] at ReCoSoC'07, in which we have presented the stack processor development methodology for Dependable Applications. In this paper will we present the Journaling Mechanism employed in the Stack Processor to make it fault tolerant. The paper will be focus on: why we need Journaling? And how the *dependable journal* is working?

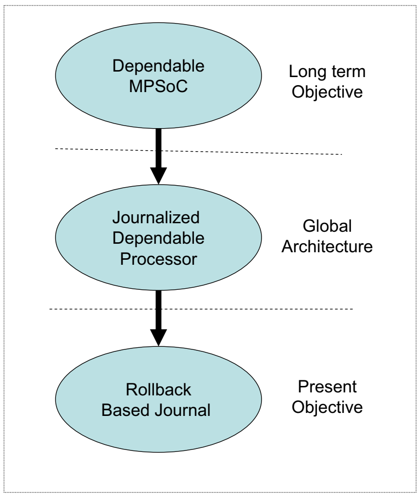


Figure 1. *Precise scope of the propose methodology*

As a first step, we have developed an architecture of dependable stack processor based on HW Journaling as a core processor for MPSoC. This paper is focusing on Journalizing Mechanism employed to make the processor dependable as shown in Fig. 1.

We have an *Error Detecting Fault Processor* working on Rollback Mechanism [10], [11]. For better understanding the reader is invited to consult the previous work presented in [12].

This paper is divided in four sections; in section two we will highlight the need of Journaling, the Journal

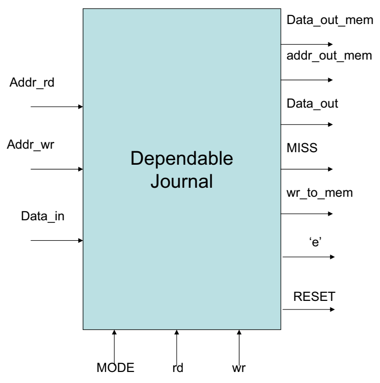
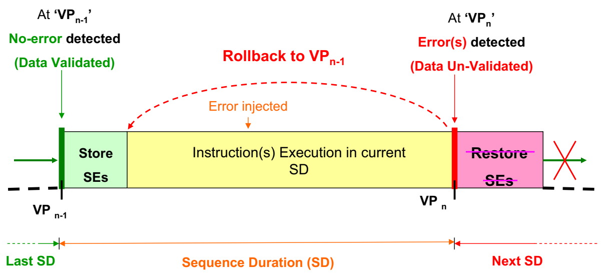


Figure 3. *Configuration of Journal*



Note: **VP** is **V**alidation **P**oint
SE is **S**tate-determining **E**lement(s) of the Processor

Figure 4. *If processor detects error(s) it Rollbacks; else data* `WRITTEN` *in Journal during current SD is Validated at VP*

Management will be discussed in section three. The section four contains the results. The conclusions have been discussed in section five.

II. Why Journaling

We are supposing that a Dependable Main Memory is attached to our system. It means that no error can itself be generated in it. All the efforts should be done to filter the external errors from entering into the *Dependable Main Memory*. That's why we are temporarily storing the data (until validation) in journal before WRITING into the *Dependable Main Memory*.

In this regard, we are proposing a *Dependable Journal to provide a temporary storage location for effective Rollback*. It has a fault detection and correction mechanism to overcome its internal temporary faults.

The technique has a little overhead in terms of area for overall architecture but a great advantage that our main memory remains always sure. There is no time overhead or `MISS` in the *Journal* as processor always checks data simultaneously in *Journal* and main memory. If data is found both in *Journal* and main memory then data from *Journal* will be preferred as it is a more recent written data.

III. Journal Management

A *VHDL-RTL* model of the Journalized processor has been implemented in Quartus II on a Stratix II family FPGA. The architectural details of stack processor have been discussed in our previous work [9]. The overview of dependable stack processor is shown in Fig. 2. The dotted square surrounding the three blocks represents the *Dependable Journal*. The *Dependable Journal* has been sub-divided into three components for the ease of the reader. In actual architecture they constitute a single block, as shown in Fig. 3 which furthermore, elaborates the IN/OUT-Ports configuration.

The Journal has internally two parts; one containing the Validated data (No-error Detection at *VP*) and second containing the data written in the present *SD* (Un-Validated Data), as discussed in [12]. The Journal has an internal mechanism to differentiate between Validated and Un-Validated Data.

Globally, all the data to be written in the main memory pass through *Journal* until being validated. The validation process in the processor is based on a *Rollback Mechanism*. The Validation Point (*VP*) occurs at the end of each Sequence Duration (SD) as shown in Fig. 4. The fundamental architecture of *Journal*, *SD* and *VP* is discussed in [12].

There are three possible data-flows from/to the Journal.

`WRITE` to Journal:

The Processor can `WRITE` data directly into the Journal and not in the Dependable Main Memory. The data stays temporarily in the Journal until being validated. The Processor can detect an internal error during the current Sequence Duration (*SD*). If no error is detected at the end of *SD*, the data is validated and then it can be transferred to the Main Memory.

Journal to Dependable Main Memory:

Only Validated data is transferred to the main memory. As data before transferring stays temporarily in the Journal and there is a possibility of provoking error due to internal and external temporarily and intermittent faults (the permanent faults are not addressed in this

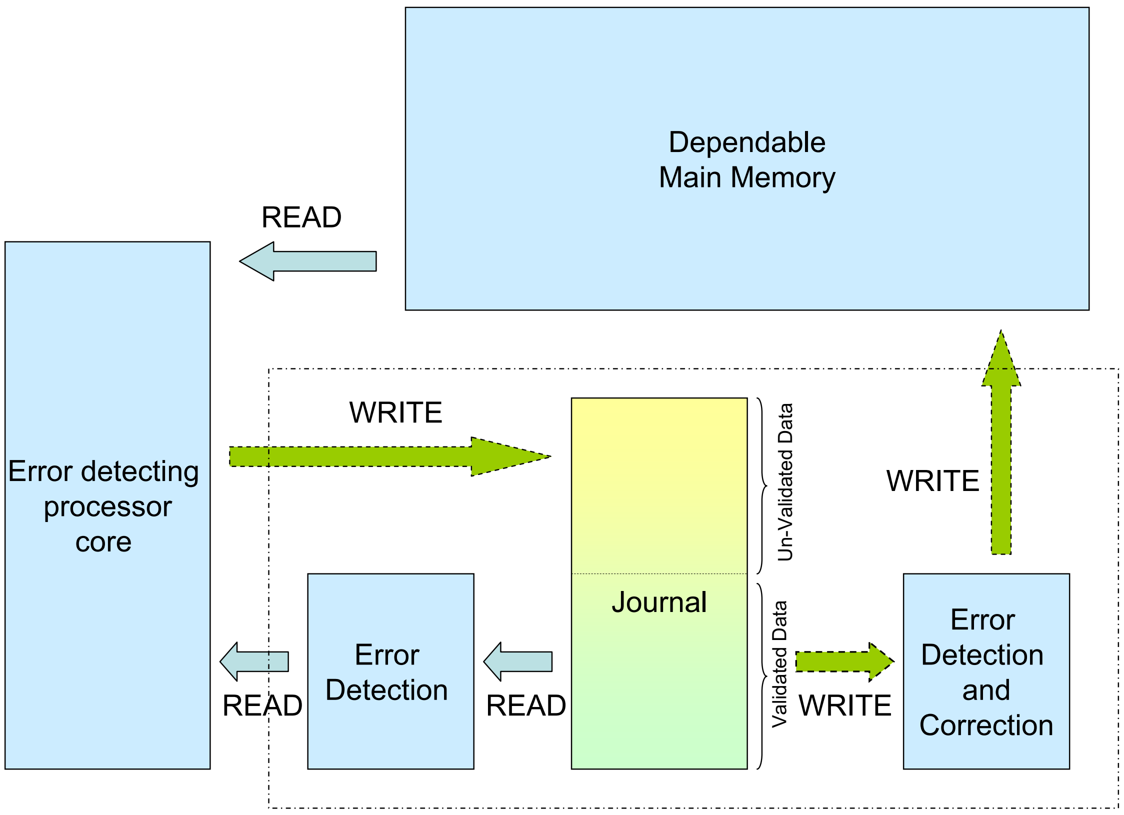


Figure 2. *Block Diagram of overall architecture*

work.). Hence this data is checked for error. There is a mechanism based on Hamming Codes [13], [14] for detection and correction of error(s).

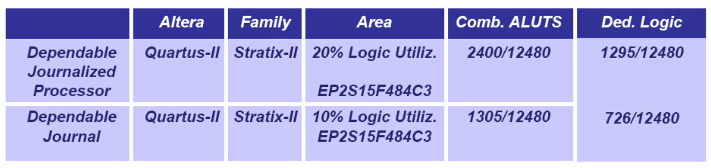
If an error is detected in the data, then it is corrected and sent to the main memory. If a non-corrigible error is detected then in this case the processor is sent a `RESET` request as shown in the flow diagram in Fig. 5.

`READ` from the Journal

The last possibility occurs when Processor want to `READ` data from the Journal. As shown in Fig. 2, the required data before sent to the processor is checked for errors. If error is detected the *Rollback Mechanism* is activated. The processor re-executes the Sequence of instructions from the last sure state. Otherwise data is sent to the Processor.

IV. Results

The VHDL-RTL model of the Journalized-Processor has been implemented in Altera Quartus-II on a Stratix-II family FPGA. The architectural details of the processor, error modeling and its performance in presence of errors have been calculated for overall architecture. Among



	Altera	Family	Area	Comb. ALUTS	Ded. Logic
Dependable Journalized Processor	Quartus-II	Stratix-II	20% Logic Utiliz. EP2S15F484C3	2400/12480	1295/12480
Dependable Journal	Quartus-II	Stratix-II	10% Logic Utiliz. EP2S15F484C3	1305/12480	726/12480

Figure 6. *Effective Area utilization on FPGA*

them area utilization of the architecture has been discussed here.

In this paper we are focusing on: why we need Journal? How it is working? And the effect of Journal-depth on the overall area of the processor. From the implementation in Fig. 6, we have found that the size of the Journal is limiting the overall area of the processor. The Journal is occuping around 50% of the total Area.

Moreover, we are designing this processor core to integrate with in an MPSoC, where there will be multicores so, small area will be preferred.

Accordingly, we have varied the depth of the Journal and observed its impact on the Area of the Processor

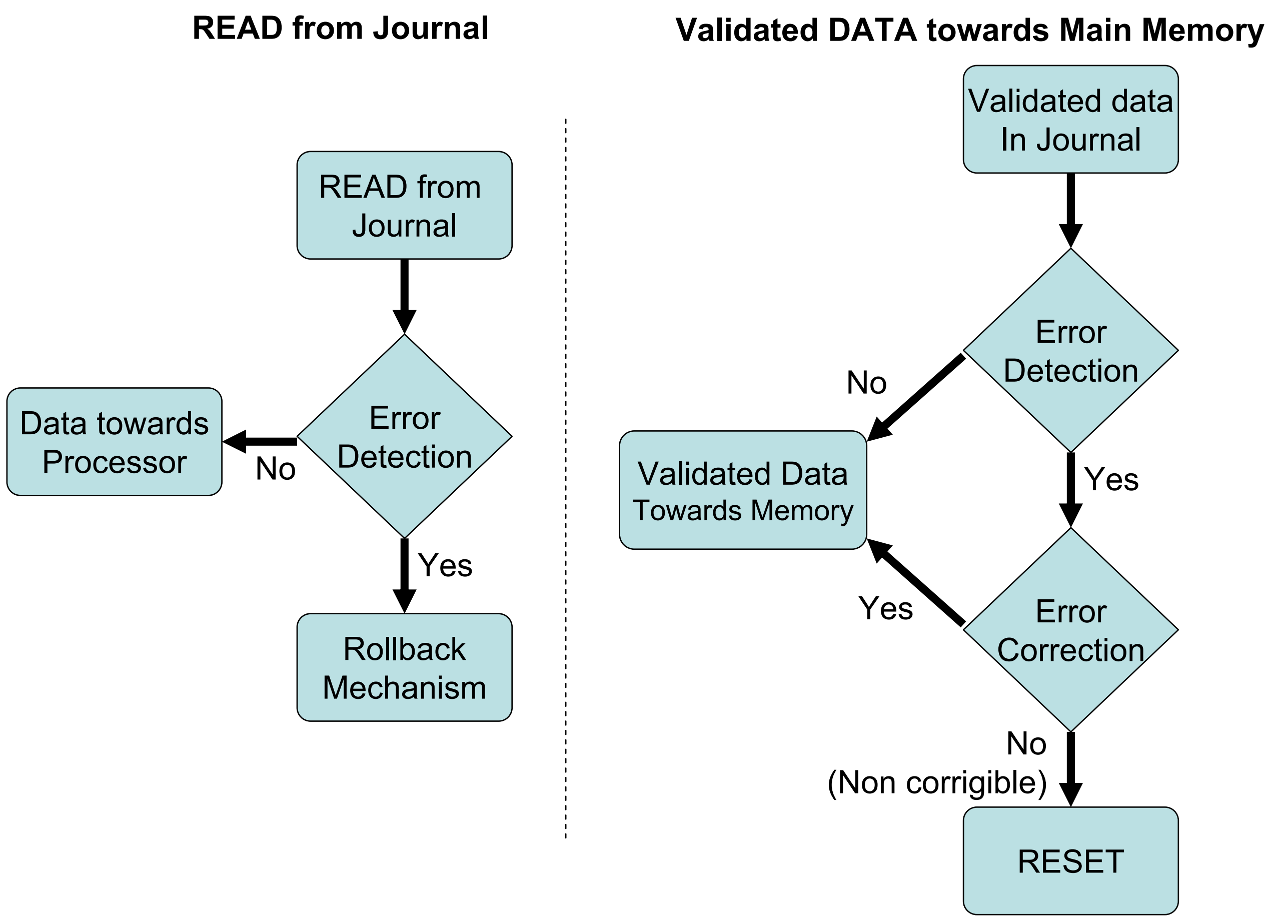


Figure 5. *Dependable Journal Management*

in the Fig 7. By increasing the depth from 16 blocks to 64 blocks, the area of the Processor has increased exponentially as shown in the Graph. From the [12] we have observed that for small *SD* small depth of the journal are feasible, which means small overall Area of the Processor.

V. Conclusions

The presence of the *Journal* facilitates the *rollback mechanism* on one hand and on other filters all possible errors from entering into the main memory. As data temporarily reside in the *Journal* until validation so error detecting and correcting based on *hamming code* provide effective double detection and single error correction. The effective area is quite small which favors our processor to become the active core of the *Dependable MPSoC*.

The depth of the journal is a limiting factor for overall area of the Journalized Dependable Processor.

References

[1] L. Lantz, “Soft errors induced by alpha particles,” IEEE Transactions on Reliability, vol. 45, pp. 174-179, June 1996.

[2] J. A. Fifield and C. H. Stapper, “High-speed on Chip ECC For Synergistic Fault-Tolerant Memory Chips,” in Proc. IEEE Journal of Solid State Circuits, vol. 26, no. 10, pp. 1449-1452, Oct. 1991.

[3] D. G. Mavis and P. H. Eaton, “Soft Error Rate Mitigation Techniques For Modern Microcircuits,” Proc. Intl. Reliability Physics Symposium, pp. 216-225, 2002.

[4] A. A. Jerraya, A. Bouchhima, F. Petrot, “Programming models and $HW - SW$ Interfaces, Abstraction for Multi-Processor SoC,” in ACM (DAC’2006), California, USA, 2006.

[5] Philip J. Koopman,“Stack Computers: The New Wave", California : Ed. Mountain View Press, 1989.

[6] A. Ramazani, M. Amin, F. Monteiro, C. Diou, A. Dandache,“A Fault Tolerant Journalized Stack Processor Architecture,” 15th IEEE International On–Line Testing Symposium (IOLTS’09), Sesimbra–Lisbonne, Portugal, 24–27 June 2009.

[7] A. Avizienis, J.C. Laprie, B. Randell and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” IEEE Transactions on Dependable and Secure Computing, vol. 1, issue no. 1, pp.11-33, January-March 2004.

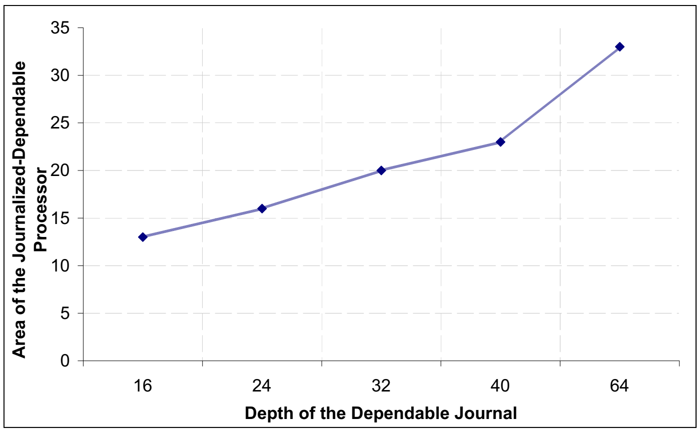


Figure 7. *Effect of Journal Depth on the overall Area*

[8] D.K. Pradhan, "Fault-Tolerant Computer System Design", Prentice Hall, 1996.

[9] M. Jallouli, C. Diou, F. Monteiro, A. Dandache, "Stack processor architecture and development methods suitable for Dependable Applications," Reconfigurable Communication-centric SoCs (ReCoSoC'07), Montpellier, France, June 18 - 20, 2007.

[10] D.B. Hunt and P.N. Marinos, "A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique," In Proceedings of 17th Annual Symposium on Fault-Tolerant Computing, pp. 170–175, 1987.

[11] N.S. Bowen, D.K. Pradhan,"Virtual checkpoints: architecture and performance," IEEE Transactions on Computers, vol. 41, issue no. 5, pp. 516–525, May 1992.

[12] M. Amin, F. Monteiro, C. Diou, A. Ramazani, A. Dandache, "A HW/SW Mixed Mechanism to Improve the Dependability of a Stack Processor", 16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS'09, Hammamet, Tunisia, December 13-16, 2009.

[13] J.F. Wakerly, "Error Detection Codes, Self-Checking Circuits and Applications", North Holland, 1978.

[14] R.W. Hamming, "Error Detecting and Error Correcting Codes", Bell System Tech. Jour., vol. 29, pp. 147-160, 1950.

A Task-aware Middleware for Fault-tolerance and Adaptivity of Kahn Process Networks on Network-on-Chip

Onur Derin, Erkan Diken
ALaRI, Faculty of Informatics
University of Lugano
Lugano, Switzerland
derino@alari.ch, dikene@usi.ch

***Abstract*—We propose a task-aware middleware concept and provide details for its implementation on Network-on-Chip (NoC). We also list our ideas on the development of a simulation platform as an initial step towards creating fault-tolerance strategies for Kahn Process Networks (KPN) applications running on NoCs. In doing that, we extend our SACRE (Self-adaptive Component Run-time Environment) framework by integrating it with an open source NoC simulator, Noxim. We also hope that this work may help in identifying the requirements and implementing fault tolerance and adaptivity support on real platforms.**

***Index Terms*—fault tolerance; KPN; middleware; NoC; self-adaptivity;**

I. Introduction

Past decade has witnessed a change in the design of powerful processors. It has been realized that running processors at higher and higher frequencies is not sustainable due to unproportional increases in power consumption. This led to the design of multi-core chips, usually consisting of multi-processor symmetric Systems-on-Chip (MPSoCs), with limited numbers of CPU-L1 cache nodes interconnected by simple bus connections and capable in turn of becoming nodes in larger multiprocessors. However, as the number of components in these systems increases, communication becomes a bottleneck and it hinders the predictability of the metrics of the final system. Networks-on-chip (NoCs) [1] emerged as a new communication paradigm to address scalability issues of MPSoCs. Still, achieving goals such as easy parallel programming, good load balancing and ultimate performances, dependability and low-power consumption pose new challanges for such architectures.

In addressing these issues, we adopted a component-based approach based on Kahn Process Networks (KPN) for specifying the applications [2]. KPN is a stream-oriented model of computation based on the idea of organizing an application into streams and computational blocks; streams represent the flow of data, while computational blocks represent operations on a stream of data. KPN presents itself as an acceptable trade-off point between abstraction level and efficiency versus flexibility and generality. It is capable of representing many signal and media processing applications that occupy the largest percentage of the consumer electronics in the market.

Our eventual goal is to run a KPN application directly on a NoC platform with self-adaptivity and fault-tolarence features. It requires us to implement a KPN run-time environment that will run on the NoC platform and support adaptation and fault-tolerance mechanisms for KPN applications on such platforms. We propose a self-adaptive run-time environment (RTE) that is distributed among the tiles of the NoC platform. It consists of a middleware that provides standard interfaces to the application components allowing them to communicate without knowing about the particularities of the network interface. Moreover the distributed run-time environment manages the adaptation of the application for high-level goals such as fault-tolerance, high performance and low-power consumption by migrating the application components between the available resources and/or increasing the parallelism of the application by instantiating multiple copies of the same component on different resources [3], [4]. Such a self-adaptive RTE constitutes a fundamental part in order to enable system-wide self-adaptivity and continuity of service support [5].

In view of the goal stated above, we propose to use our SACRE framework [4] that allows creating self-adaptive KPN applications. In [3], we listed platform level adaptations and proposed a middleware-based solution to support such adaptations. In the present paper, we define the details of the self-adaptive middleware particularly for NoC platforms. In doing that we choose to integrate SACRE with the Noxim NoC simulator [6] in order to realize functional simulations of KPN applications on NoC platforms. An important issue regarding the NoC platform is the choice of the communication model. Depending on the NoC platform, we may have a shared memory space with the Non-Uniform Memory Access (NUMA) model or we may rely on pure message passing with the No Remote Memory Access (NORMA) model [7]. Implementing KPN semantics on NORMA presents itself as the main challange. In the NUMA case, it is straightforward as long as the platform provides some synchronization primitives. Section III explains details of the middleware in the NORMA case. Section IV presents our ongoing effort to integrate SACRE and Noxim. Section V, VI list the requirements for the implementation of fault tolerance and adaptivity mechanisms.

II. Related Work

The trend from single core design to many core design has forced to consider inter-processor communication issues for passing the data between the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [8] that targets the inter-core communication in a multicore chip. MCAPI is the light-weight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [9].

However, the communication primitives available with these message passing libraries don't support the blocking write operation as required by KPN semantics. Main features in order to implement KPN semantics are blocking read and, in the limited memory case, blocking write. Key challenge is the implemention of the blocking write feature. There are different approaches addressing this issue. In [10], a programming model is proposed based on the MPI communication primitives (MPI_send() and MPI_receive()). MPI_receive blocks the task until the data is available while MPI_send is blocking until the buffer is available on the sender side. Blocking write feature is implemented via operating system communication primitives that ensure the remote processor buffer has enough space before sending the message. Another approach is presented in [11], a network end-to-end control policy is proposed to implement the blocking write feature of the FIFO queues.

Our approach is based on a novel implementation of KPN semantics on NoC platforms. We propose an active middleware layer that implements the blocking write feature through virtual channels that are introduced in opposite directions to the original ones.

III. Task-aware Middleware

A KPN application consists of a set of parallel running tasks (application components) connected through non-blocking write, blocking read unbounded FIFO queues (connectors) that carry tokens between input and output ports of application components. A token is a typed message. Figure 1 shows a simple KPN application. Running a KPN application on a NoC platform would require to map the application components on the several tiles of the NoC platform.

A. Requirements

When deciding on the implementation details of a middleware that will support running of a KPN application on a NoC platform, we came up with some requirements for the middleware. The most fundamental requirement for a middleware to support KPN semantics is the ability to transfer tokens among tiles assuring blocking read. Since unbounded FIFOs cannot be realized on real platforms, FIFOs have to be bounded. Parks' algorithm [12] provides bounds on the queue sizes through static analysis of the application. In the case of bounded queues, blocking write is also required to be supported.

Another requirement is that we would like to have platform independent application components. This will make it easier

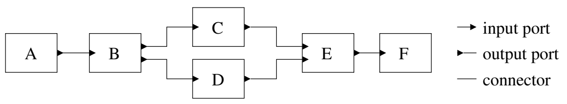


Fig. 1: A simple KPN application with application components A, B, C, D, E, F

to program for the platform by allowing the development of application components in isolation and running them without modifications. This can be achieved by separating the KPN library that will be used to program the application from the communication primitives of the platform. Middleware will link the KPN library to the platform specific communication issues.

In line with the above requirement, we would like that application components are not aware of the mapping of components on the platform. They should only be concerned with communicating tokens to certain connectors. Therefore the middleware should enable mapping-independent token routing. These requirements are of great importance if we want to achieve fault tolerance and adaptivity of KPN applications on NoC platforms in such a way that assures separation of concerns. This means that it is the platform that provides fault-tolerance and adaptivity features to the application and not the application developer.

B. Middleware implementation in the NORMA case

In the NORMA model, tasks only have access to the local memory and there is no shared address space. Therefore tasks on different tiles have to pass the tokens among each other via message passing routines supported by the network interface (NI).

In order to address the middleware requirements previously listed, our key argument is the implementation of an active middleware layer that appears as a KPN task and gets connected to other application tasks with the connectors of the specific KPN library that is adopted by the application components. Opposedly, a passive middleware layer would be a library with a platform specific API for tasks to receive and send tokens.

We build our middleware on top of *MPI_recv()* and *MPI_send()* primitives. These methods allow sending/receiving data to/from a specific process regardless of which tile the process resides on. MPI_recv blocks the process until the message is received from the specified process-tag pair. MPI_send is non-blocking unless there is another process on the same tile that has already issued an MPI_send.

Every tile has a middleware layer that consists of middleware sender and receiver processes. Figure 2 shows the middleware layers and a possible mapping of the example pipeline on four tiles of a 2x2 mesh NoC platform. There is a sender process for each outgoing connector. An outgoing connector is one that is connected to an input port of the application component that resides on a different tile. Similarly

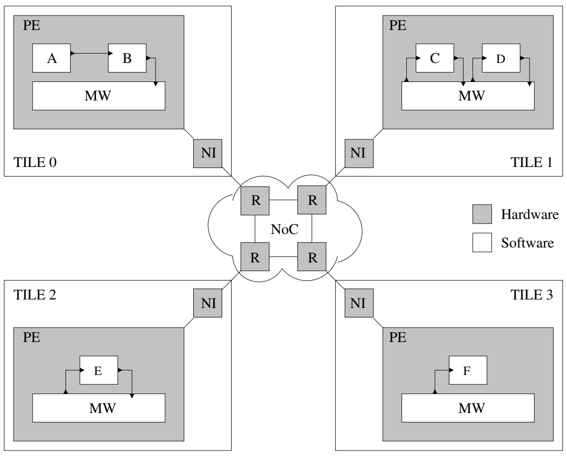


Fig. 2: KPN example mapped on 2x2 mesh NoC platform

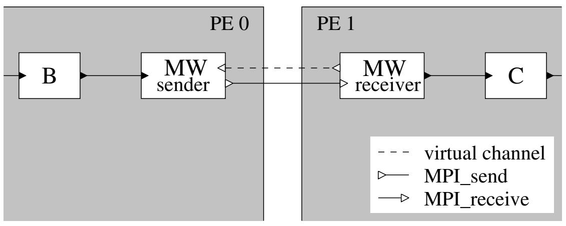


Fig. 3: Middleware details for the connector between B and C

```
loop
    $t \leftarrow input\_port.read()$
    $MPI\_recv(t_{vc}, P_R, vc)$
    $MPI\_send(t, P_R, oc)$
end loop
```

Fig. 4: Sender middleware task per outgoing connector (t: data token, t_{vc}: dummy token for virtual channel, P_R: process identifier of the remote middleware task, vc: virtual channel tag, oc: original channel tag)

```
for $i = 1$ to channel_bound do
    $MPI\_send(t_{vc}, P_R, vc)$
    $i \leftarrow i + 1$
end for
loop
    $MPI\_recv(t, P_R, oc)$
    $MPI\_send(t_{vc}, P_R, vc)$
    $output\_port.write(t)$
end loop
```

Fig. 5: Receiver middleware task per incoming connector

there is a receiver process for each incoming connector. These processes are actually KPN tasks with a single port. This is an input port for a sender process and an output port for a receiver process. The job of a sender middleware task is to transmit the tokens from its input over the network to the receiver middleware task on the corresponding tile (i.e. the tile containing the application component to receive the token). Similarly, a receiver middleware task should receive the tokens from the network and put them in the corresponding queue. Figure 3 shows the sender and receiver middleware tasks between the ports of application components B and C.

We need to implement a blocking write blocking-read bounded channel that has its source in one processor and its sink in another one. MPI_send as described above does not implement blocking write operation. It can be modified and be implemented in such a way that it checks whether the remote queue is full or not by using low-level support from the platform [10], [11]. In order to do this in a way that wouldn't require changes to the platform, we make use of the *virtual channel* concept. A virtual channel is a queue that is connected in the reverse direction to the original channel. For every channel between sender and receiver middleware tasks, we add virtual channels that connects the receiver middleware task to the sender middleware task. Figure 3 shows the virtual channel along with the sender and receiver middleware tasks for the outgoing connector from application component B to C. The receiver task initially puts as many tokens to the virtual channel as the predetermined bound of the original channel. The sender has to read a token from the virtual channel before every write to the original channel. Similarly the receiver has to write a token to the virtual channel after every read from the original channel. Effectively, virtual channel enables the sender to never get blocked on a write. The read/write operations from/to original and virtual channels can thus be done using MPI_send and MPI_receive as there is no more need for blocking write in presence of virtual channels. Figure 4 and 5 show the pseudocodes for sender and receiver middleware tasks, respectively.

With the middleware layer, an outgoing blocking queue of bound b in the original KPN graph is converted into three blocking queues: one with bound $b1$ between the output port of the source component and the sender middleware task; one with bound $b2$ between the sender and receiver middleware tasks; one again with bound $b2$ between the receiver middleware task and the input port of the sink component. Values $b1$ and $b2$ can be chosen freely such that $b1 + b2 \geq b$ and $b1, b2 > 0$.

If the middleware layer is not implemented as an active layer, then the application tasks would need to be modified to include the virtual channels. Moreover, use of virtual channels enables us to not require changes to the NoC for custom signalling mechanisms.

Another benefit of having virtual channels is avoiding deadlocks. Since MPI_send can be issued by different middleware tasks residing on the same tile in a mutually exclusive way, there may be deadlock situations for some of the task mapping decisions. For example, consider the case (see Fig. 1) where an application task (C) is blocked on a call to MPI_send until the queue on the receiver end is not full. It may be that the application task on the receiver end (E) is also blocked waiting for a token from an application task (D) on the tile where

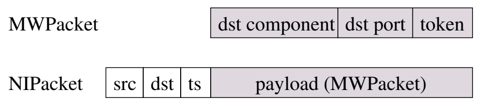


Fig. 6: Structure of packets at the NI and middleware levels (src: source tile, dst: destination tile, ts: timestamp, dst component: destination component, dst port: destination port)

C resides. Since tasks on the same tile has to wait until the MPI_send call of the other task returns, D cannot write the token to be received by E. Therefore we have a deadlock situation where C is blocked on E, E is blocked on D, D is blocked on C. With virtual channels, it is guarenteed that an MPI_send call will not ever be blocked.

The problem of deadlocking can be solved also without using virtual channels. However that would require implementing expensive distributed conditional signalling mechanisms on the NoC or inefficient polling mechanisms.

IV. Simulation with SACRE and Noxim

Noxim [6] is an open source and flit-accurate simulator developed in SystemC. It consists of tunable configuration parameters (network size, routing algorithm, traffic type etc.) in order to analyze and evaluate the set of quality indices such as delay, throughput and energy consumption.

SACRE [4] is a component framework that allows creating self-adaptive applications based on software components and incorporates the Monitor-Controller-Adapter loop with the application pipeline. The component model is based on KPN. It supports run-time adaptation of the pipeline via parametric and structural adaptations.

We started integrating SACRE and Noxim in order to be able to simulate KPN applications on NoC platforms. We aim to implement the proposed middleware for the NORMA case. However we don't have the MPI_send and MPI_receive primitives in Noxim. Actually it doesn't even come with a NI. We implemented the transport layer such that we can send data and reconstruct the data on the other end. In absence of MPI primitives in SACRE-Noxim, we propose to implement the task-aware middleware over the transport layer of the NoC network interface as described below.

We conceived the middleware as a KPN task by extending it from *SACREComponent* in order to be able to connect it to the queues of the local application tasks. Middleware is also inherited from the base SystemC module class (i.e. *sc_module*) with *send* and *receive* processes.

The send process is activated on every positive edge of the clock and reads the input ports in a non-blocking manner. When there is a token to be forwarded, it wraps the token into a *MWPacket* object as shown in Figure 6 by adding the destination task name and destination port name as the header information.

This data is looked up from the *port connection table*. This table represents the KPN application and shows which components and which ports are connected with each other as shown in Table I.

TABLE I: Port connection table for Figure 1

Source		Destination	
component	port	component	port
A	out1	B	in1
B	out1	C	in1
B	out2	D	in1
C	out1	E	in1
D	out1	E	in2
E	out1	F	in1

TABLE II: Component mapping table for Figure 2

Component	Tile
A	0
B	0
C	1
D	1
E	2
F	3

Then the *MWPacket* object is sent via the NI to the destination tile by wrapping it in a *NIPacket* object. *NIPacket* has the structure shown in Figure 6. The destination tile identifier is looked up from the *component mapping table* stored in the middleware. This table stores which components reside on each tile as shown in Table II. Currently NI transfers packets spliting them into several flits through wormhole routing.

The receiver process is also activated on every positive edge of the clock. The network interface receives the flits and reconstructs the *MWPacket* object. Then the receiver process extracts the token and puts it in the right queue by looking at the header of the *MWPacket*.

V. Fault-tolerance support

Having isolated the application tasks from the network interface, we believe it will be easier to implement fault tolerance mechanisms. Until now, we have only considered task migration and semi-concurrent error detection as fault tolerance mechanisms.

A. Task migration

In the case when a tile fails, the tasks mapped on that tile should be moved to other tiles. Therefore we will have a controller implementing a task remapping strategy. For now, we don't focus on this but rather deal with the implementation of task migration mechanisms.

Moving an application component from one tile to another requires the ability to start the task on the new tile, update the component mapping tables on each tile, create/destroy MW tasks for outgoing and incoming connectors of the migrated components and transfer the tokens already available in the connectors of the migrated components along with those components. In case of a fault, the tokens in the queues pending to be forwarded by the middleware tasks in the failed tile may be lost along with the state of the task if it had any. Similarly, there may be some number of received flits that haven't been reconstructed to make up a packet yet. We may need to put in measures to checkpoint the state of the task and the middleware queues. As a rollback mechanism, we should

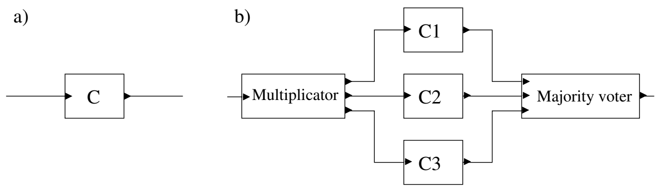


Fig. 7: Semi-concurrent error detection at application level. Component in (a) is replicated by three as shown in (b)

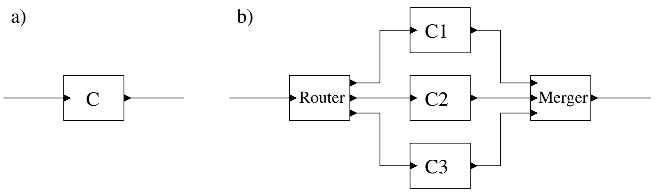


Fig. 8: Adaptation pattern for parallelization. Component in (a) is parallelized by three as shown in (b)

be able to transfer both the state of the tasks and the queues on the faulty tile to the new tiles. The flits already in the NoC buffers destined to the faulty tile should be re-routed to the new tiles accordingly. This may be easier to achieve if we implement the task-awarenes feature in the NoC routers. Otherwise, it should be the NI or the router of the faulty tile that should resend those flits back in the network with correct destinations. We need to futher analyze the scenarios according to the extend of faults (e.g. only the processing element is faulty or whole tile is faulty). However, thanks to the middleware layer, application tasks won't need to know that there has been a task migration.

B. Semi-concurrent error detection at application level

We propose to employ semi-concurrent error detection [13] as a dependability pattern for self-adaptivity. The run-time environment can adapt the dependability at the application level. This enables adaptive dependability levels for different parts of the application pipeline at the granularity of an application component.

In the case of a single component, parallel instances of the component are created on different cores along with multiplicator components and majority voter components for each input and output ports respectively as shown in Figure 7. Multiplicator component creates a copy of the incoming message for each redundant instance and forwards it to them along with a unique tag identifying the set of copied messages. Majority voter component queues up all the output messages until it has as many messages with the same tag as the number of redundant instances. Then it finds out the most recurrent message and sends it to its output connector. A time-out mechanism can also be put in place to tolerate when a core is faulty and no message is being received by a component.

VI. Adaptivity support

In [3], [4], we had listed possible run-time adaptations of KPN applications at application and platform level. Application programmer provides a set of goals to be met by the application. These goals are translated into parameters to be monitored by the platform. The adaptations are driven by an adaptation control mechanism that tries to meet the goals by monitoring those parameters. We need to elaborate on the implementation of these adaptations on the NoC platform. An example of a structural adaptation in order to meet performance and low-power goals is the parallelization pattern explained below.

A. Adapting the level of parallelism

Parallelization of a component is one type of structural adaptation that can be used to increase the throughput of the system as shown in Figure 8. This is done by creating parallel instances of a component and introducing a router before and a merger after the component instances for each of the input and output ports. A router is a built-in component in our framework that can work in a load-balancing or round-robin fashion; this component routes the incoming messages to either one of the instances depending on its policy. If there is no ordering relation between incoming and outgoing messages, the merger components simply merge the output messages from the output ports of the instances into one connector disregarding the order of messages on the basis of whichever message is first available. However, for the general class of KPN applications, semantics require that the processes comply with the monotonicity property. In that case, the ordering relation has to be preserved. For that purpose, the router component tags every message that would originally go to the component with an integer identifier that counts up for each message from value 1. Then the merger components have to queue up the output messages so as to achieve an order in terms of their tags. If there are multiple processor cores available, this mechanism would increase the parallelism of the application. However the condition for applicability of such an adaptation is the absence of inter-message dependencies.

VII. Conclusion and Future work

We propose an active middleware layer to accommodate KPN applications on NoC-based platforms. Besides satisfying KPN semantics, the middleware allows platform independent application components with regard to communication. It is solely based on MPI_send and MPI_recv communication primitives, thus it doesn't require any modification to the NoC platform. The middleware is an initial step towards implementing a self-adaptive run-time environment on the NoC platform.

As future work, the performance implications of additional middleware tasks should be investigated. The impact of virtual channel tokens on overloading of the NoC should be assessed. Although it may not always be a matter of choice, the performances of KPN applications in NUMA and NORMA architectures need to be evaluated. We have established a collaboration with University of Cagliari in order to implement the presented middleware on their FPGA-based NoC platform

[14]. This will allow us to evaluate the proposed middleware in a real setting and obtain results for the active vs. passive middleware and NORMA vs. NUMA cases.

ACKNOWLEDGMENT

This work was funded by the European Commission under the Project MADNESS (No. FP7-ICT-2009-4-248424). The paper reflects only the authors' view; the European Commission is not liable for any use that may be made of the information contained herein.

REFERENCES

[1] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[3] O. Derin and A. Ferrante, "Simulation of a self-adaptive run-time environment with hardware and software components," in *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*. New York, NY, USA: ACM, August 2009, pp. 37–40.

[4] O. Derin and A. Ferrante, "Enabling self-adaptivity in component-based streaming applications," *SIGBED Review*, vol. 6, no. 3, October 2009, special issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'09).

[5] O. Derin and A. Ferrante and A. V. Taddeo, "Coordinated management of hardware and software self-adaptivity," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 170 – 179, 2009.

[6] "Noxim NoC simulator." [Online]. Available: http://noxim.sourceforge.net

[7] E. Carara, A. Mello, and F. Moraes, "Communication models in networks-on-chip," in *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 57–60.

[8] "Multicore associations communication api." [Online]. Available: http://www.multicore-association.org

[9] "A high performance message passing library." [Online]. Available: http://www.open-mpi.org/

[10] Gabriel Marchesan Almeida and Gilles Sassatelli and Pascal Benoit and Nicolas Saint-Jean and Sameer Varyani and Lionel Torres and Michel Robert, "An Adaptive Message Passing MPSoC Framework," *International Journal of Reconfigurable Computing*, vol. 2009, p. 20, 2009.

[11] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis, "Mapping kpn models of streaming applications on a network-on-chip platform," in *ProRISC 2009: Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits*, November 2009.

[12] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California, Berkeley, CA 94720, December 1995.

[13] A. Antola, F. Ferrandi, V. Piuri, and M. Sami, "Semiconcurrent error detection in data paths," *IEEE Trans. Comput.*, vol. 50, no. 5, pp. 449–465, 2001.

[14] P. Meloni, S. Secchi, and L. Raffo, "Exploiting FPGAs for technology-aware system-level evaluation of multi-core architectures," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2010.

Dynamic Reconfigurable Computing:
the Alternative to Homogeneous Multicores under Massive Defect Rates

Monica Magalhães Pereira and Luigi Carro
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{mmpereira, carro}@inf.ufrgs.br

Abstract— **The aggressive scaling of CMOS technology has increased the density and allowed the integration of multiple processors into a single chip. Although solutions based on MPSoC architectures can increase the application's speed through task level parallelism, this speedup is still limited to the amount of parallelism available in the application, as demonstrated by Amdahl's Law. Another fundamental aspect is that for new technologies, very aggressive defect rates are expected, since the continuous shrink of device features makes them more fragile and susceptible to break. At high defect rates a large amount of processors of the MPSoC will be susceptible to defects, and consequently will fail, reducing not only yield, but also severely affecting the expected performance. In this context, this paper presents a run-time adaptive architecture design that allows software execution even under aggressive defect rates. The proposed architecture can accelerate not only highly parallel applications, but also sequential ones, and it is a heterogeneous solution to overcome the performance penalty that is imposed to homogeneous MPSoCs under massive defect rates. In the experimental results we compare performance and area of the proposed architecture to a homogeneous MPSoC solution. The results demonstrate that the architecture can sustain software acceleration even under a 20% defect rate, while the MPSoC solution does not allow any software to be executed under a 15% defect rate for the same area.**

Homogeneous MPSoC; Amdahl's Law; run-time adaptive architecure; defect tolerance.

I. Introduction

The scaling of CMOS technology has increased the density and consequently made the integration of several processors in one chip possible. Although the use of multicores allows task level parallelism (TLP) exploitation, the speedup achieved by these systems is limited to the amount of parallelism available in the applications, as already foreseen by Amdahl [1].

One solution to overcome Amdahl's law and sustain the speedup of MPSoCs is the use of heterogeneous cores, where each core is specialized in different application sets. In this way, the MPSoC can accelerate not only highly parallel applications but also the sequential ones. An example of a heterogeneous architecture is the Samsung S5PC100 [2] used in the iPhone technology.

Although the use of heterogeneous cores can be an efficient solution to improve the MPSoC's performance, there are other constraints that must be considered in the design of multicore systems, such as reliability. The scaling process shrinks the wires' diameter, making them more fragile and susceptible to break. Moreover, it is also harder to keep contact integrity between wires and devices [3]. According to Borkar [4], in a 100 billion transistor device, 20 billion will fail in the manufacture and 10 billion will fail in the first year of operation.

At these high defect rates it is highly probable that the defects affect most of the processors of the MPSoC (or even all the processors), causing yield reduction and aggressively affecting the expected performance. Furthermore, in cases when all the processors are affected, this makes the MPSoC useless. To cope with this, one solution is to include some fault tolerance approach. Although there exists many solutions proposed to cope with defects [5], most of these solutions do not cope with high defect rates predicted to future technologies. Moreover, the proposed solutions present some additional cost that causes a high impact on area, power or performance, or even in all three [6].

In this context, this paper presents a reconfigurable architecture as an alternative to homogeneous multicores that allows software execution even under high defect rates, and accelerates execution of parallel and sequential applications. The architecture uses an on-line mechanism to configure itself according to the application, and its design provides acceleration in parallel as well as in sequential portions of the applications. In this way, the proposed architecture can be used to replace homogeneous MPSoCs, since it sustains performance even under high defect rates, and it is a heterogeneous approach to accelerate all kinds of applications. Therefore, its performance is not limited to the parallelism available in the applications.

To validate the architecture, we compare the performance and area of the system to a homogeneous MPSoC with equivalent area. The results indicate that the proposed architecture can sustain execution even under a 20% defect rate, while in the MPSoC all the processors become useless even under a 15% defect rate. Furthermore, with lower defect rates, the proposed architecture presents higher acceleration when compared to the MPSoC under the same defect rates, with the TLP available in the applications lower than 100%.

The rest of this paper is organized as follows. Section 2 details the adaptive system. Section 3 presents the defect tolerance approach and some experimental results. Section 4 presents a comparison of area and performance between the reconfigurable system and the equivalent homogeneous MPSoC considering different defect rates. Finally, section 5 presents the conclusions and future works.

II. Proposed Architecture

The reconfigurable system consists of a coarse-grained reconfigurable array tightly coupled to a MIPS R3000 processor; a mechanism to generate the configuration, called Binary Translator; and the context memory that stores the reconfiguration [7-8]. Figure 1 illustrates the reconfigurable system.

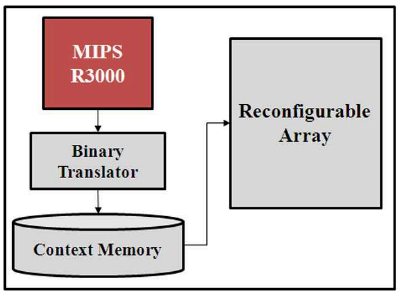


Figure 1. Reconfigurable System

The reconfigurable array consists of a combinational circuit that comprises three groups of functional units: the arithmetic and logic group, the load/store group and the multiplier group. Figure 2 presents the reconfigurable array (RA).

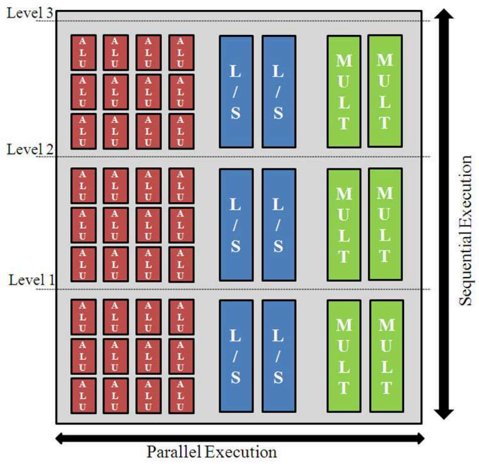


Figure 2. Reconfigurable Array

Each group of functional unit can have a different execution time, depending on the technology and implementation strategy. Based on this, in this work the ALU group can perform up to three operations in one equivalent processor cycle and the other groups execute in one equivalent processor cycle. The equivalent processor cycle is called level. Figure 2 also demonstrates the parallel and sequential paths of the reconfigurable array. The amount of functional units is defined according to area constraints and/or an application set demand (given by a certain market, e.g. portable phones).

The interconnection model implemented in the reconfigurable fabric is based on multiplexers and buses. The buses are called context lines and receive data from the context registers, which store the data from the processor's register file. The multiplexers select the correct data that will be used by each functional unit. Figure 3 illustrates the interconnection model.

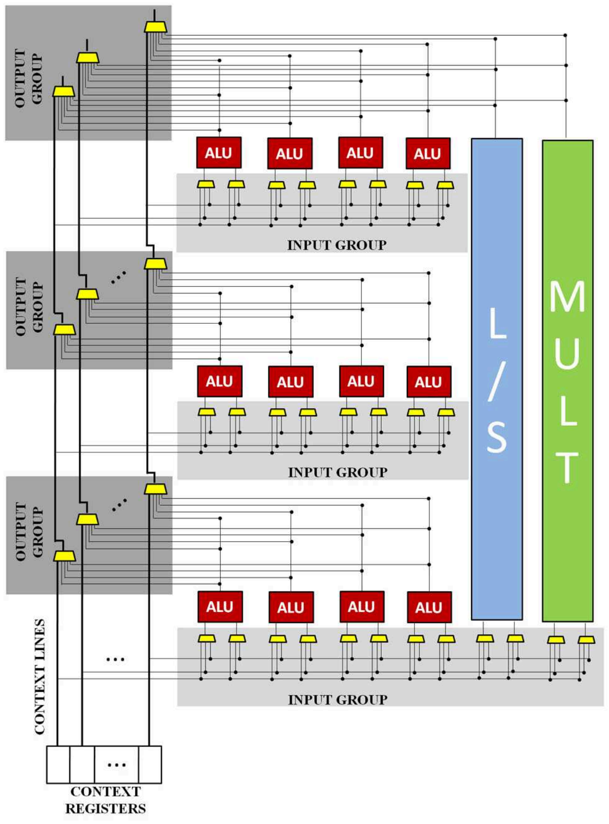
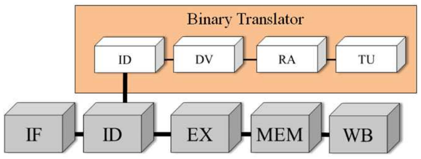


Figure 3. Interconnection Model

The different execution times presented by each group of functional units allow the execution of more than one operation per level. Therefore, the array can perform up to three arithmetic and logic operations that present data dependency among each other in one equivalent processor cycle, consequently accelerating the sequential execution. Moreover, the execution time can be improved through modifications on the functional units and with the technology evolution, consequently increasing the acceleration of intrinsically sequential parts of a code. Even non parallel code can have a better performance when executed in the structure illustrated in Figure 2, as shown in paper [7].

The Binary Translator (BT) unit implements a mechanism that dynamically transforms sequences of instruction to be executed on the array. The transformation process is transparent, with no need of instruction modification before execution, preserving the software compatibility of the application. Furthermore, the BT works in parallel with the processor's pipeline, presenting no extra overhead to the processor. Figure 4 illustrates the Binary Translator steps attached to the MIPS R3000 pipeline.



IF	Instruction Fetch	ID	Instruction Decode
ID	Instruction Decode	DV	Dependency Verification
EX	Execution	RA	Resource Allocation
MEM	Memory Access	TU	Table Update
WB	Write Back		

Figure 4. Binary Translator

A. *Configuration generation*

In parallel with the processor execution the BT searches for sequences of instructions that can be executed by the reconfigurable array (RA). The detected sequence is translated to a configuration and stored in the context memory indexed by the program counter (PC) value of the first instruction from the sequence.

During this process the BT verifies the data dependency among instructions and performs resource allocation according to data dependency and resources availability. Both data dependency and resources availability verification are performed through the management of several tables that are filled during execution. At the end of the BT stages a configuration is generated and stored in the context memory.

As mentioned before, since the BT works in parallel with the processor's pipeline, there is no overhead to generate the configuration.

B. *RA reconfiguration and execution*

While the BT generates and stores the configuration the processor continues its execution. The next time a PC from a configuration is found the processor changes to a halt stage and the respective configuration is loaded from the context memory and the RA's datapath is reconfigured. Moreover all input data is fetched. Finally, the configuration is executed and the registers and memory positions are written back.

It is important to highlight that the overhead introduced by the RA reconfiguration and data access are amortized by the acceleration achieved by the RA. Moreover, as mentioned before, the configuration generation does not impose any overhead. More details about the reconfiguration process can be found in [7].

III. DEFECT TOLERANCE

A. *Defect tolerance approach*

Reconfigurable architectures are strong candidates to defect tolerance. Since they consist essentially of identical functional elements, this regularity can be exploited as spare-parts. This is the same approach used in memory devices and has demonstrated to be very efficient [9]. Moreover, the reconfiguration capability can be exploited to change the resources allocation based on the defective and operational resources.

In addition, dynamic reconfiguration can be used to avoid the defective resources and generate the new configuration at run-time. Thus, there is no performance penalty caused by the allocation process, nor extra fabrication steps are required to correct each circuit.

Finally, as it will be shown, the capability of adaptation according to the application can be exploited to amortize the performance degradation caused by the replacement of defective resources by working ones.

Since the defect tolerance approach presented in this paper handles only defects generated in the manufacture process, the information about the defective units is generated before the reconfigurable array starts its execution, by some classical testing techniques. Therefore, the solution to provide defect tolerance is transparent to the configuration generation.

Figure 5 illustrates the defect tolerance scheme implemented in the reconfigurable array mechanism. Figure 5.a presents the resources allocation in a defect-free reconfigurable array. As already detailed, the parallel instructions are placed in the same row of the reconfigurable array and the dependent instructions, which must be executed sequentially, are placed in different rows.

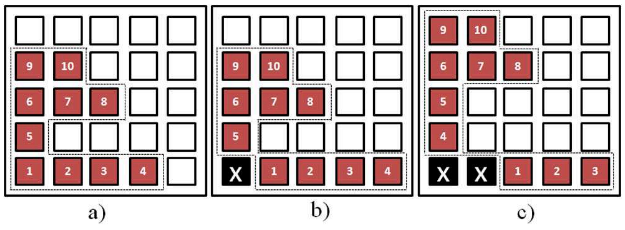


Figure 5. Resource allocation approach

To select the functional units to execute the instructions, the mechanism starts by allocating the first available unit (bottom-left). The next instructions are placed according to data dependency and resources availability. The control of available units is performed through a table that represents the reconfigurable array. The table indicates which units are available and which ones were already allocated (i.e. the units that are busy).

In Figure 5.a. the instructions 1; 2; 3; and 4 do not have data dependency among each other, hence they are all placed in the same row. On the other hand, instruction 5 has data dependency among one (or more than one) of the previous instructions. Hence, this instruction is placed in the first available unit of the second row. Continuing the allocation, instructions 6; 7 and 8 have data dependency on instruction 5. Thus, they are placed in the third row. Finally, instructions 9 and 10 have data dependency on one (or more than one) instruction from the previous row (6; 7; and 8), hence they must be placed in the fourth row.

In a defective reconfigurable array, the allocation algorithm is exactly like described above. The only difference is that to allocate only the resources that are effectively able to work, before the reconfigurable array starts and after the traditional testing steps, all the defective units are set as permanently busy in the table that controls the resource allocation process, like if they had been previously allocated. With this approach, no modification on the reconfigurable array algorithm itself is necessary.

Figure 5.b and 5.c demonstrate the resource allocation considering defective functional units. In Figure 5.b the configuration mechanism placed the instruction in the first available unit, which in this case corresponds to the second functional unit of the first row. Since the first row still has available resources to place the four instructions, the reconfigurable array sustains its execution time. In this case the presence of a defective functional unit does not affect the performance.

Figure 5.c illustrates an example where defective functional units affect the performance of the reconfigurable array. In this example, the first row has only three available functional units. In this case, when there are not enough resources in one row, the instructions are placed in the next row, and all the data dependent instructions must be moved upwards. In Figure 5.c, instruction 5 is dependent on instruction 4. Hence, instruction 5 was placed in the next row, and the same happened with other instructions (6 to 10). In this example, because of the defective units it was necessary to use one more row of the RA, consequently increasing execution time and affecting performance.

The same approach was implemented to tolerate defects in the interconnection model of Figure 3. However, the strategy can be different depending on which multiplexer is affected. If an input multiplexer is affected the strategy is to consider the multiplexer and its respective functional unit as defectives. On the other hand, if an output multiplexer is defective it is possible simply placing in the respective functional unit a different instruction that does not use the defective multiplexer.

The defect tolerance approach for functional units and interconnection model was already proposed in [10], where more details about the defect tolerance of interconnection model and experimental results can be found. These details are not in the scope of this paper, since the main focus of the current paper is to demonstrate how the reconfigurable architecture with the defect tolerance approach presented in [10] can be an efficient alternative to homogeneous multicores under high defect rates.

B. *Experimental results*

To evaluate the proposed approach and its impact on performance we have implemented the reconfigurable architecture in an architectural simulator that provided the MIPS R3000 execution trace. Furthermore, as workloads we used the MiBench Benchmark Suite [11] that contains a heterogeneous application set, including applications with different amount of parallelism.

To include defects in the reconfigurable array a tool was implemented to randomly select functional and interconnection units as defective, based on several different defect rates. The tool's input is the information about the amount of resources available in the array and its sequential and parallel distribution, as well as the defect rate. Based on this, the tool's output has the same resources information, but now with the randomly selected units set as busy. This information was used as input to the architectural simulator. In this study we used five different defect rates (0.01%; 0.1%; 1%; 10% and 20%), and the reference design was the reconfigurable array without defective units.

The size of the RA was based on several studies varying the amount of functional units and their parallel and sequential distribution. The studies considered large RAs with thousands of functional units, and also small arrays with only dozens functional units. The chosen RA is a middle-term of the studied architectures. It contains 512 functional units (384 ALUS, 96 load/stores and 32 multipliers). The area of this architecture is equivalent to 10 MIPS R3000.

It is important to highlight that despite the performance degradation presented by the reconfigurable system under a 20% defect rate, the performance was still superior to the standalone processor's performance. Hence, Figure 6 presents the acceleration degradation of the reconfigurable system, instead of the performance degradation.

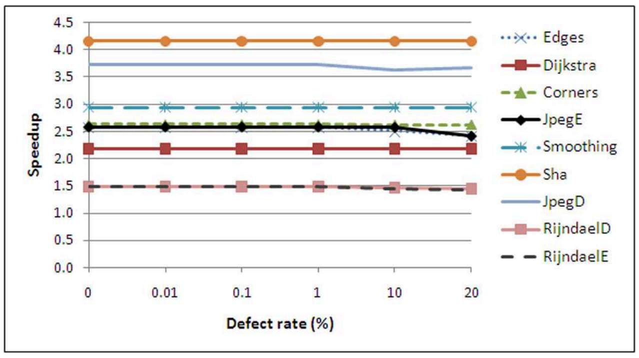


Figure 6. Acceleration degradation of the reconfigurable system

According to Figure 6, the highest acceleration penalty was presented in the execution of *jpegE*, with 6.5% of speedup reduction under a 20% defect rate. Nevertheless, the reconfigurable system is still 2.4 times faster than the standalone processor.

The mean speedup achieved by the defect-free RA in the execution of MiBench applications was 2.6 times. Under a 20% defect rate the mean speedup degraded to 2.5 times. This is less than 4% of speedup degradation.

These results demonstrate that even under a 20% defect rate, the reconfigurable array combined with the on-line reconfiguration mechanism is capable of not only ensuring that the system remains working, but it also accelerates the execution when compared to the original MIPS R3000 processor.

IV. ADAPTIVE SYSTEM X HOMOGENEOUS MPSOC

A. *Area and performance*

To demonstrate the efficiency of the proposed architecture this section presents a comparison between the adaptive system and a homogeneous MPSoC with the same area.

As mentioned before, the area of the reconfigurable system is equivalent to 10 MIPS R3000 processors, including data and instruction caches. Moreover, the mean speedup achieved by the reconfigurable system is 2.6 times for the MiBench suite.

The homogeneous MPSoC used to compare area and performance consists of ten MIPS R3000. In this analysis the communication and memory access overheads are not considered. Although the inter-processors communication is not considered, its impact would certainly be higher in the case of the MPSoC, hence all presented results are somewhat favoring the MPSoC.

As mentioned before, according to Amdahl's law, the speedup achieved by the MPSoC is limited to the execution time of the sequential portion of the application. Equation 1 repeats Amdahl's law for parallel systems:

$$Speedup(f,n)=\frac{1}{(1-f)+\frac{f}{n}} \tag{1}$$

Where f is the fraction of the application that can be parallelized and n is the number of cores.

Since the MPSoC has ten cores, by varying f and fixing n in 10 (to have the same area of the reconfigurable array, and hence normalize results by area), from Amdahl's law we obtained the results presented in Table I, where one can see the speedup as a function of f in equation (1), the part that can be parallelized.

TABLE I. ACCELERATION AS A FUNCTION OF f, n=10.

f	Speedup
0.10	1.099
0.15	1.156
0.20	1.220
0.25	1.290
0.30	1.370
0.35	1.460
0.40	1.563
0.45	1.681
0.50	1.818
0.55	1.980
0.60	2.174
0.65	2.410
0.70	2.703
0.75	3.077
0.80	3.571
0.85	4.255
0.90	5.263
0.95	6.897
0.99	9.174
1.00	10.000

Since communication and memory accesses overheads are not considered, with 10 cores it is possible to achieve a speedup of 10 times if 100% of the application is parallelized.

According to Table I it is necessary that 70% of the application be parallelized to achieve a speedup of 2.7 times, which is approximately the acceleration obtained by the reconfigurable system.

Now, one can fix the speedup and vary f to find the number of processors to achieve the required speedup. From equation 1, varying f from 0.1 to 1 we have that when $f >= 0.65$ we can achieve the speedup of 2.6. When $f < 0.65$ it is not possible to find a number of cores to achieve an acceleration of 2.6 times. Thus, even with hundreds or thousands of cores, if the application has less than 65% of parallelism, it will never achieve the speedup of 2.6, the same of the reconfigurable array. Nevertheless, with 65% it would be necessary 19 cores to achieve speedup of 2.6 times, as shown in Figure 7.

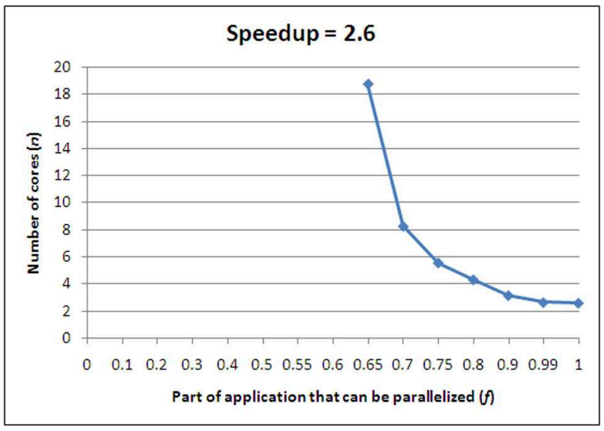


Figure 7. Number of cores as a function of f

One solution to cope with this is to improve the homogeneous core's performance to increase the speedup of the sequential execution. Therefore, one can rewrite Amdahl's law to take this into account, as it is demonstrated in equation 2. This solution was discussed in [1], where the authors presented the possible solutions to increase performance of a homogeneous MPSoC. They conclude that more investment should be done to increase the individual core performance even at high cost.

$$Speedup(f,AP,AS)=\frac{1}{\frac{1-f}{AS}+\frac{f}{AP}} \tag{2}$$

Equation (2) is an extension of Amdahl's law, and reflects the idea of improving the MPSoC overall performance by increasing core performance through acceleration of sequential portions. In equation (2), AS is the speedup of the sequential portion and AP is the speedup of the parallel portion. Table II presents values for AS, fixing the speedup in 2.6 (acceleration given by the reconfigurable array) and AP in 10 (homogeneous multicore and the reconfigurable array have the same area), while varying f. As one can see in Table II, only when f=100% that AS=0, which means that this is the only case that does not require sequential acceleration. This acceleration cannot be achieved by the homogeneous MPSoC, however as explained in section 2, the reconfigurable array can accelerate sequential portions of the application.

TABLE II. SEQUENTIAL ACCELERATION AS A FUNTION OF *f*

f	Speedup	AP	AS
0.1	2.60	10	2.402
0.2	2.60	10	2.194
0.3	2.60	10	1.974
0.4	2.60	10	1.741
0.5	2.60	10	1.494
0.6	2.60	10	1.232
0.7	2.60	10	0.954
0.9	2.60	10	0.339
0.99	2.60	10	0.035
1	2.60	10	0.000

The next section presents a comparison between the MPSoC and the RA considering the fault tolerance capability. The results are normalized by area and speedup.

B. *Defect tolerance*

To compare the performance degradation of the reconfigurable system with the MPSoC caused by the presence of defects we performed a performance simulation varying the defect rate.

To simulate the defects in both, MPSoC and reconfigurable array, a tool was implemented to randomly insert defects in the architectures. To ensure that the defect position was not affecting the results thousands of simulations were performed and in each simulation a new random set of defects was generated. Moreover, the defects generated had the same size (granularity) to both MPSoC and reconfigurable architecture.

In the first analysis we normalized RA and MPSoC by area. In the second analysis we increased the numbers of cores of the MPSoC to evaluate the tradeoff between area and fault tolerance capability.

1) *MPSoC and RA with same area:*

Figure 8 illustrates the number of cores affected by the defects in function of the defect rate in three different studies. The first analysis was performed in a homogeneous MPSoC with 10 MIPS R3000 processors without any fault tolerance approach. According to the results, when the defect rate is 15% or higher, more than 9 cores are affected. Therefore, the whole MPSoC system fails under a 15% defect rate.

The second and third analyses were performed considering that the MPSoC has some kind of fault tolerance solution implemented. In the second analysis, the fault tolerance solution consists in replicating the processor in each core. In this case, instead of having 10 cores with 10 processors, the MPSoC has 5 cores with 2 processors in each core. The second processor works as spare that is used only when the first processor fails. This solution was proposed for two main reasons. First there is no increase in area. Thus, the MPSoC still has the same area of the RA. Second, even with half of the number of cores, the MPSoC still presents higher speedup than the array when the application presents 100% of parallelism.

The solution proposed in the third analysis is also based on hardware redundancy. However, in this case instead of replicating the whole processor, only critical components of the processor are replicated, e.g. the arithmetic and logic unit. This solution presents lower area cost compared to the solution of the second analysis. However, it can be more complex to implement, since each processor must have an extra unit to implement the fault tolerance approach. Therefore, this solution considers that each processor has 3 arithmetic and logic units, where 2 ALUS are used as spare.

As can be observed in Figure 8, in both second and third analyses, under a 15% defect rate all the cores fail. This means that even with fault tolerance solutions, the MPSoC tends to fail completely at high fault rates.

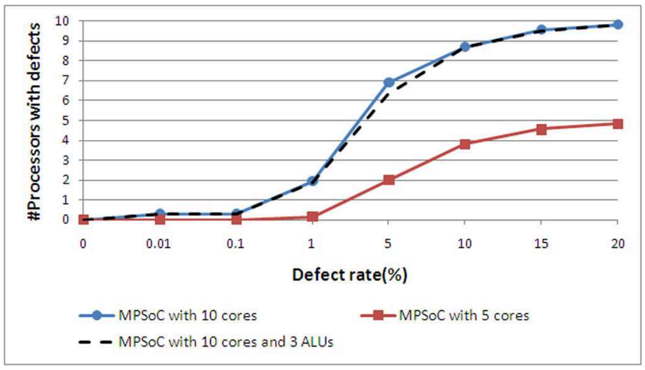


Figure 8. Defects simulation in the MPSoC

Figure 9 presents the performance degradation of the MPSoC when the number of cores is reduced due to the presence of defects. To obtain the speedup it was used the Amdahl's law represented in equation 1; the MPSoC with ten cores (without Fault Tolerance); and the one with 5 cores and 2 processors per core. Again, we considered no communication costs, and hence real results tend to be worse. The chart also presents the mean performance degradation of the reconfigurable system in the execution of the MiBench applications.

The analysis was performed using *f*=0.70 (the portion of the application that can be parallelized). This number was used because according to Table I, the speedup achieved by the MPSoC when 70% of the application can be parallelized approaches the speedup achieved by the reconfigurable system. The numbers next to the dots in the chart represents the amount of cores that are still working in the MPSoC under the defect rate.

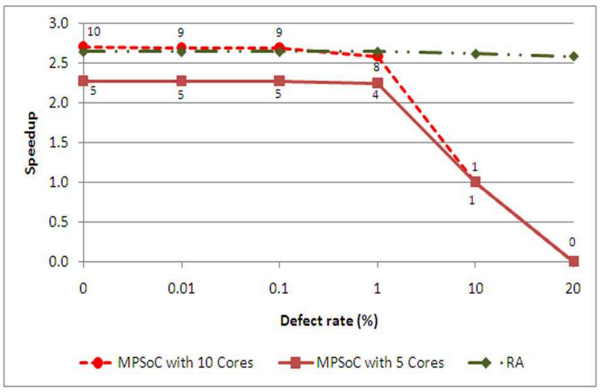


Figure 9. Performance degradation of the MPSoC

As can be observed in Figure 9, the performance of the MPSoC degrades faster than the degradation presented by the reconfigurable system, even when the MPSoC presents higher speedup in a defect-free situation (10-cores MPSoC). This happens because when a defect is placed in any part of the processor, the affected processor cannot execute properly. On the other hand, when a defect is placed in any functional unit or interconnection element of the reconfigurable array, the run-time mechanism selects another unit (or element) to replace the defective one. According to Figure 9 the execution of MiBench applications by the reconfigurable system presented less than 4% of speedup degradation even under a 20% defect rate.

It is important to highlight that in these analyses it was not considered the impact on area and performance that the implementation of the fault tolerance strategies should introduce. Again the presented results are somewhat favoring the MPSoC. Moreover, the choice of these fault tolerance solutions was based on the idea of causing the minimal impact on the area of the system to maintain both RA and MPSoC equivalent in area.

Figure 10 presents the graceful degradation of the applications *sha* and *jpegE*. These applications were selected because the first one achieved the highest speedup by the reconfigurable system among all the applications from the MiBench suite and the second presented the highest speedup degradation.

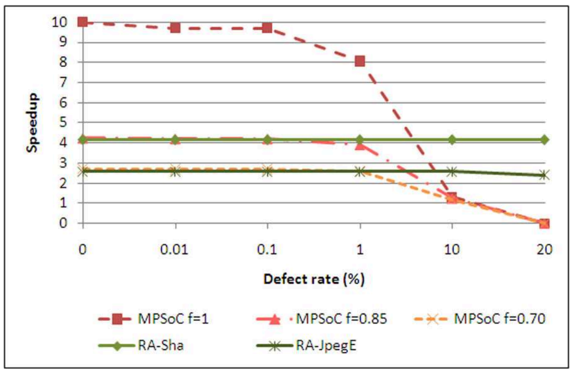


Figure 10. Graceful degradation of *sha* and *jpegE* applications

According to Figure 10 the execution of application *sha* by the reconfigurable system presented less than 1% of speedup degradation even under a 20% defect rate. On the other hand, even considering that the application was 100% parallelized (*f*=1), with an initial acceleration higher than the one achieved by the reconfigurable system, the 10-cores MPSoC stopped working under a 20% defect rate. This same behavior was observed when the amount of parallelism available was reduced (*f*=0.85 and *f*=0.70). In these cases, not only the whole system stopped working under a 20% defect rate, but the initial acceleration was equal and lower, respectively, than the one achieved by the reconfigurable system.

Moreover, as can be observed in the figure, the same behavior is presented in *jpegE* results. However, in this application execution the MPSoC presents higher speedup with *f*=0.85 than the RA that rapidly decreases to 0 when the defect rate is higher than 1%. On the other hand, the RA sustains acceleration even under a 20% defect rate that presented a speedup degradation of 6.5%.

2) *Increase MPSoC core number:*

Since the RA consists in a large amount of identical functional units that can be easily replaced, the same idea was proposed to the MPSoC: increase the number of cores to increase the reliability. Thus, this solution consists in adding more cores to the MPSoC to allow software execution under higher defect rates.

As one can observe in Figure 11, the MPSoC with 32 cores still executes under a 15% defect rate. However the execution is completely sequential (one core left under 15% defect rate). Moreover, under a 20% defect rate the 32-cores MPSoC completely fails. The speedup results presented in Figure 12 also demonstrates the rapid decrease in the MPSoC speedup even with 32 cores, while the RA sustains acceleration in both *sha* and *jpegE* even under a 20% defect rate.

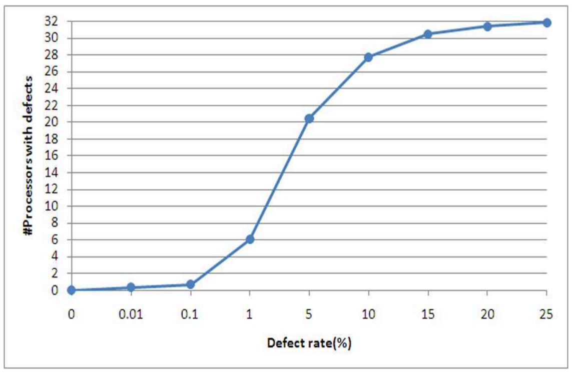


Figure 11. Defects simulation in the 32-cores MPSoC

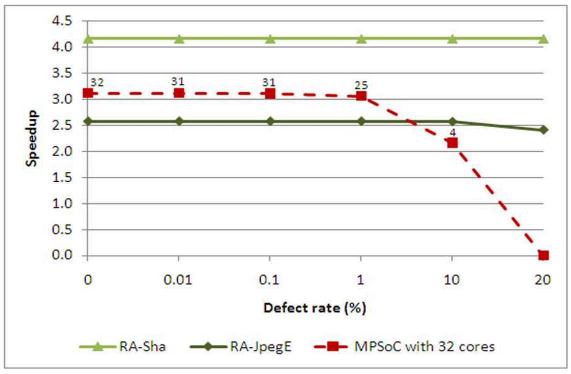


Figure 12. Graceful degradation of 32-cores MPSoC

Based on this result, one can conclude that simply replicating the cores it is not enough to increase the defect tolerance of the system to tolerate high defect rates that new technologies should introduce. Moreover, adding a fault tolerance approach can be costly in area and performance.

The analyses presented in this section demonstrate that to future technologies with high defect rates, homogeneous MPSoCs may not be the most appropriate solution. The main reasons are the fact that a defect in any part of the processor invalidates this processor. Thus, the higher is the defect rate, the more aggressive is the performance degradation, leading to a completely fail of the system under defect rates already predicted to futures technologies, such as nanowires [3]. Furthermore, solutions to provide fault tolerance in homogeneous MPSoCs under high defect rates can be costly, both in area and performance.

Another disadvantage of homogeneous MPSoCs is the fact that they can only exploit task level parallelism, depending on the parallelism available in each application. Therefore, only a specific application set that is highly parallelized can benefit from the high integration density and consequently the integration of several cores in one chip [12].

There are two main solutions to cope with this. The first one is to use heterogeneous MPSoCs, where each core can accelerate a specific application set [2]. The main problem of this solution is that like the homogeneous multicore, the heterogeneous one must also have some fault tolerance approach to cope with high fault rates, and this can increase area and performance costs.

The other possible solution is to increase the speedup of each core individually. With the improvement of each core it is possible to accelerate sequential portions of code and consequently increase the overall performance of the system. An example of this approach is to change the MIPS R3000 cores for superscalar MIPS R10000 [13]. However, this strategy can result in a significant area increase. According to [14], a MIPS R10000 is almost 29 times larger than the MIPS R3000.

The analyses also demonstrate that the proposed reconfigurable architecture ensures software execution and also accelerates the execution of several applications even under a 20% defect rate. Moreover, the reconfigurable system is a heterogeneous solution that accelerates parallel and sequential code. Thanks to this approach, the proposed architecture even exposed to high defect rates predicted to future technologies can still accelerate code, since the parallelism exploitation is not the only way to accelerate execution.

V. CONCLUSIONS

The advances on scaling of CMOS technology has increased the integration density and consequently provided the inclusion of several cores in one single chip.

The MPSoC solutions allow the acceleration of application execution through task level parallelism exploitation. However, the main problem of these solutions is the fact that they are limited to the amount of parallelism available in each application, as demonstrated by Amdahl's law.

One of the solutions to overcome this limit is using heterogeneous MPSoCs, where each core is specialized in different applications set or even increasing the speedup of each core individually. These approaches can improve performance but cannot handle high defect rates presented in future technologies.

This paper presented a run-time reconfigurable architecture that can sustain performance even under high defect rates to replace homogeneous MPSoCs solutions. The system consists of a reconfigurable array and an on-line mechanism that performs defective functional unit replacement at run-time without the need of extra tools or hardware.

The reconfigurable array design allows the acceleration of parallel and sequential portions of applications, and can be used as a heterogeneous solution to replace the homogeneous MPSoCs and ensure reliability in a highly defective environment.

To validate the proposed approach several simulations were performed to compare the performance degradation of the reconfigurable system and the MPSoC using the same defect rates, normalizing the architectures by area and speedup. According to the results, the reconfigurable system sustains execution even under a 20% defect rate, while the MPSoC with equivalent area has all the cores affected under a 15% defect rate.

Future works include analyzing power and energy of these systems and coping with transient faults in the reconfigurable system.

REFERENCES

[1] M. D. Hill, M. R. Marty, "Amdahl's Law in the Multicore Era," Computer, vol. 41, no. 7, July 2008, pp. 33-38.

[2] Samsung Electronics Co., Ltd, Samsung S5PC100 ARM Cortex A8 based Mobile Application Processor, 2009.

[3] A. DeHon and H. Naeimi, "Seven strategies for tolerating highly defective fabrication," in IEEE Design & Test, vol. 22, IEEE Press, July-Aug. 2005, pp. 306–315.

[4] S. Borkar, "Microarchitecture and Design Challenges for Gigascale Integration," keynote address, 37th Annual IEEE/ACM International Symposium on Microarchitecture, 2004.

[5] I. Koren and C. M. Krishna, "Fault-Tolerant Systems," Morgan Kaufmann, 2007.

[6] S. K. Shukla and R. I. Bahar, "Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation," Kluwer Academic Publishers, 2004.

[7] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev and L. Carro, "Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications," Proc. of Design, Automation and Test in Europe (DATE), 2008, pp. 1208-1213.

[8] A. C. S. Beck and L. Carro, "Transparent Acceleration of Data Dependent Instructions for General Purpose Processors," In International Conference on Very Large Scale Integration, 2007, pp. 66-71.

[9] E. Scott, P. Sedcole and P. Y. K. Cheung, "Fault Tolerant Methods for Reliability in FPGAs," In International Field Programmable Logic and Applications, Sept. 2008, pp. 415-420.

[10] M. M. Pereira and L. Carro, "A Dynamic Reconfiguration Approach for Accelerating Highly Defective Processors," Proc. of 17th IFIP/IEEE International Conference On Very Large Scale Integration, Oct. 2009.

[11] M. R. Guthaus, et al, "MiBench: a free commercially, representative embedded benchmark suite," Proc. of 4th IEEE International Workshop on Workload Characterization, IEEE Press, Dec. 2001, pp. 3-14.

[12] K. Olukotun, L. Hammond and J. Laudon, "Chip Multiprocessor Architecture," Mark D. Hill, 2006.

[13] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996, pp.28-40.

[14] M. B Rutzig, et al, "TLP and ILP exploitation throuhg a Reconfigurable Multiprocessor System," In 17th Reconfigurable Architectures Workshop, Atlanta, USA, 2010.

An NoC Traffic Compiler for efficient FPGA implementation of Parallel Graph Applications

Nachiket Kapre
California Institute of Technology,
Pasadena, CA 91125
nachiket@caltech.edu

André DeHon
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

***Abstract*—Parallel graph algorithms expressed in a Bulk-Synchronous Parallel (BSP) compute model generate highly-structured communication workloads from messages propagating along graph edges. We can expose this structure to traffic compilers and optimization tools before runtime to reshape and reduce traffic for higher performance (or lower area, lower energy, lower cost). Such offline traffic optimization eliminates the need for complex, runtime NoC hardware and enables lightweight, scalable FPGA NoCs. In this paper, we perform load balancing, placement, fanout routing and fine-grained synchronization to optimize our workloads for large networks up to 2025 parallel elements. This allows us to demonstrate speedups between 1.2× and 22× (3.5× mean), area reductions (number of Processing Elements) between 3× and 15× (9× mean) and dynamic energy savings between 2× and 3.5× (2.7× mean) over a range of real-world graph applications. We expect such traffic optimization tools and techniques to become an essential part of the NoC application-mapping flow.**

I. Introduction

Real-world communication workloads exhibit structure in the form of locality, sparsity, fanout distribution, and other properties. If this structure can be exposed to automation tools, we can reshape and optimize the workload to improve performance, lower area and reduce energy. In this paper, we develop a traffic compiler that exploits structural properties of Bulk-Synchronous Parallel communication workloads. This compiler provides insight into performance tuning of communication-intensive parallel applications. The performance and energy improvements made possible by the compiler allows us to build the NoC from simple hardware elements that consume less area and eliminate the need for using complex, area-hungry, adaptive hardware. We now introduce key structural properties exploited by our traffic compiler.

- When the natural communicating components of the traffic do not match the granularity of the NoC architecture, applications may end up being poorly load balanced. We discuss *Decomposition* and *Clustering* as techniques to improve load balance.
- Most application exhibit sparsity and locality; an object often interacts regularly with only a few other objects in its neighborhood. We exploit these properties by *Placing* communicating objects close to each other.
- Data updates from an object should often be seen by multiple neighbors, meaning the network must route the same message to multiple destinations. We consider *Fanout Routing* to avoid redundantly routing data.
- Finally, applications that use barrier synchronization can minimize node idle time induced by global synchronization between the parallel regions of the program by using *Fine-Grained Synchronization*.

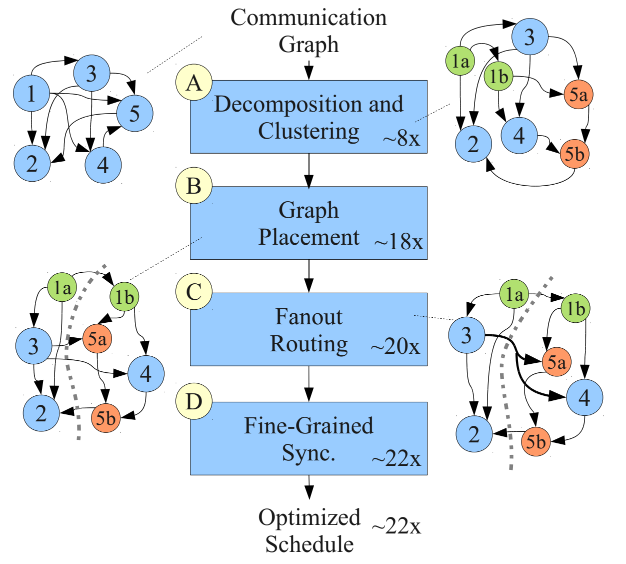


Fig. 1: NoC Traffic Compilation Flow
(annotated with `cnet-default` workload at 2025 PEs)

While these optimizations have been discussed independently in the literature extensively (*e.g.* [1], [2], [3], [4], [5]), we develop a toolflow that auto-tunes the control parameters of these optimizations per workload for maximum benefit and provide a quantification of the cumulative benefit of applying these optimizations to various applications in onchip network settings. This quantification further illustrates how the performance impact of each optimization changes with NoC size. The key contributions of this paper include:

- Development of a traffic compiler for applications described using the BSP compute model.
- Use of communication workloads extracted from ConceptNet, Sparse Matrix-Multiply and Bellman-Ford running on range of real-world circuits and graphs.
- Quantification of cumulative benefits of each stage of the compilation flow (performance, area, energy).

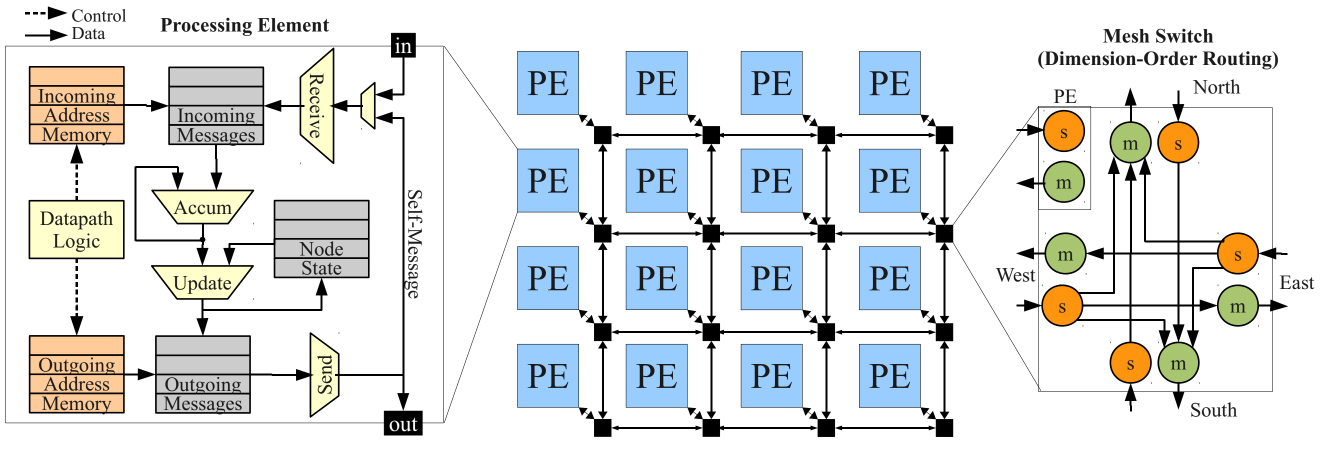


Fig. 2: Architecture of the NoC

II. Background

A. Application

Parallel graph algorithms are well-suited for concurrent processing on FPGAs. We describe graph algorithms in a Bulk Synchronous Parallel (BSP) compute model [6] and develop an FPGA system architecture [7] for accelerating such algorithms. The compute model defines the intended semantics of the algorithm so we know which optimizations preserve the desired meaning while reducing NoC traffic. The graph algorithms are a sequence of steps where each step is separated by a global BSP barrier. In each step, we perform parallel, concurrent operations on nodes of a graph data-structure where all nodes send messages to their neighbors while also receiving messages. The graphs in these algorithms are known when the algorithm starts and do not change during the algorithm. Our communication workload consists of routing a set of messages between graph nodes. We route the same set of messages, corresponding to the graph edges, in each epoch. Applications in the BSP compute model generate traffic with many communication characteristics (*e.g.* locality, sparsity, multicast) which also occur in other applications and compute models as well. Our traffic compiler exploits the *a priori* knowledge of structure-rich communication workloads (see Section IV-A) to provide performance benefits. Our approach differs from some recent NoC studies that use statistical traffic models (*e.g.* [9], [10], [11], [12]) and random workloads (*e.g.* [13], [14], [15]) for analysis and experiments. Statistical and random workloads may exaggerate traffic requirements and ignore application structure leading to overprovisioned NoC resources and missed opportunities for workload optimization.

In [9], the authors demonstrate a 60% area reduction along with an 18% performance improvement for well-behaved workloads. In [11], we observe a 20% reduction in buffer sizes and a 20% frequency reduction for an MPEG-2 workload. In [13], the authors deliver a 23.1% reduction in time, a 23% reduction in area as well as a 38% reduction in energy for their design. We demonstrate better performance, lower area requirements and lower energy consumption (Section V).

B. Architecture

We organize our FPGA NoC as a bidirectional 2D-mesh [16] with a packet-switched routing network as shown in Figure 2. The application graph is distributed across the Processing Elements (PEs) which are specialized to process graph nodes. Each PE stores a portion of the graph in its local on-chip memory and performs accumulate and update computations on each node as defined by the graph algorithm. The PE is internally pipelined and capable of injecting and receiving a new packet in each cycle. The switches implement a simple Dimension-Ordered Routing algorithm [21] and also support fully-pipelined operation using composable *Split* and *Merge* units. We discuss additional implementation parameters in Section IV-B. Prior to execution, the traffic compiler is responsible for allocating graph nodes to PEs. During execution, the PE iterates through all local nodes and generates outbound traffic that is routed over the packet-switched network. Inbound traffic is stored in the incoming message buffers of each PE. The PE can simultaneously handle incoming and outgoing messages. Once all messages have been received, a barrier is detected using a global reduce tree (a bit-level AND-reduce tree). The graph application proceeds through multiple global barriers until the algorithm terminates. We measure network performance as the number of cycles required for one epoch between barriers, including both computation and all messages routing.

III. Optimizations

In this section, we describe a set of optimizations performed by our traffic compiler.

1) Decomposition: Ideally for a given application, as the PE count increases, each PE holds smaller and smaller portions of the workload. For graph-oriented workloads, unusually large nodes with a large number of edges (*i.e.* nodes that

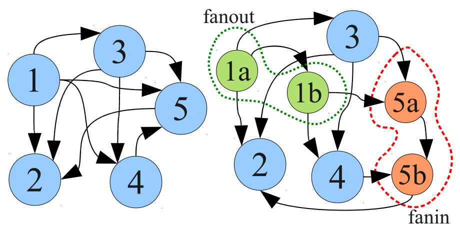


Fig. 3: *Decomposition*

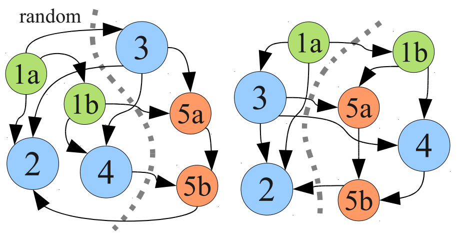


Fig. 4: *Placement*
(Random Placement vs. Good Placement)

send and receive many messages) can prevent the smooth distribution of the workload across the PEs. As a result, performance is limited by the time spent sending and receiving messages at the largest node (streamlined message processing in the PEs implies work $\propto$ number of messages per node). *Decomposition* is a strategy where we break down large nodes into smaller nodes (either inputs, outputs or both can be decomposed) and distribute the work of sending and receiving messages at the large node over multiple PEs. The idea is similar to that used in synthesis and technology mapping of logic circuits [1]. Fig. 3 illustrates the effect of decomposing a node. Node 5 with 3 inputs gets *fanin-decomposed* into Node 5a and 5b with 2 inputs each thereby reducing the serialization at the node from 3 cycles to 2. Similarly, Node 1 with 4 outputs is *fanout-decomposed* into Node 1a and 1b with 3 outputs and 2 outputs each. Greater benefits can be achieved with higher-fanin/fanout nodes (see Table I).

In general, when the output from the graph node is a result which must be multicast to multiple outputs, we can easily build an output fanout tree to decompose output routing. However, input edges to a graph node can only be decomposed when the operation combining inputs is associative. ConceptNet and Bellman-Ford (discussed later in Section IV-A) permit input decomposition since nodes perform simple integer *sum* and *max* operations which are associative and can be decomposed. However, Matrix Multiply nodes perform non-associative *floating-point accumulation* over incoming values which cannot be broken up and distributed

2) Clustering: While *Decomposition* is necessary to break up large nodes, we may still have an imbalanced system if we randomly place nodes on PEs. Random placement fails to account for the varying amount of work performed per node. Lightweight *Clustering* is a common technique used to quickly distribute nodes over PEs to achieve better load balance (*e.g.* [2]). We use a greedy, linear-time *Clustering* algorithm similar to the *Cluster Growth* technique from [2]. We start by creating as many "clusters" as PEs and randomly assign a seed node to each cluster. We then pick nodes from the graph and greedily assign them to the PE that least increases cost. The cost function ("Closeness metric" in [2]) is chosen to capture the amount of work done in each PE including sending and receiving messages.

3) Placement: Object communication typically exhibits locality. A random placement ignores this locality resulting in more traffic on the network. Consequently, random placement imposes a greater traffic requirement which can lead to poor performance, higher energy consumption and inefficient use of network resources. We can *Place* nodes close to each other to minimize traffic requirements and get better performance than random placement. The benefit of performing placement for NoCs has been discussed in [3]. Good placement reduces both the number of messages that must be routed on the network and the distance which each message must travel. This decreases competition for network bandwidth and lowers the average latency required by the messages. Fig. 4 shows a simple example of good *Placement*. A random partitioning of the application graph may bisect the graph with a cut size of 6 edges (*i.e.* 6 messages must cross the chip bisection). Instead, a high-quality partitioning of the graph will find a better cut with size of 4. The load on the network will be reduced since 2 fewer messages must cross the bisection. In general, *Placement* is an NP-complete problem, and finding an optimal solution is computationally intensive. We use a fast multi-level partitioning heuristic [17] that iteratively clusters nodes and moves the clustered nodes around partitions to search for a better quality solution.

4) Fanout Routing: Some applications may require multicast messages (*i.e.* single source, multiple destinations). Our application graphs contain nodes that send the exact same message to their destinations. Routing redundant messages is a waste of network resources. We can use the network more efficiently with *Fanout Routing* which avoids routing redundant messages. This has been studied extensively by Duato *et al.* [4]. If many destination nodes reside in the same physical PE, it is possible to send only one message instead of many, duplicate messages to the PE. For this to work, there needs to be at least two sink nodes in any destination PE. The PE will then internally distribute the message to the intended recipients. This is shown in Fig. 5. The fanout edge from Node 3 to Node 5a and Node 4 can be replaced with a shared edge as shown. This reduces the number of messages crossing the bisection by 1. This optimization works best at reducing traffic and message-injection costs at low PE counts. As PE counts increase we have more possible destinations for the outputs and fewer shareable nodes in the PEs resulting in decreasing benefits.

5) Fine-Grained Synchronization: In parallel programs with multiple threads, synchronization between the threads is sometimes implemented with a global barrier for simplicity. However, the global barrier may artificially serialize computation. Alternately, the global barrier can be replaced

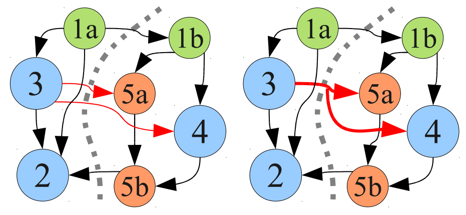


Fig. 5: *Fanout-Routing*

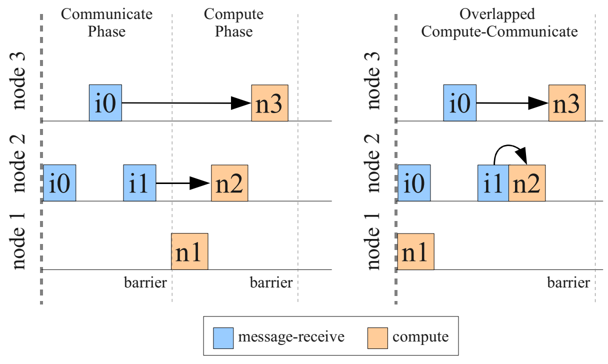


Fig. 6: *Fine-Grained Synchronization*

with local synchronization conditions that avoid unnecessary sequentialization. Techniques for eliminating such barriers have been previously studied [18], [5]. In the BSP compute model discussed in Section II, execution is organized as a series of parallel operations separated by barriers. We use one barrier to signify the end of the communicate phase and another to signify the end of the compute phase. If it is known prior to execution that the entire graph will be processed, the first barrier can be eliminated by using local synchronization operations. A node can be permitted to start the compute phase as soon as it receives all its incoming messages without waiting for the rest of the nodes to have received their messages. This prevents performance from being limited by the sum of worst-case compute and communicate latencies when they are not necessarily coupled. We show the potential benefit of *Fine-Grained Synchronization* in Fig. 6. Node 2 and Node 3 can start their *Compute* phases after they have received all their inputs messages. They do not need to wait for all other nodes to receive all their messages. This optimization enables the *Communicate* phase and the *Compute* phase to be overlapped.

IV. Experimental Setup

A. Workloads

We generate workloads from a range of applications mapped to the BSP compute model. We choose applications that cover different domains including AI, Scientific Computing and CAD optimization that exhibit important structural properties.

1) ConceptNet: ConceptNet [19] is a common-sense reasoning knowledge base described as a graph, where nodes represent concepts and edges represent semantic relationships. Queries to this knowledge base start a *spreading-activation* algorithm from an initial set of nodes. The computation spreads over larger portions of the graph through a sequence of steps by passing messages from activated nodes to their neighbors. In the case of complex queries or multiple simultaneous queries, the entire graph may become activated after a small number of steps. We route all the edges in the graph representing this worst-case step. In [7], we show a per-FPGA speedup of 20× compared to a sequential implementation.

TABLE I: Application Graphs

Graph	Nodes	Edges	Max Fanin	Max Fanout
ConceptNet				
`cnet-small`	14556	27275	226	2538
`cnet-default`	224876	553837	16176	36562
Matrix-Multiply				
`add20`	2395	17319	124	124
`bcsstk11`	1473	17857	27	30
`fidap035`	19716	218308	18	18
`fidapm37`	9152	765944	255	255
`gemat11`	4929	33185	27	28
`memplus`	17758	126150	574	574
`rdb32001`	3200	18880	6	6
`utm5940`	5940	83842	30	20
Bellman-Ford				
`ibm01`	12752	36455	33	93
`ibm05`	29347	97862	9	109
`ibm10`	69429	222371	137	170
`ibm15`	161570	529215	267	196
`ibm16`	183484	588775	163	257
`ibm18`	210613	617777	85	209

2) Matrix-Multiply: Iterative Sparse Matrix-Vector Multiply (SMVM) is the dominant computational kernel in several numerical routines (*e.g.* Conjugate Gradient, GMRES). In each iteration a set of dot products between the vector and matrix rows is performed to calculate new values for the vector to be used in the next iteration. We can represent this computation as a graph where nodes represent matrix rows and edges represent the communication of the new vector values. In each iteration messages must be sent along all edges; these edges are multicast as each vector entry must be sent to each row graph node with a non-zero coefficient associated with the vector position. We use sample matrices from the Matrix Market benchmark [20]. In [8], we show a speedup of 2-3× over optimized sequential implementation using an older generation FPGA and a performance-limited ring topology.

3) Bellman-Ford: The Bellman-Ford algorithm solves the single-source shortest-path problem, identifying any negative edge weight cycles, if they exist. It finds application in CAD optimizations like Retiming, Static Timing Analysis and FPGA Routing. Nodes represent gates in the circuit while edges represent wires between the gates. The algorithm simply relaxes all edges in each step until quiescence. A relaxation consists of computing the minimum at each node over all weighted incoming message values. Each node then communicates the result of the minimum to all its neighbors to prepare for the next relaxation.

B. NoC Timing and Power Model

All our experiments use a single-lane, bidirectional-mesh topology that implements a Dimension-Ordered Routing function. The Matrix-Multiply network is 84-bits wide while Con-

TABLE II: NoC Timing Model

Mesh Switch	Latency
$T_{through}$ (X-X, Y-Y)	2
T_{turn} (X-Y, X-Y)	4
$T_{inteface}$ (PE-NoC, NoC-PE)	6
T_{wire}	2
Processing Element	Latency
T_{send}	1
$T_{receive}$ (ConceptNet, Bellman-Ford)	1
$T_{receive}$ (Matrix-Multiply)	9

TABLE III: NoC Dynamic Power Model

Datawidth (Application)	**Block**	Dynamic Power at diff. activity (mW)				
		0%	25%	50%	75%	100%
52 (ConceptNet, Bellman-Ford)	Split	0.26	1.07	1.45	1.65	1.84
	Merge	0.72	1.58	2.1	2.49	2.82
84 (Matrix-Multiply)	Split	0.32	1.35	1.78	2.02	2.26
	Merge	0.9	1.87	2.45	2.88	3.25

ceptNet and Bellman-Ford networks are 52-bits wide (with 20-bits of header in each case). The switch is internally pipelined to accept a new packet on each cycle (see Figure 2). Different routing paths take different latencies inside the switch (see Table II). We pipeline the wires between the switches for high performance (counted in terms of cycles required as T_{wire}). The PEs are also pipelined to start processing a new edge every cycle. ConceptNet and Bellman-Ford compute simple sum and max operations while Matrix-Multiply performs floating-point $accumulation$ on the incoming messages. Each computation on the edge then takes 1 or 9 cycles of latency to complete (see Table II). We estimate dynamic power consumption in the switches using XPower [22]. Dynamic power consumption at different switching activity factors is shown in Table III. We extract switching activity factor in each Split and Merge unit from our packet-switched simulator. When comparing dynamic energy, we multiply dynamic power with simulated cycles to get energy. We generate bitstreams for the switch and PE on a modern Xilinx Virtex-5 LX110T FPGA [22] to derive our timing and power models shown in Table II and Table III.

C. Packet-Switched Simulator

We use a Java-based cycle-accurate simulator that implements the timing model described in Section IV-B for our evaluation. The simulator models both computation and communication delays, simultaneously routing messages on the NoC and performing computation in the PEs. Our results in Section V report performance observed on cycle-accurate simulations of different circuits and graphs. The application graph is first transformed by a, possibly empty, set of optimizations from Section III before being presented to the simulator.

V. Evaluation

We now examine the impact of the different optimizations on various workloads to quantify the cumulative benefit of our traffic compiler. We order the optimization appropriately to analyze their additive impacts. First we load balance our workloads by performing *Decomposition*. We then determine how the workload gets distributed across PEs using *Clustering* or *Placement*. Finally, we perform *Fanout Routing* and *Fine-Grained Synchronization* optimizations. We illustrate scaling trends of individual optimizations using a single illustrative workload for greater clarity. At the end, we show cumulative data for all benchmarks together.

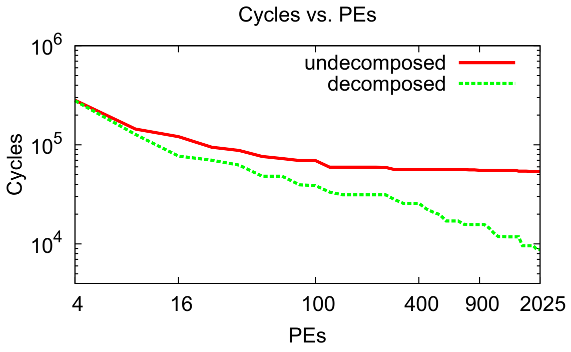


Fig. 7: *Decomposition* (`cnet-default`)

A. Impact of Individual Optimizations

1) Decomposition: In Fig. 7, we show how the ConceptNet `cnet-default` workload scales with increasing PE counts under *Decomposition*. We observe that, *Decomposition* allows the application to continue to scale up to 2025 PEs and possibly beyond. Without *Decomposition*, performance quickly runs into a serialization bottleneck due to large nodes as early as 100 PEs. The decomposed NoC workload manages to outperform the undecomposed case by 6.8× in performance. However, the benefit is lower at low PE counts, since the maximum logical node size becomes small compared to the average work per PE. Additionally, decomposition is only useful for graphs with high degree (see Table I). In Figure 8 we show how the *decomposition limit* control parameter impacts the scaling of the workload. As expected, without decomposition, performance of the workload saturates beyond 32 PEs. Decomposition with a limit of 16 or 32 allows the workload to scale up to 400 PEs and provides a speedup of 3.2× at these system sizes. However, if we attempt an aggressive decomposition with a limit of 2 (all decomposed nodes allowed to have a fanin and fanout of 2) performance is actually worse than undecomposed case between 16 and 100 PEs and barely better at larger system sizes. At such small decomposition limits, performance gets worse due to an excessive increase in the workload size (*i.e.* number of edges in the graph). Our traffic compiler sweeps the design space and automatically selects the best decomposition limit.

2) Clustering: In Fig. 9, we show the effect of *Clustering* on performance with increasing PE counts. *Clustering* provides an improvement over *Decomposition* since it accounts for compute and message injection costs accurately, but that improvement is small (1%–18%). Remember from Section III, that *Clustering* is a lightweight, inexpensive optimization that

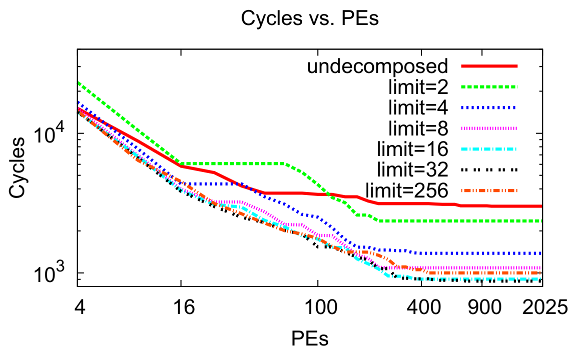


Fig. 8: *Decomposition Limits* (`cnet-small`)

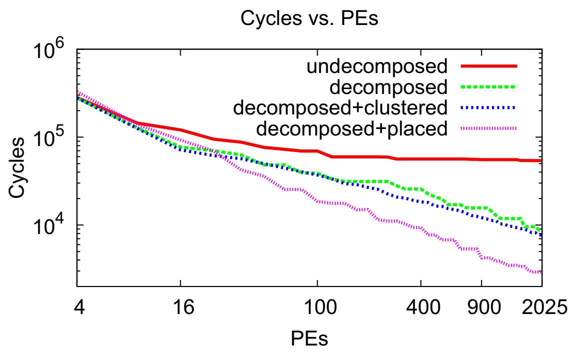


Fig. 10: *Decomposition*, *Clustering* and *Placement* (`cnet-default`)

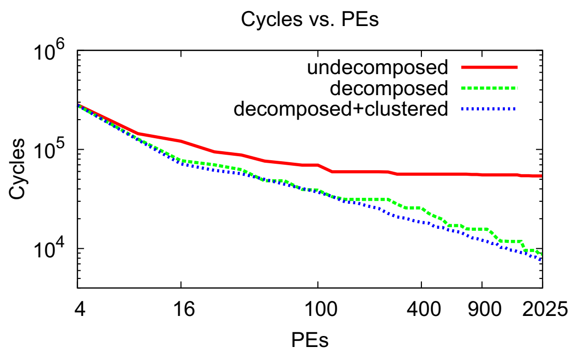


Fig. 9: *Decomposition* and *Clustering* (`cnet-default`)

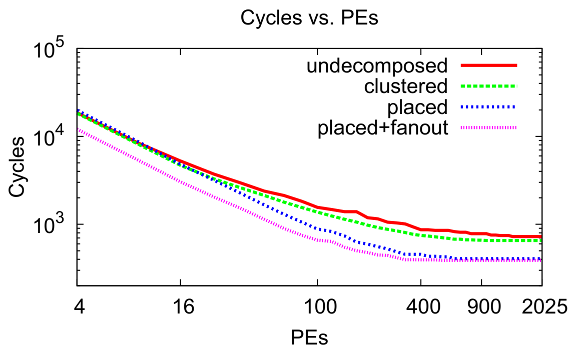


Fig. 11: *Clustering*, *Placement* and *Fanout-Routing* (`ibm01`)

attempts to improve load balance and as a result, we expect limited benefits.

3) Placement: In Fig. 10, we observe that *Placement* provides as much as 2.5× performance improvement over a random placed workload as PE counts increase. At high PE counts, localized traffic reduces bisection bottlenecks and communication latencies. However, *Placement* is less effective at low PE counts since the NoC is primarily busy injecting and receiving traffic and NoC latencies are small and insignificant. Moreover, good load-balancing is crucial for harnessing the benefits of a high-quality placement (See Figure 15 with other benchmarks).

4) Fanout-Routing: We show performance scaling with increasing PEs for the Bellman-Ford `ibm01` workload using *Fanout Routing* in Fig. 11. The greatest performance benefit (1.5×) from *Fanout Routing* comes when redundant messages distributed over few PEs can be eliminated effectively. The absence of benefit at larger PE counts is due to negligible shareable edges as we suggested in Section III.

5) Fine-Grained Synchronization: In Fig. 12, we find that the benefit of *Fine-Grained Synchronization* is greatest (1.6×) at large PE counts when latency dominates performance. At low PE counts, although NoC latency is small, elimination of the global barrier enables greater freedom in scheduling PE operations and consequently we observe a non-negligible improvement (1.2×) in performance. Workloads with a good balance between communication time and compute time will achieve a significant improvement from fine-grained synchronization due to greater opportunity for overlapped execution.

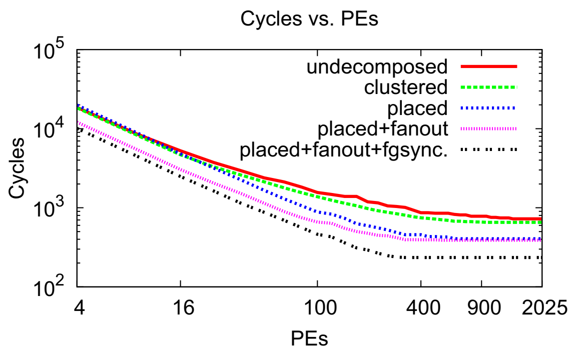


Fig. 12: *Clustering*, *Placement*, *Fanout-Routing* and *Fine-Grained Synchronization* (`ibm01`)

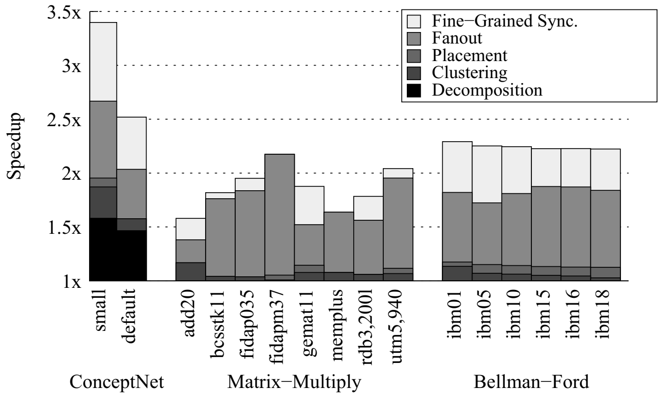


Fig. 13: Performance Ratio at 25 PEs

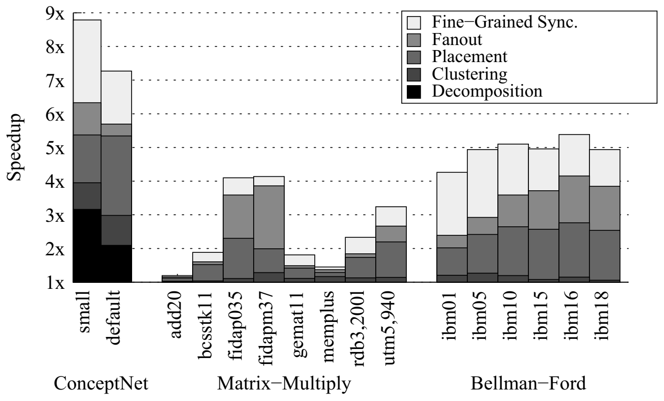


Fig. 14: Performance Ratio at 256 PEs

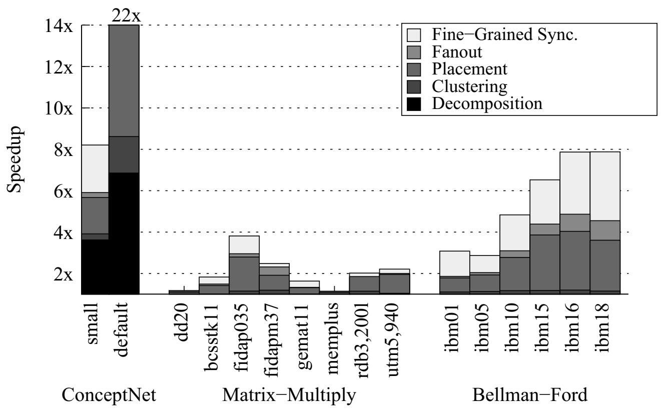


Fig. 15: Performance Ratio at 2025 PEs

B. Cumulative Performance Impact

We look at cumulative speedup contributions and relative scaling trends of all optimizations for all workloads at 25 PEs, 256 PEs and 2025 PEs.

At 25 PEs, Fig. 13, we observe modest speedups in the range 1.5× to 3.4× (2× mean) which are primarily due to *Fanout Routing*. *Placement* and *Clustering* are unable to contribute significantly since performance is dominated

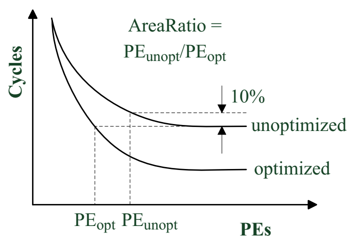


Fig. 16: How we compute area savings

by computation. *Fine-Grained Synchronization* also provides some improvement, but as we will see, its relative contribution increases with PE count.

At 256 PEs, Fig. 14, we observe larger speedups in the range 1.2× to 8.3× (3.5× mean) due to *Placement*. At these PE sizes, the performance bottleneck begins to shift to the network, so reducing traffic on the network has a larger impact on overall performance. We continue to see performance improvements from *Fanout Routing* and *Fine-Grained Synchronization*.

At 2025 PEs, Fig. 15, we observe an increase in speedups in the range 1.2× to 22× (3.5× mean). While there is an improvement in performance from *Fine-Grained Synchronization* compared to smaller PE cases, the modest quantum of increase suggests that the contributions from other optimizations are saturating or reducing.

Overall, we find ConceptNet workloads show impressive speedups up to 22×. These workloads have decomposable nodes that allow better load-balancing and have high-locality. They are also the only workloads which have the most need for *Decomposition*. Bellman-Ford workloads also show good overall speedups as high as 8×. These workloads are circuit graphs and naturally have high-locality and fanout. Matrix-Multiply workloads are mostly unaffected by these optimization and yield speedups not exceeding 4× at any PE count. This is because the compute phase dominates the communicate phase; compute requires high latency (9 cycles/edge from Table II) floating-point operations for each edge. It is also not possible to decompose inputs due to the non-associativity of the floating-point accumulation. As an experiment, we decomposed both inputs and outputs of the `fidapm37` workload at 2025 PEs and observed an almost 2× improvement in performance.

C. Cumulative Area and Energy Impact

For some low-cost applications (*e.g.* embedded) it is important to minimize NoC implementation area and energy. The optimizations we discuss are equally relevant when cost is the dominant design criteria.

To compute the area savings, we pick the smallest unoptimized PE count that requires 1.1× the cycles of best unoptimized case (the 10% slack accounts for diminishing returns at larger PE counts (see Figure 16). For the fully optimized workload, we identify the PE count that yields

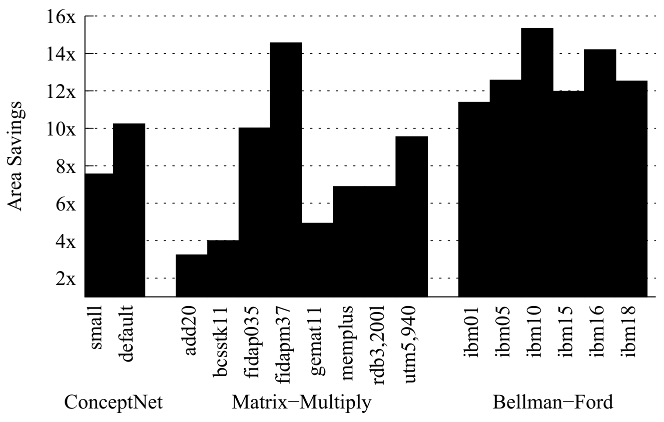


Fig. 17: Area Ratio to Baseline

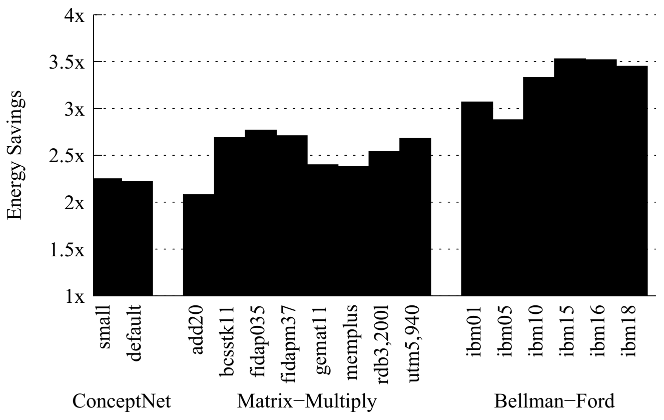


Fig. 18: Dynamic Energy Savings at 25 PEs

performance equivalent to the best unoptimized case. We report these area savings in Figure 17. The ratio of these two PE counts is 3–15 (mean of 9), suggesting these optimizations allow much smaller designs.

To compute energy savings, we use the switching activity factor and network cycles to derive dynamic energy reduction in the network. Switching activity factor is extracted from the number of packets traversing the *Split* and *Merge* units of a Mesh Switch over the duration of the simulation $Activity = (2/Ports) \times (Packets/Cycles)$. In Figure 18 we see a mean 2.7× reduction in dynamic energy at 25 PEs due to reduced switching activity of the optimized workload. While we only show dynamic energy savings at 25 PEs, we observed even higher savings at larger system sizes.

VI. Conclusions and Future Work

We demonstrate the effectiveness of our traffic compiler over a range of real-world workloads with performance improvements between 1.2× and 22× (3.5× mean), PE count reductions between 3× and 15× (9× mean) and dynamic energy savings between 2× and 3.5× (2.7× mean). For large workloads like `cnet-default`, our compiler optimizations were able to extend scalability to 2025 PEs. We observe that the relative impact of our optimizations changes with system size (PE count) and our automated approach can easily adapt to different system sizes. We find that most workloads benefit from *Placement* and *Fine-Grained Synchronization* at large PE counts and from *Clustering* and *Fanout Routing* at small PE counts. The optimizations we describe in this paper have been used for the SPICE simulator compute graphs which are different from the BSP compute model. Similarly we can extend this compiler to support an even larger space of automated traffic optimization algorithms for different compute models.

References

[1] R. K. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *Proc. Intl. Symp. on Circuits and Systems*, 1982.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[3] D. GreenField, A. Banerjee, J. G. Lee, and S. Moore, "Implications of Rent's rule for NoC design and its fault tolerance," in *NOCS First International Symposium on Networks-on-Chip*, 2007.

[4] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Enginering Approach*. Elsevier, 2003.

[5] D. Yeung and A. Agarwal, "Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient," *SIGPLAN Notices*, vol. 28, no. 7, pp. 187–192, 1993.

[6] L. G. Valiant, "A bridging model for parallel computation," *CACM*, vol. 33, no. 8, pp. 103–111, August 1990.

[7] M. deLorimier, N. Kapre, A. DeHon, et al "GraphStep: a system architecture for Sparse-Graph algorithms," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 143–151.

[8] M. deLorimier, and A. DeHon "Floating-point sparse matrix-vector multiply for FPGAs" in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 2005.

[9] W. Ho and T. Pinkston, "A methodology for designing efficient on-chip interconnects on well-behaved communication patterns," in *Proc. Intl. Symp. on High-Perf. Comp. Arch.*, 2006.

[10] V. Soteriou, H. Wang, , and L.-S. Peh, "A statistical traffic model for on-chip interconnection networks," in *Proc. Intl. Symp. on Modeling, Analysis, and Sim. of Comp. and Telecom. Sys.*, 2006.

[11] Y. Liu, S. Chakraborty, and W. T. Ooi, "Approximate VCCs: a new characterization of multimedia workloads for system-level MPSoC design," *DAC*, pp. 248–253, June 2005.

[12] G. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Trans. VLSI Syst.*, vol. 12, no. 1, pp. 108–119, January 2004.

[13] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proc. Intl. Conf. Supercomput.*, 2006.

[14] R. Mullins, A. West, and S. Moore, "Low-latency virtual-channel routers for on-chip networks," in *ISCA*, 2004.

[15] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 1025–1040, August 2005.

[16] N. Kapre, N. Mehta, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216.

[17] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

[18] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," *SIGPLAN Not.*, vol. 30, no. 8, pp. 144–155, 1995.

[19] H. Liu and P. Singh, "ConceptNet – A Practical Commonsense Reasoning Tool-Kit," *BT Technical Journal*, vol. 22, no. 4, p. 211, October 2004.

[20] NIST, "Matrix market," <http://math.nist.gov/MatrixMarket/>, June 2004, maintained by: National Institute of Standards and Technology.

[21] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, 1993.

[22] *The Programmable Logic Data Book-CD*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, 2005.

Investigation of Digital Sensors for Variability Characterization on FPGAs

Florent Bruguier, Pascal Benoit and Lionel Torres
LIRMM, CNRS - University of Montpellier 2
161 rue Ada, 34392 Montpellier, France
Email: {firstname.lastname@lirmm.fr}

***Abstract*—In this paper, we address the variability problems in FPGA devices both at design-time and at run-time. We consider a twofold approach to compensate variations, based on measurements issued from digital sensors implemented with FPGA building blocks. We compare two digital structures of sensors; the Ring Oscillator and the Path Delay Sensor. The Ring Oscillator allows a fine grain variability characterization thanks to a tiny replicable hard-macro structure (2 Slices). Although less accurate, we show that the Path Delay Sensor has a lower total area overhead as well as a smaller latency compared to the ring oscillator implementation. In our experiments conducted on Spartan-3 FPGAs, the ring oscillator is used to perform intra-chip cartographies (1980 positions), while both sensors are then compared for characterizing inter-chip performance variations. We conclude the two structures are efficient for fast variability characterization in FPGA devices. The Ring Oscillator is the best structure for design-time measurements, whereas Path Delay Sensor is the preferred structure to allow rapid performances estimations at run-time with a minimal area overhead.**

I. Introduction

Variability has become a major issue with recent technologies in the semiconductor industry [1]. While process variations impact process, supply voltage and internal temperature [2], chip performances are also dependent on environmental and on applicative changes that may further influence chip's behavior [3].

Field-Programmable Gate Arrays (FPGAs) devices are not spared by these unpredictable disparities. As underlined in [4], both inter-die and within-die variations affect FPGAs. So, it is necessary to implement solutions in order to compensate these variations.

Because of their inherent reconfigurability, it is possible to place a component of a design at a specific place on the FPGA floorplan, and to relocate it when it is required. In the literature, it has been suggested either to model FPGA variability [5] [6], or to measure it [7]. Both methods suggest then to constraint or to adapt the design so that it takes into account these variations and improves performance yield.

In this paper, we investigate the problem of variability characterization on FPGA. We provide a twofold method for variability compensation based on digital sensors used either at design-time or at run-time. Our study is focused on digital sensors, directly implemented in the FPGA building blocks. Their role is to measure the effective performance of the device. The novelty of this paper is that we compare two digital sensor structures, analyze their accuracy and area overhead in order to determine how they could be efficiently use to characterize the variability of any FPGA. In the scope of this paper, experiments was conducted on Xilinx Spartan-3 FPGAs.

The remainder of the paper is organized as follows. The next section presents the related works in the area of variability analysis and compensation techniques on FPGAs. Section III introduces a new multi-level compensation flow. In section IV, two digital sensors are presented to measure performance variations. Finally, in section V, results are exposed and discussed: overhead comparison, experimental setup, intra and inter-chip comparisons are analyzed to determine the efficiency and the accuracy of the provided digital sensors.

II. Related Works

Few recent papers suggest techniques to characterize and compensate performance variability on FPGAs. A first approach is based on modelization. Three papers have suggested theoretical techniques [5] [6] [8]. They are all verified through timing modeling. First, a multi-cycle Statistical Static Timing Analysis (SSTA) placement algorithm is exposed in [5]. In simulation, it is possible to improve performance yield by 68.51% compare to a standard SSTA. A second approach proposes a variability aware design technique to reduce the impact of process variations on the timing yield [8]. Timing variability is reduced thanks to the increase of shorter routing segments. Another side, the author of [6] confronts different strategies for compensate within-die stochastic delay variability. Worst case design, SSTA, entire FPGA reconfiguration and sub-circuits relocation within a FPGA are considered. SSTA provides better results than Worst case design although both reconfiguration methods allow significant improvements. However, in these papers, only theoretical techniques and simulated results are exposed.

A second approach is based on delay measurements [7]. In this paper, a ring oscillator is placed on the FPGA. The frequency of each oscillator is measured. A cartography of the chip is done. Nevertheless, the study proposes here to characterize 8 LUTs together as well as the impact of external variation is not introduced.

These two sorts of approaches suggest it is required to have two stages of characterization with digital sensors: one off-line and one on-line.

III. Multi-level compensation flow

In order to tackle FPGA variability issues, a multi-level compensation flow is proposed. Basically, it uses FPGA digital resources (LUTs and interconnects) to implement Hard Macro sensors. An overview of our methodology is depicted in Figure 1. It is divided into two parts. In the first phase, the FPGA performance is deeply analyzed off-line at a fine granularity by our dedicated monitors. At run-time, a reduced monitoring setup is implemented within the system itself, in order to check the run-time performances. Each level is further explained in the two following sections.

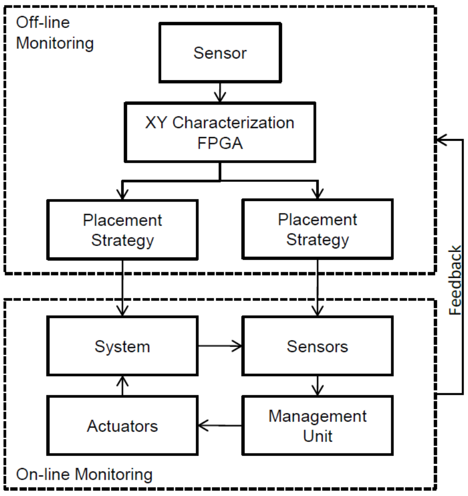


Fig. 1. Overall flow

A. Off-line monitoring and module placement strategy

The first step of the flow is depicted in Figure 2 and is divided into two global parts: the FPGA characterization and the module placement strategy. The monitoring system is directly applied at the technological level, *i.e.* it is intended to check at a fine granularity intrinsic performances of the FPGA device.

In order to realize an accurate characterization of performances, an array of sensors covering the whole area is used (Fig.2(a)). Sensor data are collected and analyzed to build a cartography of the floorplan (Fig.2(b)).

Once the cartography of the FPGA is built, a placement strategy is performed, considering both the system and its run-time monitoring service (Fig.2(c)). Basically, it consists in placing critical modules on " best performance " areas. A subset of on-line sensors is also implemented within the system in order to check at run-time the evolution of the performances. The sensor placement strategy takes into account requirements of run-time modules as well as the result of the off-line cartography.

B. On-line monitoring and dynamic compensation

The second stage of the compensation flow is based on hardware run-time monitoring. The run-time system implemented in the FPGA is composed of a microprocessor, some peripherals, a Management Unit (MU), a set of sensors and actuators. The on-line monitoring process is illustrated in Figure 2(c).

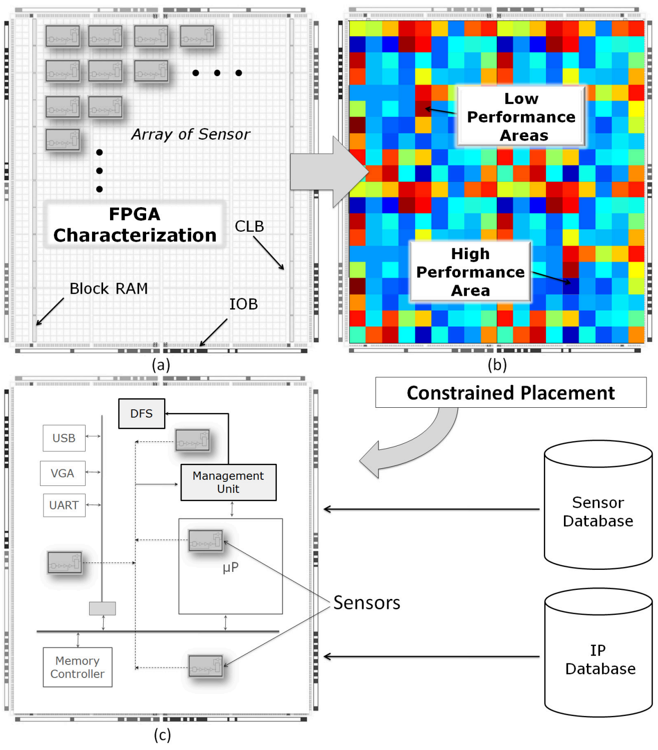


Fig. 2. Off-line and On-line Monitoring

Our objective is to perform a dynamic compensation of system variations. For this purpose, a subset of digital sensors using the FPGA resources is implemented. Digital sensors measure performances; data monitoring are then collected and analyzed by a management unit. Based on the information available, this unit can adapt the system to the actual performances; for instance, it is possible to adjust the frequency with a DFS actuator (Dynamic Frequency Scaler).

As depicted in the figure 1, a deeper FPGA performance analysis can be triggered when the management unit identifies suspicious system behavior. A partial or total analysis can be then performed in order to build a new cartography and to update the module placement strategy.

IV. Digital hardware sensors in FPGA

The objective of the previously described compensation flow is to adapt the system in relation to the effective performances of the FPGA device. Next, we study how to measure locally and globally this performance. The idea developed in this paper is to use digital hardware sensors designed with internal resources of the FPGA, namely CLBs and switch matrices. In this section, we present the principle and the implementation of two structures: a Ring Oscillator and Path Delay Sensor.

A. Ring Oscillator Sensor

The Ring Oscillator Sensor is based on the measurement of the oscillator frequency. In [9], the author exposes an internal temperature sensor for FPGA. The frequency of a ring oscillator is measured and converted into temperature. However, the ring oscillator is implemented into an old technology where process variability is very low.

An update of this sensor is exposed here. Its structure is depicted in Figure 3. The main part of the sensor is a $2p + 1$ inverter chain. The oscillation frequency directly depends on the FPGA performance capabilities. The first logic stage enables the oscillator to run for a fix number of periods of the main clock. The flip-flop at the end of the inverter chain is used as a frequency divider and allows filtering glitches from the oscillator. The final logic stage counts the number of transitions in the oscillator and transmits the count result. Then, the count result is used to calculate the oscillator frequency as follows:

$$F = \frac{count * f}{p} \tag{1}$$

where F is the ring oscillator frequency, $count$ is the 14-bit value of the counter, f is the operating frequency of the clock and p is the number of enabled clock periods for which the sensor is active.

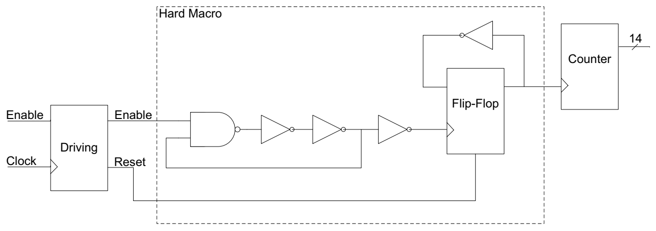


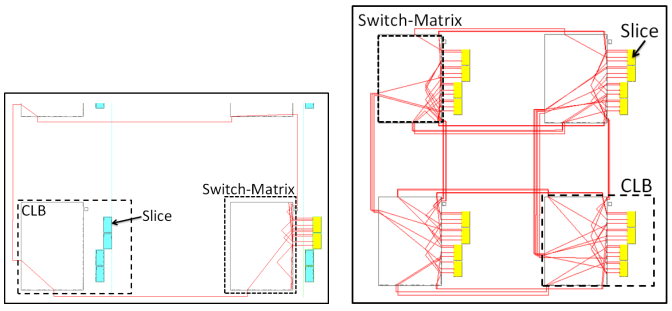
Fig. 3. Ring Oscillator

In order to use this sensor for the FPGA characterization, a three-inverter ring oscillator was implemented. With this configuration, the core of the sensor (ring oscillator + first flip-flop) takes only 4 LUTs. A Hardware Macro was designed so that the same sensor structure can be mapped at each floorplan location (Fig. 4(a)). It possibly allows characterizing separately each CLB of an FPGA.

B. Path Delay Sensor

In ASIC, in order to estimate the speed of a process, sensors for Critical Path Monitoring (CPM) are used. A. Drake presents a survey of CPM [10]. It exists a lot of techniques to manage Critical Path but very few are used in FPGAs. The Path Delay Sensor proposed here is directly inspired by CPM. The structure of the Path Delay Sensor is depicted in Figure 5. The idea of the Path Delay Sensor is to adapt CPM to FPGA. Indeed, the regularity of the FPGA structure enables to create more easily a critical path replica in FPGA than in ASIC.

The Path Delay Sensor is composed of n LUTs and n flip-flops (FF). The LUTs are chained together and a FF is set



(a) Ring Oscillator (b) Path Delay

Fig. 4. Hard Macro implementation for Spartan 3

at the output of each LUT. A clock signal is applied to the chain and is propagated into the LUT. At each rising edge, a n-bits thermometer code is available at the output of FFs. This thermometer code is representative of LUTs and interconnects performances.

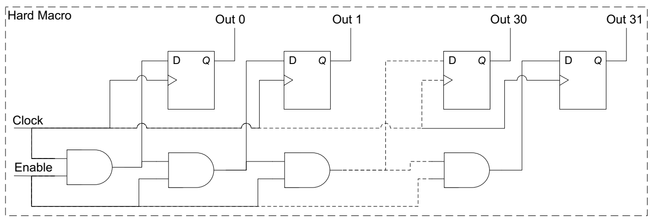


Fig. 5. Path delay

For example, the sensor is running and the thermometer code is stored. This code looks like: "11111111111111000000000000001111". It is then analyzed. The position Nz of last 0 is identified. Two different utilizations are then feasible :

- A fast one where direct compare Nz to the length of a critical path.
- A slow one where the time T required to cross one LUT and the associate interconnect is approximate:

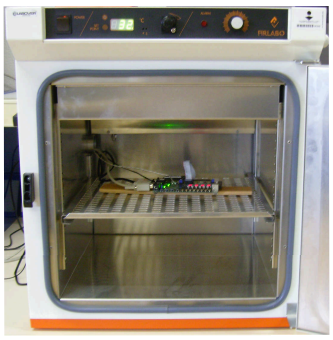
$$T = \frac{Nz + 2}{f} \tag{2}$$

where f is the frequency of the clock signal applied to the sensor and $Nz + 2$ represents the number of crossed LUTs over one for the crossing of the sample rate FF. The time T measured here enables to fast estimate the maximum frequency of one critical path.

In order to have relevant information, the size of this sensor must take into account the FPGA family in which it operates. For example, for a Spartan-3 device, this sensor is composed of 32 stages. It allows propagating a complete period of the Spartan-3 reference frequency clock ($50MHz$). The figure 4(b) shows the Hard Macro integration of this sensor.

V. Experimental results

The Ring Oscillator and Path Delay Sensor described in the previous section are studied and compared. They were both implemented into a Xilinx Spartan 3 Starter Kit Board with a XC3S1000-4FT256 (Fig.6(b)). This FPGA has a nominal operating point of $1.2V$ @ $25°C$. In order to ensure reproducible results, the temperature is kept constant in a thermal chamber during all measurements (this instrument only allows heating, and then experiments are done at $40°C$; this point will be discussed further) (Fig. 6(a)). We analyze first the resource overhead required by both structures. Then, the measurement errors are exposed and after that their impact is discussed. Finally, we provide both intra and inter variability characterizations of Spartan-3 FPGA devices.


(a) Experimental setup in the thermal chamber

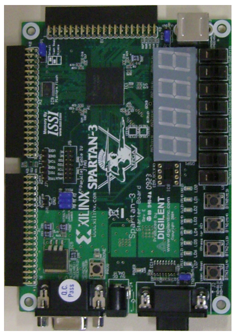

(b) Spartan 3 board

Fig. 6. Experimental setup

A. Overhead comparison

This section introduces an overhead comparison of the two sensors (Table. I). The area impact and the computing time for each sensor are presented.

TABLE I
OVERHEAD COMPARISON

	Hard Macro Size (# Slices)	Total Size (# Slices)	Sensor Latency (# Cycles)
Ring Oscillator	2	80	2046
Path Delay	16	16	2

The Hard Macro corresponds to the “ probe ” of the sensor. A smaller size for the probe allows characterizing a smaller area during the off-line monitoring phase. That’s why the Hard Macro Size is directly connected to the minimal size probing. Regarding on-line Monitoring, a small Hard Macro size is preferable to put it close to the critical modules (*e.g.* critical path). In this case, the Ring Oscillator Sensor is more interesting.

The total size of the sensor acts out the space needed for one full implementation of the sensor. This is to compare to the total space available in the FPGA. Indeed, the space used for sensor implementation is no longer available for the monitoring design. The Path Delay Sensor is better in this case.

The computing latency represents the number of clock periods between two successive measurements. Since the Ring Oscillator Sensor requires 2046 cycles, the Path Delay Sensor allows updating more rapidly the performance measurements. However, this potential advantage is to be considered with the processing capabilities of the management unit.

It is possible to take benefit from each sensor in different contexts. The Ring Oscillator Sensor will be preferably used for off-line monitoring characterization and for an accurate management of performance. The Path Delay Sensor will be preferred for a dynamic and direct critical path delay management.

In the next sections, the results of our experiments are presented. Our objective is to analyze the effectiveness of each sensor in its chosen field.

B. Impact of sources of error

This part proposes a study of the impact of each error source. We will reach successively voltage error, temperature error and toggle count error.

Xilinx Spartan-3 Starter Kit does not provide the feature to manage directly the power supply voltage. Hence, the voltage regulator of our boards was substituted by an external voltage regulator. The figure 7 depicts the normalized output frequency of the Hard Macro depending on the power supply voltage. A voltage variation of $0.4V$ around the default value implies a 22.5% of the sensor frequency. In our experiments, the supply voltage variation was only about $0.01V$ between boards. It involves a variation about $0.65MHz$ on the frequency (Tab. II). The supply voltage variation due to the fluctuation effects of the board regulator is less than $0.001V$. This has an impact of less than $65kHz$ on the frequency.

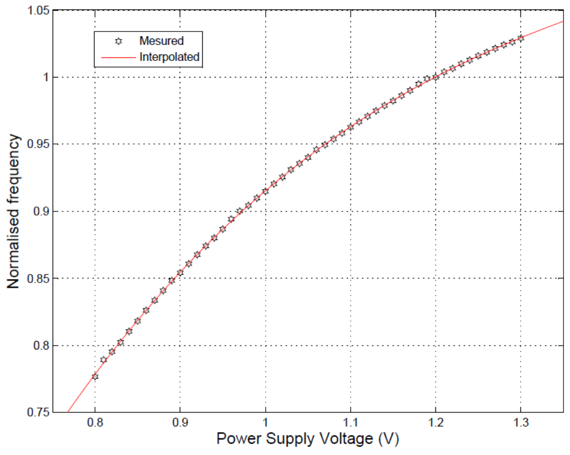


Fig. 7. Normalized Output frequency versus Power Supply Voltage

A representation of the normalized frequency depending on the temperature variation is illustrated in figure 8. The frequency of the ring oscillator Hard Macro decreases linearly as the temperature increases. During our experiments, the FPGA device is placed into a thermal chamber with a constant temperature (the variation is more or less $0.5°C$). This change causes a fluctuation of about $0.1MHz$ of the oscillator frequency.

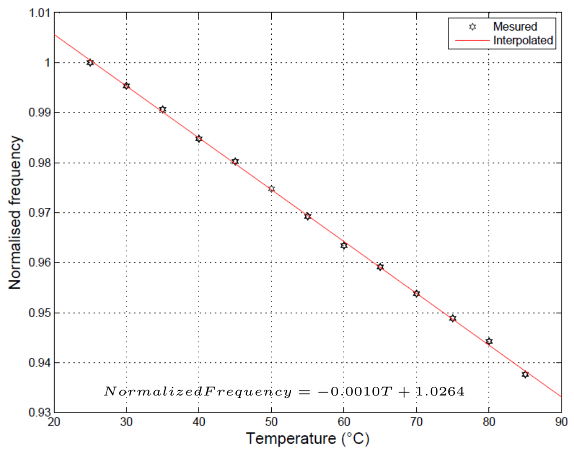


Fig. 8. Normalized Output frequency versus external temperature

Regarding the ring oscillator, the fluctuation of the toggle count is also to take into account. For a best efficiency, this sensor was dimensioned for an error of a maximum of 3 in the toggle count. It results an error on the measurement about $73kHz$.

TABLE II
ESTIMATE OF SOURCES OF ERROR

	Error	Frequency impact
Intra-board Voltage	$0.001V$	$0.065MHz$
Inter-board Voltage	$0.01V$	$0.65MHz$
Temperature	$0.5°C$	$0.1MHz$
Toggle count	3	$0.073MHz$

The total error due to external sources for intra-chip measurement is around $238kHz$. Nevertheless, all measurements are average 500 times thereby error due to variations is decreased. In fact, the total variation is around 6 times the standard deviation σ. The average standard deviation $\bar{\sigma}$ is then:

$$\bar{\sigma} = \frac{\sigma}{\sqrt{N}} \tag{3}$$

where N is the number of samples. Consequently, the total error attributable to external sources of variation is around 0.005% which corresponds to $10kHz$. This variation will be compared to the results obtained in next sections.

C. Intra-Chip Characterization using Ring Oscillator Sensor

The Ring Oscillator Sensor was used to achieve full cartographies of the Spartan-3 S1000 chip. During the measurements, the external temperature of the chip is kept constant at a temperature of $40°C$. This sensor is alternately placed at each CLB location, arranged into an array of 40 * 48 positions. Each measurement provided by the sensor is sent to an external computer, via an UART (Universal Asynchronous Receiver Transmitter), which performs the cartography of the chip (Fig. 9). The resulting frequency presented here corresponds to the mean oscillation frequency at the output of the Hard Macro.

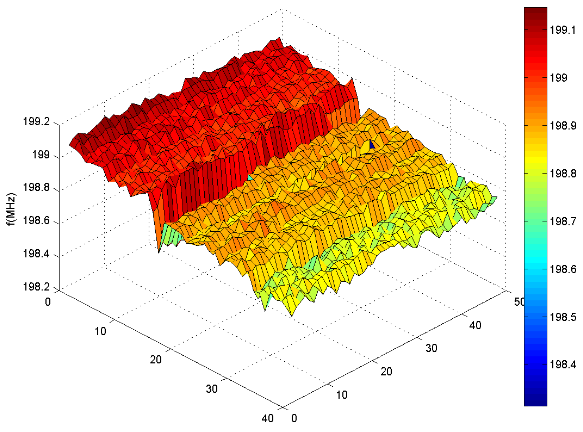


Fig. 9. Oscillator frequency cartography at T = $40°C$ for Board 3

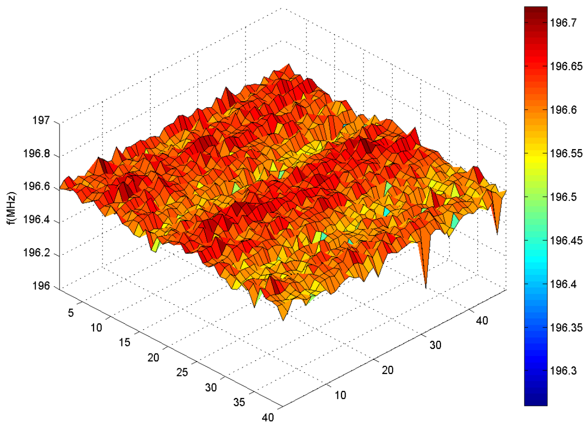


Fig. 10. Oscillator frequency cartography at T = $40°C$ for Board 4

It takes around 5 hours for a complete cartography with a 500 times averaging for each point.

The figures 9 and 10 depict the results of two cartographies. The cartography of the Board 4 is relatively constant with a maximum frequency variation about $458kHz$ while the second cartography presents a variation relatively large with a maximum frequency variation about $832kHz$. The cartography of the board 3 shows two features. First, there is a gradient of frequency all over the chip. Second, some slices on the middle of the chip are much less efficient than others. This type of results reinforce us in the necessity of a fine-grain cartography in order to achieve placement strategy for optimal performances on FPGAs.

Note that variations measured in a same board ($> 100kHz$) are minor compared to the error calculate previously ($\approx 10kHz$). Since we can assume that the supply voltage and the external temperature are fixed, we can infer that the variations measured on the frequency are effectively due to internal performance variations.

In future work, cartographies will be conducted with other temperature settings in order to confirm the trend on a same chip.

D. Inter-Chip Characterization

A similar characterization was conducted on several boards. Sample results are summarized in Table III. The figures 9 and 10 illustrate the difference between two boards. We can see that there are significant discrepancies between boards. Indeed, we observe here a difference of about 16.2% of the average frequency.

In this section, the Path Delay Sensor is used to perform a comparison between multiple boards. In Table IV, we have compared the five boards from the experience. This table introduces a time, which corresponds to the delay required by a signal edge to cross one LUT and the associate interconnect of the sensor. The Path Delay Sensor corroborates the results obtained with the Ring Oscillator Sensor (Fig. 11). However, with the Path Delay Sensor, we cannot distinguish the best FPGA between boards 3 and 4. This sensor is less accurate, but nevertheless allows fast estimations. For this reason, it will be used for on-line monitoring services.

TABLE III
COMPARISON OF RING OSCILLATOR FREQUENCY BETWEEN MULTIPLE BOARDS

Board	1	2	3	4	5
Frequency (MHz)	215.0	204.0	198.9	196.6	185.0

TABLE IV
COMPARISON OF PATH DELAY SPEED BETWEEN MULTIPLE BOARDS

Board	1	2	3	4	5
T (ns)	2.86	3.07	3.2	3.2	3.33

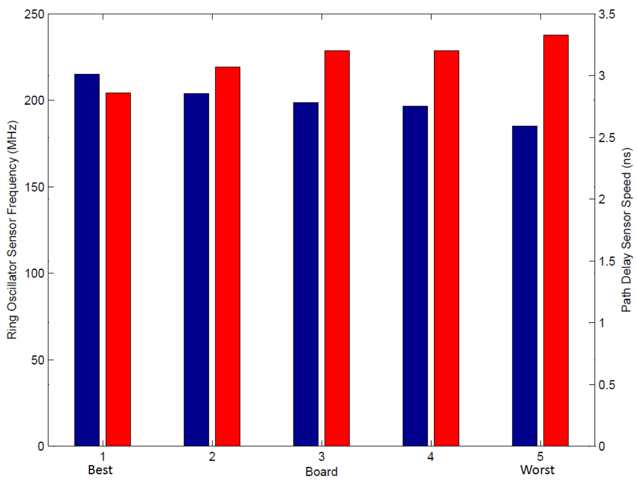


Fig. 11. Comparaison between multiple boards

VI. Conclusion and Future Works

Managing variability is one of the major issues in recent silicon technologies. As previously mentioned in the literature, FPGA devices are also subject to process, voltage, and temperature variations. In this paper, we have considered a twofold compensation flow for FPGAs, based on the use of digital sensors directly implemented in the reconfigurable resources. For this purpose, this article brought new results on the comparison of a Ring Oscillator structure and a Path Delay Sensor. In our experiments on Spartan-3 FPGAs, the ring oscillator was successfully used to perform intra-chip cartographies (1980 positions). Both sensors were evaluated for characterizing inter-chip performance variations.

We conclude that both structures are efficient for fast variability characterization in FPGA devices. The ring oscillator is the best structure for design-time measurements, whereas the Path Delay Sensor will be the preferred structure to allow rapid performances estimations at run-time with a minimal area overhead.

In future work, both sensors will be studied to perform run-time performance measurements. We will particularly focus on strategies to efficiently manage sensors (number, placement) and collect monitored information in order to adapt the system.

References

[1] "International Technology Roadmap for Semiconductors," 2009. [Online]. Available: http://www.itrs.net/Links/2009ITRS/Home2009.htm

[2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," *Design Automation Conference, 2003. Proceedings*, pp. 338–342. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1219020

[3] O. Unsal, J. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin, "Impact of Parameter Variations on Circuits and Microarchitecture," *IEEE Micro*, vol. 26, no. 6, pp. 30–39, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4042630

[4] P. Sedcole and P. Y. K. Cheung, "Parametric yield in FPGAs due to within-die delay variations:a quantitative analysis," *International Symposium on Field Programmable Gate Arrays*, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1216949

[5] G. Lucas, C. Dong, and D. Chen, "Variation-aware placement for FPGAs with multi-cycle statistical timing analysis." in *FPGA*, P. Y. K. Cheung and J. Wawrzynek, Eds. ACM, 2010, pp. 177–180. [Online]. Available: http://dblp.uni-trier.de/db/conf/fpga/fpga2010.html#LucasDC10

[6] P. Sedcole and P. Y. K. Cheung, "Parametric Yield Modeling and Simulations of FPGA Circuits Considering Within-Die Delay Variations," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 2, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1371582

[7] P. Sedcole and P. K. Y. Cheung, "Within-die delay variability in 90nm FPGAs and beyond," in *IEEE International Conference on Field Programmable Technology*. IEEE, 2006, pp. 97–104. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4042421

[8] A. Kumar and M. Anis, "FPGA Design for Timing Yield Under Process Variations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 3, pp. 423–435, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4799224

[9] S. Lopez-Buedo, J. Garrido, and E. Boemo, "Dynamically inserting, operating, and eliminating thermal sensors of FPGA-based systems," *IEEE Transactions on Components and Packaging Technologies*, vol. 25, no. 4, pp. 561–566, 2002. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1178745

[10] A. Drake, *Adaptive Techniques for Dynamic Processor Optimization*, ser. Series on Integrated Circuits and Systems. Boston, MA: Springer US, 2008. [Online]. Available: http://www.springerlink.com/content/r61t506740v74220

Investigating Self-Timed Circuits for the Time-Triggered Protocol

Markus Ferringer
Department of Computer Engineering
Embedded Computing Systems Group
Vienna University of Technology
1040 Vienna, Treitlstr. 3
Email: ferringer@ecs.tuwien.ac.at

Abstract—**While asynchronous logic has many potential advantages compared to traditional synchronous designs, one of the major drawbacks is its unpredictability with respect to temporal behavior. Without having a high-precision oscillator, a self-timed circuit's execution speed is heavily dependent on temperature and supply voltage. Small fluctuations of these parameters already result in noticeable changes of the design's throughput and performance. This indeterminism or jitter makes the use of asynchronous logic hardly feasible for real-time applications.**

Based on our previous work we investigate the temporal characteristics of self-timed circuits regarding their usage in the Time-Triggered Protocol (TTP). We propose a self-adapting circuit which shall derive a suitable notion of time for both bit transmission and protocol execution. We further introduce and analyze our jitter compensation concept, which is a three-fold mechanism to keep the asynchronous circuit's notion of time tightly synchronized to the remaining communication participants. To demonstrate the robustness of our solution, we will perform temperature and voltage tests, and investigate their impact on jitter and frequency stability.

I. Introduction

Asynchronous circuits elegantly overcome some of the limiting issues of their synchronous counterparts. The often-cited potential advantages of asynchronous designs are – among others – reduced power consumption and inherent robustness against changing operating conditions [1], [2]. Recent silicon technology additionally suffers from high parameter variations and high susceptibility to transient faults [3]. Asynchronous (delay-insensitive) design offers a solution due to its inherent robustness. A substantial part of this robustness originates in the ability to adapt the speed of operation to the actual propagation delays of the underlying hardware structures, due to the feedback formed by completion detection and handshaking. While asynchronous circuits' adaptive speed is hence a desirable feature with respect to robustness, it becomes a problem in real-time applications that are based on a stable clock and a fixed (worst-case) execution time. Therefore, asynchronous logic is commonly considered inappropriate for such real-time applications, which excludes its use in an important share of fault-tolerant applications that would highly benefit from its robustness. Consequently, it is reasonable to take a closer look at the actual stability and predictability of asynchronous logic's temporal behavior. After all, synchronous designs operate on the same technology, but hide their imperfections with respect to timing behind a strictly time driven control flow that is based on worst-case timing analysis. This masking provides a convenient, stable abstraction for higher layers. In contrast, asynchronous designs simply allow the variations to happen and propagate them to higher layers. Therefore, the interesting questions are: Which character and magnitude do these temporal variations have? Can these variations be tolerated or compensated to allow the usage of self-timed circuits in real-time applications?

In our research project ARTS[1] (Asynchronous Logic in Real-Time Systems) we are aiming to find answers to these questions. Our project goal is to design an asynchronous TTP (Time-Triggered Protocol) controller prototype which is able to reliably communicate with a set of synchronous equivalents even under changing operating conditions. TTP was chosen for this reference implementation because it can be considered as an outstanding example for hard real-time applications. In this paper we present new results based on our previous work. We will investigate the capabilities of self-timed designs to adapt themselves to changing operating conditions. With respect to our envisioned asynchronous TTP controller we will also study the characteristics of jitter (and the associated frequency instabilities of the circuit's execution speed) and our corresponding compensation mechanisms. We implement and investigate a fully functional transceiver unit, as required for the TTP controller, to demonstrate the capabilities of the proposed solution with respect to TTP's stringent requirements.

The paper is structured as follows: In Section II we give some important background information on TTP, the research project ARTS, and the used asynchronous design style. Section III presents related work and describes our previous work and the respective results. We show and discuss experimental results in Section IV, before concluding in Section V.

II. Background

A. Time-Triggered Protocol

The Time-Triggered Protocol (TTP) has been developed for the demanding requirements of distributed (hard) real-time

[1]The ARTS project receives funding from the FIT-IT program of the Austrian Federal Ministry of Transport, Innovation and Technology (bm:vit, http://www.bmvit.gv.at/), project no. 813578.

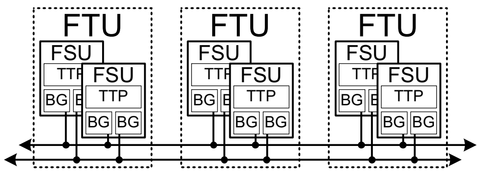


Figure 1. TTP system structure.

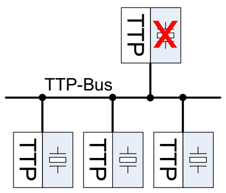


Figure 2. ARTS system setup.

systems. It provides several sophisticated means to incorporate fault-tolerance and at the same time keep the communication overhead low. TTP uses extensive knowledge of the distributed system to implement its services in a very efficient and flexible way. Real-time systems in general and TTP in particular are described in detail in [4], [5].

A TTP system generally consists of a set of Fail-Silent Units (FSUs), all of which have access to two replicated broadcast communication channels. Usually two FSUs are grouped together to form a Fault-Tolerant Unit (FTU), as illustrated in Figure 1. In order to access the communication channel, a TDMA (Time Division Multiple Access) scheme is implemented: Communication is organized in periodic TDMA rounds, which are further subdivided into various sending slots. Each node has *statically* assigned sending slots, thus the entire schedule (called Message Descriptor List, MEDL) is known at design-time already. Since each node a priori knows when other nodes are expected to access the bus, message collision avoidance, membership service, clock synchronization, and fault detection can be handled without considerable communication overhead. Explicit Bus Guardian (BG) units are used to limit bus access to the node's respective time slots, thereby solving the babbling idiot problem. Global time is calculated by a fault-tolerant, distributed algorithm which analyzes the deviations in the expected and actual arrival times of messages and derives a correction term at each node.

The Time-Triggered Protocol provides very powerful means for developing demanding real-time applications. The highly deterministic and static nature makes it seemingly unsuited for an implementation based on asynchronous logic. Hence, these properties also make TTP an interesting and challenging topic for our exploration of predictability of self-timed logic.

B. ARTS Project

The aim of the research project ARTS (Asynchronous Logic in Real-Time Systems) is to integrate asynchronous logic into real-time systems. For this purpose, an asynchronous TTP controller is developed and integrated into a cluster of (synchronous) TTP chips, as illustrated in Figure 2. The asynchronous device should be capable of successfully taking part in time-triggered communication, thereby using solely the system inherent determinism and a priori knowledge of TTP to derive a suitable and precise time reference for both bit-timing as well as high-level services.

The central concern for the project is the predictability of asynchronous logic with respect to its temporal properties. We therefore investigate jitter sources (e.g., data dependencies, voltage fluctuations, temperature drift) and classify their impact on the execution time. Using an adequate model allows us to identify critical parts in the circuit and implement measures for compensation. Another issue concerns timeliness itself, as without a reference clock we do not have an *absolute* notion of time. Instead, we will use the strict periodicity of TTP to continuously re-synchronize to the system and derive a time-base for message transfer.

In this perspective, the main challenge relies in the method of resynchronization, as the controller will use the data stream provided by the other communication participants to dynamically adapt its internal time reference. The chosen solution is to use a free-running, self-timed counter for measuring the duration of external events of known length (i.e., single bits in the communication stream). The so gained reference measurement can in turn be used to generate ticks with the period of the observed event. This local time base should enable the asynchronous node to derive a sufficiently accurate time reference for both low-level communication (bit-timing, data transfer) as well as high-level services (e.g. macrotick generation). The disturbing impact of environmental fluctuations is automatically compensated over time, because periodic resynchronization will lead to different reference measurements, depending on the current speed of the counter circuit.

C. Asynchronous Design Style

Our focus is on delay insensitive (DI) circuits[2], as they exhibit more pronounced "asynchronous" properties than bounded delay circuits. More specifically, we use the level-encoded dual-rail approach (LEDR [6], [7], used in Phased Logic [8] and Code Alternation Logic [9]), which encodes a logic signal on two physical wires. We prefer the more complex 2-phase implementation over the popular 4-phase protocol [2], [10], as it is more elegant and we already gained some practical experience with it. LEDR periodically alternates between two disjoint code sets for representing logic "HI" and "LO" (two phases φ_0 and φ_1, see Figure 3), thus avoiding the need to insert NULL tokens as spacers between subsequent data items. On the structural level, LEDR designs are based on Sutherland's micropipelines [11]. In its strongly indicating mode of operation the performance is always determined by the slowest stage.

[2]Typically the mandatory delay constraints are hidden inside the basic building blocks, while the interconnect between these modules is considered unconstrained

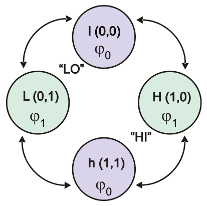


Figure 3. LEDR coding.

Using this approach, completion detection comes down to a check whether the phases of all associated signals have changed and match. As usual, handshaking between register/pipeline stages is performed by virtue of a *capture done* (or simply *cDone*) signal [11]. The two-rail encoding as well as the adaptive timing make LEDR circuits very robust against faults and changing operating conditions, unfortunately at the cost of increased area consumption and reduced performance. Interfacing single-rail inputs/outputs require special "interface gates" [8], which must properly align ("synchronize") these to the LEDR operation phases. Concrete implementations and variations of such interface gates are described in [12].

III. State of the Art

A. Related Work

Various options for building adequate time references are commonly used in today's logic designs, e.g., crystal oscillators provide high precision at high frequencies. RC oscillators [13] or simple inverter-loops are alternatives if operating frequency and precision are no major concerns. It is also possible to generate clock signals in a fault-tolerant, distributed way [14].

Another important alternative for generating precise time references are self-timed oscillator rings, which seem to be perfectly suited for the chosen asynchronous design methodology. A lot of research has been conducted on self-timed oscillator rings (which are also based on micropipelines). For example, in [15] a methodology for using self-timed circuitry for global clocking has been proposed. The same authors also used basic asynchronous FIFO stages to generate multiple phase-shifted clock signals for high precision timing in [16]. Furthermore, it has been found that event spacing in self-timed oscillator rings can be controlled [17], [18]. The Charlie- and the drafting-effects have thereby been identified as major forces controlling event spacing in self-timed rings [16], [19].

One of the major requirements of our time base is to be stable in order to allow for reliable bus-communication. In the synchronous world, the term "jitter" is commonly used to classify the deviations of a (clock-) signal from its ideal behavior [20], [21]. It is also important for our investigations to fully understand the sources and effects of jitter to implement adequate countermeasures. In order to classify the frequency stability of single execution steps and other timed signals, we use Allan variance plots [22], [23], which provide the appropriate means for a detailed analysis. Instead of a single number, Allan deviation is usually displayed as a graph describing gradually increasing durations τ of the averaging window. It therefore combines measures for both short-term and long-term stability in a single plot. Thorough circuit analysis, jitter estimation, classification, and interpretation, in combination with a corresponding model have already been elaborated in a previous paper. The next section provides a short summary of the performed work and the respective results we found.

B. Previous Work

1) Jitter in asynchronous circuits: In our previous work we investigated the temporal behavior of self-timed (delay insensitive) circuits not only on an experimental basis, but also from a theoretical point of view. To fully understand the complex characteristics of logic circuits it is necessary to separate and classify the sources and manifestations of jitter. While jitter terminology and measurement techniques are well established for synchronous designs, measuring jitter effects in asynchronous circuits significantly differs in that no reference values are available. After all, it is a desired property of asynchronous logic to adapt its speed of operation to the given conditions. We therefore define *execution period jitter*, or just execution jitter, to be the variation in the durations of a specific LEDR-register. The inherent handshaking guarantees the *average* rate of phase changes for all coupled registers to be the same. However, due to the fact that LEDR circuits are "elastic", there may be substantial short-term differences in the execution speeds of different pipeline stages.

From an abstract point of view, we can categorize jitter in two major groups. On the one hand, *systematic jitter* describes all effects that can be reproduced by our system setup. On the other hand, *random jitter* is observed if the timing variations are not controllable by means of system setup. The following classification can be made for systematic effects:

- *Data-Dependent Execution Jitter (DDEJ)* deals with cases where the actual data values induce (systematic) jitter on a signal (circuit state, Simultaneous Switching Noise [24], . . .).
- Consequently, *Data-Independent Execution Jitter (DIEJ)* subsumes all non-data-dependent systematic jitter effects (global changes of temperature and voltage, e.g).

2) Timing Model: Keeping the above classification of jitter sources in mind, we can examine sources of data-dependent and random jitter from a logic designer's point of view. With the resulting model we can track the sources of data-dependent jitter to their roots. Figure 4 illustrates the remarkable effects of data-dependent execution jitter. It shows the jitter histogram of a free-running, self-timed, 4-bit counter. The solid black line represents a histogram obtained by simulation of the proposed timing model. As some of the humps are very close together, their superpositions often appear as single peak only. If we had turned random jitter off entirely in the simulation, we could see 16 sharp peaks in the graph (one for each counter value). On the other hand, the filled area shows the jitter histogram taken from an FPGA measurement. Again, the peaks are superpositions of different delays caused by the different counter values. The differences between simulation and actual measurement

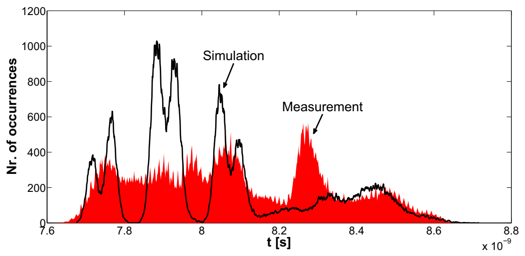


Figure 4. Jitter histogram simulated/measured, 4-bit counter.

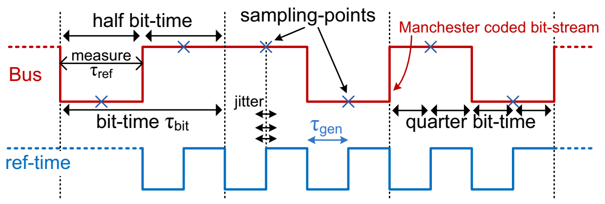


Figure 5. Manchester code with sampling points.

can be explained by the fact that placement and routing information of logic gates is not considered for simulation (i.e., we perform pre-layout simulations only). However, the results are still accurate enough to allow for identification of the main sources of jitter, possible bottlenecks, and a quantitative estimation of the expected jitter characteristics.

3) Time Base Generation: In order to allow for reliable TTP communication, the resulting asynchronous controller must have a precise notion of time. As there is no reliable reference time available in the asynchronous case, we design a circuit that uses the TTP communication stream to derive a suitable, stable time-base. We construct an adjustable tick-generator and periodically synchronize it to *incoming* message-bits. In our configuration, the bit-stream of TTP uses Manchester coding, thus there is at least one signal transition for each bit which we can potentially use for recalibration. The Manchester encoding is a line code which represents the logical values 0 and 1 as falling and rising transitions, respectively. Consequently, each bit is transmitted in two successive symbols, thus the needed communication bandwidth is double the data rate. The top part of Figure 5 shows three bits of an exemplary Manchester coded signal, whereby the transitions at 50% of the bittime define the respective logical values. This encoding scheme has the advantage of being self-clocking, which means that the clock signal can be recovered from the bit stream. From an electrical point of view, Manchester encoding allows for DC-free physical interfaces.

Figure 5 further illustrates the properties that our design needs to fulfill. As already mentioned, Manchester coding uses two symbols to transmit a single bit, thus the "feature-size" τ_{ref} of the communication stream is half the actual bit-time τ_{bit}. It can also be seen that the sampling points need to be located at 25% and 75% of τ_{bit}, respectively. We intend to achieve this quarter-bit-alignment by doubling the generated tick-frequency ($\tau_{gen} = \frac{\tau_{ref}}{2}$). Consequently, each rising edge

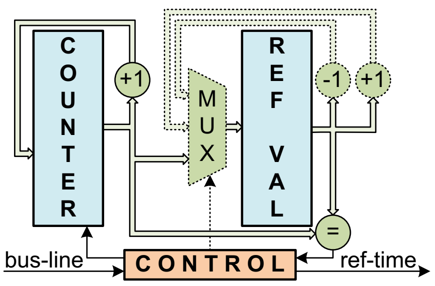


Figure 6. Basic structure of the timer-reference generation circuit.

of signal *ref-time* defines an optimal sampling point. As our circuit is implemented asynchronously, the generated reference signal will be subject to jitter. Furthermore, temperature and voltage fluctuations will also change the reference's signal period. It is therefore necessary to make the circuit self-adaptive to changing operating conditions.

The basic structure of the circuit is shown in Figure 6. As one can see, the interface of our design is quite simple. There is only one input (*bus-line*, the receive-line of the TTP bus), as well as one output (*ref-time*, an asynchronously generated signal with known period). The dashed components and the MUX have been added to the circuit to allow rate correction on a by-bit basis in combination to the absolute measurement of τ_{ref} during the first bit of a message (SOF, Start-of-Frame). If the control block detects the SOF signature, it resets the free-running counter unit. The asynchronous counter periodically increments its own value at a certain (variable) rate, which mainly depends on the circuit structure, placement and routing, and environmental conditions. After time τ_{ref}, the corresponding end of SOF will eventually be detected by the control-block. As a consequence, the current counter value is preserved in register *ref-val* (reference value) and the counter is restarted. The controller is now able to reproduce the measured low-period τ_{ref} by periodically counting from zero to *ref-val*, and generating a transition on *ref-time* for each compare-match. In order to achieve the 25%/75%-alignment, we double the output frequency by simply halving *ref-val*.

IV. Experimental Results

In this section we will present a detailed analysis of the experiments we performed with the proposed circuit from Figure 6. We will vary temperature and operating voltage and monitor the generated time reference under these changing conditions. Simultaneously, we will also evaluate the robustness and effectiveness of the three compensation mechanism implemented in the final design:

1) *Low-Level State Correction*: Measuring period τ_{ref} of the SOF sequence retrieves an absolute measure of the reference time. However, as only one measurement is performed per message, quantization errors and other systematic, data-dependent delay variations significantly restrict the achievable precision. The possible resolution depends on the speed of the free-running counter, and

is at about 25ns for our current implementation[3].

2) *Low-Level Rate Correction*: As the Manchester code always provides a signal transition at 50% of τ_{bit}, we can continuously adapt the measured reference value *ref-val*. We only allow small changes to *ref-val*: It is either incremented or decremented by one, depending on whether the expected signal transitions are late or early, respectively. The advantage of this additional correction mechanism is that quantization errors and data-dependent effects are averaged over time, thus increasing precision.
3) *High-Level Rate Correction*: The software-stack controlling the message transmission unit can add another level of rate correction. As it knows the expected (from the MEDL) and actual (from the transceiver unit) arrival times of messages, the difference of both can be used to calculate an error-term. High-level services and message transmission can in turn be corrected by this term to achieve even better precision. The maximum resolution which can be achieved by this technique depends on the baud-rate, and is half a bit-time.

Remark: We are well aware that the presented results can only be seen as snapshot for our specific setup and technology. Changing the execution platform will certainly change the outcomes of our measurements, as jitter and the corresponding frequency instabilities mainly depend on the circuit structure and the used technology. However, from a qualitative point of view, our results are valid for other platforms and technologies as well, even if concrete measurements must be taken for a quantitative evaluation.

A. *Time Reference Generator*

Before we start with the message transmission unit, which implements all of the above compensation mechanisms, we want to take a closer look at the basic building block (cf. Figure 6). Clearly, compensation method (3) is not present, as we just investigate the time reference generation unit. This unit does not actually receive or transmit messages, it just generates signal *ref-time* out of the incoming signal transitions on the TTP bus. The measurement setup is fairly simple: There is a (synchronous) sender, which periodically sends Manchester coded messages. The asynchronous design uses these messages to generate its internal time-reference. All measurements have been taken while the bus was idle. This way, we can observe the circuit's capability of reproducing the measured duration without any disturbing state- or rate correction effects. If not stated otherwise, the measurements are taken at ambient temperature and nominal supply voltage.

First we take a look at the frequency stability of *ref-time* and *cDone*. The first part of the Allan-plot in Figure 7, ranging from approximately $2*10^{-8}s$ to $10^{-4}s$ on the x-axis, is obtained by monitoring the handshaking signal *capture-done* from a register cell. The second part, which starts at

[3]Notice that FPGAs are not in any way optimized for LEDR circuits. Dual-rail encoding introduces not only considerable interconnect delays, but also significant area overhead compared to ordinary synchronous logic.

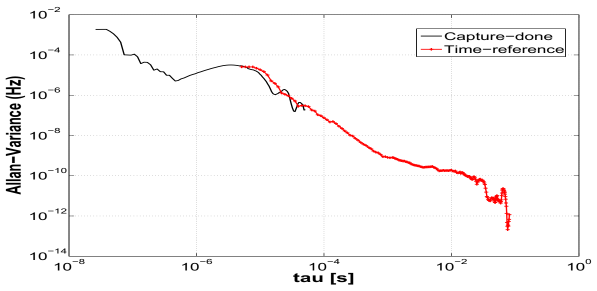


Figure 7. Allan-Variance.

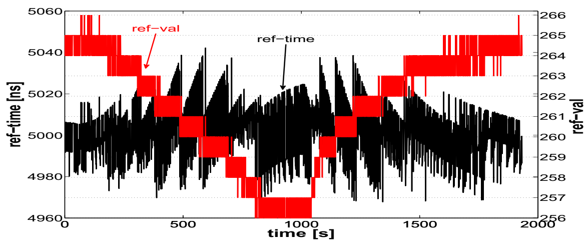


Figure 8. *ref-val* vs. timer reference period for temperature-tests.

$3*10^{-6}s$ and thus slightly overlaps with *capture-done*, has been obtained by measuring *ref-time*. Notice that it is no coincidence that both parts in the figure almost match in the overlapping section: Signal *ref-time* is based upon the execution of the low-level hardware and is therefore directly coupled to the respective jitter and stability characteristics. It is obvious from the graph that the stability increases to about $10^{-10}Hz$ for $\tau \approx 10^{-2}s$. Furthermore, the reference signal is far more stable than the underlying generation logic (*cDone*), as periodically executing the same operations compensates data-dependent jitter and averages random jitter. Although the underlying low-level signals jitter considerably due to data-dependent jitter, the circuit's output is orders of magnitudes more stable, as these variations are canceled out during the periodic executions.

One of the major benefits of the proposed solution is its robustness to changing operating conditions, thus we additionally vary the environment temperature and observe the changes in the period of *ref-time*. We heat the system from room temperature to about $83°C$, and let it cool down again. Figure 8 compares *ref-val* to the signal period of *ref-time*. While the ambient temperature increases, *ref-val* steadily decreases from 265 down to 256. The period of *ref-time* makes an approximately 19ns-step (the duration of a single execution step) each time *ref-val* changes. During the periods where the changes in execution speed cannot be compensated (because they are too small), *ref-time* slowly drifts away from the optimum at $5\mu s$. Without any compensation measures the duration of *ref-time* would be about $5180ns$ at the maximum temperature, instead of being in the range of approximately $5\mu s \pm 38ns$ (i.e. the duration of $\pm$ two execution steps), no matter what temperature. Notice that the performance of the self-timed circuit decreases by 3.5% at the maximum

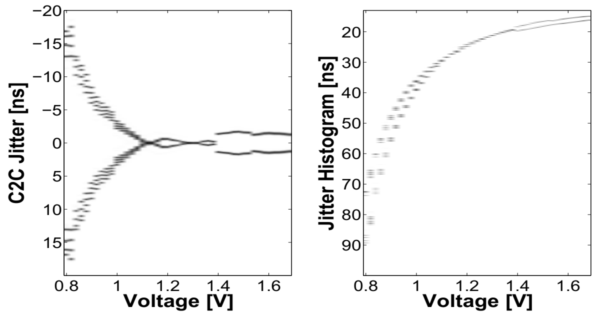


Figure 9. Cycle-to-cycle jitter (left), Jitter histogram (right).

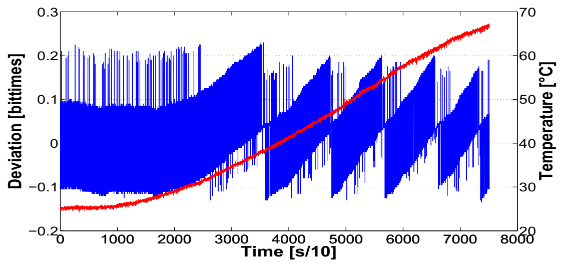


Figure 10. Relative deviation from optimal sending slot and operating temperature.

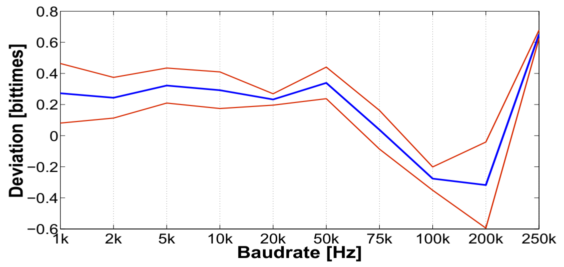


Figure 11. Mean relative deviation from optimal sending point vs. baudrate.

temperature, which seems to be relatively low, but it certainly is a showstopper for reliable TTP communication.

Far more pronounced delay variations can be obtained by changing the core supply voltage. We applied 0.8V to 1.68V in steps of 20mV core voltage to our FPGA-board. This time the execution speed of our self-timed circuit increased from about 80ns per step to approximately 15ns per steps, as shown in Figure 9(right). This plot illustrates the jitter histogram on the y-axis versus the FPGA's core supply voltage on the x-axis. Thereby, the densities of the histogram are coded in gray-scale (the darker the denser the distribution). It is evident from the figure that performance increases exponentially with the supply voltage. This illustration also shows other interesting facts: For one, almost all voltages have at least two separate humps in their histograms. These are caused by data-dependencies that originate in the different phases $\varphi_{0,1}$. Furthermore, for low voltages, additional peaks appear in the histograms and the separations between the phases increase as well. This can be explained as data-dependent effects caused by different delays through logic stages are magnified while the circuit slows down. This property is better illustrated in Figure 9(left), where cycle-to-cycle execution jitter is plotted over the supply voltage. The graph appears almost symmetrically along the x-axis, which is caused by the continuous alternation of phases.

We conclude that varying operating conditions not only affect the speed of asynchronous circuits, but also the respective jitter characteristics. In this perspective, slower circuits tend to have higher jitter, which is further magnified by increased quantization errors due to the low sampling rate.

B. Transceiver Unit

The message transmission unit will implement all three compensation methods mentioned at the beginning of Section IV. We intend to use this unit directly in the envisioned asynchronous TTP controller, thus we need to examine the gained precision of this design with respect to timeliness. The interface from the controlling (asynchronous) host to the sub-design of Section IV-A is realized as dual-ported RAM: Whenever the bus transceiver receives a message, it stores the payload in combination with the receive-timestamp[4] in RAM and issues a receive-interrupt. Likewise, the host can request to transfer messages by writing the payload and the estimated sending time into RAM and asserting the transmit request.

[4]We define the internal time to be the number of ticks of signal *ref-time*, i.e., the number of execution steps performed by the bus transceiver unit.

During reception of messages, the circuit can continuously recalibrate itself to the respective baudrate, as Manchester code provides at least one signal transition per bit. However, between messages and during the asynchronous node's sending slot, resynchronization is not possible. In these phases we need to rely on the correctness of *ref-time*. The Start-Of-Frame sequence of each message must be initiated during a relatively tight starting window, which is slightly different for all nodes and is continuously adapted by the TTP's distributed clock synchronization algorithm. Failing to hit this starting window is an indication that the node is out-of-sync.

As we are interested in the accuracy of hitting the starting window, we configured the controlling host in a way that it triggers a message-transmission 25 bittimes after the last bit of an incoming message. We simultaneously heated the system from room temperature to about $68°C$ to check on the expected robustness against the respective delay variations. The results are shown in Figure 10, where the deviation from the optimal sending-point (in units of bittimes) and the operating temperature are plotted against time. Similar to Figure 8 one can see that while the circuit gets warmer (and thus slower), the deviation steadily increases. As soon as the accumulated changes in delay can be compensated by the low-level measurement circuitry (i.e., *ref-val* decreases), the mean deviation immediately jumps back to about zero. We can see in the figure that the timing error is in the range from approximately -0.1 to $+0.2$ bittimes, which will surely satisfy the needs of TTP.

The next property we are interested in is the circuit's behav-

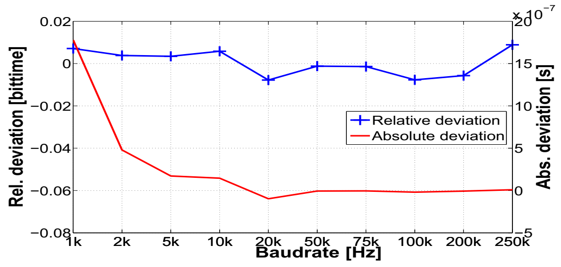


Figure 12. Mean relative/absolute deviation from optimum bit-time.

ior with respect to different baudrates. Although low bitrates have the advantage of minimizing the quantization error, jitter has much more time to accumulate compared to high data rates. It is thus not necessarily true that lower baudrates result in a more stable and precise time reference. On the other hand, if the data rate is too high, it is not possible to reproduce τ_{ref} correctly, and even small changes of the reference value *refval* lead to large relative errors in the resulting signal period. The optimum baudrate will therefore be located somewhere between these extremes. Figure 11 illustrates this by plotting the mean deviations of the optimum sending points versus the bitrate (the "corridor" additionally shows the respective standard deviations). Notice that the y-axis shows the relative deviation in units of bit-times. Therefore, for example, the absolute deviation of the 1kHz bit-rate is more than 50 times larger than that of 50kHz. Clearly, TTP does not support baudrates as low as 1kHz. Reasonable data rates are at least at 100kHz and above (up to 4Mbit/s for Manchester coding). Our current setup allows us to use 100kbit/s for communication with acceptable results. However, we hope to be able to achieve 500kbit/s in our final system setup (with a more sophisticated development platform and a further optimized design).

Finally, we take a look at the accuracy of the generated time reference for different baudrates. Figure 12 therefore shows the mean relative (again in units of bit-times) and the absolute (in seconds) deviations of the actual reference periods from their nominal values. For all baudrates, the relative deviations are within a range of approximately ± 0.01 bit-times, or $\pm 1\%$, while the absolute timing errors are significantly larger for baudrates below 50kbit/s.

V. Conclusion

In this paper we introduced the research project ARTS, provided information on the project goals and explained the concept of TTP. We proposed a method of using TTP's bit stream to generate an internal time reference (which is needed for message transfer and most high-level TTP services). With this transceiver unit for Manchester coded messages we performed measurements under changing operating temperatures and voltages. The results clearly show that the proposed architecture works properly. The results further indicate that the achievable precision is in the range of about 1%. This is not a problem while other (synchronous) node are transmitting messages, as resynchronization can be performed continuously. However, during message transmission, the design depends on the quality of the generated reference time. Our measurement show that we are able to hit the optimum sending point with a precision of approximately ± 0.3 bit-times (assuming an interframe gap of 25 bits), which should be enough for the remaining nodes to accept the messages.

However, there still is much work to be done. The presented temperature and voltage tests are only a relatively small subset of tests that can be performed. One of the most interesting questions concerns the dynamics of changing operating conditions: How rapidly and aggressively can the environment change for the asynchronous TTP controller to still maintain synchrony with the remaining system? It should be clear from our approach that an answer to this question can only be given with respect to the concrete TTP schedule, as message lengths, interframe gaps, baudrate, etc. directly influence the achievable precision of our solution. The next steps of the project plan include the integration of the presented transceiver unit into an asynchronous microprocessor, the implementation of the corresponding software stack, and the interface to the (external) application host controller. Once the practical challenges are finished, thorough investigations of precision, reliability and robustness of our asynchronous controller will be performed.

References

[1] C. J. Myers, *Asynchronous Circuit Design*. Wiley-Interscience, John Wiley & Sons, Inc., 605 Third Avenue, New York, N.Y. 10158-0012: John Wiley & Sons, Inc., 2001.
[2] J. Sparso and S. Furber, *Principles of Asynchronous Circuit Design - A Systems perspective*. MA, USA: Kluwer Academic Publishers, 2001.
[3] N. Miskov-Zivanov and D. Marculescu, "A systematic approach to modeling and analysis of transient faults in logic circuits," in *Quality of Electronic Design, 2009. ISQED 2009.*, March 2009, pp. 408–413.
[4] H. Kopetz and G. Grundsteidl, "TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems," *Symposium on Fault-Tolerant Computing, FTCS-23.*
[5] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. MA, USA: Kluwer Academic Publishers, 1997.
[6] M. E. Dean, T. E. Williams, and D. L. Dill, "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," in *Proceedings of the 1991 University of California/Santa Cruz conference on Advanced research in VLSI.* Cambridge, MA, USA: MIT Press, 1991, pp. 55–70.
[7] A. McAuley, "Four state asynchronous architectures," *IEEE Transactions on Computers*, vol. 41, no. 2, pp. 129–142, Feb 1992.
[8] D. Linder and J. Harden, "Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 1031–1044, Sep 1996.
[9] M. Delvai, "Design of an Asynchronous Processor Based on Code Alternation Logic - Treatment of Non-Linear Data Paths," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Dec. 2004.
[10] K. Fant and S. Brandt, "NULL Convention LogicTM: a complete and consistent logic for asynchronous digital circuit synthesis," in *ASAP 96. Proceedings of International Conference on Application Specific Systems, Architectures and Processors, 1996.*, Aug 1996, pp. 261–273.
[11] I. E. Sutherland, "Micropipelines," *Communications of the ACM, Turing Award*, vol. 32, no. 6, pp. 720–738, JUN 1989, iSSN:0001-0782.
[12] M. Ferringer, "Coupling asynchronous signals into asynchronous logic," *Austrochip 2009, Graz, Austria*, http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1735.
[13] F. Bala and T. Nandy, "Programmable high frequency RC oscillator," in *18th International Conference on VLSI Design.*, Jan. 2005, pp. 511–515.

[14] M. Ferringer, G. Fuchs, A. Steininger, and G. Kempf, “VLSI Implementation of a Fault-Tolerant Distributed Clock Generation,” in *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006. DFT ’06.*, Oct. 2006, pp. 563–571.

[15] S. Fairbanks and S. Moore, “Self-timed circuitry for global clocking,” 2005, pp. 86 – 96.

[16] ——, “Analog micropipeline rings for high precision timing,” in *10th International Symposium on Asynchronous Circuits and Systems, 2004.*, April 2004, pp. 41–50.

[17] V. Zebilis and C. Sotiriou, “Controlling event spacing in self-timed rings,” in *11th IEEE International Symposium on Asynchronous Circuits and Systems, 2005. ASYNC 2005.*, March 2005, pp. 109–115.

[18] A. Winstanley, A. Garivier, and M. Greenstreet, “An event spacing experiment,” in *Eighth International Symposium on Asynchronous Circuits and Systems, 2002. Proceedings.*, April 2002, pp. 47–56.

[19] J. Ebergen, S. Fairbanks, and I. Sutherland, “Predicting performance of micropipelines using charlie diagrams,” mar-2 apr 1998, pp. 238 –246.

[20] M. Shimanouchi, “An approach to consistent jitter modeling for various jitter aspects and measurement methods,” in *Proceedings of IEEE International Test Conference, 2001.*, 2001, pp. 848–857.

[21] I. Zamek and S. Zamek, “Definitions of jitter measurement terms and relationships,” in *Proceedings of IEEE International Test Conference, 2005. ITC 2005.*, Nov. 2005, pp. 10 pp.–34.

[22] D. W. Allan, N. Ashby, and C. C. Hodge, “The Science of Timekeeping,” 1997, application Note 1289. http://www.allanstime.com/Publications/DWA/Science_Timekeeping/TheScienceOfTimekeeping.pdf.

[23] D. Howe, “Interpreting oscillatory frequency stability plots,” in *IEEE International Frequency Control Symposium and PDA Exhibition, 2002.*, 2002, pp. 725–732.

[24] B. Butka and R. Morley, “Simultaneous switching noise and safety critical airborne hardware,” in *IEEE Southeastcon, 2009. SOUTHEASTCON ’09.*, March 2009, pp. 439–442.

First Evaluation of FPGA Reconfiguration for 3D Ultrasound Computer Tomography

M. Birk, C. Hagner, M. Balzer, N.V. Ruiter
Institute for Data Processing and Electronics
Karlsruhe Institute of Technology
Karlsruhe, Germany
{birk, balzer, nicole.ruiter}@kit.edu

M. Huebner, J. Becker
Institute for Information Processing Technology
Karlsruhe Institute of Technology
Karlsruhe, Germany
{michael.huebner, becker}@kit.edu

Abstract—**Three-dimensional ultrasound computer tomography is a new imaging method for early breast cancer diagnosis. It promises reproducible images of the female breast in a high quality. However, it requires a time-consuming image reconstruction, which is currently executed on one PC. Parallel processing in reconfigurable hardware could accelerate signal and image processing. This paper evaluates the applicability of the FPGA-based data acquisition (DAQ) system for computing tasks by exploiting reconfiguration features of the FPGAs. The obtained results show, that the studied DAQ system can be applied for data processing. The system had to be adapted for bidirectional data transfer and process control.**

Keywords-Altera FPGAs, Reconfigurable Computing, 3D Ultrasound Computer Tomography

I. Introduction

Breast cancer is the most common type of cancer among women in Europe and North America. Unfortunately, an early breast cancer diagnosis is still a major challenge. In today’s standard screening methods, breast cancer is often initially diagnosed after metastases have already developed [1]. The presence of metastases decreases the survival probability of the patient significantly. A more sensitive imaging method could enable detection in an earlier state and thus, enhance survival probability.

At the Institute for Data Processing and Electronics (IPE) a three-dimensional ultrasound computer tomography (3D USCT) system for early breast cancer diagnosis is being developed [2]. This method promises reproducible volume images of the female breast in 3D.

Initial measurements of clinical breast phantoms with the first 3D prototype showed very promising results [3, 4] and led to a new optimized aperture setup [5], which is currently built and shown in Figure 1. It will be equipped with over 2000 ultrasound transducers, which are in particular 628 emitters and 1413 receivers. Further virtual positions of the ultrasound transducers will be created by rotational and translational movement of the complete sensor aperture.

In USCT, the interaction of unfocused ultrasonic waves with an imaged object is recorded from many different angles and afterwards computationally focused in 3D. During a measurement, the emitters sequentially send an ultrasonic wave front, which interacts with the breast tissue and is recorded by the surrounding receivers as pressure variations over time. These data sets, also called A-Scans, are sampled and stored for all possible sender-receiver-combinations, resulting in over 3.5 millions data sets and 20 GByte of raw data.

For acquisition of these A-Scans, a massively parallel, FPGA-based data acquisition (DAQ) system is utilized. After DAQ, the recorded data sets are transferred to an attached computer workstation for time-consuming image reconstruction steps. The reconstruction algorithms need a significant acceleration of factor 100 to be clinically relevant.

A promising approach to accelerate image reconstruction is parallel processing in reconfigurable hardware. This preliminary work investigates the applicability of the above mentioned DAQ system for further data processing tasks by a reconfiguration of the embedded FPGAs.

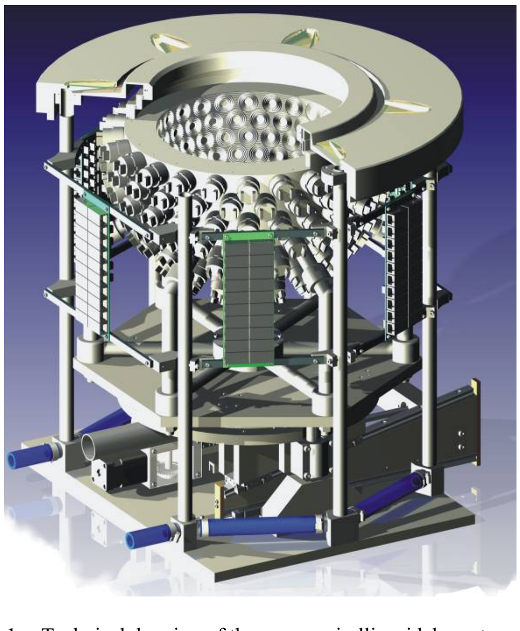


Figure 1. Technical drawing of the new semi-ellipsoidal aperture. It will be equipped with 628 ultrasound senders and 1413 receivers. During measurement, emitters sequentially send an ultrasonic wave front, which interacts with the breast and is recorded by the surrounding recievers.

The remainder of this paper is structured as follows: Section II describes the investigated FPGA-based DAQ system in detail. In Section III, the examined reconfiguration methodology is discussed. This includes derived design considerations and necessary system adaptations. Section IV illustrates an experimental procedure, which was used as proof of functionality of the used reconfiguration methodology and as a performance test of the DAQ system architecture. The paper is concluded in Section V. Therein, the attained performance results and limiting factors are discussed. Section VI gives an outlook into future work in this field of research.

II. Data Acquisition System

The investigated data acquisition (DAQ) system has been developed at IPE as a common platform for multi-project usage, e.g. in the Pierre Auger Observatory [6], the Karlsruhe Tritium Neutrino Project [7], and has also been adapted to the needs of 3D USCT. The DAQ system is described in detail in the following subsections.

A. *Setup & Functionality*

In the USCT configuration, the DAQ system consists of 21 expansion boards: one second level card (SLC) and 20 identical first level cards (FLC). Up to 480 receiver signals can be processed in parallel by processing 24 channels on each FLC, resulting in a receiver multiplex-factor of three. The complete system fits into one 19” crate, which is depicted in Figure 2. The SLC is positioned in the middle between 10 FLCs to the right and left, respectively.

The SLC controls the overall measurement procedure. It triggers the emission of ultrasound pulses and handles data transfers to the attached reconstruction PC. It is equipped with one Altera Cyclone II FPGA and a processor module (Intel CPU, 1 GHz, 256 MB RAM) running a Linux operating system.

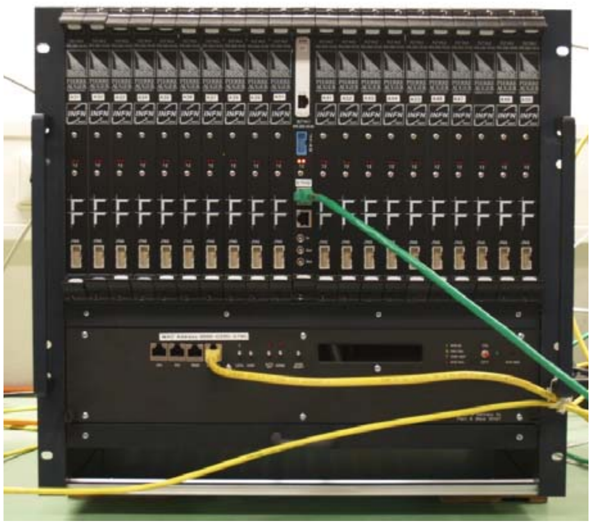


Figure 2. Image of the DAQ system in the USCT configuration. It is composed of one Second Level Card (SLC) for measurement control and communiation management (middle slot) and 20 First Level Cards (FLC) for parallel sensor signal aqcuision and data storage.

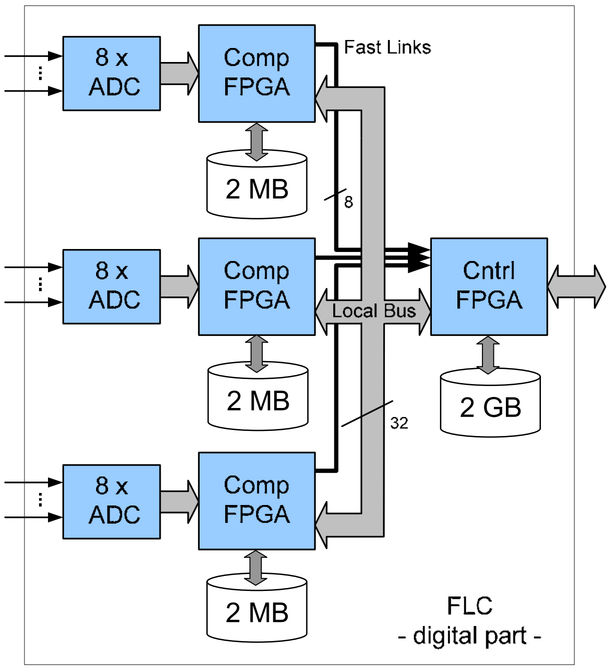


Figure 3. Block diagram of the digital part of an FLC in the 3D USCT DAQ system. It is equipped with four Altera Cyclone II FPGAs. One is used for local control (control FPGA, Cntr FPGA) and three for signal acuisition (computing FPGAs, Comp FPGA). Each Comp FPGA is fed by an 8fold ADC and is attached to a 2 MB QDR static RAM. The Cntrl FPGA is attached to an 2 GB DDRII dynamic RAM. There are two separate means of communication between the FPGAs: the slow local bus (Local Bus, 80 MB/s) and a fast data link (Fast Link, 240MB/s)

Communication with the attached PC is either possible via Fast Ethernet or an USB interface. For communication between SLC and the FLCs within the DAQ system a custom backplane bus is used.

B. *First Level Card*

A FLC consists of an analogue and a digital part. Only the digital part will be considered throughout this paper. A block diagram of this part is given in Figure 3. Besides three 8fold ADCs for digitization of the 24 assigned receiver channels, one FLC is equipped with four Altera Cyclone II FPGAs, which are used for different tasks:

- Control FPGA (Cntrl FPGA): One FPGA is used as local control instance. It handles communication and data transfer to the other FPGAs and to the SLC via backplane bus.
- Computing FPGA (Comp FPGA): The three other FPGAs are used for actual signal acquisition. Each of these is fed by one ADC and thus processes 8 receiver channels in parallel.

As intermediate storage for the acquired A-Scans, there are two different types of memory modules: Each computing FPGA is connected to a distinct static RAM module (QDRII, 2 MB each) and the control FPGA is attached to a dynamic RAM module (DDRII, 2 GB), summing up to a system capacity of 40 GB.

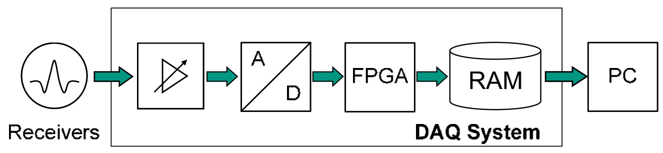


Figure 4. Data-flow of the DAQ system in the current configuration: the acquired signals from the ultrasound receivers are conditioned and afterwards digitized. They are digitally pre-filtered and decimated in the FPGAs and stored in on-board memory. After a complete measurement, they are transferred to an attached PC for further processing.

There are two separate means of communication between the control FPGA and the computing FPGAs, see also Figure 3: a slow local bus with a width of 32bit (Local Bus, 80 MB/s) and 8bit wide direct data links (Fast Links, 240MB/s per computing FPGA). Additionally, there are several connections for synchronization on board.

III. Methodology

As outlined in Section I, 3D USCT promises high-quality volumetric images of the female breast and has therefore a high potential in cancer diagnosis. However, it includes a set of time-consuming image reconstruction steps, limiting the method's general applicability.

To achieve a clinical relevance of 3D USCT, i.e. application in clinical routine, image reconstruction has to be accelerated by at least a factor of 100. A promising approach to reduce overall computation time is parallel processing of reconstruction algorithms in reconfigurable hardware.

In the current design, the DAQ system is only used for controlling the measurement procedure and acquisition of the ultrasound receiver signals. The overall data-flow of the DAQ system is shown in Figure 4. The acquired signals are conditioned and subsequently digitized, digitally pre-filtered and decimated in the FPGAs and afterwards stored in on-board memory. After a complete measurement cycle, the resulting data is transferred to an attached PC for signal processing and image reconstruction.

In this preliminary work, the utilization of the FPGAs in the DAQ system for further processing tasks has been investigated. Due to resource limitations, the full set of the processing algorithms cannot be configured statically onto the FPGAs in an efficient manner, either alone or even less in combination with the abovementioned DAQ functionality.

Therefore, a reconfiguration of the FPGAs is necessary to switch between different configurations, enabling signal acquisition and further processing on the same hardware system.

As the DAQ system has not been designed for further processing purposes, the scope of this work was to identify its capabilities as well as architectural limitations in this regard.

Only a reconfiguration of the FPGAs on the FLCs has been investigated, since these hold the huge majority of FPGAs within the complete system. Therefore, only these cards and their data-flow are considered in the following sections. Furthermore, an interaction of different FLCs has not been considered in this preliminary study.

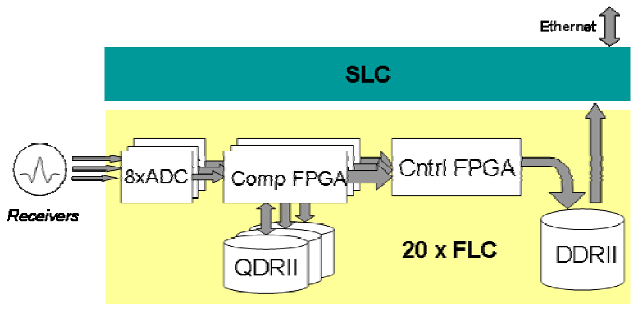


Figure 5. Detailed data-flow on one FLC during the conventional acquisition mode: Every FLC processes 24 receiver channels in parallel, whereas a group of 8 signals is digitized in a single ADC. The resulting digital signals are digitally filtered and averaged in the computing FPGAs. Finally, the signals are transmitted to the control FPGA and stored in DDRII memory.

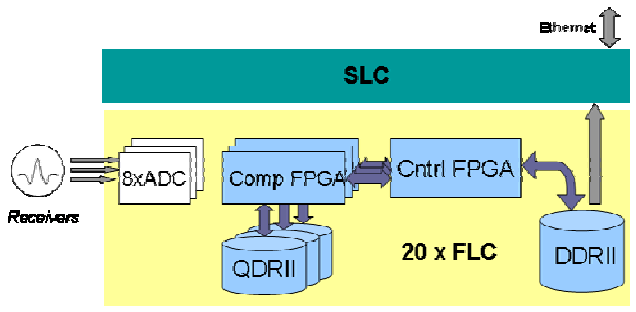


Figure 6. Detailed data-flow on one FLC during the newly craeated processing mode: As the data sets were previously stored in DDRII memory, they are transferred back to QDRII memory and processed in the computing FPGAs. Finally, the resulting data is stored again in DDRII memory.

The hardware setup of a FLC was given in Section II and shown in Figure 3. The detailed data-flow on a FLC in conventional operation mode is shown in Figure 5. During DAQ, 24 receiver channels are processed per FLC. The signals are split into groups of 8. Every group is digitized in one ADC and fed into one computing FPGA. Within a FPGA, the signals are digitally filtered and averaged by means of the attached QDR memory. Finally, the measurement data is transmitted via fast data links to the control FPGA, where it is stored in DDRII memory and afterwards transmitted to the SLC via backplane bus.

In this work, after completion of a measurement cycle, i.e. the data is stored in DDRII memory, the FPGAs were reconfigured to switch from conventional acquisition to data processing mode. As depicted in Figure 6, instead of transmitting the data sets via SLC to the attached PC, they were loaded back to QDR II and subsequently processed in the computing FPGAs. After completion, the resulting data was transmitted back to the control FPGA and again stored in DDRII memory. For providing this reconfiguration methodology, the following tasks had to be performed:

- Preventing data loss during reconfiguration
- Establishing communication and synchronization
- Implementing bidirectional communication interfaces

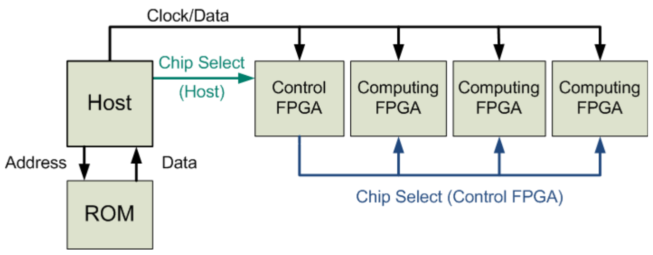


Figure 7. Passive serial configuration of a the FPGAs on a FLC at start-up time: first the control FPGA and then the computing FPGAs are configured in parall with data from an embedded configuration ROM.

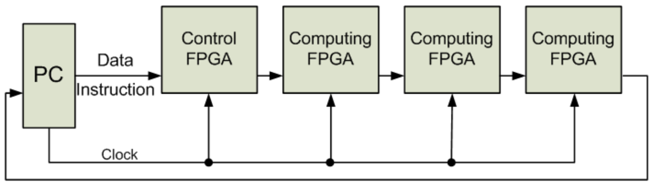


Figure 8. JTAG chain for reconfiguration of a the FPGAs on a FLC. In JTAG reconfiguration mode, each FPGA can be selected seperately for a reconfiguration, whereas deselected FPGAs remain in normal operation mode. Reconfiguration of FPGAs in chain takes place sequentially.

A. Preventing data loss during reconfiguration

The DAQ system is built up of Altera Cyclone II FPGAs, which do not allow partial reconfiguration [8]. Therefore, the complete FPGA chip has to be reconfigured. To prevent a loss of measurement data during the reconfiguration cycle, all data has to be stored outside the FPGAs in on-board memory, i.e. QDRII or DDRII memory.

The QDRII is static memory, so that stored data is not corrupted during reconfiguration of the FPGAs on the FLC. However, only the larger memory (DDRII) is capable of holding all the data sets recorded on one FLC. This dynamic memory module needs a periodic refresh cycle to keep stored data. On the FLC, the control FPGA is responsible for triggering these refresh cycles.

During a reconfiguration this FPGA is not able to perform this task. Since a refresh interval of the dynamic memory module is in the order of a few microseconds and a reconfiguration of the control FPGA takes even in the fastest mode about 100ms [8], it must not be reconfigured, or otherwise data in DDRII memory is lost.

Due to this requirement, only the three computing FPGAs are allowed to be reconfigured during operation. At a normal start-up of the DAQ system, all FPGAs on a FLC are configured via passive serial mode [8] with data from an embedded configuration ROM. As depicted in Figure 7, first the control FPGA and then all three computing FPGAs are configured in parallel in this mode.

In the current hardware setup it is not possible to exclude the control FPGA from a configuration in passive serial mode. Thus, each FPGA on the FLC has to be addressed and reconfigured separately, which is only possible in JTAG configuration mode [8] by using a JTAG chain through all four FPGAs as shown in Figure 8. In JTAG mode, each FPGA within the chain has to be configured sequentially.

B. Communication and Synchronization

Another important task in establishing the described reconfiguration methodology was to organize communication and control on the FLC as well as synchronization of parallel processing on the computing FPGAs.

As described in Section II, there are two means of communication between the computing FPGAs and the control FPGA, see also Figure 3: the slow local bus (Local Bus) and fast direct data links (Fast Links).

In conventional DAQ operation mode, measurement data is transmitted only in the direction from computing FPGAs to the control FPGA. Unfortunately, due to operational constraints in the FPGA pins, which are assigned for the fast links, this connection can only be used in the abovementioned sense, i.e. unidirectional.

Thus, in processing mode, the slower local bus has to be used for data transfer, since only this connection allows a bidirectional communication. The complete communication infrastructure is shown in Figure 9.

As the control FPGA is not reconfigured during operation, it must be statically configured to handle data transfer in each system state, i.e. DAQ as well as processing mode. Furthermore, it must be able to determine the current state in order to act appropriately. As also depicted in Figure 9, a single onboard spare connection (*conf_state*) is used for that purpose, which is connected to all four FPGAs.

In addition, each computing FPGA can be addressed and selected directly by the control FPGA via further point-to-point links to establish process control and synchronization. The respective *chip_select* signal triggers processing in a computing FPGA and by the *busy* signal completion of processing is indicated to the control FPGA.

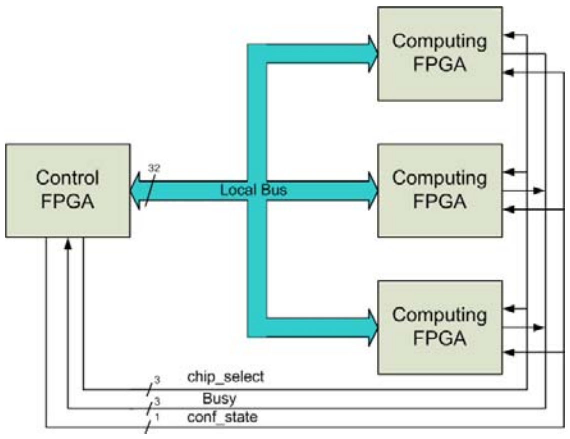


Figure 9. Communication structure during processing mode on a FLC: bidirectional data transfer is only possible via the slower Local Bus (80 MB/s). Separate point-to-point links (chip_select & busy) are used for control and synchronization of parallel processing. A further single point-to-point link is connected to all four FPGAs and indicates the current system state, i.e. DAQ or processing mode.

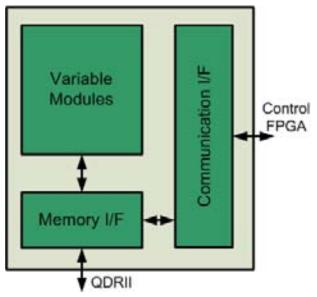


Figure 10. Block diagram of an computing FPGA: defined interfaces for communiation with the control FPGA (communication I/F) via Local Bus or Fast Links and access to QDRII memory (memory I/F). The variable algorithmic part is also indicated.

C. *Communication Interfaces*

A further task was structuring communication and memory interfaces in the computing FPGAs. As a result, modular interfaces for transmitting data over the Local Bus (communication I/F) and storing data in QDRII memory (memory I/F) were created. Figure 10 shows a block diagram of these modules on the computing FPGAs. The created modular design allows a simple exchange of algorithmic modules without the need to change further elements.

The communication interface is controlled by the control FPGA. It performs data transfers via Local Bus during processing or via fast data link during DAQ mode. The memory interface handles accesses to the QDRII memory. It can be either accessed by the control FPGA via Local Bus or by the algorithmic modules. In the current configuration, an algorithmic module only interacts with the memory interface and thus, only processes data which has already been stored in QDRII memory.

In order to guarantee a seamless data transfer over the Local Bus, the respective memory interface in the control FPGA had to be supplemented by a buffered access mode to the DDRII memory. When a data transfer is initialized, enough data words are pre-loaded into a buffer, so that the transmission is not interrupted during a refresh cycle.

IV. Experimental Results

The reconfigurable computing system was tested by acquisition of a test pulse. The used pulse was in the same frequency range as regular measurement data and was handled as a normal data set (A-Scan). This was followed by the reconfiguration of the computing FPGAs and an exemplary data processing. The main goals were to determine the required transfer time per data set over the Local Bus as well as reconfiguration times.

A. *Test setup*

For functional validation and performance measurements, a reduced setup of the complete DAQ system, containing a SLC but only one FLC was used. However, as in the current configuration only a single FLC without interactions with other FLCs has been considered, this setup allows a projection for a fully equipped system.

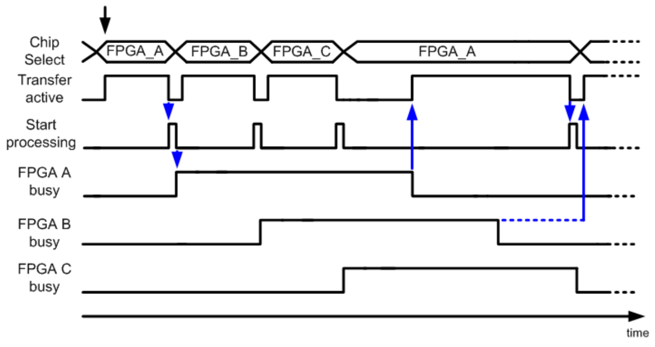


Figure 11. Detailed test procedure after DAQ and manual reconfiguration of the computing FPGAs via JTAG. Firstly, computing FPGA A is supplied with a set of A-Scans and processing on this FPGA is started. While processing is underway, also data transfer and processing on the computing FPGAs B and C is initiated. After completion of processing on FPGA A, the resulting data is tranferred back to DDRII memory and further unprocessed A-Scans are loaded. This scheme is applied repeatedly until all data sets are processed.

B. *Detailed Test Procedure*

The system has been tested as follows: At system start-up, the initial DAQ configuration was loaded into the FPGAs as outlined in Section IIa. Afterwards, the test pulse has been applied at the inputs of the ADCs on the FLC. This pulse was digitized and finally stored in DDRII memory.

The further detailed procedure is indicated in Figure 11. After a manual reconfiguration of the computing FPGAs via JTAG, the first set of A-Scans were transferred to the first computing FPGA (FPGA A) via Local Bus, stored in its attached QDRII memory and subsequently processed. While data in this FPGA is being processed, the other two computing FPGAs (FPGA B and FPGA C) are supplied with their initial data sets and processing on these FPGAs is started.

After completion of processing in FPGA A, the resulting A-Scan data was transmitted back to DDRII memory and subsequently further unprocessed A-Scans were sent to this FPGA. This scheme is repeatedly applied until all A-Scans have been processed.

C. *Results*

A JTAG configuration of a single computing FPGA requires 1.8s, resulting in a reconfiguration time of 5.4s for one FLC, when only the three computing FPGAs are configured in a JTAG chain. A reconfiguration of all 60 computing FPGAs, distributed over the 20 FLCs in the complete DAQ system would take up to 2 minutes by building up a JTAG chain through all FPGAs. The determined JTAG reconfiguration times are illustrated in Table I.

TABLE I. JTAG Reconfiguration Times

Procedure	Required time
Reconf. of one computing FPGA	1.8s
Reconf. of one FLC	5.4s
Extrapolated reconf. of the DAQ system	~2min

TABLE II. COMPUTING FPGA OCCUPATION

Components	Configuration		
	Data Acquisition	***Data Processing***	***Comm & Mem I/F only***
Logic Elements	5766 (17%)	9487 (29 %)	1231 (4%)
Embedded Multipliers	68 (97%)	64 (91%)	0 (0%)
Memory Bits	4236 (<1%)	393268 (81%)	0 (0%)

The transfer of one data set via Local Bus in either direction, i.e. from control FPGA to computing FPGA or vice versa, takes 75us. Usage of the Local Bus limited to one computing FPGA at a time and the same bus is used for data transfer to and from all three computing FPGAs.

Assuming the applied data parallel processing strategy, i.e. each computing FPGAs performs the same computation on a different data set, a high efficiency can only be reached, if the following condition holds:

The parallelized processing time per A-Scan on a computing FPGA has to be longer than 450us, which is 6 times the transmission time of a single data set. In this context, parallelized time is the processing time per A-Scan on one computing FPGA divided by the number of concurrently processed A-Scans on this FPGA. In this case, transfer time to and from all three FPGAs could be hidden. As observable in Figure 11, this requirement was not completely fulfilled by our exemplary data processing.

Table II outlines the occupation of the computing FPGA during the test procedure. The extensive use of embedded multipliers in DAQ mode, which are required due to the hard real-time constraints, state a clear demand for the established reconfiguration methodology. Furthermore, the implemented communication and memory interfaces are lightweight, occupying only 4% of the device's logic elements.

As the main result of this preliminary work, the possibility of reusing the existing DAQ system for data processing has been shown. By the reconfiguration of the FPGAs the functionality of the complete system has been increased.

V. CONCLUSIONS & DISCUSSION

In this paper, the concept of a reconfigurable computing system based on an existing DAQ system for 3D USCT has been presented.

The main drawback of the existing system is the slow data transfer over the Local Bus, which limits the achievable performance during the processing phase. This issue could partly be resolved by using a modified communication scheme, where data transfers from the computing FPGAs to the control FPGA is still done via Fast Links as in DAQ mode and the Local Bus is only used for the opposite direction.

Likewise, due to the long reconfiguration time, the number of reconfiguration cycles gives an essential contribution to the total processing time. To which extend this constraint will restrict the applicability of the presented methodology can not be assessed at this point and needs a further investigation. However, the reconfiguration time could be significantly reduced by separate JTAG chains for each FLC and concurrent reconfiguration.

VI. OUTLOOK

For future work, two obvious aspects have already been derived in the last section. Namely, reducing data transfer time by a modified communication scheme in the processing phase and reducing reconfiguration time by parallel JTAC chains.

Further tasks will also be porting processing algorithms to the DAQ system and thus, evaluating the established reconfiguration ability in real application.

What has not been considered so far is a direct communication between the computing FPGAs on a FLC and an interaction of different FLCs in general. This would open up manifold implementation strategies for algorithmic modules, besides the applied data parallel scheme.

In the long term, the next redesign of the DAQ system will put special focus on processing capabilities, e.g. high-speed data transfer.

VII. REFERENCES

[1] D. van Fournier, H.J.H.-W. Anton, G. Bastert, "Breast cancer screening", P. Bannasch (Ed.), Cancer Diagnosis: Early Detection, Springer, Berlin, 1992, pp. 78-87.

[2] H. Gemmeke and N. V. Ruiter, "3D ultrasound computer tomograph for medical imaging," Nucl. Instr. Meth., 2007

[3] N.V. Ruiter, G.F. Schwarzenberg, M. Zapf and H. Gemmeke, "Conclusions from an experimental 3D Ultrasound Computer Tomograph," *Nuclear Science Symposium*, 2008.

[4] N.V. Ruiter, G.F. Schwarzenberg, M. Zapf, A. Menshikov and H. Gemmeke, "Results of an experimental study for 3D ultrasound CT," *NAG/DAGA Int. Conf. On Acoustics*, 2009

[5] G.F. Schwarzenberg, M. Zapf and N.V. Ruiter, "Aperture optimization for 3D ultrasound computer tomography," *IEEE Ultrasonics Symposium*, 2007

[6] H. Gemmeke, et al., "First measurements with the auger fluorescence detector data acquisition system," *27th International Cosmic Ray Conference*, 2001.

[7] A. Kopmann, et al. "FPGA-based DAQ system for multi-channel detectors," *Nuclear Science Symposium Conference Record*, 2008.

[8] Altera Corporation, "Cyclone II Device Handbook", 2007

ECDSA Signature Processing over Prime Fields for Reconfigurable Embedded Systems

Benjamin Glas, Oliver Sander, Vitali Stuckert, Klaus D. Müller-Glaser, and Jürgen Becker
Institute for Information Processing Technology (ITIV)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: {glas,sander,becker,kmg}@itiv.uni-karlsruhe.de

Abstract—**Growing ubiquity and safety relevance of embedded systems strengthens the need to protect their functionality against malicious attacks. Communication and system authentication by digital signature schemes is a major issue in securing such systems. This contribution presents a complete ECDSA signature processing system over prime fields on reconfigurable hardware. The flexible system is tailored to serve as a autarchic subsystem providing authentication transparent for any application. Integration into a vehicle-to-vehicle communication system is shown as an application example.**

I. Introduction

With emerging ubiquitity of embedded electronic systems and a growing part of distributed systems and functions even in safety relevant areas the security of embedded systems and their communication gains importance quickly. One major concern of security is authenticity of communication peers and the information exchange. Especially if many different remote participants have to communicate or not all participants are known in advance, asymmetric signature schemes are beneficial for authentication purposes. In contrast to symmetric schemes like the Keyed-Hash Message Authentication Code HMAC [1], asymmetric signature schemes like RSA [2], DSA [3] and the ECDSA scheme [3] considered in this contribution get along without key exchange or predistributed keys, relaying usually on a certification authority as trusted third party instead.

This benefit comes at the cost of a much greater computational complexity of these schemes compared to authentication techniques based on symmetric ciphers or solely on hashing. This imposes major problems especially for embedded systems where resources are scarce.

This contribution presents a hardware implemented system for complete prime field ECDSA signature processing on FPGAs. It can be integrated as an autarchic subsystem for signature processing in embedded systems. As an application example the integration in a vehicle-to-vehicle communication system is presented.

The remainder of this paper is organized as follows. In the following section II some related work is given, section III presents basics of the implemented signature scheme ECDSA and section IV outlines the assumed situation and requirements for the system. The structure and implementation of the signature system itself is presented in section V and section VI shows an application example and integration in a wireless communication system. Section VII details on performance and resource usage. The contribution is concluded in section VIII.

II. Related work

Since elliptic curves where proposed as basis for public key cryptography in 1985 by Koblitz [4] and Miller [5] independently, many implementations of prime field ECDSA and elliptic curve cryptography (ECC) in general have been published. Software implementations on general purpose processors need lots of computation power. The eBACS ECRYPT benchmark [6] gives values for 256 bit ECDSA of e.g. 1.88 msec for generation and 2.2 msec for verification on a Intel Core 2 Duo at 1.4 GHz, and 2.9 msec resp. 3.4 msec on an Intel Atom 330 at 1.6 GHz. Values for a crypto system based on a ARM7 32-bit microcontroller are given in [7] for a key bitlength of 233 bit. Using a comb table precomputation ($w = 4$) 742 ms are needed for a generation and 1240 ms for a verification of an ECDSA signature.

To achieve usable throughputs and latencies on embedded systems, various specialized hardware has been proposed, e.g. many approaches for implementation of $\mathbb{F}_p$ arithmetic and the ECC primitives point add and point double on reconfigurable hardware. A survey of hardware implementations can be found in [8]. Güneysu et al present in [9] a very fast approach based on special DSP FPGA slices. The implementation presented here is based on a $\mathbb{F}_p$ ALU presented by Gosh et al [10].

Nevertheless open implementations of complete signature processing units performing complete ECDSA are scarce. Järvinen et al [11] present a Nios II based ECDSA system on an Altera Cyclone II FPGA for a key length of 163 Bit performing signature generation in 0.94 msec, verification in 1.61 msec.

This contribution presents an FPGA-based autarchic ECDSA system for longer key lengths of 256 bit containing all necessary subsystems for application in embedded systems on reconfigurable hardware.

III. ECDSA fundamentals

The Elliptic Curve Digital Signature Algorithm ECDSA is based on group operations on an elliptic curve E over a finite field $\mathbb{F}_q$. Mostly two types of finite fields are technically used:

Fields $\mathbb{F}_{2^n}$ of characteristic two and prime fields $\mathbb{F}_p$ with large primes p.

Algorithm 1 ECDSA signature generation

```
Input: Domain parameter $D = (q, a, b, G, n, h)$, secret key $d$, message $m$
Output: Signature $(r, s)$
1: Chose random $k \in [1, n-1], k \in \mathbb{N}$
2: Compute $kG = (x_1, y_1)$
3: Compute $r = x_1 \mod n$. If $r = 0$ goto step 1.
4: Compute $e = H(m)$
5: Compute $s = k^{-1}(e + dr) \mod n$. If $s = 0$ goto step 1.
6: return $(r, s)$.
```

For the use of ECDSA a set of common domain parameters is needed to be known to all participants. These are the modulus q identifying the underlying field, parameters a, b defining the elliptic curve E used, a base point $G \in E$, the order n of G and the cofactor $h = \frac{\text{order}(E)}{n}$. The signature generation and verification for a key pair (Q, d) can then be performed using the secret key d or the public key Q respectively. The procedures needed are shown in algorithms 1 and 2.

Algorithm 2 ECDSA signature verification

```
Input: Domain parameter $D = (q, a, b, G, n, h)$, public key $Q$, message $m$,
    signature $(r, s)$.
Output: Acceptance or Rejection of the signature
1: if $\neg(r, s \in [1, n-1] \cap \mathbb{N})$ then
2:     return "reject"
3: end if
4: Compute $e = H(m)$
5: Compute $w = s^{-1} \mod n$.
6: Compute $u_1 = ew \mod n$ and $u_2 = rw \mod n$.
7: Compute $X = (x_X, y_X) = u_1 G + u_2 Q$.
8: if $X = \infty$ then
9:     return "reject"
10: end if
11: Compute $v = x_X \mod n$.
12: if $v = r$ then
13:     return "accept"
14: else
15:     return "reject"
16: end if
```

IV. Setup and Situation

We assume an embedded system communicating with several peers which are not entirely known in advance. Therefore the exchanged signed messages are sent with a certificate attached that is issued by a commonly trusted certification authority.

This contribution focuses on prime field ECDSA as it is proposed also for the application example. Implemented are especially two elliptic curves recommended by the U.S. National Institute of Standards and Technology (NIST) in [12], namely the curves $p224$ and $p256$ with bitlengths 224 and 256 respectively and the corresponding domain parameters also given in the standard.

The proposed system works as a security subsystem exclusively performing signature processing and passing and receiving messages m to and from the external system.

V. Signature Processing System

Processing of ECDSA consists of several layers of computation. On the top level the signature generation and verification algorithms as well as the certificate validation are performed. These signature scheme-dependent layer is based on the group operations point add (PA) and point double (PD) in the underlying elliptic curve. These are in turn based on the underlying finite prime field ($\mathbb{F}_p$) arithmetic, that is modular arithmetic modulo a prime p. In a even higher layer there is also the communication protocol to consider at least partially as needed for the signature system.

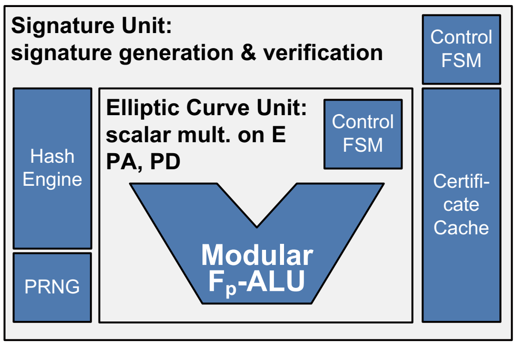


Fig. 1. Overview of the signature system

The architecture and presentation of the system reflects this layering. The two upper layers are implemented as finite state machines (FSM) and make use of a basic $\mathbb{F}_p$ arithmetic logical unit (ALU) and some additional auxiliary modules. Figure 1 outlines the structure of the system. The different building blocks are detailed in the following paragraphs.

A. $\mathbb{F}_p$ modular ALU

The central processing is done by a specialized $\mathbb{F}_p$-ALU for primes of maximum 256 bit length. It is based on the ALU proposed by Ghosh et al in [10]. Figure 2 depicts the structure. The ALU contains one $\mathbb{F}_p$ adder, subtractor, multiplier and divider/inverter each. All registers and datapaths between the modules are 256 bit wide so that complete operands up to 256 bit width (as in the $p256$ case) can be stored and transmitted within a single clock cycle. Four Inputs, two outputs and four combined operand/result register as well as a flexible interconnect allow for a start of two operations each at the same time as long as they do not use the same basic arithmetic units. The units perform operations independently, so that using different starting points parallel execution in all four subunits is possible. This allows for parallelisation in the scalar multiplication (see paragraph V-B1).

The $\mathbb{F}_p$-adder and -subtractor perform each operation in a single clock cycle as a general addition/subtraction with subsequent reduction. The $\mathbb{F}_p$ multiplying module computes the modular multiplication iteratively as shift-and-add with reduction mod p in every step. It therefore needs $|p|$ clock cycles for one modular multiplication, $|p|$ being the bitlength

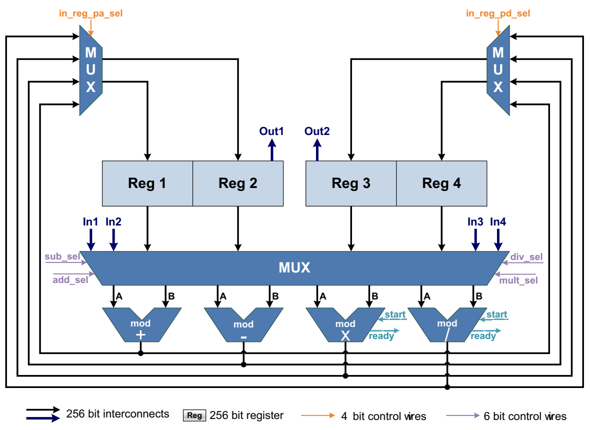


Fig. 2. Schematic overview of the $\mathbb{F}_p$ ALU

of the modulus and thereby also the maximum bitlength of the operands.

Modular inversion and division is the most complex task of the ALU. It is based on a binary division algorithm on $\mathbb{F}_p$, see [10] for details. The runtime depends on the input values, maximum runtime being $2|p|$ clock cycles, in the $p256$ case therefore up to 512 cycles. Statistical analysis showed an average runtime of $1.5 \cdot |p|$ clock cycles.

ALU control is performed over multiplexer and module control wires and is implemented as finite state machine presented in the following paragraph. The complete ALU allocates 14256 LUT/FF pairs in a Xilinx Virtex-5 FPGA and allows for a maximum clock frequency of 41.2 MHz (after synthesis).

In addition to the 256 bit arithmetic based on the modulus $p256$ the ECDSA unit also implements the arithmetic for modulus $p224$. This can be done using the same hardware and is also implemented in the overlaying FSM. Theoretically all moduli up to 256 bit width are supported by the ALU. Details on resource consumption and performance values are given in section VII.

B. *Elliptic curve processing*

On the elliptic curve E addition of points is defined as group operation. Doubling of a point is specially implemented as it requires a different computation because general point addition is not defined with operands being equal. A comprehensive introduction to elliptic curve arithmetic including algorithms can be found in [13].

The operation schedules for point addition and point doubling for execution on the ALU are given in tables I and II.

The execution schedules map the operations to the executing units using three auxiliary register t_1, t_2, t_3 for storing intermediate results.

1) Scalar multiplication on E: Scalar multiplication is the central step 2 of the signature generation algorithm 1. Computation is done iteratively using the so-called Montgomery ladder [14], [15] showed in algorithm 3.

Step	$\mathbb{F}_p$ unit	# cycles
1. $t_1 = y_2 - y_1$	sub	1
2. $t_2 = x_2 - x_1$	sub	1
3. $t_2 = t_1/t_2 (= \lambda)$; $t_3 = x_1 + x_2$	div; add	max. $2\|p\|$
4. $t_1 = t_2 \cdot t_2$	mult	$\|p\|$
5. $t_1 = t_1 - t_3 (= x_3)$	sub	1
6. $t_1 = x_1 - t_1$	sub	1
7. $t_1 = t_2 \cdot t_1$	mult	$\|p\|$
8. $t_1 = t_1 - y_1 (= y_3)$	sub	1
		max. $4\|p\| + 5$

TABLE I
HARDWARE EXECUTION OF POINT ADDITION

Step	$\mathbb{F}_p$ unit	# cycles
01. $t_1 = x_1 \cdot x_1$	mult	$\|p\|$
02. $t_2 = t_1 + t_1$	add	1
03. $t_1 = t_1 + t_2$	add	1
04. $t_1 = t_1 + a$	add	1
05. $t_2 = y_1 + y_1$	add	1
06. $t_2 = t_1/t_2 (= \lambda)$; $t_3 = x_1 + x_1$	div; add	max. $2\|p\|$
07. $t_1 = t_2 \cdot t_2$	mult	$\|p\|$
08. $t_1 = t_1 - t_3 (= x_3)$	sub	1
09. $t_1 = x_1 - t_1$	sub	1
10. $t_1 = t_2 \cdot t_1$	mult	$\|p\|$
11. $t_1 = t_1 - y_1 (= y_3)$	sub	1
		max. $5\|p\| + 7$

TABLE II
HARDWARE EXECUTION OF POINT DOUBLING

The operations in the branches inside the FOR-loop, meaning steps 5 and 6 in the IF-branch rsp. 8 and 9 in the ELSE-branch can be executed in parallel. Since it is a point addition and a point doubling each, a real parallel execution on the ALU is possible using a tailored scheduling. Figure 3 depicts the implemented schedule. Execution time is therefore at maximum $((|p| - 1) \cdot (6|p| + 7) + (5|p| + 7)) = 6|p|^2 + 6|p|$ clock cycles for the combination of point add and point double.

2) Double scalar multiplication: For verification of ECDSA signatures two independent scalar multiplications have to be executed (see algorithm 2 step 7). Instead of computing independently in sequence it is faster to compute them together using an approach published originally by Shamir [16] also known as "Shamirs trick" shown in algorithm 4.

Algorithm 3 Scalar multiplication in E

Input: Point $P \in E$; Integer $k = \sum_{i=0}^{l-1} k_i 2^i$ with $k_i \in \{0, 1\}$ and $k_{l-1} = 1$.
Output: Point $Q = kP \in E$.

```
 1: $P_1 = P$
 2: $P_2 = 2P$
 3: for $i = l - 2$ downto 0 do
 4:   if $k = 0$ then
 5:     $P_1^{new} = 2P_1^{old}$
 6:     $P_2^{new} = P_1^{old} + P_2^{old}$
 7:   else
 8:     $P_1^{new} = P_1^{old} + P_2^{old}$
 9:     $P_2^{new} = 2P_2^{old}$
10:   end if
11: end for
12: return $Q = P_1$
```

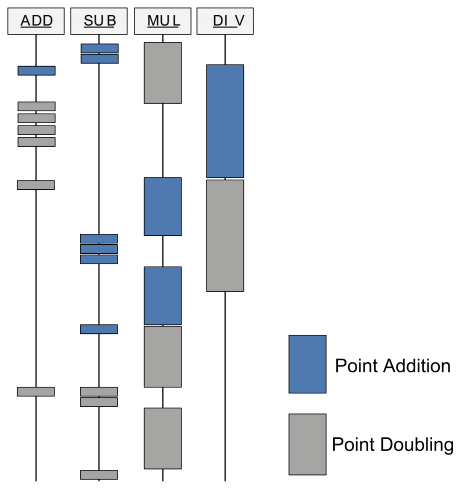


Fig. 3. Parallel Scheduling of PA and PD

Algorithm 4 Simultaneous multiple point multiplication

Input: Point $P, Q \in E$; Integers $k = \sum_{i=0}^{l-1} k_i 2^i$ and $m = \sum_{i=0}^{l-1} m_i 2^i$ with $k_i, m_i \in \{0, 1\}$ and $k_{l-1} \vee m_{l-1} = 1$.
Output: Point $X = kP + mQ \in E$.

```
1: Precomputation: $P + Q$
2: $X = O$ (point at infinity)
3: for $i = l - 2$ downto 0 do
4:     $X = 2X$
5:     $X = X + (k_i P + m_i G)$
6: end for
7: return $X$
```

In contrast to algorithm 3 the central operations in steps 4 and 5 cannot be parallelized as they depend on each other. The maximum time consumption of the algorithm is therefore $((4|p| + 5) + |p| \cdot ((5|p| + 7) + (4|p| + 5))) = 9|p|^2 + 16|p| + 5$ clock cycles. This is nevertheless less than the $2 \cdot (6|p|^2 + 6|p|) + (4|p| + 5)) = 12|p|^2 + 16|p| + 5$ cycles two independent scalar multiplications would consume. Assuming uniform distribution step 5 is omitted in 25% of the cases leaving an estimated runtime of $((4|p| + 5) + |p| \cdot ((5|p| + 7) + 0,75 \cdot (4|p| + 5))) = 8|p|^2 + 14,75|p| + 5$ clock cycles.

C. Signature and certificate control system

On top of the elliptic curve (EC) operations and the control FSM performing them the actual signature algorithms and the certificate verification are implemented. This is done in a seperate FSM (see figure 1), controlling the EC arithmetic FSM, some registers and the auxiliary hashing and random number generation. Figure 4 shows the sequence of operations of the signature verification. See algorithms 1 and 2 for the implemented procedures.

This FSM is the upmost layer of the signature module and provides a register interface for operands like messages, signatures, certificates and keys. For integration in an embedded system it has to be wrapped to support the message format and create the inputs to select the function needed. An example for an integration is given in section VI.

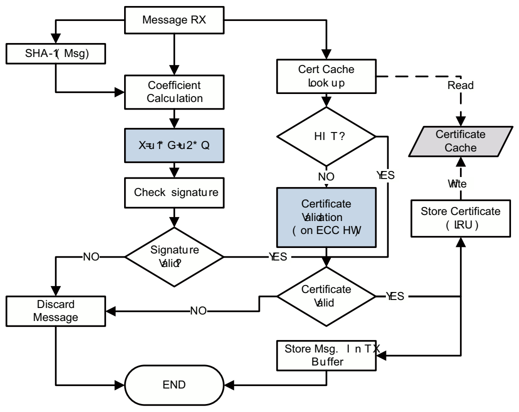


Fig. 4. Procedure for signature verification

D. SHA2 Hashing Module

The SHA2 hashing unit provides functions SHA-224 and SHA-256 according to the Secure Hash Algorithm (SHA) standard [17]. It is based on a freely available verilog SHA-256 IP-core[1] adapted with a wrapper performing precomputation of the input data and providing a simple register interface accepting data in 32 bit chunks. In addition the core has been enhanced to support SHA-224. Figure 5 shows an overview.

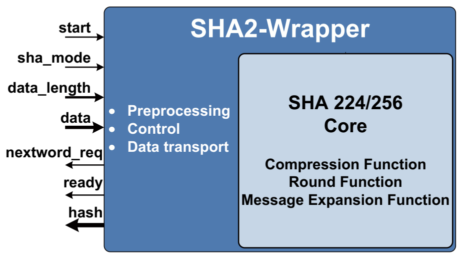


Fig. 5. Schematic overview of hashing submodule

The unit processes input data in blocks of 512 bit needing 68 clock cycles each at a maximum clock frequency of 120 MHz (after synthesis) and a ressource usage of 2277 LUT/FF-pairs. After finishing the operation the result is available in a 256 bit register.

E. Pseudo-Random Number Generation

For ECDSA signature generation, a random value k is needed. To provide this k the system incorporates a Pseudo-Random Number Generator (PRNG) consisting of a two linear feedback shift registers (LFSR), one with 256 bit length, feedback polynomial $x^{255} + x^{251} + x^{246} + 1$ and a cycle length of $2^{256} - 1$ [18] and a second LFSR with 224 bit length, feedback polynomial $x^{222} + x^{217} + x^{212} + 1$ and a cycle length of $2^{224} - 1$.

The LFSR occupies 480 LUT/FF pairs and allows for a maximum clocking of 870 MHz although operated in the

[1]Available as *SHA IP Core* at http://www.opencores.com

system in the general system clock of 50 MHz. It is operated continuously to reduce predictability of the produced numbers. The current register content is read out on demand.

F. Certificate Cache

Usually digital signatures or their respective public key needed for verification are endorsed by a certificate issued by a trusted third party, a so-called certification authority (CA), to prove its authenticity. Verification of the certificate requires a signature verification itself and is therefore equally complex than the main signature verification of the message. If communicating several messages with the same communication peer using the same signature key the certificate can be stored hence saving the effort for repetitive verification.

The system incorporates a certificate cache for up to 81 certificates stored in two BRAM blocks. It can be searched in parallel to the signature verification. Replacement of certificates is performed using a least recently used (LRU) policy.

VI. Application Example

The system offers complete ECDSA signature and certificate handling and can be used in a variety of embedded systems seeking authentication and security of communication. As an application example we show the integration into a vehicle-to-X (V2X) communication system. V2X communication is an emerging topic aiming at information exchange between vehicles on the road and between vehicles and infrastructure like roadside units. This can be used to enhance safety on roads, optimize traffic flow and help to avoid traffic congestions.

To be able to base decisions and applications on information received from other vehicles, trustworthiness of this information is mandatory. To ensure the validity and authenticity of information, signature schemes are used to protect the messages broadcasted by the participating vehicles against malicious attacks. As V2X communication is at present in the process of standardization, no fixed settings are available yet, but the use of ECDSA is proposed in the IEEE 1609 Wireless Access in Vehicular Environments standard draft [19] as well as the proposals of european consortia [20].

In the chosen realization V2X communication is performed by a modular FPGA-based On-Board-Unit (OBU) presented in [21]. It consists of different functional modules connected by a packet-based on-chip communication system [22]. The signature verification system is integrated as a submodule and performs signature handling for incoming and outgoing messages automatically, being therefore transparent to the other modules except for the unavoidable processing latency. It is connected to two different on-chip communication systems, one transmitting unsecured messages and the other transmitting only secured messages containing signatures and certificates. A short description of the security system and its system integration is given in [23]. Figure 6 depicts the wrapped signature system with the interfacing to both communication structures.

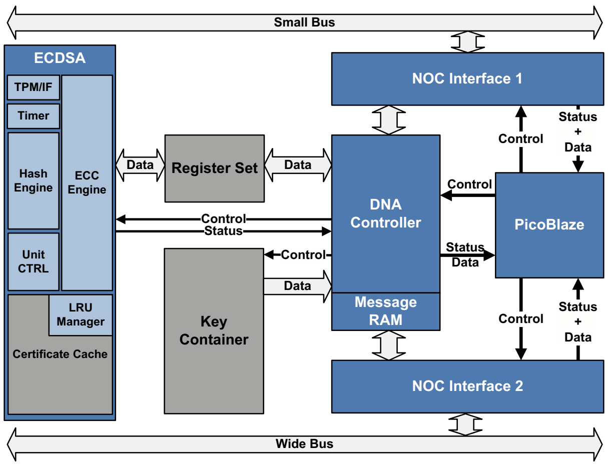


Fig. 6. Wrapping of the signature system for V2X-integration

The signature system accepts incoming messages, verifies signatures and certificates and passes only verified messages on to further processing. In case of an invalid signature the outer system is informed. For outgoing messages signatures are generated and the corresponding certificate is attached to the message which is then passed on to the wireless interface.

A. Key container

In the V2X environment privacy of participants is of major importance. As messages containing vehicle type and values like current position, speed and heading are continuously broadcasted from twice to up to ten times a second, these messages could easily be used by an eavesdropper to trace participants. To counter such attempts anonymity in the form of pseudonyms is used that are changed on a regular basis. A number of pseudonymes for change are stored directly in the signature modules key container (see figure 6). It also contains the public keys of trusted certification authorities needed for verification of certificates. The change itself is triggered by a dedicated message sent to the signature processing system by the central information processing module of the C2X system. For all other modules this privacy function is fully transparent as well.

VII. Resources and Performance

The presented system has been realized using a Xilinx XC5VLX110T Virtex-5 FPGA on a Digilent XUP ML509 evaluation board. The following values refer to an implementation of the complete signature generation and verification unit with interfacing for the application example given above. Table III shows an overview of the resource usage.

After integration of all submodules the ECDSA unit allows for a maximum clock frequency of 50 MHz that has been successfully tested. Table IV shows signature verification performance values of the ECDSA unit at 50 MHz. Values for signature generation are given in table V.

In both tables the *worst case* values given are calculations based on the statistically estimated runtime of the algorithms

	Lut-FF Pairs *(Synthesis)*	**rel. res. usage on FPGA**	**max. frequency [MHz]**
Signature unit	32,299	46.7%	50
ECDSA unit	24,637	36%	50.1
Hashing unit	2,277	3%	120.8
PRNG	482	0.7%	872.6
$\mathbb{F}_p$-ALU	14,256	20%	41.2
$\mathbb{F}_p$-ADD	858	1.2%	83
$\mathbb{F}_p$-SUB	857	1.2%	92.8
$\mathbb{F}_p$-MUL	2,320	3.4%	42.3
$\mathbb{F}_p$-DIV	5,670	8.2%	73.4

TABLE III
RESOURCE USAGE ON A XC5VLX110T WITH 69,120 LUTS

Verification		**secp224r1**	**secp256r1**
Compute time	*worst-case*	7,23	9,42
[*ms/Sig*]	*simulated*	7,17	9,09
Throughput	*worst-case*	138	106
[*Sig/s*]	*simulated*	140	110
Latency	*worst-case*	361151	471111
[*cycles/Sig*]	*simulated*	358478	454208

TABLE IV
PERFORMANCE OF SIGNATURE VERIFICATION AT 50 MHZ

for scalar multiplication. As these runtimes depend on the operand values, the measured computation times are different.

Generation		**secp224r1**	**secp256r1**
Compute time	*worst-case*	5,56	7,26
[*ms/Sig*]	*simulated*	5,45	7,15
Throughput	*worst-case*	180	138
[*Sig/s*]	*simulated*	184	140
Latency	*worst-case*	278097	362881
[*Cycles/Sig*]	*simulated*	272345	357315

TABLE V
PERFORMANCE OF SIGNATURE GENERATION AT 50 MHZ

VIII. CONCLUSION AND FURTHER WORK

We presented a hardware implemented subsystem for ECDSA signature processing for integration into embedded systems based on reconfigurable hardware. It can be integrated as a stand-alone subsystem performing transparent authentication functionality for communication systems. Applicability of the system has been shown using vehicle-to-X communication as a practical example.

The performance values presented in section VII are sufficient for applications like entry control systems or electronic payment where the number of communication peers is small. For V2X communication even larger throughput up to over 2000 signatures per second are necessary. Further work therefore includes speeding up the computation and reducing the footprint. Also the use of low cost FPGAs is required for the use in embedded systems.

REFERENCES

[1] FIPS, "Pub 197: Advanced Encryption Standard (AES)," federal information processing standards publication, U.S. Department of Commerce, Information Technology Laboratory (ITL), National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2001.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[3] FIPS, "Pub 186-3: Digital signature standard (dss)," federal information processing standards publication, U.S. Department of Commerce, Information Technology Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2009.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[5] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology: CRYPTO '85 Proceedings*, pp. 417–426, 1986.

[6] eBACS, "ECRYPT Benchmarking of Cryptographic Systems," website, http://bench.cr.yp.to/ebats.html, 2010.

[7] M. Drutarovsky and M. Varchola, "Cryptographic System on a Chip based on Actel ARM7 Soft-Core with Embedded True Random Number Generator," in *DDECS '08: Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, (Washington, DC, USA), pp. 1–6, IEEE Computer Society, 2008.

[8] G. M. de Dormale and J.-J. Quisquater, "High-speed hardware implementations of Elliptic Curve Cryptography: A survey," *Journal of Systems Architecture*, vol. 53, pp. 72–84, 2007.

[9] T. Güneysu and C. Paar, "Ultra high performance ecc over nist primes on commercial fpgas," in *CHES*, pp. 62–78, 2008.

[10] S. Ghosh, M. Alam, I. S. Gupta, and D. R. Chowdhury, "A robust gf(p) parallel arithmetic unit for public key cryptography," in *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, (Washington, DC, USA), pp. 109–115, IEEE Computer Society, 2007.

[11] Kimmo Järvinen, Jorma Skyttä, "Cryptoprocessor for Elliptic Curve Digital Signature Algorithm (ECDSA)," tech. rep., Helsinki University of Technology, Signal Processing Laboratory, 2007.

[12] NIST, "Recommended elliptic curves for federal government use," tech. rep., National Institute of Standards and Technology, U.S. Department of Commerce, 1999.

[13] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004.

[14] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[15] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 177, pp. 243–264, january 1987.

[16] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, vol. 31, july 1985.

[17] FIPS, "Pub 180-2: Secure hash standard (shs)," federal information processing standards publication, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Gaithersburg, USA, 2002.

[18] Roy Ward, Tim Molteno, "Table of Linear Feedback Shift Registers," 2007. electronically available at http://www.otagophysics.ac.nz/px/research/electronics/papers/technical-reports/lfsr_table.pdf.

[19] IEEE Vehicular Technology Society, ITS Committee, *IEEE Trial-Use Standard for Wireless Access in Vehicular Environments (WAVE) - Security Services for Applications and Management Messages*, 06.07.2006.

[20] COMeSafety Project, "European ITS Communication Architecture - Overall Framework," 2008. available at www.comesafety.org.

[21] O. Sander, B. Glas, C. Roth, J. Becker, and K. D. Müller-Glaser, "Design of a vehicle-to-vehicle communication system on reconfigurable hardware," in *Proceedings of the 2009 International Conference on Field-Programmable Technology (FPT09)*, pp. 14–21, Institute of Electrical and Electronics Engineers, IEEE, Dec. 2009.

[22] O. Sander, B. Glas, C. Roth, J. Becker, and K. D. Müller-Glaser, "Priority-based packet communication on a bus-shaped structure for FPGA-systems," in *DATE*, 2009.

[23] B. Glas, O. Sander, V. Stuckert, K. D. Müller-Glaser, and J. Becker, "Car-to-car communication security on reconfigurable hardware," in *VTC2009-Spring: Proceedings of the IEEE 69th Vehicular Technology Conference*, (Barcelona, Spain), 2009.

A Secure Keyflashing Framework for Access Systems in Highly Mobile Devices

Alexander Klimm, Benjamin Glas, Matthias Wachs, Jürgen Becker, and Klaus D. Müller-Glaser
Institut für Technik der Informationsverarbeitung, Universität Karlsruhe (TH)
email: {klimm,glas,mueller-glaser,becker}@itiv.uni-karlsruhe.de

Abstract—**Public Key Cryptography enables for entity authentication protocols based on a platform's knowledge of other platforms' public key. This is particularly advantageous for embedded systems, such as FPGA platforms with limited or none read-protected memory resources. For access control to mobile systems, the public key of authorized tokens need to be stored inside the mobile platform. At some point during the platform's lifetime these might need to be updated in the field due to loss or damage of tokens. This paper proposes a secure scheme for key flashing of Public Keys to highly mobile systems. The main goal of the proposed scheme is the minimization of online dependencies to Trusted Third Parties, certification authorities, or the like to enable for key flashing in remote locations with only minor technical infrastructure. Introducing trusted mediator devices, new tokens can be authorized and later their public key can be flashed into a mobile system on demand.**

I. Introduction

Embedded systems in various safety critical application domains like automotive, avionic and medical care perform more and more complex tasks using distributed systems like networks of electronic control units (ECUs). The introduction of Public-Key Cryptography (PKC) to embedded systems provides essential benefits for the production of electronic units needing to meet security requirements as well as for the logistics involved. Due to the nature of PKC, the number of keys that need to be stored in the individual platforms is minimized. At the same time only the private key of the platform itself needs to be stored secretly inside each entity - in contrast to symmetric crypto systems where a secret key needs to be stored inside several different entities at the same time. In context of PKC, if one entity is compromised, the others remain uneffected.

Computational efforts of cryptographic functionalities are very high and time consuming if carried out on today's standard platforms (i.e. microcontrollers) for embedded applications. Integrating security algorithms into FPGA platforms provides for high speed up of demanding PKC crypto systems such as *hyperelliptic curve cryptography* (HECC). By adding dedicated hardware modules for certain parts of a crypto algorithm, a substantial reduction of computation time can be achieved [10] [9].

Besides encrypting or signing messages, PKC can be employed to control user access to a device via electronic tokens. Examples for this are Remote Keyless Entry (RKE) systems [17] in the automotive domain, or Hilti's TPS technology [2]. These systems incorporate contactless electronic tokens that substitute classical mechanical keys. The owner or authenticated user identifies himself to the user device (`UD`) by possession of the token. The `UD` and token are linked. Only if a linked token is presented to the `UD` it is enabled or access to the `UD` is granted. In order to present a token to a `UD`, information needs to be exchanged between the two over an usually insecure channel. To prevent the usage of a device or its accessibility through an unauthorized person this data exchanged needs to be secured.

Authentication schemes based on Public Key Cryptography such as the Needham-Schroeder protocol [11], Okamoto Protocol [12], and Schnorr-Protocol [16] provide authentication procedures where no confidential data needs to be transmitted. Secret keys need only be stored in the tokens and not in the `UD` thus omitting the need for costly security measures in the `UD`. Only public keys need to be introduced into the `UD` (see section II). This operation certainly does need to be secured against attacks. For real-world operation this operation is done in the field where the `UD` is not necessarily under the control of the manufacturer (`OEM`) and a live online connection to the `OEM` is not possible.

In this paper we propose a system to introduce public keys into FPGA based user devices to pair these with a new token. The proposed key flashing method allows for authorization of the flashing process through an `OEM`. At the same time it can be carried out with the `UD` in the field and with no active online connection while flashing the key.

Introduction or flashing of new keys to an embedded device can be seen as a special case of a software update. Here the main focus is usually on standardization, correctness, robustness, and security. Recent approaches for the automotive area have been developed e.g. in the german HIS [8], [7] or the EAST-EEA [3] project. A general approach considering security and multiple software providers is given in [1]. Nevertheless general update approaches are focused on the protection of IP and the provider against unauthorized copying and less on the case that the system has to be especially protected against unwanted updates as in our keyflashing scenario.

The remainder of this paper is structured as follows. In section II we present the basic application scenario followed by a short introduction of public key cryptography in section III. The requirements for the targeted scenario are described in IV. In section V the protocol is shown and some implementational results are given in section VI. We conclude in section VII.

II. Application Scenario

A mobile user device (UD) such as a vehicle, construction machine or special tool with restricted access is fitted with an FPGA based access control system. This allows only the owner or certified users access to the device's functionalities or even the device itself. This is achieved with a transponder (TRK) serving as an electronic version of a mechanical key. The transponder is able to communicate to the UD via a wireless communication channel. The user device accepts a number of transponders. If one of these is presented to the user device it authenticates the transponder and the device is unlocked, thus granting access.

Authentication is done using Public Key Cryptography (PKC). The Public Key of the transponders are stored securely inside the user device thus establishing a "'guest list"' of legal users to the device. During production two initial Public Keys are introduced into the user device.

In case of loss of a transponder it is desirable to replace it, particularly if the user device itself is very costly or actually irreplaceable. Since the user device is mobile, replacement of the transponder's public key usually needs to be done in the field. This might include very remote areas with minor to none communication infrastructure.

III. Basic PKC Functionalities

In 1976, Whitfield Diffie and Martin Hellman introduced PKC crypto systems [6]. Two different keys are used, one public and the other secret (SK). SK and VK are a fixed and unique keypair. It is computational infeasible to deduce the private or secret key (SK) from the public key[1] (VK). With VK a message M_p can be encrypted into M_c but not decrypted with the same key. This can only be done with knowledge of SK. If an entity Alice wants to transmit a message $M_{Alice,plain}$ to an entity Bob, it encrypts it with Bobs public key VK_{Bob}. Only Bob can retrieve the plain text from the encrypted message, by applying the appropriate decryption algorithm using his own secret key SK.

PKC can also be used to digitally sign a message. For this a signature scheme is used that is usually different from the encryption scheme. When signing a message the secret key is used and the signature can be verified by using the according public key. In other words, if Bob wants to sign a message, he uses his own private key that is unique to him. This key is used to encrypt the hash value of the message $M_{Bob,plain}$. The resulting value $\{HASH(M_{Bob,plain})\}_{sig}$ is transmitted the together with $M_{Bob,plain}$. A receiver can validate the signature by using Bob's public key and retrieving $HASH(M_{Bob,plain})$. From $M_{Bob,plain}$ the receiver can reconstruct the received hash value and compare it with the decrypted value. If both match the signature has been validated.

IV. Situation and requirements analysis

In our application scenario we have the following main entities:

[1]In the case of signature schemes the public key is often called verification key.

- A user device UD that can only be accessed or used by an authenticated user
- A human user OWN. He is authorized to access or use UD if he possesses a legit token
- A transponder key token TRK_{orig} originally linked to UD and a second token TRK_{new} that shall be flashed to UD additionally.
- The manufacturer OEM that produces UD

UD accepts a number of TRK to identify an authenticated human user OWN of the UD. At least two tokens are linked to a UD, by storing the respective public keys VK_{TRK} inside the UD. The OEM is initially the only entity allowed to write public keys into any UD.

Solely the public keys stored inside the UD are used for any authorization check of TRKs using any PKC authentication protocol (e.g. [11], [12], [16]). The OEM's public key VK_{OEM} is stored in the UD as well.

OEM, TRK, and UD can communicate over any insecure medium, through defined communication interfaces.

A. Goals

A new TRK_{new} should be linked to a UD to substitute for an original TRK_{orig} that has been lost or is defective. From this point on we'll call the process of linking TRK_{new} to a UD *flashing*. Flashing a TRK should be possible over the complete life cycle of the UD. When flashing the UD it is probably nowhere near the OEM's location while flashing of a TRK needs to be explicitly authorized by the OEM. Any TRK can only be flashed into a single UD. Theft or unauthorized use of the UD resulting from improper flashing of the TRK needs to be prohibited.

In addition we demand that online connection of UD and OEM during flashing a TRK must not be imperative.

B. Security Requirements

The protocol shall allow dependable authorized flashing under minimal requirements while preventing unauthorized flashing reliably. Therefore it has to guarantee the following properties, while assuming communication over an unsecured open channel:

- **Correctness:** In absence of an adversary the protocol has to deliver the desired result, i.e. after complete execution of the protocol the flashing should be accomplished.
- **Authentication:** The flashing should only be feasible if both OEM and OWN have been authenticated and have authorized the operation.
- **No online dependency:** The protocol shall not rely on any live online connection to the OEM.
- **Confidentiality:** No confidential data like secret keys should be retrievable by an adversary.

C. Adversary model

We assume an in processing power and memory polynomially bounded adversary $\mathcal{A}$ that has access to all inter-device communications, meaning he can eavesdrop, delete, delay, alter, replay or insert any messages. We assume further that the adversary is attacking on software level without tampering with the participating devices.

Without choosing particular instances of the cryptographic primitives we assume that the signature scheme used is secure against existential forgery of signatures and regard the hashing function used as a random oracle.

V. Key Flashing Concept

Focus of the proposed key flashing protocol is the introduction of a public key $\mathtt{VK}_{TRK}$ into UD. We abstract over the implementation of communication interfaces, PKC systems as well as the immediate implementation of the devices and entities themselves.

Two basic Flashing Scenarios are conceivable. One is that TRKs are flashed directly by the OEM, either during production or via an online connection. We concentrate on the second one, flashing of TRKs through an authorized service point (SP) with no immediate online connection to the OEM.

A. Entities

In addition to the entities introduced above (UD, OWN, TRK and OEM) we use two additional participants, namely a service point SP and an employee SPE of this service point conducting the flashing procedure.

1) OEM - Manufacturer: The OEM manufactures the UD and delivers it to OWN. OWN is issued the corresponding TRKs linked to the UD. All UDs are obviously known to the OEM. The verification keys $\mathtt{VK}_{TRK}$ are stored by the OEM together with their pairing to the UD. Therefore the OEM knows what TRK is linked to what UD. We regard the entity OEM as a trusted central server with a database functionality.

The OEM can store data, sign data with $\mathtt{SK}_{OEM}$ and send data. It possesses all cryptographic abilities for PKC based authentication schemes and can thereby authenticate communication partners.

2) UD - User Device: UD is enabled only when a linked TRK is presented by authenticating the TRK via a PKC authentication scheme. All linked TRKs' public keys $\mathtt{VK}_{TRK}$ are stored in the UD. Additionally the public key of the OEM $\mathtt{VK}_{OEM}$ is stored in the UD and can not be erased or altered in any way and UD has a OEM-issued certificate for it's own public key certifying being a genuine part. UD grants read access to all stored public keys. Write access to the memory location of $\mathtt{VK}_{TRK}$ is only granted in the context of the proposed key flashing scheme.

UD possesses all cryptographic abilities for PKC based authentication schemes and can thereby authenticate communication partners.

3) OWN - Legal User: OWN is the legal user of UD and can prove this by possession of a linked $\mathtt{TRK}_{orig}$.

4) TRK - Transponder: TRK[2] possesses a keypair $\mathtt{VK}_{TRK}$/$\mathtt{SK}_{TRK}$ for PKC functionality. It is generated inside the TRK to ensure that the secret key $\mathtt{SK}_{TRK}$ is known solely to TRK. Read access to $\mathtt{VK}_{TRK}$ is granted to any entity over a communication interface.

[2]TRKs can be manufactured by a supplier that has been certified by the OEM

TRK possesses cryptographic primitives for PKC based authentication schemes on prover's side and can thereby be authenticated by communication partners.

5) SP - Service Point: SP is a service point in the field such as a wholesaler, certified by the OEM. Typically a SP is a computer terminal. Access to the terminal is secured by means of a password as in standard PC practice. A SP can communicate to the OEM as well as to the UD. At the same time it is able to read the $\mathtt{VK}_{TRK}$ of any TRK.

Furthermore the SP constitutes a trusted platform meaning that it always behaves in the expected manner for the flashing procedure and accommodates a trusted module responsible for:

- storage of authorized $\mathtt{VK}_{TRK}$
- secure counter
- key management of authorized $\mathtt{VK}_{TRK}$

SP possesses cryptographic primitives for PKC based authentication schemes on prover's and verifier's side and can thereby be authenticated by communication partners as well as authenticate communication partners.

6) SPE - Employee of Service Point: A SPE is a physical person that is operating the SP and is regarded as a potential attacker of the flashing operation. Access control of a SPE to the SP is enforced via password or similar. SPE is responsible for the system setup for the flashing application consisting of establishing the communication links of UD, SP, TRK, and OEM if needed.

UD, $\mathtt{TRK}_{new}$, and SP are under control of the SPE and the communication links to UD, $\mathtt{TRK}_{orig}$, $\mathtt{TRK}_{new}$, SP, and OEM can be eavesdropped, the trusted module can not be penetrated though.

B. Steps

The following steps are necessary to introduce an new $\mathtt{VK}_{TRK}$ into a UD avoiding online dependency. All of them are included in figure 1.

1) Delegation of trust to SP
2) Authorization of SPE by SP
3) Authorization of $\mathtt{TRK}_{new}$ by OEM
4) Introducing an authorized $\mathtt{TRK}_{new}$ into a UD

Authorization of SPE can be done e.g. via a password (knowledge) or by biometrical identification (physical property). The delegation of trust and authorization of $\mathtt{TRK}_{new}$s is very closely related and described in section V-C. These steps form the first phase of the flashing process and can be done in advance without UD and OWN but need a communication link to OEM. The final introduction of a new $\mathtt{VK}_{TRK}$ is the second phase and is detailed in section V-D. It does no longer depend on interaction with OEM.

C. Trust Delegation and $\mathtt{TRK}_{new}$ Authorization

To be able to perform a key flashing procedure without an active link to OEM a local representative has to be empowered by the OEM to perform the flashing, assuming that UD trusts

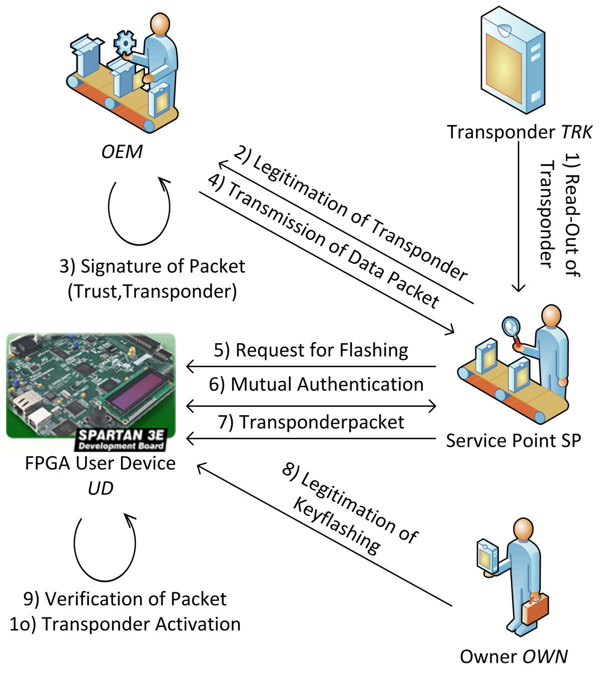

OEM
Transponder TRK
1) Read-Out of Transponder
2) Legitimation of Transponder
4) Transmission of Data Packet
3) Signature of Packet (Trust,Transponder)
5) Request for Flashing
6) Mutual Authentication
7) Transponderpacket
Service Point SP
SPARTAN 3E Development Board
FPGA User Device UD
8) Legitimation of Keyflashing
9) Verification of Packet
1o) Transponder Activation
Owner OWN

Fig. 1. Flashing Scheme

only the OEM to flash legit keys. This is done by presenting a credential to UD accounting that flashing is authorized by OEM. The exchange of this credential is denoted in the following as *trust delegation*.

In our case SP is the local representative. In order to request the flashing credential from the OEM, SPE has to be authenticated first to prevent SP abusive operations. Afterwards SP can connect to OEM and request a trust credential. This is issued only after mutual authentication and only to known partner service points. It is always valid for only a limited time and limited number of flashing operations to minimize negative impact of compromised SPs. This is controlled and enforced by the trusted component inside SP using the secure unforgeable counter keeping track of the number of flashing cycles.

The public key of a $\texttt{TRK}_{new}$ needs to be authorized by OEM. SP can read out $\texttt{VK}_{TRK}$ and send it to OEM. If SP is allowed to flash TRKs into a UD, the OEM sends the authorized $\texttt{VK}_{TRK}$ back to SP which is stored in SP's trusted module. Only a limited number of authorized TRKs can be stored at any given point in time.

As soon as a TRK has been authorized by the OEM, physical access to the TRK needs to be controlled. The authorization process of TRKs is the only step that demands for a data connection between SP and OEM. This does not necessarily need to be an online connection since data could be transported via data carriers such as CDs, memory sticks, or the like.

D. *Flashing of* TRK

The actual flashing of a $\texttt{TRK}_{new}$ to a given UD demands for a valid new transponder $\texttt{TRK}_{new}$, authorization by OEM and OWN, former either directly or delegated to SP using the credential introduced above, latter done by presenting a valid linked $\texttt{TRK}_{orig}$ assumed to be solely accessible by OWN. If an online connection to OEM is available the protocol can be performed by UD and OEM directly, SP only relaying communication.

In either case UD and SP authenticate each other mutually using their respective OEM-issued certificates. UD additionally checks authorization by OWN, testing whether a valid linked token is present or not. If all these tests passed, SP presents the authorized and OEM-signed new $\texttt{TRK}_{new}$ to UD which checks the OEM signature and credential. In the case of successful verification UD accepts the new token $\texttt{TRK}_{new}$ and adds $\texttt{VK}_{TRK}$ to it's internal list of linked tokens.

E. *Entity Requirements*

Regarding the proposed flashing protocols certain requirements for the entities' functionalities have to be satisfied. An overview is given in table I

	OEM	SP	UD	TRK
Initiate Communication	●	●		
Acknowledge Communication	●		●	●
Generation of Keypairs	●			●
Signatures Generation	●	●	●	●
Signature Verification	●	●	●	
Random Number Generation	●	●	●	
Datamanagement for suppliers	●			
Datamanagement for User Devices	●			
Datamanagement for Service Points	●			
Datamanagement for TRKs		●	●	
Secure Storage for delegated Trust		●		
Knowledge of OEM's public key		●	●	

TABLE I
ENTITY REQUIREMENTS

Data management is one of the key requirements in the protocol in the sense that public key data needs to be stored. Secure storage for delegated trust has some additional requirements such as intrusion detection to protect data from being altered in any way. At the same time it is mandatory that this data is always changed correctly as demanded by the protocol. Also the OEM's public key needs to be firmly embedded into the entity and must not be altered in any case, otherwise the OEM can not be identified correctly from the protocol's point of view.

VI. IMPLEMENTATION

The protocol has been implemented as a proof of concept in a prototypical setup based on a network of a standard PC representing OEM and SP. Furthermore Digilent *Spartan3E Starter Boards* with a Xilinx XC3S500 FPGA represent TRKs and UDs.

In figure 2 all implemented instances are depicted. TRK, SP, and UD have to be connected when flashing the key. The OEM connection needs to be established anytime prior to the flashing according to the proposed protocol and is connected via TCP/IP with the SP.

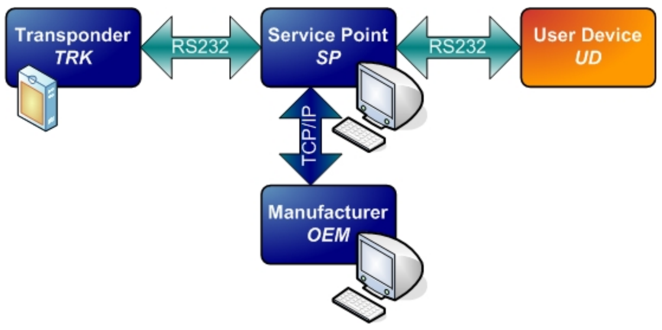


Fig. 2. Component Interaction

Key Length	1024 Bit
Exponent	$2^{16} + 1$ (65537)
Padding Scheme	PKCS#1 v1.5
Signature Scheme	PKCS#1 v1.5
Hashing Scheme used for signing	SHA1

TABLE II
PARAMETERS FOR RSA-SYSTEM

Module	lines of code	percentage
Main application	1234	41.77
GUI	264	8.94
Cryptography	385	13.03
Interaction	383	12.97
Communication	545	18.45
Data Management	143	4.84
Total	2954	100

TABLE III
PROPERTIES OF OEM COMPONENT

Slices	1.791 of 4.656 (38%)
Slices: FlipFlops uses	1.590 of 9.312 (17%)
Slices: LUTs used	1.941 of 9.312 (20%)
BlockRAMs used	16 of 20 (80%)
Equivalent Logic Cells	1.135.468
Minimal clock period	18,777 ns
Maximum clock frequency	53,257 MHz

TABLE IV
FPGA RESOURCES

All other communication is done over RS232 interfaces that are available both on PC and the FPGA boards as well. These can be substituted for other communication structures if needed, i.e. wireless transmitters.

A. Choice of cryptographic Algorithms

The proposed keyflashing concept demands for asymmetric encryption and a cryptographic hashfunction. RSA [15] is chosen for encryption and signing, SHA1 for hash functionality. Both schemes are today's standard and have not been broken yet, but can be substituted in our implementation for more secure schemes if needed. RSA as well as SHA1 implementations are freely available as software and hardware IP for numerous platforms. In table II the RSA parameters chosen are given.

All signatures in our context are SHA1-hash values of data that has been encrypted according to the signing scheme PKCS#1 v1.5 [14]. Such a signature has a length of 128 Byte when using a keylength of 1024 bit and hashvalues of 160 bit bitlength.

B. OEM/Service Point - Software Platform

Both components OEM and SP have been implemented on a standard PC. All functionalities have been implemented in software under the .NET frameworks version 2.0 using C#. The .NET framework provides the Berkeley Socket-interface for communication over the PC's serial interface. At the same time in includes the `Cryptography`-namespace providing all needed cryptographic primitives including hashing functions and a random number generator that are based on the FIPS-140-1 certified Windows CryptoAPI. The software is modularized to enable for easy exchange of functional blocks and seamless substitution of algorithms. Software modules communicate only over defined interfaces to enable for full functional encapsulation. For ease of usage a graphical user interface (GUI) is included as well in both entities.

C. Transponder/UserDevice - FPGA platform

The targeted user device is an FPGA. To enable for easy reuse of functionalities the exemplary TRK has been implemented on FPGA as well, but can also be integrated into a smart card or RFID chip as long as the appropriate cryptographic primitives are provided.

A MicroBlaze softcore processor is incorporated that provides all functionality including cryptographic functions. Hardware peripherals such as a LCD controller have been integrated for debugging purposes. To enable for handling of big numbers, as are used in the cryptographic functions of the protocol, the libraries `libtommath` [5] and `libtomcrypt` [4] are used. Only necessary components have been extracted from those libraries and are integrated into TRK and UD.

D. Resource Usage

The resource usage of the components OEM and SP are very similar, since almost identical functional software blocks are used in both. Table III gives an exemplary overview of the lines of codes of the OEM implementation. The memory footprint of the compiled OEM implementation is 129 KB (139 KB for the SP implementation). At start up 15400 KB of main memory is used. The execution times for RSA- and SHA1-operations were measured on a PC (2 GHz, 1024 MB RAM) and are all in the range of milliseconds.

Resource usage of the FPGA based components UD and TRK are given in table IV. By implementing all functionality on a MicroBlaze softcore, the hardware usage is quite moderate. On the other hand the software footprint is 295 KB for the UD implementation, due to the non-optimized memory usage of the crypto library used.

Shown in table V are the execution times of the divers protocol instances. The duration of parts of the protocol that are

Protocol instance	Duration (min:sek.ms)
ReadOut of Transponder	01:32.000
Mutual Authentication of UD und TRK	03:14.000
Direct Keyflashing	
Keyflashing to Transponder by OEM	23:50.000
Keyflashing by ServicePoint	
Delegation of trust OEM to SP	00:00.350
Transponderdelegation	00:00.250
Keyflashing to Transponder by SP	12:43.000

TABLE V
PROTOCOL EXECUTION TIMES

based soley on OEM and SP is in the area of few milliseconds. As soon as mobile devices (UD, TRK) process parts of the protocol, speed is declining since all crypto operations are currently carried out on an embedded microcontroller. Main factor here is the RSA decryption operation. With appropriate hardware support, choice of parameters and cryptosystem, substantial speedups can be achieved as shown in [9].

VII. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a scheme for flashing public keys into mobile FPGA devices under the constraint that no online connection to the system manufacturer (OEM) is mandatory. It is applicable for a variety of embedded systems that need to implement and enforce access or usage restrictions in the field. The scheme was implemented as a proof-of-concept using a combination of PC-based and FGPA-based protocol participants.

A. Security Analysis

Looking at the security of the proposed concept some points can be identified where security relies on policies and implementing rules while other issues are covered by design.

Using PKC primitives and trusted computing approaches the protocol ensures confidentiality of secret keys and mutual authentication of SP and OEM, OWN and UD, SP and UD, SP and SPE. But due to the necessity of online-independence there are some assumptions that have to be made to guarantee security. This is mainly the trustworthiness of the SP in combination with the physical protection of authorized $\mathtt{TRK}_{orig}$.

If these assumptions are broken e.g. by theft of authorized TRK, the corresponding SP and the SPE password, unauthorized flashing may be possible. As countermeasures the usage of the protocol can be adapted to dilute effects of such events. So the number of allowed authorized TRK should be as low as possible and the SP should be implemented using trusted components and based on a trusted platform secrets should be especially protected against misuse by a physical attacker.

B. Future Work

Flashing speed is of utmost importance in real world implementation. To make allowance for a real world integration of the proposed flashing schemes, optimizations regarding usage and speed of the computational units involved are needed. In the current prototype the MicroBlaze processor has been used for simplicity. Speed up can be achieved with a hardware/software codesign as done in [10]. For maximal speed a full FPGA hardware implementation is desirable, as has been done in [13] for cryptographic functionalities of a HECC system.

The user authentication via PKC can be a solution for dedicated function enabling. Different functionalities can be configured onto an FPGA using partial dynamic reconfiguration. By either allowing or prohibiting, the configuration of a certain bitstream depending on the user employing the system, usage policies could be enforded thus opening up new business models for suppliers of FPGA based systems.

One crucial point is the protection of the TRK's public key stored in the UD against physical attackers. The possibility of countermeasuring attacks that might alter stored keys on a physical level needs to be investigated in the future as well.

REFERENCES

[1] Andr Adelsbach, Ulrich Huber, and Ahmad-Reza Sadeghi. Secure software delivery and installation in embedded systems. In Robert H. Deng, editor, *ISPEC 2005*, volume 3439 of *LNCS*, pages 255–267. Springer, 2005.

[2] Hilti Corporation. Electronic theft protection. Available electronically at www.hilti.com, 2007.

[3] Gerrit de Boer, Peter Engel, and Werner Praefcke. Generic remote software update for vehicle ecus using a telematics device as a gateway. *Advanced Microsystems for Automotive Applications*, pages 371–380, 2005.

[4] Tom St. Denis. Libtomcrypt. http://libtomcrypt.com/.

[5] Tom St. Denis. Libtommath. http://math.libtomcrypt.com/.

[6] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[7] Herstellerinitiative Software (HIS). *HIS-Presentation 2004-05*, 2005. available electronically at www.automotive-his.de.

[8] Herstellerinitiative Software (HIS). *HIS Security Module Specification v1.1*, 2006. available electronically at www.automotive-his.de.

[9] Alexander Klimm, Oliver Sander, and Jurgen Becker. A microblaze specific co-processor for real-time hyperelliptic curve cryptography on xilinx fpgas. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[10] Alexander Klimm, Oliver Sander, Jürgen Becker, and Sylvain Subileau. A hardware/software codesign of a co-processor for real-time hyperelliptic curve cryptography on a spartan3 fpga. In Uwe Brinkschulte, Theo Ungerer, Christian Hochberger, and Rainer G. Spallek, editors, *ARCS*, volume 4934 of *Lecture Notes in Computer Science*, pages 188–201. Springer, 2008.

[11] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[12] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pages 31–53, London, UK, 1993. Springer-Verlag.

[13] J. Pelzl, T. Wollinger, and C. Paar. *Embedded Cryptographic Hardware: Design and Security*, chapter Special Hyperelliptic Curve Cryptosystems of Genus Two: Efficient Arithmetic and Fast Implementation. Nova Science Publishers, NY, USA, 2004. editor Nadia Nedjah.

[14] RSA Laboratories Inc: RSA Cryptograpy Standard PKCS No.1. Elektronisch verfügbar unter http://www.rsasecurity.com/rsalabs.

[15] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[16] Claus P. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 239–252, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[17] Henning Wallentowitz and Konrad Reif, editors. *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*. Vieweg, Wiesbaden, 2006.

Teaching Reconfigurable Processor: the Biniou Approach

Loic Lagadec [*] [†], Damien Picard [*] [†] and Pierre-Yves Lucas

[*] Université Européenne de Bretagne, France.
[†] Université de Brest ; CNRS, UMR 3192 Lab-STICC, ISSTB,
20 avenue Le Gorgeu
29285 Brest, France.
loic.lagadec@univ-brest.fr

***Abstract*—This paper presents the Biniou approach, that we have been developping in the Lab-STICC, together with a "software for embedded systems" master course and its related project. This project addresses building up a simple RISC processor, that supports an extensible instruction set thanks to its reconfigurable functional unit. The Biniou approach covers tasks ranging from describing the RFU, synthesizing it as VHDL code, and implementing applications over it.**

I. Introduction

a) The Master curriculum LSE: The master curriculum LSE (Software for Embedded Systems) opened two years ago at the university of Western Brittany. It addresses emerging trends in embedded systems, mainly from a software point of view despite warmly welcoming EE (Electronic Engineering) students. It highly focuses on RC (Reconfigurable Computing) based embedded systems, with a set of courses for teaching hardware/configware/software co-design.

b) The underlying research support: The research group behind this initiative is the Architectures & Systems team from the Lab-STICC (UMR 3192). This group owns a legacy expertise in designing parallel reconfigurable processor (the Armen [1] project was initiated in 1991) but focuses on CAD environment developments (Madeo framework [2]).

c) The Madeo framework: The madeo framework is an open and extensible modeling environment that allows representing reconfigurable architectures. It acts as a one-stop shopping point providing basic functionalities to the programmer (place&route, floorplanning, simulation, etc.). Based on Madeo, several commercial and prospective architectures have been designed (Virtex, reconfigurable datapath, etc.), and some algorithms have been tailored to address nano-computing architectures (WISP, etc.).

d) Biniou: The Madeo project ended in 2006 while being integrated as facilities in a new framework. This new framework, named Biniou, embeds additional capabilities such as, from the hardware side, VHDL export of the modeled architecture and wider interchange format and extended synthesis support from the software side. Biniou targets Reconfigurable SOCs design and offers middleware facilities to favor a modular design of reconfigurable IPs within the SOC.

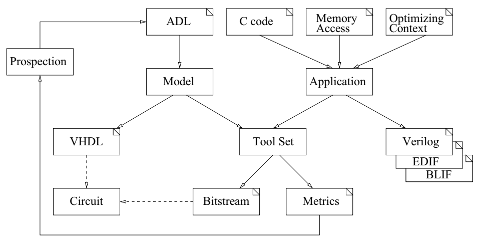


Fig. 1. Overview of the Biniou flow.

Hence Biniou enhances durability and modularity, and shorten development time of programming environments.

Figure 1 provides an overview of Biniou. In the application side (right) an application is specified as C-code, memory access patterns and some optimizing contexts we use to tailor the application. This side outputs some post-synthesis files conforming to mainstream formats (Verilog, EDIF, BLIF). Results can be further processed by the Biniou P&R layer to produce a bitstream. Of course the bitstream matches the specification of the underlying reconfigurable target, be the target modeled using a specific ADL. A model is issued on which the P&R layer can operate as previously mentioned, and a behavioral VHDL description of the target is generated for simulation purposes and FPGA implementation.

Also some debugging facilities can be added either in the architecture itself or as parts of the application [3].

e) From research to teaching: Biniou has been exercised as a teaching platform for Master 2 students from the LSE class. This happened in the one-year DOP (standing for Description and Physical Tools) course, during which students had to perform practical sessions and to lead a project covering VHDL hand-writing, reconfigurable architecture modeling and programming, code generation, modules assembly to exhibit a simple processor with a reconfigurable functional units allowing to extend its instruction set.

The rest of this paper is organized as follows: section II describes the DOP course, section III introduces the project while section IV points out the results that came out from the project. Finally some concluding remarks and perspectives are presented in section V.

II. An overview of the DOP master course

A. Students profile

1) Former curriculum: The master gathers students from both CS (Computer Science) and EE former curricula. The current master size is 12, coming from half a dozen countries. Half of the students are former local students hence own a local background in term of CS but suffer from lacks in electronic system design.

As this situation could carry some risks, we chose to make students pair-achieve the project. In this way, beyond simply averaging the pre-requisites matching so that the pairs are equally offered a chance to succeed, we intended to favor incidental learning as pointed out by Shanck [4].

2) Foreigners issues: Unfortunately, one remaining difficulty the foreigners must face, lies in gaining the visa to enter the country, what can take an unpredictable long delay. We activated a facility that the remote-teaching service of the UBO offers, that is a secured collaborative web site, on which all the slides and supports remain at the student's disposal as soon as he got an access code - that we provide early enough to serve as a remedial class. Also, this offers monitoring features (recent activities such as document downloads or due work uploads are logged on).

B. Course organization

Courses are organized around two main topics covering the hardware (architectures) and software (CAD tools and compiler basics) aspects of reconfigurable computing. Traditionally, these topics are not grouped in a same curriculum and are either taught in CS or EE. DOP courses enable students to build from their previous knowledge a cross-expertise giving a complete vision of the domain.

1) An overview of the reconfigurable computing landscape: This course aims at giving a global overview of the reconfigurable computing (RC) landscape. It focuses on both industrial and academic architectural solutions and is structured in three parts:

- Overview of RC for embedded systems (2 sessions)
- Virtualization techniques for RC (2 sessions)
- Modeling and generation of reconfigurable architectures (1 session)

a) Overview of RC for embedded systems: This part introduces the increasing needs for performance of embedded systems executing digital signal processing applications, such as smartphones, set-top boxes, HD cameras and so on. A major part of students are not familiar with the concept of computation acceleration, a comparison between the Moore and Shannon law illustrates the need for different architectures and computation models than traditional GPP.

A first solution proposed to student for tackling the Shannon law is to implement intensive tasks as ASICs in order to exploit instruction parallelism. However, flexibility, NRE costs and short life cycle issues related to ASICs are pointed out leading to look for a trade-off between flexibility and performance with reconfigurable computing as an answer. Three types of architectures are presented: fine-grained FPGA, coarse-grained architectures and reconfigurable processors.

Before describing the main architectural characteristics, general definitions are given for reconfigurable computing, reconfigurable architectures and the notion of granularity.

FPGAs are introduced by presenting the FPGA marketplace and the main suppliers. We also show ASICs costs trends for justifying the growing interest for reconfigurable technology and an evolution of the FPGA application domain which tends to diversify over time.

In order to give to students the main architectural concepts behind FPGAs, we present a first simple architecture. A mesh of basic processing elements (PE) composed of one 4-LUT with its output possibly latched. Combination of the basic blocks (LUT, switch, buses and topology) is presented as a template to be extended and complexified (in terms of routing structure and processing elements) for building real FPGA. A more complex view is given by an example from the industry, a Xilinx Virtex-5, with an emphasis on locating template basic blocks within Xilinx schematics. As a result, students are able to locate the essential elements for a better understanding of state-of-the-art architectures.

The coupling of a reconfigurable unit with a processor (as know as reconfigurable processor) is presented as another alternative for accelerating intensive DSP tasks. The concept of instruction set metamorphosis [5] is defined and a set of architectures are described. For example, P-RISC [6], Garp [7], XiRISC [8] and Molen [9]. A specific focus is set on the Molen programming model and its architectural organization. The Molen approach is presented as a meeting point between the software domain (sequential programming and compiler) and the hardware domain (specific instruction designed in hardware).

Drawbacks of fine-grained architectures such as low computation density and routing congestion are highlighted to introduce coarse-grained architectures. This type of reconfigurable architecture is firstly presented as a specialization of FPGA (in term of routing resources and processing elements) suited for DSP application domain. Architectures presented are Kress-Array [10], Piperench [11], PACT XPP [12], Morphosys [13]. Programming model issues are discussed with a comparison between software oriented approach (generally using subsets of C) and hardware approach (netlist based descriptions). A case study of the DREAM architecture is presented with an emphasis on the compiler friendly approach of the tools targeting the PicoGA [14], [15].

b) Virtualization techniques for RC: Second part of the course aims at giving details on advanced use of reconfigurable architectures. The motivation is to leverage, thanks to virtualization techniques, some well-known limitations of RA: limited amount of resources, lack of high-level programming model and non-portability of bitstream.

In a first step virtualization is defined as a general concept originally applied to computer (e.g. virtual memory and virtual machine).

In order to leverage resource amount limitation, time multiplexing is defined and its support by reconfigurable architectures is detailed. It starts from temporal partitioning, dynamic reconfiguration and the different configuration plan structures: multi-context and partial reconfiguration. These notions are illustrated by architectures such as DPGA [16], WASMII [17] and their programming flows.

Computation model for reconfigurable computing is addressed by two different approaches. The first approach describes programming model directly supported by hardware such as Piperench or STREAM [18]. These architectures provide facilities: CAD tools and runtime management for virtualizing resources. Application portability is ensured by the availability of the programming model in a device family, similarly to GPP ISA family.

A second approach is to use soft-cores comparable to software virtual machines. Soft-cores are implemented on off-the-shelf reconfigurable devices avoiding a costly ASIC design but sacrificing some performances. Illustrative examples are given by Mitrion [19] and Quku [20].

c) Modeling and generation of reconfigurable architectures: This course aims at giving to student a deep understanding of FPGA internal behavior.

In a first part, every elements of a basic FPGA (used as an example in the first course) are detailed and a VHDL behavioral description is explained. It starts from atomic elements, such as pass gates, multiplexers and shows how to interconnect them for building up input/output blocks and configurable logic blocks. A daisy-chain architecture is detailed as well as a configuration controller.

A second part describes the Biniou generation of the architecture from an ADL description. This part makes students ready for practical sessions and for the project.

A FPGA is described using an ADL increasing the level of abstraction compared to a VHDL description. The configuration plan is described as a set of domains permitting to exploit partial reconfiguration. The approach relies on model transformation, with an automatic VHDL code generation from an high-level description.

2) Software part: The software part of the courses addresses both state-of-the-art tools and algorithms in one hand as well as locally designed tools in another hand. The key idea is that students are naturally attracted to learning classical (or vendors') tools so that they can bring a direct added-value to any employer of the field, hence get in an interesting and well-paid job.

However, tools obviously encapsulate the whole domain-specific expertise, and letting students "open the box" closes the gap between "lambda users" and experts. This takes up the challenge of providing a valuable and inovative curriculum.

Several computational models are introduced with a special highlight on temporal versus spatial computing. In addition, several scheme of mixing up temporal and spatial computing are presented, such as reconfigurable functional units, reconfigurable co-processor, etc.

The link takes place when introducing the Molen paradigm, with hardware blocks running as accelerated functions that provide results back to the processor.

To implement accelerated functions, a resources allocation is required, that remains highly dependent of the hardware platform.

We introduce Data Flow Graph (DFGs) and Control Data Flow Graph (CDFGs) representations that act as entry point to the resources allocation.

Here happens the circuit synthesis, and some algorithms (A* point to point routing, pathfinder global routing, placement, TCG floorplanning, etc.) and backend tools are presented (Madeo, VPR, etc.).

C. A Morpheus inheritance

Biniou is partially issued from our contribution to the Morpheus FP6 project [21], standing for "Multi-purpose dynamically reconfigurable Platform for Intensive Heterogeneous processing" (2006-2009). MORPHEUS addresses innovative solutions for embedded computing based on a dynamically reconfigurable platform and adequate tools. The main challenge of the "platform work package" was to propose a smart reconfigurable computing hardware carefully crafted for use with methods and tools developed in the "methodologies and tools" work package. The approach of the MORPHEUS toolset provides both an effortless management of reconfigurable accelerated functions within a global application C code, and an easy design of the accelerated functions through high level description and synthesis. There was the place in which our contribution - being at the birth of Biniou - mainly appears.

Our second contribution was within the "training work package" that intents to setup courses and to affect the curricula for hardware and software engineers targeting heterogeneous and reconfigurable SoCs. The LSE Master is one practical result coming out from Morpheus.

D. Practical sessions

Practical sessions are organized as three activities. The first activity is to gather documentation and publications related to a particular aspect of the course; the students have to present their short bibliographic study individually in front of the whole class.

The second activity is centered on algorithms used for implementing application over a reconfigurable architecture: point-to-point and global routers, floorplanners, placers. Some data structures such as transitive closure graphs are introduced later on in order to point out the need for refactoring and design patterns use [22]. This bridges the software expertise to the covered domain (CAD tools for RC). Another place where this link appears is when designing a BLIF CABA netlist simulator in a couple of hours by simply combining well known software design patterns: observer (propagation of change), state (current value, and future value for flip-flops), composite (both hierarchical combinations of modules and single netlist appear in the same way).

The third activity is related to tools and formats. Three slots are dedicated to VHDL that most of the students do not know. Manual description of fine-grained reconfigurable architecture is introduced within this amount of time. Some sessions are dedicated to practicing required tools during which students manipulate logic synthesis tools (SIS, ABC), file formats conversion (PLA, BLIF, EDIF), behavioral synthesis and CDFG-to-Verilog translation according to some data access pattern (Biniou).

Biniou lets students create their own FPGA, that is further reused in the project under a tuned up version, and highlights the configuration scheduling issue.

We also offer a Web based tool [23] to output RTL netlists that students use to exercise several options when generating their netlists.

III. The Project

A. Overview of the project

The project consists in designing a simple RISC processor, that can perform spatial execution through a reconfigurable functional unit. This scheme conforms to the Molen paradigm.

Figure 2 illustrate the schematic view of the whole processor, including the RFU.

The processor supports a restricted instruction set (table I). Instructions SET and EXRU respectively configures and activates the RFU.

Opcode	Code
NOT	0000
AND	0001
OR	0010
XOR	0011
LSL	0100
LSR	0101
ADD	0110
SUB	0111
CMP	1000
JMP	1001
JE	1010
JNE	1011
SET	1100
EXRU	1101
LOAD	1110
STORE	1111

TABLE I
INSTRUCTION SET OPCODES.

In order to keep the project reasonably simple, we restrict the use of the RFU to implementing DFGs on one hand, and we provide students with the Biniou framework on the other hand. Restricting the use of the reconfigurable part as a functional units (as depicted by figure 3) also mitigates the complexity of the whole design. However, this covers the need for being reachable by average students while preserving the ability to arouse top students' curiosity. by offering a set of interesting perspectives for further developments.

We believe that this project is a perfect starting point to let students to build and stress new ideas in many disciplines related to RC-computing such as spatial versus temporal execution, architectures, programming environments and algorithms.

1) Context: This project takes place during the fall semester, from mid October to early January. A noticeable point is that almost no free slots within the timetable are dedicated to this project that overlaps with courses as well as with "concurrent" projects. This intends to stress students and make them aware of handling competing priorities.

To prevent students from postponing managing this project we use the Moodle platform for monitoring activities, collecting deliverables and broadcasting updates/comments/additional information.

2) Expected Deliverables: We define three milestones and three deliverables. The milestones are practical session in front of the teacher, one out of which session is shared with another course as students learn and practice logic synthesis from behavioral code.

Three main milestones:

M1: RISC Processor, running its provided test programs
M2: RFU, with Galois Field based operations implemented as bitstream
M3: Integration, Final review

3) Schedule: The schedule is provided during the project "kick-off". The collaborative platform allows specifying time-windows during which deliverables can be submitted. Reminders can be sent by mail when the dead line is approaching. Once the dead line is over, over due deliverables are applied a penalty per extra half-day.

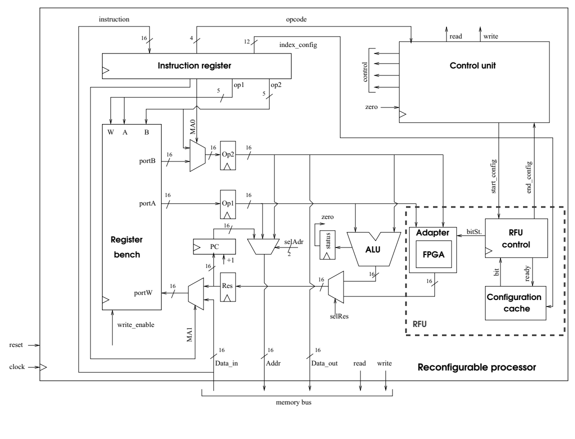


Fig. 2. Schematic of the entire reconfigurable processor.

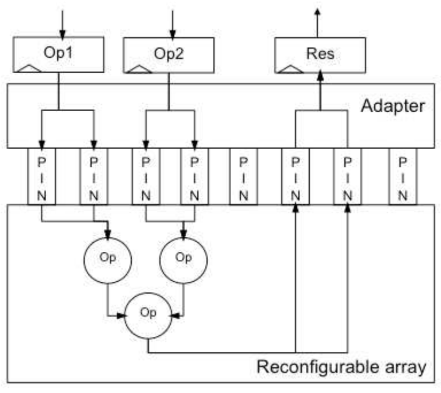


Fig. 3. RFU is interfaced with the processor registers through an adapter.

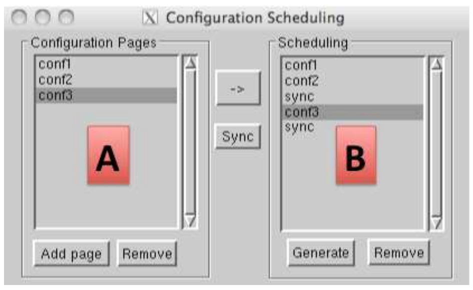


Fig. 4. User interface of the configuration scheduler. Configuration pages (part A) are scheduled (part B) for producing the configuration controller program initialized by a generated testbench.

B. Provided facilities

In order to make this project feasible on time, some building blocks are implemented during practical sessions or ready-to-use elements are given to students as starting points.

The processor is partially implemented during practical sessions dedicated to VHDL. Besides designing basics combinatorial and sequential circuits, advanced exercises lead students to design an ALU controlled by a FSM. The goal is to exercise the execution of a simple dataflow graph over the ALU. At the end of these sessions, they have implemented a first prototype of a controller and a complete ALU.

Concerning the reconfigurable part, Biniou facilities alleviate the students' workload. Biniou generates a reconfigurable matrix specified in Madeo-ADL, and we provide skeletons of element descriptions (e.g. IOB) that students finalize during a practical session. Then students generate a reconfigurable matrix.

A programmable configuration controller is also provided which interfaces the reconfigurable matrix. This controller manages partial reconfiguration and configuration pages scheduling. The manual scheduling of the configuration page comes from a user interface depicted by figure 4. For validation purposes, a global testbench is also generated enabling to test the reconfigurable matrix configuration and execution. It instantiates the matrix, the configuration controller and performs initialization steps such as sending the scheduling program to the controller.

1) Processor soft-core: A preliminary version with missing control structures was provided in order to ensure a minimal compatibility of the designs. Obviously, the matter here was to ease evaluation from a scholar point of view as well as to force students to handle kind of legacy system and refactoring rather than full re-design.

We also provided the instruction set and opcode (but without compiler). In an ideal world, and with a more generous amount of time to spend on the project, as the design is highly modular, building a working design by picking best-fit modules out of several designs would have also been an interesting issue.

2) Decoder: It outputs signals from input instruction according to layout on table II.

a) Testbench program: Students are familiar with agile programming, and we assume in a software-coding context

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				MA		OP1					OP2				

TABLE II
INSTRUCTION LAYOUT.

to reward test-first approach. When designing a processor the same approach applies but at a wider granularity. Hence, we distributed some testbench programs, one of which is provided in table III.

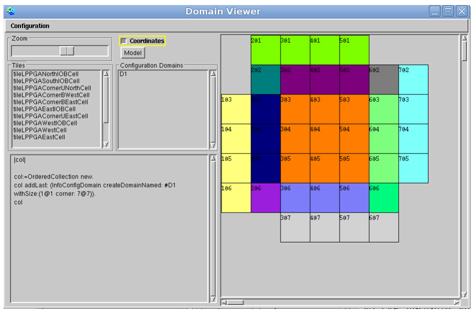
Analyzing at specific timestamps (including after the application stops) the internal states (some signals plus registers contents), leads to design scoring.

```
RAM(0)<=1110010000000000;
−−LOAD R0,0;Initializing
RAM(1)<=0110010000011111;
−−ADD R0,31;R0=31
RAM(2)<=0110010000011111;
−−ADD R0,31;R0=62
RAM(3)<=0110010000011111;
−−ADD R0,31;R0=93
RAM(4)<=0110010000000111;
−−ADD R0,7;R0=100
RAM(5)<=1111010000000000;
−−STORE [R0],0; Memory init
RAM(6)<=1110010000101011;
−−LOAD R1,11; Selecting op
RAM(7)<=1110010001000001;
−−LOAD R2,1;Mask
RAM(8)<=1110010010000000;
−−LOAD R4,0;Bit counter
RAM(9)<=1110010010110000;
−−LOAD R5,16; Loop counter
RAM(10)<=1000010010100000;
−−CMP R5,0;Loop ending?
RAM(11)<=1010010000010100;
−−JE 20;(OP2 as addr)Jump up to end
RAM(12)<=1110000001100001;
−−LOAD R3,R1;Ref value copying
RAM(13)<=0001000001100010;
−−AND R3,R2;Mask forcing
RAM(14)<=1000010001100001;
−−CMP R3,1;set-to-1 bit detection?
RAM(15)<=1011010000010001;
−−JNE 17;(OP2 as addr) Else jump
RAM(16)<=0110010010000001;
−−ADD R4,1; Increment if true
RAM(17)<=0101010000100001;
−−LSR R1,1; Next bit
RAM(18)<=0111010010100001;
−−SUB R5,1;Decrement
RAM(19)<=1001010000001010;
−−JMP 10;(OP2 as addr) Jump to start
RAM(20)<=1111000000000100;
−−STORE [R0],R4; Saving result
RAM(21)<=1001010000010101;
−−JMP 21; Infinie loop
−−endtest2
```

TABLE III
EXAMPLE PROGRAM: A BIT COUNTER.

3) Reconfigurable FU design: Designing a RFU does through sizing the matrix, defining a basic cell and isolating border cells that deserve special attention because their structure is slightly different from the common template. The basic cell is both used as is for the internal cells and tuned to generate the border cells.

Defining the domains appears as shown by figure 5.



```
tileLPPGANorthIOBCell
tileLPPGASouthIOBCell
tileLPPGACornerUNorthCell
tileLPPGACornerBWestCell
tileLPPGACornerBEastCell
tileLPPGAEastIOBCell
tileLPPGACornerUEastCell
tileLPPGAWestIOBCell
tileLPPGAWestCell
tileLPPGAEastCell
```

```
D1
```

```
|col|

col:=OrderedCollection new.
col addLast: (InfoConfigDomain createDomainNamed: #D1
withSize:(1@1 corner: 7@7)).
col
```

Fig. 5. On the right, view of the different cell types composing the matrix (border cells, middle cells, IO cells). On the left, configuration domains are defined as a set of rectangular boxes. They can be reconfigured independently from each other.

The basic cell schematic view is provided by figure 6.

Ultimately, the full matrix appears as an array of N^2 cells as illustrated by the snapshot of the Biniou P&R layer (figure 7).

4) Application synthesis over the RFU: To let students figuring out the benefit of adding the RFU to the processor design, it is desirable that students can assess and compare the impact of several options. One classical approach consists in implementing a DFG to exhibit spatial execution. Another option lies in implementing combinational operations (such as a multiplier) instead of performing a loop of processors instructions (addition, shifts, etc.). In both cases, the RFU extends the instruction set. On the opposite, the underlying arithmetic can vary keeping the instruction set stable, but this goes through either a library-based design or dedicated synthesizers. Libraries are typically targeted to a reduced set of pre-defined macroblocks, and they are not easily customizable to new kinds of functions or use-context.

We chose to focus on the second item as this seems to carry extra added-value compared to classical flows, while neglecting the need for a coding extra effort. Figure 8 illustrates the Biniou behavioral application synthesizer. The

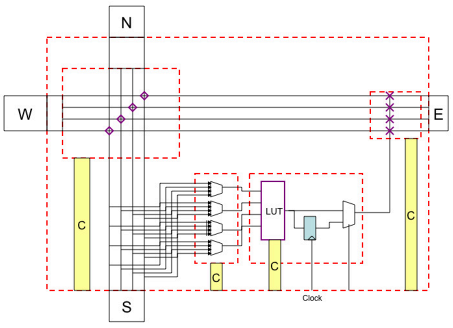


Fig. 6. Structure of a basic cell within the RFU matrix.

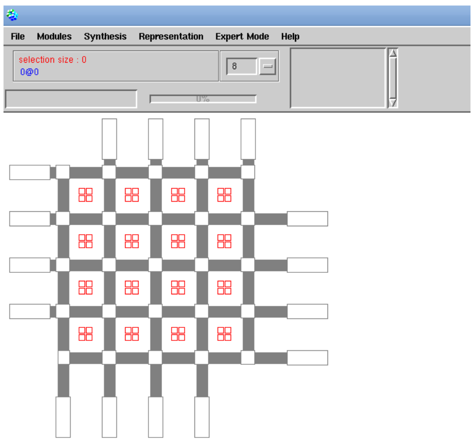
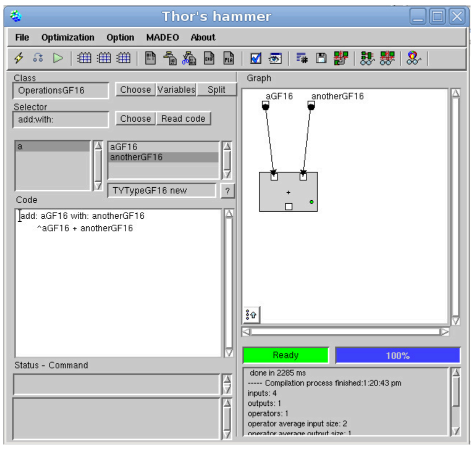


Fig. 7. Whole view of the RFU's fine grained reconfigurable matrix.

optimizing context here is made up of typing as Galois Field GF^{16} values the two parameters. A so-called high-level truth table is computed per graph node for which values are encoded and binarized before the logic minimization starts. The result appears as a context-dependent BLIF file. This BLIF file is further processed by the Biniou P&R layer, that relies on a pathfinder like algorithm. As application is simple enough to keep the design flatten, no need exists for using a floorplanner. However, for modular designs, a TCG based floorplanner is integrated within Biniou.

Some constraints are considered, such as making some location immutable to conform to the pinout of the adapter (figure 3). Once the P&R process ends, a bitstream is generated. Each element of the matrix both knows its state (used, free, which one out of N, etc.) and its layout structure. The full layout is gained by composing recursively (bottom up) these sub-bitstreams. An interesting point is that the bitstream structure can vary independently from the architecture by applying several generation schemes. It's highly valuable in a partial



Class: OperationsGF16

Selector: add:with:

```
a
```

```
aGF16
anotherGF16
```

TYTypeGF16 new ?

```
add: aGF16 with: anotherGF16
    ^aGF16 + anotherGF16
```

```
done in 2285 ms
----- Compilation process finished:1:20:43 pm
inputs: 4
outputs: 1
operators: 1
operator average input size: 2
operator average output size: 1
```

Fig. 8. Specification of a GF^{16} adder.

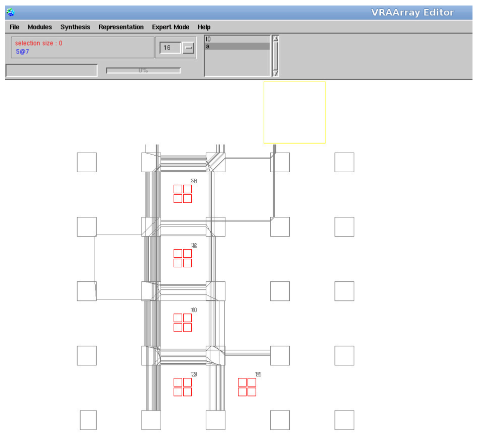


Fig. 9. An application placed and routed over the RFU.

reconfiguration scope when the designer faces several axes during the architectural prospection phase. In the frame of the project an example of bitstream structure is provided by the figure 10.

5) Reconfigurable Functional Unit Integration: The reconfigurable functional unit (RFU) is composed of three main components: the reconfigurable matrix (RM) generated by Biniou, a configuration cache and the RFU controller both hand-written (see bottom right in figure 2).

Configuration is triggered by the processor controller which reacts to a SET instruction by sending a signal to the RFU controller. The RFU controller drives the configuration cache controller, which provides back on demand a bitstream. The processor controller gets an acknowledgement after the configurations completes.

One critical issue about the processor-RFU coupling lies in data transferts to/from the RFU. Students have to design a simple adapter which connects a set of RFU's iopads to the processor registers holding input and output data (Op1, Op2 and Res in figure 2) with regards to the ones assigned to the I/O of a placed and routed application (see figure 9). Figure 3 gives a detailed view of the adapter.

IV. Results coming out of the project

A. Environment results

1) Simulation: The simulation environment is ModelSim [24] as illustrated by figure 11.

The loader module - that loads up the program - was not provided but students could easily get one by simply reusing and adapting the generated testbench. Only 1 group out of five got it right.

This allowed to set a properly initialized state prior to execution's start. Of course, this was a critical issue, and students would have done well to fix it in an early stage as tracing values remained the one validation scheme. This was all the more important as the full simulation took a long time to complete and rerun had a real cost for students.

The simulation of the processor itself is time-affordable but the full simulation takes around 4 hours, including bitstream loading, and whole testbench program execution.

2) Optimizations: Students came to us with several policies to speed up the simulation. A first proposal is to let simulation happen at several abstraction levels, with a high rate of early error detection. Second, some modules have been substituted by a simpler version. As an example, by providing a RFU that only supports 8 bit ADD/SUB operations, the bitstream is downscaled to 1 bit with no compromise on the architecture decomposition itself. This approach is very interesting as it confines changes to the inside of the RFU while still preserving the API. In addition, it joins back the concern of grain increase in a general scope (i.e. balancing the computation/flexibility and reducing the bitstream). Also this approach must be linked to the notion of ”mock object” [25] software engineers are familiar to, when accelerating code testing.

Third, as the application is outputted as RTL code, the code can be used as a hard FU instead of using reconfigurable one. In this way, the students validated the GF based synthesis.

Grabbing these last two points, the global design can be validated very fast, be the scalability issue. This issue has been ignored during the project, but is currently addressed as the global design is been given a physical implementation.

3) Reports: Students had to provide three reports, one per milestone. The reports conformed to a common template, and ranged from 10 to 25 pages each. The last report embedded the previous ones so that the final document was made available straight after the project and students were given another chance to correct their mistakes.

Some recommendations were mandatory such as embedding all images as source format within the package, so that we could reuse some of them. As an illustration, more or less half of the figures in this paper come from students' reports.

The students had no constraints over the language but some of them chose to give back English-written reports. We selected some reports to be published on line as examples or next year students.

4) Oral defense: The last deliverable was made up of a report, working VHDL code and an oral defense. Students had to expose within 10 minutes, in front of the group, course teachers, and a colleague responsible for the ”communication and job market” course.

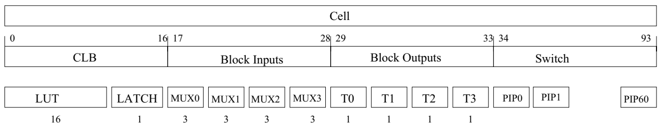


Fig. 10. Example of a bitstream hierarchical organization.

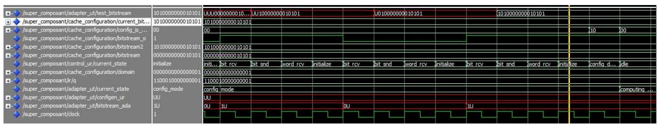

```
/super_composant/adapter_ut/test_bitstream                 1010000000010101
/super_composant/cache_configuration/current_bit...        1010000000010101
/super_composant/cache_configuration/config_is_...         00
/super_composant/cache_configuration/bitstream_o           1
/super_composant/cache_configuration/bitstream2            1010000000010101
/super_composant/cache_configuration/bitstream             0000000000010101
/super_composant/control_ur/current_state                  initialize
/super_composant/cache_configuration/domain                0000000000000001
/super_composant/ir/q                                      1100010000000001
/super_composant/adapter_ut/current_state                  config_mode
/super_composant/adapter_ut/configen_ur                    UU
/super_composant/adapter_ut/bitstream_ada                  1U
/super_composant/clock                                     1
```

Fig. 11. Modelsim simulation.

Some students chose to center their defense on the project and the course versus project adequation, some others around the "product", that was their version of the processor. One confined its presentation inside an introduction of reconfigurable computing and was considered offside.

B. Physical realization

The physical implementation was out of the scope of the project. Several reasons motivated this choice, the first of which was timing issues but also FPGA boards availability.

We offered one student to finalize this during a 5 weeks individual project. This is still under development by now. The development platform we use for this demonstrator is a Virtex-5 FXT ML510 Embedded Development Platform.

V. Conclusion

A. Forces

One interesting point regarding this project lies in the change in the students feeling. When we presented at the first time the project, thought they would never complete the goals. After the first milestone, one group gave up to avoid paying the over due penalty and bounded their work to the first deliverable. They finally reached 7 points out of 20. The other groups faced the challenge and discovered that the key issue lies in getting proper tools to free oneself from manually developing both architectures and application mapping. The final results were very likely acceptable and we collected several working packages.

With this experience in mind, students are now ready for entering a very competitive job market, with a deep understanding of both hardware design over reconfigurable architecture, micro-processors and reconfigurable cross integration and tools&algorithms development.

B. Future evolutions

Obviously, the testbench examples we provided are not sufficient to practice real metrics based measurements. Exploring the benefits of this approach (e.g. measuring speed-up) requires an easy path from a structured programming language such as C to the processor execution. Hence, the application's change would carry no need for hand-written adjustments. From our point of view, such an add-on would be a fruitful upgrade to the course, and would spawn new opportunities for cross H/S expertise; keeping in mind that the DOP course intends to get out with highly trained students sharing skills in both area.

Developing a small compiler was out of the scope of this project due to some timing constraints, but remains one hot spot to be further addressed. This could benefit from some Biniou facilities such as the C-entry both the logic and CDFG synthesizers support.

An open option is then to benefit from another course and invited keynoters to fulfill the prerequisites so that adapting/developing simple C parser becomes feasible in the scope of our project, at the cost of around an extra week.

References

[1] J. M. Filloque, E. Gautrin, and B. Pottier, "Efficient global computations on a processor network with programmable logic," in *PARLE (1)*, pp. 69–82, 1991.

[2] L. Lagadec, *Abstraction and Modélisation et outils de* CAO *pour les architectures reconfigurables*. PhD thesis, Université de Rennes 1, 2000.

[3] L. Lagadec and D. Picard, "Software-like debugging methodology for reconfigurable platforms," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–4, 2009.

[4] R. Schank tech. rep., Institute for the Learning Sciences (ILS) at Northwestern University.

[5] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, 1993.

[6] R. Razdan, K. S. Brace, and M. D. Smith, "Prisc software acceleration techniques," in *ICCS '94: Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, (Washington, DC, USA), pp. 145–149, IEEE Computer Society, 1994.

[7] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[8] F. Campi, R. Canegallo, and R. Guerrieri, "Ip-reusable 32-bit vliw risc core," in *Proceedings of the 27th European Solid-State Circuits Conference*, vol. 18, pp. 445–448, 2001.

[9] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.

[10] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Using the kress-array for reconfigurable computing," in *Proceedings of SPIE*, pp. 150–161, 1998.

[11] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, pp. 70–77, Apr 2000.

[12] J. Becker and M. Vorbach, "Coarse-grain reconfigurable xpp devices for adaptive high-end mobile video-processing," *SOC Conference, 2004. Proceedings. IEEE International*, pp. 165–166, Sept. 2004.

[13] G. Lu, M. hau Lee, H. Singh, N. Bagherzadeh, F. J. Kurdahi, and E. M. Filho, "Morphosys: A reconfigurable processor targeted to high performance image application," in *Proceedings of the International Symposium on Parallel and Distributed Processing*, pp. 661–669, 1999.

[14] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, P. Rolandi, C. Mucci, A. Lodi, A. Vitkovski, and L. Vanzolini, "A dynamically adaptive dsp for heterogeneous reconfigurable platforms," in *DATE'07*, 2007.

[15] C. Mucci, C. Chiesa, A. Lodi, M. Toma, and F. Campi, "A c-based algorithm development flow for a reconfigurable processor architecture," in *IEEE International Symposium on System on Chip*, 2003.

[16] A. DeHon, "Dpga utilization and application," 1996.

[17] X.-P. Ling and H. Amano, "Wasmii: a data driven computer on a virtual hardware," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machine*, pp. 33–42, 1993.

[18] E. Caspi, M. Chu, Y. Huang, J. Yeh, Y. Markovskiy, A. Dehon, and J. Wawrzynek, "Stream computations organized for reconfigurable execution (score): Introduction and tutorial," in *in Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 605–614, Springer-Verlag, 2000.

[19] Mitrionics, "http://www.mitrionics.com/."

[20] S. Shukla, N. W. Bergmann, and J. Becker, "Quku: A two-level reconfigurable architecture," in *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, (Washington, DC, USA), p. 109, IEEE Computer Society, 2006.

[21] "Multi-purpose dynamically reconfigurable platform for intensive heterogeneous processing." http://www.morpheus-ist.org/pages/part.htm.

[22] S. R. Alpert, K. Brown, and B. Woolf, *The Design Patterns Smalltalk Companion*. Boston, MA, USA: Addison-Wesley, 1998.

[23] "Madeo-web, the madeo+ web version." http://stiff.univ-brest.fr/MADEO-WEB/.

[24] "Modelsim." http://www.model.com/.

[25] D. Picard and L. Lagadec, "Multilevel simulation of heterogeneous reconfigurable platforms," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

Behavioral modeling and C-VHDL co-simulation of *Network on Chip* on FPGA for Education

C. Killian, C. Tanougast, M. Monteiro, C. Diou, A. Dandache
LICM, University Paul Verlaine of Metz
Metz, France
{Cedric.Killian, Camel.Tanougast}@univ-metz.fr

S. Jovanovic
IADI, University Henri Poincaré
Nancy, France
s.jovanovic@chu-nancy.fr

Abstract—**In this paper, we present a behavioral modeling and simulation of *Network on Chip* (*NoC*), some parts being implemented on a FPGA education Board. This modeling can be effectively used by students for educational purposes. Indeed, experience proves that behavioral modeling and simulation of the *NoC* concepts help the students to understand *MPSoC,* design approach, and make more practical and reliable designs related to *NoC*. We have chosen C-VHDL co-simulation for educational purposes; all design steps being given in the study. The experience covers *NoC* concepts for *MPSoC* design from postgraduate education to Ph.D level education. For each targeted audience, different properties of the system can be emphasized. Our approach of embedded systems education is based on classic referenced research findings.**

Keywords: NoC, C-VHDL Co-simulation, SoC design, Education purpose.

I. Introduction

Given the evolution and the increasing complexity of *System on Chip* (*SoC*) toward *Multiprocessors Systems on Chip* (*MPSoC*), communication interconnection of modules or *Intellectual Property* (*IP*) constituting these systems, had undergone topological and structural evolution. Actually, the trend is moving towards the full integration of a *Network on Chip* to implement the transmission of data packets among the interconnected nodes which are computing modules or *IP*s (processors, memory controllers, and so on.). Fig. 1 illustrates this trend of on chip interconnection.

In this study, we present a behavioral modeling and simulation of the *Network on Chip* (*NoC*) communication used to overcome some difficulties encountered in understanding the concepts of *Multi-Processor System on Chip* (*MPSoC*) communication design. Indeed, we observed that students learning prototyping based on FPGA education boards had some difficulties and were wasting time on the embedded communication design linking computing nodes of the *MPSoC* in the same chip. To overcome this problem, a *NoC* behavioral modeling and simulation was designed, some parts being synthesized on a FPGA board used at our laboratory. This functional modeling and simulation was prepared as an electronic design example of the configurable *on chip* communication part in the embedded system course plan. We have preferred a *NoC* modeling and behavioral simulation was designed with *C-VHDL* co-simulation in *Modelsim* tool environment for educational purposes. The setup demonstrated in this paper is suited for introductional modeling and verification of the designed *NoC* for *MPSoC* design based on FPGA technology. More precisely, this paper introduces an initiation educational project of modeling, and functional simulation of a mesh *NoC* [1], led by postgraduate students - Master 2 *RSEE* (*Radiocommunications and Embedded Electronic Systems*) specialty education of the Paul Verlaine University of Metz.

The paper outline is as follows. Pre-design studies are introduced in the Section II. We start with a short technical overview (from an educational point of view) of the Modeling of the *NoC* concepts, and the *NoC* model of the case study in this Section. We cover key properties of *NoC* and relate them to training of students. In the Section III, all modules of the modeling and functional simulation are introduced separately in details waveform simulation reports from the test example, and advantages of the behavioral simulation are assessed in terms of practicality, time saving and reliability. Finally, Section IV concludes the education experiences for the paper.

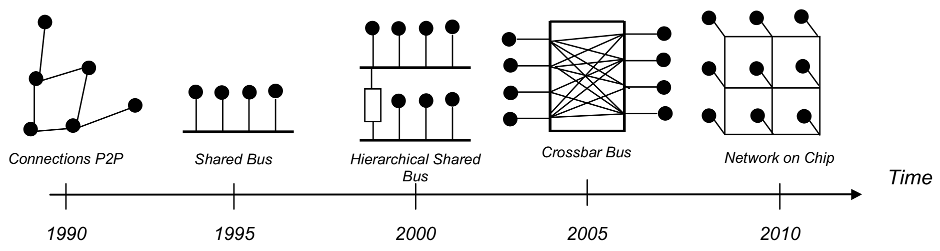


Figure 1. Interconnection evolution in *Systems on Chip*.

II. MODELING AND SIMULATION OF NoC : CASE STUDY

The network transmission is realized through routers constituting the network, and by using switching techniques and routing rules [1]. In this context we propose an initiation project to the *RSEE* student using *Modelsim* hardware simulator tool [2].

We propose to students the modeling of a *NoC* by using switching data packets constituted of messages, and based on mesh topology of configurable size (3x3, 6x6 and 10x10), Wormhole switching rules and standard *XY* routing rules [1]. Fig.2 illustrates the *NoC* to be designed by the students. It is a mesh-network where each one is associated to a *Processing Element* (*PE*). Each *PE-router* pair possesses a specific address and can emit messages through the network.

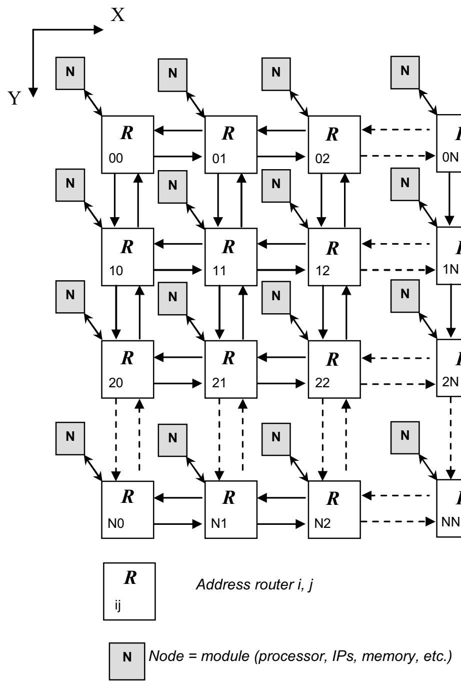


Figure 2. Illustration of the proposed n*x*n Mesh *NoC to model*.

Each packet is constituted of *flits* (*flow control units*) corresponding to data word with fixed size. We define the *Phits* (*physical units*) notion corresponding to the information unit able to be emitted in one cycle trough a physic channel in the network.

1) Routeur behavioral VHDL modelling

The students model a VHDL behavioral description of a *NoC* router characterized by data packets composed of 5 *flits*. The *phit* size corresponds to the channel size. The router input buffer is equal to one *flit* size. The router specifications are described as follows:

- Data packets composed of 5 flits of 9 bits size (5 x 9 bits)

- Data channel of 9 bits (*phit* size)

- One buffer of 2 *flits* deep (2 x 9 bits) in each input of routers

- An internal switching logic based *Crossbar*

- 4 data transmit directions (East, West, South and North)

- *Wormhole* switching rules: the connections between the inputs and direction outputs of the routers are maintained until all the packets *flits* have been sent.

Fig.3 gives the description of the specification of one *NoC* router. The global *NoC* is then made by a structural VHDL description realized by router module instantiations in order to design a *NoC* with a configurable size of 3x3, 6x6 and 10x10 (see Fig. 2).

2) Routing algorithm

The algorithm proposed to the student is an *XY* static routing algorithm. Thus, the data packets are routing toward the destination *PE* through the network router first along *X* axis, and then along *Y* axis. The *XY* routing algorithm is realized in VHDL functional description from the following algorithm:

- If *X_router* < *X_destination*, direction paquets = *Direction_East*

- If *X_router* > X_*destination*, direction paquets = *Direction_West*

- If *X_router* = *X_destination* and *Y_router* > Y_ *destination*, direction paquets = *Direction_South*

- If *X_router* = *X_destination* and *Y_router* < *Y_ destination*, direction paquets = *Direction_North*

- If *X_router* = *X_destination* and *Y_router* = *Y_ destination*, direction paquets = *Local_PE*

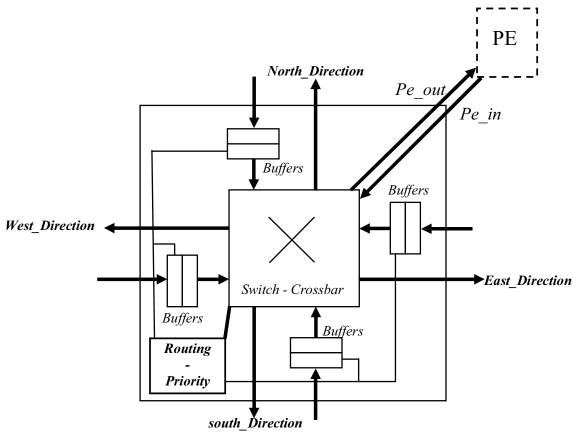


Figure 3. Illustration of the structural architecture of one *NoC* router.

Fig. 4 illustrates the *XY* routing algorithm of data packets between the *PE_00* and *PE_22*. An erroneous description of this algorithm can lead to *deadlock* or *livelock* of data packets [1] during the behavioral simulation steps of the modeled *NoC*. The elaboration of the priority rules of the data packets in routers is defined and designed by the students. These rules need to be associated with the routing algorithm, the number of *flits* constituting the data packets and the wormhole switching rules. The goal is leading students use an iterative and progressive description of the routing and priority rule module gradually in the modeling and simulation phases highlighting situations of temporary *deadlock*, *starvation,* and the impacts on the data packets transmission *latency* notion.

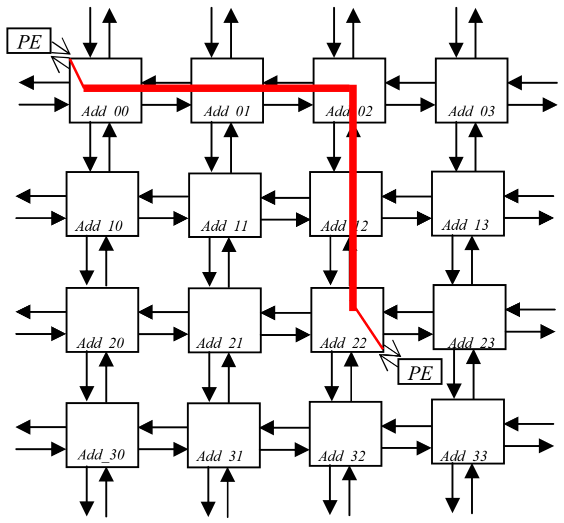


Figure 4. Illustration of the *XY* routing algorithm.

III. *C-VHDL* CO-SIMULATION

From a description in *C* language modeling the data packets transmission and reception by all *PE* associated at each *NoC* router, the students realized a detailed *NoC* VHDL behavioral description (*routers*, *control data flow*, *logic routing*, etc.). From this hardware language description, a validation via *C-VHDL* co-simulation by using *VHDL FLI* (*Foreign Language Interface*) on the *ModelSim* tool is performed [3]. *NoC* design and simulation steps developed by the students, correspond to a *VHDL* description associated to a *C/C++* program modeling all *PEs* providing emission or reception of data packets transmitted in the modeled *NoC*. Fig. 5 depicts the interface and the association of co-simulation in *ModelSim* environment. From files containing the data packets to be sent in the *NoC* by some *PE* transmitters to *PE* receivers, an interface between the *NoC* VHDL description and the *C/C++* function modeling the packets transmitting or receiving for each *PE,* is executed in *ModelSim*.

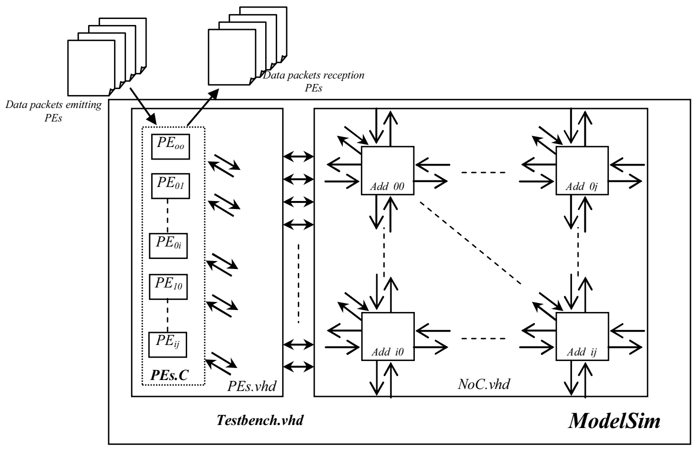
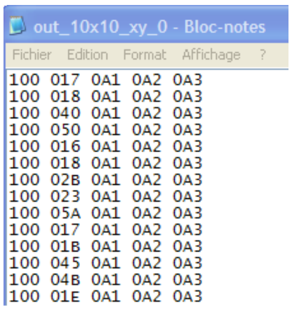


Figure 5. C-VHDL co-simulation in *Modelsim* environment.

In fact, *PEs* are modeled in the simulation through a *DLL* compilation of the *C/C++* function modeling each *PE* of the *NoC*. This *DLL* is defined as the architectural parts of *PEs* in the *VHDL* description of the simulated *NoC* in *ModelSim*. Fig. 7 shows the integration of the *C/C++* functionality modeling *PEs* in the VHDL description of the *NoC*.

Fig. 6 presents the file contents associated to each *PE* and containing the packets (of 5 *flits* size) to transmit. The first *flit* corresponds to the source address of the emitting *PE*. The second one is the address of the destination *PE*. The next *flits* are the data to transmit between two *PEs* of the network.


out_10x10_xy_0 - Bloc-notes
Fichier Edition Format Affichage ?

```
100 017 0A1 0A2 0A3
100 018 0A1 0A2 0A3
100 040 0A1 0A2 0A3
100 050 0A1 0A2 0A3
100 016 0A1 0A2 0A3
100 018 0A1 0A2 0A3
100 02B 0A1 0A2 0A3
100 023 0A1 0A2 0A3
100 05A 0A1 0A2 0A3
100 017 0A1 0A2 0A3
100 01B 0A1 0A2 0A3
100 045 0A1 0A2 0A3
100 04B 0A1 0A2 0A3
100 01E 0A1 0A2 0A3
```

Figure 6. Data packets examples to transmit by PE address 00

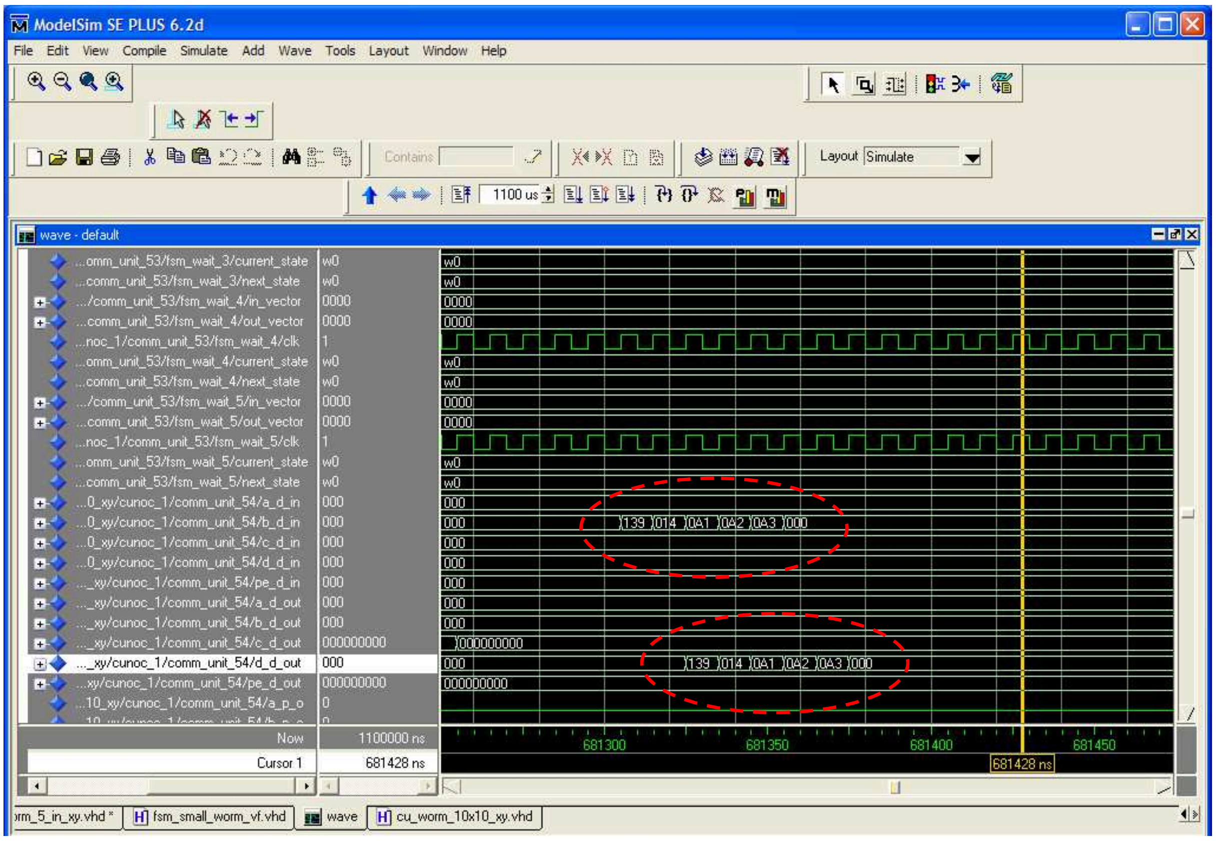
During the co-simulation phases, students are aware about the *deadlock*, *livelock* and *starvation* risks of data packets in the modeled *NoC* [1]. These risks are mainly due to the limited resources sharing and the resources access rules. They are usually studied and analyzed in the *NoC* design suitable for the conception of specific *MPSoC*. Fig. 8 and 9 give examples of co-simulation results in *ModelSim*. Fig. 8 presents a data packets received from a *wormhole switching* using the *XY* routing in the switch associated with the *PE-54*. Fig. 9 shows the reception of data packets by the *PE_54*. This co-simulation allows a rapid behavioral validation of the design *NoC* while highlighting his characteristic performances as the *latency* and *throughput* notions. Fig. 10 gives an example of generated files by the simulation results given the *latency* results in terms of the cycle number of transmitted packets in the *NoC* by an emitter *PE* toward a destination *PE*.

IV. CONCLUSION

This paper proposes, for educational purposes, the modeling, simulation and performance evaluation of the chip network communication architecture using *C-VHDL* co-simulation with *ModelSim*. This approach overcomes some understanding difficulties encountered during the teaching of the communication concepts in *Multi-Processor System on Chip* (*MPSoC*) design. It alerts the students to the fundamental impact of the interconnection in *MPSoC* to meet the required performances. The data packet transmission modeling and simulation is described in *C/C++* allowing the rapid simulation required to validate the functionality, and the performances of a *Network on Chip.* Routers are described in VHDL aiming the synthesis towards FPGA technology. The educational purpose is to provide the students the fundamental concepts in the design, and development of on chip interconnection, a significant key of *MPSoC* performances.

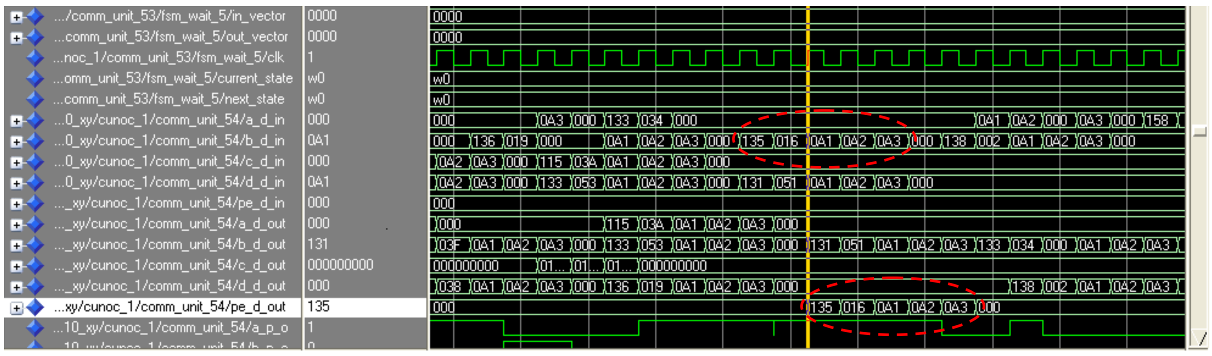
```
architecture arch_c_module_10x10_xy of c_module_10x10_xy is

attribute foreign:string;

attribute foreign of arch_c_module_10x10_xy: architecture is "c_module_10x10_xy_init ./c_module_10x10_xy_with_lat_vf_6x6_5flits.dll";

begin
end;
```

Figure 7. Attribution *FLI* of abstract model by *DLL* in the VHDL description.



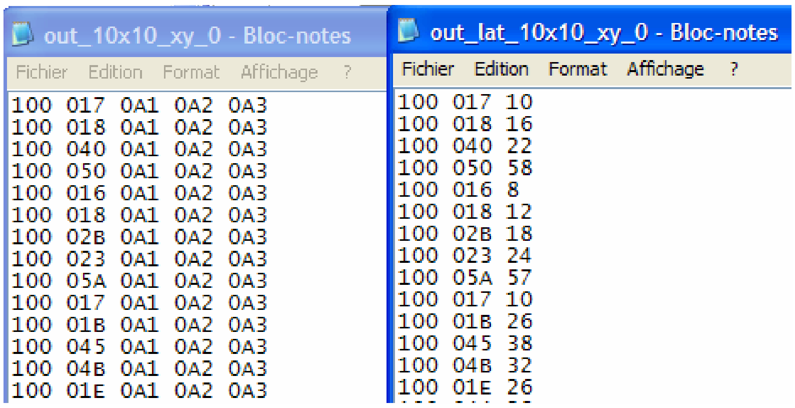
ModelSim SE PLUS 6.2d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Layout Simulate
wave - default
...comm_unit_53/fsm_wait_3/current_state w0
...comm_unit_53/fsm_wait_3/next_state w0
.../comm_unit_53/fsm_wait_4/in_vector 0000
...comm_unit_53/fsm_wait_4/out_vector 0000
...noc_1/comm_unit_53/fsm_wait_4/clk 1
...comm_unit_53/fsm_wait_4/current_state w0
...comm_unit_53/fsm_wait_4/next_state w0
.../comm_unit_53/fsm_wait_5/in_vector 0000
...comm_unit_53/fsm_wait_5/out_vector 0000
...noc_1/comm_unit_53/fsm_wait_5/clk 1
...comm_unit_53/fsm_wait_5/current_state w0
...comm_unit_53/fsm_wait_5/next_state w0
...0_xy/cunoc_1/comm_unit_54/a_d_in 000
...0_xy/cunoc_1/comm_unit_54/b_d_in 000
...0_xy/cunoc_1/comm_unit_54/c_d_in 000
...0_xy/cunoc_1/comm_unit_54/d_d_in 000
..._xy/cunoc_1/comm_unit_54/pe_d_in 000
..._xy/cunoc_1/comm_unit_54/a_d_out 000
..._xy/cunoc_1/comm_unit_54/b_d_out 000
..._xy/cunoc_1/comm_unit_54/c_d_out 000000000
..._xy/cunoc_1/comm_unit_54/d_d_out 000
...xy/cunoc_1/comm_unit_54/pe_d_out 000000000
...10_xy/cunoc_1/comm_unit_54/a_p_o 0
139 014 0A1 0A2 0A3 000
139 014 0A1 0A2 0A3 000
Now 1100000 ns
Cursor 1 681428 ns
681300 681350 681400 681450
681428 ns
rm_5_in_xy.vhd * fsm_small_worm_vf.vhd wave cu_worm_10x10_xy.vhd

Figure 8. Simulation results of the transmit data packets in the modeled *NoC* based on the *Wormhole* switching and *XY* algorithm.



.../comm_unit_53/fsm_wait_5/in_vector 0000
...comm_unit_53/fsm_wait_5/out_vector 0000
...noc_1/comm_unit_53/fsm_wait_5/clk 1
...comm_unit_53/fsm_wait_5/current_state w0
...comm_unit_53/fsm_wait_5/next_state w0
...0_xy/cunoc_1/comm_unit_54/a_d_in 000
...0_xy/cunoc_1/comm_unit_54/b_d_in 0A1
...0_xy/cunoc_1/comm_unit_54/c_d_in 000
...0_xy/cunoc_1/comm_unit_54/d_d_in 0A1
..._xy/cunoc_1/comm_unit_54/pe_d_in 000
..._xy/cunoc_1/comm_unit_54/a_d_out 000
..._xy/cunoc_1/comm_unit_54/b_d_out 131
..._xy/cunoc_1/comm_unit_54/c_d_out 000000000
..._xy/cunoc_1/comm_unit_54/d_d_out 000
...xy/cunoc_1/comm_unit_54/pe_d_out 135
...10_xy/cunoc_1/comm_unit_54/a_p_o 1
135 016 0A1 0A2 0A3 000

Figure 9. Simulation results of the received data packets in the *PE_54*.


out_10x10_xy_0 - Bloc-notes
Fichier Edition Format Affichage ?
out_lat_10x10_xy_0 - Bloc-notes
Fichier Edition Format Affichage ?

```
100 017 0A1 0A2 0A3
100 018 0A1 0A2 0A3
100 040 0A1 0A2 0A3
100 050 0A1 0A2 0A3
100 016 0A1 0A2 0A3
100 018 0A1 0A2 0A3
100 02B 0A1 0A2 0A3
100 023 0A1 0A2 0A3
100 05A 0A1 0A2 0A3
100 017 0A1 0A2 0A3
100 01B 0A1 0A2 0A3
100 045 0A1 0A2 0A3
100 04B 0A1 0A2 0A3
100 01E 0A1 0A2 0A3
```

```
100 017 10
100 018 16
100 040 22
100 050 58
100 016 8
100 018 12
100 02B 18
100 023 24
100 05A 57
100 017 10
100 01B 26
100 045 38
100 04B 32
100 01E 26
```

Figure 10. Latency results in term of clock simulation cycle of the transmitted data packets by the *PE_00*.

This experience covers the major *NoC* concepts for *MPSoC* design from postgraduate education to Ph.D level education. Moreover, it shows the usefulness of tool like *ModelSim* in the validation of a functional architecture in its working environmental context through co-simulation. This lab is complementary to the teaching of high level modeling system languages such as *SystemC*. Indeed, the increasing complexity of *System on Chip* rely more and more on specific tools whose mastery is fundamental for microelectronic design education.

REFERENCES

[1] G. De Micheli et L. Benini, « Networks on Chips, Technology and tools », Morgan Kaufmann publishers, 2006.

[2] Mentor Graphics, « Modelsim SE User’s Manuel, Sofware, Version 6. 4 », 2008. http://www.mentor.com/.

[3] Mentor Graphic, « ModelSim, Foreign Language Interface, version 5.6d», August 2002.

Experimental Fault Injection based on the Prototyping of an AES Cryptosystem

Jean-Baptiste Rigaud*, Jean-Max Dutertre*, Michel Agoyan†, Bruno Robisson†, Assia Tria†,
*École Nationale Supérieure des Mines de Saint Étienne,†CEA-LETI,
SAS Department,
Centre Microélectronique de Provence Georges Charpak,
880 avenue de Mimet 13541 Gardanne, FRANCE
Email: *name@emse.fr,†firstname.name@cea.fr

***Abstract*—This paper presents a practical work for Masters students in Microelectronics Design with optional modules in cryptography and secured circuits. This work targets the design and the prototyping of the Advanced Encryption Standard algorithm on a Spartan 3 Xilinx platform and how fault injection techniques could jeopardize the secrecy embedded within a design dedicated to the security thanks to simple equipments: a fault injection platform based on the use of an embedded FPGA's Delay Locked Loop.**

I. INTRODUCTION

This paper proposes a wide spectrum practical work for Masters students in Microelectronics Design with optional modules in cryptography and secured circuits. This is the development of an experimental fault injection based on the prototyping of an AES cryptosystem. The main goal is the application of academic courses around VHDL, design methodology, FPGA prototyping, cryptography and security of integrated circuits. The work has two parts. The first one is the VHDL implementation of a standard symmetric key algorithm: a 128-bit AES. Then, this cypher block is prototyped on Spartan 3 development board [12]. A serial communication interface completes the design. It allows communication between a PC and the Xilinx board using ad hoc commands. Lastly, the automated generation of test programs for this environment is addressed. All this part is mainly composed of lab work.

The second part of the course is dedicated to the design of a fault injection platform. It includes both lectures and laboratory work. The lectures consist in introducing the theory of digital ICs' timing constraints, Differential Fault Analysis (DFA) and the use of Xilinx FPGA's Digital Clock Managers (DCM). Then, during the laboratories, the students apply these principles to designing fault injection platform (implemented on a Xilinx Virtex 5 demo board), and performing fault injection experiments by various means on the previously designed AES. This course part is devoted to make the students aware of fault attacks against cryptosystems.

II. THE ATTACKED CIRCUIT : AES CRYPTOSYSTEM

The first part focuses on the description of the cryptosystem to be designed. Second, the AES VHDL modeling and prototyping is presented. Then, the AES test environment is targeted with the insertion of an UART interface. Finally, a test pattern generation program is proposed using Perl programming language.

A. Advanced Encryption Standard Algorithm

The Rijndael block cipher [4] has been standardized as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST) in 2001 [8] . It replaces the Data Encryption Standard (DES) for symmetric-key encryption.

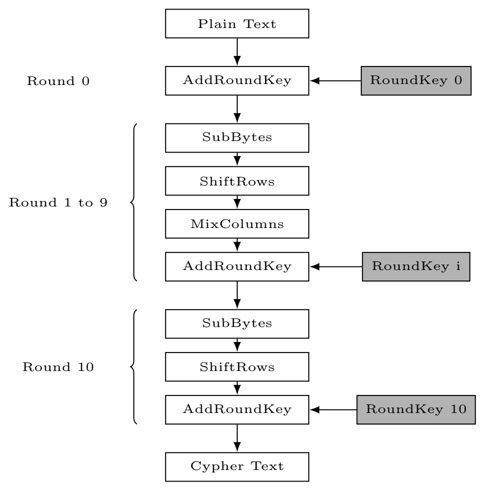


Fig. 1. AES algorithm

It is a Substitution-Permutation Network (SPN) block cipher whose operations are based on binary extension fields. It processes a 128-bit plaintext and a key of 128, 192 or 256 bits long to produce a 128-bit ciphertext. From now on, we only consider a 128-bit key AES. The AES encryption algorithm is divided into two processes: the "Data path" and the "Key Schedule". The words processed by these two parts are two 4x4 matrices of bytes called "states".

1) Data path Encryption Process: AES has an iterative structure (Fig. 1). The data path is a sequence of ten sub-processes called "rounds". A regular round is composed of the four transformations called “SubBytes”, “ShiftRows”, “MixColumns” and “AddRoundKey”. Before the first round, the original key and the plaintext are added. The last round uses all but the MixColumns transformation.

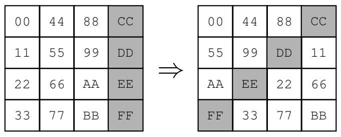


Fig. 2. ShiftRows transformation

SubBytes transformation substitutes each byte of the state according to a table called “Sbox”.

ShiftRows transformation is a cyclic permutation on the state rows (fig.2).

MixColumns is a matrix product of the current state S by a constant matrix C over $GF(2^8)$ (fig.3).

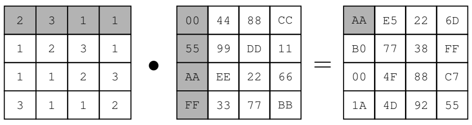


Fig. 3. MixColumns transformation

The AddRoundKey transformation adds (using a bit wise XOR) the round data to the round key processed by the KeySchedule.

2) KeySchedule Process: Each round key is derived from the previous one through the operation described in Figure 4. The first round key (RoundKey 0) is the secret key. For each round two transformations, “RotWord” and “SubBytes” are applied to the last column of the round key state. This column is then added to a round constant (“RCon”). The next round key state is obtained column wise: the next first column is the result of a XOR operation between this modified column and the former first one. Each following column of the former state is added to the column computed just before.

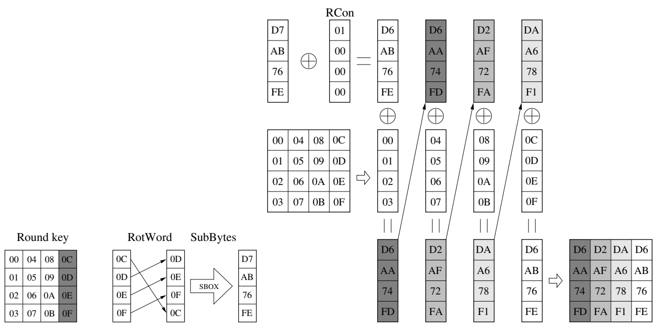


Fig. 4. Keyscheduling

B. AES Design

The AES is a good example for teaching integrated circuits design: it involves a lot of architecture dilemnas with very interesting issues such as the synchronization of data path and key expansion or the Sbox modeling. This task provides the students with the opportunity to apply by themselves the whole FPGA design flow.

1) Specifications and framework: In addition to the AES standard, the following specifications are given to the students:

- The inputs and outputs are 128-bit long.
- A “start” and a “done” signals trigger the beginning and the end of each encryption.
- An asynchronous active low reset initializes the whole circuit.
- The clock nominal frequency is 100 MHz.
- A complete cyphering has to be done in eleven clock cycles.
- There is no area constraint.

The following design constraints are directly linked to the evaluation environment described in the next section. The AES clock is also provided by the external environment and connected to the second FPGA board (cf. III). The “start” signal is a trigger for the clock fault generator (fig. 11). The target is a Spartan-3AN (*XC3S700AN*) evaluation board . All the circuits and test benches are modeled in VHDL. The simulations (at functional, post-synthesis and post-place and route levels) are performed with *Modelsim*^{TM} from *Mentor Graphics*. The synthesis, place and route and bitstream generation are performed with the *Xilinx ISE*^{TM} development suite.

2) AES Hardware Description: The AES circuit is composed of three parts (fig. 5) which are the “Data Path” which encompasses the four transformations, the “KeyExpander” and the “StateController” which is the finite state machine sequencing the entire algorithm. The StateController enables signals to activate each part of the circuit. Multiplexers and demultiplexers are inserted in order to transfer and select data between different rounds (they are also controlled by this subcircuit). KeySchedule computes the round keys on the fly and each round is computed within a clock cycle. Data is latched just before the S-Boxes (both in the data and key paths).

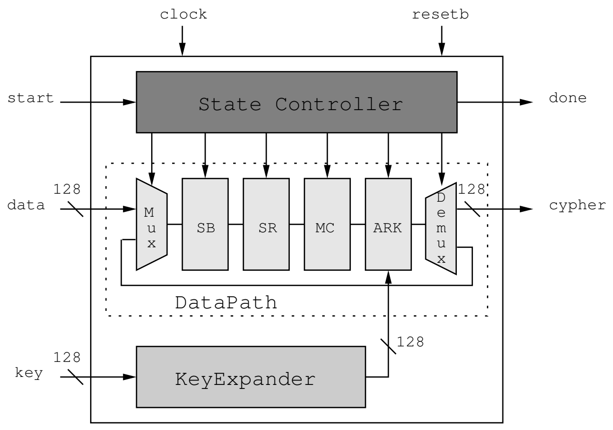


Fig. 5. AES block diagram

The timing constraint of 11 clock cycles per cyphering means that 16 S-Boxes are needed for the SubBytes operation and 4 for the KeyExpansion. Several solutions exist to design S-Boxes ([11], [7]). In this paper we choose the look-up table approach. It is also possible to use integrated block RAM (BRAM) configured as ROM used depending on the FPGA target chosen. The latter approach allows the students to discover a way to generate memory blocks with *Xilinx Coregen* application for example.

C. Evaluation Environment

One of the goals of this course is to give a concrete lab work in design debugging or design testing. Once the AES modeling is completed and all the simulations work fine, it is important to test it in a real environment scenario.

Here, a dedicated evaluation environment between a PC and the Spartan-3 board is proposed to the students. A serial communication based on the RS232 protocol is used. Figure 6 shows the communication scheme between PC and AES block.

1) A Simple Communication Protocol: The aim in this part is to perform cyphering, i.e. to program the key, give a plaintext and retrieve the cypher. The following commands are used:

- m or M followed by 64 ASCII characters followed by ”EOL”: to send a 128-bit message
- k or K followed by 64 ASCII characters followed by ”EOL”: to send a 128-bit key
- g or G followed by ”EOL”: stands for go, to start the encryption

The interface has to acknowledge the *M* and *K* commands by sending the string of 2 characters *OK*. The go command is acknowledged by the 32 hexadecimal characters of the result and *OK*.

The same computation can be done several times where only the key or the message is changed between two AES executions. All these commands are sent via the RS232 port via an serial ”terminal”.

The automation of the cyphering will be described in subsection II-C3.

2) UART Hardware Implementation: A light UART is used to ensure communication between the PC and the AES. It only takes into account the baud rates. Each frame is 8-bit long with one stop bit. Other features such as parity checking are not considered here. This part could be improved in the future.

This entity is composed of 4 blocks :

- Baud generator: generates the correct local clock frequency depending on the baud rate. The Baud rate is initially controlled by external on-board dip switches but can be hard-coded as well.
- Transmitter: sends serialized data to the PC to acknowledge a command or to send back the cyphertext.
- Receiver: receives deserialized data from the PC.
- Controller: is a finite state machine dedicated to decoding the received commands, to control the AES and to send back results and acknowledging to the PC.

This communication interface is designed within one same AES design flow. Once the bit stream is loaded into the FPGA, the first step in the debugging process is to send a command from a serial terminal and wait for the answer (“OK”) from the FPGA. Then test patterns provided by the NIST are used.

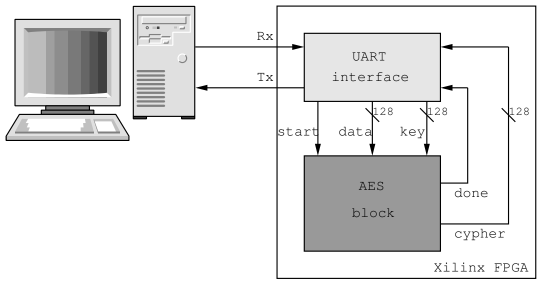


Fig. 6. AES test environment

3) An Automated Cyphering : Once the communication between the AES and the PC is validated, this first part course ends with the generation of test programs. The goal here is to create scenario files based on the communication commands (cf. II-C1). After a quick overview of Perl programming language [9], students are asked to generate multiple encryption scenarios for the FPGA. They use features like simple text handling and easy configuration of the serial port (open, close, baud rate,etc.). At last, with “*Crypt::OpenSSL::AES*” [3], a Perl wrapper module of the OpenSSL’s AES library, they can compare, on the fly, the received cyphertexts with the expected results.

With this we conclude the first part of the course. It has presented the AES algorithm, its design and prototyping on Xilinx development board, the integration of a communication interface (UART block) and the automation of AES executions with an external PC and Perl programming. This last step is very important for the experimental fault injection in the next part of the article.

III. Design and use of an FPGA-based attack platform

A. Overview of the course

This part of the course includes both lectures and laboratory work. The lectures introduce the theory of the timing constraints related to the synchronous operation of digital ICs, the concept of Differential Fault Attack (DFA) applied to the Advanced Encrytpion Standard (AES) algorithm and the use of an FPGA on board delay locked loop (DLL) to design an attack platform. The laboratory work is divided into two parts: the synthesis and test of the modified clock signal intended to inject faults and a fault injection experiment on our AES board, as designed in section II. After completing this course, students will be aware of the threats that fault injection techniques pose to the physical implementation of cryptographic algorithms.

B. Theoretical work

This subsection introduces the theoretical background used for fault injection purposes.

1) Digital IC timing constraints: Almost all digital ICs work according to the principle of synchrony: their internal computations are sampled by a global clock signal (except asynchronous circuits which are out of scope of this course). Figure 7 symbolizes the internal architecture of any synchronous circuit: combinational logic surrounded by registers (i.e. banks of D flip-flops). The data are released from the first register banks on a clock rising edge and then processed by the logic before being latched into the next bank on the next clock rising edge. It takes a certain amount of time, called the propagation delay, to process the data through the logic. As a consequence, the time between two rising edges (i.e. the clock period) depends on the propagation delay.

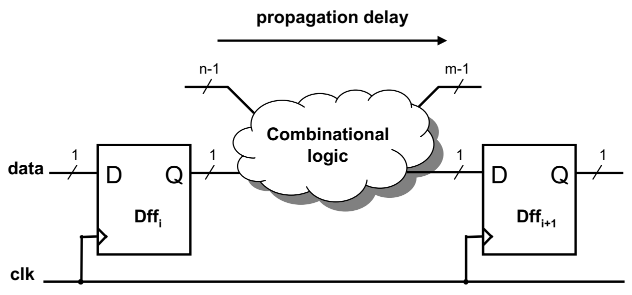


Fig. 7. Internal architecture of Digital ICs

More precisely, to operate without any computation error, the data must arrive at the second register's input before a required time. The data arrival time is expressed in equation 1, where $t_{clk\ to\ Q}$ is a latency time between the clock's rising edge and the arrival of the data on the register output Q, and where $t_{Max(prop\ delays)}$ is the biggest propagation delay through the combinational logic (namely its critical time).

$$t_{data\ arrival} = t_{clk\ to\ Q} + t_{Max(prop\ delays)} \tag{1}$$

Equation 2 gives the required time for which the data must be present. This is the sum of the clock period (T_{clk}) and a time skew (T_{skew}) which reflects the clock propagation time between the two registers minus the register's setup time (δ_{setup}), the setup time being the amount of time for which the D flip-flop input must be stable before the clock's edge to ensure reliable operation.

$$t_{data\ required} = T_{clk} + T_{skew} - \delta_{setup} \tag{2}$$

Hence, the timing constraint (Equation 3) is derived from the two previous equations:

$$T_{clk} > t_{clk\ to\ Q} + t_{Max(prop\ delays)} - T_{skew} + \delta_{setup} \tag{3}$$

The violation of the timing constraint results in computational errors. This principle is well known and frequently used as fault injection means to attack secure circuits [1].
Moreover, the faults' locations are linked to the propagation delays. Consider a combinational logic block implementing a given logical function (depicted in figure 8). Each output possesses its own propagation delay: $t_{prop\ delay(i)}$.

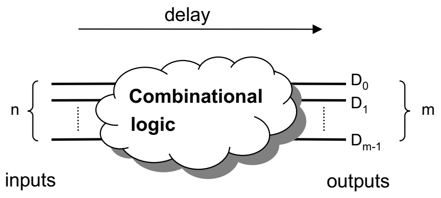


Fig. 8. Propagation delay

Then, any fault will be injected in a place (output D_i) where the propagation delay is large enough to induce a timing constraint violation as expressed in equation 4:

$$t_{prop\ delay(i)} > T_{clk} + T_{skew} - t_{clk\ to\ Q} - \delta_{setup} \tag{4}$$

Besides, these delays depend on the handled data. In other words, each propagation delay is changing with the data and so do the faults' locations (we will return to this point later in subsection III-C2). In addition, the propagation delays also depend on the power supply voltage and the chip's temperature.

2) Differential Fault Attack on AES: Fault attacks consist in injecting faults in the encryption process of a cryptographic algorithm through unusual environmental conditions. They may result in reducing the ciphering complexity (a round reduction number for example, see [2]) or the injected faults may allow an attacker to gain some information on the encryption process by comparing the correct with the faulty ciphertexts. This technique is called Differential Fault Attack (DFA). An extended explanation of DFA's theory (as done during our lectures) would be too long and of little relevance to this paper. However, the reader could find a complete description of two major DFA schemes in [10] and [6].
These attacks allow an attacker to retrieve the secret key used by the AES cryptosystem. The key points are that the fault injection means must allow the attacker to control precisely:

- the exact time of fault injection (i.e. only for a given and precise round),
- the number of faults (i.e. to limit the faults locations to a byte or even to one bit).

If not, it will be impossible to extract any information related to the encryption key.

3) FPGA-based attack platform: A basic approach to inject faults through timing constraints violation (as shown in subsection III-B1) is overclocking. It consists in decreasing the clock's period until faults appear by setup time violation. However, it provides no timing control: faults are injected at each clock cycle, which is not suitable for DFA. To overcome this, we choose local overclocking (or clock glitching), which

is based on inducing a timing violation by modifying only one clock period. The corresponding modified clock signal is denoted clk' in Figure 9.
Moreover, it offers to our students the opportunity to use and reconfigure dynamically the Delay Locked Loop (DLL) embedded in the Digital Clock Managers (DCM) of the Xilinx Virtex 5 FPGA [5].

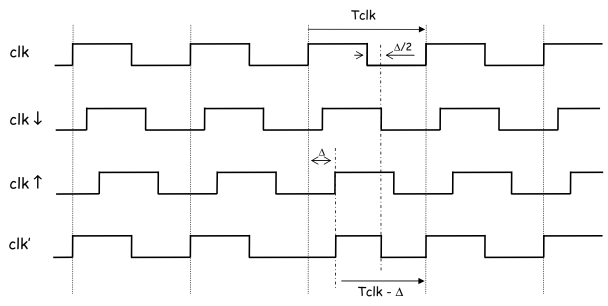


Fig. 9. Faulty clock signal generation

The modified clock signal, $clock'$, is built from a correct one, clk, as depicted in Figure 9. The DLL allows delaying an input clock signal by a programmable duration. A first $\Delta/2$ delayed clock, $clock \downarrow$, is derived from the input clock to produce the falling edge of the modified cycle and a second one Δ delayed, $clock \uparrow$, to produce its rising edge. Then, they are combined to obtain the modified clock signal, $clock'$, used for fault injection purposes. As a consequence, the duration of the corresponding clock period is decreased by an amount of time Δ. A trigger signal is used to indicate the location of the modified cycle. This technique allows choosing the fault injection cycle. Furthermore, the ability to set precisely Δ enables a fine control over the number of faulted bits (the experimental results reported in subsection III-C2 demonstrate the ability to inject one bit faults).

C. Laboratory work

The laboratory work addresses both the VHDL description and synthesis of the fault injection platform described in subsection III-B3 and the implementation of the fault injection.

1) Synthesis: This experimental work takes place after the lectures described in subsection III-B and the VHDL synthesis of the AES crytposystem described in section II have been given. The students have to complete a very concise task: design a fault injection platform based on local overclocking as described in the corresponding lecture (see subsection III-B3) on a Xilinx Virtex 5 demo board [13]. The following guidelines are given:

- use a dynamic configuration mode to set Δ via a serial communication port,
- reuse and adapt the IP already developped in section II-C to implement the communication protocol,
- set the CLOCKOUT_PHASE_SHIFT's attribute of the DLL to VARIABLE_POSITIVE to obtain an elementary variation step, δ_t, equal to 35 picoseconds for Δ,
- use a nominal clock frequency of 100 MHz (which is consistent with the AES test chip).

No instructions are given for the design of the combinational logic block devoted to obtaining the faulty clock ($clock'$ in figure 9) from the DLL input's clock signals and the trigger signal issued by the AES chip. In addition, the students need to discover by themselves that a programmable counter is required to monitor the instant the faulty period is generated to be able to target any of the AES rounds.
The achievement of this work is validated by measuring the modified clock signal for different settings of Δ and of the targeted rounds.

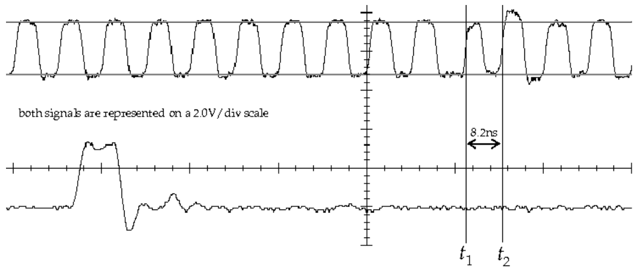


Fig. 10. Faulty clock (uppermost) and trigger signal(lowermost)

Figure 10 illustrates an oscilloscope's screen shot typically asked for validation, where Δ is set to 1.8 picoseconds and the modified period located during the ninth round of the AES.

2) Fault injection experiments: The main part of the laboratory work is focused on fault injection experiments. A first part is dedicated to fault injection by using local overclocking and to controlling precisely the injection process. Then, a second part addresses the ability to inject faults by modification of the power supply voltage and the temperature of the chip.
The experimental setup is depicted in Figure 11.

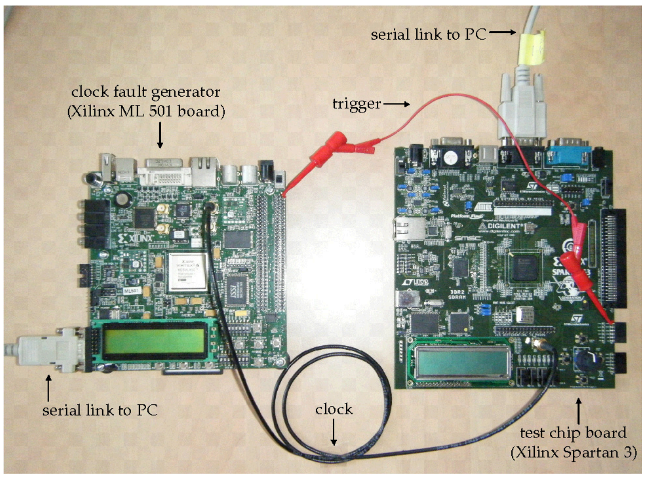


Fig. 11. Experimental setup

The first test campaign is ran as described in Algorithm 1, where δ_t is the variation elementary step of Δ.

Algorithm 1 Test Campaign Pseudo-Code

```
send the key $K$ and the plaintext $M$ to the test chip.
$\Delta \leftarrow 0$.
while (clock_period $> \Delta$) do
    encrypt and retrieve the ciphertext
    $\Delta \leftarrow \Delta + \delta_t$
end while
```

It is carried out to illustrate the fault injection process. The final round of the AES is targeted to allow a direct reading of the errors by direct comparison between a correct and a faulty ciphertexts. The AES plays the role of a big propagation delay. The bar chart of Figure 12 shows the faults' timing and nature as a function of the faulty period's duration (the horizontal axis). It uses a color code to reflect the nature of the faults (no fault, one-bit, two-bit and more faults) and their time of appearance. As expected, the comparisons between correct and faulty ciphertexts reveal that the device progressively transits from normal operation to multi-bit faults. This is done by exhibiting none, one-bit, two-bit and multiple-bit faults.

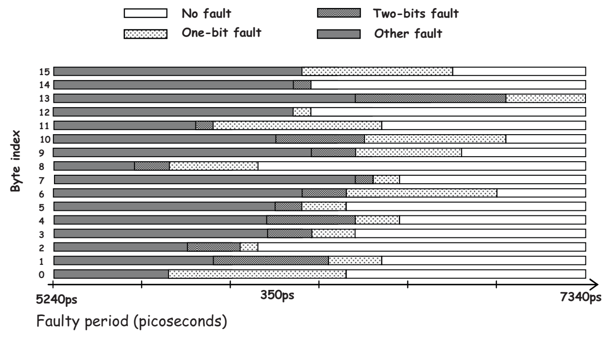


Fig. 12. Fault injection as a function of faulty period duration

These statistics were obtained thanks to the very small value of the faulty period granularity, namely 35 ps. This allows injecting a one-bit fault at every round of the ciphering process with a high degree of confidence, which is a requirement for many DFA methods.
The next point is about controlling the faults' location. The second bar chart in Figure 13 is obtained with the same experimental protocol and with the same secret key but with a different plaintext. The first injected fault is a single bit fault on byte number three for a clock period equal to 7585 ps, whereas the first one-bit fault injected in the previous experiment was on byte thirteen for a clock period equal to 7340 ps. This confirms, with many other experimental results, that the critical time's location and value vary with the data (as stated in III-B1). This experiment is carried out to illustrate the ability to change the fault location by changing the plaintext.

Another means of injecting faults is to increase the combinational logic's propagation delays until faults appear due to tim-

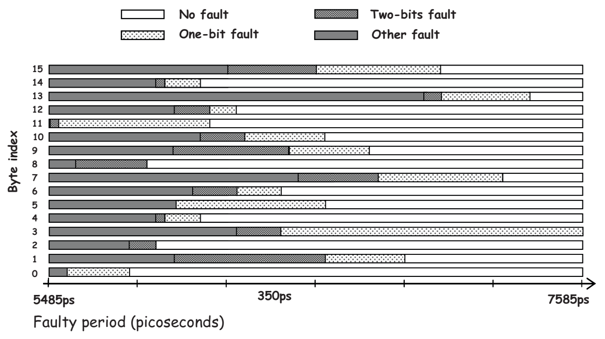


Fig. 13. Fault injection for a different plaintext

ing constraints violation according equation 3. As suggested in III-B1, this is achieved by decreasing the device's power supply voltage (V_{DD}) or increasing the chip's temperature. The students have to carry out these experiments and to report the corresponding critical time as a function of V_{DD} and the temperature. Figures 14 and 15 illustrate the corresponding typical results.

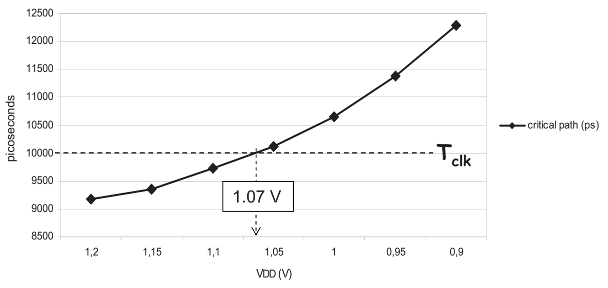


Fig. 14. Fault injection based on power supply decrease

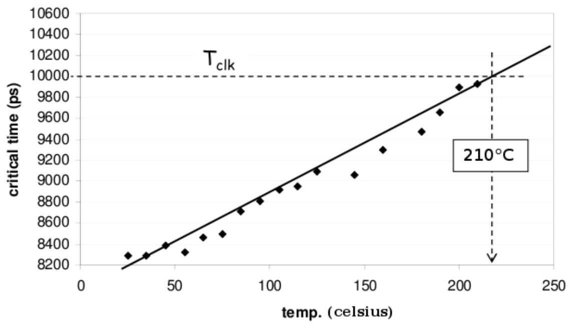


Fig. 15. Fault injection based on temperature increase

As expected, as V_{DD} decreases, the critical time increases. And when it crosses the clock period (T_{clk}) value (namely 10,000 ps) at $V_{DD} = 1.07V$, the first fault appears.

Similarly, an increase in the chip's temperature results in a linear increase in the critical time (on the variation range). Faults are injected for temperatures above 210^{o}C when the critical time goes beyond the nominal clock period.
The objective of these laboratory experiments is to make the students aware of fault injection means based on setup time violation and of the ability they offer to control the faults' sizes and locations. As a result, they will take care of the way they design crytposystems in their professional life or in further research activities.

IV. Conclusion

This paper presents an ambitious two-in-one course. It mainly targets Masters students in Microelectronics Design with optional modules in cryptography and secured circuits. It offers two parts.

The first one presents the full design of AES cryptosystem. The students start from specifications and in the end they test their prototyped circuit in a complete test environment. They have to implement the communication interface and to generate all the test sequences. The Xilinx Spartan 3 platform is used for the practical experimentations. Some evolutions can be done as modifying the data path length, exploring different architectures for the Sbox or completing the UART interface. The whole design can also be split in smaller student groups.

The second part is mainly dedicated to the design and use of an FPGA-based attack platform. At first, lectures introduce the theory of fault injection through timing constraints violation, the basis of DFA and the use of a DLL to build a modified clock signal for fault injection. Then, laboratory works allow the students to become familiar with the practice of fault injection and the combined threats over secure ICs. This second part is an optional extension of the first one. However, it could be teached independently by providing an already programmed AES test board. This part is also used as an introduction course on IC security for the PhD and internship students in our research group.

References

[1] Hagai BarEl, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. In *Special Issue on Cryptography and Security 94(2)*, pages 370–382, 2006.

[2] Hamid Choukri and Michael Tunstall. Round reduction using faults. *Proc. Second Int'l Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC '05)*, 2005.

[3] CPAN. http://search.cpan.org/ttar/crypt-openssl-aes-0.01/lib/crypt/openssl/aes.pm.

[4] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 2002.

[5] Bernhard Fechner. Dynamic delay-fault injection for reconfigurable hardware. In *Parallel and Distributed Processing IEEE Symposium*, page 282.1, Washington, DC, USA, April 2005. IEEE Computer Society.

[6] Christophe Giraud. DFA on AES. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *Advanced Encryption Standard – AES*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2005.

[7] Olivier Faurax Julien Francq. Security of several aes implementations against delay faults. In *Proceedings of the 12th Nordic Workshop on Secure IT Systems (NordSec 2007)*, October 2006.

[8] NIST. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication, n. 197, November 26, 2001.

[9] Perl. http://http://www.perl.org.

[10] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In *Proc. Cryptographic Hardware and Embedded Systems (CHES '03),*, Lecture Notes in Computer Science, pages 77–88, 2003.

[11] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An asic implementation of the aes sboxes. In *CT-RSA*, pages 67–78, 2002.

[12] Xilinx. http://www.xilinx.com/products/spartan3/3a.htm.

[13] Xilinx. http://www.xilinx.com/products/virtex5/index.htm.

Reducing FPGA Reconfiguration Time Overhead using Virtual Configurations

Ming Liu‡†, *Zhonghai Lu*†, *Wolfgang Kuehn*‡, *Axel Jantsch*†

‡ II. Physics Institute
Justus-Liebig-University Giessen (JLU), Germany
{ming.liu, wolfgang.kuehn}@physik.uni-giessen.de

† Dept. of Electronic Systems
Royal Institute of Technology (KTH), Sweden
{mingliu, zhonghai, axel}@kth.se

***Abstract*—Reconfiguration time overhead is a critical factor in determining the system performance of FPGA dynamically reconfigurable designs. To reduce the reconfiguration overhead, the most straightforward way is to increase the reconfiguration throughput, as many previous contributions did. In addition to shortening FPGA reconfiguration time, we introduce a new concept of Virtual ConFigurations (VCF) in this paper, hiding dynamic reconfiguration time in the background to reduce the overhead. Experimental results demonstrate up to 29.9% throughput enhancement by adopting two VCFs in a consumer-reconfigurable design. The packet latency performance is also largely improved by extending the channel saturation to a higher packet injection rate.**

I. Introduction

Partial Reconfiguration (PR) enables the process of dynamically reconfiguring a particular section of an FPGA design while the remaining part is still operating. This vendor-dependent technology provides common benefits in adapting hardware modules during system run-time, sharing hardware resources to reduce device count and power consumption, shortening reconfiguration time, etc. [1] [2] [3]. Typically partial reconfiguration is achieved by loading the partial bitstream of a new design into the FPGA configuration memory and overwriting the current one. Thus the reconfigurable portion will change its behavior according to the newly loaded configuration. Despite the flexibility of changing part of the design at system run-time, overhead exists in the reconfiguration process since the reconfigurable portion cannot work at that time due to the incompleteness of the configuration data. It has to wait to resume working until the complete configuration data have been successfully loaded in the FPGA configuration memory. Therefore in performance-critical applications which require fast or frequent switching of IP cores, the reconfiguration time is significant and should be minimized to reduce the overhead.

To reduce the dynamic reconfiguration overhead, the most straightforward way is to increase the data write-in throughput of the configuration interface on FPGAs, specifically the Internal Configuration Access Port (ICAP) on Xilinx FPGAs [4]. As an additional approach, we address the challenge of reducing the reconfiguration overhead by employing the concept of virtualization on FPGA configuration contexts. The remainder of the paper will be organized as follows: In Section II, related work of reducing dynamic reconfiguration overhead will be discussed. In Section III, we introduce the concept of Virtual ConFigurations (VCF), with which the dynamic reconfiguration time may be fully or partly hidden in the background. Experimental results will demonstrate the performance benefits of using VCFs in Section IV, in terms of data delivery throughput and latency. Finally we conclude the paper and propose our future work in Section V.

II. Related Work

FPGA dynamic reconfiguration overhead refers to the time spent on the module reconfiguration process. At that time, the reconfigurable region on the FPGA cannot effectively work due to the lack of a complete bitstream, and consequently it has negative effects on the system performance. Reconfiguration overhead may be minimized either with a reasonable scheduling policy which decreases the context switching times of hardware modules, or by reducing the required time span for each configuration. There is related discussion on the former approach in our previous publication of [5]. We observe from the experimental results that only less than 0.3% time is spent on the configuration switching with a throughput-aware scheduling policy, not exacerbating much the overall processing throughput of the under-test system; With regard to the latter approach, design optimization approaches have been previously adopted to increase the configuration throughput. For instance in [6], [7] and [8], authors explore the design space of various ICAP designs and enhance the reconfiguration throughput to the order of magnitude of Megabytes per second. Unfortunately the reconfiguration time is still constrained by the physical bandwidth of the reconfiguration port on FPGAs. Other approaches of compressing the partial bitstreams are discussed in [8] and [9] for shrinking the reconfiguration time under the precondition of a fixed configuration throughput. In addition to all the above described contributions, in this paper we will address the challenge of reducing the reconfiguration overhead, employing the concept of virtualization on FPGA configuration contexts.

III. Virtual Configuration

In canonical PR designs, one Partially Reconfigurable Region (PRR) has to stop from working, when a new module is to be loaded to replace the existing one by run-time reconfiguration. This is the overhead of switching hardware processes, which restricts the overall system performance. As a solution, we propose the concept of Virtual ConFiguration (VCF) to hide the configuration overhead of a PR design. As shown in Figure 1, two copies of configuration contexts, each of which represents a VCF, are altogether dedicated to a single PRR on a multi-context FPGA [10] [11] [12]. The active VCF may still keep working in the foreground when module switching is expected. The run-time reconfiguration only happens invisibly in the background, and the new partial bitstream is loaded into configuration context 2. After the reconfiguration is finished, the newly loaded module can start working by being swapped with the foreground context, migrating from the background to the foreground. The previously active configuration will be deactivated into the background and wait for the next time reconfiguration. The configuration context swapping between the background and the foreground is logically realized by changing the control on the PRR among different VCFs. It does not need to really swap the configuration data in the FPGA configuration memory, but instead switches control outputs taking effect on the PRR using multiplexer (MUX) devices. Hence the configuration context swapping takes only very short time (normally some clock cycles), and is tiny enough to be negligible compared to the processing time of the system design.

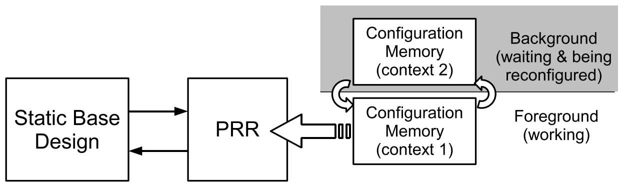

Static Base Design
PRR
Configuration Memory (context 2)
Background (waiting & being reconfigured)
Configuration Memory (context 1)
Foreground (working)

Figure 1. Virtual reconfigurations on multi-context FPGAs

With the approach of adopting VCFs, the reconfiguration overhead can be fully or partly removed with the duplicated configuration contexts. The timing advantage is illustrated in Figure 2, comparing to the canonical PR designs without VCFs. We see in Figure 2a, the effective work time and the reconfiguration overhead have to be arranged in sequence on the time axis, in the canonical PR design without VCFs. By contrast in Figure 2b, the reconfiguration process only happens in the background and the time overhead is therefore hidden by the working VCF in the foreground.

In normal FPGAs with only single-context configuration memories, VCFs may be implemented by reserving duplicated PRRs of the same size (see Figure 3). At each time, only one PRR is allowed to be activated in the foreground and selected to communicate with the rest static design by MUXes. The other PRR waits in the background for reconfiguration and will be swapped to the foreground to work after the module is successfully loaded. Taking into account the resource utilization overhead of reserving duplicated PRRs, usually we do not adopt more than 2 VCFs.

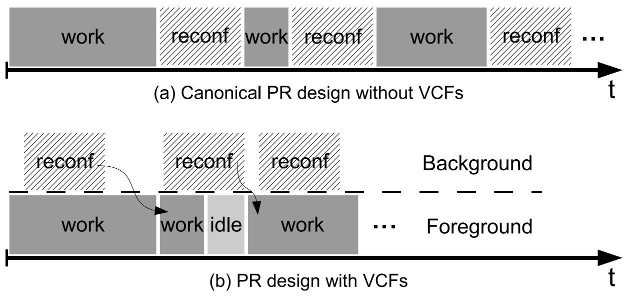


Figure 2. Timing diagrams of PR designs without or with VCFs

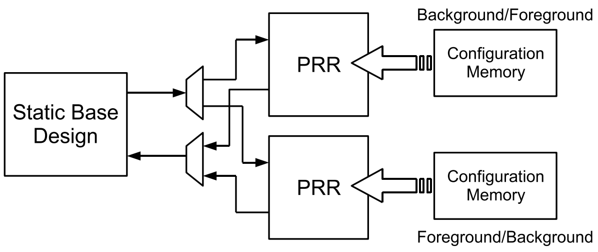


Figure 3. Virtual reconfigurations on single-context FPGAs

IV. Experiments

A. Experimental Setup

To investigate the impact of VCFs on performance, we set up a producer-consumer design with run-time reconfiguration capability. As illustrated in Figure 4, the producer periodically generates randomly-destined packets to 4 consumers and buffers them in 4 FIFOs. Each FIFO is dedicated to a corresponding consumer algorithm, which can be dynamically loaded into the reserved consumer PRR. The scheduler program monitors the “almost_full” signals from all FIFOs and arbitrate the to-be-loaded consumer module using a Round-Robin policy. Afterwards, the loaded consumer will consume its buffered data in a burst mode, until it has to be replaced by the winner of the next-round reconfiguration arbitration. The baseline canonical PR design has only one configuration context and must stop the working module before the reconfiguration starts. In the PR design with VCFs, we adopt only two configuration contexts since the on-chip area overhead of multiple configuration contexts should be minimized. Experimental measurements have been carried out in cycle-accurate simulation using synthesizable VHDL codes. Simulation provides much convenience for observing all the signals in the waveform and debugging the design. It will have the same results when implementing the design on any dynamically reconfigurable FPGA. Both the baseline and the VCF designs run at a system clock of 100 MHz. The overall on-chip buffering capability is parameterized in the order of KiloBytes. For the reconfiguration time of each module, we select 10

μs which is a reasonable value when using the practical Xilinx ICAP controller for partial reconfiguration [7]. The generated packets are 256-bit wide. The FIFO width is 32 bits. Before packets go into the FIFO, they are fragmented into flits.

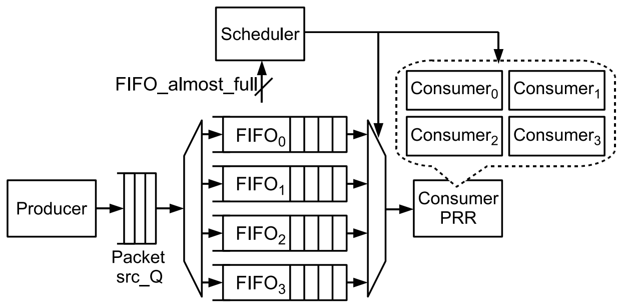


Figure 4. Experimental setup of the consumer-reconfigurable design

B. Results

We did measurements on the received packet throughput in the unit of packets per cycle per consumer node, with the FIFO depth of 512, 1K and 2K respectively. Measurement results are demonstrated in Figure 5. We observe from the figure that:

1) As the packet injection rate increases, the on-chip communication becomes saturated progressively due to the limitation of the packet consuming capability;
2) For both types of PR designs (red or light curves for with 2 VCFs and blue or dark curves for without), larger FIFO depths lead to higher saturated throughput, since the data read-out burst size can be increased by larger buffering capability, and the reconfiguration time overhead is comparatively reduced;
3) Introducing VCFs can further reduce the reconfiguration overhead by hiding the reconfiguration time in the background. In the most obvious case of 1K FIFO depth, two VCFs increase the throughput from 0.0127 packets/cycle/node to 0.0165, achieving a performance enhancement of 29.9%. Other two cases of 512 and 2K FIFO depth have a performance enhancement of 26.4% and 17.9% respectively.

We enlarged the time span of each configuration from 10 μs to 50 μs and did further throughput measurements with a middle-size FIFO depth of 1K. Results are demonstrated in Figure 6, comparing the PR design using 2 VCFs with the one without VCF. We observe that the overall system throughput is worsened by the increased reconfiguration time overhead, specifically from a saturated value of 0.0127 (see Figure 5) into 0.00492 packets/cycle/node for the non-VCF design. The increased reconfiguration time also easily results in the channel saturation at an even lower packet injection rate of about 1 packet per 50 cycles. In this test, we can still see the performance improvement of 27.6%

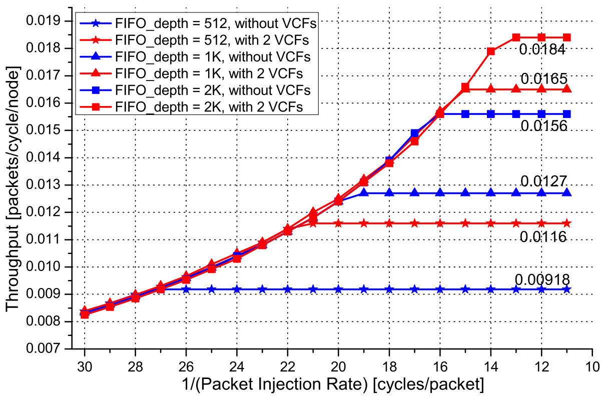


Figure 5. Throughput measurement results (reconfiguration time = 10 μs)

(0.00628 vs. 0.00492 packets/cycle/node), using 2 VCFs to partly counteract the reconfiguration overhead. The channel saturation point is extended to about 1 packet per 35 cycles by duplicated VCFs

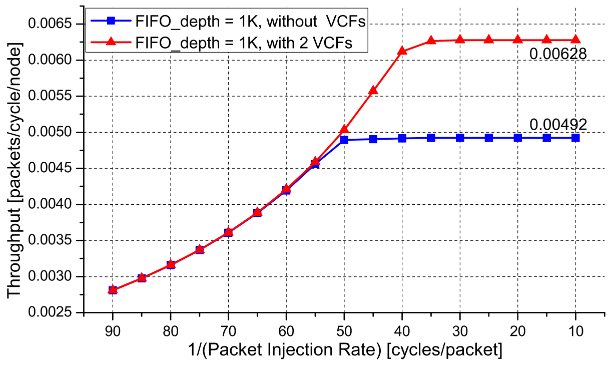


Figure 6. Throughput measurement results (reconfiguration time = 50 μs)

Except for the throughput comparison, we collected also statistics on packet latency performance to demonstrate the effect of using VCFs. We discuss the average latency of a certain amount of packets, and exclude the system warm-up and cool-down cycles out of measurements, only taking into account steady communications. The latency is calculated from the instant when the packet is injected into the source queue to that when the packet is received by the destination node. It consists of two components: the queuing time in the source queue and the network delivery time in flit FIFOs. Measurements were conducted in the experimental setup with the smaller reconfiguration time of 10 μs and the middle-size FIFO depth of 1K. Results are illustrated in Figure 7. We observe that 2 VCFs have a slight reduction effect on the packet latency before the channel saturation. In this curve segment, packets do not stay in the source queue for too long time, but they must wait in flit FIFOs until their specific destination node is configured to read them out in a burst mode. Therefore we see two comparatively flat curve segments before the channel saturation, because of the steady switching frequency of consumer nodes. Nevertheless after the channel's packet delivery capability is saturated, packets have to spend much time waiting in the source queue to enter the flit FIFOs. Thus the average latency of packets deteriorates significantly and generates rising curve segments in the figure. By contrast, using 2 VCFs may reduce the reconfiguration overhead and extends the channel saturation to a higher packet injection rate. It reduces the packet wait time in the source queue and introduce them into the flit FIFOs at an early time, leading to a large improvement on the packet latency performance.

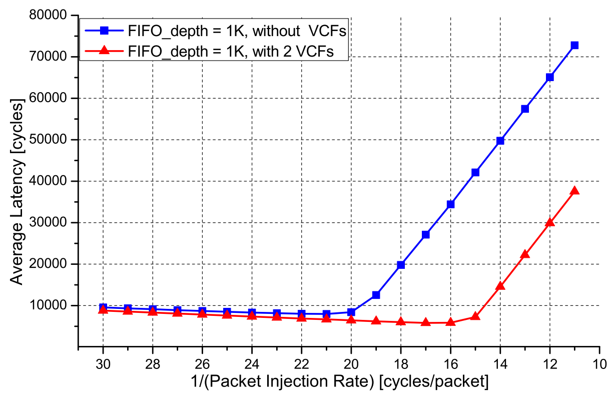


Figure 7. Latency measurement results (reconfiguration time = 10 μs)

V. Conclusion and Future Work

In this paper, we introduce the concept of virtual configurations to hide the FPGA dynamic reconfiguration time in the background and reduce the reconfiguration overhead. Experimental results on a consumer-reconfigurable design demonstrate up to 29.9% throughput improvement of received packets by each consumer node. The packet latency performance is largely improved as well, by extending the channel saturation to a higher packet injection rate. This approach is well suited for PR designs on multi-context FPGAs. For single-context FPGAs, performance improvement is accompanied by resource utilization overhead of reserving duplicated PR regions.

In the future work, we will take advantage of VCFs in practical PR designs for specific applications. Research and engineering work on multi-context FPGAs will also be useful to popularize this technology in dynamically reconfigurable designs with high performance requirements.

Acknowledgment

This work was supported in part by BMBF under contract Nos. 06GI9107I and 06GI9108I, FZ-Juelich under contract No. COSY-099 41821475, HIC for FAIR, and WTZ: CHN 06/20.

References

[1] C. Kao, "Benefits of Partial Reconfiguration", *Xcell Journal*, Fourth Quarter 2005, pp. 65 - 67.

[2] E. J. Mcdonald, "Runtime FPGA Partial Reconfiguration", *In Proc. of 2008 IEEE Aerospace Conference*, pp. 1 - 7, Mar. 2008.

[3] C. Choi and H. Lee, "An Reconfigurable FIR Filter Design on a Partial Reconfiguration Platform", *In Proc. of First International Conference on Communications and Electronics*, pp. 352 - 355, Oct. 2006.

[4] Xilinx Inc., "Virtex-4 FPGA Configuration User Guide", UG071, Jun. 2009.

[5] M. Liu, Z. Lu, W. Kuehn, and A. Jantsch, "FPGA-based Adaptive Computing for Correlated Multi-stream Processing", *In Proc. of the Design, Automation & Test in Europe conference*, Mar. 2010.

[6] J. Delorme, A. Nafkha, P. Leray and C. Moy, "New OPBHW-ICAP Interface for Realtime Partial Reconfiguration of FPGA", *In Proc. of the International Conference on Reconfigurable Computing and FPGAs*, Dec. 2009.

[7] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration", *In Proc. of the International Conference on Field Programmable Logic and Applications*, Aug. 2009.

[8] S. Liu, R. N. Pittman, and A. Forin, "Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller", *In Proc. of the International Symposium on Field-Programmable Gate Arrays*, Feb. 2010.

[9] J. H. Pan, T. Mitra, and W. Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs", *In Proc. of the International Conference on Computer-Aided Design*, Nov. 2004.

[10] Y. Birk and E. Fiksman, "Dynamic Reconfiguration Architectures for Multi-context FPGAs", *International Journal of Computers and Electrical Engineering*, Volume 35, Issue 6, Nov. 2009.

[11] M. Hariyama, S. Ishihara, N. Idobata and M. Kameyama, "Non-volatile Multi-Context FPGAs using Hybrid Multiple-Valued/Binary Context Switching Signals", *In Proc. of International Conference Reconfigurable systems and Algorithms*, Aug. 2008.

[12] K. Nambaand H. Ito, "Proposal of Testable Multi-Context FPGA Architecture", *IEICE Transactions on Information and Systems*, Volume E89-D, Issue 5, May. 2006.

Timing Synchronization for a Multi-Standard Receiver on a Multi-Processor System-on-Chip

Roberto Airoldi, Fabio Garzia and Jari Nurmi
Tampere University of Technology
Department of Computer Systems
Korkeakoulunkatu 1, P.O. Box 553, FI-33101
Tampere, Finland
email: *name.surname*@tut.fi

Abstract—**This paper presents the implementation of timing synchronisation for a multi-standard receiver on Ninesilica, a homogeneous Multi-Processor System-on-Chip (MPSoC) composed of 9 nodes. The nodes are arranged in a mesh topology and the on chip communication is supported by a hierarchical Network-on-Chip. The system is prototyped on FPGA. The mapping on Ninesilica of the timing synchronisation algorithms showed a good parallelization efficiency leading to speed-ups up to 7.5x when compared to a single processor architecture.**

I. Introduction

In the last twenty years the development of embedded systems has been driven by the boom of wireless technology. A continuous development of new wireless protocols has imposed new constraints on the receivers. Today's state of the art devices are able to work over a heterogeneous set of networks, such as: 2G, 3G, bluetooth Wi-Max and Wi-Fi. Traditional design approaches, based on a collage of single-standard receivers, introduce limitations in the number of supported wireless protocols due to constraints such as power and area consumption, flexibility and efficiency. In the past few years research institutes from both academia and industry moved their focus towards Software Defined Radio (SDR). SDR platforms might be a feasible approach to implement highly flexible transceivers. Ideally a SDR terminal would be able to work over different networks just re-configuring/re-programming the software running on the platform, according to the user's willing. Hence SDR enabling platform must provide a high computational power to meet the strict real-time requirements of today's and tomorrow's wireless standards within high flexibility. Many solutions have been proposed from both academia and industry to meet the requirements for SDR applications. Reconfigurable architectures (such as Montium [1]) as well as DSP solutions (see [2] [3]) have been widely explored for SDR applications. Furthermore, in the past few years Multi-Processor Systems-on-Chip (MPSoCs) have gained a growing interest from the research community as a possible way to meet high performance required by wireless standards within high flexibility[4].

Two communication protocols are mostly utilised for the physical layer processing in wireless communications: W-CDMA and OFDM. In this paper the authors present the implementation of the timing synchronisation procedure for W-CDMA and OFDM systems on a homogeneous MPSoC.

This paper is organised as follows: in the next section a brief overview of the Multi-processor System-on-Chip architecture is given; in section III timing synchronisation for W-CDMA and OFDM systems and their mapping onto the MPSoC are analysed; finally in section IV and V results and conclusions are drawn.

II. Ninesilica MPSoC overview

Ninesilica is a homogeneous MPSoC composed of nine nodes arranged in a 3x3 mesh topology. Ninesilica is derived from the Silicon Café template, developed at Tampere University of Technology. The template allows the creation of either heterogeneous or homogeneous multi-processor architecture with a generic number of nodes. The communication between nodes take place through a hierarchical Network-on-Chip (NoC)[6] that taps directly into the node communication system.

Fig. 1 presents a schematic view of Ninesilica while figure 2 shows the internal structure of a single node. Each node hosts a COFFEE RISC processor [5], data and instruction memories and a Network Interface (NI). The NI is composed of two parts: initiator and target. The initiator is responsible for routing remote data (coming from the NoC) inside the node. On the other hand the target is responsible to route data from the node to the NoC. The central nodes is also equipped with I/O interfaces and takes care of the data distribution among the other nodes. Moreover it acts as a schedule manager distributing tasks to the others nodes (also referred as computational nodes). Indeed its main task is to control the communication flow and the other nodes activities.

The system was prototyped on an Stratix IV FPGA device. The synthesis results are collected in Table I. More details about the architecture can be found in [7].

III. W-CDMA and OFDM timing synchronisation

Considering the signal processing chain for wireless standard's physical layer it is possible to identify two different communication techniques that cover most of the wireless

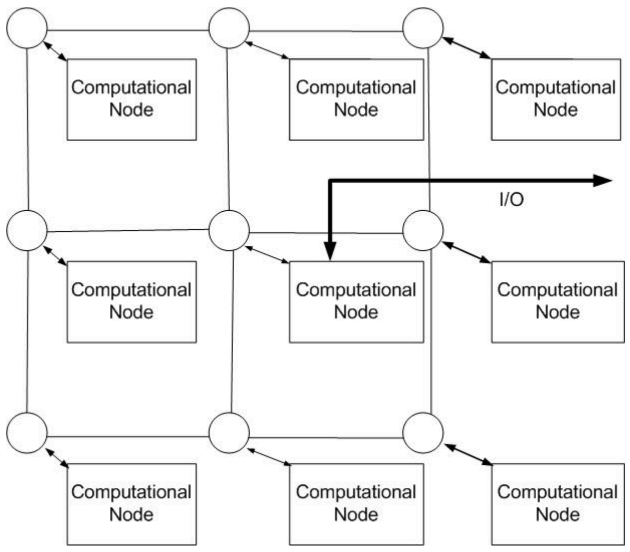


Fig. 1. Schematic view of Ninesilica MPSoC

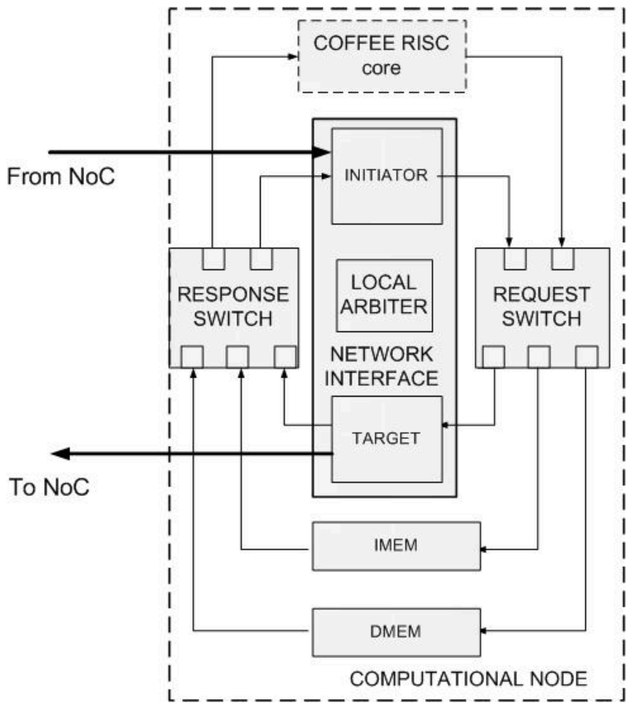


Fig. 2. Detailed view of a Ninesilica node

signal processing for today and next generation communication. W-CDMA [8] is utilised as physical layer for UMTS systems while OFDM [9] is for example utilised in Wi-Fi (IEEE 802.11a/g/n), Wi-Max (IEEE 802.16) and 3GPP-LTE.

A very critical step in the signal processing for wireless systems is the timing synchronisation. Timing synchronisation procedure takes care of synchronising the mobile terminal to the baseband station that offers the best available down-link. Hence errors in the timing would lead into an incorrect demodulation of the incoming data. Moreover this procedure is highly computational demanding and should be performed continuously to keep the synchronisation. Independently from the radio communication protocol utilised, the timing synchronisation step is based on the evaluation of correlation sequences.

TABLE I
STRATIX IV SYNTHESIS RESULTS OF NINESILICA MPSOC

Component	Adapt. LUT	Registers	Utilisation %
COFFEE RISC	7054	4941	2.0
Local network node	296	226	0.1
Computational Node	7360	5167	2.1
Global Network	5104	4170	1.3
Total	71679	50897	20

A. W-CDMA Timing Synchronisation mapping on Ninesilica

W-CDMA transmissions are organised in frames. Each frame is composed of 15 different slots. To ease the synchronisation procedure between cell and mobile terminal W-CDMA protocol utilises 3 separate channels: Primary Synchronisation Channel (P-SCH), Secondary Synchronisation Channel (S-SCH) and Common Pilot Channel (CPICH). However for the timing synchronisation only 1 channel is utilised. As figure 3 shows, P-SCH transmits the same sequence of 256 sample for each slot. This sequence is common to all the transmitting cells and it is known at the receiver. The slot boundary identification is then done through a match filter. The match filters correlate the incoming data stream to the known sequence. Output values in the sequence that exceed a pre-fixed threshold indicates a match in the slot search. Once the position of a slot is known it is possible to perform a multi-path detection for a better accuracy in the synchronisation. The match filter computes a sum of 256 complex multiplications. This operation is performed on Ninesilica in a distributed way. Each computational node computes a part of the sum returning the partial results to the control node. Hence the control node performs the final sum and check if the threshold was exceeded. In that case the multi-path correction can be performed, otherwise a new evaluation of the match filter is done.

The multi-path estimation and correction is based on the observation of four consecutive slots. The incoming data stream is correlated to the known sequence over a multi-path window (1024 samples). The correlation sequences are then averaged and the maximum value of the sequence indexes the multi-path that offers the best signal-noise ratio. The data of the four slots are equally divided among the computational nodes. Each node working on independent sets of data returns the correlation values to the central node which performs the average and the maximum search to identify the best path.

B. OFDM Timing Synchronisation mapping on Ninesilica

OFDM timing synchronisation for IEEE 802.11a is obtained through a delay and correlate approach. Each transmission

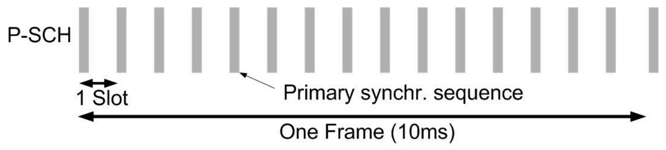


Fig. 3. Data structure of the P-SCH over 1 frame period

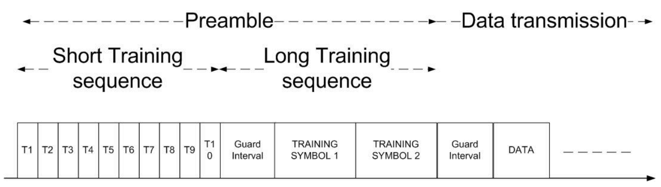


Fig. 4. Data structure of a IEEE 802.11a OFDM preamble

begin with a preamble. This preamble is utilised by the receiver to synchronise to the transmitter. Figure 4 shows its structure. The first part of the preamble is the short training sequence. This sequence is the continuous repetition of a short sequence. For this reason the course grain timing synchronisation can be done through a delay and correlate approach. The received data-stream is correlated to a delayed version of itself. The delay introduced is equal to the length of the short sequence. Peaks in the correlation sequence identifies the beginning of a communication. Ninesilica performs this operation in a distribute way. The central node distribute a chunk of data among the computational nodes, which perform the correlation on independent sections of data returning the processed data to the control node. The control node evaluates if a communication was found by analysing the correlation values received. If not a new chunk of data is sent for a further correlation analysis. After the initial synchronisation is performed a multi-path estimation can take place.

OFDM multi-path estimation and correction is based on the correlation between the long training sequences of the communication preamble and a known sequence. This operation is performed following the same approach of the W-CDMA case. Data is sent to the computational nodes which process independently the data and return the correlation evaluations to the control node. The control node finally analyses the correlation sequence refining the timing synchronisation with the estimation of the multi-path.

IV. Results

The mapping of the timing synchronisation algorithm for OFDM and W-CDMA receivers on the Ninesilica was evaluated in terms of number of clock cycles spent to accomplish the procedure. Data for the simulation was provided by a Matlab model of WCDMA and OFDM systems. It was utilised a signal to noise ratio of $-20dB$. Moreover the Matlab model was utilised as reference for the validation of the simulation results. Furthermore the simulation results were compared to an implementation on a single processor to determine the scalability of the algorithm parallelization.

A single correlation point for W-CDMA is performed in 2,546 clock cycles on Ninesilica and 12,381 on a single COFFEE core, leading to a speed-up of 5x. For the OFDM system a single correlation point (delay and correlate approach) takes respectively 88 and 350 clock cycles on Ninesilica and COFFEE core, giving a speed-up of 4x. The execution of the whole synchronisation process for W-CDMA on Ninesilica gives a speed up of 7.5x when compared to a single COFFEE core execution. For the OFDM system the achieved speed up is 6.7x.

V. Conclusions

In this work the authors presented the timing synchronisation for W-CDMA and OFDM systems on Ninesilica. Ninesilica takes advantage of data level parallelism leading to high speed-ups if compared to a single core. Future work will explore the scalability of the system with the number of nodes.

Acknowledgement

The author gratefully acknowledges Nokia Foundation for their support.

References

[1] G. Rauwerda, P. M. Heysters, and G. J. Smit, "An OFDM Receiver Implemented on the Coarse-grain Reconfigurable Montium Processor," in *roceedings of the 9th International OFDM Workshop (InOWo'04)*, (Dresden, Germany), pp. 197–201, September 15-16 2004.

[2] J. Glossner, D. Iancu, M. Moudgill, G. Nacer, S. Jinturkar, and M. Schulte, "The sandbridge sb3011 sdr platform," in *Mobile Future, 2006 and the Symposium on Trends in Communications. SympoTIC '06. Joint IST Workshop on*, pp. ii–v, June 2006.

[3] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A high-performance dsp architecture for software-defined radio," *Micro, IEEE*, vol. 27, pp. 114–123, Jan.-Feb. 2007.

[4] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 1701–1713, Oct. 2008.

[5] J. Kylliinen, T. Ahonen, and J. Nurmi, "General-purpose embedded processor cores - the COFFEE RISC example," in *Processor Design: System-on-Chip Computing for ASICs and FPGAs* (J. Nurmi, ed.), ch. 5, pp. 83–100, Kluwer Academic Publishers / Springer Publishers, June 2007. ISBN-10: 1402055293, ISBN-13: 978-1-4020-5529-4.

[6] T. Ahonen and J. Nurmi, "Hierarchically heterogeneous network-on-chip," in *Proceedings of the 2007 International Conference on Computer as a Tool (EUROCON '07)*, pp. 2580–2586, IEEE, 9-12 September 2007. ISBN: 978-1-4244-0813-9, DOI: 10.1109/EURCON.2007.4400469.

[7] R. Airoldi, F. Garzia, and J. Nurmi, "Implementation of a 64-point FFT on a Multi-Processor System-on-Chip," in *Proceedings of the 5th International Conference on Ph.D. Research in Microelectronics & Electronics (Prime '09*, (Cork, Ireland), pp. 20–23, IEEE, July 2009.

[8] E. Dahlman, P. Beming, J. Knutsson, F. Ovesjo, M. Persson, and C. Roobol, "W-CDMA - the radio interface for future mobile multimedia communications," *Vehicular Technology, IEEE Transactions on*, vol. 47, pp. 1105–1118, November 1998.

[9] X. Wang, "OFDM and its application to 4G," in *Proc. International Conference on Wireless and Optical Communications 14th Annual WOCC 2005*, p. 69, 22–23 April 2005.

Mesh and Fat-Tree comparison for dynamically reconfigurable applications

Ludovic Devaux, Sebastien Pillement, Daniel Chillet, Didier Demigny
University of Rennes I / IRISA
6 rue de Kerampont, BP 80518, 22302 LANNION, FRANCE
Email: Ludovic.Devaux@irisa.fr

***Abstract*—Dynamic reconfiguration of FPGAs allows the dynamic management of various tasks that describe an application. This new feature permits, for optimization purpose, to place tasks on line in available regions of the FPGA. Dynamic reconfiguration of tasks leads notably to some communication problems since tasks are not present in the matrix during all computation time. This dynamicity needs to be supported by the interconnection network. In this paper, we compare the two most popular interconnection topologies which are the Mesh and the Fat-Tree. These networks are compared considering the dynamic reconfiguration paradigm also with the network performances and the architectural constraints.**

I. Introduction

Evolution of technologies permits to support complex signal processing applications. The number of tasks constituting an application grows up and starts to outnumber the available resources provided by many FPGAs. Facing this implementation constraint, FPGAs that can be reconfigured on-the-fly were proposed. Hence, at each moment, only the hardware tasks which need to be executed are configured in the FPGA fabric. Assuming that all the tasks do not have to be executed simultaneously, they are allocated and scheduled at runtime. This is the Dynamic and Partial Reconfiguration (DPR) paradigm.

The management of the DPR leads to high challenges to be effective. Thus, the interconnection architecture needs to be compliant with the dynamic implementation, and location, of the tasks. This architecture should support the constraints induced by the DPR paradigm by providing a flexible way for transferring data between every sets of logical resources that are used to define the hardware parts of the tasks. These sets can be the hardware implementation of the tasks (static or dynamic), shared elements (memory, input/output), or also hardware processors running software tasks.

In this article, we compare the two more popular interconnection networks called Meshes and Fat-Trees. Our proposed comparison is based over the adequacy between these networks and current partially and dynamically reconfigurable FPGAs. Doing so, we present which interconnection network is the most useful for implementing real life complex applications using dynamic reconfiguration. The paper is organized as follow. In section II, FPGAs supporting dynamic reconfiguration are introduced. In section III, Mesh and Fat-Tree architectures are discussed considering typical applicative requirements and FPGA's characteristics. To conclude, we indicate which of the Mesh or the Fat-Tree topology best fits present applications using the dynamic reconfiguration and current FPGA architectures.

II. Context

A. Reconfigurable architectures

The industrial products supporting dynamic reconfiguration are Atmel AT40K series [1], Altera Stratix IV series [2] and Xilinx Virtex2 pro, Virtex4, Virtex5 and most recently Virtex6 series [3], [4]. Atmel FPGAs are very limited in number of available reconfigurable resources so they do not support complex applications like signal processing [1]. Altera circuits are not reconfigurable like Atmel or Xilinx FPGAs [2]. Indeed, they currently do not support the dynamic reconfiguration for all logical resources but only for inputs/output parameters. Thus, Xilinx series are the best choice for current dynamic hardware dependent researches. Considering that Xilinx is currently the market leader in dynamically reconfigurable devices, the interconnection networks are adapted to Xilinx FPGA characteristics and especially on the Virtex5, and Virtex6 series that are the most recent ones.

Dynamic reconfiguration of Xilinx FPGAs is allowed by the PlanAhead tool [5]. PlanAhead allows to specify dynamically reconfigurable regions, so called PRRs (Partially Reconfigurable Regions). A PRR is implemented statically despite the fact that its content is dynamic. Thus, at runtime, dynamic reconfiguration takes place inside the PRRs. This is the base element of dynamic reconfiguration. In Xilinx Virtex2 pro FPGAs, the dynamic reconfiguration impacted the wall columns of resources even if only a little part of then is declared to be part of a PRR. This limitation had a great impact on the possibility to implement complex application, but it no longer exists in Virtex4, virtex5, and Virtex6 series. In these series, the dynamic reconfiguration only impacts the resources inside a PRR so that several PRR can be declared using the same columns of resources.

III. Comparison of Mesh and Fat-Tree topologies

In this section, The Mesh and Fat-Tree topologies are presented in detail. In order to compare them, hardware costs, network performances, and adequacy to be implemented in present FPGAs, are studied.

A. Used resources

A Mesh (Figure 1.(a)) is a direct network in which each switch connects a hardware task. On the contrary, a Fat-Tree (Figure 1.(b)) is an indirect network. Since the number of connected tasks is N and the number of inputs/outputs of each switch is k, then the number of switches in a Fat-Tree is calculated by

$$S_{Fat-Tree} = \frac{2N}{k}(\log_{\frac{k}{2}} N) \quad [6]$$

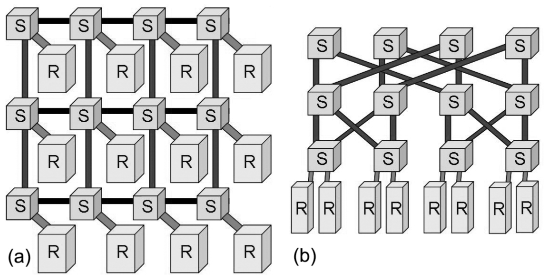


Fig. 1. Presentation of a (a) Mesh and (b) Fat-Tree topologies. *S* boxes are switch elements while *R* boxes represent a set of logical resources used to implement the tasks.

In this formula, assuming that the fat-tree is complete in terms of connected tasks, N is expressed by $N = 2^x$ where x is an integer and $x \geq 1$. When N does not match the previous formula, designers should build the network considering the admissible value of N just higher in order to keep the complete tree based structure of the network. The number of connection links needed by the Fat-Tree is calculated by

$$L_{Fat-Tree} = N(\log_{\frac{k}{2}} N) \quad [6]$$

In a Mesh, if the number of connected tasks is N whose value can be every positive integer, and if D is the radix of the Mesh, then the number of switches needed to build a regular bi-dimensional square Mesh is calculated by

$$S_{Mesh} = \left\lceil \sqrt{N} \right\rceil^2 = D^2$$

The number of needed communication links is calculated by

$$L_{Mesh} = N + 2(D^2 - D)$$

Results presented Table I can be calculated from previous formulas. Thus, a Mesh has the advantage of consuming less resources for routing purpose than a Fat-Tree. But, from these results we can see that the difference between Fat-Tree and Mesh resources scales down when the number of connected tasks is low.

TABLE I
NUMBER OF SWITCHES AND LINKS NEEDED TO IMPLEMENT A MESH AND A FAT-TREE DEPENDING ON THE NUMBER OF CONNECTED TASKS.

Connected Tasks	FAT-TREE		MESH	
	Switch	Link	Switch	Link
2	1	2	4	6
4	4	8	4	8
8	12	24	9	20
16	32	64	16	40
32	80	160	36	92
64	192	384	64	176

B. Network characteristics for dynamic operation

An application is typically constituted of some statically implemented tasks and of several dynamic tasks. In order to fit the largest scope of applications, no assumptions are made over the implementation of the tasks. So, they are implemented heterogeneously in terms of needed logical resources.

Concerning the placement, every task can be connected everywhere to the network even if it leads to the worst cases of communication. Indeed, two tasks exchanging a large amount of data (data-flow applications) can be connected to the same switch in a Fat-Tree or to two neighbor switches in a Mesh, but also to the opposite sides of the network. This concept is presented Figure 2.

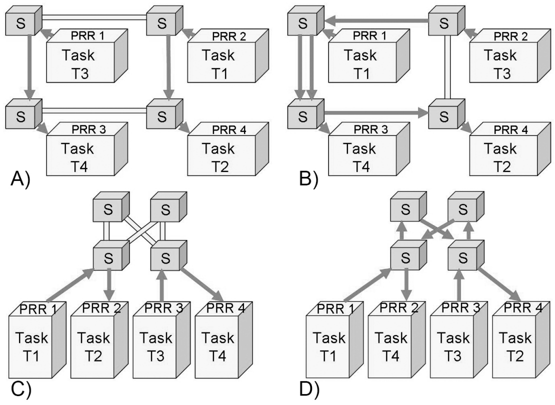


Fig. 2. Mesh (A,B) and Fat-Tree (C,D) topologies interconnecting 4 PRRs. Dynamic tasks *T1* and *T3* communicate respectively with tasks *T2* and *T4*.

From this point of view, a Fat-Tree topology has many advantages compared to a Mesh. Indeed, a Fat-Tree avoids deadlock risks that can occur in a Mesh if the control does not possess specific mechanisms to anticipate this risk [6]. The fact that the bandwidth is constant between each hierarchical level of the Fat-Tree is also a very interesting characteristic. Thus even if communicating tasks are placed at the opposite sides of the network, a Fat-Tree guaranties every data to be routed over the network without any contention because there is always at least one communication way available with a constant bandwidth. While a Mesh is a direct network, the routing can not be guaranteed like in a Fat-Tree because of the low available bandwidth compared to the number of switches [7]. Futhermore, in a Mesh, the communication requirements between tasks and shared elements induce the creation of hot-spots, increasing the likelihood of livelocks and deadlocks. One way to avoid contention risks is to use virtual channels. However, this solution has a very high cost in terms of used memories.

The two networks are simulated using ModelSim 9.5c [8] for a 32 bits data width, and with a buffer depth of 4x32 bit words. Concerning network performances, the highest data rates are chosen in order to place the two topologies into the worst functioning cases. So, for a connection of 8 tasks, their transfer rate are fixed to 800Mbit/s each. Simulations were realized sending 1562 packets of 16 words each for an injection time of 1ms (Figure 3).

From these results, with a transfer rate fixed to 800Mbit/s per task, Mesh and Fat-Tree topologies provide equal latencies until 4 simultaneously connected tasks. Then, due to a change of the network scales, a Mesh present a lower average latency until 9 connected tasks. Fat-Tree provides a just higher average latency but it can connect 12 tasks without saturating. This is a very important result because, with a uniform repartition of the data traffic, a Mesh seems interesting for interconnecting a low number of tasks. For more connected tasks, a Fat-Tree is well suited. However, These results were obtained from a simulation with a Uniform repartition of data

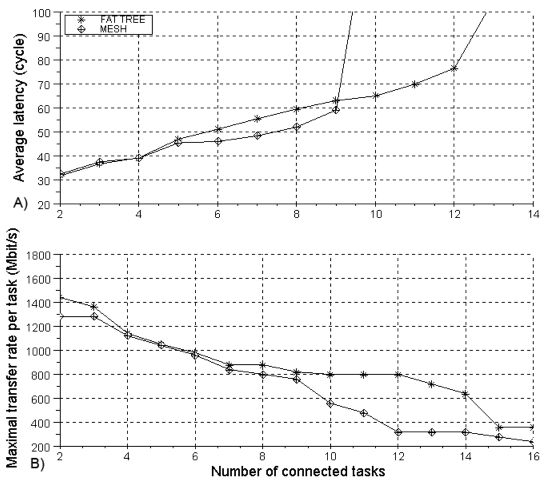


Fig. 3. (A) Comparison of the average latencies for a connection of 8 tasks, and (B) maximal admissible data rates per task depending on the number of simultaneously connected tasks.

and without any hot-spot. In usual systems using shared elements, a Mesh will present hot-spots near these elements and the resulting average latency will grow up significantly. While this problem has no influence over the Fat-Tree topology, the latter is more suited than a Mesh for an implementation in real life applications.

Considering the maximal admissible transfer rates per tasks, a Fat-Tree supports equal or higher rates than a Mesh. So, in real life applications where bandwidth is a major requirement, Fat-Trees should be implemented instead of Meshes.

C. Network implementation

The implementation of Fat-Tree and Mesh topologies in a Xilinx Virtex5 VC5VSX50T lead to the results presented in Table II. Considering the FPGA resource utilization, implementing a NoC as a central column presents many advantages. Thus, depending on the hierarchical level, the Fat-Tree can be implemented with a very limited number of resources. Therefor, the resources remain free for other tasks or for a processor implementation.

Thus, with this concept of implementation, both Mesh and Fat-Tree topologies are compliant with present technology and can be implemented with

TABLE II
NUMBER OF USED RESOURCES NEEDED TO IMPLEMENT A MESH AND A FAT-TREE DEPENDING ON THE NUMBER OF CONNECTED TASKS.

Resources	FAT-TREE		MESH	
	4 tasks	8 tasks	4 tasks	8 tasks
Registers	900	2700	524	1549
LUTs	3416	10248	1992	6083
Free registers	97%	92%	98%	95%
Free LUTs	90%	69%	94%	81%

the guarantee that their specifications (routing, latency...) are respected even if the FPGAs and the tasks are heterogeneous. However, considering its structure, the Fat-Tree is particularly suited for this concept of implementation.

IV. CONCLUSION

In this article, we have presented a comparison between the two more popular interconnection networks, the Mesh and the Fat-Tree. It appeared that a Mesh has the advantage to consume less routing resources than the Fat-Tree. However, considering present applications that do not often need much than ten simultaneously implemented dynamic tasks, and rarely much than fifteen, the difference in terms of logical resources utilization for routing purpose can be acceptable. Furthermore, if both of them are compliant with present FPGA specifications, the demonstration was made that a Fat-Tree is more adapted to the dynamic reconfiguration paradigm. It presents equal or higher network performances, a deadlock free optimal routing algorithm, and a material structure allowing to provide a constant bandwidth to every tasks everywhere into the network.

REFERENCES

[1] ATMEL, *AT40K05/10/20/40AL. 5K - 50K Gate FPGA with DSP Optimized Core Cell and Distributed FreeRam, Enhanced Performance Improvement and Bi-directional I/Os (3.3 V).*, 2006, revision F.
[2] Altera, *Stratix IV Device Handbook - Volume 1, ver 4.0, Nov 2009.*, 11 2009.
[3] Xilinx, *Virtex-5 FPGA Configuration User Guide*, 2008, v3.5.
[4] ——, *XST User Guide for Virtex-6 and Spartan-6 Devices*, December 2, 2009, uG687 (v 11.4).
[5] ——, *PlanAhead User Guide - version 1.1*, 2008.
[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2006, ch. Appendix E : Interconnection Networks.
[7] V.-D. Ngo and H.-W. Choi, "Analyzing the performance of mesh and fat-tree topologies for network on chip design," *Computeur Science*, vol. 3824, pp. 300–310, 2005.
[8] M. graphics, *ModelSim LE/PE Users Manual 6.5.c*, 2009.

Technology Independent, Embedded Logic Cores

Utilizing synthesizable embedded FPGA-cores for ASIC design validation

Joachim Knäblein, Claudia Tischendorf, Erik Markert, Ulrich Heinkel
Chair Circuit and System Design
Chemnitz University of Technology
Chemnitz, Germany

Abstract—**This article describes an approach to embed technology independent, synthesizable FPGA-like cores into ASIC designs. The motivation for this concept is to combine the best aspects of the two chip design domains ASIC and FPGA. ASICs have better timing performance, are cheaper in mass production and less power consumptive. FPGAs have the big advantage to be reconfigurable. With FPGA-like cores being embedded into ASICs this extraordinary FPGA feature is transferred to the ASIC domain. The main innovative aspect of the approach proposed in this paper is not the concept of combining ASIC and FPGA on one die. This has already been done before. The novelty is to find ways to use standard components and cells for the FPGA part to be able to enhance ASIC designs without being restricted by technological and vendor related barriers.**
Among many other applications reconfigurability can be leveraged to improve verification problems, which arise with today's 100 million gate designs. Dedicated, synthesized PSL [23] monitors, which are loaded in embedded FPGA cores, accelerate the process of narrowing error locations on the chip.

Keywords-component; ASIC; FPGA; synthesis; PSL;

I. Introduction

Whether to use FPGAs and/or ASICs in the design of a new system is a fundamental question. Both chip domains have their benefits and drawbacks, which must be considered carefully. ASICs have better timing performance, better scalability, more options, are cheaper in mass production and less power consumptive. On the other hand FPGA based designs offer faster design cycles, better debugging and bug fixing capabilities, less costs for small lots and reconfiguration capabilities. Depending on numerous parameters, the one or the other concept is more suitable for a particular design job.
An obvious question is why not to combine the best of both worlds and embed a FPGA-like core into an ASIC design? This paper deals with technical aspects of the implementation of this idea based on components and cells, which are available in (almost) all technologies and from all manufacturers. In the course of this paper such an embedded, independent FPGA core shall be denoted as EPLA (Embedded Programmable Logic Array).
The paper is organized as follows: First related work is listed and the advantages of synthesizable FPGA cores are discussed. Then appropriate structures of logic elements and interconnection concepts are investigated. Finally a special application of such embedded cores is presented.

II. Related Work

The idea of embedding an EPLA core into ASICs is not new. Several companies (Abound Logic currently [1], Adaptive Silicon in 2002 [2]) attempted to introduce such a concept in the market. A similar approach was implemented by a project funded by the European Union [3]. And there are even more projects pursuing this idea (e.g. GARP [24]). Nevertheless it seems, that it is difficult to be commercially successful with the technology dependent, synthesizable core idea, although there is a certain demand in the industry for such a concept: Adaptive Silicon disappeared after the year 2002 from the market and Abound Logic moved their scope away from the embedded core concept to their own family of stand-alone FPGA devices. Two of the reasons for this lacking market acceptance might be:

- **Technology dependency**
  In order to achieve a good ratio between physical silicon area consumption and implemented logic, the cores are typically optimized on transistor level. As a consequence it takes a lot of effort and time to be able to offer them for a particular technology and a particular manufacturer. In a time with fast innovation cycles in chip design it is not possible to supply solutions for all technologies of all manufacturers. This limits the manufacturer selection range of potential EPLA customers very much and thus, those customers may tend to implement without an EPLA instead of being forced to go with a certain technology/manufacturer.
- **Cost overhead**
  As described above, the individual design of EPLA cores for particular technologies is a very complex job and customers of course have to pay for this. On the other side the customer must justify the use of an embedded core in his application from the commercial point of view. If these aspects do not match, it makes no sense to embed an EPLA in the design.

Consequently, the aspects manufacturer independency and cost effectiveness are vital for dissemination of EPLA approaches in the chip design industry. This paper introduces a concept for EPLA cores, which takes that into account.

III. Manufacturer Independent FPGA Cores

This chapter discusses how an EPLA can be realized in a manufacturer independent way. To be independent means:

- The EPLA must be built from cells, which are available in (almost) every technology library, i.e. only standard logic cells and RAM blocks are permitted
- The EPLA design must fit in a standard ASIC design flow
- The core is limited to ASIC capabilities in size and timing performance

Typically FPGA architectures comprise logic element arrays, which are variably interconnected. Such a logic element (LE) consists of one or more look-up-tables (LUTs), which realize a part of the logic function of the design to be mapped onto the FPGA. In addition, the LE contains one or more flip-flops in order to allow sequential behavior. The basic principles were patented by Xilinx co-founder Ross Freeman in 1988 [4]. This patent however expired in 2004 and this paved the way to use such principles for a wide range of applications.

A. EPLA Structure

The challenge is to find a LE architecture and an interconnection scheme, which fulfills the above independency requirements and is still efficient with respect to the physical area consumption and timing performance of the core. Other than expected, not the LE design is the most challenging task when developing EPLA architectures, but it is the interconnection scheme. This is the case, because the number of possible interconnections of n LEs is $O(n^2)$ [5]. Simplified example: for a core which comprises 1000 4-input LEs, the number of possible interconnections (crossbar) is 4 million. Since every interconnection must be switchable in the general case, this results in a reconfiguration storage requirement of 4 million storage elements only for the interconnection definition. On the other side typically only a very small amount of these possible interconnection are really needed and thus such a trivial interconnection approach would be a terrible waste of silicon resources. Therefore this full interconnection scheme must be reduced by two measures:

- Interconnection clusters must be defined, which have full internal interconnection capabilities. The connections within the clusters must be efficient with respect to timing and area consumption. The interconnection in-between the clusters must be reduced as far as possible, because these are limited in number and time and area consuming.
- The input count of the LEs must be chosen as high as possible. The reason for this is that the more fine grained an architecture is, the more logic cells are needed to form a given logic function. The more logic cells must be interconnected, the more wires are needed for this job and wiring must be minimized, because it is costly. Another driver for this large input count is the so called "group optimization", which is described in section C.

The counter problem of high LE input count, however, is that the gate effectiveness (the ratio between logic gate count and physical gate count) of a LUT decreases with the input count of the LUT. The reason for this is that a LUT with k inputs can realize 2^{2^k} different boolean functions as can be deduced from a truth table representation. Consider a LE structure like shown in Figure 1.

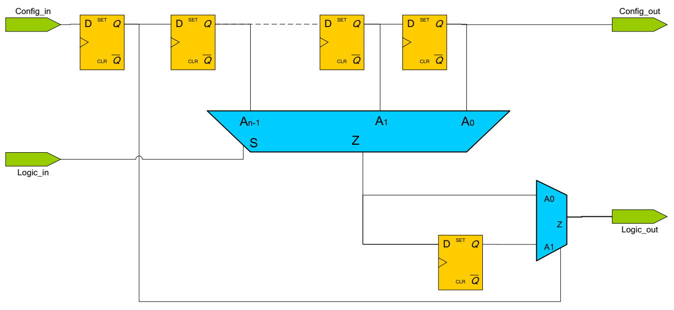


Figure 1: Logic structure of a logic element (LE)

In order to decode this number of boolean functions 2^k storage elements are needed. On the other side in [6] an upper bound for the gate count of a k-input logic cone is derived to be $O(2^k/k)$. Thus the maximum gate effectiveness of a LUT is

$$O\left(\frac{2^k}{k} * 2^{-k}\right) = O\left(\frac{1}{k}\right)$$

In other words: the maximum gate effectiveness decreases with the input count of the LUT on a reciprocal basis. Moreover, this optimum effectiveness can only be achieved, if an LE is packed with logic. In typical designs the distribution of input logic cones may vary from 2 inputs to more than 1000 inputs. If the EPLA architecture only offers LE with a huge number of inputs, a lot of logic resources are wasted even if such a large LE is able to realize a particular number of smaller input cones. The following example illustrates this aspect:

Example:
Given is an 8-input LE with 4 outputs. This LE can realize logic in the range of one boolean function with 8 independent variables or four functions with two variables. The maximum effectiveness of the first case is a factor of $2^8/8*2^{-8}=1/8$ whereas the effectiveness of the second case is a factor of.$4*2^2/2*2^{-8}=1/32$.
At this point we have gathered a number of parameters, which play an important role in the design of an technology independent EPLA. These parameters are:

- Number of LEs in the core
- Input count of the LEs
- Number of outputs of the LEs
- Size of cluster with full interconnection capability

B. Logic Element Design

So far we have dealt in this paper with considerations on how LEs should be structured and connected in the EPLA. The next thing is to reason about the internals of a logic element with n inputs and m outputs. The main building block of an LE is the LUT that realizes up to m logic functions of n input variables. For the implementation of this LUT a RAM is used which holds m output bits for each of the 2^n input combinations. This is shown in Figure 2.

Depending on the technology of the manufacturer an efficient realization with a RAM block may only be reasonable for greater values of n and m. If n is less than a particular, technology dependent value, the realization of the LUT might be more efficient when based on flip-flops or latches like shown in Figure 1. The area consumption of the RAM based solution can be optimized if multi-read-ports are available in a technology. Then the LUTs of several LEs can share one RAM block like shown in Figure 3. Such shared RAM blocks consume less area than the sum of individual RAM blocks would do.

The rest of the components of an LE is straightforward. Every output of the LUT feeds a flip-flop that can be optionally bypassed. Each configuration capability in a LE like this bypass multiplexer control needs one or more storage elements. In order to set those storage elements in a simple but efficient way, they are connected to form a long shift register like known from the scan chain approach. Configuring the EPLA with a particular circuit means to load the core with specific data by using this chain and by loading the RAM based LUTs.

C. Interconnection Scheme

Several of the LEs described above are combined to form a LE cluster. In addition, such a cluster comprises an interconnection block, which connects LE outputs to other LE inputs inside this cluster (see Figure 4 for details). The interconnection block must meet several requirements. The terminology for discussion of such requirements in the following is borrowed from switching theory [19].

1. Allow arbitrary permutations of inputs N at M outputs with $N \geq M$. Such a network is called “non-blocking”. In [19] a distinction is used between “rearrangeably non-blocking” and “strictly non-blocking”. This distinction is only of interest for the case of dynamical switching, which does not apply to our problem.
2. Allow connection of one input to more than one output. In the following text this feature is called “multicast”.
3. The network should consume as low gate count as possible
4. The network should have as low propagation delay as possible
5. Groups of outputs are defined. Within such groups the order of input permutations is “don’t care”, i.e. it only matters that a particular input route appears inside the output group. Its position inside the group is irrelevant. This aspect allows for particular optimizations. The reason why this optimization is possible, is that in the application of the interconnection block the order of inputs for e.g. a specific target LE is not significant, because the LUT input configuration can be chosen freely.

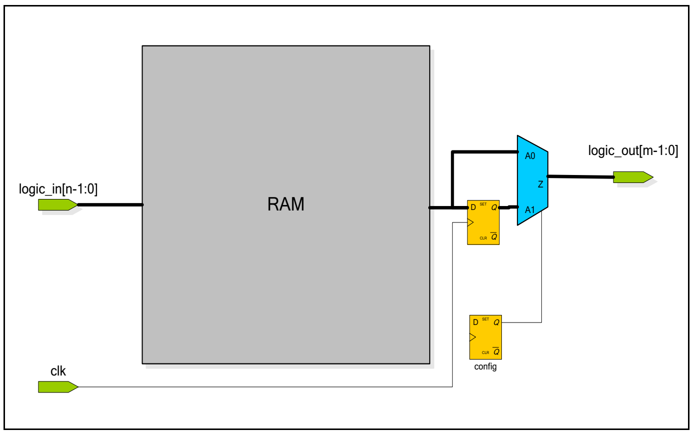


Figure 2: RAM based logic element

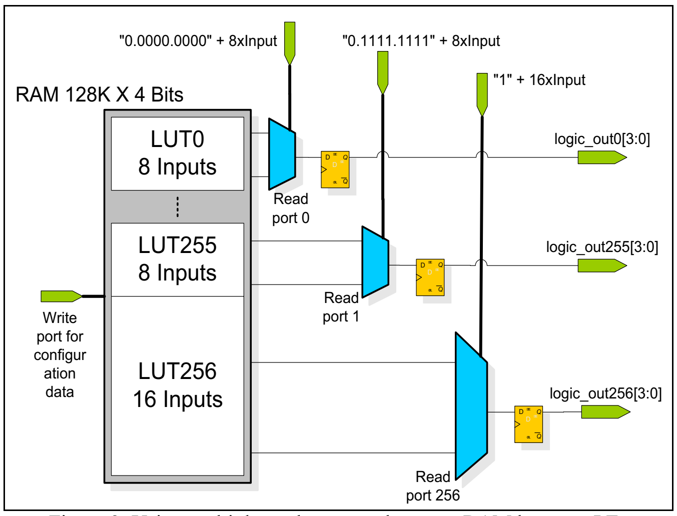


Figure 3: Using multiple read ports to share one RAM between LEs

Several interconnection block concepts have been investigated:

- Multiplexer based interconnection concepts
- The multicast Clos network [15][16][17]
- The Beneš network [18][19] and optimizations [20] with multiplexer based multicast extension
- The reverse Omega network [19] used in a Fat Tree architecture

Since the reverse Omega network has some interesting properties, this particular interconnection approach is discussed here deeper.

The structure of a reverse Omega network is shown in Figure 5. On the positive side the reverse Omega network offers permutation capability at low gate and delay cost compared to all the other concepts. The number of 2:1 multiplexers in the structure for $M = N$ is $\frac{N}{2}\log_2 N$ and the delay in units of 2:1 multiplexer delays is $\log_2 N$.

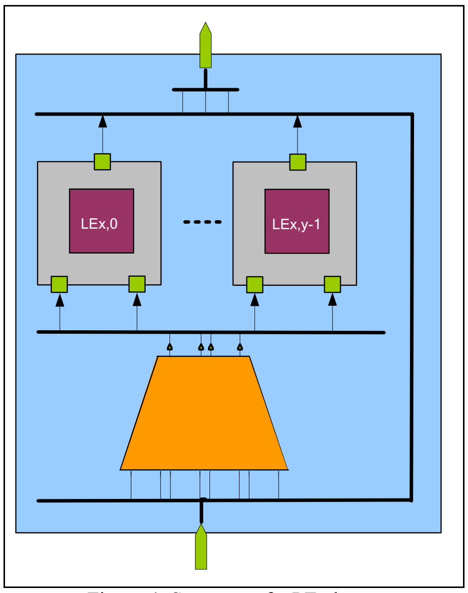


Figure 4: Structure of a LE cluster

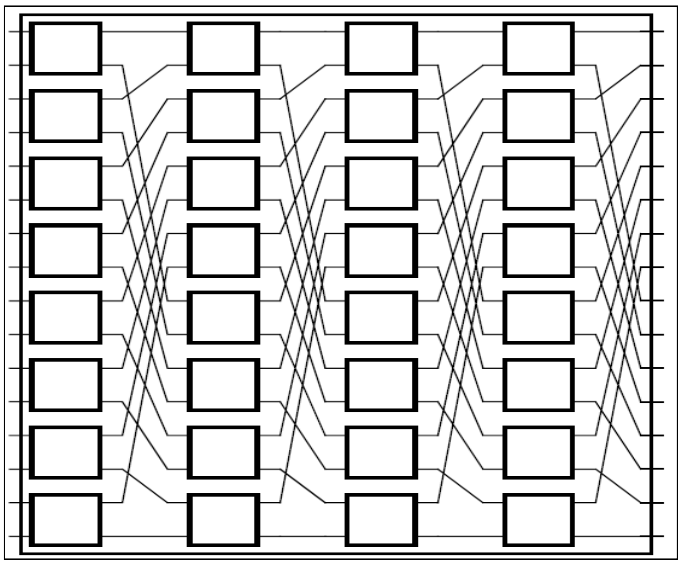


Figure 5: A 16 x 16 Reverse Omega network

On the negative side the Omega network is only non-blocking under very strict conditions and it does not support multicast. In [19], p. 114 it is shown, that an Omega network is only non-blocking, if the input-to-output assignment is sorted in a so called cyclic compact monotone sequence (CCM). This aspect restricts the routing ability of the Omega network very much, but can be overcome with the approach described in the following.
Although the Omega network is used normally in its symmetric form, i.e. $M = N$, it is possible to reduce it to asymmetric use, i.e. $N \geq M$. The simplest way to achieve this is to set $N - M$ output ports to unconnected and let the synthesis process do the rest. Remarkably, the ports, which are set to unconnected, must be carefully selected to achieve the best reduction result. It can be shown, that for M and N being a power of 2, the multiplexer count of a N:M reverse Omega Network is approximately:

$$c_m = \begin{cases} N * \log_2 N - (N - M), & M \geq \frac{N}{2} \\ (N-1) * \log_2 M + (N-1), & M < \frac{N}{2} \end{cases}$$

D. *Combining Fat Tree and Omega Network (FTON)*

The Fat Tree concept is described in [21]. A binary Fat Tree consists of several stages splitting input-to-output connections into two branches in every stage. The splitting is done here using reverse Omega networks. The reverse variant is chosen, because our application requires, that the output and not the inputs are CCM. The idea is illustrated in Figure 6. Due to the output group optimization aspect, the routing within the Omega networks can be chosen in a way that blocking does not take place. The gate count behavior of the fat tree Omega network is shown in Figure 7 for $M = N/4,\ G = M/8$ on logarithmic scale.
Due to the multi stage structure the delay behavior of the *binary* Fat Tree Omega network is bad compared to the other interconnection realizations, but the binary tree can be generalized towards a r-Tree meaning not two, but r branches are used in each stage. Then, for constant r over the stages, $log_r G$ stages are needed, resulting in a delay of:

$$\begin{aligned} d &= log_2 N + log_2 M/r + log_2 M/r^2 + log_2 M/r^3 + \ldots = \\ &log_2 N + log_2 M + log_2 M + \ldots - log_2 r(1+2+\ldots) = \\ &log_2 N + (log_r G-1) * (log_2 M - ½ * log_2 r * log_r G) \end{aligned}$$

This equation is valid for $2 \leq r \leq G$. For $r = G$ the entire circuit is reduced to a single stage with an individual network for every output group.
The delay behavior is shown in Figure 8.

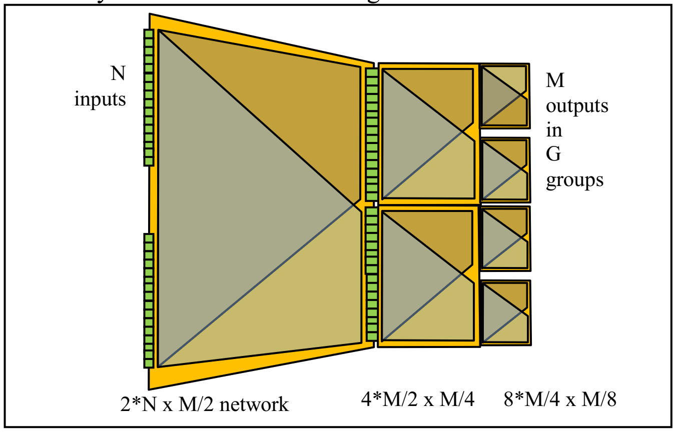


Figure 6: Fat Tree realized with Omega networks for $r = 2$

When inspecting the diagrams it is obvious that there is a competition between gate count and delay. The less the gate count, the higher the delay. The best interconnection concepts are the Benes Network and the FTON. The Benes based realization makes the interconnection block with the least gate count, but the highest delay. With the FTON, gate count can be traded for reduced delay in a wide range by variation of parameter r.

E. Architecture of the EPLA

With all the considerations we now define an architecture for the EPLA. The interconnection concept has three layers as shown in Figure 9. The first layer deals with inter-cluster connections. Since this is the most powerful interconnection layer the software flow described in section F takes care that strongly connected components (i.e. LEs) are concentrated inside clusters. The next interconnection layer connects adjacent LE clusters. Again, the fitting software takes care to allocate strongly connected clusters in adjacent blocks.

The rest of the connections is wired by the last interconnection layer, the global switch.

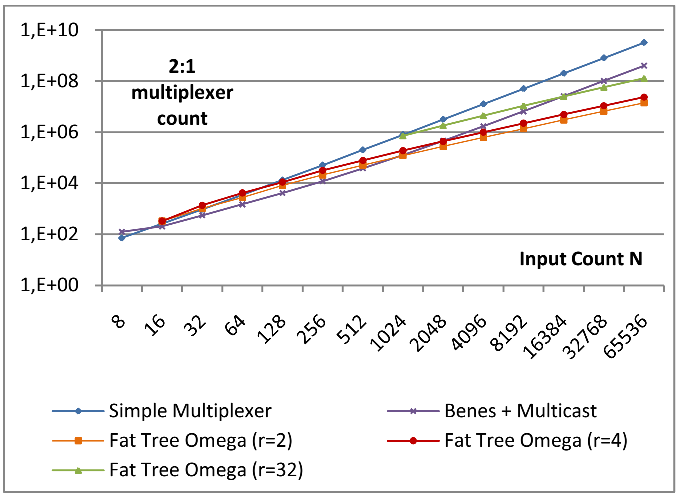


Figure 7: Gate consumption of the Fat Tree Omega network compared to other listed concepts

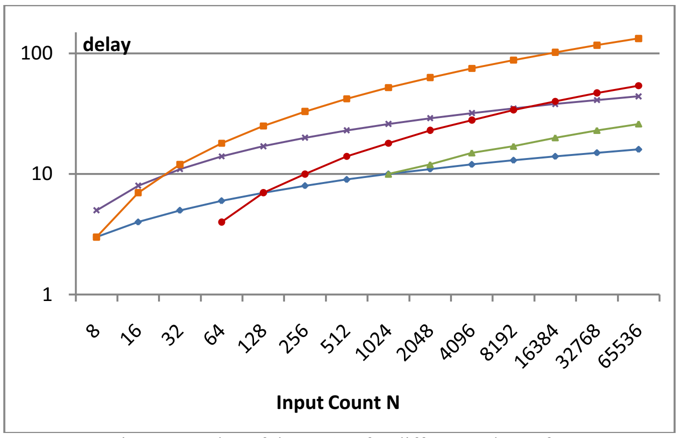


Figure 8: Delay of the FTON for different values of *r* *(same color coding like above)*

F. Tool Flow

In the previous section the hardware architecture of the FPGA core was discussed. This chapter introduces some aspects of the mapping tool flow, i.e. the software flow, which is applied to convert a VHDL model to an EPLA load. This flow is depicted in Figure 10.

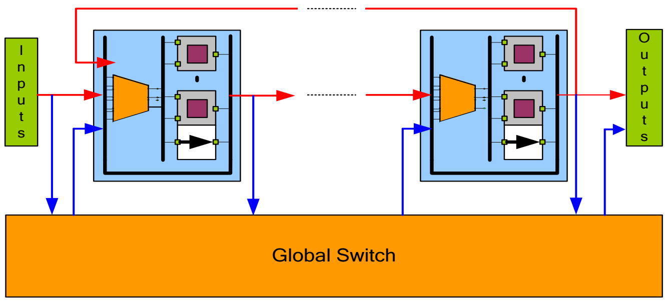
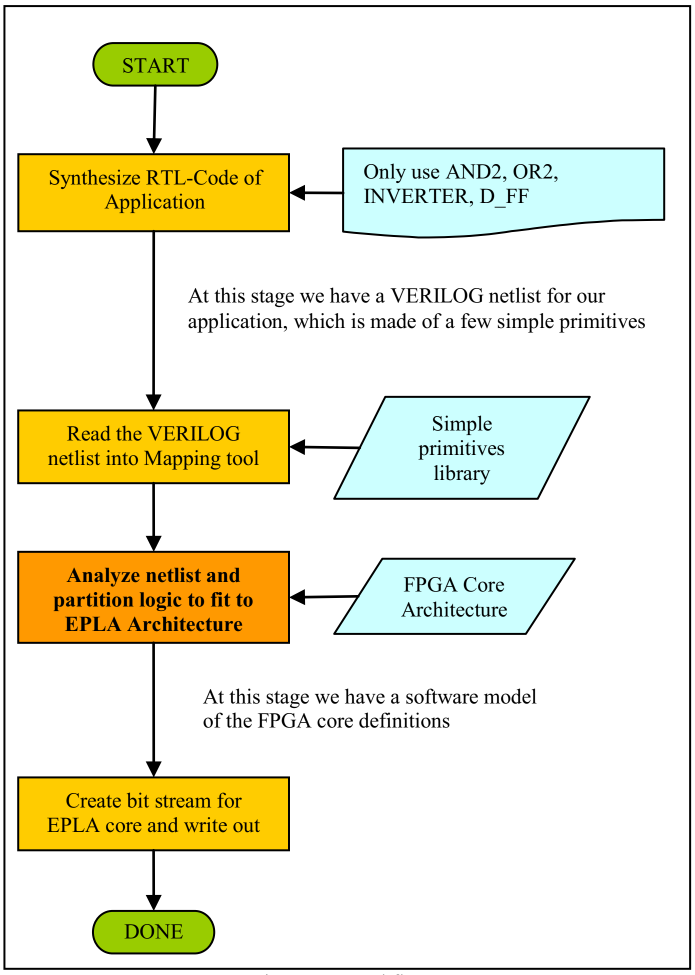


Figure 9: Architecture of the EPLA



START
Synthesize RTL-Code of Application
Only use AND2, OR2, INVERTER, D_FF
At this stage we have a VERILOG netlist for our application, which is made of a few simple primitives
Read the VERILOG netlist into Mapping tool
Simple primitives library
Analyze netlist and partition logic to fit to EPLA Architecture
FPGA Core Architecture
At this stage we have a software model of the FPGA core definitions
Create bit stream for EPLA core and write out
DONE

Figure 10: Tool flow

A first synthesis of the VHDL model is done with a commercial tool and the resulting netlist is created in verilog. During the synthesis step the compiler is restricted to a set of simple cells (e.g. AND, OR, NOT, flip-flop) to simplify the following processing and reduce the primitive library. In the next step the verilog netlist is read into a tool, which does the fitting of the netlist to the architecture as described below.

G. *Decomposing the Logic Cones*

The logic cones of the first synthesis step are not appropriate to be fit into a k-input LUT. Logic cones, which have more than k input variables must be split into pieces with k inputs at maximum. This splitting process is called decomposition and can be done based on BDD (binary decision diagram) [7] operations. The decomposition topic has been covered in academic circles very deeply and therefore several implementations are freely available. For this job the package BDS-pga 2.0 [8] has been chosen for the decomposition step. This package, however, does neither support verilog input nor allows sequential elements in the netlist. Therefore a new tool has been developed, which only re-uses the decomposing algorithms of BDS-pga. The result of this decomposition step is a data structure, which consists of a tree of AND, OR, XOR, XNOR, MUX operators applied to the input signals of a k-ary chunk of logic.

H. *Placement*

Placing is the process, which assigns particular logic cone pieces to a dedicated EPLA LE. Starting with the data structure of the decomposing step, chunks of logic are searched, which fit into the current LE allocation. Metrics for this search are:

- How well does a chunk fill the potentially partly used LE
- How many inputs can be reused when allocating the chunk to the LE

These criteria are evaluated for every LE, which is not yet full. The LE, which has the best ranking, is chosen to house the logic cone piece under investigation. The result of this process is a list of (partly) used LEs, which carry the logic of the netlist.

I. *Routing*

Routing is the process of defining interconnections between LEs. Starting with the above list of logic pieces, an interconnection table is created, which tells, which LE output is connected to which LE inputs. This interconnection table can be regarded as a graph and thus graph theoretical considerations can be applied. Remember, that such LEs must be packed to clusters, which have tight interconnection. The inter-cluster-connections shall be reduced to a minimum, because such connections are expensive with respect to EPLA resources. This can be achieved by applying a graph partitioning algorithm to the interconnection table. There are many graph partitioning algorithms with the Kernighan/Lin [9] being the most famous. Unfortunately, this algorithm is $O(n^3)$ and therefore not suitable for the size of our problem. However, alternative algorithms like the Fiduccia/Mattheyses algorithm [22] have been developed to overcome this deficiency. The original version of this algorithm only supports splitting into two partitions. Therefore there was a need to generalize this algorithm to multi-partitions. As a result of this partitioning step LE clusters of almost equal LE counts are found, which have minimized inter-cluster connections.

J. *Creating the Load File for the FPGA*

Based on the results of the decomposition and the place & route steps, the load file is generated by the tool. The LUT logic is created as a truth table to be loaded into the LUT RAMs. The single storage elements of the bypass multiplexer and the interconnection networks form a long shift register. The configuration data for these (e.g. bypass, interconnection configuration) are created as a bit stream file to be loaded into the storage element chain.

IV. Utilizing the EPLA

The EPLA concept described in this paper can be utilized for several purposes:

- **Updating of ASIC functionality e.g. because implemented standards have changed**
  The ASIC can be manufactured before standards are settled finally. Once the standard definition phase is finished the ASIC can be adapted to the new version by updating the built-in EPLA circuit.
- **Fixing of ASIC bugs**
  In case a bug is present in the ASIC functionality it might be fixed by loading an error correction configuration in the EPLA. An architecture, which supports such a kind of application, is shown in Figure 11.
- **Saving additional accompanying FPGA devices on the board**
  Sometimes functionality, which was defined after the manufacturing of the ASIC, is implemented in additional FPGA devices on the board. Such FPGA device can be saved, if the ASIC itself contains such an option.
- **Deployment of the same ASIC in different hardware environments**
  Depending on the application it may make sense to develop a universal ASIC for multiple purposes, which incorporates an EPLA. The adaptation of the ASIC for a concrete function is done by loading the EPLA with a particular configuration. This approach, however, makes only sense, if the number of different configurations exceeds the number of 50 or the configurations are not known at design time. The reason for this is that the EPLA consumes a lot more ASIC silicon than the function, which can be realized in the EPLA.
- **Using the EPLA as a bed for variable circuit monitors**
  This approach is extraordinary in particular and therefore this concept is described in detail in the next section.

Figure 11 illustrates the test and modification point architecture for bug fixing and monitoring purposes. This architecture gives the chance to insert a module at anticipated ASIC path locations. If no monitor and no bug fixing is required, the EPLA is just a pass through for the paths. If a monitor is required e.g. to locate a bug location, the EPLA configuration is modified with the insertion of such a circuit (shown in red) at a particular path location.

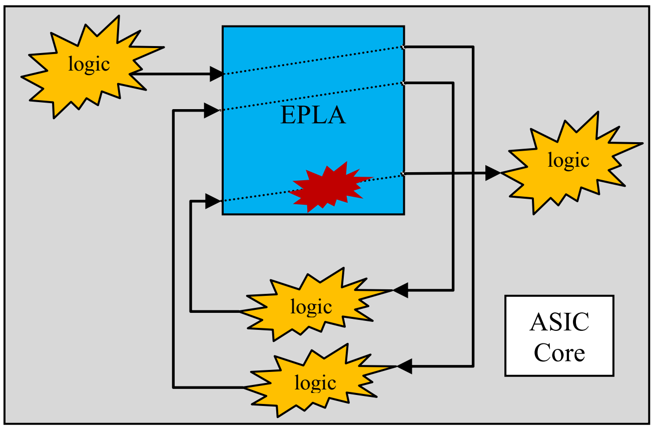


Figure 11: Architecture for variable test and modification point insertion

A. *Utilizing Cores as a Bed for Runtime Monitors*

In the previous chapter a method for the embedding of technology independent, synthesizable FPGA cores in an ASIC was introduced. The advantage of this method is that all kind of circuits can be implemented after the ASIC was manufactured. This also includes modules, which support the validation engineer during his work. A problem during this phase in general is the observabililty of the design. This lacking observability is the main reason, why identifying the location of a design error on hardware level is a time consuming job. And thus it might be useful to be able to integrate monitors in the design at runtime. Such monitors do concurrent checking of the behavior of the design and flag errors if something suspicious happens. Monitors could be embedded in the hardcoded part of an ASIC. Then they only are of limited benefit, because their scope is very restricted. Here the EPLA technology can be helpful. A user-defined monitor is loaded into the FPGA core as needed. Such monitors can be derived from pre-existing PSL [12][23] assertions. This synthesis step is described in the following section.

B. *Automata Construction*

PSL comes with distinct flavors for each different implementation language. In this work we concentrate on the VHDL flavor, but the algorithm can also be applied to all other languages. PSL is divided into four layers of abstraction. The boolean layer contains only the expressions of the underlying flavor, such as the VHDL operators *and*, *or* and *not*. All temporal aspects are contained in the next layer, the temporal layer. It supports expressions like *always* and *until*. The verification layer and modeling layer describe how the property must be used and they specify aspects of the verification environment. To generate a checker automaton it is only necessary to consider the boolean and the temporal layer. In fact, the boolean layer is transparent to the generation algorithm, since the expressions in this layer can be virtually substituted by simple boolean variables. Thus, only the operators of the temporal layer will influence the generation algorithm. The IEEE PSL standard [13] supports three sets of operators in the temporal layer. Sequential Extended Regular Expressions (SERE) and the usual LTL operator's represent the Foundation Language (FL). Additionally, the Optional Branching Extensions (OBE) provides a CTL-like set of operators. From the multitude of PSL operators, the standard defines a subset of only a few simple operators given in Table 1 that can be used to express all other more complex operators.

Table 1: Simple Subset

FL	SERE
b	r
s	r
(p)	$r_1 ; r_2$
$s!$	$r_1 : r_2$
$!b$	$r_1 \mid r_2$

The mapping of the subset to the operators is defined by so-called rewrite rules. Each PSL expression is recursively constructed from its basic parts. For example, to construct the automaton for the expression *always{a; b},* first the automaton for *{a; b}* is constructed and then used by the algorithm for the always operator. The result of each step of the construction algorithm is a nondeterministic finite automaton with epsilon transitions (ϵ-NFA). ϵ-transitions are executed without any input. First, all the ϵ-transitions are eliminated. The result still contains non-deterministic transitions. In order to generate synthesizable VHDL code it has to be converted into a deterministic finite automaton (DFA). Listing 1 describes the algorithm, which transform an NDFA into a DFA.

Listing 1: NDFA → DFA

```
A → generate automat (DFA)
$A_1$ given automat
for all states $Z$ in $A_1$ do
        find all following states of $Z$
        combine all transition conditions
        create new transitions from $Z$ to combinations of the following
states
set start state
```

In the first step all outgoing transitions from the start state are determined, which do not contain ϵ-transitions. Afterwards all combinations of these transitions conditions and all following states are generated. These new states and transitions were added to the new deterministic automaton. This process will be performed for all states of the non-deterministic automaton. During this algorithm different final states are generated. One or more final states are definitely correct. Additionally there is one more final state to signal the first failure of the expression. As long as no final state is reached, the result of the expression is still pending. The main state of the automaton is the final error state, because the designer is only interested in the point in time when the property actually fails.
The generation process of a specification is supported by the specification platform "SpecScribe" [14]. This tool offers the possibility to define requirements, components and

dependencies in a hierarchical way. One of these requirements can be defined as ”verification property”. The PSL expression is added. After that, a finite state machine will be generated. To simulate a monitoring automaton or implement it into a design, different representations can be generated automatically, e.g. synthesizable VHDL or SystemC.

C. Results

Currently the EPLA fitting tool is in experimental status. In comparison to commercial tools, which reach an estimated ratio between physical gate count to logic gate count of 20-50, the achieved factor of > 50 is still subject to further investigations.

The PSL approach has been verified so far on a Xilinx FPGA board with a Virtex-II Pro. The board was programmed with a design, which receives packages and echoes them after being processed. A finite state machine was integrated into this design. The final error state is indicated by an LED, whenever the PSL expression failed. The structure of the board is shown in Figure 12.

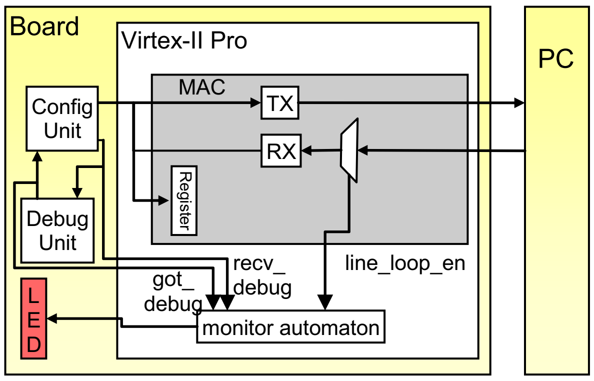


Figure 12: Structure of demo board

V. Conclusion and Outlook

This paper presented an approach for the synthesis of technology independent FPGA cores (EPLA) as part of SoC ASIC designs. Such cores can be used for a wide spectrum of applications.

As an example an interesting concept was presented, which supports the test engineer in his work. In the case of functional deficiencies dedicated monitors are loaded on the technology independent, synthesizable EPLA core in order to narrow the specific location of a design error. For this purpose assertions written in PSL are synthesized and loaded into the EPLA.

The status of the described scenario is currently experimental. Therefore the fitting results of a given RTL code to an EPLA with respect to area and timing optimizations are still a matter of further investigations.

Another topic under investigation is the ambition to reconfigure the EPLA as fast as possible in order to reduce the reconfiguration impact to a minimum.

It is planned to automatically introduce an error reporting infrastructure into the design that connects all monitor automata, collects the results and reports the corresponding data either to a processor within or to an entity outside the chip.

VI. References

[1] Homepage of FPGA Manufacturer Abound Logic: http://www.aboundlogic.com/
[2] Design & Reuse Article: Adaptive silicon’s MSA 2500 programmable logic core TSMC test chips are fully functional. June 6, 2001
[3] Voros, N., Rosti, A., Hübner, M., eds.: Dynamic System Reconfiguration in Heterogeneous Platforms - The MORPHEUS Approach. Springer (2009)
[4] Freeman, R.H.: Configurable electrical circuit having configurable logic elements and Configurable Interconnects. (Feb 19, 1988) Google Patent Repository.
[5] MathWorld, W.: Introduction to big-o-notation. http://mathworld.wolfram.com/AsymptoticNotation.html
[6] Erickson, J.: Cs 497: Concrete models of computation. http://compgeom.cs.uiuc.edu/jeffe/teaching/497/13-circuits.pdf (Spring 2003)
[7] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transaction (1986) C-35(8):677-691.
[8] Homepage of BDS-pga: http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/
[9] Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Sys. Tech. J., Vol. 49, 2, pp. 291-308 (1970)
[10] Hendrickson, B., Leland, R.: The chaco user’s guide: Version 2.0. Sandia National Laboratories (1994)
[11] IEEE: Standard VHDL Language Reference Manual. IEEE Std 1076-2002 (Revision of IEEE Std 1076) (2002)
[12] Eisner, C., Fisman, D.: A Practical Introduction to PSL (Series on Integrated Circuits and Systems). Springer-Verlag New York, Inc. (2006)
[13] IEEE: Standard for Property Specification Language PSL. IEEE Std 1850 (2005)
[14] Pross, U., Richter, A., Langer, J., Markert, E., Heinkel, U.: “Specscribe – Specification data capture, analysis and exploitation”, Software Demonstration at DATE’08 University Booth (2008)
[15] Charles Clos: “A study of non-blocking switching networks”, Bell System Technical Journal 32, March 1953
[16] Yuanyuan Yang: “A Class of Interconnection Networks”, IEEE TRANSACTIONS ON COMPUTERS, VOL. 47, NO. 8, August 1998
[17] Yuanyuan Yang, Gerald M. Masson: “The Necessary Conditions for Clos-Type Nonblocking Multicast Networks”, IEEE TRANSACTIONS ON COMPUTERS, VOL. 48, NO. 1, November 1999
[18] Václav E. Beneš: "Mathematical Theory of Connecting Networks and Telephone Traffic", Academic Press, 1965
[19] Achille Pattavina: “Switching theory: architectures and performance in broadband ATM networks”, Wiley 1998, ISBN-13: 978-0471963387
[20] Bruno Beauquier, Eric Darrot; „On Arbitrary Waksman Networks and their Vulnerability”, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, October 1999
[21] C.E. Leiserson, “Fat-trees: Universal Networks for Hardware-Efficient Supercomputing,” IEEE Transactions on Computers, 34(10):892-901, Oct. 1985
[22] C. M. Fiduccia and R. M. Mattheyses. „A linear time heuristic for improving network partitions“, 19th Design Automation Conference, 1982
[23] Accelera, “Property Specification Language Reference Manual”, v1.1, June 9, 2004
[24] John R. Hauser and John Wawrzyneck, *Garp,* “A MIPS Processor with a Reconfigurable Coprocessor”, Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97, April 16-18, 1997).

A New Client Interface Architecture for the Modified Fat Tree (MFT) Network-on-Chip (NoC) Topology

Abdelhafid Bouhraoua and Muhammad E. S. Elrabaa
Computer Engineering Department
King Fahd University of Petroleum and Minerals
PO Box 969, 31261 Dhahran, Saudi Arabia
{abouh,elrabaa}@kfupm.edu.sa; orwa@diraneyya.com

Abstract— **A new client interface for the Modified Fat Tree (MFT) Network-on-Chip (NoC) is presented. It is directly inspired from the findings related to lane utilization and maximum FIFO sizes found in simulations of the MFT. A new smart arbitration circuit that efficiently realizes a round-robin scheduler between the receiving lanes has been developed. Simulation results show a clear viability and efficiency of the proposed architecture. The limited number of the active receiving links has been verified by simulations. Simulations also show that the central FIFO size need not be very large.**

Keywords — **Networks-On-Chip, Systems-on-Chip, ASICs, Interconnection Networks, Fat Tree, Routing**

I. Introduction

There has been a significant amount of effort made in the area of NoCs, and the focus has mostly been on proposing new topologies, and routing strategies. However, recently the trend has shifted towards engineering solutions and providing design tools that are more adapted to reality. For example, power analysis of NoC circuitry has intensively been studied [1, 2], more realistic traffic models have been proposed [3], and more adapted hardware synthesis methodologies have been developed.

However, high throughput architectures haven't been addressed enough in the literature although the need for it started to become visible [4]. Most of the efforts were based on a regular mesh topology with throughputs (expressed as a fraction of the wire speed) not exceeding 30% [5]. In [6, 7] a NoC topology based on a modified Fat Tree (MFT) was proposed to address the throughput issue. The conventional Fat Tree topology was modified by adding enough links such that contention was completely eliminated thus achieving a throughput of nearly 100% [6] while eliminating any buffering requirement in the routers. Also, simplicity of the routing function, typical of Trees, meant that the router architecture is greatly simplified. These results did not come without a price, mainly the high number of wires at the edge of the network in this case. Also buffering was pushed to the edge of the network at the client interfaces. Many of these issues have been discussed in [6, 7].

In order to overcome these limitations a new client interface architecture is proposed. This interface aims at reducing the number of parallel FIFOs into a single centralized FIFO. The modification of the client interface opens the door for a more practical implementation of the MFT NoC. This paper presents the new client interface architecture and its different circuitry. It also shows through simulation that the new architecture draws on the practical results to reduce the amount of required hardware resources. MFT NoCs are first briefly reviewed in the next section. The newly proposed client interface is then presented in section 3. Simulations results are presented in section 4 followed by conclusions in section 5.

II. Modified Fat Tree NoCs

MFT is a new class of NoCs based on a sub-class of Multi-Stage Interconnection Networks topology (MIN). More particularly, a class of bidirectional folded MINs; chosen for its properties of enabling adaptive routing. This class is well known in the literature under the name of Fat Trees (FT) [8]. The FT has been enhanced by removing contention from it as detailed in [7]. Below is a brief description of the FT and MFT network topologies.

A. FT Network Topology

A FT network, Figure 1, is organized as a matrix of routers with n rows; labeled from 0 to $n-1$; and $2^{(n-1)}$ columns; labeled from 0 to $2^{(n-1)}-1$. Each router of row 0 has 2 clients attached to it (bottom side). The total number of clients of a network of n rows is 2^n clients. The routers of other rows are connected only to other routers. So, in general, a router at row r can reach $2^{(r+1)}$ clients.

B. MFT Topology

Contention is removed in MFT by increasing the number of output ports in the downward direction of each router [6, 7], Figure 2. At each router, the downward output ports (links) are double the number of upper links. Then the input ports of the adjacent router (at the lower level), to which it is connected are also doubled. This is needed to be able to connect all the output ports of the upper stage router. This will continue till the client is reached where it will have $2^n - 1$ input links. Each of these I/P links will feature a FIFO buffer as called for by the original MFT architecture [6, 7].

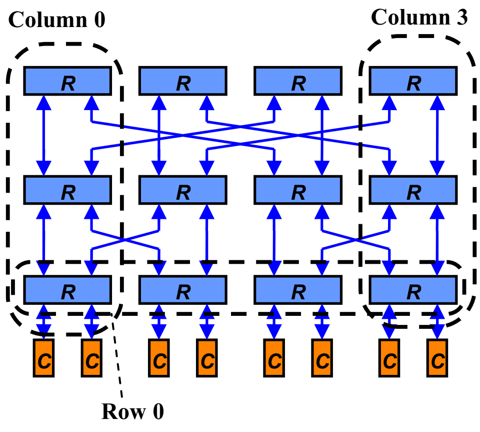


Figure 1: Regular Fat Tree Topology (8 clients)

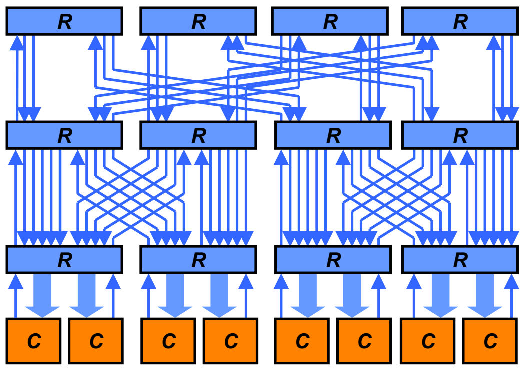


Figure 2 – Modified FT Topology

III. CLIENT INTERFACE

As was explained in section II, the original MFT architecture requires $2^n - 1$ input FIFOs at the client interface. The sizes of these FIFOs is set by the NoC designer depending on many factors such as the communication patterns among clients, emptying (data consumption) rate by a client, application requirements (latency), …etc. [7]. Figure 3 below shows the structure of the client interface in the original MFT.

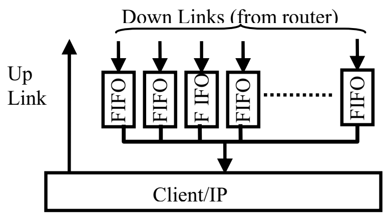


Figure 3: Client interface of the original MFT

It is evident that although these FIFOs may be of a small size, their structure represents the largest part of the cost in terms of area for the MFT network since the routers are bufferless and have a very low gate count. Also, extensive simulations with different traffic generators showed that only a small fraction of FIFO lanes are active per client simultaneously [6, 7].

In order to reduce the wasted FIFOs space represented by the original MFT's client interface, a newly designed interface is proposed, Figure 4. It is made of two parts; an upper part consisting of several bus-widener structures that will be named *parallelizers* from this point forward and a lower part that is simply a single centralized FIFO memory to which all the outputs of the different parallelizers are connected through a single many-to-one multiplexer.

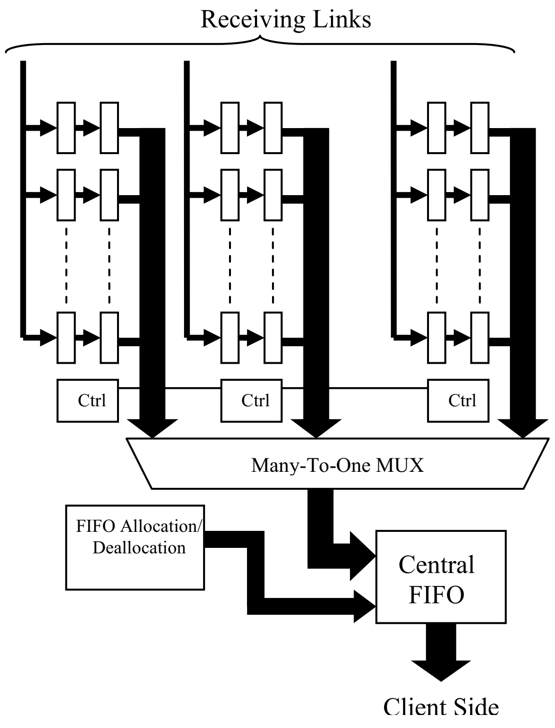


Figure 4: Block Diagram of the New Client Interface.

Each one of the parallelizers is made of two layers. The first layer is a collection of registers connected in parallel to the incoming data bus from one of the receiving ports. Packet data is received into one of these registers one word at a time. When this layer is full, an entire line made by concatenating all the registers of the first layer is transferred to a second set of registers (the second layer in the parallelizer) in a single clock cycle. The ratio between the width of the parallel bus and the width of a single word is called *the parallelization factor*.

Packets portions from different sources are received on different parallelizers simultaneously and independently. When the first portion of a packet is received and transferred to the second layer of the parallelizer, a flag is set to request transfer of a new packet to the FIFO. The control logic responsible for these transfers will first attempt to reserve space in the FIFO corresponding to one packet. The condition here for this architecture to produce efficient results is the adoption of a fixed packet size. This condition simplifies the space allocation in the FIFO and alleviates the control logic from any space or fragmentation management due to variable size allocation and disposal. In the case the FIFO is full and no space could be reserved, the request is rejected and the backpressure mechanism is triggered on that requesting port.

The control logic continuously transfers the received packet words to the FIFO. Every time a packet portion enters the second layer of registers in one of the parallelizers a flag is set to indicate the presence of data. Those parallelizers which are currently receiving packets are said to be active. Only active parallelizers are continuously polled to check the presence of data. The polling follows a round-robin policy. A single clock cycle is used to process the currently selected parallelizer.

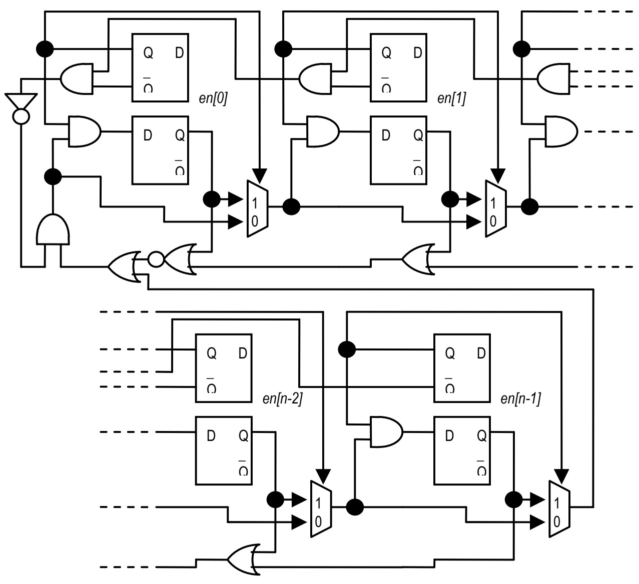


Figure 5: Intelligent Request Propagation Circuitry.

Polling the active parallelizers only supposes some mechanism to “skip” all the non-active parallelizers between two active ones. In order to avoid wasting clock cycles crossing those non-active parallelizers, a special request propagation circuit has been designed. Figure 5 shows this circuit’s schematic. The upper set of flip-flops correspond to the status flag indicating whether a parallelizer is active or not while the lower one is used to indicate which parallelizer is selected to transfer its data during a given clock cycle. The multiplexers are used to instantly skip the non-active parallelizers.

As a result of this fast polling scheme packets arriving simultaneously on different parallelizers may be received in different order. Packet order from the same source is still guaranteed though because the network is bufferless.

IV. Simulation Results

Simulations were carried out using a cycle-accurate C-based custom simulator (developed in-house) that supports uniform and non-uniform destination address distribution as well as bursty and non-bursty traffic models. The packet size was fixed at 64 words. Only bursty traffic with non-uniform address generation was used. The varying parameters were: the network size, central FIFO size and the parallelizer factor value.

Five injection rates corresponding to 0.5, 0.6, 0.7, 0.8 and 0.9 words(flits)/cycle/client were simulated. The first result confirming the viability of the solution was the throughput that matched the input rate in most of the cases and with a maximum difference lower than 1% in few cases.

In all the figures that follow, the latency is expressed in clock cycles.

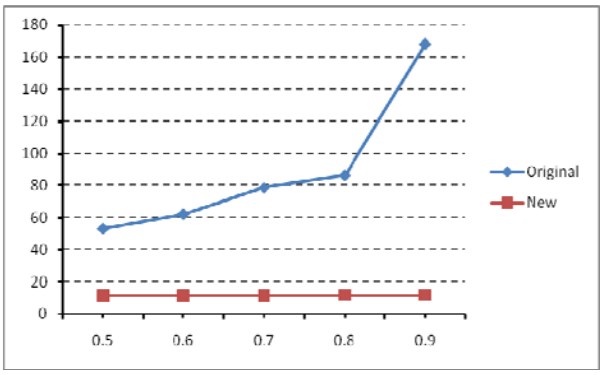


Figure 6: Latency Comparison with the original client interface.

Latency values were reduced dramatically because of the output rate of the FIFO. Dual-port memories are the natural choice for implementing FIFOs. Both data buses are generally the same size on both ports of the dual-port memory. Therefore, the FIFO data bus has the same size as the paralellizers bus. A wide output bus translates in fewer clock cycles to read or write an entire packet. More packets are moved per unit of time which means that packets spend less time in the FIFO waiting to be sent out leading to smaller latencies as shown in Figure 6. It is important to note that the latency figures across the network did not change and is expected to be small as the entire network is bufferless.

Figure 7 shows a subset of the obtained simulation results. The 32 clients network results are shown (left to right) for parallelization factor values of 8, 16 and 32 for different central FIFO sizes (from 8 packets to 32 packets). The latency figures are very low compared to those obtained with the previous architecture of the client interface (Figure 6). The latency range corresponding to a wider parallelizer is lower than the range corresponding to a narrower one. The other results (not shown here for lack of space) are similarly lower for wider parallelizers. These findings confirm the efficiency of the proposed solution.

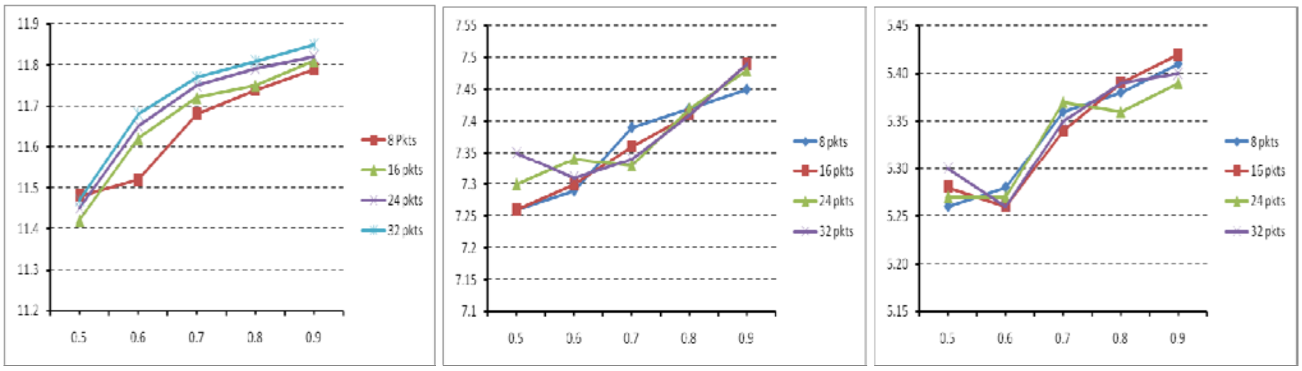


Figure 7 - Simulation Results: latency (clock cycles) versus injection rate (flits/clock cycle/client) for three parallelization factors (8, 16 and 32 from left to right) for several central FIFO sizes (8, 16, 24 and 32 packets).

The central FIFO size has little impact on the results which favors size reduction as a FIFO with a size as low as 8 packets can produce acceptable results.

A tentative synthesis of the new structure yielded about 35K gates per client for a parallelization factor of 8 for a network of 64 clients. This represents approximately an area of 0.185 mm^2 for the 0.13 μ technology. Added to that the dual-port SRAM area of 0.009, 0.018, 0.028 and 0.038 mm^2 for a FIFO size that accommodates respectively 8, 16, 24 and 32 packets. This represents a significant improvement compared to the 2.1 mm^2 occupied by the client interface in the previous architecture and which uses 63 SRAM FIFOs of 2K-Bytes each.

V. Conclusions

A new architecture of the client interface of the MFT NoC has been proposed. This new architecture considerably reduces the hardware resources necessary to implement the receiving client interface. Detailed block diagrams and of this architecture have been shown and described. Its operations and step by step behavior have been described as well. A new arbitration circuit that intelligently "skips" disabled request lines to realize an efficient round-robin where no clock cycles are wasted is presented. Simulations have given clear evidence on the viability of a single centralized FIFO that is simultaneously filled by several, yet limited number, of receiving links. The limited number of the active receiving links has been verified by simulations. The simulation results have shown a considerable reduction of latency compared with the previous solution. They have also shown the little impact of the FIFO size on the latency which implies that a larger size FIFO is not necessary. The client interface synthesis yielded smaller area than in the previous architecture of the client interface by one order of magnitude.

Acknowledgments

This work was supported by King Fahd University of Petroleum and Minerals (KFUPM) through grant # IN070367.

References

[1] E. Nilsson and J. Öberg, "Reducing power and latency in 2-D mesh NoCs using globally pseudochronous locally synchronous clocking", CODES+ISSS 2004.

[2] E. Nilsson and J. Öberg, "Trading off power versus latency using GPLS clocking in 2D-mesh NoCs", ISSCS 2005.

[3] V. Soteriou, H. Wang and L. Peh, "A Statistical Traffic Model for On-Chip Interconnection Networks", in *Proceedings of the 14th IEEE Intl. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, Sept. 2006, pp 104-116.

[4] Freitas H.C., Navaux P. "A High Throughput Multi-Cluster NoC Architecture", *11th IEEE International Conference on Computational Science and Engineering*, July 16-18, 2008 Sao Paulo, Brazil, pages 56-63

[5] K. Goossens, J. Dielissen, A. Radulescu, "Æthereal network on chip: concepts, architectures, and implementations", IEEE Design and Test of Computers, Volume 22, Issue 5, Sept.-Oct. 2005 Page(s)414 – 421.

[6] A. Bouhraoua and Mohammed E.S. El-Rabaa, "A High-Throughput Network-on-Chip Architecture for Systems-on-Chip Interconnect," *Proceedings of the International Symposium on System-on-Chip* (SOC06), 14-16 November 2006, Tampere, Finland.

[7] A. Bouhraoua and Mohammed E.S. El-Rabaa, "An Efficient Network-on-Chip Architecture Based on the Fat Tree (FT) Topology", *Special Issue on Microelectronics*, *Arabian Journal of Science and Engineering,, Dec. 2007,* pp 13-26.

[8] C. Leiserson, "Fat-Trees: Universal Networks forHardware-Efficient Supercomputing", *IEEE Transactions on Computers*, Vol. C-34, no. 10, pp. 892-901, October 1985.

Implementation of Conditional Execution on a Coarse-Grain Reconfigurable Array

Fabio Garzia, Roberto Airoldi, Jari Nurmi
Department of Computer Systems
Tampere University of Technology
33101 Tampere, FI
Email: *name.surname*@tut.fi

***Abstract*—This paper presents a method to implement *switch/case* type conditional execution on a coarse-grain reconfigurable array based on a SIMD paradigm. The implementation do not introduce dedicated hardware but it utilizes only the functional units of the processing elements composing the machine and the possibility to reconfigure each processing element at run-time in one clock cycle. The method is employed to map algorithms for linear search or calculation of the maximum value on vectorized data.**

I. Introduction

During recent years academic and industrial research groups have proposed several coarse-grain reconfigurable architectures (CGRA). The typical CGRA is characterized by a one or two dimensional array of processing elements (PEs), modeled on general-purpose computer systems. A common choice is to provide very simple processing elements based on programmable arithmetic-logic modules and characterized by multiplexing logic for the interconnections between PEs. It is the case of Morphosys [1] and Montium [2]. These machines are particularly suitable to map SIMD applications. However, they provide poor support for control tasks. The main issue is the mapping of conditional execution. In Morphosys the designers introduced the support for guarded execution and pseudo-branches [1] using dedicated logic to store branch tags and implement predication. In Montium [2] an external sequencer takes care of conditional execution.

In this work we propose an alternative approach to implement *switch/case* type of conditional execution on a coarse-grain reconfigurable machine. This approach uses the functional units already provided in each PE of the machine and the possibility to reconfigure functionalities and interconnections in one clock cycle.

The paper is organized as follows. First we present some details of the reconfigurable device. Then we describe the implementation of the *switch/case* conditional execution and some practical example of its employment. Finally we draw some conclusions.

II. CREMA Architecture

CREMA [3] is a Coarse-grain REconfigurable array with Mapping Adaptiveness. Its architecture is based on a matrix of 4×8 coarse-grain processing elements (PEs) that can process two 32-bit inputs and generates two 32-bit outputs per clock cycle. Operations supported are integer and floating-point arithmetics, shifting, LUT-based and boolean. In addition, CREMA supports 16 different inter-PE interconnections divided into nearest-neighbor, interleaved, and global interconnections.

CREMA is based on a template that is customized according to the requirements of a kernel to execute. This is the reason for the term *"mapping adaptiveness"*. The idea is that the application developer maps a certain kernel onto CREMA template. This mapping generates a set of contexts that are used in the run-time execution of the kernel by CREMA. Based on the set of contexts required, a minimal version of CREMA is generated, in which all the unused functional units and interconnections are removed from the initial template.

The reduced version of CREMA implements run-time reconfiguration, because each PE may support more than one operation or connection. The run-time choice between the functional units and the interconnections provided in hardware is performed setting a configuration word in each PE. Changing the configuration word corresponds to changing the current functionality of the PE. In practice, configuration words are stored in a PE memory, that can host more than one configuration words allowing a one-cycle reconfiguration.

III. Implementation of a *switch/case* statement

A control flow mechanism based on a *switch/case* statement is implemented using these features of the CREMA template:

1) the Look-Up Table (LUT) among its functional units;
2) the possibility for any PE to acquire the configuration word from the upper PE instead of using its own configuration memory;
3) the run-time reconfiguration performed in one clock cycle.

A *switch/case* conditional statement is based on the execution of different operations according to the value of a selection variable. To implement it on CREMA array, we assume that the possible values of the selection variable can be used to address a memory, i.e., they compose a sequence of integers starting from zero or can be easily converted into such a sequence using arithmetic or logic operations. The idea is that this values can select one location of a LUT instantiated in one PE. In addition, consider that each PE can be configured by the upper PE instead of its own configuration memory.

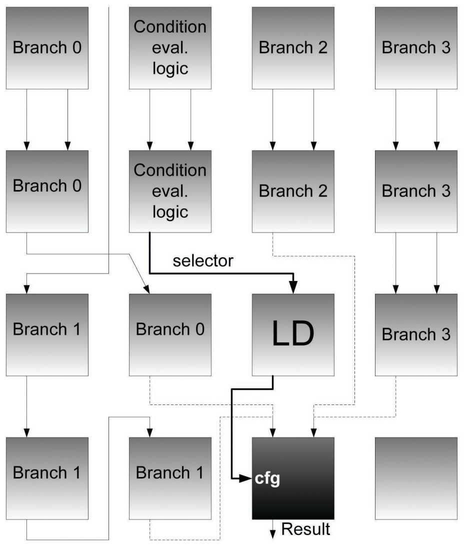


Fig. 1. Implementation of a *switch/case* with four branches.

Therefore, we can map a selection mechanism followed by a conditional operation using two consecutive PEs in a column of the array (see Fig. 1). The first PE ("LD" in the figure) loads a value from its LUT addressed by the condition variable and the second PE (in black in the figure) implements the functionality specified by the output of the LUT.

There is no theoretical limitation on the size of the *switch/case*, because the size of the LUT can be decided at design time. However, there is a practical constraint. The different operations required by the branches of the *switch/case* must be implemented using different configurations of the same PE. This is not always possible. An alternative is to use the PE as a selector, whose only task consists of taking a value from one of the possible execution paths mapped using more than one PE. Fig. 1 depicts a situation in which the *switch/case* is characterized by 4 branches, each branch is mapped on a set of PE ("Branch #") and the condition is also evaluated using two PEs. This way the implementation of the *switch/case* mechanism is constrained only by the array resources.

Notice that the *switch/case* mechanism can be used to implement an *if-then-else*. It is sufficient to convert the *if-then-else* in a *switch/case* with two branches. However, the condition evaluation may be more difficult to map.

IV. Test Cases

The mechanism illustrated here can be used to implement a linear search on vectorized data. For example, it is possible to check in a vector of integers if there are elements greater than, less than or equal to a fixed value. Due to the nature of CREMA execution, the outcome is always another vector. For example, this vector may have ones in the position of the elements that satisfy the required condition and zeros elsewhere. Such a vector can be used for further processing by CREMA, like adding together all the ones to know how many elements satisfy the fixed condition. These linear searches can be performed in N clock cycles by CREMA, where N is the number of elements of the array.

One of the implementation issues in these algorithms is to match the condition with a *switch/case* statement, that implies that a value is associated to each result of the condition evaluation. Our approach does not insert adhoc logic for condition evaluation, therefore the value must come from one of the provided PE operation. In most cases, the solution is simple. If we need to evaluate the condition $x > C$, we can perform in a PE the operation $x - C$ and then consider the most significant bit by shifting the result by 31, so that we get 0 or 1 according to the condition. This means that we need only two PEs. On the other hand, the condition $x = C$ requires a subtraction and then a bitwise OR between all the bits of the result.

The same mechanism was used to implement an iterative algorithm for the search of a maximum inside a vector. The algorithm is composed of several steps. At each step the elements are grouped in pairs and the maximum inside each pair is stored into the memory. At each step we get a new vector that is half of the size of the previous one. The steps are iterated until only one element is left. Therefore the execution requires $\log_2 n$ steps. The algorithm employs another feature of CREMA that allows to write the results in the output memory according to a specific pattern. The algorithm was used in the implementation of a W-CDMA cell search on a platform based on CREMA [4].

V. Conclusion

In this paper we propose a method to map *switch/case* conditional execution on a coarse-grain reconfigurable array. The method does not require additional logic for predication or branch support, but it is based on the possibility to reconfigure the PE functionality in one clock cycle and to map condition evaluation and conditional operations onto the PEs. The method has been employed for the mapping of linear search on vectorized data or calculation of the maximum value inside a vector.

Acknowledgment

This work was partially funded by a grant awarded by the Finnish Cultural Foundation, which is gratefully acknowledged.

References

[1] M. Anido, A. Paar, and N. Bagherzadeh, "Improving the operation autonomy of simd processing elements by using guarded instructions and pseudo branches," in *Proceedings of Euromicro Symposium on Digital System Design*, 2002, pp. 148 – 155.

[2] G. Smit, P. Heysters, M. Rosien, and B. Molenkamp, "Lessons Learned from Designing the MONTIUM - a Coarse-grained Reconfigurable Processing Tile," in *Proc. International Symposium on System-on-Chip*, 2004, pp. 29–32.

[3] F. Garzia, W. Hussain, and J. Nurmi, "Crema: A coarse-grain reconfigurable array with mapping adaptiveness," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL2009)*. Prague, CZ: IEEE, September 2009, pp. 708–712.

[4] F. Garzia, "From run-time reconfigurable coarse-grain arrays to application-specific accelerator design," Ph.D. dissertation, Tampere University of Technology, December 2009.

Dynamically Reconfigurable Architectures for High Speed Vision Systems

Omer Kilic, Peter Lee
School of Engineering and Digital Arts
University of Kent
Canterbury, KENT, CT2 7NT, England
Email: {O.Kilic, P.Lee}@kent.ac.uk

***Abstract*—High-Speed Vision applications have very specific and demanding needs. Many vision tasks have high computational requirements and real-time throughput constraints. In addition to a large number of fine-grain computations, vision tasks also have complex data flow which varies significantly in a single application and across different applications.**

Conventional architectures, like Microprocessors and standard Digital Signal Processors (DSP) are not optimised for these kinds of highly specific applications and as a result do not perform well because of their mostly sequential nature. By using a combination of reconfigurable architectures that can adapt themselves to the requirements of the system dynamically, significant performance improvements can be achieved.

I. Introduction

This PhD project focuses on evaluating the performance gains achieved by utilising a combination of processing devices, each with different characteristics and strengths to satisfy the requirements of high speed vision systems. In the proposed system, a combination of a Field Programmable Gate Array (FPGA), a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU) will be employed. The FPGA portion of the system will be responsible for image acquisition and the low-level pre-processing tasks such as filtering, edge detection and thresholding and the CPU will act as the system supervisor overseeing the operation of the FPGA and the GPU sections of the system. The CPU will also be responsible for coordinating the outside connectivity of the system, which can be in the form of network communication or standard embedded interfaces. For processing data on the GPU, CUDA$^{\text{TM}}$(Compute Unified Device Architecture) framework will be used.

While there are some custom highly specialised systems available that utilise this sort of heterogeneous combination of devices, there is a lack of an open, non-proprietary platform, hence the ultimate aim of this project is to define a flexible low-cost framework for high speed vision systems that utilises commercial off-the-shelf (COTS) peripherals, where vision tasks can be described in a high level system description package. The system description package will define the overall purpose of the system and it will also provide flexible constructs that enable the system to adapt itself to the changes in the operating environment.

In Section 2, a general overview of processing requirements of Vision Systems will be discussed. Section 3 will outline advantages of using reconfigurable architectures and Section 4 will provide details about the proposed system. Section 5 will outline the challenges and Section 6 will provide the conclusion for this paper.

II. Vision Systems

The goal of Machine Vision is to robustly extract useful, high level information from images and video. The type of high-level information that is useful depends on the application. Vision systems have a wide range of applications from highly specialised instrumentation and process automation tasks to general consumer electronics, although this proposed architecture mainly focuses on providing a flexible low-cost framework for industrial instrumentation tasks.

Vision algorithms have several stages of processing. The lowest level operations, performed on raw sensor data usually involve a lot of arithmetic operations on pixel values. These operations can benefit from the parallel nature of the architecture in use quite significantly and reconfigurable architectures have been most widely used for accelerating these algorithms. Examples of computations at this level include filtering, edge detection, and edge-based segmentation, among others[1].

An example application that can benefit from our proposed system is Stereo Vision. Stereo Vision is a traditional method for acquiring 3-dimensional information from a stereo image pair. The instruction cycle time delay caused by numerous repetitive operations causes the real-time processing of stereo vision to be difficult when using a conventional computer[2]. By abstracting the different layers of the Stereo Vision system to run in parallel on different devices, significant performance improvements can be expected.

III. Reconfigurable Architectures

Reconfigurable architectures utilise hardware that can be adapted at run-time to facilitate greater flexibility without compromising performance. Fine-grain and coarse-grain parallelism can be exploited because of this adaptability, which provides significant performance advantages compared to conventional microprocessors[3]. An example of this adaptability could be an image processing system that dynamically adjusts the window size of the algorithm employed. The ability to change the functionality of the device during run-time also makes these architectures more flexible compared to traditional methods like using Application Specific Integrated Circuits (ASIC) or static instruction-set processors. Depending on the application, they may also provide cost-savings by reducing the required logic density.

Most research on reconfigurable computing has been done based on Field Programmable Gate Arrays (FPGAs) due to their flexible configuration and their shorter design cycle compared to ASICs. The FPGA approach adds design flexibility and adaptability with optimal device utilization while conserving both board space and system power, which is often not the case with DSP chips. Currently, FPGAs are capable of supporting multi-million gate designs on a chip and the computer vision community has become aware of the potential for massive parallelism and high computational density in FPGAs. In addition, the software-like properties afforded by Hardware Description Languages (HDLs) such as encapsulation and parameterization allow creating more abstract, modular and reusable architectures for image algorithms.[4]

IV. Proposed System

As discussed earlier, the proposed system will employ FPGA and GPU co-processors, connected to a CPU host machine which will act as the co-ordinator of the entire system. This host machine will be responsible for parsing the system description package and off-loading relevant processing tasks on to the co-processors. It will also be responsible for interfacing the system to the outside world, which can be in the form of network communication or conventional embedded interfaces.

The system description package will define the overall purpose of the system and will have flexible constructs to provide adaptability to the changes in the processing tasks. By providing a flexible format, the need for constant monitoring and the reconfiguration overhead of the system can be reduced. In order to maximise the performance potential of the inherent parallel nature of the FPGA and the GPU devices, concurrent programming languages (and languages with concurrency support) are being investigated for use within the system description package.

The system can essentially be partitioned into two sub-systems:

- The low-level image acquisition and pre-processing unit, powered by an FPGA device which will have an array of reconfigurable blocks that can be modified to customise the operation of the device dynamically
- Further processing and characterisation unit, powered by the CPU and the GPU on the host machine that can further process the images coming from the FPGA unit

These units are tightly coupled with each other and preventing bottlenecks will be a very important part of the implementation. A high speed PCI Express interface will be employed to provide connectivity between these sub-systems.

A. FPGA Co-processor

Programmable Logic Devices, mainly FPGAs, have always been the choice for high speed and computationally intensive applications. The speed advantage of FPGAs derives from the fact that programmable hardware is customised to a particular algorithm. Thus the FPGA can be configured to contain exactly and only those operations that appear in the algorithm. In contrast, the design of a fixed instruction set processor must accommodate all possible operations that an algorithm might require for all possible data types[5].

Because of the flexible nature of FPGAs, reconfigurable architectures often make use of these devices. Modern FPGAs have native support for dynamic reconfigurability and contain dedicated DSP resources and high speed memory interfaces which makes them particularly suitable for low level image pre-processing tasks.

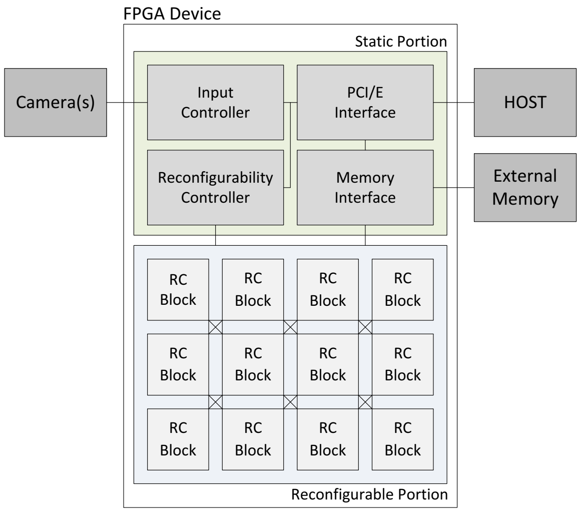


Fig. 1. FPGA Architecture

Figure 1 outlines the proposed FPGA Co-processor architecture. The static part of the system consists of:

- Input controller, responsible for data/image acquisition from outside sources, such as a single or several high speed camera(s)
- PCI Express Interface, to transfer data in and out of the FPGA co-processor and to provide host machine access to the Reconfigurability Controller within the device
- Memory Interface, to provide external high speed memory access to the system that may be used by certain algorithms and possibly for buffering the data between the FPGA and the host machine
- Reconfigurability Controller, that deals with the operation and (re)configuration of the reconfigurable blocks. Host machine controls the operation of the Reconfigurability Controller depending on the system description package

The dynamic part of the system has reconfigurable blocks arranged in a flexible matrix structure which enables the system to continue processing even when a number of these blocks are under reconfiguration. The functionality of these blocks can be dynamically altered to enhance the performance and/or the accuracy of the algorithm used or simply to replace the running algorithms with a new set. This provides flexibility so the system will be able to adapt itself to the changes in the operating environment easily.

B. Host Machine

After the initial image acquisition and pre-processing stages, the image data will be transferred to the host machine where further processing and classification can commence. This host machine will have an x86 CPU and it will run a customised operating system. The co-processors will be connected to this system via PCI Express interface.

A CUDA framework capable NVIDIA®GPU will be used as an arithmetic accelerator to enhance the performance of certain image processing algorithms that can benefit the massively parallel nature of a GPU device. CUDA is the programming language provided by NVIDIA to run general purpose applications on NVIDIA GPUs. It incorporates an API (Application Programmer Interface), a runtime, couple of higher level libraries and a device driver for the underlying GPU. The most important thing about the CUDA is that it has almost addressed some of the inherited general purpose computing problems with GPUs. CUDA's API for the programmer is an extension to the C programming language and CUDA allows developer to scatter data around the DRAM as well as it features a parallel data cache or on chip shared memory for bringing down the bottleneck between the DRAM and the GPU[6]. In the past GPUs have been used as general-purpose computational units by wrapping computations in graphics function libraries but with the emergence of CUDA this level of abstraction is not necessary anymore.

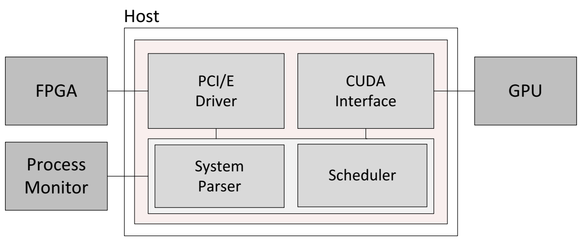


Fig. 2. Host Architecture

Figure 2 outlines the elements of the host machine. These elements consist of:

- System Parser, that reads the system description package and the Scheduler that offloads relevant processing tasks to the co-processors
- CUDA Interface, that manages and coordinates CUDA processing tasks running on the GPU
- PCI Express Driver, that interfaces the host machine with the FPGA co-processor

V. Challenges

Challenges in the development of this system include:

- Understanding the operation of Partial Dynamic Reconfiguration on the FPGA device used, which include investigation of bus macros and generation of partial bitstreams in an efficient and effective way.
- The structure of dynamically reconfigurable blocks with emphasis on details such as block size (granularity of the blocks) and support for the FPGA specific silicon features such as DSP blocks, Block RAMs, etc.
- The arrangement of dynamically reconfigurable blocks in a flexible matrix structure and placement issues on the actual silicon device
- Getting the PCI Express communication between the FPGA and the Host System working and coming up with a solution that deals with the prevention of bottlenecks between the devices
- Designing the Input Controller and the physical hardware so that the system can be interfaced with high speed cameras
- The definition of a comprehensive system description package format, where the operation of the entire system accompanied by the relevant processing directives can be described in a flexible and simple format. The parallel nature of the system is a major factor in the choice of a language for the system package format and several programming languages with concurrency features are being investigated.

After the initial implementation phase, the system will be benchmarked with a few common algorithms and applications such as the Hough Transform, Object Recognition/Tracking and Stereo Vision. The outcome of these tests will help us fine tune the system and improve the overall performance.

VI. Conclusion

A multi-device, dynamically reconfigurable architecture is proposed to satisfy the requirements of high-speed/real-time vision systems.

Initial specification of this system has been defined and implementation of different sub-systems is under way.

If successful, authors believe that this system will provide a flexible low-cost framework for high speed vision systems.

References

[1] M. A. Iqbal and U. S. Awan, "Run-time reconfigurable instruction set processor design: Rt-risp," in *Proc. 2nd International Conference on Computer, Control and Communication IC4 2009*, 17–18 Feb. 2009, pp. 1–6.

[2] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S.-K. Park, M. Kim, and J. W. Jeon, "Fpga design and implementation of a real-time stereo vision system," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 20, no. 1, pp. 15 –26, jan. 2010.

[3] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1201–1217, July 2002.

[4] C. Torres-Huitzil, S. Maya-Rueda, and M. Arias-Estrada, "A reconfigurable vision system for real-time applications," Dec. 2002, pp. 286–289.

[5] M. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005, http://www.amazon.co.uk/dp/0387261052.

[6] N. Karunadasa and D. Ranasinghe, "Accelerating high performance applications with cuda and mpi," pp. 331 –336, dec. 2009.

Virtual SoPC rad-hardening for satellite applications

SYRIUS project

L. Barrandon[1], T. Capitaine[2], MIS laboratory, Amiens, France

L. Lagadec[3], CACS team, Lab-STICC, Brest, France

N. Julien[4], CACS team, Lab-STICC, Lorient, France

C. Moy[5], SCEE team, SUPELEC/IETR, Rennes, France

T. Monédière[6], OSA department, XLIM laboratory, Limoges, France

[1]ludovic.barrandon@u-picardie.fr, [2]thierry.capitaine@u-picardie.fr, [3]loic.lagadec@univ-brest.fr, [4]nathalie.julien@univ-ubs.fr, [5]christophe.moy@supelec.fr, [6]thierry.monediere@xlim.fr

Abstract— **Our contribution addresses the specific problematic of the design of satellites' on-board computers based on the use of non rad-hard devices while keeping the security and functioning safety constraints. The topics dealt with in this context will be based on the dynamic programming related to the exploitation of data provided by the development of auto diagnostic tools for embedded re-programmable devices, considering safety/power consumption trade-offs.**

Keywords-virtual layer; auto-diagnostic; non rad-hard FPGAs; satellite on-board computer; power/energy consumption; SDR.

I. Introduction

Cosmic rays and solar winds (protons) can induce two types of failures in space-borne digital devices: single-event upsets (SEUs are transient and cause bit inversions) and single-event latch-ups (SELs are permanent and destroy logic and routing resources). Anti-fuse FPGAs can circumvent the SEUs issue but do not permit reconfiguration: triple redundancy techniques are needed to prevent malfunctions due to SELs.

The main motivation of the SYRIUS project (SYstèmes embaRqués génériques reconfigUrables pour Satellites i.e. Generic and reconfigurable embedded systems for satellites) is to design a generic and reconfigurable embedded system taking into account the up-to-date methods and technologies to ensure reliability, low power/energy consumption and flexibility as stated in [1]. The use of non radiation-hardened devices stands for a technological paradigm to enable space electronics industries not to implement either rad-hard device (expensive, difficult to maintain, under ITAR -International Traffic in Arms Regulations- and technologically old-fashioned) or ASICs (long design processes, extremely costly, poorly/not reconfigurable and "right-first-time" in 60% of the cases).

The system will be composed of the following « modules »: auto-diagnostic, smart patch-antenna arrays, dynamic partial reconfiguration for reliability and power/energy consumption optimization, digital radio. They will integrate the spatial-domain environmental constraints so as to obtain the related certifications and validations (radiations, vibrations and thermal test-bench) mandatory to integrate and launch this platform on a 30x30x30cm satellite (French National Space Agency –CNES– RISTRETTO format [2]). Communication, reconfiguration and exploitation of embedded scientific experimentations will be done via our GENSO ground station [3] (i.e. several current or future projects will be able to join the satellite's payload as scientific experimentations).

II. Technical Shifts

A. Killing three birds with one stone

Present space digital hardware designs mainly use expensive and ITAR-constrained rad-hard devices and ASICs. Using off-the-shelf Systems on Programmable Chip (SoPC) (1) would dramatically reduce the costs in terms of hardware (2) shorten time-to-market (enabling design-and-reuse) and (3) offer the possibility to use up-to-date technologies. This last remark associated with SoPC reconfigurability lead to consider many potential improvements for space embedded systems which are intended to be studied in the frame of SYRIUS:

- Remote redefinition of the satellite's mission;
- Software Defined Radio (SDR) and Smart Radio [4] with protocol testing and auto-adaptive functionalities;
- Design under safety and power/energy constraints;
- "Programmed death": the SoPC will experience a gradual destruction of its resources until a minimum functional state is reach whereas rad-hard devices can become permanently faulty at anytime.

B. Auto-diagnostic and DPR

The detection and localization of faults associated with the Dynamic Partial Reconfiguration (DPR) is of major interest: the goal is to take advantage of the reconfigurability and the regularity of the SRAM FPGAs' structure to consider unused logic as spare resources in case of SELs. This can be done thanks to a priori allocation using spare configurations, selecting the best partitioning granularity (programmable-logic-block level, triple modular redundancy with standby configurations or modular level) or with dynamic processes to re-route or repair algorithms [5]. Built-in self test (BIST) methods are to be implemented and can be inspired from [6] (off-line method) or [7] where roving self-test areas are exhaustively displaced across the FPGA while in operation and from [8] implementing competing configurations.

C. Methodology

1) Virtual layer

In the software domain, the notion of virtual machine (VM) is renowned for its ability to isolate the physical target (OS and hardware) from the application layer. This method ensures both portability and sustainability. In the embedded systems domain, constrained environments can take advantage of the VM approach: the resources needed by the virtual layer can be compensated by the design of a VM optimized according to the physical target. In the fault-tolerant context, the common method consists in post-processing the netlists to insert redundancy before place-and-route steps. TMRTool [9] and BL-TMR [10] software can handle this technique. Its weakness is due to the need to implement redundancy for each new application, that's why we believe that factorizing this effort within a virtual layer is of particular interest: it would enable to implement the design without manipulating the netlists.

The hardware targets are usually considered as reliable which is not true in this non rad-hard context. A challenge is to implement failure schemes and correction methods in the virtual layer to ensure robustness while relaxing failure management tools.

2) *Safety engineering and power/energy consumption*

Whereas research is prosperous in both domains, there are few studies dealing simultaneously with these two constraints [11]. We are willing to develop an innovative design technique based on a state-graph driven system-level modeling. It will provide an optimum solution led by both quality of service and power/energy consumption and implement decision algorithms to reconfigure the application.

III. Hardware architecture

The SYRIUS system is composed of a ground section for radio-communications, scientific experimentation, failure recovery and fault analysis and the satellite (payload and on-board computer itself). This last module stands for the core of the system and is multi purpose: analog and digital radio management, power management, autodiagnostic, fault recovery and reconfiguration management.

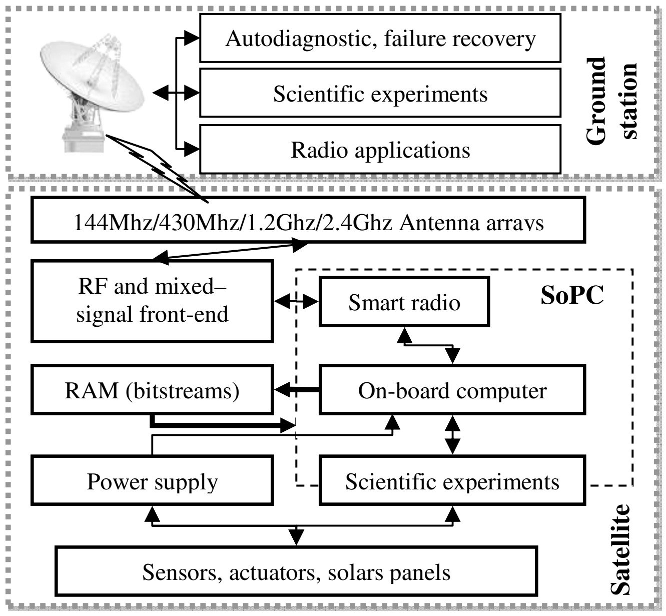


Figure 1. Hardware architecture of the SYRIUS platform.

A multi-frequency patch-antenna array will be designed to be mounted on each face of a satellite ([12], [13]), using the state-of-the art results about auto-adaptive coupling to optimize the Earth-satellite radio link budget and, consequently, to reduce the electrical power and energy consumption.

A secured management sub-system, necessary to any satellite, will ensure the vital tasks and the mission redefinition by dynamic reprogrammation of digital devices via a ground station. In this context, an auto-adaptive and multi-protocol SDR sub-system will be implemented.

IV. Perspectives

Designing this generic space-borne on-board computer aims at reducing costs, time-to-market and validation cycles keeping high reliability constraints. Preparing the launch of this system for testing purpose will drive our developments.

Acknowledgment

The SYRIUS project has been submitted to the French National Research Agency and is intended to involve two societies (Steel Electronique and Nova Nano) and the AMSAT association for amateur radio.

References

[1] Sghairi, M.; Aubert, J.-J.; Brot, P.; de Bonneval, A.; Crouzet, Y.; Laarouchi, Y., "Distributed and Reconfigurable Architecture for Flight Control System", 28th Digital Avionics Systems Conference, DASC 2009, IEEE/AIAA, 23-29 Oct. 2009, pp. 6.B.2-1 - 6.B.2-10.

[2] M. Saleman, D. Hernandez, C. Lambert, "RISTRETTO: A French Space Agency Initiative for Student Satellite in Open Source and International Cooperation," AIAA/USU Conf. on Small Satellites, Logan, UT, USA, Aug. 10-13, 2009, SSC09-VII-8.

[3] T. Capitaine, V. Bourny, L. Barrandon, J. Senlis, A. Lorthois, "A satellite tracking system designed for educational and scientific purposes", ESA 4S (Small Satellite Systems and Services) Symposium 31 May - 4 June 2010, Funchal, Madeira.

[4] W. Jouini, C. Moy, J. Palicot, "On decision making for dynamic configuration adaptation problem in cognitive radio equipments: a multi-armed bandit based approach", 6th Karlsruhe Workshop on Software Radios, March 3 - 4, 2010.

[5] M. G. Parris, "Optimizing Dynamic Logic Realizations for Partial Reconfiguration of Field Programmable Gate Arrays", School of Electrical Engineering and Computer Science, University of Central Florida, Orlando.

[6] T. Nandha Kumar, C. Wai Chong, "An automated approach for locating multiple faulty LUTs in FPGA" Microelectronics Reliability 48 (2008) pp 1900-1906.

[7] J.M. Emmert, C.E. Stroud, M. Abramovici, "Online Fault Tolerance for FPGA Logic Blocks", IEEE Trans. on VLSI Syst. 2007 vol 15, 216-226.

[8] R.F. Demara, K. Zhang, "Autonomous FPGA fault handling through competitive runtime reconfiguration", NASA/DoD Conference on Evolvable Hardware, Washington D.C., U.S.A., 2005, 109-116.

[9] Xilinx, "TMR tool", www.xilinx.com/ise/optional_prod/tmrtool.htm

[10] Brigham young University, "BYU EDIF Tools Home Page", http://sourceforge.net/projects/byuediftools/.

[11] Toshinori Sato, Toshimasa Funaki, "Dependability, Power, and Performance Trade-off on a Multicore Processor", Asia & South Pacific Design Automation Conf., ASPDAC, 21-24 March 2008, pp. 714–719.

[12] Saou-Wen Su, Shyh-Tirng Fang, Kin-Lu Wong "A Low-Cost Surface-Mount Monopole Antenna for 2.4/5.2/5.8-GHz Band Operation" Microwave and Optical technology letters, vol 36, March 2003.

[13] C. Ying and Y.P. Zhang "Integration of Ultra-Wideband Slot Antenna on LTCC Substrate", Electronics Letters, 27 May 2004, vol 40, N°11.

ReCoSoC 2010

ISBN 978-3-86644-515-4

ISSN 1869-9669
ISBN 978-3-86644-515-4