A Reconfigurable Processor for Heterogeneous Multi-Core Architectures

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Artjom Grudnitsky

aus Odessa

Datum der mündlichen Prüfung:	21. Dezember 2015
Erster Gutachter:	Prof. Dr.-Ing. Jörg Henkel, Karlsruher Institut für Technologie (KIT), Fakultät für Informatik, Lehrstuhl für Eingebettete Systeme
Zweiter Gutachter:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker, Karlsruher Institut für Technologie (KIT), Fakultät für Elektrotechnik und Informationstechnik, Institut für Technik der Informationsverarbeitung

Artjom Grudnitsky
Erich-Kästner-Str. 22
76149 Karlsruhe

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen — die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Artjom Grudnitsky

Abstract

Performance of applications with computationally intensive kernels can be improved by a general-purpose processor used together with application-specific hardware accelerators. To accelerate a wide range of applications with different kernels, a run-time reconfigurable fabric (such as an embedded FPGA) is used. Each application can deploy its own accelerators onto the fabric in order to improve its execution speed. Such *reconfigurable processors* have been explored in research and industry and have demonstrated performance improvements for various applications using the fabric.

However, current state-of-the-art approaches for use of reconfigurable processors in *multi-tasking* scenarios and *multi-core* systems have several disadvantages. Specialized task schedulers are designed for reconfigurable systems that have limited flexibility: they either stall during accelerator reconfiguration or use an "all-or-nothing" approach for running kernels on the fabric (i.e. by either using a single set of accelerators to implement the kernel, and if that is not available then by running the kernel on the processor pipeline without using any fabric). Reconfigurable processors allowing a trade-off between used fabric area and achieved performance use classic task schedulers designed for non-reconfigurable systems and suffer the drawback of long reconfiguration time, which reduces application speedup. Existing approaches for sharing the reconfigurable fabric in multi-core systems can be categorized into two classes: allowing access of all cores to a single shared fabric, and providing a dedicated fabric to each core. Both classes have drawbacks in efficiently using fabric resources: systems with a dedicated fabric have no (or limited) adaptation to workloads changing at run-time, while systems with a shared fabric have limited parallelism when the fabric is accessed by multiple cores simultaneously.

The goal of this work is to design concepts for using reconfigurable processors efficiently in multi-tasking scenarios and for efficiently sharing the reconfigurable fabric in a multi-core system. The key contributions are:

Task scheduling for reconfigurable processors
The drawback for the adaptivity of reconfigurable processors is the comparatively long time to reconfigure an accelerator, which negatively impacts application performance. In multi-tasking workloads this reconfiguration time can be effectively "hidden" by a task scheduler. First, the notion of task efficiency is introduced to describe how well a task benefits from the current state of the reconfigurable processor (i.e. currently reconfigured accelerators in relation to the accelerators required by the task). Based on task efficiency, a scheduler for workloads with deadlines is developed, with the goal of meeting deadlines better. Additionally, a combined scheduling and fabric

allocation approach is developed for workloads without deadlines, with the goal of improving makespan (completion time of the taskset). Both schedulers detect when task efficiency drops, and attempt to schedule another task until task efficiency of the first task improves, resulting in improved overall system performance.

Reconfigurable fabric sharing in multi-cores
When run on the reconfigurable fabric, kernels utilize different sets of fabric resources (e.g. accelerators, fabric-internal storage, memory bandwidth to external memory) throughout their run-time. Using a reconfigurable processor to run a single kernel results in a large amount of idle fabric resources. In a multi-core system these idle resources can be used to run other kernels in parallel on the fabric. A novel way to share the fabric in a multi-core is proposed, addressing the drawbacks of state-of-the-art approaches, with the goal of improving the performance of the whole multi-core. By using a heterogeneous multi-core consisting of both a reconfigurable and non-reconfigurable cores (connected to the fabric of the reconfigurable core), the fabric can be accessed by all cores, while still prioritizing the reconfigurable core, allowing it to process the most demanding applications. Concurrent execution of kernels on the fabric is enabled by the merging concept, which combines simultaneous accesses from different cores to the fabric in hardware.

Flexible fabric use in multi-tasking and multi-core systems
Using reconfigurable fabric in the above mentioned multi-tasking scenarios and multi-core systems with dynamic workloads requires that kernels from different tasks can be executed on the fabric, no matter what parts of the fabric were allocated to the task. This high degree of flexibility is not met when kernels are prepared (or: synthesized) for fabric execution at compile-time, as the decision which parts of the fabric is allocated to a task or core is done at run-time for dynamic workloads. Synthesizing kernels fully at run-time would result in a high overhead, thus the proposed approach only performs those parts of kernel synthesis at run-time that depend on current allocation of fabric area to tasks. In order to further reduce overhead, a novel software-cache for synthesized kernels is introduced along with a technique to reconfigure accelerators into cache-friendly locations on the fabric.

Evaluation shows that the proposed scheduler for workloads with deadlines was able to improve tardiness by 1.22× on average compared to the closest competitor. The proposed scheduler and fabric allocator for makespan improvement was able achieve only 6% worse results than the lower bound, while the results of other schedulers were between 12% and 20% worse than this lower bound. In multi-core systems the proposed fabric sharing technique achieved better performance than either type of state-of-the-art approach – 1.3× better on average on the non-reconfigurable cores of a multi-core (with the same performance on the reconfigurable core), or by 3.1× better on the reconfigurable core (with only 2% worse performance on the reconfigurable core). The improved flexibility provided by synthesizing kernels partially at run-time improved application performance by 1.3× on average.

As part of this work, a multi-core reconfigurable processor was designed and implemented as an FPGA prototype, as well as successfully integrated into the heterogeneous many-core developed as part of the Invasive Computing project.

Zusammenfassung

Die Ausführungsgeschwindigkeit von Anwendungen mit rechenintensiven Programmabschnitten ("Kerneln") kann verbessert werden, indem sie auf Prozessoren ausgeführt werden, welche durch anwendungsspezifische Hardware-Beschleuniger erweitert sind. Um eine möglichst grosse Auswahl an Anwendungen zu beschleunigen, werden laufzeitrekonfigurierbare Beschleuniger eingesetzt, in Form einer sogenannten rekonfigurierbaren "Fabric", wie sie z.B. in einem FPGA vorzufinden ist. Jede Anwendung kann ihren eigenen anwendungsspezifischen Beschleuniger auf die Fabric rekonfigurieren, um erfährt so einen Zuwachs in der Ausführungsgeschwindigkeit. Solche *rekonfigurierbaren Prozessoren* wurden bereits in Forschung und Industrie untersucht und eingesetzt.

Beim Einsatz von rekonfigurierbaren Prozessoren in *Multi-Tasking* Szenarien (d.h. mehrere lauffähige Anwendungen) und *Mehrkernsystemen* finden sich jedoch mehrere Probleme beim Stand der Technik. Spezialisierte Task Scheduler sind für rekonfigurierbare Systeme mit eingeschränkter Flexibilität entwickelt: entweder blockieren solche Systeme die Anwendungsausführung während der Rekonfiguration, oder ein Kernel kann in nur einer Variante auf der Fabric ausgeführt werden, und das System muss auf die deutlich langsamere Ausführung auf dem Prozessorkern zurückgreifen, wenn diese Variante nicht eingesetzt werden kann. Flexible rekonfigurierbare Prozessoren, die solche Einschränkungen nicht haben, benutzten klassische Task Scheduler, die für nicht-rekonfigurierbare Prozessoren entwickelt wurden. Der Nachteil von rekonfigurierbaren Prozessoren, die lange Rekonfigurationszeit der Beschleuniger, wird von solchen Task Schedulern ignoriert, was sich in einer Geschwindigkeitsverringerung bei den Anwendungen auswirkt. Bestehende Ansätze für den Einsatz einer Fabric in Mehrkernsystemen können in zwei Klassen aufgeteilt werden: solche, bei denen alle Kerne auf eine gemeinsame Fabric zugreifen können und solche, bei denen eine dedizierte Fabric an jeden Kern gekoppelt ist. Beide Klassen haben Effizienzprobleme bei der Nutzung von Ressourcen auf der Fabric. Systeme mit einer dedizierten Fabric können sich nicht (oder nur stark eingeschränkt) an dynamische Anwendungsszenarien anpassen, während bei Systemen mit einer gemeinsamen Fabric die Möglichkeit zur gleichzeitigen Ausführung von mehreren Kerneln nicht oder nur stark eingeschränkt vorhanden ist.

Das Ziel dieser Arbeit ist der Entwurf von Konzepten für die effiziente Nutzung von rekonfigurierbaren Prozessoren in Multi-Tasking Szenarien und Mehrkernsystemen. Im Einzelnen sind das:

Task Scheduling für rekonfigurierbare Prozessoren
Der Nachteil von rekonfigurierbaren Prozessoren ist die lange Dauer um einen Beschle-

uniger zu rekonfigurieren, was sich negativ auf die Anwendungsgeschwindigkeit auswirkt. In Multi-Tasking Szenarien kann ein Task Scheduler diese Rekonfigurationszeit "verstecken". Dafür wird zuerst der Begriff der Task Effizienz eingeführt, um zu beschreiben wie gut eine laufende Anwendung vom derzeitigen Zustands des rekonfigurierbaren Prozessors profitiert (abhängig von den derzeitig auf der Fabric rekonfigurierten Beschleunigern im Verhältnis zu den von der Anwendung gewünschten Beschleunigern). Aufbauend auf der Task Effizienz wird ein Scheduler für Szenarien mit Deadlines entwickelt, mit dem Ziel diese Deadlines besser einzuhalten. Weiterhin wird ein gemeinsamer Ansatz für Task Scheduling und Allokation von Fabric an Anwendungen vorgestellt, mit dem Ziel den Makespan (d.h. die früheste Zeit zu der alle Anwendungen beendet sind) zu reduzieren. Beide vorgestellten Scheduler erkennen wenn die Task Effizienz einer Anwendung einbricht, unterbrechen diese, und lassen stattdessen eine andere Anwendung laufen, bis die sich die Task Effizienz der ersten Anwendung verbessert. Dieser Ansatz ergibt eine Verbesserung der Geschwindigkeit des gesamten Systems.

Gemeinsame Nutzung der Fabric in einem Mehrkernsystem
Kernel, die auf der Fabric ausgeführt werden, nutzen nur einen Bruchteil der auf der Fabric zur Verfügung stehenden Ressourcen (wie z.B. Beschleuniger, Fabric-interner Speicher, Speicherbandbreite zu externem Speicher) während ihrer Laufzeit. Die Ausführung eines einzelnen Kernels auf der Fabric hat also ungenutzte Fabric Ressourcen zur Folge. In einem Mehrkernsystem können diese Ressourcen genutzt werden, um Kernel parallel auf der Fabric laufen zu lassen. Eine neue Art um die Fabric gemeinsam zu nutzen wird vorgestellt, welche die Nachteile von existierenden Techniken behebt. Es wird eine heterogene Mehrkernarchitektur vorgeschlagen, bestehend aus einem rekonfigurierbaren und mehreren nichtrekonfigurierbaren Kernen (welche an die Fabric des rekonfigurierbaren Kerns angebunden sind). Die Fabric kann also von allen Kernen genutzt werden, Zugriffe vom rekonfigurierbaren Kern werden dabei priorisiert, was es ihm erlaubt, die anspruchsvollsten Anwendungen auszuführen. Die gleichzeitige Ausführung von mehreren Kerneln in der Fabric wird durch das Konzept des Verschmelzens von Fabric-Zugriffen ermöglicht, was durch eine Hardware-Erweiterung realisiert wird.

Flexible Nutzung der Fabric in Multi-Tasking- und Mehrkernsystemen
Die Nutzung der Fabric in Multi-Tasking- und Mehrkernsystemen (wie oben beschrieben) mit dynamischen Anwendungsszenarien (d.h. es ist im Voraus nicht bekannt, welche Anwendungen laufen werden und wann sie gestartet werden) erfordert, dass Kernel von unterschiedlichen Anwendungen auf der Fabric ausgeführt werden können, unabhängig wieviel und welche Teile der Fabric einer Anwendung zugewiesen wurden. Dieses hohe Maß an Flexibilität kann nicht erreicht werden, wenn die Kernel nicht zur Laufzeit synthetisiert werden, da die Entscheidung welche Teile der Fabric an eine laufende Anwendung zugewiesen werden für dynamische Anwendungsszenarien auch zur Laufzeit gemacht wird. Andererseits ist die vollständige Synthese der Kernel zur Laufzeit sehr zeitaufwändig, weshalb ein Ansatz vorgeschlagen wird, bei dem nur derjenige Teil der Kernelsynthese, der von der Aufteilung der Fabric abhängt zur Laufzeit durchgeführt wird, während der Rest der Synthese zur Compile-Zeit stattfindet. Um den Overhead zu reduzieren, wird ein Software-Cache für synthetisierte

Kernel und eine Methode zur Cache-freundlichen Rekonfiguration der Beschleuniger entwickelt.

Bei der Auswertung zeigt sich, dass der vorgestellte Scheduler zum besseren Einhalten von Deadlines ein 1.22× besseres Ergebnis in der kumulativen Deadline-Überschreitung als der nächstbeste Scheduler erreicht. Der kombinierte Scheduler und Fabric Allokator erreicht Makespans, welche nur 6% von der unteren Schranke entfernt sind, während andere Ansätze 12% bis 20% schlechter als die untere Schranke sind. In Mehrkernsystemen ergab die Technik zur gemeinsamen Nutzung der Fabric eine bessere Geschwindigkeit als andere Ansätze: 1.3× schneller auf den nicht-rekonfigurierbaren Kernen (bei gleicher Geschwindigkeit auf dem rekonfigurierbaren Kern), oder 3.1× schneller auf dem rekonfigurierbaren Kern (bei nur 2% Geschwindigkeitsverlust auf dem rekonfigurierbaren Kern). Die verbesserte Flexibilität, die durch die Synthese der Kernel zur Compile- und Laufzeit erreicht wurde, spiegelte sich in einer Verbesserung der Geschwindigkeit um 1.3×.

Als Teil dieser Arbeit wurde ein Mehrkernprozessor mit einem rekonfigurierbaren Kern entworfen und als Prototyp auf einem FPGA implementiert. Zusätzlich wurde dieser rekonfigurierbare Prozessor in die heterogene Vielkernarchitektur des Projektes “Invasives Rechnen” integriert.

List of Own Publications

Publications Providing Major Contributions to this Thesis

[GBH16] A. Grudnitsky, L. Bauer, and J. Henkel. "Efficient Partial Online Synthesis of Special Instructions for Reconfigurable Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2016). forthcoming.

[GBH14a] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "COREFAB: Concurrent Reconfigurable Fabric Utilization in Heterogeneous Multi-Core Systems". In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. Best Paper Nomination. 2014.

[GBH14b] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "MORP: Makespan Optimization for Processors with an Embedded Reconfigurable Fabric". In: *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. 2014, pp. 127–136.

[BGSH12] Lars Bauer, Artjom Grudnitsky, Muhammad Shafique, and Jörg Henkel. "PATS: a Performance Aware Task Scheduler for Runtime Reconfigurable Processors". In: *Field-Programmable Custom Computing Machines (FCCM)*. European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) Paper Award. 2012, pp. 208–215.

[GBH12] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "Partial Online-Synthesis for Mixed-Grained Reconfigurable Architectures". In: *Design Automation and Test in Europe Conference (DATE)*. 2012, pp. 1555–1560.

Note:

Preliminary work on algorithms for placement and binding (Sections 5.4 and 5.5) was done earlier as part of the author's diploma thesis [Gru09] in the context of a single-task reconfigurable processor. As part of this PhD thesis this work was extended for multi-tasking, a Linux-based run-time system with placement and binding support capable of serving multi-tasking scenarios was implemented, and measurements of this implementation on an FPGA prototype of a reconfigurable processor lead to the design of a μProgram caching system and the cache-aware placement algorithm (Section 5.6).

[Gru09] Artjom Grudnitsky. "Extending the RISPP Simulator with Floorplan-Aware Special Instruction Management and Execution". Diploma Thesis. 2009.

Publications Providing Minor Contributions to this Thesis

[BGDK+15] Lars Bauer, Artjom Grudnitsky, Marvin Damschen, Srinivas Rao Kerekare, and Jörg Henkel. "Floating Point Acceleration for Stream Processing Applications in Dynamically Reconfigurable Processors". In: *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. Invited Paper for the Special Session "Dynamics and Predictability in Stream Processing – A Contradiction?" 2015.

[PSOE+15] Johny Paul, Walter Stechele, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, Wolfgang Schröder-Preikschat, Manfred Kröhnert, Tamim Asfour, Ericles Sousa, Vahid Lari, Frank Hannig, Jürgen Teich, Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "Resource Awareness on Heterogeneous MPSoCs for Image Processing". In: *Journal of Systems Architecture*. Vol. 61. 10. 2015, pp. 668–680.

[MGMB+13] Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, and Jörg Henkel. "Hardware Acceleration for Programs in SSA Form". In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2013.

[HHBW+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R.K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and Kobbe S. "Invasive Manycore Architectures". In: *16th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Invited Paper for the Special Session "Design and Prototyping of Invasive MPSoC Architectures". 2012, pp. 193–200.

[HBHG11] Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky. "i-Core: A run-time adaptive processor for embedded multi-core systems". In: *Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Invited Paper for the Special Session "Runtime Adaptive Embedded Systems and Architectures". 2011.

List of Supervised Student Works

[Mai16] Eduard Maier. "Design of pipelined floating point accelerators for a reconfigurable processor". Studienarbeit. 2016.

[Qua15] Thorsten Quadt. "Operating System Support for Heterogeneous Runtime Reconfigurable Multi-Core Processors". Diploma Thesis. 2015.

[Rie15] Martin Riedlberger. "Erweiterung des i-Core Prozessors und Portierung auf die VC707 Enwicklungsplattform". Studienarbeit. 2015.

[Bla14] Matthias Blankertz. "Exploration of reconfigurable accelerators on the Zynq architecture". Studienarbeit. 2014.

[Hel13] David Hellmeister. "Entwicklung eines parallelen Laufzeitsystems zur dynamischen Rekonfiguration des i-Core Prozessors". Diploma Thesis. 2013.

[Mod13] Tobias Modschiedler. "Erweiterung der LEON3-CPU um einen Permutationsregistersatz zum beschleunigten Abbau der SSA-Zwischendarstellung". Diploma Thesis. 2013.

[Rie13] Martin Riedlberger. "Entwicklung des rekonfigurierbaren Prozessors i-Core auf Basis des LEON3 Systems". Diploma Thesis. 2013.

[Typ12] Marc Typke. "Achieving behavioral Accuracy on a reconfigurable Processor Simulator". Bachelor Thesis. 2012.

Note:

Supervised student works were done in the scope of building and maintaining the prototype and simulator of the reconfigurable processor used in this thesis.

Acknowledgments

I want to thank my advisor, Prof. Jörg Henkel for offering me the opportunity to pursue a Ph.D. under his guidance. He gave me a great amount of freedom in my research, while his feedback and comments continuously helped me improve my work. I would also like to thank my co-advisor, Prof. Jürgen Becker for his valuable comments.

Lars Bauer deserves a great amount of my gratitude for discussions and feedback which have positively influenced my work.

I want to thank my colleagues and friends with whom I worked during the past years: Thomas Ebi, Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Benjamin Vogel, Volker Wenzel, Hongyan Zhang and other members of the CES and the other chairs of the Institut für Technische Informatik (ITEC). Marvin Damschen, Srinivas Kerekare and Lars Bauer deserve my special gratitude for providing comments for my thesis. I would like to thank Muhammad Shafique for his comments during rehearsals of my paper presentations.

Colleagues from other chairs whom I want to thank for fruitful (and/or just fun) discussions are: Jan Micha Borrmann (FZI), Sebastian Buchwald (IPD), Manfred Kröhnert (HIS), Manuel Mohr (IPD) and Carsten Tradowsky (ITIV/FZI).

Over the past years I supervised multiple student works (details on page xii). I would like to thank all these students for their contributions and hard work.

Finally, I want to dedicate this thesis to my parents and my grandfather. Their continued support has strengthened my resolve and helped me master the challenges of the past few years.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Performance improvement in the micro-processor industry is driven by two main factors: improvements in processor micro-architecture and advancements in chip fabrication technology.

Fabrication technology has been continuously improving, leading to doubling of transistors per die every 18 months (*Moore's Law*), with Moore's observations starting in 1959 [Moo98] and the trend continuing into the 2010s. However, during a discussion of their quarterly results in 2015, Intel stated that transistor doubling has slowed down to every 2 – 2.5 years [Cla15], indicating that while fabrication technology will likely continue to improve, it will take increasingly longer to reach a new technology node. Additionally, costs for transitioning the fabrication process to smaller technology nodes are very high, as new process steps (e.g. multi patterning) have to be introduced, requiring new machinery and increasing the time to fabricate a batch of wafers.

Processor micro-architecture has been another driving force in increasing processor performance, as observed by *Pollack's Rule*, which states that "performance increase due to micro-architecture advances is roughly proportional to the square root of the increase in complexity". However, advancements in single-core performance have also been slowing down, as techniques that have yielded the biggest gains (e.g. pipelining, branch prediction, out-of-order execution) have already been implemented and according to focus has shifted to mitigating issues arising from transistor scaling, such as power efficiency, reducing thermal issues, etc.

Industry has dealt with this slowdown of improvement in single-core performance by increasing the number of cores per chip, resulting in multi-core systems – starting with dual-core CPUs introduced by Intel and AMD in 2005 for desktop computing (for mobile computing the first mobile dual-core chipsets were by Nvidia, deployed by LG in 2011). Multi-cores are an evolution of multi-processor systems, where multiple single-core processors were placed on one circuit board and communicated using an off-chip interconnect. Integrating multiple cores (along with additional peripherals) as a System-on-Chip (SoC) allows for lower communication latency, power usage and more area-efficient devices.

Specialized multi-core systems available today have 10s of cores (e.g. 72 64-bit RISC Cores on the EZchip (formerly: Tilera) TILE-Gx72[EZc15]) and the International

Technology Roadmap for Semiconductors (ITRS) predicts that this trend will continue, nearing 1000 cores on a chip by the mid 2020s [ITR11]. An increase in the number of cores does not necessarily result in better performance, which depends on the type of parallelism inherent in the applications that run on the system. Architectures such as superscalar cores exploit Instruction-Level Parallelism (ILP), i.e. the parallism between instructions in close proximity within one application.

Multi-cores are designed to benefit applications exhibiting the more coarse-grained Task-Level Parallelism (TLP), i.e. parallelism between different tasks of an application. TLP of a typical desktop application is not very high, as shown in [BDMF10], with most applications utilizing only 2 cores in an 8 core system, with only a video authoring application utilizing all 8. The authors of [BBFB07] come to a similar conclusion, showing that most workloads do not exhibits a high degree of parallelism. This suggests that while using more than 1 core does improve performance for most applications, going beyond 2 or 4 core systems only benefits few specialized applications. Some systems also exhibit "bursts" of activity, resulting in a high degree of parallelism for a limited amount of time. For example, mobile systems such as smartphones often have a similar activity pattern, with the device idle most of the time, but under heavy load during user interaction, especially when using demanding applications, such as games or graphics intensive applications [RC12].

Another architectural trend has been the introduction of *heterogeneity* into SoCs. Here, additional specialized processing cores are integrated into the SoC, such as cryptography accelerators, graphics engines, etc. Heterogeneity is not only used for high performance, but also for energy-efficient computation as discussed in [Mit15], where the most energy efficient core is selected for an application, and the other cores can be powered down. Mobile chipsets are usually heterogeneous multi-cores, such as the Qualcomm Snapdragon 805 with 4 ARM cores, a GPU and a VLIW core. In contrast, homogeneous multi-core processors consist of identical cores, e.g. such as common server/desktop multi-cores, although some of the latter started including GPUs as part of the SoC. [BC11] predicts that future architectures will favor heterogeneous multi-cores to cope with rising energy/power issues of multi-core systems.

The specialized components (such as GPUs) in a heterogeneous multi-core processor improve performance and/or power-efficiency compared to a GPP for specific application domains, but are less useful for domains that were not designed for. For example, an accelerator such as a GPU will achieve significant speedup for streaming applications, but will not benefit kernels that operate on smaller data sets and are part of control-flow dominant code, where a GPP with custom accelerators would be better suited. Therefore, a heterogeneous system is not always superior to a regular homogeneous multi-core, depending on the actual workload of the system. Using a type of specialized core not well suited for the targeted scenario will waste silicon area, reduce power-efficiency and performance. An SoC designer will therefore need to have a clear understanding of the applications that will be run on the system throughout its lifetime, in order to choose the right type of heterogeneity. Some systems, such as server CPUs are used in a very wide range of application

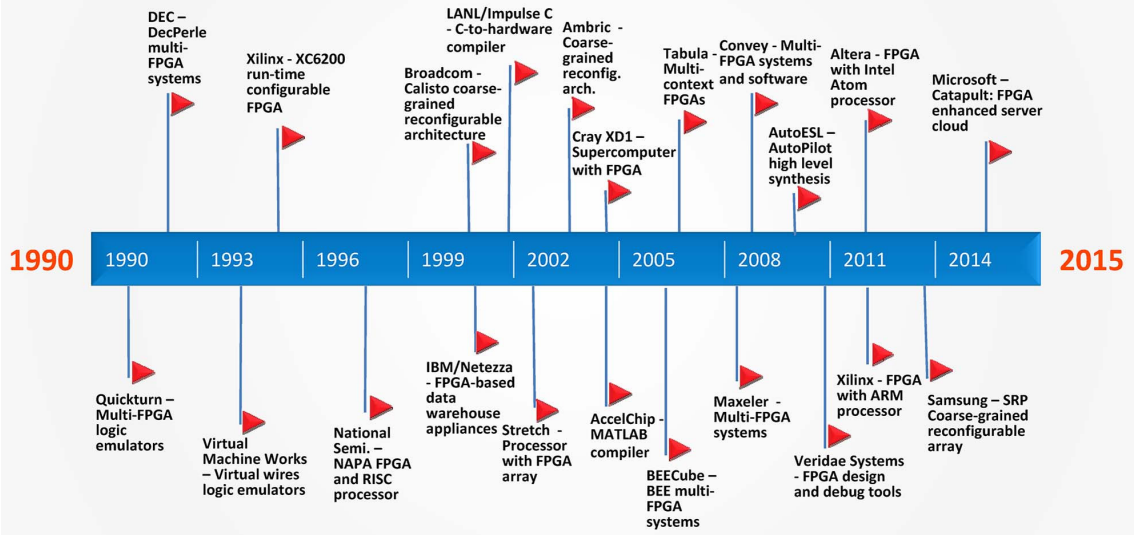


Figure 1.1: Timeline of commercial reconfigurable systems (from [TPD15]).

domains, with different requirements, e.g. database clusters need high bandwidth, weather forecasting requires high performance in floating point computations, while high-frequency trading systems need very low latency. Such widely varying (and sometimes conflicting) requirements make the optimal choice of a general-purpose heterogeneous architecture difficult or impossible.

Reconfigurable Processors

A way to address these varying requirements is by using hardware capable of run-time reconfiguration. Reconfigurable processors provide a high degree of flexibility by adapting their hardware to the needs of an application at run-time. A reconfigurable processor is based on a GPP, extending it by a reconfigurable fabric, a special area designed to house application-specific hardware accelerators. Accelerators implement control- and data-flow of a computationally intensive software part (i.e. kernel) as a hardware data-path. An application running on the reconfigurable processor can reconfigure an accelerator for its current (or future) kernel. The kernel will then be executed on the reconfigurable fabric (i.e. *in hardware*) using the appropriate accelerators, instead of on the GPP core (i.e. *in software*), thereby improving execution speed and/or energy efficiency.

Unlike the specialized cores in a non-reconfigurable system, reconfigurable accelerators can be designed and deployed after the system has been fabricated, shipped and is already in use. New accelerators can be part of an application or a third-party software library (e.g. a library for scientific computation may include accelerators for matrix multiplication). The coupling between the fabric and the GPP core of a reconfigurable processor, the fabric architecture and the interface between an application and the fabric are some of the design choices for a reconfigurable processor (see Sections 2.1 to 2.3 and 2.6 for details).

Figure 1.1 highlights some commercial efforts in reconfigurable computing in the past 25 years, as discussed in the survey [TPD15]. While early efforts designed stand-alone devices for specialized uses, recent developments focused on integrating reconfigurable fabrics with general-purpose processors and use of such architectures in fields such as cloud computing.

Some examples of kernels that can be accelerated on reconfigurable processors are from the domains of cryptography (e.g. AES encryption, SHA hashing), video/audio coding (e.g. motion estimation, AdPCM encoding), scientific computing (e.g. matrix multiplication), data mining (e.g. k-means clustering). Achieved speedups vary depending on the kernel, the architecture of the reconfigurable processor and the accelerator, with example speedups being in the range of 3.6×–7.2× for video encoding applications [MBTC06; BSH08b; LSV06]. The fabric area required by different applications varies, as it depends on the complexity of their accelerators. For example, an accelerator for JPEG encoding requires 5× the area of an accelerator for SHA hashing [LSC09]. The same kernel can often be implemented in different accelerator variants, with larger variants exploiting more of the parallelism inherent in a kernel. This allows for an area/performance trade-off.

As with other types of computational elements (GPP, GPU, etc.), the reconfigurable fabric is a resource, providing benefits in the form of increased performance or better power-efficiency compared to using a different type of computational element, but also incurring costs in the form of increased silicon area and static power consumption, even when the fabric is not in use. Therefore, in order to gain benefit from the fabric (and therefore justify the use of reconfigurable processors for general-purpose computing), its utilization should be as high as possible, thereby increasing the value of the reconfigurable processor to the SoC. Otherwise, the flexibility provided by the reconfigurable processor may be outweighed by its disadvantages, and the system designer may be better off using a different type of core, even if it provides worse performance or only accelerates a fraction of the applications intended for the system.

If the reconfigurable processor is limited to executing one task at a time, the possibilities for efficient fabric utilization are limited. Tasks that only use one kernel will usually load their required accelerators at start, and the fabric will not be reconfigured until the task terminates, thus any fabric resources not used by the task will remain idle over the task duration and contribute to inefficient fabric use. More complex tasks have multiple kernels, and will thus repeatedly switch between accelerators, depending on the execution path taken (which in turn may depend on input data). Even here, some execution paths will require less reconfigurable area than others, thus in general the fabric will not be fully utilized. A naïve method to increase utilization would be to reduce fabric size until for any application only the smallest accelerators fit on the fabric. Even then the smallest accelerators of two different applications are likely to have different area requirements. Furthermore, using the smallest (and thereby slowest) accelerators may not meet the required performance.

Apart from inefficient fabric use, another drawback of reconfigurable processors is the long time to reconfigure an accelerator. This problem only occurs in some architectures, but those are the most flexible ones (fine-grained reconfigurable, see also Section 2.1.2). During the time that a reconfiguration is performed, task execution speed is very low, thus applications that perform reconfigurations often (e.g. due to input-data dependent behavior switching between different kernels) may not receive the performance required by using the fabric.

Summarizing the issues:

- For desktop, mobile and some server systems simply increasing the amount of cores in a multi-core system will not provide performance benefits due to TLP limits of typical workloads.
- Heterogeneous components in multi-cores are usually too specialized, and benefit only few applications.
- Reconfigurable cores provide the flexibility to accelerate a large range of applications, but to compete with non-reconfigurable components their reconfigurable fabric needs to be utilized as much as possible and the performance degradation due to reconfiguration needs to be reduced.

This thesis will demonstrate will address these issues by showing that even the relatively small degree of parallelism present in common multi-tasking scenarios is sufficient to utilize the fabric of a reconfigurable core efficiently. This is done by *sharing the fabric resources* between different tasks on the same core and/or between different cores.

1.2 Thesis Contribution

The goal of this thesis is to show how reconfigurable processors can be used to improve performance of modern multi-core platforms. To do so, the drawbacks of reconfigurable processors need to be addressed: inefficient fabric use and performance degradation due to long reconfiguration time. This thesis proposes new task-scheduling and fabric allocation approaches to allow efficient multi-tasking and a new way to share the fabric in a multi-core in order to reduce fabric inefficiency and improve performance.

In detail, the contributions of this thesis are:

Task scheduling for reconfigurable processors Existing task scheduling approaches either assume static workloads (i.e. workloads that are fixed at compile-time), are designed for inflexible architectures that do not provide the required degree of adaptivity to handle dynamic workloads well, or use classic task schedulers that do not exploit the adaptivity of reconfigurable processors. The proposed metric of task efficiency is introduced in order to capture how well a task benefits from the fabric

at a given point in time. Task efficiency changes during run-time of an application, and the goal is to keep task efficiency high to improve system performance. Two task schedulers are proposed, based on this metric: one for tasksets with deadlines, resulting in the ability to meet tighter deadlines, that could not be met with other schedulers, and one optimized for tasks without deadlines, which improves completion time of tasksets.

Task scheduling is discussed in Chapter 3.

Using the reconfigurable fabric efficiently in a multi-core Fabric resources include area for reconfigurable accelerators, bandwidth between the fabric and the memory hierarchy, etc. Kernels have different characteristics, such as being memory-bound or computationally bound, thus some kernels use more of one fabric resource, while neglecting others. Such characteristics can vary even within one kernel (e.g. memory-bound phase at start or end of the kernel to fetch/write data), thus throughout kernel execution not all available fabric resources are utilized fully. To address this inefficient use of fabric resources, an approach to share the fabric in a multi-core is proposed. Non-reconfigurable cores are granted access to the fabric of the reconfigurable core. State-of-the-art approaches either use homogeneous multi-cores, where either each core is connected to its own fabric which limits the flexibility for a core to use more or less fabric resources, depending on application demands. Other architectures have all cores connected to one fabric, but they are incapable of using the fabric in parallel.

The approach proposed in this thesis aims to combine the advantages of both types of state-of-the-art architectures, while mitigating their drawbacks. The proposed architecture is a *heterogeneous* multi-core, where GPP cores (with lightweight modifications) are connected to the fabric of the reconfigurable core, providing the flexibility of a shared fabric. Multiple kernels (dispatched by different cores) can execute on the fabric concurrently, thus a memory-bound and computationally bound kernel can be run together, allowing fabric resources to be used more efficiently. This is done through merging of fabric accesses performed by multiple cores. By doing this directly in hardware on a cycle-by-cycle basis, the impact of performance degrading conflicts is kept low, and overall multi-core performance is improved.

Sharing reconfigurable fabric in a multi-core is discussed in Chapter 4.

Flexible fabric use in multi-tasking and multi-core systems Sharing the reconfigurable fabric between different tasks (on a single core using multi-tasking or in a multi-core) requires a high degree of flexibility of how kernels can be executed. At any time, a new task can enter or leave the system, taking away or giving back fabric area from an already running task. In order to make use of this constantly changing fabric share efficiently, the task can not have the kernel generated for fabric execution at compile-time. Instead, the kernel generation is split between compile-time and run-time: scheduling of the kernel is done at compile time, but binding the kernel to a fabric share that is currently available to the task is done at run-time. In order

to reduce the overhead of the run-time part of kernel generation, a software cache and a cache-aware accelerator placement algorithm (which is responsible for loading accelerator into the fabric share of a task) is proposed.

The approach for flexible fabric use is discussed in Chapter 5.

These contributions were evaluated on the*i*-Core reconfigurable processor, which was designed in the scope of this thesis. Furthermore, the *i*-Core was integrated into the heterogeneous many-core architecture developed in the scope of the Invasive Computing project. FPGA prototype demonstrators of both the Invasive Computing many-core system with the *i*-Core as well as a stand-alone *i*-Core-based multi-core were implemented (the implementation of the many-core system was a group project in cooperation with other chairs from the universities KIT, TUM and FAU).

1.3 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 provides background knowledge and discusses related work. After introducing the concept of reconfiguration, an overview of existing single-core reconfigurable processors is provided, with a more detailed discussion of fabric architectures. The concept of Special Instructions, an interface between the application and the reconfigurable fabric is presented along with the run-time system that manages the reconfigurable fabric to make these Special Instructions runnable on the fabric. Next, existing reconfigurable multi-core systems are discussed, followed by an introduction to task scheduling and an overview of task scheduling techniques for reconfigurable processors. Finally, the *i*-Core reconfigurable processor is presented, which is used to illustrate the concepts of this thesis in later chapters.

The contribution of using a reconfigurable processors for efficient task scheduling is presented in Chapter 3. The challenges of increasing task throughput and meeting soft realtime constraints are discussed, along with the respective metrics of *makespan* and *tardiness*. A motivational example illustrates how a reconfigurable processor can exploit the effect of reconfiguration hiding to create better schedules. Next, two schedulers are presented, focussing on reducing tardiness and makespan, respectively.

The next part of the contribution is the efficient use of the fabric in a multi-core reconfigurable processor, discussed in Chapter 4. The disadvantages of existing reconfigurable multi-core systems are summarized, resulting in the problem of fabric underutilization. The concept of merging fabric accesses is introduced, which is central to reducing underutilization. The hardware components to provide access to the fabric by all cores in a multi-core system and to implement access merging are then presented. Merging conflicts, an issue that reduces performance is then analyzed and a way to reduce conflict occurrences is presented.

In order to allow for the largest amount of flexibility in respect to fabric use, kernels intended for execution on the fabric should not be entirely generated at compile time. Chapter 5 presents a way to offload part of this kernel generation to run-time, resulting in a more efficient use of the fabric in multi-tasking and multi-core systems. The problems of accelerator placement, binding of the kernel to the fabric and caching of the result for overhead reduction are discussed and solved.

The proposed approaches are evaluated in Chapter 6, analyzing each individual contribution by itself, as well as performing a joint evaluation.

Chapter 7 concludes the thesis with a summary and promising future research directions.

2 Background and Related Work

The focus of this thesis are reconfigurable processors in multi-tasking scenarios and multi-core systems. This chapter provides the background necessary for the following technical chapters and discusses related work. An overview of reconfigurable systems is provided in Section 2.1. Sections 2.2 and 2.3 discuss the interface between applications and the fabric (relevant for all following chapters) and how the fabric can be managed by a run-time system (relevant for Chapter 5). Section 2.4 provides an overview of reconfigurable multi-core systems, related to the content of Chapter 4. A task-scheduling overview (related to the content of Chapter 3), including techniques for reconfigurable processors is provided in Section 2.5. The concepts presented in the later chapters are illustrated using the *i*-Core reconfigurable processor, which was developed in the scope of this thesis and is introduced in Section 2.6.

2.1 Reconfigurable Systems

Reconfigurable processors allow adaption of their own hardware to the requirements of an application. While this hardware adaption has been previously done in Application-Specific Instruction Set Processors (ASIPs, [KMN02]) at design-time, a reconfigurable processor adapts at run-time. This benefits dynamic workload scenarios, where the composition of the workload and the behavior of the individual applications depends on run-time factors (such as different input data or user interaction).

The defining feature of reconfigurable processors is the ability to reconfigure parts or the whole architecture after the system has been designed and deployed. For this thesis, reconfigurable accelerators are of particular interest, while other parts of the processor architecture (such as the pipeline, caches and bus) are assumed to be static, i.e. non-reconfigurable. The component of the processor where accelerators can be reconfigured (or "loaded") is called the *reconfigurable fabric*.

While different types of configuration exist (such as one-time configuration performed after device fabrication as part of customization of the device), in the scope of this work, the fabric is assumed to have no limit on the amount of reconfigurations, and reconfiguration does not require any specialized external devices (e.g. EPROMs memory is erased using a UV light source). Furthermore, the fabric is assumed to be *run-time reconfigurable*, i.e. a new configuration can be loaded while the rest of the processor is running. Most of the fabrics described in Sections 2.1.1 and 2.1.2 also have the property of being *partially run-time reconfigurable*, which allows only a part of the fabric to be reconfigured, while the remainder of the fabric is in use. This

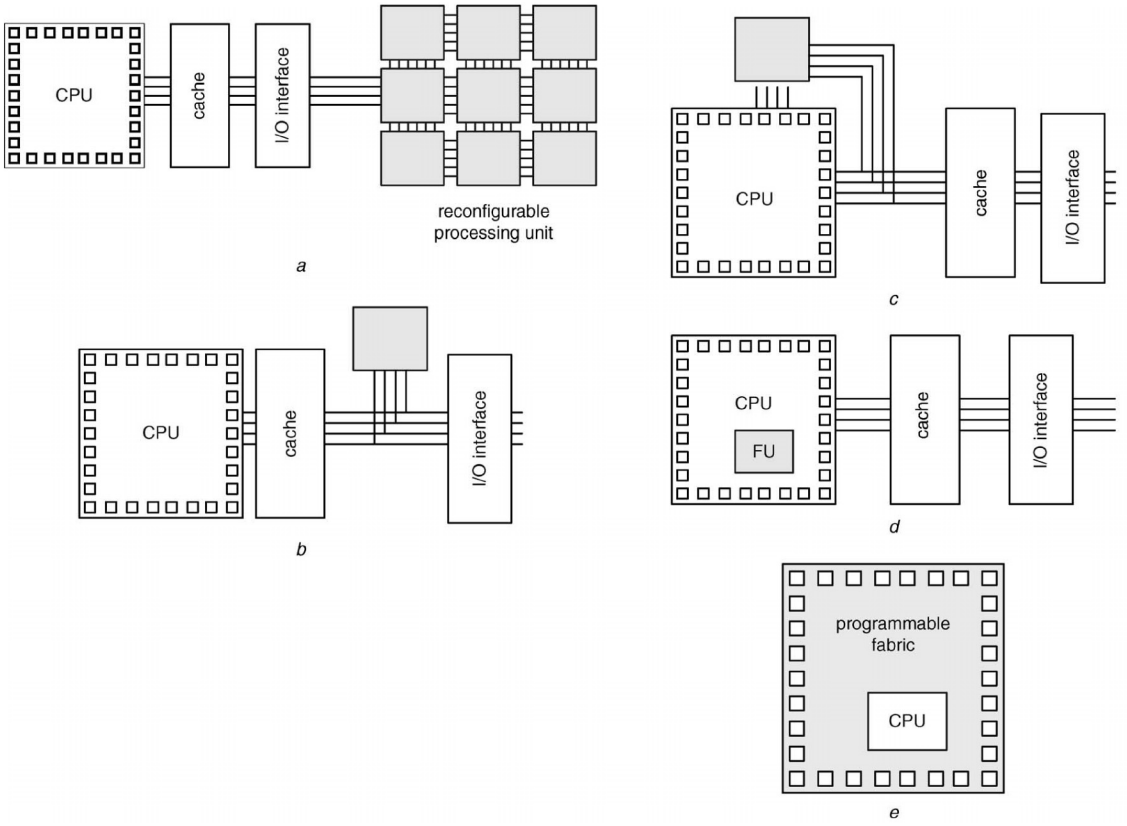


Figure 2.1: Different types of coupling (from [TCWM+05]). a) External stand-alone processing unit. b) Attached processing unit. c) Co-processor. d) Reconfigurable functional unit. e) Processor embedded in reconfigurable fabric.

capability allows the system to reconfigure a new accelerator onto the fabric for future use, while during this reconfiguration multiple other already loaded accelerators are used to accelerate a currently running application.

The non-fabric part of the processor is the general-purpose processor (GPP) core[1]. Details of the GPP core are outside of the scope of this work, background knowledge for processor architecture can be found in [HP11].

The coupling of fabric and GPP core is an important design parameter of the reconfigurable system. Figure 2.1 shows the 5 types of coupling established in [TCWM+05]. For the remainder of this chapter the term *tight coupling* will be used to refer to class d) "Reconfigurable functional unit", where the fabric is integrated into the pipeline of the GPP core similar to an ALU or a dedicated divider. The GPP core can access a tightly coupled fabric with a 0 cycle latency (same as with any other functional unit that is part of the GPP core). The term *loose coupling* will be used to refer to classes a)–c), where fabric access latency is usually 100s of cycles or more. Tightly coupled fabrics require modification of the GPP core, which

[1] A GPP is assumed, although specialized processors, such as DSPs may be used as well

may not be an option for a system designer (if the core is only available as a "black box" which can/may not be modified, or the GPP core is provided with guarantees regarding its behavior which would be voided if modification were performed).

Independent of the coupling, a design characteristic of the fabric is granularity. Granularity specifies at which level (e.g. bit, byte, word) operations are performed. Coarse-grained fabrics operate on words (e.g. of 32-bit length), similar to a GPP. Fine-grained fabrics are optimized for bit-level operations, allowing efficient implementation of operations such as bit-reversal in a word, which if implemented on a coarse-grained fabric would require multiple logic operations. On the other hand, implementing word-level operations is usually less efficient on fine-grained fabrics, as multiple bit-level processing elements need to be used, with additional routing resources connecting them.

Further information and overviews of reconfigurable cores are provided in [GCSB+06; VS07; HM09a].

2.1.1 Coarse-Grained Fabrics

Coarse-grained reconfigurable systems operate on data of word-level granularity. Possible operations are defined at design-time and are generally standard arithmetic/logical (e.g. boolean operations, integer addition/subtraction) as well as some specialized operations (e.g. filters, complex number arithmetic), depending on the target application. Processing elements (PEs) that implement these operations are usually arranged in a grid-like array, thus the name coarse-grained reconfigurable array (CGRA) is often used in literature. Interconnect between PEs depends on the specific architecture, but often a Manhattan-style interconnect (each PE connected to its immediate north, east, south, west neighbors) is used. The interconnect allows implementation of more complex kernels by combining multiple PEs. Despite the limited number of operations implemented on the PEs, a major advantage of coarse-grained arrays is reconfiguration speed: while a fine-grained reconfigurable fabric can take milliseconds to reconfigure an accelerator (depending on its size), a CGRA can usually reconfigure its PEs in one cycle. This is due to the fact that coarse-grained reconfiguration is more like "mode-switching" in an ALU, than providing completely new functionality, which is what reconfiguration for fine-grained fabrics does.

A designer considering a CGRA as an accelerator for a system-on-chip will have to weigh the efficient implementation of a limited amount of operations against the drawback of inefficient (or even impossible) implementation of kernels that consist of operations that are not supported on the PEs (e.g. bit-/byte- level operations such as bit-reversal in a word).

The Expression-Grained Reconfigurable Array (EGRA) is a heterogeneous CGRA proposed in [ABP11]. The idea is to use a domain- or application-specific customization of a CGRA at design-time by implementing specialized operations only in some designated PEs. This allows more efficient processing, as a complex operation can then be implemented on a single customized PE, instead of requiring multiple

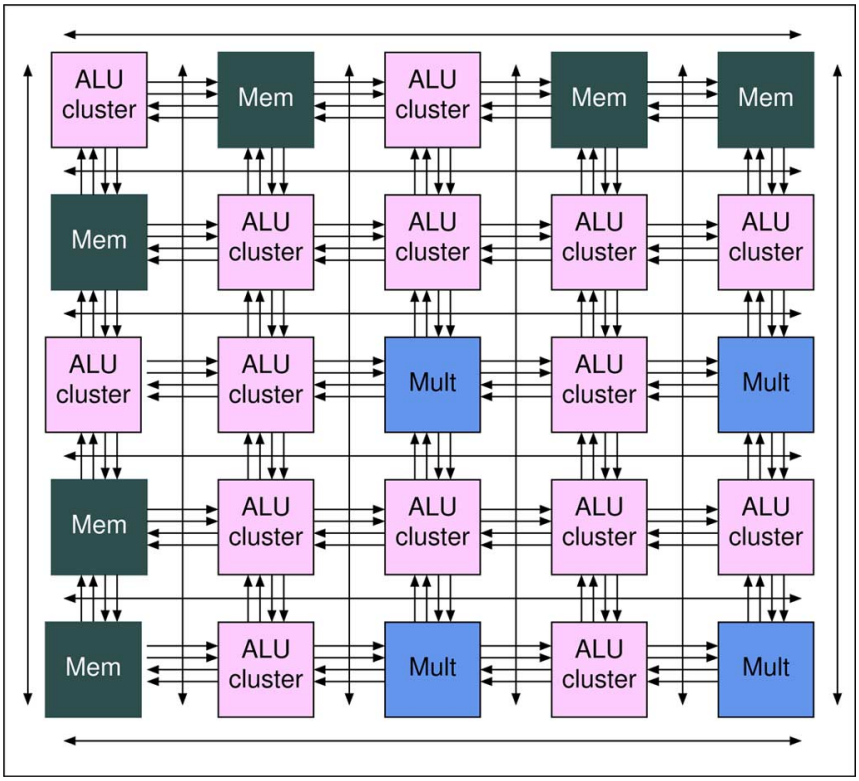


Figure 2.2: Example configuration of a EGRA heterogeneous coarse-grained architecture (from [ABP11]).

general PEs in a CGRA. A drawback is that during mapping a data-flow graph onto the EGRA, complex operation nodes are restricted to the designated PEs only. PEs are also quite powerful, as instead of a single ALU, a cluster of ALUs constitutes one PE, allowing processing of whole expressions (hence the name of the architecture). Figure 2.2 shows an example configuration of EGRA, with 4 PEs supporting only multiplication, while other PEs provide arithmetic/logic operations. Additional customization allows substituting PEs with dedicated memory cells for storing intermediate results and buffering input/output data within the EGRA.

The Samsung reconfigurable processor (SRP) (Figure 2.3) is based on a fabric based on that of the ADRES CGRA [MVVM+03; BBDG08]. The SRP has been used for low-power computing, such as mobile biomedical applications [KCCK+12] and high-performance scenarios, such as H.264 UHD Decoding [LSKK+11]. The SRP fabric can be used in either very long instruction word processor (VLIW) mode or CGRA mode, with mode switching supported at run-time. CGRA mode is intended for acceleration of loop bodies, with the loop body being scheduled to the PEs by the compiler. Different sizes of the CGRA can be used for computation (e.g. a low-power 2x2 configuration or a high-performance 3x3 configuration) with inactive PEs powered down. VLIW mode uses 2 PEs (2-issue VLIW) and is intended for control-flow dominant code sections. In addition to Manhattan-style connections, a PE is also connected to its diagonal neighbors.

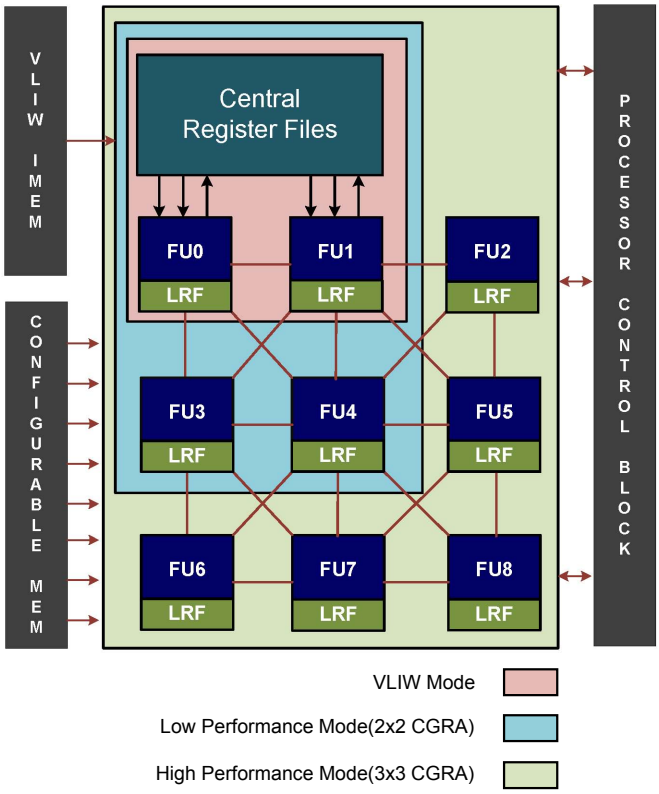


Figure 2.3: Ultra-Low Power Samsung Reconfigurable Processor with a 3x3 array of the coarse-grained ADRES fabric (from [KCCK+12]).

The Ambric MPPA (Massively Parallel Processor Array) [BJW07] uses RISC cores instead of ALU-like components as PEs. The cores are modified for streaming instructions and data over interconnect channels, instead of actively fetching data over a bus as in a regular multi-core. The PEs no longer operate in a lock-step manner, but instead are flow-controlled, with computations triggered by data availability on the channels. [PH11] proposes a hybrid architecture that consists of both MPPA PEs for implementing control-flow heavy code parts and regular CGRA PEs for implementing data-flow heavy code. Communication in such a hybrid is realized by implementing sufficiently large FIFO buffers at a channel end and stalling a producer PE if a consumer PE processes data at a slower rate.

2.1.2 Fine-Grained Fabrics

Fine-grained reconfigurable fabric are suited for processing of sub-word (bit- and byte-level) granularity. The basic building block for implementing operations is the look-up table (LUT). A N:M LUT can implement an arbitrary boolean function with N inputs and M outputs, for example a 4:1 LUT can be used to implement a boolean 4-input AND function. Multiple LUTs can be combined to implement more complex functions.

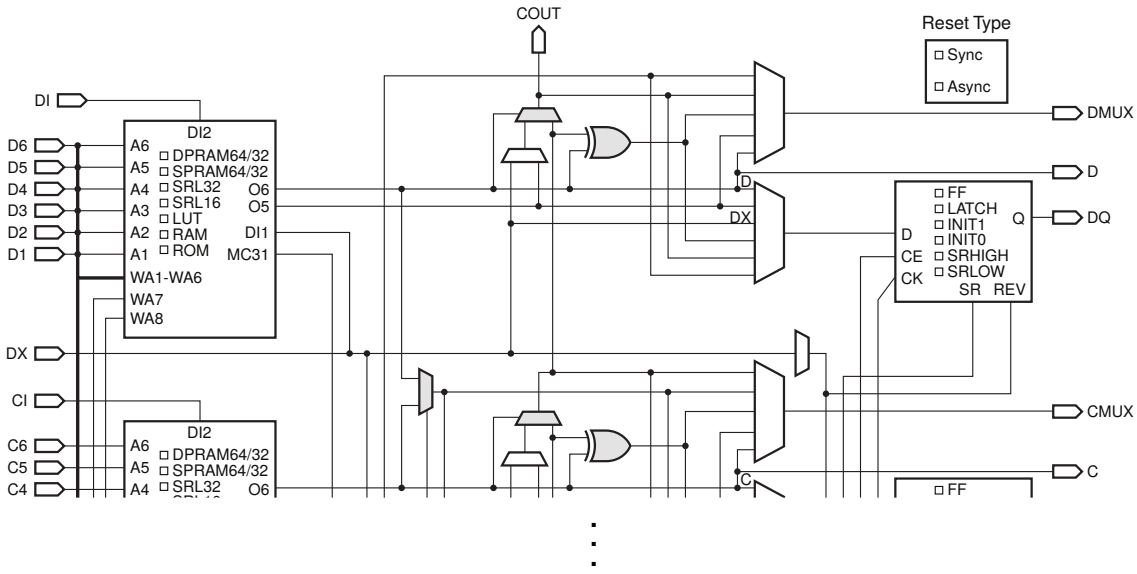


Figure 2.4: Excerpt from a SLICE (part of a CLB) diagram, which is used to implement logic on the fine-grained fabric of a Virtex-5 FPGA (from [Xil12]).

A widespread type of fine-grained fabric is the Field-Programmable Gate Array (FPGA). Modern FPGAs combine several LUTs and additional components (such as multiplexers, logic gates, flip-flops, etc.) into configurable logic blocks (CLBs). An excerpt from a CLB of a Xilinx Virtex-5 FPGA is shown in Figure 2.4. CLBs are arranged in a grid-like structure and are connected with a programmable interconnect, that allows transfer of single bit values. Figure 2.5 shows the overall logic and interconnect structure of a Xilinx Spartan FPGA.

Reconfiguration of a fine-grained fabric can take a significant amount of time. For example, reconfiguration time measurements of the Xilinx Zynq [Xil13] fabric showed that a full reconfiguration takes 30 – 42 ms[2]. For larger fabrics reconfiguration time increases accordingly.

Some systems such as PR-HMPSoC ([NK14], see also Section 2.4) use standard FPGAs as fabric, while others use customized fabrics in order to remove unneeded elements (superfluous routing capabilities or embedded hard-blocks such as SRAM memory blocks) to reduce silicon area requirements or power consumption.

The XiSystem SoC [LCBM+06] is based on the XiRisc processor [LTCC+03; MBTC06]. The XiRisc core is tightly coupled to the fine-/medium-grained PiCoGa fabric [LTC03], shown in Figure 2.6. Logic blocks are organized in rows, and an external controller is used to enable/disable each row in a given cycle. The fabric uses 4:2 LUTs and has thus medium granularity. Unlike most FPGAs, PiCoGa provides multiple configuration contexts. A configuration context establishes the

[2]Reconfiguration was performed from one of the ARM cores of the Zynq through the PCAP reconfiguration port of the Zynq fabric.

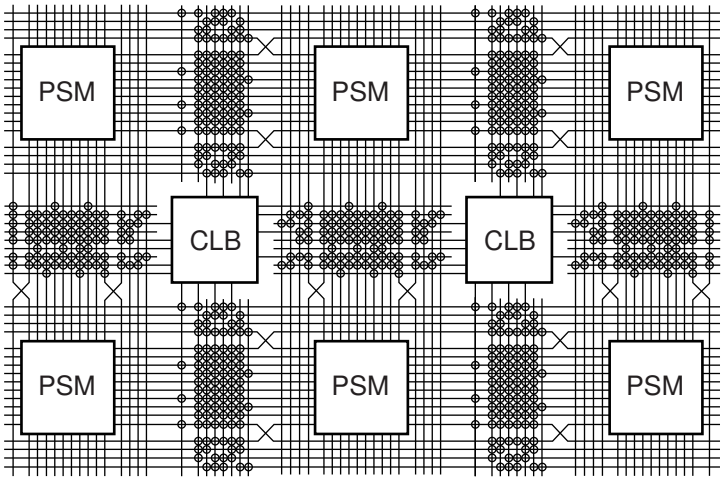


Figure 2.5: Fine-grained reconfigurable fabric in Xilinx Spartan FPGAs (from [Xil08]). CLBs (Configurable Logic Blocks) are used to implement logic. PSMs (Programmable Switch Matrices) are used to connect multiple CLBs to implement larger circuits.

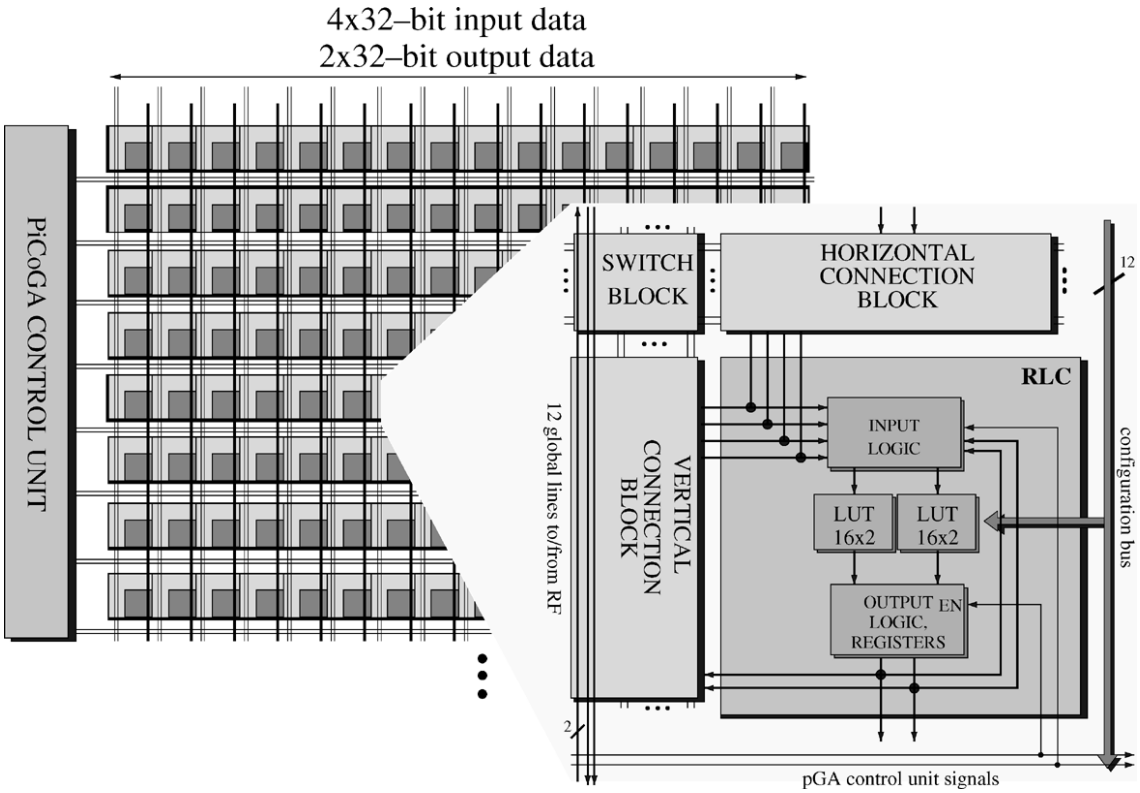


Figure 2.6: PiCoGa fine-grained fabric in the XiRisc processor (from [LCBM+06]). The XiSystem SoC [LCBM+06] adds the additional eFPGA fine-grained fabric.

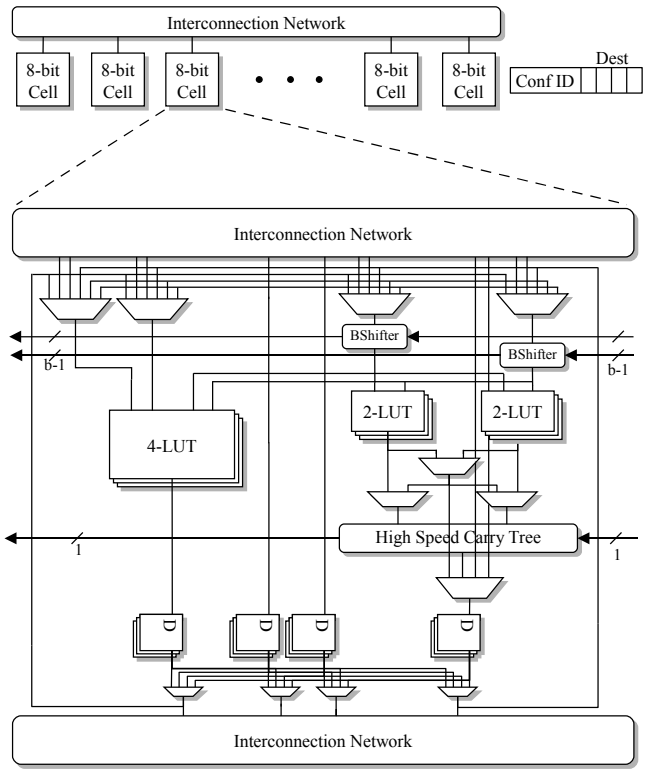


Figure 2.7: A row of logic cells in the ReMAP fine-grained reconfigurable fabric (from [WA10]).

configuration of the logic and routing elements on a fabric. With multiple contexts, it is possible to switch between different fabric configurations within one cycle (writing a new configuration to a context still takes a significant amount of time). In addition to the PiCoGA as part of the XiRisc core, XiSystem also features a loosely coupled fine-grained embedded FPGA (eFPGA). The eFPGA is connected to both the system bus and I/O pads and is intended for implementation of different communication protocols, such as I^2C, CAN, UART.

The reconfigurable fabric used in [YWWZ+10] (see also Section 2.4) is a Xilinx Virtex-5 like fabric, although with multiple configuration contexts. The fabric is divided into reconfigurable processing units (RPUs), rectangular regions on the fabric which are used independently from each other, although neighboring RPUs can be merged to accommodate accelerators that do not fit into one RPU. Each RPU has a controller that is responsible for finding the correct configuration context and controlling execution of the accelerator loaded into the RPU.

ReMAP ([WA10], see also Section 2.4) uses a custom reconfigurable fabric called SPL (Specialized Programmable Logic). The logic elements consist of 4-input and 2-input LUTs, barrel shifters, carry-chains and flip-flops, organized in logic cells. A logic cell receives an 8-bit input, and performs the same operation on all 8 bits. 16 cells form a row, with the whole fabric consisting of 24 rows. Figure 2.7 shows a row

in the SPL.

While most fabrics are designed either for high performance of kernels running on the fabric, low power consumption or silicon area use, the fabric of the WARP processor [LSV06] is optimized for fast accelerator synthesis. Most approaches synthesize accelerators at design-time, with some approaches combining multiple (offline synthesized) accelerators to implement complex kernels at run-time (see also Section 2.3). WARP however, performs accelerator synthesis (high-level synthesis, placement and routing) at run-time. As this process requires a large amount of memory and processing time, which only increases for complex fabrics, the fabric used in WARP is simplified to make online synthesis feasible. LUTs are 3-input, 2-output, with 2 LUTs combined into one CLB. CLBs also provide signals to implement a carry-chain with their horizontal neighbors. No other elements (e.g. dedicated logic gates, multiplexers, FIFOs) are provided in a CLB. Interconnect between CLBs is Manhattan-style, with additional connections to every other CLB in the horizontal and vertical direction. The simplified fabric structure allows running synthesis tools on an ARM-based GPP core with fairly low resource usage (average of 1.2 seconds time to synthesize and 3.6 MB memory requirements).

The Rotating Instruction Set Processing Platform (RISPP) uses a fine-/medium-grained fabric tightly coupled to a GPP core [BH11]. The fabric is divided into rectangular reconfigurable containers, each of which can be loaded with a fine-grained reconfigurable accelerator. Additionally, a typical kernel is implemented using multiple accelerators, instead of implementing the whole kernel as a single accelerator. This has the advantage of (i) reducing fabric fragmentation, as the accelerator of a small kernel no longer blocks the whole fabric for larger kernels (as the larger kernels can be distributed among multiple reconfigurable containers), (ii) the ability to share an accelerator between different tasks, and (iii) a reduced reconfiguration overhead, as the fabric can already be used when few small accelerators have finished reconfiguring, instead of waiting for one very large accelerator to reconfigure (see also Section 2.2 for further discussion on this concept). Containers are connected via a segmented bus, which operates on 32-bit word granularity. Additionally, the fabric provides two programmable memory ports for autonomous memory access without involving the GPP core as well as non-reconfigurable components for frequently used operations (such as addition or re-arranging bytes in a word). The fabric is controlled by a controller, which allows mapping of larger kernels to the fabric by using a combination of multiple accelerators, memory ports and non-reconfigurable components in multi-cycle operations. Due to its flexibility the RISPP fabric was used in the *i*-Core reconfigurable processor developed for this thesis (Section 2.6) and is described in more detail in Section 2.6.2.

Systems using multiple fabrics of both coarse and fine granularity have been proposed. Morpheus [TKBP+07] has a PiCoGa and eFPGA fabric, similar to XiSystem, but also an additional coarse-grained fabric. KAHRISMA ([KBSS+10], also described in more detail in Section 2.4) features both fine- and coarse-grained fabrics.

2.2 Special Instructions

The general-purpose processor core of a reconfigurable processor is used to execute the non-accelerated parts of an application (e.g. control-flow dominant, non-parallelizable code parts), and the computationally intensive kernels are run on the reconfigurable fabric. As discussed in Section 2.1, coupling between the GPP core and the fabric can vary from tight to loose, depending on which type of parallelism is targetted by the platform (ranging from instruction-level parallelism to task-level parallelism). In loosely coupled systems, the fabric can be accessed as a memory-mapped device or via a Network-on-Chip and the GPP core requires no modification to interact with the fabric. However, a tightly coupled fabric that is integrated into the GPP core itself (as an additional functional unit) cannot be accessed using the memory interface of the GPP core and load/store instructions. Instead, an extension of the core Instruction Set Architecture (cISA) is used, called *Special Instructions* (SIs). The term SI will be used throughout this thesis, however in existing literature the terms "Custom Instructions" [CG13] and "Instruction Set Extension" [GPYB+06] can be encountered as well.

SIs serve as the interface between applications and the reconfigurable fabric. An SI corresponds to a computationally intensive code fragment (which can be a whole kernel, or just a part of it), for which an implementation on the fabric is available. SIs are not limited to reconfigurable processors – ASIPs use SIs to provide applications access to their non-reconfigurable accelerators as well. However, while for ASIPs the set of available SIs and their implementations as accelerators are defined at design-time, reconfigurable processors are more flexible in this regard. The set of available SIs, i.e. which parts of the opcode space of the cISA is extended, is a design-time decision, as the GPP core itself (and in particular its instruction decoder) is generally non-reconfigurable for efficiency reasons. However, the functionality (or implementation) of an SI can be provided along with an application at compile-time, and thus does not need to be defined when designing the reconfigurable processor.

Before implementing the SI datapaths for the fabric, the SIs have first to be identified in an application. [GB11] provides an overview of SI identification techniques. Techniques vary depending on the scale at which SI identification is performed: at a fine scale small independent code fragments are implemented as SIs [AC01], while at a larger scale a whole loop or procedure is implemented as an SI [ADÖ05]. Another difference when identifying SIs is the degree of human effort involved: fully manual identification requires the application designer to find promising code fragments, while automated approaches use toolchains to analyze the application and identify SIs (e.g. in [HSM08]). Some approaches are intended to support the application designer [GMP15], making them a hybrid between manual and fully automatic identification.

Once identified, an SI can be described by a control-data-flow graph. For tightly coupled architectures, the control-flow part can be executed on the GPP core while only the data-flow is executed on the fabric. This can be done with little overhead, as the tight coupling does not incur additional latency when switching between

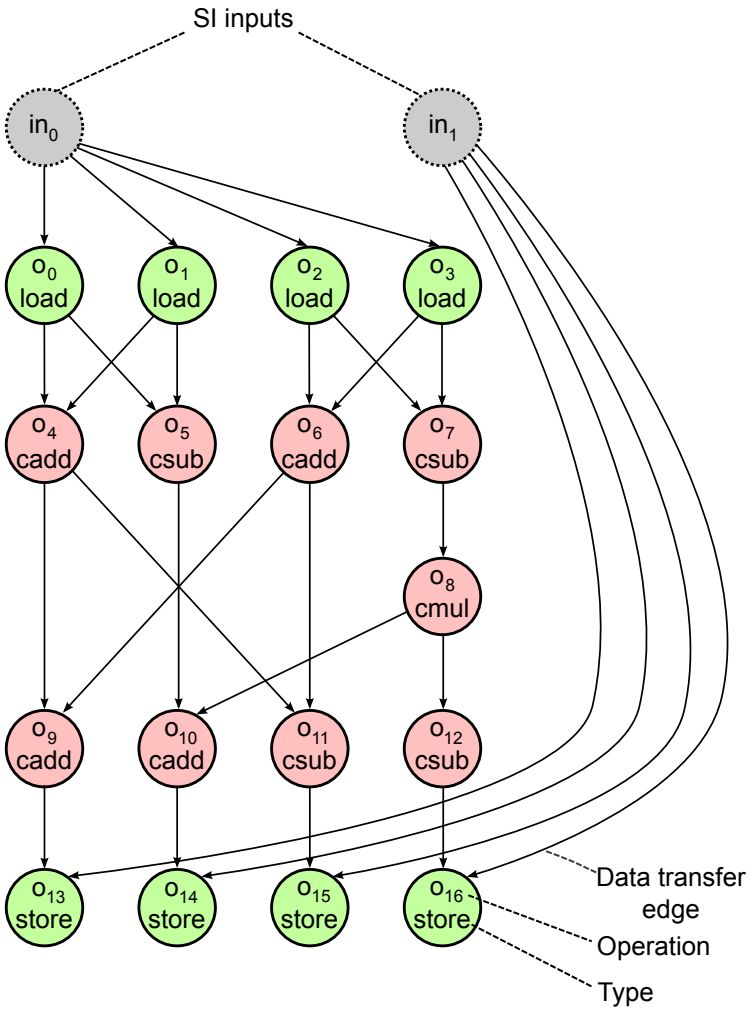


Figure 2.8: Data-flow graph for a 4-input FFT SI.

execution on the core and the fabric (in contrast to loosely coupled architectures). Figure 2.8 shows an example data-flow graph (DFG) of a 4-input FFT. The graph consists of operation nodes (e.g. complex arithmetic operations "cadd", "csub", "cmul" in the example) and data transfer edges for passing operands or (intermediate) results to operation nodes. Operation nodes may also define memory operations ("load" and "store" nodes in the example) and transfer of input/results between the fabric and the GPP core (dotted nodes "in0" and "in1" for SI input parameters in the example).

The implementation of an SI from the DFG depends on the structure of the fabric. For fabrics with large reconfigurable containers the whole DFG may be implemented as one monolithic accelerator. However, as discussed in Section 2.1.2, there are advantage to using smaller fixed-sized accelerators: reduced fragmentation, sharing of accelerators and reduced reconfiguration overhead. For a fabric using small fixed-sized containers, each type of operation node is available as an accelerator. Such a fabric is assumed in the following. If an operation is so complex that the resulting

accelerator does not fit into a container, it is split into smaller operations. To implement an SI, its DFG has to be scheduled under a resource constraint (i.e. how many accelerators of each type are available). Multiple schedules can be provided for a single SI, allowing a trade-off between fabric area requirement and achieved SI performance. Each such schedule with a different resource constraint for the same SI is called an *SI variant*.

The DFG is scheduled using standard scheduling algorithms, such as LIST or Force-Directed Scheduling [Mic94]. The number of control steps in the scheduled DFG corresponds to the latency (and thus performance) of the SI. For the example in Figure 2.8, "csub" and "cadd" shall be implemented by a single accelerator (using data-path merging techniques, such as [MBSA05]), while "cmul" is implemented as a separate accelerator. A schedule for the resource constraint of 2 instances of the combined "cadd/csub" accelerator and 1 "cmul" accelerator is shown in Figure 2.9, resulting in a schedule consisting of 6 control steps. No accelerators are used for the load/store operations, and for this example it is assumed that the fabric can perform them all in one control step. If the memory bandwidth of the fabric is limited, a large number of load/store operations may require additional control steps, thus increasing the latency of the SI.

While an SI variant already contains some implementation-specific information (i.e. how many accelerators of each type are available), most fabric-specific information is not yet specified, e.g. which operation node is mapped to which container (if there are multiple containers with the same type of accelerator), how are data transfer edges realized in the fabric, where are intermediate results stored, etc. With this information, an SI μProgram can be generated, which allows the SI variant to be run on the fabric. As the μProgram is fabric-specific, it will be discussed in Section 2.6.3 using the example of an existing reconfigurable fabric. Approaches that identify, schedule and generate accelerators and micro-programs (in the following called "μPrograms") for SIs include [LWB14] at compile-time and [LSV06] which performs all these tasks at run-time at a considerable cost but without any involvement by the application developer. Similar to the approach presented in this thesis, [MAA15] discusses a technique where SI identification and accelerator synthesis are performed at compile-time, but μProgram generation is performed at run-time.

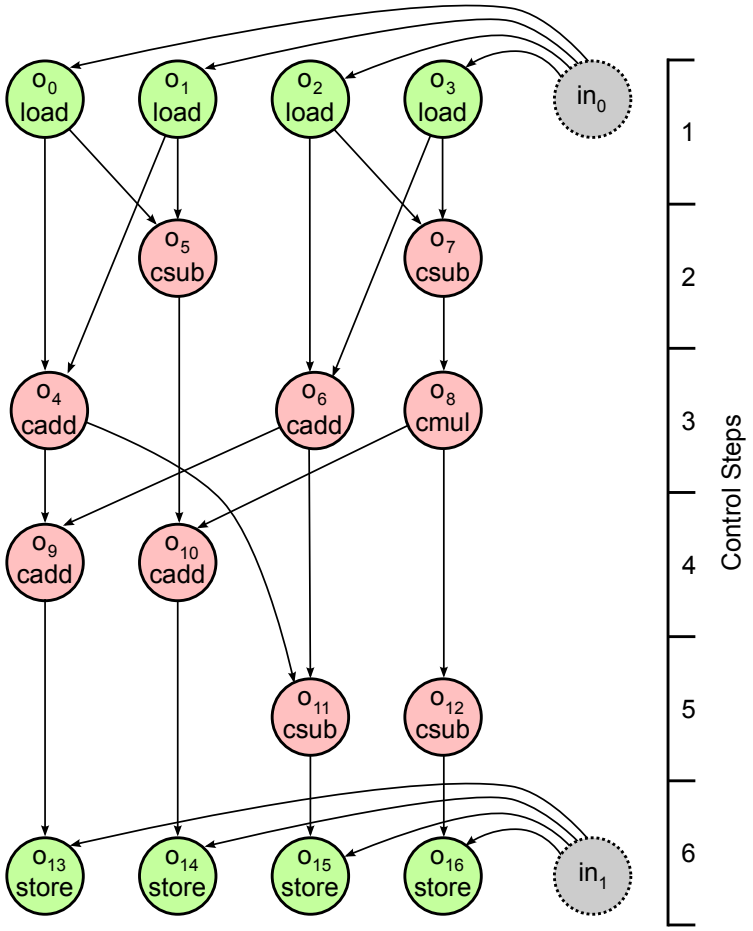


Figure 2.9: 4-input FFT SI scheduled under the resource constraint of 2 "cadd/csub" accelerators and 1 "cmul" accelerator. The resulting SI variant takes 6 control steps.

2.3 Run-Time Systems for Fabric Management

Special Instructions provide the interface for applications to *use* the reconfigurable fabric. However, to *manage* the fabric, other facilities are required. Fabric management requires solving the following problems:

- *Selection* – Selecting which SIs and which SI variant for every SI will be run on the fabric.
- *Reconfiguration Scheduling* – Establishing a reconfiguration queue for loading multiple accelerators onto the fabric.
- *Placement* – Determining where on the fabric to place the accelerators, and which already loaded accelerators to replace.

These three problems need to be solved before new SIs can be used. Solving these problems constitutes the *kernel prefetch*, which is performed by a run-time system, which is either part of the operating system (OS), a user-space application or a

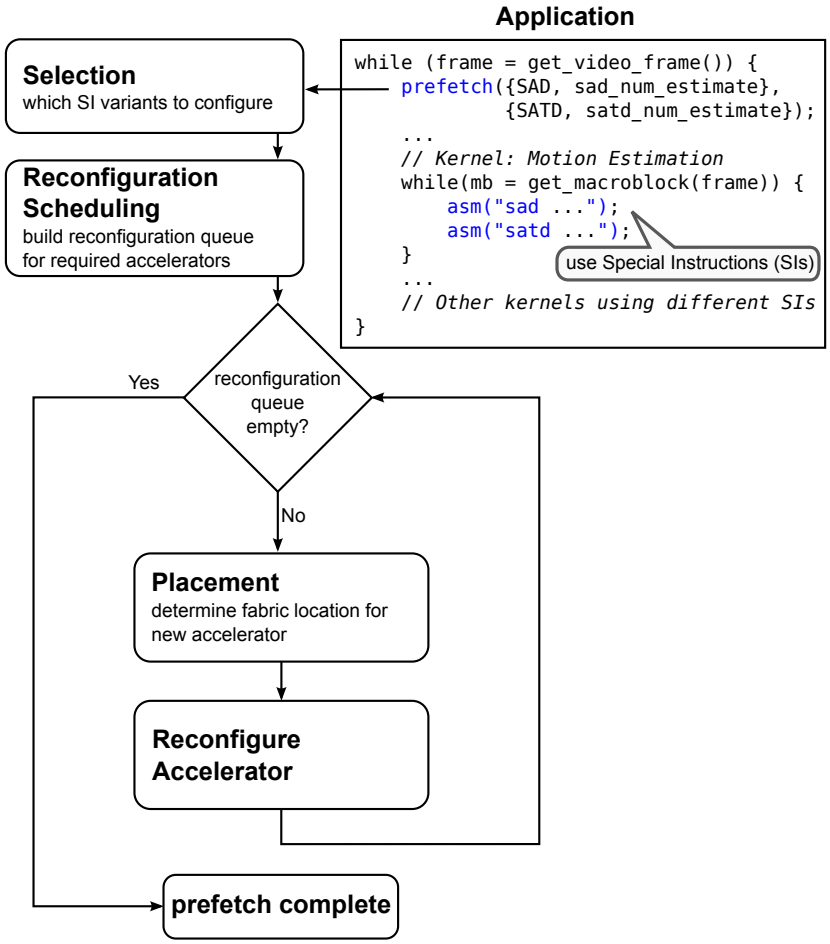


Figure 2.10: Prefetch of a kernel during execution of an application.

library (with the required low-level facilities to access the reconfigurable hardware provided by the OS). The application contains a prefetch call before a new kernel (which contains SIs), in the form of a function- or a system-call. Run-time system decisions may depend on the expected number of executions for the SIs of a kernel. This information can be determined by profiling the application at compile time [LH02], or by monitoring at run-time (e.g. from prior executions of the same code part). Figure 2.10 shows an overview of a prefetch triggered during execution of an application.

Some literature focuses on solving one of the problems, such as Placement: [KD06] describes an offline approach for accelerator placement by formulating the problem as an ILP, while [WSP03] proposes an online approach for 2D placement (which however does not regard communication-induced constraints between placed modules). Other works focus on a more complete description of the run-time system.

[HV09] proposes the Aggregate Gains algorithm, which determines at run-time which SIs to load onto the fabric (for the evaluated architecture, every SIs is monolithic, i.e. implemented by a single accelerator). The system aims to optimize overall perfor-

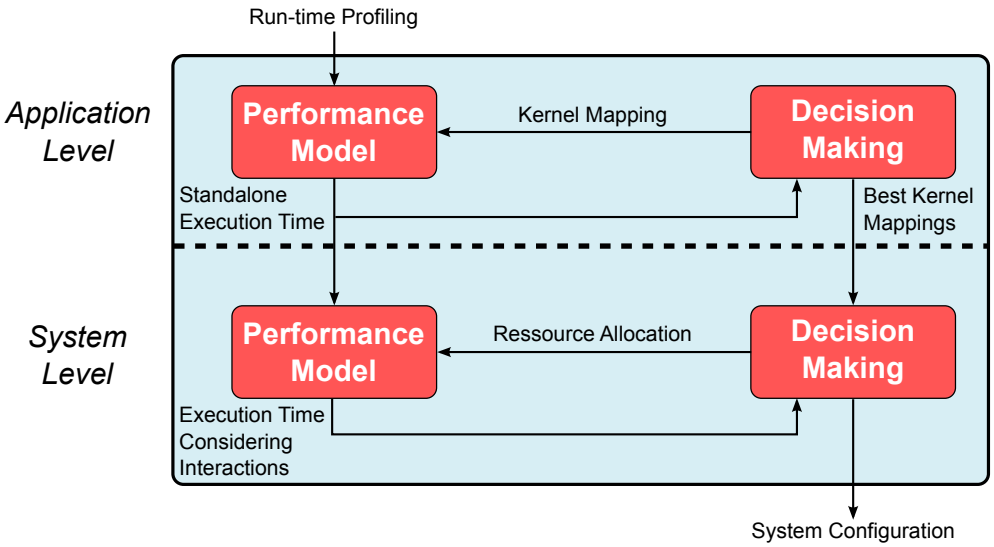


Figure 2.11: Two-layer run-time system for the MOLEN reconfigurable processor (from [MSPZ+13]).

mance of all running applications in a multi-tasking scenario, under the assumption that a prefetch is only performed when the application is scheduled (applications can be preempted and re-scheduled at a later time). When an application is scheduled, the algorithm decides if the application will use the fabric, or if it will run entirely on the GPP core. If free area is available on the fabric, an accelerator is loaded when the performance gain is greater than the overhead to load the accelerator. If no free area is available, the system removes an already loaded accelerator and loads the new one, if the performance gain from using the new accelerator (including the reconfiguration overhead) is greater than keeping the old accelerator. Performance of applications is weighted by how recently they have been scheduled (i.e. frequently scheduled applications are prioritized for using fabric over less frequently executed ones).

[LEP13] proposes an approach aimed at applications with a high degree of branching in their control flows. The goal is to load accelerators for high-priority SIs, with the priority being computed from the likelihood of SI execution and estimated performance gain. The likelihood that an SI will be executed is obtained by monitoring the paths in the control flow taken during application execution and using this information to predict future execution (a technique similar to branch-prediction in processors). Performance gain of using an SI (compared to executing it on the GPP core) is obtained from the run-time improvement and the estimated execution frequency. A combination of offline profiling and run-time monitoring is used to keep track of run-time improvement and execution frequency for each SI. Having prioritized SIs, the system then loads the highest priority SIs onto the fabric. [LEP15] presents a prototype implementation of the approach.

A two-layer run-time system is proposed in [MSPZ+13]. Figure 2.11 shows the structure of the system. The application level layer is instantiated for each appli-

cation and uses a dynamic performance model to predict application performance, while the decision making component computes the best SI placement (or "kernel mapping") within the fabric area reserved for the application. The system-level layer is instantiated once and is responsible for allocation of reconfigurable area (Section 2.3.1) using a global performance model that is constructed using information from application-level layer performance models.

The RISPP run-time system [BH11] uses a combination of offline-profiling and run-time monitoring to predict SI execution frequencies [BSKH07]. Scheduling first selects the set of SIs that will be reconfigured and then selects the SI variant for each SI [BSKH08; BSH08b]. A similar approach is also used in the KAHRISMA run-time system [ASBH11b]. The run-time system features "upgradeable" SIs, where less area-demanding variants generally use a subset of accelerators used by more area-demanding (and thus faster) variants. Reconfiguration scheduling takes this into account to ensure that when prefetching a new kernel, SIs will be available for execution on the fabric early (as less area-demanding variants) with subsequent reconfigurations gradually improving performance further. Placement of new accelerators considers the performance impact of removing existing accelerators [BSH09], removing those that have the least impact on overall application performance. Modifications of Selection with the focus on using energy-efficient SIs have been explored in [SBH10; SBH14].

2.3.1 Allocation of Reconfigurable Area

When multiple tasks run on a reconfigurable fabric (either in a time-sharing manner when run from a single core, or concurrently when the fabric is accessed by multiple cores), the fabric needs to be partitioned among the tasks. Otherwise (i.e. if all accelerators required for running a task would be loaded when the task is scheduled) the long accelerator reconfiguration would prolong context switching time significantly (each context switch would take milliseconds), making the system react extremely poorly to events causing a context switch. Partitioning allocates a certain share of the fabric to a task, which can then reconfigure its accelerators into this share.

Allocation of fabric area to a task can be done as a stand-alone step or in combination with task scheduling (see Section 2.5 and the proposed approach in Section 3.5). For static workloads, allocation can be performed at compile-time, while dynamic workloads (for which the taskset is unknown at compile-time) require run-time allocation. In this case, fabric re-allocation has to be performed whenever a new task starts or terminates. Re-allocation may reduce the fabric share of an already running task, requiring the task to adapt if it is to continue using the fabric (see Chapter 5 for detailed discussion and a solution to this problem).

Standard methods include

- allocation of equal-sized fabric shares to each task,

- share size dependent on user-specified task-priority (e.g. a task with priority 3 would get 3× the area of a task with priority 1)

A refinement of task-priority based allocation is to profile application performance for different share sizes, and give those tasks that can efficiently use large fabric shares a higher priority. Applications with simple kernels that experience no further performance improvement beyond a small share size can be given a low priority.

Other techniques include [BRAS11], which proposes a run-time mapping approach of multiple tasks to a fabric, aimed at reducing reconfiguration latency. The approach attempts to reuse existing configurations and thus reduce the amount of configurations during deployment of a new task. Fabric allocation for multi-core systems is discussed in [SBAH11], where the fabric is accessible by any of the cores. The focus is on deciding the share of the fabric that can be used by a core, which is determined using a minority game based approach. [ASBH11a] proposes "lending" of reconfigurable area to other tasks. Assuming that task A has loaded an accelerator in its share that could be used by task B and that A does not currently need the accelerator. Then task B can use (but not replace, i.e. load its own accelerator into this location on the fabric) this accelerator to speed up its current kernel. This kind of lending of accelerators is also possible when a reconfigurable fabric is used by different cores concurrently, although conflicts occur when both tasks attempt to use the same accelerator (conflicts and resource sharing in the fabric are discussed in Chapter 4).

Fabric area allocation is a necessary step to support single- and multi-core multi-tasking on reconfigurable systems. However, approaches that perform fabric allocation can be combined with any task-scheduling approach (as long as it performs task scheduling only).

2.4 Reconfigurable Multi-Core Systems

Multi-core systems can be divided into two classes: fixed multi-core and reconfigurable multi-core [WMGR+10]. Fixed multi-cores have a hardware structure that is fixed

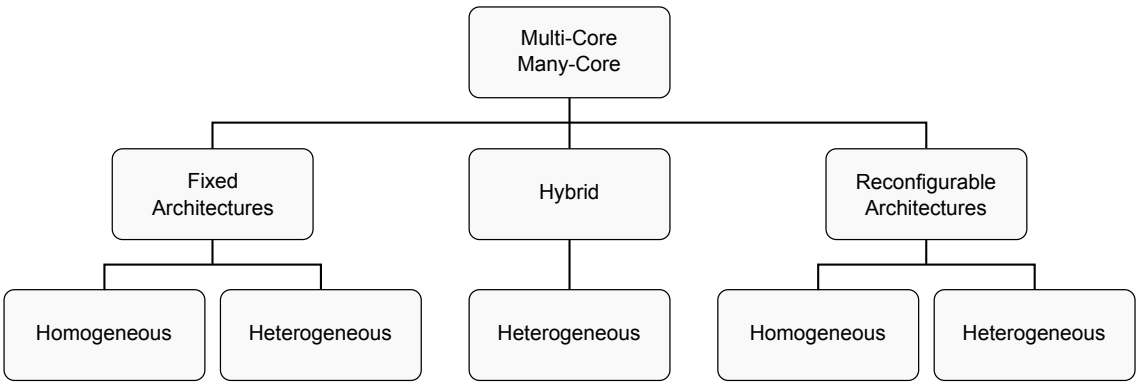


Figure 2.12: Multi-/many-core taxonomy (from [WMGR+10]).

at design-time, while in reconfigurable multi-cores the hardware structure can be changed after fabrication. Unless stated otherwise, in this section the term "multi-core" also includes "many-core" systems, which include hundreds of processing elements. Hybrid multi-cores are a combination of fixed and reconfigurable, where both types of system are integrated on a chip, but are segregated. The different classes of multi-cores can also be further characterized as homogeneous or heterogeneous. Figure 2.12 shows the different combinations as defined in [WMGR+10].

In the scope of this thesis, only reconfigurable multi-cores will be discussed further. The type of reconfiguration can be divided into multiple classes (e.g. reconfiguration of interconnect, data-path, power modes, memory, etc.). For the following discussion, two classes will be used: multi-cores with a reconfigurable fabric capable of loading accelerators and multi-core systems featuring other forms of reconfiguration without the ability to load accelerators.

Non-Accelerator based Reconfiguration in Multi-Core Systems

A multi-core system with a reconfigurable interconnect between the cores is presented in [RASS+08]. The system is divided into two parts: a static part for computation and a reconfigurable part for communication. In addition to processing cores, the static part also includes memories. The reconfigurable part is based on a Network-on-Chip (NoC) and is used as an interconnect between each component of the static part. Run-time reconfiguration of the NoC allows modification of routing tables and configuration of direct connections between (non-neighboring) switches.

The QuadroCore [Pur10] is a multi-core that features reconfiguration in the interconnect between specific pipeline stages of the cores (Figure 2.13). The Execute stages of the GPP cores (Motorola NCore, 32-bit RISC) of the 4-core system allow switching the processor into specialized modes which benefit applications with a certain type of parallelism. In Wide-Word-ALU mode, up to four ALUs are combined for SIMD operations and accelerate applications that exhibit data-level parallelism. Register Sharing mode allows one core to save the result of an instruction to the register file of a different core, accelerating transfer of results between cores. Other modes include Fast Memory Access (where memory access for all cores is handled by a single master core), SIMD and MIMD modes.

Core Fusion is an architecture where simple cores in a multi-core system can be fused on-the-fly into more complex cores [IKKM07]. Based on 2-way out-of-order cores, 2 or 4 cores may be combined, resulting in a 4-way or 8-way out-of-order core. To support this, the front-end and back-end have to be extended with additional hardware to support renaming and operand transfer when multiple cores are fused. Fusion of cores is controlled by two additional instructions, which are used to annotate parallel regions in an application.

A similar approach of merging existing simple cores into more complex out-of-order cores is proposed in [PM12]. The architecture, Bahurupi, is a multi-core consisting of 2-way out-of-order cores, and in its default configuration benefits applications that

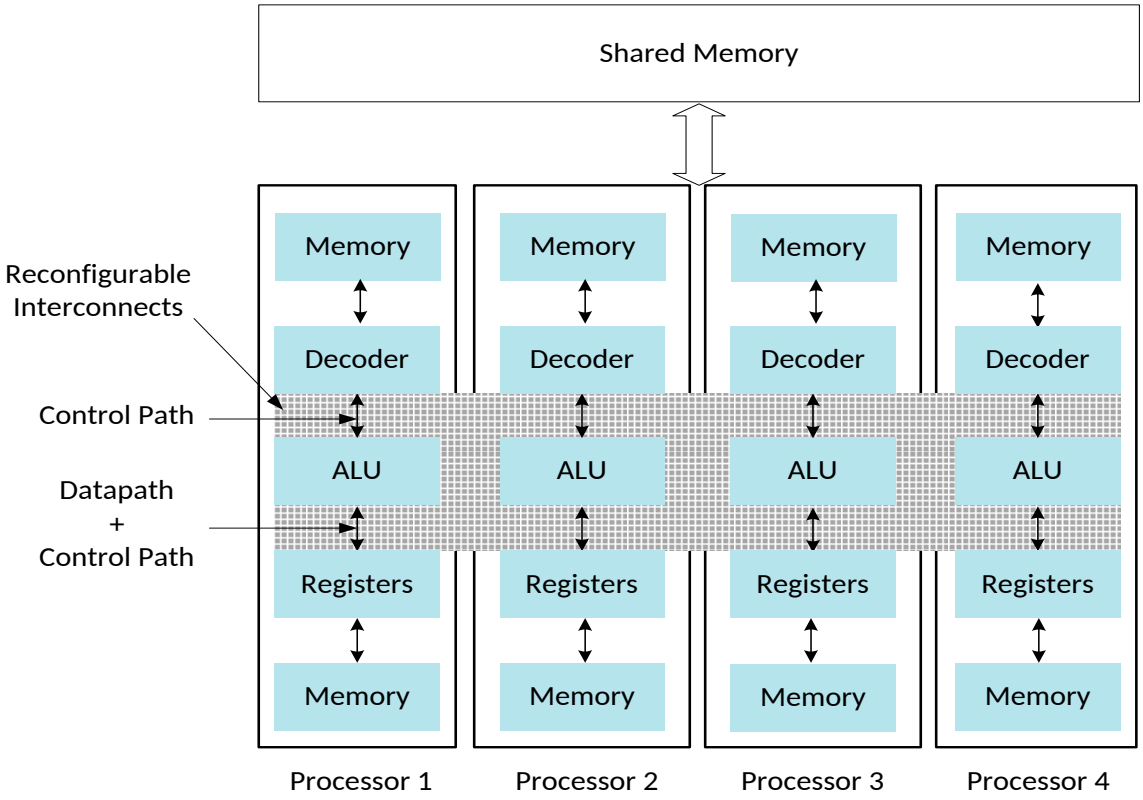


Figure 2.13: QuadroCore processor with reconfigurable interconnect between cores (from [Pur10]).

have a high degree of task-level parallelism. Multiple cores can also be switched into a (up to) 4-core coalition, where 4 such cores can be used as an 8-way out-of-order core, improving performance in applications which exhibit high instruction-level parallelism. Unlike in Core Fusion, where the cores were designed specifically for the architecture, Bahurupi requires only minor modification of standard 2-way cores, and relies on compiler-support to resolve data dependencies between code fragments executed on the cores.

CoreSymphony [NSMK11] is similar to the previous two architectures, again allowing narrow-issue cores to be fused into a wide-issue core. While Core Fusion features centralized components for renaming and steering, CoreSymphony steers interdependent instructions to the same core, thus eliminating the additional latency overhead incurred by centralized components.

Multi-Core Systems with Reconfigurable Accelerators

The KAHRISMA (KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array) architecture [KBSS+10] consists of front-end components for instruction fetching, decoding and dispatching and a fabric with both fine-grained (FG-) and coarse-grained (CG-) reconfigurable processing elements (EDPE – Encapsulated

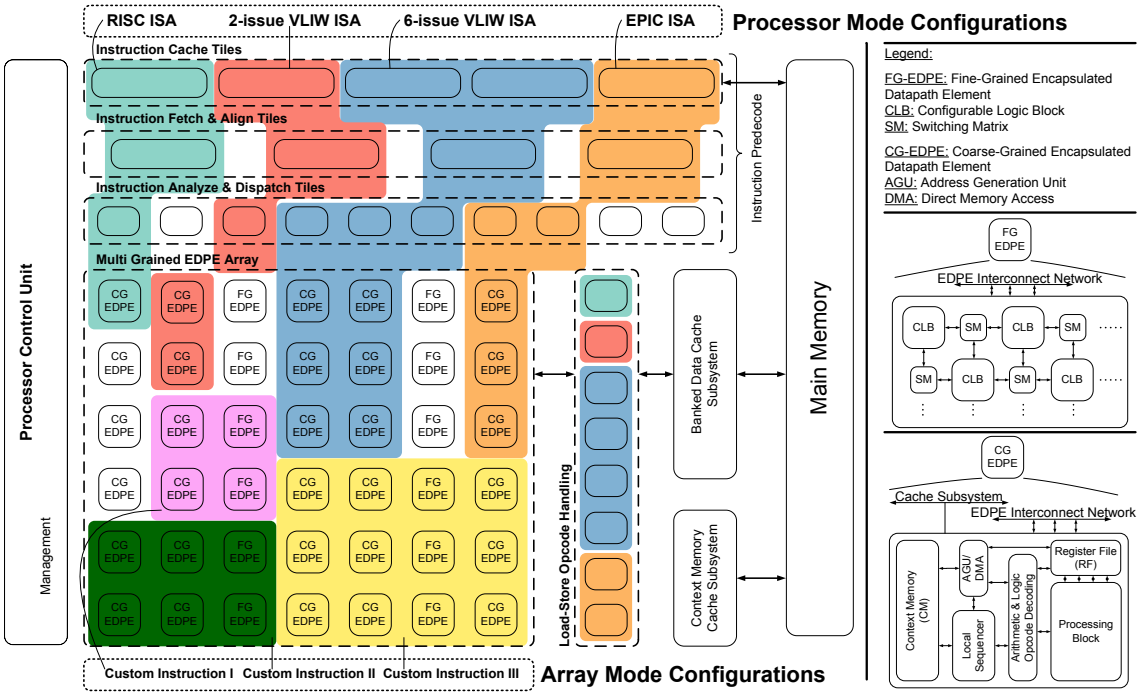


Figure 2.14: KAHRISMA reconfigurable system. Processor cores can be dynamically instantiated by combining front-end elements ("tiles" at the top) and coarse-grained reconfigurable processing elements (CG-EDPE) from the multi-grained EDPE array. The array can additionally be used to implement SIs (from [KBSS+10]).

DataPath Element). Figure 2.14 shows an example configuration of KAHRISMA. Instances of processor cores can be configured on the fly by combining front-end components with a number of CG-EDPEs. Using multiple instances of front-end and coarse-grained elements, wider issue cores can be generated. Additionally, the fabric can be used to implement SIs independent of the cores. An SI can use accelerators on both CG- and FG-EDPEs as well as directly access the memory system without involving processor cores. The flexibility of the architecture, in particular the switching between different ISAs at run-time, required the development of a mixed-ISA compilation toolchain [Str13].

A multi-core architecture that uses a fine-grained reconfigurable fabric is presented in [YWWZ+10]. The fabric is partitioned into reconfigurable areas (RPUs – reconfigurable processing units) of equal size. The cores and the fabric are connected via a crossbar, allowing any core to access any RPU. However, the fabric can not access main memory directly (instead a GPP core must perform the access and provide data to the RPU) and there is no communication between RPUs. This impairs memory-intensive kernels, as memory accesses are handled by a GPP core and the implementation of the kernel is restricted to the size of one RPU.

In CReAMS [RBC11] a multi-core is composed of cores (each with a reconfigurable fabric) that use a shared L2 cache for communication. Each core has a 5-stage

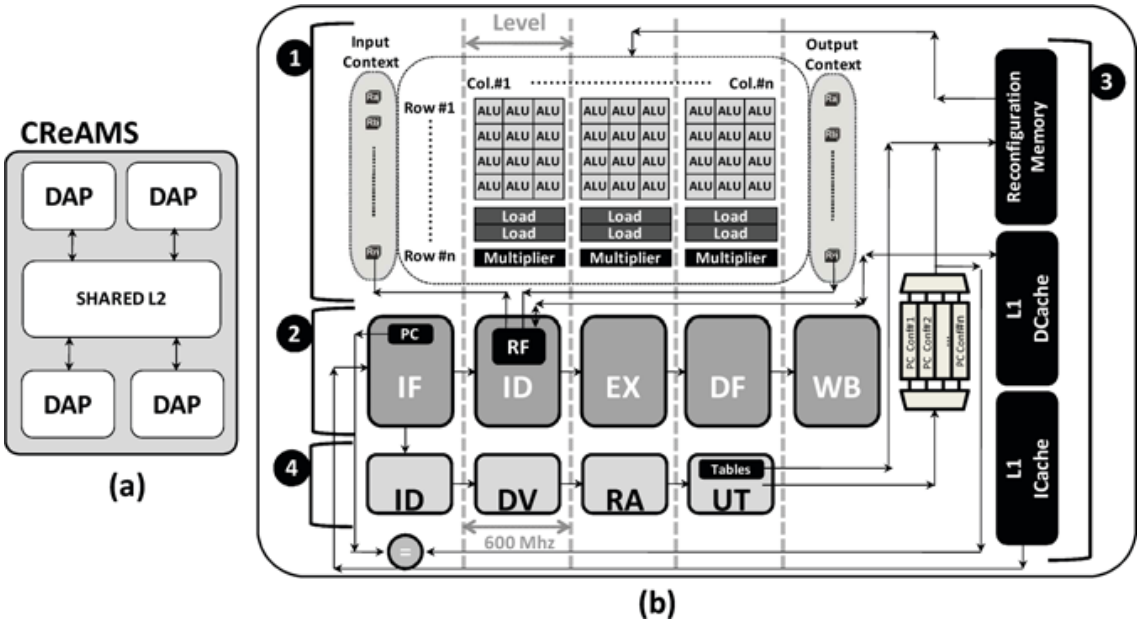


Figure 2.15: CReAMS reconfigurable multi-core. (a) Multi-core consisting of Dynamic Adaptive Processors (DAP). (b) Details of a DAP with embedded coarse-grained fabric (from [RBC11]).

pipeline (SPARC V8 instruction set) that is tightly coupled to a coarse-grained fabric. The architecture of the multi-core and a single core is shown in Figure 2.15. The system does not use explicit SIs, but instead detects accelerated instruction sequences on the fly (using the Dynamic Instruction Merging technique proposed in [BRGC08]) and maps them to the fabric using the Dynamic Detection Hardware component (④ in the figure).

PR-HMPSoC [NK14] is an architecture consisting of several NoC-connected fine-grained reconfigurable regions. Each region can be configured with an accelerator or a processor core ("soft-cores" such as the Xilinx MicroBlaze, which are implemented using reconfigurable logic only) and is also connected to a dedicated memory block. A master processor (which is one of the few components not implemented in reconfigurable logic) is responsible for handling reconfiguration of the regions.

RAMPSoC, an approach for creating multi-core architectures with reconfigurable accelerators is presented in [GHSB08; GHPB08]. The architecture is customized at design-time by selecting which type of interconnect (NoC, bus, etc.) is used and which type of processing elements are used: a combination of reconfigurable fabrics, GPP cores and reconfigurable processors is possible. In a reconfigurable processor, each core has access to its own fabric only.

[HYMN+09] studies how a reconfigurable multi-core can be used for implementing the WiMAX physical layer. The architecture does not use a traditional reconfigurable fabric, but instead the cores themselves include a number of arithmetic/logic components, with the connections between them being reconfigured at run-time to

implement the required functionality, making this a coarse-grained fabric. Multiple such cores use a shared data-memory, but each core has its own instruction memory and a small core-local scratchpad memory.

A shared reconfigurable fabric in a multi-core system is proposed in [CM11]. The cores in the system are based on the Stretch processor [Str] and a coarse-grained reconfigurable fabric. The fabric is tightly coupled to all processor cores. In case of concurrent accesses to the fabric by multiple cores, round-robin arbitration is used to determine which core can use the fabric and which is stalled.

ReMAP [WA10] uses a reconfigurable fabric for both communication between cores, acceleration of kernels and a combination of both. Fabric-based communication is aimed at streaming or pipelined applications, with different application stages mapped to different cores. The fabric is then used to establish a dedicated communication channel between the respective cores, providing better performance than transferring data via shared memory. Computationally intensive stages in such applications can also be completely offloaded to the fabric, allowing processing of data while it is in transit between cores. The system uses a custom LUT-based fabric architecture, which is tightly coupled to the cores (as an additional functional unit). Concurrent access to the fabric by multiple cores is handled either by arbitrating accesses to the fabric in a round-robin fashion, or by partitioning the fabric among the cores.

In addition to the research platforms described above, commercial reconfigurable multi-cores are available as well. These includes SoCs from Altera and Xilinx.

Certain devices (SE, SX and ST models) in the Cyclone V device family [Alt15] are an SoC that consists of a "hard-processor system" (HPS, a single- or dual-core ARM Cortex-A9) loosely coupled to an FPGA (i.e. fine-grained reconfigurable fabric). In addition to the cores and the fabric, the SoC includes an SDRAM controller, a small (64 KB) on-chip RAM and ROM, a DMA controller, an L2 Cache for the ARM cores, and several peripherals (UART, Timer, I/O, etc.).

The Xilinx Zynq SoC [Xil] consists of an ARM Cortex-A9 dual-core and a Xilinx Kintex reconfigurable fabric. The SoC also contains a 256 KB SRAM-based scratch-pad (OCM – on-chip memory), and (similar to the Altera Cyclone V) DDR and DMA controllers, an L2 Cache and peripherals. Figure 2.16 shows a block diagram of the Zynq. A case study of the Zynq was done in [BSB13], where a Kalman filter for a hearing aid was implemented on an Intel Core i5, an ARM Cortex-A9, as an SI on the Zynq SoC, and as an ASIC. Given a performance requirement (a filter cycle had to be completed in a certain amount of time), the focus was on power and energy efficiency. Evaluation has shown that power dissipation and energy consumption of the fabric-based SI was better than the implementation on the Core i5 or the Cortex-A9 (although energy consumption on the fabric was only slightly better compared to the Cortex-A9).

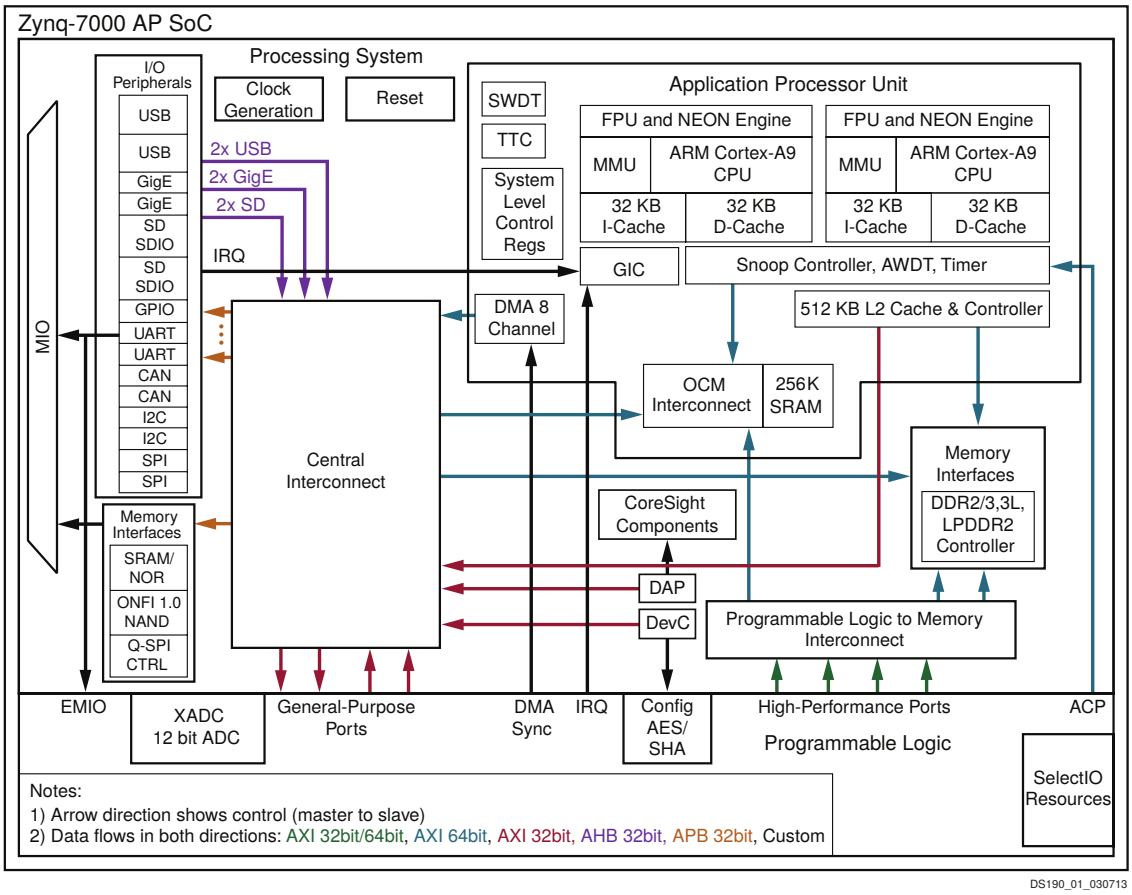


Figure 2.16: Zynq reconfigurable multi-core, consisting of an ARM Cortex-A9 dual-core (upper right) and a Xilinx Kintex reconfigurable fabric (bottom) (from [Xil]).

2.5 Task Scheduling

In the following, the background and notations for scheduling are introduced. After that, existing approaches for task scheduling in reconfigurable systems are presented.

Scheduling Basics

The Handbook of Scheduling [Leu04] describes scheduling as follows:

> Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures. Depending on the situation, resources and activities can take on many different forms. Resources may be machines in an assembly plant, CPU, memory and I/O devices in a computer system, runways at an airport, mechanics in an automobile repair shop, etc. Activities may be various

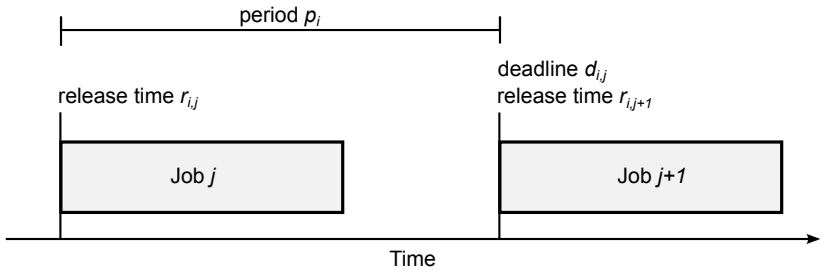


Figure 2.17: Release times, deadlines and period for periodic task T_i with implicit deadlines.

operations in a manufacturing process, execution of a computer program, landings and take-offs at an airport, car repairs in an automobile repair shop, and so on.

For this work, task scheduling on a processor is of interest, i.e. the decision at which point in time a particular task shall be run.

The set of available tasks is the *taskset* T consisting of N tasks $T_i, i = [1..N]$. Tasks can be periodic or aperiodic. A periodic task consists of multiple jobs with $T_{i,j}$ referring to the jth job in task T_i. Jobs are the regular invocations of an application at specified times (*release time* $r_{i,j}$), with *deadlines* $d_{i,j}$ specifying the time at which a job j must be completed. Release times are assumed to be without jitter, i.e. a job can start immediately at the time it is released. Furthermore, *implicit deadlines* are assumed, where the release time of a job $j+1$ is equal to the deadline of the previous job j. The *period* p_i is then the time between consecutive release times. Figure 2.17 shows the timing for a periodic task T_i with implicit deadlines. An example for a periodic task is a video encoder, with each job corresponding to the encoding of one video frame, and deadlines resulting from the required framerate (e.g. 20 FPS would require a deadline of 50 ms after a job is released). An aperiodic task consists of only one job, after which the task is finished. If task T_i is aperiodic, it has a *completion time* C_i, i.e. the time when T_i finishes its execution. Similarly, jobs of periodic tasks also have completion times $C_{i,j}$.

Task scheduling algorithms can have different goals, reducing the number (or severity) of deadline misses or finishing a taskset as quick as possible. The objective functions used in this thesis are:

- *Makespan* – the time when a taskset is completed, i.e. $\max(C_1, .., C_N)$
- *Overall Tardiness* – accumulated time (over all tasks and jobs) by which deadlines were violated, i.e. $\sum_{i=1}^{N} \sum_j \max(C_{i,j} - d_{i,j}, 0)$

Fundamentally, schedulers are divided into two groups: offline and online schedulers. *Offline schedulers* have full knowledge of the taskset and all task characteristics (i.e. number of jobs for each task, release time and deadlines) before constructing a schedule, and during execution of the schedule the taskset will not be modified. Such information is available in a *static workload*, which does not allow new tasks

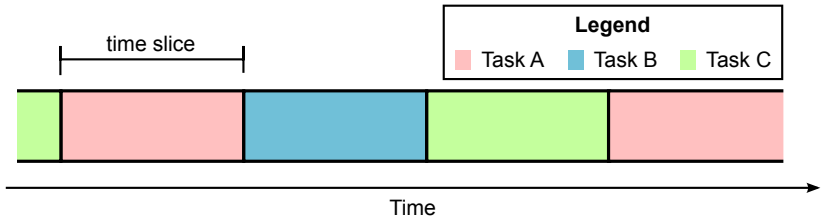


Figure 2.18: Three tasks scheduled using a Round Robin scheduler.

entering the system and thus affecting the schedule. The other class consists of *online schedulers*, which do not have full knowledge of the taskset, and only use currently available information to construct a shedule. Such schedulers are suitable when using a *dynamic workload* where new tasks can arrive at any time. This thesis assumes dynamic workloads, thus online schedulers are of particular interest. Preemption support is also assumed, i.e. the scheduler can interrupt a running task and schedule a different one at any time.

A frequently used scheduler for single-core systems with workloads containing tasks with deadlines is the Earliest Deadline First (EDF) scheduler. Given a set of tasks with jobs that have been released, EDF prioritizes the task where the current job has the closest deadline (most "urgent" task). If it is possible to schedule all tasks without deadline misses, then EDF will generate such a schedule [Leu04], thus EDF allows a processor utilization of up to 100%. As EDF recomputes priorities of active tasks when performing a scheduling decision, it is a *dynamic priority scheduler*.

Rate-Monotic Scheduler (RMS) is another scheduler for workloads with deadlines. Unlike EDF, it is a *static priority scheduler*, where task priorities are assigned once and do not change at run-time, and whenever two tasks T_i and T_j have jobs ready, the task with the higher priority is scheduled, even if the deadline of the task with the lower priority is closer. A task's priority is computed as the inverse of its period, i.e. $1/p_i$. RMS will schedule tasks without deadline misses, provided that the processor utilization bound is $\leq \ln 2$ ($\approx 69.3\%$) ([Leu04], Section 28.4).

For workloads without deadlines, Round Robin (RR) is a popular scheduler. Here, the scheduling decision is performed repeatedly after a constant amount of time called *time slice*. Tasks that are ready to run are organized in a queue, and when performing a scheduling decision, RR stops the currently running task, places it at the end of the queue, removes the task at the front of the queue and schedules it (Figure 2.18 shows an example schedule). The scheduler ensures that no task starves, and is thus a "fair" scheduling policy.

Scheduling for Reconfigurable Systems

A survey that focuses on hardware-support for multi-threading in reconfigurable systems is presented in [ZKG09]. Reconfigurable processors are categorized into systems with implicit, explicit, and no architectural support. Systems with implicit

multi-threading perform scheduling in hardware, providing efficiency at the cost of flexibility. An example of an implicit multi-threading system is [UMKU06], where scheduling is performed on instruction-level, with instructions of a fixed number of threads executed according to a scheduling policy (round-robin or fixed priority). Explicit multi-threading systems perform part of the scheduling decision in software (e.g. task scheduling), part in hardware (thread scheduling, reconfiguration queue scheduling). [MNMB+04] is an example for an explicit multi-threading system. In a multi-core system a system-wide task mapper is implemented in software and assigns tasks to cores, while a core-specific scheduler determines when tasks run on this core (scheduling as defined further above). The paper does not go into details which core-specific scheduling algorithms are used.

In [CMM12] a reconfigurable fabric is shared between multiple cores in a shared-memory system. The work focuses on online-scheduling of tasks among multiple cores. For tasks that use the fabric, the proposed scheduler reserves both processor time and a portion of the fabric for a task when constructing a schedule. The system model is limited in that it assumes that a task either requires a fixed amount of fabric area, or that it is a software-only task executed on the GPP core ("all-or-nothing" approach to SI execution). The model neither supports flexible SIs with different variants nor allows task execution while it is performing reconfigurations.

Several reconfigurable processors with support for multi-tasking implement entire tasks either in hardware or in software. In such systems the hardware tasks execute in parallel whereas the software tasks execute sequentially on the GPP core without acceleration. A hierarchical approach to dynamically decide which task shall execute on the reconfigurable fabric is proposed in [NMBV+03]. ReconOS [AHKL+14; LP09] is an operating system that provides services to threads that can either be mapped to reconfigurable hardware or run in software. It uses an infrastructure for transparent inter-task communication – independent of whether a task executes in hardware or software [LP07]. [PAAS+06] proposes OS extensions for support of reconfigurable processors, among them a hardware module that performs thread scheduling. [TRC11] presents a scheduler that assigns periodic tasks to heterogeneous processing elements (including a reconfigurable fabric) offline and that integrates sporadic tasks by extending the resulting schedule at run-time. Modified EDF scheduling algorithms are proposed in [DP06], under the assumptions that tasks run fully on the reconfigurable fabric and that multiple tasks can run on the fabric concurrently without conflicts. These assumptions assume a very constrained system, as here tasks must be implemented on the fabric or they can not run. Additionally the model assumes that multiple accelerators can be used without conflicts, which is not true if accelerators use shared resources shared resources such as memory ports, accelerator interconnect, etc. All these approaches perform a binary decision whether to implement a task entirely in hardware or software, thus some tasks are significantly accelerated and other tasks are not accelerated at all.

Architectures that are not constrained by such a binary decision use tasks with SIs, allowing execution of performance-critical application parts on the fabric. This enables a high degree of flexibility, as given a particular area of the fabric, the

application (or the run-time system on its behalf) can decide which SIs to run on this limited fabric area and which to execute on the GPP core. Proteus [Dal03] proposes a mechanism that allows to preempt SIs during their executions (e.g. to handle interrupts or to switch to another task) and resume their execution later. The actual task scheduler used is a Round Robin scheduler. [LMBG09] presents an online task-scheduler and fabric allocator for a reconfigurable processor. Scheduling and allocation is formulated as a 2D model, where a task is defined by its execution time and required fabric area. This approach requires knowing the execution time of a task before it starts and only supports tasks that perform all their reconfigurations at the start of a task, which limits its applicability.

The reconfigurable multi-core processor presented in [ASBH11a] does not use a task scheduler as at most one task executes per core. The reconfigurable fabric is allocated based on the deadlines of the parallel executing applications. Frequent fabric re-distributions are performed whenever any of the tasks proceeds to its next kernel, resulting in frequent reconfigurations, which can cause performance degradation in fine-grained reconfigurable fabrics. Reconfiguration latency reduction is the goal of [RMAS+09], where a design-time approach is used to map multiple applications onto the fabric.

[HM09b] assumes a reconfigurable processor with multiple SI variants per SI. The approach uses a combination of task scheduling and SI variant selection with the goal being reduction of processor utilization while meeting the deadlines of all tasks. The task scheduling approach assumes a static workload and uses list scheduling with fixed priorities (corresponding to task deadlines) to schedule tasks.

A compiler-assisted scheduling approach for the MOLEN reconfigurable platform [VWGB+04] is presented in [SSB09]. The compiler provides information about the temporal distance between kernels to the run-time system. For the actual task scheduling, a Round Robin scheduler is used.

2.6 i-Core Reconfigurable Processor

The contributions of this thesis in Chapters 3 to 5 are illustrated using the *i*-Core reconfigurable processor which was designed and implemented in the scope of this thesis. The *i*-Core ("invasive Core") was designed for a heterogeneous many-core system which was developed for the Invasive Computing project (see Section 2.6.4)[3]. The *i*-Core is a reconfigurable single-core processor with a fine-grained reconfigurable fabric which is tightly coupled to the core pipeline. The concepts of the *i*-Core are based on the Rotating Instruction Processing Platform (RISPP, [BH11]). In

[3] In addition to providing application-specific acceleration through the reconfigurable fabric, the *i*-Core also provides adaptivity in the micro-architecture (e.g. cache, pipeline, etc). Except for the permutating register file extension mentioned in the next subsection, micro-architectural adaptivity was researched at the "Institut für Technik der Informationsverarbeitung" (ITIV), KIT, and is not part of this thesis. For the sake of simplicity, any future references to the *i*-Core will therefore not include micro-architectural adaptivity.

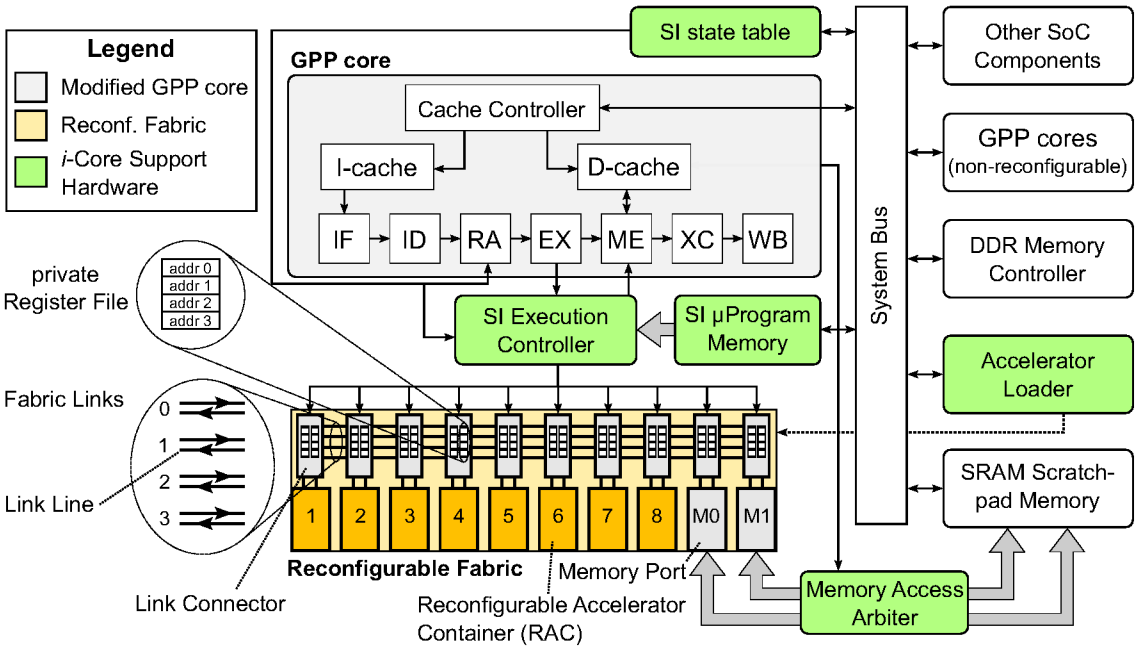


Figure 2.19: Architecture of the *i*-Core reconfigurable processor SoC.

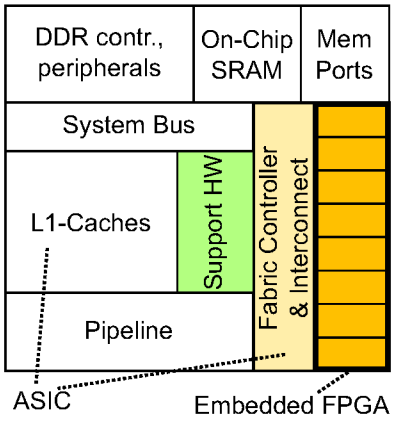


Figure 2.20: Envisioned floorplan of the *i*-Core reconfigurable processor SoC with the fabric implemented as an embedded FPGA.

particular, the reconfigurable fabric [BSH08a] and the run-time system for managing the fabric (Section 2.3) were used and extended from the RISPP implementation.

2.6.1 GPP core, System-on-Chip and Fabric Coupling

Figure 2.19 shows the architecture of the *i*-Core System-on-Chip (SoC), which is an extension of the Aeroflex Gaisler LEON-3 [Aer] SoC[4]. The AMBA bus and the

[4]The whole LEON-3 SoC is freely available as VHDL code under an open-source license, allowing the modification required for adapting the SoC to accommodate a reconfigurable processor.

components in white (DDR Memory, unmodified LEON-3 GPP cores, Ethernet and UART interfaces, etc.) are taken from the LEON-3 SoC. The shaded components are modified versions of existing components (GPP core is an extended LEON-3 core) or completely new ones, such as the run-time reconfigurable fabric and the *i*-Core Support Hardware (additional components for managing run-time reconfiguration and SI execution). With a reconfigurable core and LEON-3 GPP cores, the *i*-Core SoC is a heterogeneous shared-memory multi-core system.

The LEON-3 is a 32-bit RISC processor with an in-order pipeline and a SPARC V8 [SPA] Instruction Set Architecture (ISA). In order to use the LEON-3 as the GPP core of the *i*-Core, the processor pipeline had to be modified to allow tight coupling of the fabric and the core pipeline (Section 2.1). The (unmodified) 7-stage pipeline is organized as follows:

1. Instruction Fetch (IF) – The instruction word is fetched from memory/the instruction cache.
2. Instruction Decode (ID) – The instruction is decoded into the opcode and operand addresses/immediate values.
3. Register Access (RA) – Operands for the instruction are fetched from the register file.
4. Execute (EX) – The operation corresponding to the opcode is performed. For arithmetic instructions the opcode and operands are sent to the arithmetic-logical unit (ALU), for branch instructions the branch condition is evaluated and the program counter is updated with a new value. Memory instructions (load, store) take at least 2 cycles and are started in the EX stage.
5. Memory (ME) – Memory operations started in EX are finished in ME.
6. Exception (XC) – If an exception (interrupt or trap) is detected, it is handled here by starting execution of the corresponding trap handler.
7. Write-back (WB) – Instructions that update the register file commit their result.

Special Instructions for the *i*-Core are implemented as an extension of the SPARC V8 ISA in the free opcode space designated as "unimplemented instructions". This requires extension of the Instruction Decoder (ID stage) to correctly recognize SIs and their operands. If an SI is detected, the *i*-Core needs to determine how to execute the SI – on the fabric or emulated on the core pipeline. An SI can be executed on the fabric if (i) all accelerators required for the SI are loaded on the fabric and (ii) the SI μProgram that controls SI execution on the fabric (Section 2.6.3) is available. If both conditions are met, the SI is marked as "executable on the fabric" in the *SI State Table* (Figure 2.19). During the RA stage a lookup is performed in this table for the detected SI. If the SI is not available in hardware, an "unimplemented instruction" trap is generated, which is used during the later XC stage to start emulation of the SI on the core pipeline. Otherwise, SI execution on the fabric starts in the EX stage. The reconfigurable fabric is connected to the pipeline as an additional multi-cycle functional unit. Pipeline registers containing the SI operands (which can come from the register file of the GPP core or the immediate field of the SI) are connected to the fabric links. Once SI execution starts in the EX stage, the pipeline is stalled until

the SI is finished[5]. Once the SI has finished execution, the pipeline is unstalled and the results of the SI are transferred from the fabric links into the pipeline registers during the ME stage. The SI then proceeds through the remainder of the pipeline (XC, WB stage) as any other SPARC instruction.

If the SI can not be executed on the fabric (e.g. due to not all accelerators being loaded), then the software code that is equivalent to the SI functionality is executed on the core pipeline (SI emulation on core pipeline). While SI emulation is much slower than execution on the fabric, it ensures that the application executes correctly, independent of the state of the fabric (if multiple tasks use the fabric, the run-time system may decide not to reconfigure any accelerators to some of the slower SIs). SI emulation is started from the "unimplemented instruction" trap handler which is executed if an SI is in the XC stage and an unimplemented instruction trap was triggered during the RA stage when the SI State Table was checked. The trap handler identifies which emulation code to run, passes the operands and executes the emulation function for the SI on the GPP core. Once finished, the trap handler ensures that the SI results are written to the correct registers.

Apart from the adaptivity provided by the reconfigurable fabric, the *i*-Core provides adaptivity in the pipeline in the form of a *permuting register file*. A permutation can be used to describe a chain of swap operations. Such chains of swap operations are often generated by a compiler for procedure calls, control-flow and loops. To speed up such swap-chains, the *i*-Core allows to process a whole swap chain in a single cycle. This is implemented by extending the register file with a virtual-physical address translation table (i.e. when an application uses a register file address, it now uses the virtual address, and the hardware extension translates it into the physical address, which is used to access data from the register file). Swap-chains are then processed by executing a "permute register addresses" instruction, which applies a user-specified permutation describing a chain of swap operation (e.g. r1 $\rightarrow$ r5 $\rightarrow$ r2) to the address translation table during the ID pipeline stage. When extending a compiler to automatically replace chains of swap operations by "permute register addresses" instructions, application performance is improved without the need for application-specific accelerators. [MGMB+13] provides details on the hardware design and implementation and the required compiler extension (the latter was developed at the Programming Paradigms group, IPD, KIT).

2.6.2 Reconfigurable Fabric

When an application wants to speed up a computationally intensive kernel, it needs to load accelerators into the reconfigurable fabric of the *i*-Core. The lower left part of Figure 2.19 shows a detailed view of the fabric. The fabric is partitioned into *Reconfigurable Accelerator Containers* (RACs) (numbered 1..8 in Figure 2.19), fine-grained reconfigurable regions which can be reconfigured at run-time with the

[5]The same stalling mechanism is used as for the LEON-3 multi-cycle divider unit, which is also connected to the pipeline as an additional functional unit.

data-paths of application-specific accelerators. Size of the reconfigurable fabric (in the number of RACs) is a design-time parameter and a trade-off of area vs. performance. Figure 2.20 shows how the components of the *i*-Core may be arranged for a possible tape-out, where only the RACs are implemented as an embedded FPGA (as in [NSBN08; SNBN06]). All other components of the SoC (including the fabric interconnect and support hardware) should be implemented as an ASIC for improved speed, area and power consumption.

Accelerators are loaded into RACs on the fabric by the *Accelerator Loader*, which performs DMA transfers of an accelerator data-path from memory (accelerator are usually part of the application binary, or an accelerator library) to the reconfiguration port of the fabric[6]. To start reconfiguration, an application (or the OS) sends the memory address of the accelerator to the Accelerator Loader and the reconfiguration is performed asynchronously, while the *i*-Core continues with application execution. Once finished, the Accelerator Loader notifies the *i*-Core via an interrupt. The *i*-Core can then load additional accelerators, or mark an SI as "executable on the fabric" in the SI State Table, if the just completed reconfiguration was the last accelerator required by the SI.

The RACs on the fabric are connected by a segmented bus consisting of *Fabric Links*. RACs are not connected directly to the links, but via *Link Connectors*, modules that handle routing to/from the RACs and that contain two small *private Register Files (pRFs)* each (one pRF per in-/output connection between the link connector and the attached RAC). The reason for separating RACs and connectors is that in an ASIC implementation of the reconfigurable fabric (Figure 2.20) only the RACs need to be implemented as reconfigurable logic (i.e. by using an embedded FPGA). The connectors and links would be implemented as more efficient non-reconfigurable logic. Furthermore, this separation allows the connector to be used even while a RAC is being reconfigured.

While RACs (more precisely the accelerators loaded into the RACs) are responsible for the computation part of an SI, communication is handled by the links and storage by the pRFs. Links allow processing of data by multiple accelerators by enabling one accelerator to transfer intermediate results to another. The fabric is synchronous and once an accelerator A has finished computation, its intermediate result is stored in the pRF of the link connector attached to the RAC where A is loaded. In the next cycle, the result can be retrieved from the pRF and sent to a different RAC (or even be further processed by A in the same RAC) for further processing via a link.

A link connector (see Figure 2.21) has link inputs from and outputs to its immediate neighbors and is attached to a RAC. Each link consists of a pair of unidirectional link lines, and all link lines can be used independently from each other. The part of a link line between two neighboring link connectors is called a *link segment*. The link connector can route the value from any link line into its attached RAC for use as an input value by the accelerator loaded into the RAC. The output of an accelerator can

[6] ICAP (Internal Configuration Access Port) for the Xilinx-FPGA based prototype implementation of the *i*-Core

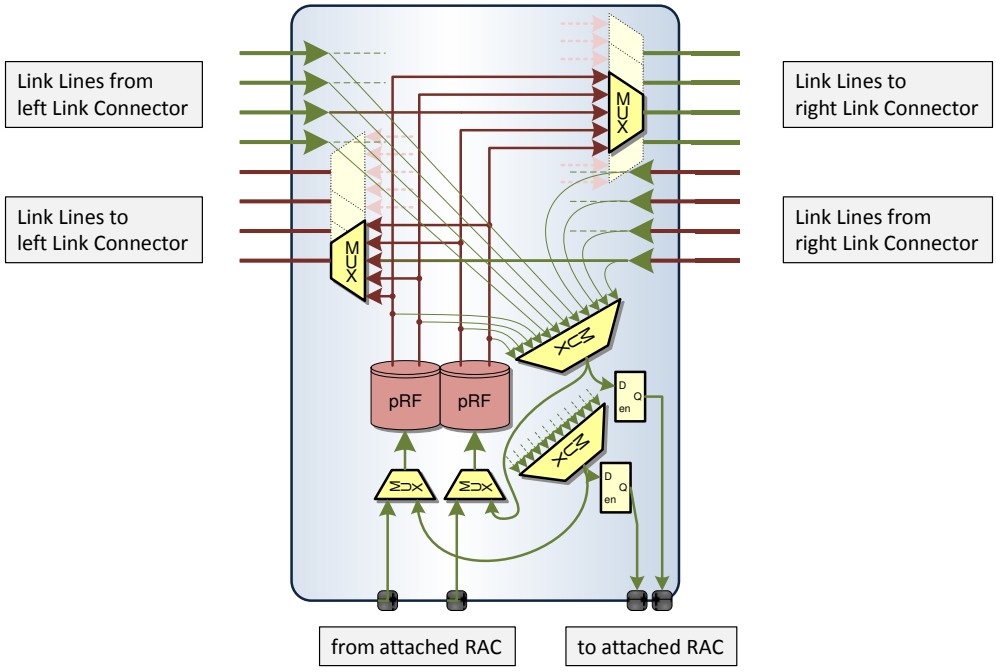


Figure 2.21: Link connector in the *i*-Core fabric, based on the RISPP Bus Connector (modified from [BH11]).

not be written directly onto the link lines (it can only be written into the pRF of the RAC) as that would allow loops (e.g. RAC 1 sends data to RAC 3, which processes it in its accelerator and sends it back to RAC 1, which processes it and sends it to RAC 3 again, etc.), which would make synthesis of the fabric as a synchronous circuit impossible. However, the data from a pRF can be written onto any link by the link connector, thus the results of an accelerator can be sent to any other accelerator one cycle after they have been computed (although this transfer may take longer than one cycle in certain cases, as discussed later in Section 5.3).

In addition to RACs, which are used for reconfigurable accelerators, the fabric also features programmable memory ports (2 independent ports, each 128-bit wide in the *i*-Core implementation). As with a RAC, a memory port is connected to a link connector, although instead of 2 pRFs, the link connector for a memory port has 4 pRFs (and accordingly extended routing capabilities to allow routing of data between fabric links and pRFs). Each memory port is programmed with four parameters: base address, stride, skip and span. These four parameters are used to generate a sequence of addresses which are then used for every future read or write from the memory port (until the memory port is programmed with a new address sequence). The resulting address sequences are based on [CELM+03; LC06] and are flexible enough to access contiguous vectors, striding vectors (only every *n*th element is accessed) and 2-D sub-arrays. The parameters are defined as follows: the base address defines where the address sequence starts. The distance between consecutive accesses is determined by the stride parameter. Span defines how many consecutive

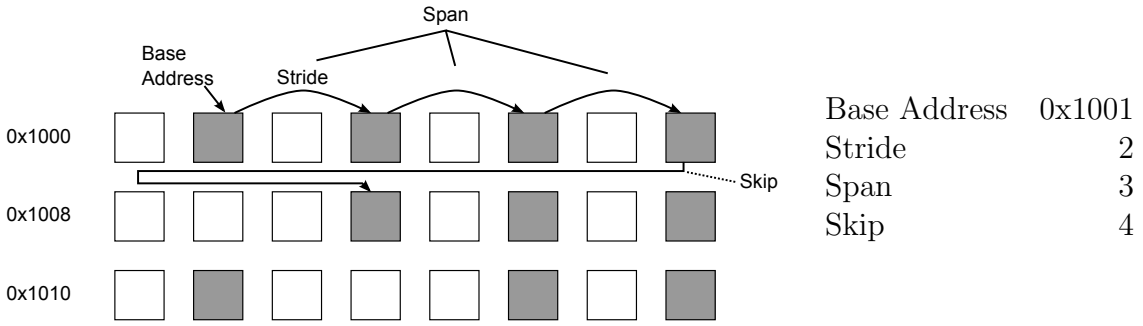


Figure 2.22: Example address pattern generated by programming the memory port with the parameters on the right.

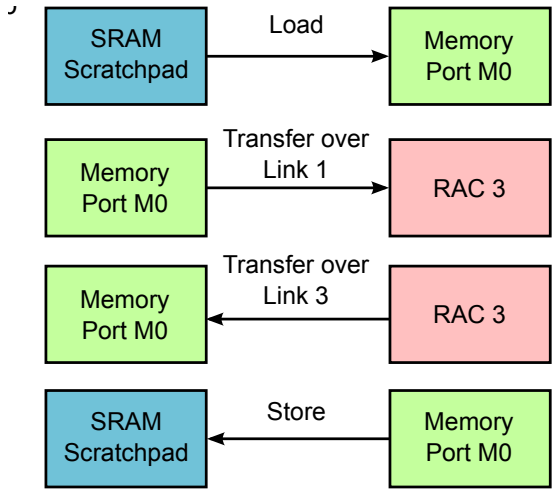
accesses are performed before the value of the skip parameter is added to the current address. An example for an address sequence generated by the memory port is shown in Figure 2.22.

While the RACs of the fabric are fine-grained reconfigurable, multiple fabric resources are coarse-grained reconfigurable, and their configuration can be switched in one cycle: link connectors (reading from/writing to fabric links), pRFs (reading/writing data), memory ports (defining new address sequences, reading and writing). Some accelerators may even feature multiple modes, such as the combined "cadd/csub" accelerator from the FFT SI example in Figure 2.8 supporting both complex addition and subtraction. The fabric supports switching modes of an accelerator without the need for fine-grained reconfiguration (e.g. the complex arithmetic accelerator can perform addition in one cycle, and subtraction in the next)[7]. The coarse-grained configuration of fabric resources is used to implement the data-flow graph of an SI, while the accelerators implement the operations of the SI. The data how to configure the fabric resources in each cycle during SI execution is stored as the *SI μProgram* in the *SI μProgram Memory*. The μProgram is used by the *SI Execution Controller* to control the fabric resources in each cycle while an SI is executed, thus effectively implementing the SI on the fabric.

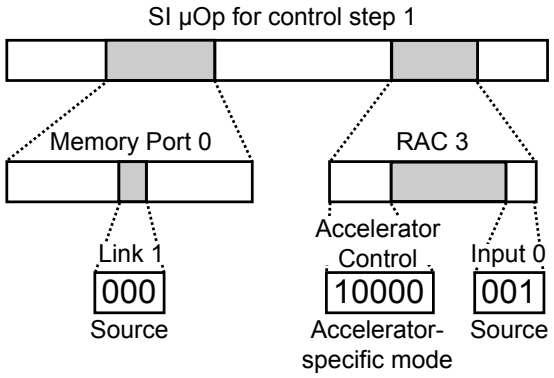
2.6.3 Executing SIs using μPrograms

An SI μProgram consists of a sequence of μOps, where each μOp encodes the configuration of all fabric resources for one particular cycle. μPrograms can either be generated at compile-time (manually or by a toolchain), or automatically at run-time. *i*-Core μPrograms and μOps are similar in concept to *ρμ*-code and *μ*-instructions in MOLEN [VWGB+04] Figure 2.23a shows an example SI that loads data through memory port M0, processes it in an accelerator (which is loaded into RAC 3) and writes the result back through the same memory port (for simplicity, programming

[7] In fact, reconfiguration of the RAC contents while it is being used in an SI is not supported, as typical SI execution takes 10s to 100s of cycles, while RAC reconfiguration takes 10,000s of cycles – reconfiguration during SI execution would therefore prolong SI latency by over 100×, negating any performance benefits that could be gained by using the fabric



(a) Example SI μProgram.



(b) μOp for cycle 1 as used by the SI Execution Controller.

Figure 2.23: Example SI μProgram and μOps.

of the memory port and interaction with pRFs are omitted). A μOp is encoded as a bitvector, which is subdivided into slices. Each slice contains the configuration for one particular fabric resource (e.g. a link connector). Figure 2.23b shows the 2nd μOp for the example SI.

A μProgram is executed by the SI Execution Controller, which is implemented as a finite state machine (FSM), shown in Figure 2.24. When no SI is running, then the fabric resources are held in an "idle" state (no memory port read/write operations, no pRF activity). When an SI is encountered in the ID stage of the core pipeline, in addition to the information if the SI is executable on the fabric, the address and size of the SI μProgram are retrieved from the SI State Table. Once the SI enters the EX stage, the pipeline is stalled, the address and size of the μProgram are sent to the SI Execution Controller and it starts SI execution (FSM "run" state), after setting its internal SI program counter to the starting location of the μProgram in the SI μProgram memory. In each cycle during the "run" state, the SI Execution Controller increments the SI program counter and fetches a new μOp from SI μProgram memory, and outputs it to the fabric, thereby configuring the fabric links, pRFs, link connectors and accelerator-specific modes for the current cycle. Once the SI program counter reaches the value of the (μProgram starting address) + (μProgram size), with the latter is provided in the number of μOps, the SI has been completely processed on the fabric. The SI Execution Controller enters the "finish" state, where the values of the left-bound fabric links of RAC 0 are used as the result of the SI and transferred into the pipeline registers for instruction results in the EX stage[8]. In the following cycle, the SI Execution Controller goes back to the "idle" state and remains there until the next SI has to be executed. The FSM will not perform any transitions if there are outstanding memory operations

[8]RAC 0 does not necessarily process these values, nor does the attached link connector store them in its pRF – the values are simply transferred back to the pipeline.

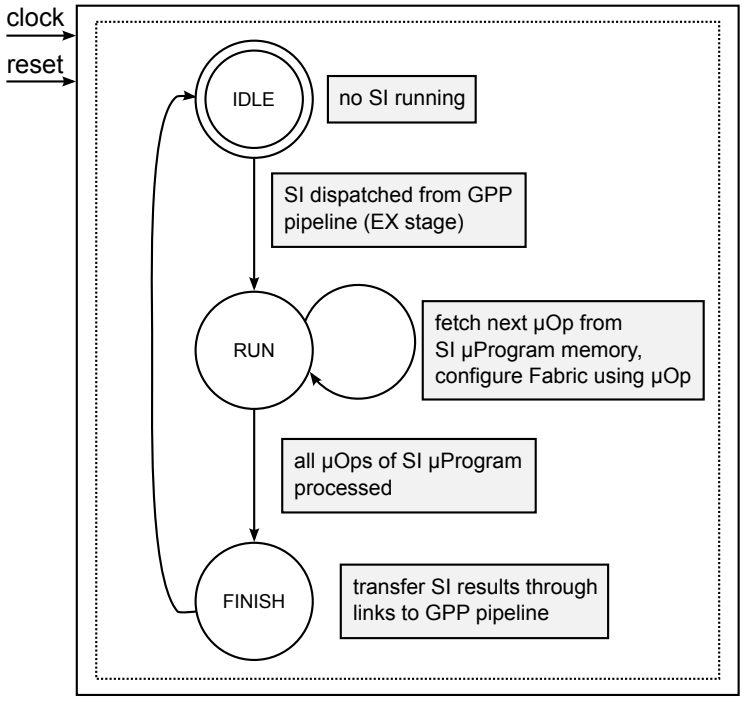


Figure 2.24: SI Execution Controller finite state machine for executing SI μPrograms on the reconfigurable fabric.

from the memory ports of the fabric, e.g. when a read operation to DDR memory has been issued and takes multiple cycles to complete. In such a situation where data is read using a memory port in one μOp (e.g. μOp 0 in Figure 2.23a), and in the next μOp this read data is processed further (μOp 1 in Figure 2.23a), not waiting until the memory port has finished its read operation would result in the SI working on invalid data. The memory ports have dedicated connections to the SI Execution Controller which are used to keep the FSM from transitioning while at least one memory port waits to finish its operation.

2.6.4 The i-Core as Part of the Invasive Computing Many-Core

The Invasive Computing project investigates many-core architectures, programming models and application scenarios [THHS+11]. A central idea is *resource-aware programming*, where a running application can request hardware resources (such as processing elements, memory and interconnect bandwidth), depending on its the phase of the application (e.g. initialization, parallel computation, collecting results). Figure 2.25 shows an example of a simple application running on a many-core platform. Shortly after starting, the application requests ("invades" in the project terminology) on-chip memory and a large amount of interconnect bandwidth to transfer input data from external memory. Next, the application releases ("retreats from") the interconnect resources between the external memory and the on-chip memory, and instead invades several processing elements along with the interconnect

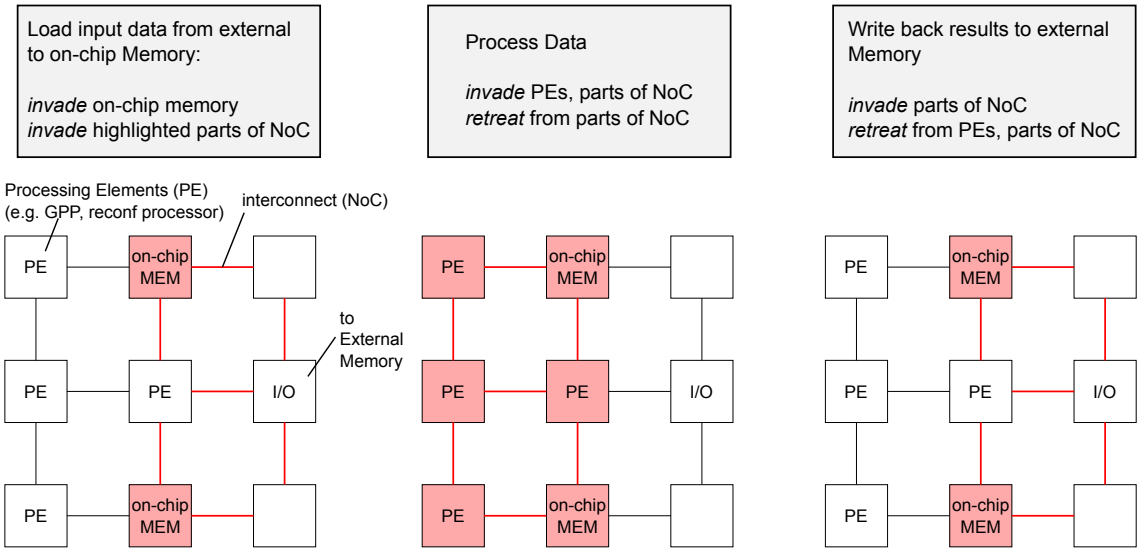


Figure 2.25: Example of an application with three distinctive phases running on an Invasive Computing platform. In each phase the application can request ("invade") and release ("retreat") resources.

between these and the previously invaded on-chip memory, in order to process the input data in parallel. Finally, it retreats from the processing elements, and again request additional interconnect bandwidth to write the results back to external memory.

The many-core is intended for running multiple applications which compete for hardware resources. A distributed and scalable agent-based system for handling resource requests from different applications and finding a suitable resource allocation has been proposed in [Kob15]. This agent-based resource management runs as part of a many-core operating system [OSKB+11].

The many-core architecture [HHBW+12] is shown in detail in Figure 2.26. The figure shows an example configuration of 3x3 tiles. A tile can contain larger amounts of on-chip memory, I/O for interfacing with external memory and peripherals, or computational elements. Computational tiles are shared memory multi-cores that either consist of GPP cores only, or a combination of GPP cores and the i-Core reconfigurable processor. Each of the computational tiles also contains a small amount of on-chip memory (not to be confused with tiles containing large amounts of on-chip memory, but no processing elements). These tiles are suitable for accelerating application parts that exhibit a large degree of task-level and instruction-level parallelism. An additional type of tile is the TCPA [MBHK+12] tile. It is a coarse-grained reconfigurable array that is suitable for streaming applications and accelerating larger kernels (e.g. full loops). The tiles are connected by a network-on-chip, which offers packet-based routing and establishing virtual channels between any two tiles [Hei14]. By invading parts of the NoC, applications can receive guaranteed interconnect bandwidth. In addition to application performance, an additional goal

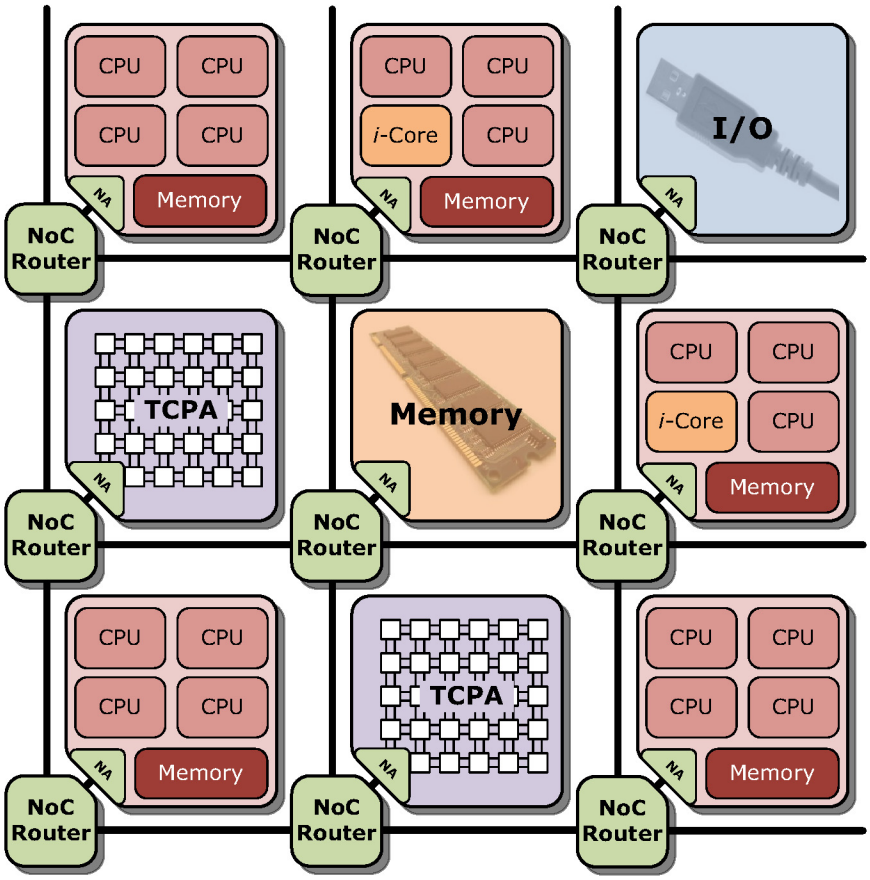


Figure 2.26: Tiled many-core developed in the scope of the Invasive Computing project (from [Hei14]). This 3x3 tile configuration shows the heterogeneous tiles used for computing, storage and I/O, connected by a NoC. the *i*-Core reconfigurable processor is available in two of the tiles.

of Invasive Computing is energy- and power-efficient computing, as explored in [SVH13].

As part of the many-core invasive architecture, the *i*-Core reconfigurable processor can be invaded by applications that use SIs. For example, when entering a computationally intensive phase, an application might ask the resource management system for either 6 GPP cores, or 1 *i*-Core. Depending on resource availability (the *i*-Core may already be running several applications, with little space available on the fabric), the application would either be dispatched to the GPP cores (which would not use SIs, and instead execute its kernels in software), or the *i*-Core, where it would use SIs to accelerate its kernels on the fabric.

In addition to being a resource for applications to invade, the resource-aware computing paradigm has also been applied to managing the fabric of the *i*-Core. When prefetching a kernel, the application can override run-time system decisions by explicitly requesting certain accelerators (and not just SIs) to be loaded onto the fabric. If no such manual fine-tuning is required, the application programmer can

simply specify which SIs will be executed in the coming kernel and leave the rest of fabric management (which SI variants and accelerators to configure) to the run-time system.

2.7 Summary

Reconfigurable processors adapt to the requirements of an application by allowing the application to load application-specific hardware accelerators into a designated component in the processor, the reconfigurable fabric. Different architectures of reconfigurable processors and fabrics have been explored in research, investigating how the fabric can be coupled to the processor (tightly or loosely) and how the fabric itself should be structured (fine-grained or coarse-grained). A common approach is the tight coupling of a general-purpose processor (GPP) core and a reconfigurable fabric, by integrating the fabric into the core pipeline.

An application running on the reconfigurable processor can access the fabric by using an instruction set extension of the GPP core, so called Special Instructions (SIs). An SI describes a data-flow graph of a computationally intensive part (kernel) of the application and allows it to be executed as a multi-cycle operation on the fabric, usually involving multiple accelerators. In order to simplify using the reconfigurable fabric, fabric management is performed by a run-time system. Given a list of SIs that an application wants to use, the run-time system determines which accelerators to load, where and in which order to load them, as well as how to use multiple accelerators to implement complex SIs. The result of fabric management are accelerators loaded onto the fabric, as well as SI μPrograms, which specify cycle by cycle how an SI is to be executed on the fabric. Reconfigurable multi-core and multi-tasking systems are based on these principles for single-core, single-tasking reconfigurable processors.

Existing work for task-scheduling designed for reconfigurable processors only supports systems with inflexible "all-or-nothing" SIs, i.e. either run a kernel on the GPP core (slowly) or use a single compile-time determined fabric allocation to run the kernel in hardware. However, state of the art single-tasking reconfigurable processors allow a kernel to be implemented in different variants, being able to use any amount of fabric allocated to them. Task schedulers for such systems use classic scheduling algorithms that are not aware of the reconfigurable architecture of the system or assume static workloads. Chapter 3 presents specialized task schedulers for reconfigurable processors with flexible SIs and dynamic workloads.

When designing a multi-core system with a reconfigurable fabric, an important question is how the fabric is shared between the cores. Existing approaches either assign a part of the fabric to a specific core, or allow access of multiple cores to the fabric. However, as discussed in more detail in Section 4.2, both approach have restrictions to use the fabric efficiently in a multi-core system, thus in Chapter 4 a new approach to fabric sharing in a multi-core system is presented.

The work in the coming chapters is evaluated on the i-Core reconfigurable processor, which was developed as part of this work and is based on the state of the art RISPP processor. Using a fine-grained reconfigurable fabric tightly coupled to a GPP core with flexible SIs, the processor is a suitable platform to illustrate the concepts of using a reconfigurable processor in multi-tasking and multi-core systems.

3 Multi-Tasking in Reconfigurable Processors

The fine-grained fabric in reconfigurable processors allows a very high degree of adaptivity to application requirements and thereby high system performance, by allowing deployment of application-specific accelerators. The price for this flexibility is the considerable time to reconfigure a fine-grained accelerator, which is in the range of milliseconds, or 100,000s of processor cycles. Assuming the same set of accelerators, when compared to an ASIP, where accelerators are always available, frequent reconfigurations can therefore degrade performance of a reconfigurable processor. Frequent reconfigurations are necessary in highly dynamic applications, where the set of required accelerators depends on input data, or for particularly complex applications, for which all accelerators cannot be loaded onto the fabric at once.

Two points alleviate the overhead of fine-grained reconfiguration:

1. Reconfiguration is not a blocking operation, i.e. the processor can continue executing applications while an accelerator is being loaded.
2. Already loaded accelerators can be used while others reconfigure, as reconfiguration does not inhibit fabric use.

A single application can benefit from this only to a limited degree: 1) would allow the application to proceed, but while the required accelerators are being loaded, application performance is much slower compared when using the fabric with all accelerators available. 2) would allow using SIs for which accelerators are already loaded, but often an application will reconfigure accelerators for exactly those SI that it wishes to use, thus diminishing the benefit.

However, in a multi-tasking workload the overhead of fine-grained reconfiguration can be effectively "hidden", by scheduling tasks in such a way that they run only when (preferably) all of their accelerators have finished loading. While the accelerators are loaded, a different task can be scheduled in order to improve system performance.

This chapter focuses on task scheduling in a reconfigurable processor, with the focus of improving the performance for whole tasksets (instead of single tasks). Two schedulers are proposed, one focussing on soft-deadline workloads, and one on batch workloads with the goal of makespan reduction.

The chapter is organized as follows: Section 3.1 provides a motivation for the performance degradation during accelerator reconfiguration using the example of a

H.264 video encoder, an application that benefits from the adaptivity provided by the fabric, but also incurs frequent reconfigurations. Here, the notion of reconfiguration-induced cycle loss (RiCL) is introduced, which quantifies the performance degradation in an application due to reconfigurations. Section 3.2 provides an overview of the task and system assumptions used throughout the chapter. The problem of reconfiguration-induced performance degradation is formalized in Section 3.3, introducing the metric of task efficiency, which is at the core of the proposed schedulers. The scheduler for soft-deadline workloads, PATS, is presented in Section 3.4 and aims to reduce tardiness (accumulated deadline violation). Section 3.5 presents the second approach, a combined scheduler and fabric allocator, aimed at non-deadline tasks, with the goal of reducing overall makespan. A case study for each of the schedulers is presented in their respective sections, providing an in-depth view of their scheduling decisions when compared to existing approaches.

3.1 Motivation

Video encoders are an example for applications that when run on a reconfigurable processor, achieve significant speedups ([MBTC06; BSH08b; LSV06] report 3.6×–7.2×). A simplified flow of an H.264 video encoder is shown in Figure 3.1. To encode one frame, raw video data is processed by a sequence of kernels. Each of these kernels consists of one or more SIs (there are several hundreds to thousands SI executions per kernel), each of which can be used in different variants (same functionality, but trading off area vs. performance), depending on the number of RACs on the fabric. In the H.264 implementation for the *i*-Core reconfigurable processor (see Section 2.6), the first kernel "Motion Estimation" can use up to 9 RACs, the second kernel "Encoding Engine" up to 11 RACs, and the third kernel "Loop Filter" up to 4 RACs. Fitting all required accelerators onto the fabric would (i) require a very large fabric, and (ii) would make it unusable for other applications running at the same time as the encoder (executed either on the same core, or on different cores sharing the same fabric, as presented in Chapter 4).

Thus, before a new kernel starts, the encoder performs a prefetch, providing the list of SIs that will be used in the coming kernel (see Section 2.3). For this prefetch, the run-time system determines the *SI variant* for each SI that will be available once the prefetch is complete. The run-time system also determines the accelerators that need to be loaded onto the fabric to implement the requested SI variants. Before all the accelerators for a requested SI variant are loaded, the SI can still be run in a different, but slower variant (if no accelerators are available in the cISA implementation). For each SI, there is a latency difference between the currently available SI variant (e.g. a high-latency variant using few RACs) and the requested SI variant (a low-latency variant using more RACs) , while the prefetch is running. This latency difference is caused by the long reconfiguration time of accelerators. The *Reconfiguration-induced Cycle Loss* (RiCL) of an application is then this latency difference accumulated over

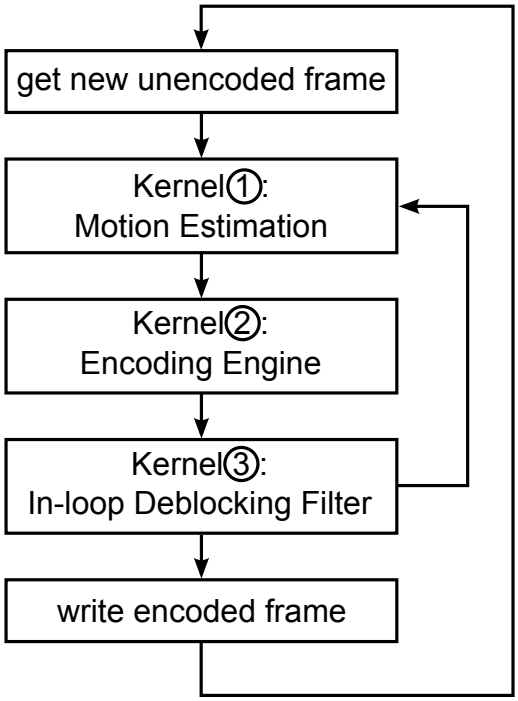


Figure 3.1: Flow of H.264 video encoder, showing its dynamic nature. 3 different kernels are continuously executed in sequence, resulting in a large amount

all SI executions during the time an application runs, indicating the performance loss of the application due to accelerator reconfigurations.

Figure 3.2 shows RiCL during encoding of one frame by the H.264 video encoder when executed on the *i*-Core. ①, ② and ③ mark the starts of the prefetches on the x-axis for the respective kernel. The height of a bar at a particular point on the x-axis corresponds to the RiCL of the application in the current time frame. Immediately after the prefetch for kernel ① starts, RiCL is highest, as no accelerators from the previously executed kernel ③ can be reused for the current kernel, and thus the currently available SI variants are slow. Once reconfigurations for the prefetch are completed, RiCL drops until it reaches 0 when the prefetch is complete, at which time the requested SI variants are identical to the available SI variants. The high RiCL shown exhibited by the encoder is representative of all applications that are composed of multiple kernels, as switching between kernels requires reconfigurations.

An application exhibiting a high cumulative RiCL will take longer to finish than the same application with a low cumulative RiCL. In Figure 3.2, the time between the vertical bars marked ① is the time to encode 1 frame. If RiCL were reduced to 0, the time to encode 1 frame would be significantly reduced: encoding would still start at the leftmost ①, but would be finished at the dashed green line, representing a speedup of 1.35×.

If the system does not support multi-tasking, or only one task is executable, then RiCL can not be improved without redesigning the hardware. As RiCL depends on the time required to complete the reconfiguration of an accelerator (the slower

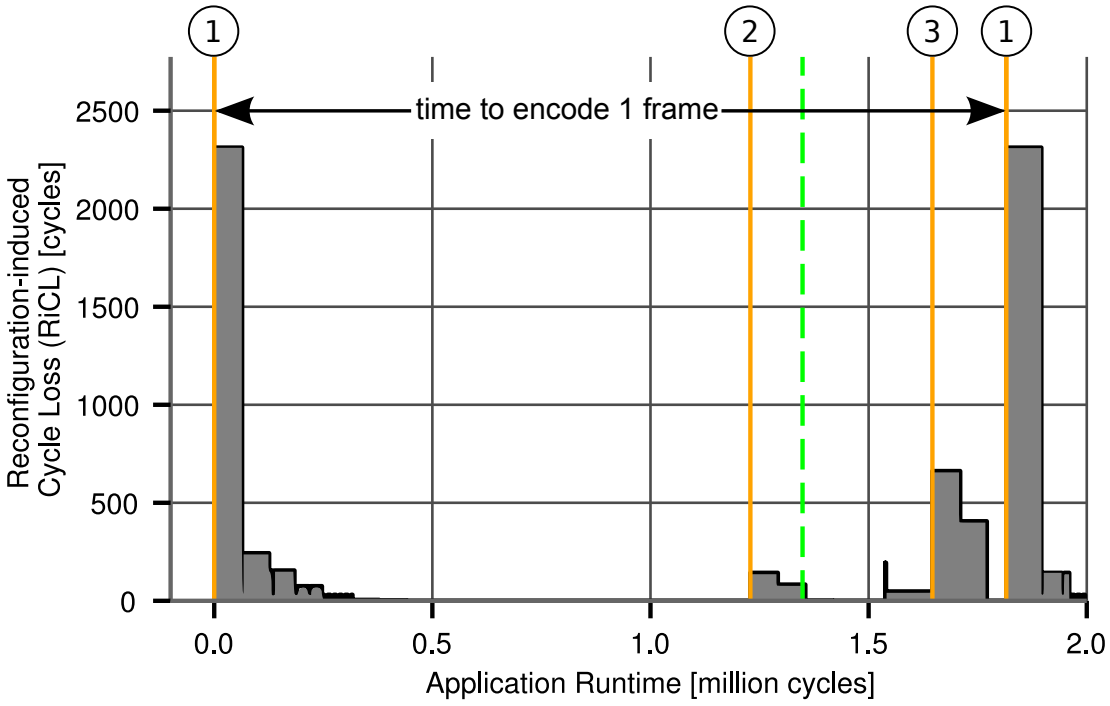


Figure 3.2: Performance loss in the H.264 video encoder due to reconfiguration overhead. The green dashed line shows when encoding would be finished if RiCL was zero.

reconfigurations are performed, the higher RiCL), this would entail increasing the reconfiguration bandwidth. However, if the task scheduler has multiple candidates that are executable, then a different task B could be scheduled during the time that RiCL is high for task A. This would effectively *hide* the reconfiguration overhead of task A.

The task schedulers proposed in this chapter aim to reduce RiCL. However, tracking RiCL in a real system is not feasible, as that would require monitoring every SI execution, computing the difference between the requested SI candidate and incrementing appropriate counters, requiring significant hardware overhead for a monitoring system. Instead, the notion of *task efficiency* will be introduced in Section 3.3 to describe the effects behind RiCL in a more manageable way.

3.2 System Overview

The workloads used for both schedulers are dynamic, i.e. new tasks can arrive at any time. This models the behavior of systems with user interaction, such as desktop or mobile systems, where applications can be started by the user or by different parts of the operating system at any time. To handle such dynamic workloads, the proposed techniques are online schedulers.

As the proposed schedulers exploit the reconfiguration hiding effect, the accelerators are assumed to be fine-grained and take a significant amount of time to reconfigure ($> 10,000$ processor cycles). For coarse-grained fabrics the techniques may still be applicable, as long as reconfiguration takes significant time. This may be the case if processing a kernel on a coarse-grained fabric requires loading of a large μProgram in addition to configuring the accelerators.

Scheduling decisions are performed by the operating system. This can either be a full-fledged OS, such as GNU/Linux, or a trimmed down OS optimized for small embedded systems. The task scheduler has to support preemptive scheduling, i.e. the currently executing task can be interrupted at any time by the operating system, so that the task scheduler can decide which task to run next, and switch to a different task if required. Management of the reconfigurable fabric, esp. the distribution of RACs among tasks is either handled by a user-space run-time system, or the OS kernel. Whether an SI should be run on the fabric or if the equivalent code is executed on the core pipeline, can either be decided by the run-time system, or by the application itself.

3.3 Metrics

A general overview of task scheduling was provided in Section 2.5. This section introduces the metrics and task characteristics that were developed for the reconfigurable schedulers presented later in this chapter.

Applications that run on a reconfigurable processor can be categorized into three classes, depending on their kernel execution behavior:

MKT Multi-Kernel Task. An MKT executes more than one hardware accelerated kernel over its lifetime, issuing a prefetch before each kernel, and thus generally triggering reconfigurations before each kernel. The kernel execution can happen periodically, e.g. the H.264 video encoder example in Figure 3.2 executes three different kernels sequentially to process a frame.

SKT Single-Kernel Task. An SKT executes one single hardware accelerated kernel (the same kernel can be executed multiple times) and issues one prefetch for it (typically a short time after the application starts), and thus no further reconfigurations are required after the initial prefetch until the task ends.

ZKT Zero-Kernel Task. The kernels of a ZKT (or "Software" task) are executed on the core pipeline, and thus ZKTs do not issue any prefetches or SIs. After an MKT or SKT executes its last prefetch, it behaves like a ZKT.

This categorization is done offline by the application developer, or with a profiling tool.

In MKTs, for each kernel that performs a prefetch i, the *Average Time Between Prefetches (ATBP)* determines the time between prefetch i and the following prefetch $i+1$. ATBP is determined offline by a profiling tool examining the time between each

type of kernel (e.g. for the H.264 Encoder there would be three ATBP values: ATBP for "Motion Estimation"→"Encoding Engine", ATBP for "Encoding Engine"→"Loop Filter" and ATBP for "Loop Filter"→"Motion Estimation"). ATBP is only required for MKTs (as SKTs only issue one prefetch).

In order to describe how much a task benefits from the reconfigurable fabric at the current point in time, the notion of *task efficiency* is introduced. Task efficiency is then used as the base for the schedulers presented in the remainder of this chapter. The remainder of this section will derive task efficiency.

ZKTs run at the same speed for all fabric sizes (i.e. number of RACs), while SKTs and MKTs benefit from increasing fabric sizes up to a saturation point (as is shown in Section 6.2, Figure 6.4). This fabric-size dependent characteristic is called *task performance*. Task performance is profiled offline (along with ATBP) by executing the task alone (i.e. no multi-tasking) on different fabric sizes (i.e. by assigning the task different numbers of RACs).

The relative task performance $RP_{i,n}$ of task i on a fabric of n RACs is defined in Equation (3.1), where $C_{i,j}$ is the completion time of i, when executing task i in single-tasking mode on a fabric of size j.

$$RP_{i,n} = \frac{C_{i,0}}{C_{i,n}} \tag{3.1}$$

Relative task performance is normalized into the range $[0, 1]$ once the task enters the system to allow comparing performance of different tasks. The normalized task performance $P_{i,n}$ is shown in Equation (3.2), where N is the fabric size of the actual system.

$$P_{i,n} = \begin{cases} 1 & \text{if } n \geq N \\ \dfrac{RP_{i,n}}{RP_{i,N}} & \text{otherwise} \end{cases} \tag{3.2}$$

Tasks that use the reconfigurable fabric (i.e. MKT and SKT) run at different speeds depending on how many of the prefetched accelerators have already finished reconfiguration. When no prefetched accelerators are available (e.g. a short time after the prefetch was issued) the task execution speed is low, which leads to a large RiCL (as shown in Section 3.1). In order to compare different execution speeds between different tasks, the relative *task efficiency* metric is introduced.

Task efficiency $E_{T,K}(f)$ for a fabric configuration f (i.e. which accelerator is loaded into which RAC on the fabric) is defined as follows: After the prefetch for the current kernel K for a task T has completed, i.e. there are no pending reconfigurations for this task, the task efficiency for T is defined to be 1. If some reconfigurations are not yet completed for T, then not all SIs of the currently executing kernel K can be executed in the requested variant (as decided by the run-time system or directly determined by the prefetch instruction) and the latency for these SIs is higher. Let $L_{T,S}^{avail}(f)$ be the latency for the currently available variant for an SI S of task T. Similarly, $L_{T,S}^{req}$ is the latency for S once the prefetch is complete with all required

accelerators for S loaded, and $L_{T,S}^{worst}$ the latency of S if it is executed in software on the core pipeline[1]. $\Delta L_{T,K}(f)$ (defined in Equation (3.3)) is then the accumulated latency improvement that will be achieved by finishing the current prefetch, weighted by the SI execution frequencies w_S (which are gathered from offline profiling and online-monitoring during program execution).

$$\Delta L_{T,K}(f) = \sum_{\text{SIs} S \in K} w_S (L_{T,S}^{avail}(f) - L_{T,S}^{req}) \tag{3.3}$$

By putting $\Delta L_{T,K}(f)$ in relation to $L_{T,K}^{worst}$ (Equation (3.4)), the accumulated SI latencies for software execution (again, weighted by the SI execution frequencies w_S), task efficiency $E_{T,K}(f)$ can be computed as in Equation (3.5).

$$Ł_{T,K}^{worst} = \sum_{\text{SIs } S \in K} w_S L_{T,S}^{worst} \tag{3.4}$$

$$E_{T,K}(f) = 1 - \frac{\Delta L_{T,K}(f)}{L_{T,K}^{worst}} \tag{3.5}$$

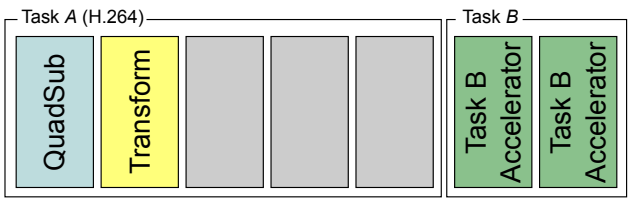
The difference between task performance and task efficiency is that task performance is a static characteristic (usually implemented as a lookup table), while task efficiency is a dynamic value, changing with each prefetch and with each reconfiguration that loads or removes accelerators beneficial for the currently executing kernel. In short, task efficiency measures how close a task is to its maximum achievable speed on a normalized scale, allowing comparison between efficiencies of different tasks, which is used for task scheduling later in this chapter.

If a task does not run at an efficiency of 1.0, then $\Delta L_{T,K}(f)$ is larger than 0. $\Delta L_{T,K}(f)$ is therefore similar to the RiCL introduced in Section 3.1. The RiCL metric denotes how many cycles a task looses accumulated over all its SIs over all its execution time due to reconfiguration delays compared to a hypothetical situation where the accelerators that are requested by a prefetch would be immediately available (i.e. under the assumption that the reconfiguration would happen instantaneously). The difference between RiCL and $\Delta L_{T,K}(f)$ is that RiCL is observed *after* a task or kernel has finished execution, while $\Delta L_{T,K}(f)$ is a *prediction* (based on estimated SI weights, which are derived from prior execution of the same kernel and offline profiling) of RiCL before a kernel starts

Reducing RiCL results in performance improvement and is therefore at the core of the scheduling strategies proposed in this section.

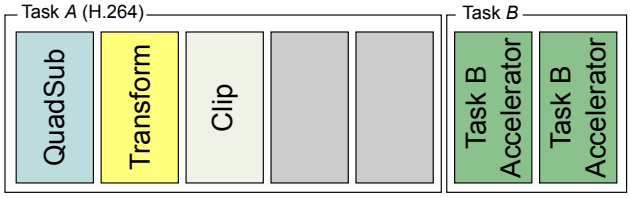
Figure 3.3 illustrates how task efficiency changes during a prefetch for a kernel. Figures 3.3a to 3.3d show the configuration of the fabric (images on the left) and the

[1] $L_{T,S}^{worst}$ and $L_{T,S}^{req}$ are not dependent on a particular fabric configuration. $L_{T,S}^{req}$ is dependent on the prefetch which, depending on run-time system design, may take fabric configuration into account when selecting SI variants. In general however, both latencies are independent of f.



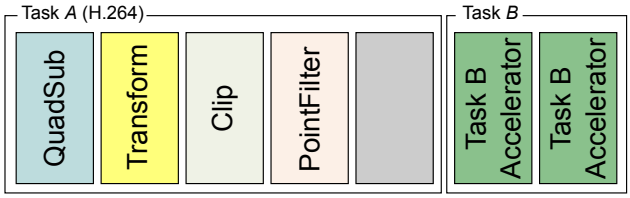
SI	Latency		
	Current	Required	Worst
DCT	21	19	306
HT	14	10	201
MCHZ	166	12	166
Task Efficiency		0.76	

(a) Fabric configuration immediately after starting the prefetch for the Encoding Engine kernel. Remaining accelerators to be loaded: "Clip", "PointFilter", "Transform".



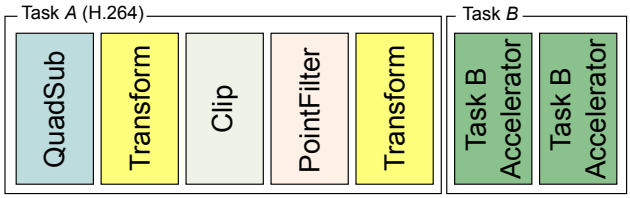
DCT	21	19	306
HT	14	10	201
MCHZ	109	12	166
Task Efficiency		0.85	

(b) Fabric configuration after loading a "Clip" accelerator. Remaining accelerators: "PointFilter", "Transform".



DCT	21	19	306
HT	14	10	201
MCHZ	12	12	166
Task Efficiency		0.99	

(c) Fabric configuration after loading a "PointFilter" accelerator. Remaining accelerators: "Transform".



DCT	19	19	306
HT	10	10	201
MCHZ	12	12	166
Task Efficiency		1.00	

(d) Fabric configuration after loading a "Transform" accelerator. All accelerators for the kernel have finished loading.

Figure 3.3: Varying task efficiency for an H.264 video encoder during loading of accelerators for the kernel "Encoding Engine".

corresponding task and SI efficiencies (tables on the right) after each reconfiguration. For simplicity's sake, the weights of all SIs are assumed to be 1. In the example, a H.264 video encoder task is using 5 RACs of the reconfigurable fabric (2 further RACs are reserved for a different task B). In Figure 3.3a the task issues a prefetch for the "Encoding Engine" kernel (this example uses a simplified version with fewer SIs than the actual application). Two accelerators ("QuadSub" and "Transform") have been already loaded by a previous kernel, and the Encoding Engine SIs can use them immediately (thus DCT and HT already have a very low latency). Three more accelerators need to be loaded until the task reaches full efficiency. Each subsequent

reconfiguration increases task efficiency, although the amount by which it is increased is not constant (the first reconfiguration increases efficiency by 0.09, the second by 0.14 and the third only by 0.01).

The schedulers in the following two sections are based on the concept of task efficiency and RiCL reduction.

3.4 Scheduler for Soft-Deadline Workloads

In addition to the system model discussed in Section 3.2, the scheduler presented in this section is intended for a system with the following assumptions:

1. The workloads consisting of both periodic and aperiodic tasks with soft deadlines.
2. A periodic tasks yields (suspends itself) only if it has finished its current job (i.e. tasks do not yield voluntarily).
3. There is no jitter on job release (i.e. when a task arrives, it is immediately ready to execute).

As new tasks can enter the system at any time, the proposed scheduler is an online algorithm. The goal is to minimize overall tardiness, i.e. the sum of processor cycles by which deadlines have been violated (if any were).

The proposed task scheduler, PATS (Performance-Aware Task Scheduler), is based on the concept of task efficiency (Section 3.3). It is built upon existing schedulers for workloads with deadlines, in particular Earliest Deadline First (EDF) and Rate-Monotonic Scheduling, which are both commonly used schedulers for non-reconfigurable systems. The overall ideal is to keep the task efficiency of the running task high by switching to a different task if a drop in efficiency is detected for the currently running task. PATS is responsible only for selecting the running task, not for fabric allocation (i.e. assigning RACs to tasks), which is handled by other parts of the OS or run-time system (see Section 2.3.1) for existing fabric allocation approaches).

Figure 3.4 shows a flowchart of PATS. The scheduler is invoked if (i) a time slice has finished, (ii) a task has finished its job and used the "yield" system call, (iii) a task has issued a prefetch, requesting a new set of SIs (due to the start of a new kernel), or (iv) a new job was released for a task. PATS then determines which task to run next. The scheduler consists of 2 steps: Current Task Parking and Candidate Scoring/Next Task selection.

Current Task Parking The scheduler manages tasks using two task queues, where tasks are inserted depending on whether a job has been released or not. These queues are: *NRQ* (Not Released Queue) which contains periodic tasks that cannot be run, as the previous job (if any) has finished and the next job is not released yet and *RQ* (Runnable Queue), which contains tasks that either have a job ready, or are

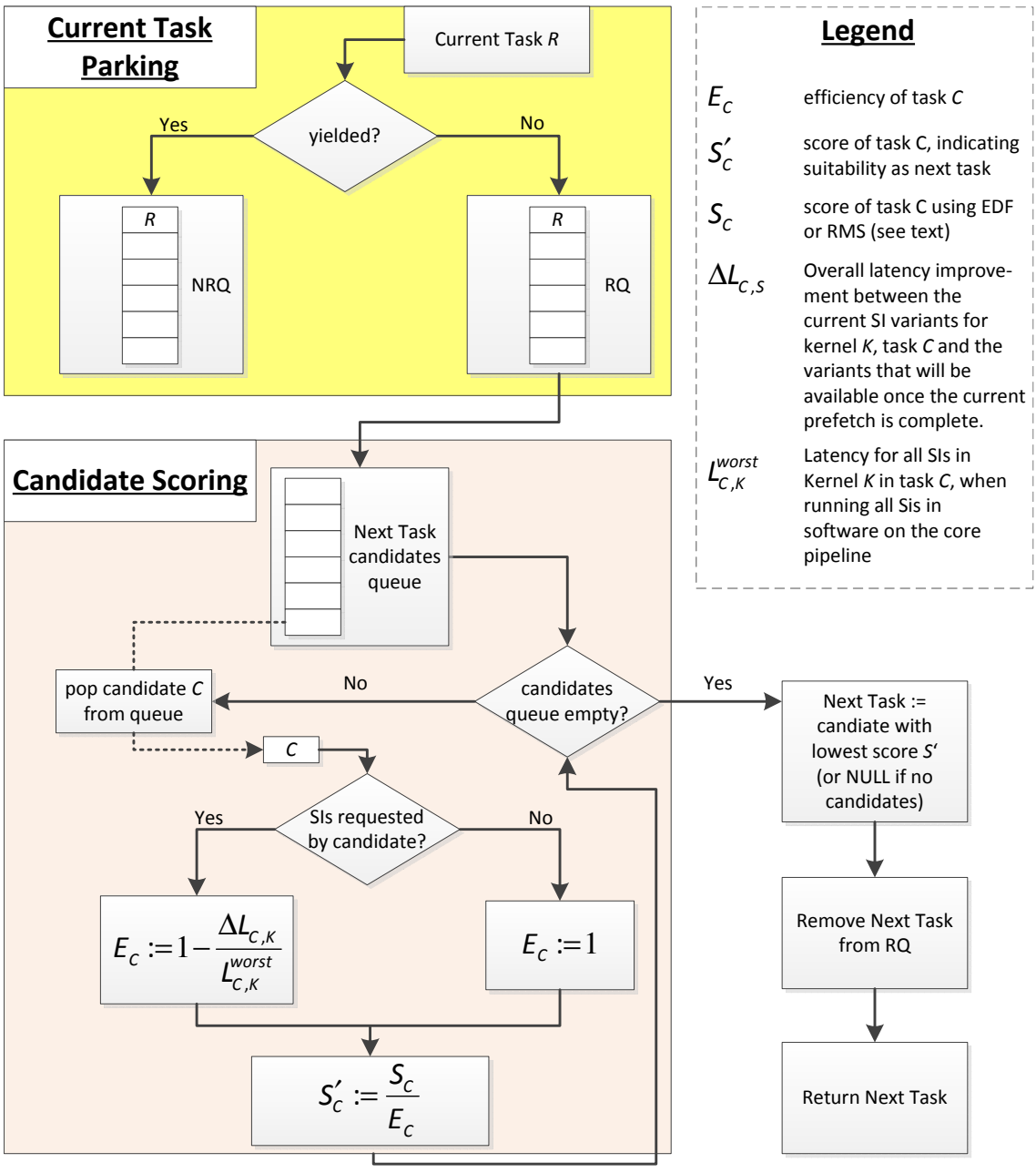


Figure 3.4: Performance Aware Task Scheduler.

aperiodic and have not finished yet. When started, PATS moves the current task R into one of these two task queues. If R has yielded, it has finished its job and is inserted into NRQ, where it waits for its next job to be released. Otherwise, R is inserted into RQ and is a potential candidate for the next task.

Candidate Scoring The tasks in RQ are candidates for the next task. Each of the candidate tasks is scored, and the task with the *lowest* score is selected as the next

task[2]. The score S'_C for a task C is computed by dividing the base score S_C of the candidate by the task efficiency of the candidate. The base score is computed in an EDF-like or RMS-like fashion. As EDF can efficiently utilize up to 100% of the processor for non-reconfigurable systems (as long as all deadlines are met), EDF-like scoring is used if no deadline violations occurred in a parametrizable timeframe (e.g. in the last 100 ms). If deadline violations occurred in the timeframe, RMS-like scoring is used, the system is overloaded and RMS-like scoring is used instead, as EDF behaves unpredictably for overloaded systems. Equation (3.6) shows how the candidate score is computed, where $T_C^{next\_DL}$ is the next deadline of task T_C and T_C^{period} is its period.

ZKTs or tasks that are not currently executing a kernel have an efficiency of 1, for all other tasks efficiencies are computed according to Equation (3.5). Task efficiencies are computed for the fabric configuration at the time the scheduler is invoked and for those kernels that each task is currently executing, thus in Figure 3.4 E_C is used instead of $E_{C,K}(f)$, as f and K are fixed.

$$S'_C := \begin{cases} \dfrac{T_C^{next\_DL} - t}{E_C} & \text{if no deadlines violated in last timeframe (EDF-like scoring)} \\ \dfrac{1/T_C^{period}}{E_C} & \text{otherwise (RMS-like scoring)} \end{cases} \tag{3.6}$$

If all candidates have an efficiency of 1, PATS behaves the same as EDF or RMS and selects the task with closest deadline. If a candidate has an efficiency < 1, it will have an increased score, allowing other tasks to be scheduled first while the low-efficiency candidate finishes its prefetch. The low efficiency candidate can then be scheduled in a later invocation of PATS, as its efficiency will be higher and its deadline will be closer.

3.4.1 Case Study

In the following case study the behavior of PATS will be compared to EDF for one specific workload and fabric size. Results for multiple benchmarks and different fabric configurations are presented in Section 6.3. The results are obtained using a cycle-accurate simulator (see Section 6.2) of a single-core system with an i-Core reconfigurable processor and a reconfigurable fabric consisting of 10 RACs.

The workload consists of 3 applications, 2 of which (Task A and Task B) are periodic and use the reconfigurable fabric, while Task C is software-only and is aperiodic (Table 3.1) with a very large deadline. While the application for both tasks A and B is the same, for this scenario the system has been configured so that a task may only use accelerators in the RACs that it has been assigned (thus task B may not use an

[2] if RQ is empty, then PATS returns NULL and the OS can schedule the idle task, put the processor into a low-power state, etc.

Task	Application	Class	RACs used	deadline [ms]	# jobs
Task A	H.264 video encoder	MKT	5	40	20
Task B	H.264 video encoder	MKT	5	40	20
Task C	SHA	ZKT	0	100,000	1

Table 3.1: Workload for EDF and PATS case study

accelerator that has been loaded by task A). Therefore tasks A and B are independent and any other MKT applications could be used instead of any of them.

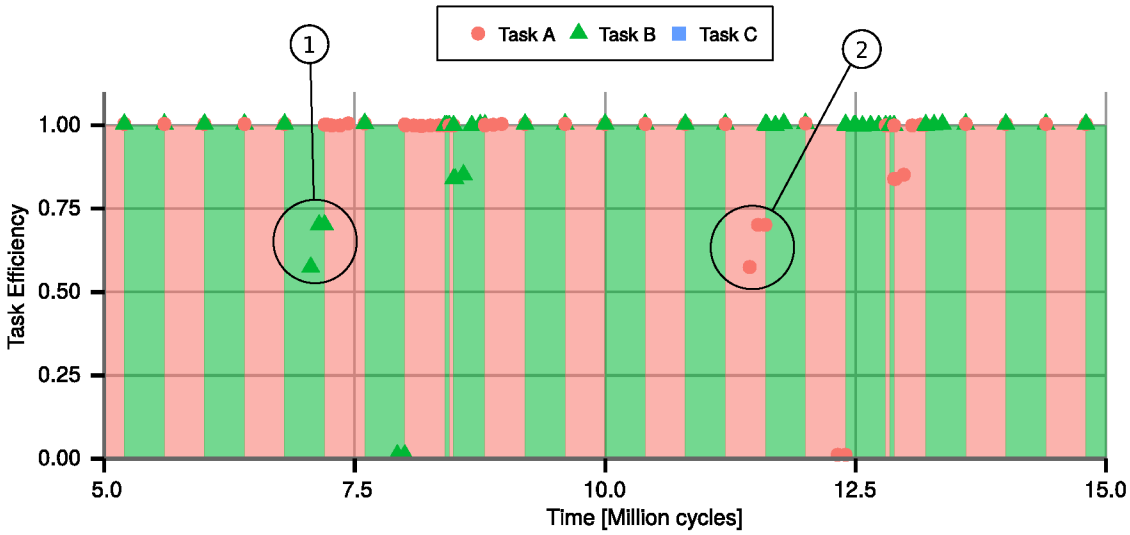
This workload represents a typical scenario in desktops: There are one or two demanding "foreground" tasks (such as video en-/decoding, games, etc.) with a periodic behavior and soft deadlines (encode/decode a frame to reach a minimal FPS rate) and one or more background tasks (such as backup tasks, virus scanners, file system indexing, application/OS updates) which are not periodic and can be interrupted without compromising quality of service. These background tasks can therefore be scheduled in a best-effort manner, but it is preferable that they finish as soon as possible. A good scheduling strategy should therefore ensure that the deadlines of the periodic tasks A and B are met (and if not, then minimize the accumulated tardiness), and as a secondary goal minimize the execution time of the aperiodic task C.

When scheduling this workload with EDF, not all deadlines are met, and the accumulated tardiness is 154,6 million cycles. Task C finishes 128 million cycles after start. PATS meets all deadlines when scheduling this workload (tardiness = 0 cycles) and finishes Task C 115 million cycles after start (i.e. 11% faster than EDF).

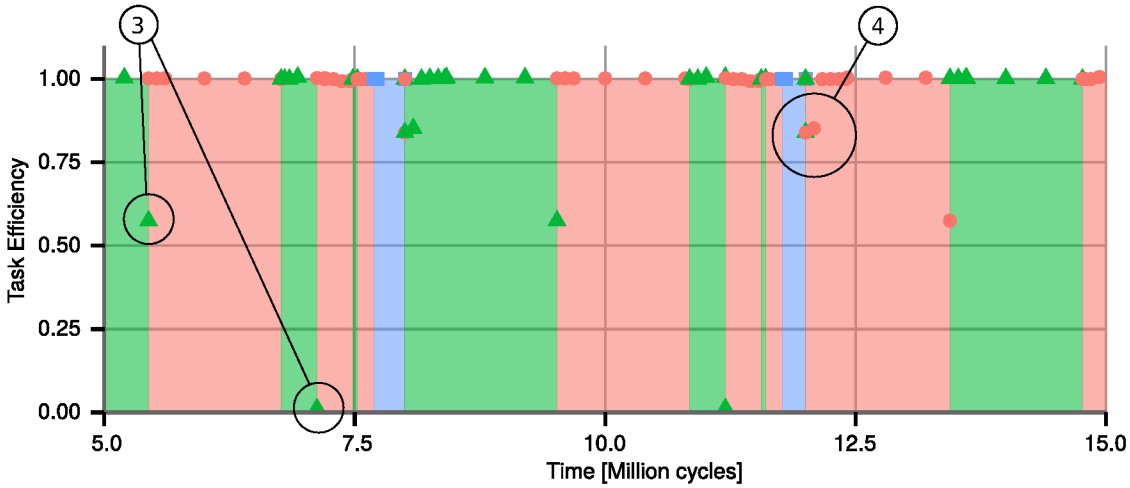
Figure 3.5 shows an excerpt of the scheduling decisions made by EDF and PATS. The colored stripes indicate task schedule phases, i.e. what task was running at this time. The small symbols (such as in the circled area ①) indicate the task efficiency that a task has at a point in time.

A task running at an efficiency of 1.0 will finish its job in less time than at any lower efficiency, thus tasks that have a low efficiency can lead to missing deadlines and a later completion time for aperiodic tasks. The EDF schedule shows drops in task efficiency in the middle of a task schedule phase, such as in ① and ②. Such drops are caused by the start of a new kernel, and the subsequent loading of accelerators. Until all accelerators are loaded, task efficiency is lowered. Lowered efficiency is also visible in the trace for PATS scheduling. However, when low efficiency is observed, PATS switches to a different high efficiency task, thus the this low efficiency is generally at the end of a schedule phase, such as at ③. Sometimes a close deadline will force PATS to schedule a task at lower efficiency, such as at ④. In these instances combining the deadline with task efficiency allows PATS to delay scheduling such tasks (thus running the previous task at higher efficiency) while still meeting the deadlines.

The result is that tasks scheduled by PATS finish their jobs earlier, allowing both to meet the tight deadlines and in addition to schedule the background task (blue



(a) EDF scheduler.



(b) PATS scheduler.

Figure 3.5: Schedule and efficiency trace on a reconfigurable processor for a 3 task workload.

stripes in Figure 3.5b), while EDF misses the deadlines and thus can not afford to schedule the background task (no blue stripes in the Figure 3.5a).

3.5 Scheduler and Fabric Allocator for Makespan Optimization

The scheduler presented in this section has the goal of minimizing the makespan, and thereafter named Makespan Optimizer for Reconfigurable Systems, MORP. The workload consists of aperiodic tasks with no deadlines. The system model discussed

in Section 3.2 applies as well, with new tasks being able to enter the system at any time, therefore an online algorithm is used. For such dynamic workloads makespan is applied to the time between the release of the first task until no more tasks remain to be scheduled.

Unlike the PATS scheduler discussed in the previous section, MORP also handles fabric allocation, i.e. assigning RACs to tasks. MORP also requires OS support for dynamic time slices (to allow triggering MORP at a configurable time in the future).

The overall idea is to hide the prefetches of one task, called the *primary task*, by executing a different task, called the *secondary task* while the reconfigurations for the primary task are performed. To do so, MORP selects a secondary task when the primary task is approaching its next kernel, i.e. its next prefetch. Tasks that do not execute SIs (i.e. ZKTs) or tasks that reach a high task efficiency with few RACs are good candidates for secondary tasks. A secondary task may also receive a small amount of reconfigurable containers (called *reallocation*) from the primary task to run much faster, at the cost of slightly reduced performance of the primary task. When the primary task issues its next prefetch, the system switches to the secondary task while the prefetch for the primary task is performed. The RiCL of the primary task is reduced (as its reconfiguration time is hidden by the secondary task) without significantly increasing the RiCL of the secondary task, thus a net reduction of RiCL is achieved and thereby the makespan is reduced.

During compilation and profiling, each program is annotated with the ATBP (average time between prefetches, see Section 3.3), which is used to trigger MORP *before* a prefetch starts. The basic time unit on which MORP operates is the average time required to load one accelerator. In the following explanation (and in Figure 3.6), time specifications such as "N reconfigurations" mean "the time required to load N accelerators".

Figure 3.6 shows the flow-chart for MORP. The system is usually either in the *Primary Task Execution* state or the *Secondary Task Execution* state. First, a primary task pt is selected from the current taskset T. As MORP aims to reduce RiCL of the primary task, it is preferably a MKT characterized by a high RiCL. The *Select Primary Task* function selects pt from taskset T as follows:

- Select MKT with highest RiCL (gathered from offline profiling) from T. If no MKTs are left in T, then
- Select SKT with highest RiCL from T. If no SKTs are left in T, then
- Select any ZKT from T. If no ZKTs are left in T, then taskset T has been finished, wait for new tasks.

The system allocates the entire fabric to pt, starts it, and enters the *Primary Task Execution* state (see Figure 3.6).

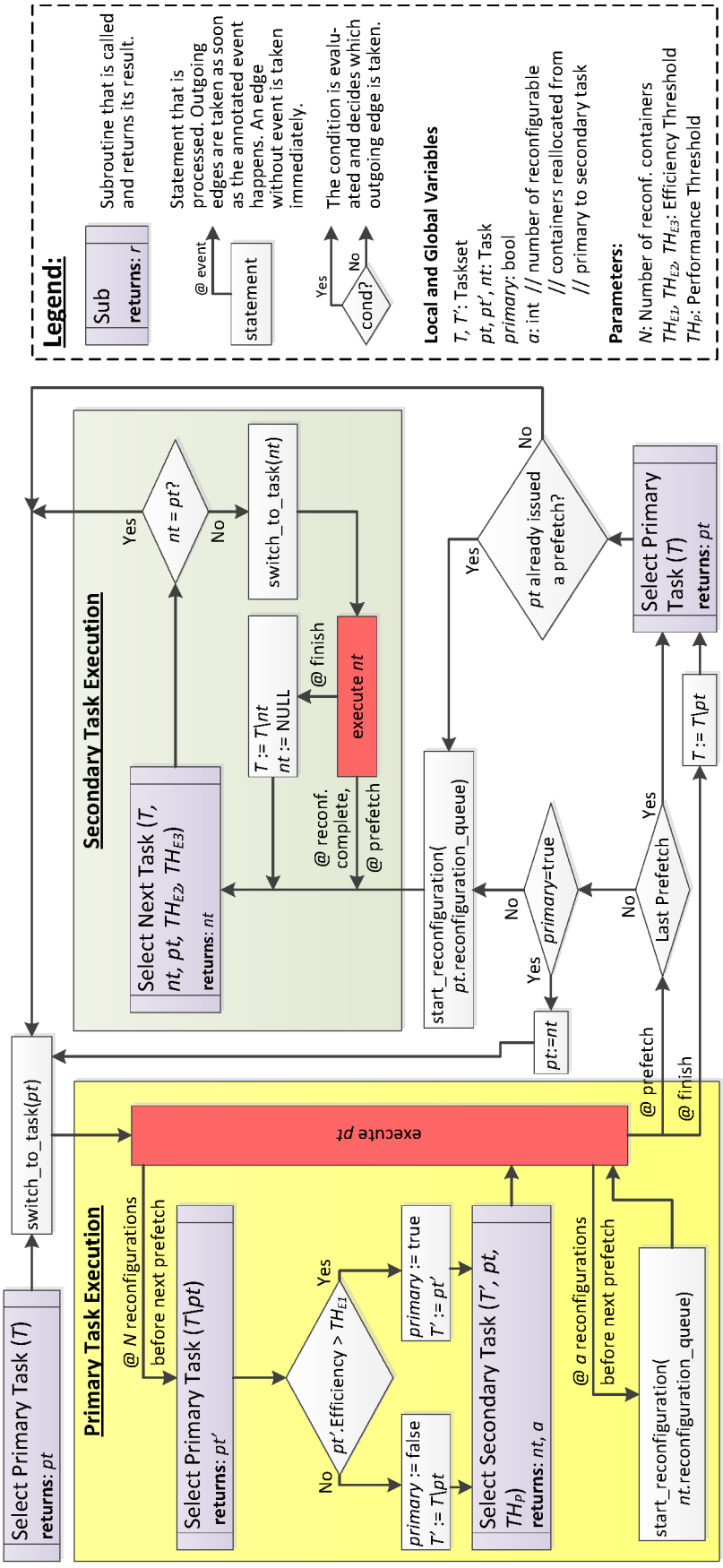


Figure 3.6: Overview of Task Scheduling, Reconfiguration Sequence, and Dynamic Fabric Re-Distribution for Primary and Secondary Tasks.

Primary Task Execution The scheduler is triggered N reconfigurations before the next prefetch of pt (by using the ATBP of the primary task pt) in order to decide whether to continue executing pt or switch to a different primary task pt'. The value of N, which determines when these decisions what to do after the prefetch, is set to half the fabric size, e.g. for a fabric with 10 RACs, primary task selection start 5 reconfigurations before the next prefetch. This provides time to reconfigure enough accelerators to raise task efficiency in case a low-efficiency task has to be selected to run after pt.

Select Primary Task is used to select a prospective next primary task pt' from taskset $T \setminus pt$ (i.e. all remaining MKTs, SKTs and ZKTs except the current primary task). If the efficiency of pt' is sufficiently high (greater than the threshold TH_{E1}), pt' is chosen to replace pt as the new primary. This is done by using pt' as the only input candidate to the *Select Secondary Task* function, which guarantees that pt' is selected to run after pt, and that RACs will be reallocated to pt' to further improve its efficiency. TH_{E1} needs to be fairly high, as primary tasks tend to require a lot of accelerators, often equal to the number of RACs available on the fabric. Switching to a new primary task with low efficiency would require multiple reconfigurations until the new task reaches full efficiency. During this time RiCL would be high, thereby increasing the makespan. If the efficiency of pt' is lower than the threshold TH_{E1}, pt is kept as the primary task, and a secondary task is selected out of $T \setminus pt$ to bridge the time between the start of the prefetch of pt (when its efficiency drops) and when the efficiency of pt has increased sufficiently. The secondary task effectively hides the reconfiguration latency of pt.

Secondary task selection is shown in Figure 3.7. The goal is to select the next task nt (along with the number of accelerators a that shall be reallocated to it) that provides the highest total performance for the primary task pt and the secondary task nt. Taking performance for both tasks into account is crucial, as reconfigurations for nt start while pt still executes, thus pt will run with reduced efficiency for a short time until the system switches to nt. The performance of all kernels of all tasks for different fabric sizes is known from offline profiling and is used in Figure 3.7 to search an appropriate secondary task. The task that maximizes performance is returned as the secondary task nt, along with the number of accelerators a that are required for providing that performance. If reallocation would cause the performance of either the secondary or the primary task to drop below the threshold TH_P, reallocation is not performed, as it is not deemed profitable. In this case, a secondary task may still be used later, but without reallocation.

If the parameter TH_P is chosen too high, it would discard most secondary task candidates, e.g. if set to 1.0, no efficiency loss for the primary task would be tolerated, thus only very few (if any) RACs would be considered for reallocation to the secondary task. The secondary task would have to reach maximum performance with none or very few RACs. Meeting such tough constraints is almost impossible, thus the system would not perform any reallocation at all. Setting TH_P too low would allow reallocation of too many RACs to the secondary task. The primary task usually has more time-consuming kernels than the secondary task, thus reduced performance of

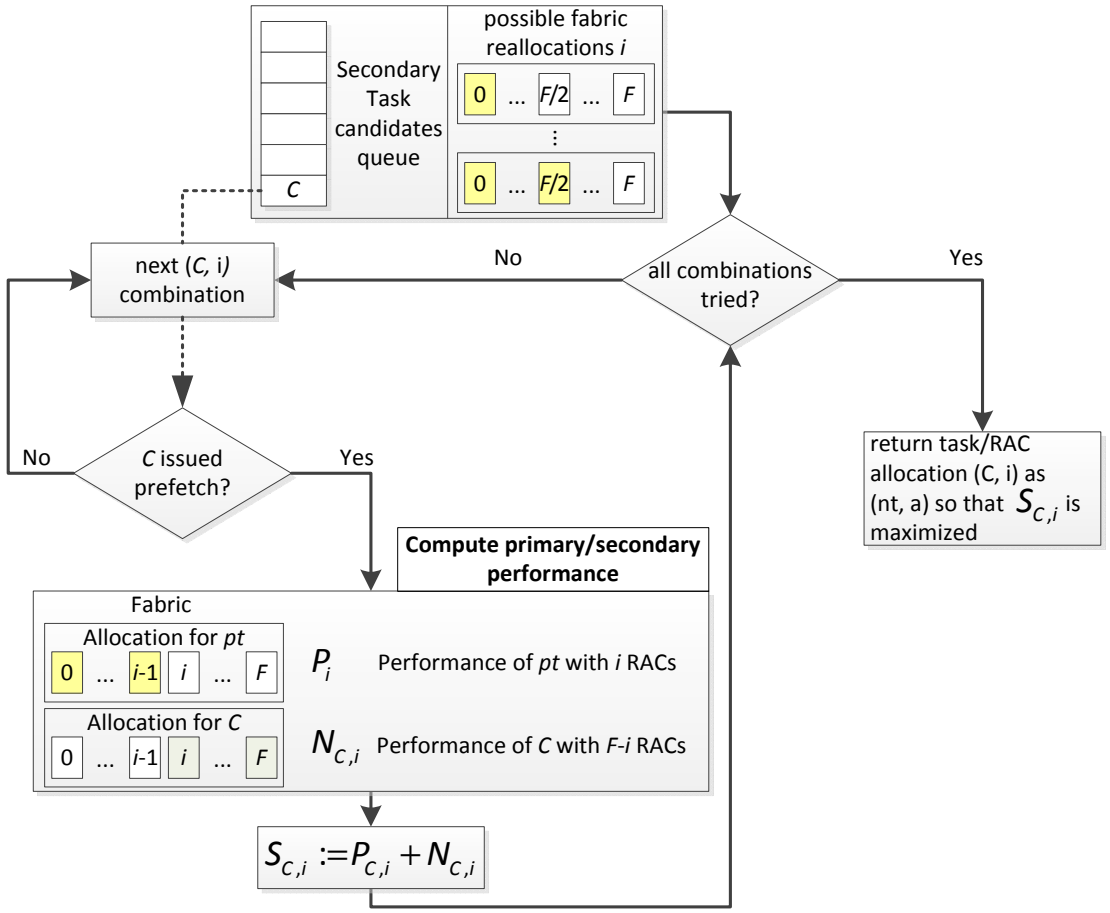


Figure 3.7: *Select Secondary Task* algorithm to determine a secondary task and amount of fabric for it to bridge the prefetch of the primary task.

the primary task has a strong negative impact on the makespan, more than can be made up for by improved secondary task performance.

After selecting the secondary task and determining the amount of RACs for reallocation, the system resumes executing *pt*. If *Select Secondary Task* decided to reallocate, then *a* reconfigurations before its next prefetch MORP is triggered again and starts reconfigurations for *nt*. While these reconfigurations are performed, *pt* continues running until its next prefetch (although with slightly reduced task efficiency, as with fewer RACs it will have to use slower SI variants).

The events before and after the prefetch of *pt* are illustrated in Figure 3.8. Task efficiency of the currently running task is shown with a solid line, while a dashed line is used for the non-running task. Ideally, reconfiguration for the secondary task (see ① in Figure 3.8) is finished immediately before the primary task issues its next prefetch. Due to variations in the time between prefetches for the primary task (e.g. due to different input-data affecting kernel duration), its prefetch may be issued earlier or later than expected (differences between expected prefetch time from offline profiling and actual prefetch time are evaluated in the results) and thus reconfigurations of the secondary task may be finished earlier or later than

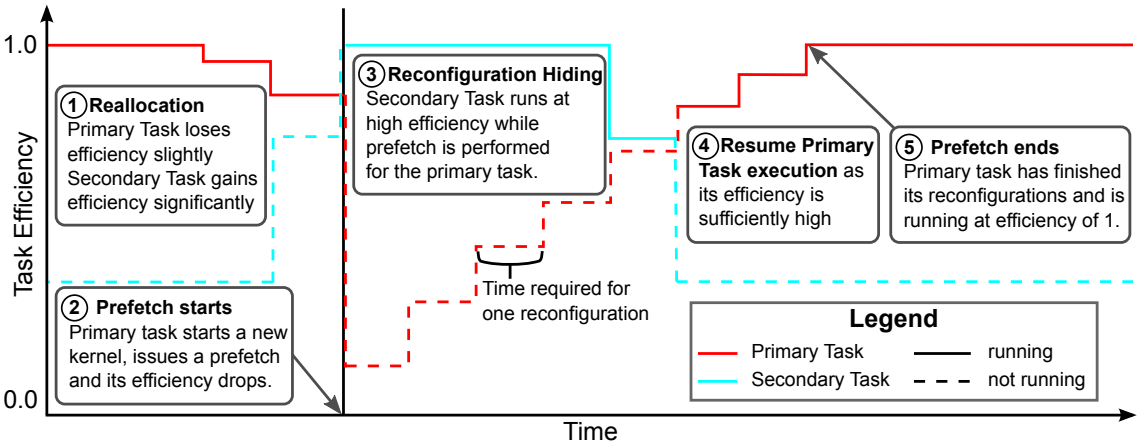


Figure 3.8: Efficiency of primary and secondary tasks in the proximity of a primary task prefetch.

the prefetch of the primary task. If the reconfigurations are finished earlier, RiCL for the primary task *before* the prefetch will be increased more than expected by Figure 3.7 due to the longer time the primary task runs with less reconfigurable fabric. If reconfigurations for the secondary task are not yet finished at the time the primary task issues its prefetch, its pending reconfigurations are aborted (to start the new reconfigurations for the primary task), and the secondary task executes with reduced efficiency until switching back to the primary task.

As generally a task will only have accelerators on the fabric for its current kernel, immediately after the primary task has issued its prefetch, its efficiency will drop (② in Figure 3.8). MORP now exits the *Primary Task Execution* state (see Figure 3.6). Before *Secondary Task Execution* can start, the system handles the following special cases: If the primary task issued its specially annotated *last prefetch* or terminates, then a new primary task needs to be selected and its pending reconfigurations need to be started before proceeding to the *Secondary Task Execution* state. If the next task *nt* that was selected during the *Primary Task Execution* state was a primary task, then the system directly switches to the new primary task, returns to the *Primary Task Execution* state, and finishes any outstanding reconfigurations for the new primary task while it is running, if required. If the selected next task is not a primary task, reconfigurations for the primary task are started, and the system enters the *Secondary Task Execution* state.

Secondary Task Execution In the *Secondary Task Execution* state, a secondary task is executed at high efficiency, while the primary task performs its reconfigurations and thereby increases its own efficiency (reconfiguration hiding, ③ in Figure 3.8). Which task to execute next (and thus whether to finish secondary task execution, or not) is decided by the *Select Next Task* function as follows:

1. If *pt* has an efficiency higher than a threshold ($E_{pt} > TH_{E2}$), then secondary task execution is finished and primary task execution is resumed.

2. If there is an MKT or SKT that has not issued its first prefetch yet (i.e. is still in its initialization phase), then this task is used as secondary task. Until such a task reaches its first prefetch, it has an efficiency of 1.
3. If a secondary task *nt* has been previously selected by the *Select Secondary Task* function and its efficiency is sufficiently high ($E_{nt} > TH_{E3}$), then use *nt* as secondary task.
4. Use a ZKT, if available. ZKTs have an efficiency of 1, and should therefore not be "squandered" if a high-efficiency MKT/SKT can be used instead. ZKTs are therefore only used if the previously selected secondary task *nt* is below the efficiency threshold TH_{E3}.
5. If no ZKTs are left, then *nt* is used even if its efficiency is below threshold TH_{E3}.

The system proceeds with the *Secondary Task Execution*, hiding the reconfiguration latency of the primary task (③ in Figure 3.8), until the primary task has sufficiently high efficiency. At this point the system returns all reallocated RACS (if any) from the secondary to the primary task and loads any remaining accelerators there. While finishing the prefetch, the system switches to *Primary Task Execution* ④. *pt* executes with increasing efficiency until the prefetch is finished ⑤.

If parameter TH_{E2} is too low, RiCL reduction after a prefetch will be minor, if it is very large (near 1.0), a secondary task will be wasted reducing minuscule amounts of RiCL and will likely no longer be available during later prefetches where they could be used more effectively to reduce larger amounts of RiCL. TH_{E3} values near 1.0 would cause ZKTs to be used even though fabric has been reallocated to a SKT. Low TH_{E3} would allow secondary tasks with a low efficiency (e.g. due to misprediction of the prefetch, reconfigurations for the secondary could not be started) to be used, and RiCL reduction would not occur.

3.5.1 Case Study

In this section a case study will be used in order to demonstrate MORP scheduling and reallocation decisions. A more thorough evaluation of MORP is provided in Section 6.3. As with PATS, the results in this section were obtained using a cycle-accurate simulator (see Section 6.2 for details) of the *i*-Core reconfigurable processor with a reconfigurable fabric consisting of 10 RACs. The workload used for this case study consists of H.264 video encoder and SUSAN (both MKT applications), AdPCM Decoding and Rijndael encryption (SKT applications), and SHA hashing (ZKT, as this version intentionally does not use SIs).

Figure 3.9 shows the values of RiCL over time for the taskset scheduled by FCFS (First Come First Serve) and MORP. The small stripes on top of the figures marked "Reconfiguration Trace" show the reconfigurations performed by the system and the large stripes marked "Scheduling Trace" show the scheduled task. The colors indicate which task reconfigures or is currently running, respectively. Black dots are plotted whenever RiCL is greater than 0.

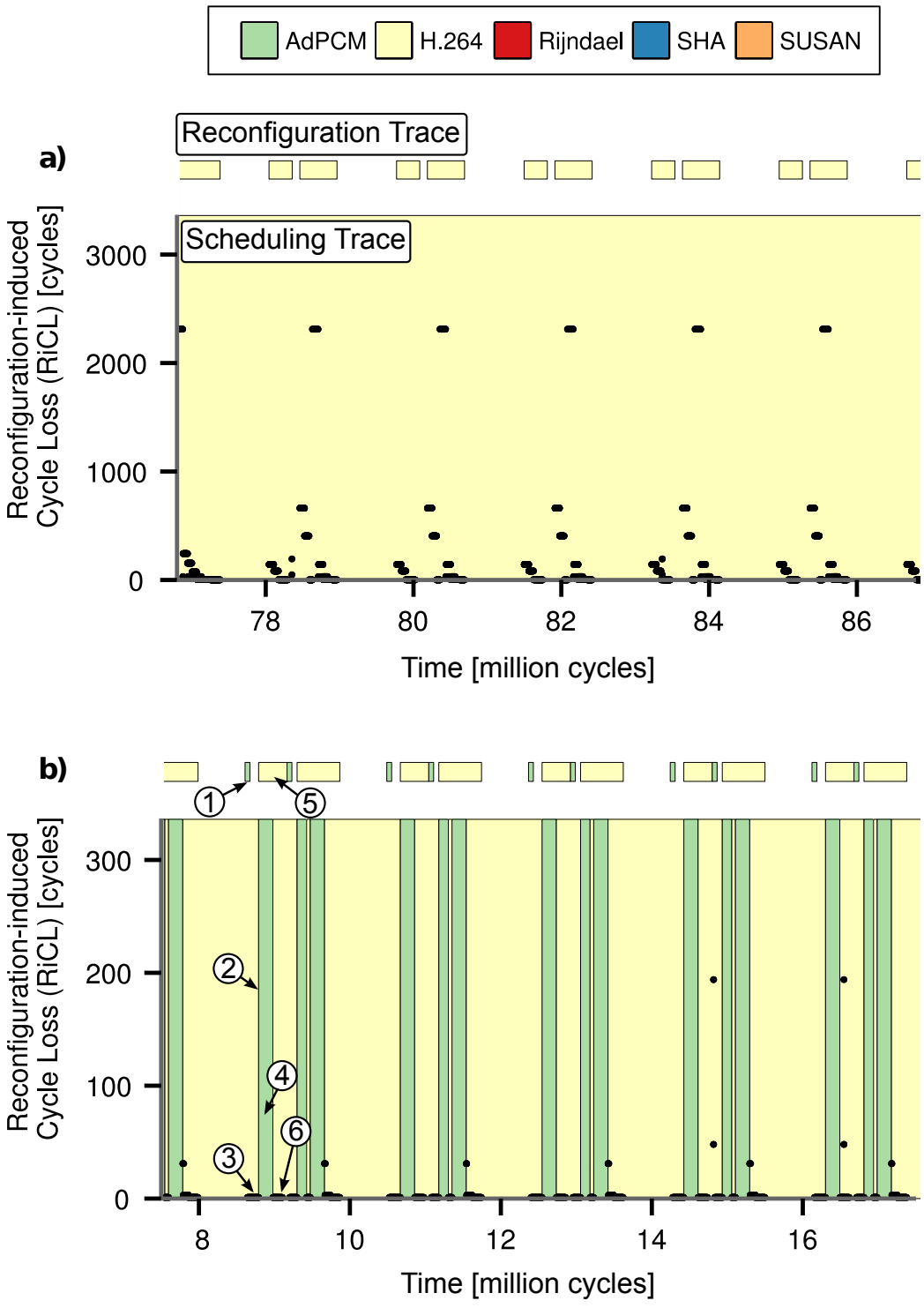


Figure 3.9: Scheduling and reconfiguration traces and RiCL of Taskset 2 using **a)** FCFS, and **b)** MORP scheduling. The range of the y-scale in a) is different than in b).

Figure 3.9a) shows an excerpt of 10 million cycles from the H.264 task scheduled with FCFS. As motivated in Figure 3.2, traditional schedulers (such as FCFS) disregard RiCL, resulting in a larger makespan. The repetitive sequence of the H.264 Encoder (encoding each frame and executing 3 kernels per frame) can be seen in the periodic pattern of the RiCL values. RiCL rises after each prefetch (start of small stripes at the top), and takes some time to reach 0. During this time the task runs at lower efficiency contributing to a higher overall makespan.

The same taskset scheduled by MORP is shown in Figure 3.9b). As both MORP and FCFS schedule the tasks at different points in time, showing an excerpt in the same timerange as in fig. 3.9a) would have shown the trace of a different task, thus a range was used where the H.264 task is scheduled by MORP, to make the decisions of both schedulers comparable. The y-scale (which shows the RiCL) ranges only to 300, unlike in Figure 3.9a), where it ranges to 3000.

Shortly before each prefetch, the AdPCM task is selected as secondary task, and 2 RACs are reallocated to AdPCM to improve its task efficiency. The reconfiguration for AdPCM ① is finished a short time before H.264 issues its prefetch ②, thus RiCL before the prefetch is marginally increased ③, as H.264 runs with 8 RACs instead of 10 for a short time. After the prefetch of H.264, MORP switches to the AdPCM task (which was selected as secondary task) ④, and executes it while the accelerators for H.264 are being loaded ⑤. Once the efficiency of H.264 is sufficiently high, MORP switches back from AdPCM to H.264. The RiCL after the prefetch is significantly lower ⑥ than when scheduled with FCFS.

In the direct comparison in Figure 3.9, MORP reduced the RiCL value down to only 6% of the FCFS RiCL value. RiCL in Figure 3.9b) at ③ is due to the difference between the Average Time Between Prefetches (ATBP) from offline profiling (see Section 3.3) and the actual time between prefetches during the execution of the taskset. It should be noted that ATBP is only an estimation gathered from offline profiling, and MORP does not require it to be perfectly precise.

3.6 Summary

While providing significant advantages in terms of performance and flexibility, fine-grained reconfigurable fabrics have the drawback of requiring considerable time to load accelerators. This negatively impacts performance of applications, as while they load their accelerators, they cannot fully utilize the fabric, resulting in a low execution speed. The impact is more significant for complex applications with multiple kernels, as they require frequent reconfigurations. Such applications particularly benefit from application-specific acceleration, but the performance degradation caused by frequent reconfigurations diminishes their speedup on reconfigurable processors.

Formally, this performance degradation can be described as varying task efficiency, a metric indicating how well a task currently profits from reconfigurable fabric. Task efficiency is maximized once all required accelerators have finished loading onto the

fabric but which is low after starting a new kernel, when the SIs have to be executed on the core pipeline due to of the required accelerators available on the fabric.

Two schedulers for different scenarios have been designed around the concept of task efficiency. The soft-deadline scheduler PATS is intended for workloads consisting of periodic (e.g. video/audio applications) and aperiodic tasks (e.g. system maintenance tasks). It has the goals of reducing tardiness (accumulated time of deadline violations of a taskset) and reducing completion time for non-critical tasks. When performing scheduling decisions, PATS uses both the deadline and the current task efficiency to compare schedulable candidates. This allows delaying low-efficiency tasks with a far enough deadline until their efficiency is high and they can be executed faster, resulting in better system performance and the ability to meet tighter deadlines, as shown in a case-study.

For makespan optimization of workloads without deadlines, MORP manages both task scheduling and fabric allocation. Similar to PATS, execution of different tasks is interleaved in such a way that a task B executes while the efficiency of a different task A is increased by successive reconfigurations. To ensure that task B has a high efficiency, MORP can provide it with a small share of the fabric, taken from task A, aiming to ensure that overall performance of both tasks is high. While case studies show a detailed view of the behavior of both schedulers, complete results are presented in Section 6.3.

In summary, with the introduction of task efficiency, multi-tasking for flexible reconfigurable processors (i.e. those allowing an area vs. performance trade-off through the use of multiple SI variants) can now be efficiently supported. This allows drawbacks of reconfigurable processors to be minimized by employing scheduling techniques that exploit the varying task efficiency of applications to hide reconfigurations and keep overall system performance high.

4 Sharing the Reconfigurable Fabric in Multi-Core Systems

When choosing the cores for an SoC, a system designer will have to weigh the flexibility provided by a reconfigurable processor against its disadvantages compared to non-reconfigurable accelerators (area overhead, performance advantage of non-reconfigurable accelerators). In order to outweigh these disadvantages, the fabric of the reconfigurable processor should serve multiple applications at once. Chapter 3 has shown how multi-tasking on a reconfigurable processor can be used to allow several applications running on the same core to benefit from the fabric. However, while the fabric was used to accelerate more than one application, at any given point in time only one application ran on the reconfigurable processor and used the fabric. Unlike in a single-core system, in a multi-core system applications can be distributed among all cores for better performance, but unless all cores are reconfigurable, acceleration on the fabric will be available only to a few of these applications (those running on the reconfigurable cores that have access to the fabric). In this chapter, an approach will be presented that allows multiple cores to use a single reconfigurable fabric simultaneously. This extends the benefits of the reconfigurable fabric to other cores in a multi-core system, while keeping the overhead for fabric area low.

Section 4.1 motivates the problem of inefficient use of the reconfigurable fabric. The two closest state-of-the art approaches for using reconfigurable fabrics in multi-cores are discussed in Section 4.2. Section 4.3 and Section 4.4 formalize the problem (fabric under-utilization) and proposed solution (SI merging), respectively. Section 4.5 discusses the details of the suggested approach, COREFAB, along with its hardware components. Section 4.6 presents an in-depth view of the presented approach, with further results in Chapter 6.

4.1 Motivation

The speedup that can be attained by executing an SI on the fabric instead of executing the equivalent ISA instructions on the GPP pipeline depends on the inherent parallelism of the respective kernel and the amount of fabric used for its implementation. For example, [LSC09] compares the area and performance for multiple kernel implementations. Exploiting its full parallelism and thereby attaining its highest performance, the implementation of a JPEG encoding kernel requires 5×

as much area as a kernel performing SHA cryptographic hashing. Assuming a scenario where a single-core reconfigurable processor runs a JPEG and SHA application, and

1. the hardware implementations of the JPEG and SHA kernels require 100% and 20% of the fabric area, respectively,
2. the reconfigurable core executes SHA and JPEG in alternation for the same amount of time, and
3. the fabric is reconfigured when the application switches between JPEG and SHA,

the reconfigurable core will use 60% of the fabric area on average, leaving 40% unused. If the JPEG kernel uses less than 100% of the fabric area (which is a more likely scenario, as there usually are more demanding kernels in the media domain that the fabric should be able to accelerate), then an even larger percentage of the fabric will be unused.

This effect is *fabric under-utilization*, i.e. during the time that the fabric is used, at least some fabric resources are not fully utilized. A system designer integrating a fabric into an SoC expects that the benefits in terms of performance and reduced overall power consumption outweigh the investment in terms of silicon area and (leakage) power. Thus, fabric utilization should be kept as high as possible, as these benefits only manifest themselves when the fabric is actually used, while the drawbacks are always present. In general, fabric under-utilization is an indication of inefficient use of the fabric.

Methods to reduce fabric under-utilization are: (i) reducing fabric size, (ii) increasing SI size, (iii) concurrent execution of multiple SIs.

reducing fabric size In a small fabric even small kernels would use a large part of the fabric area, reducing under-utilization. As most SIs are available in multiple SI variants with varying fabric area requirements, most SIs will have a variant that will fit on a smaller fabric. There are multiple disadvantages with this approach: the performance reduction due to using a smaller SI may not be acceptable to the designer. Applications that switch rapidly between multiple SIs will either have to reconfigure new accelerators after each SI, or only use one SI on the fabric, while running the others in software. Only the latter alternative is feasible, as considering that a single reconfiguration takes tens of thousands of cycles, while software emulation of SI code is usually less than that. Similarly, in a multi-tasking scenario there would only be sufficient fabric area to accommodate one application.

increasing SI size The part of the run-time system that is responsible for selecting SI variants to reconfigure to the fabric will aim to maximize SI performance, and thus generally select the largest SI variants. SI variant size can be increased further only by implementing a larger part of an application as an SI. For some kernels this may be possible (e.g. by implementing an unrolled version of a loop body as an SI), but for most kernels the largest possible part of a kernel is already used.

concurrent execution of multiple SIs by allowing multiple SIs to run on the fabric simultaneously, a larger part of the fabric is used, reducing under-utilization. This approach is not constrained by limited instruction-level parallelism in a kernel (as with the point above), and still allows very demanding kernels to fit onto the fabric (unlike with the reduced fabric size approach). The remainder of this chapter will focus on concurrent SI execution.

Executing multiple SIs from the same application at the same time would only be feasibly if these SIs are placed directly after each other in the instruction stream (i.e. no other instructions in between) and this still would incur significant synchronization overhead as both SIs might access same data memory (e.g. Read-After-Write conflict between the second and the first SI, both of which access main memory). Such limitations and overheads do not exist when the two SIs come from different applications executing on different cores, thus the remainder of the chapter will focus on this scenario.

4.2 Related Approaches

A broad overview of existing work on multi-core and reconfigurable architectures is provided in Section 2.4. This section will discuss the two most relevant approaches pertaining to using a fabric in a multi-core. These two approaches will also serve as comparison partners during evaluation.

Reconfigurable fabric in a multi-core can either be provided to each core as *dedicated fabric* (Figure 4.1) or a *shared fabric* (Figure 4.2) can be used that is accessible by all cores.

In the dedicated fabric approach, for a given area constraint A on total fabric size each core is assigned a fraction of the fabric (e.g. $A/4$ in a 4-core multi-core and equal-sized fractions). Other resources, such as memory-bandwidth between the fabric and an on-chip SRAM or the memory hierarchy are also reduced accordingly. If the SIs deployed by the different cores have highly varying fabric resource demands (e.g. demanding video encoder SIs on core 1 and smaller CRC SIs on core 2), then the under-utilization problem will still exist, and performance demands for the applications with the more demanding SIs may not be met. Additionally, fabric support hardware that is mandatory for SI execution, such as the SI μProgram memory and SI Execution Controller will have to be replicated for each fabric as well. Other support hardware such as the Accelerator Loader (see Section 2.6.2) can be shared between all fabric instances, unless parallel reconfiguration is required. The advantage is that all dedicated fabrics can be used independently from each other (i.e. concurrent execution of SIs), thus there will be no slowdown of SIs due to concurrent access to shared fabric resources.

Dedicated fabrics are used in [WA10], where reconfigurable fabrics are loosely coupled to multi-core clusters in many-core systems. The fabric is partitioned into equal shares and each core uses a private fabric share (another option is to use fabric

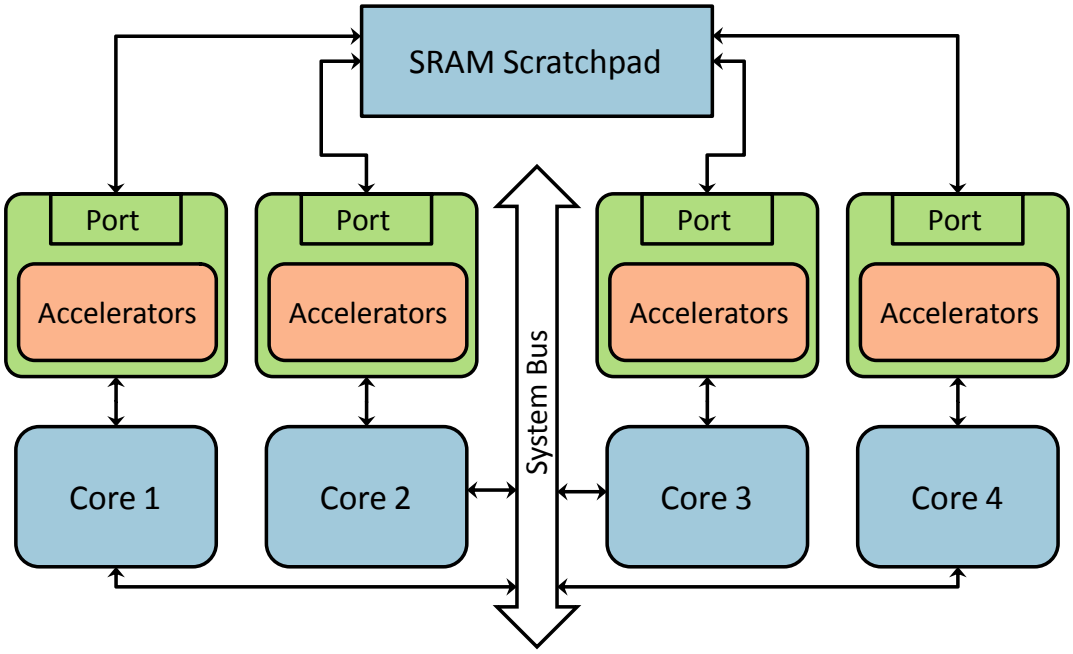


Figure 4.1: Multi-core architecture with *dedicated fabrics*.

in a time-multiplexed fashion, comparable to the shared fabric approach described below).

In the shared fabric approach, all cores can access the whole fabric, providing the flexibility to adapt to SIs with varying demands, and thereby benefit different workloads (i.e. applications with similar/different fabric requirements). The disadvantage is that at any given cycle, only one SI can run on the fabric, thus SIs running on the fabric concurrently need to be serialized. While fabric under-utilization will be lower than for a single-core reconfigurable system (as the additional SIs from multiple cores will increase the overall utilization of the fabric), overall under-utilization will still be at most the average of the under-utilization of all SIs.

A shared reconfigurable fabric is used in [CM11], where a dual-core system is connected to a reconfigurable fabric. Access to the fabric is exclusive to a single core at any point in time. If multiple cores wish to access the fabric at the same time, a round-robin arbiter determines the core which is granted access in a given cycle.

Apart from the two listed examples for the two classes of fabric sharing in multi-cores, Section 2.4 discusses more reconfigurable multi-core architectures.

The approach proposed in this thesis (COREFAB – COncurrent REconfigurable FAbric utilization) aims to combine the advantages of dedicated and shared fabrics: concurrent execution of SIs and flexibility regarding varying SI demands. The goal is to improve performance of the whole reconfigurable multi-core by reducing fabric under-utilization. The base system is a multi-core where only one core ("reconfigurable core") has access to the fabric. COREFAB provides the protocol

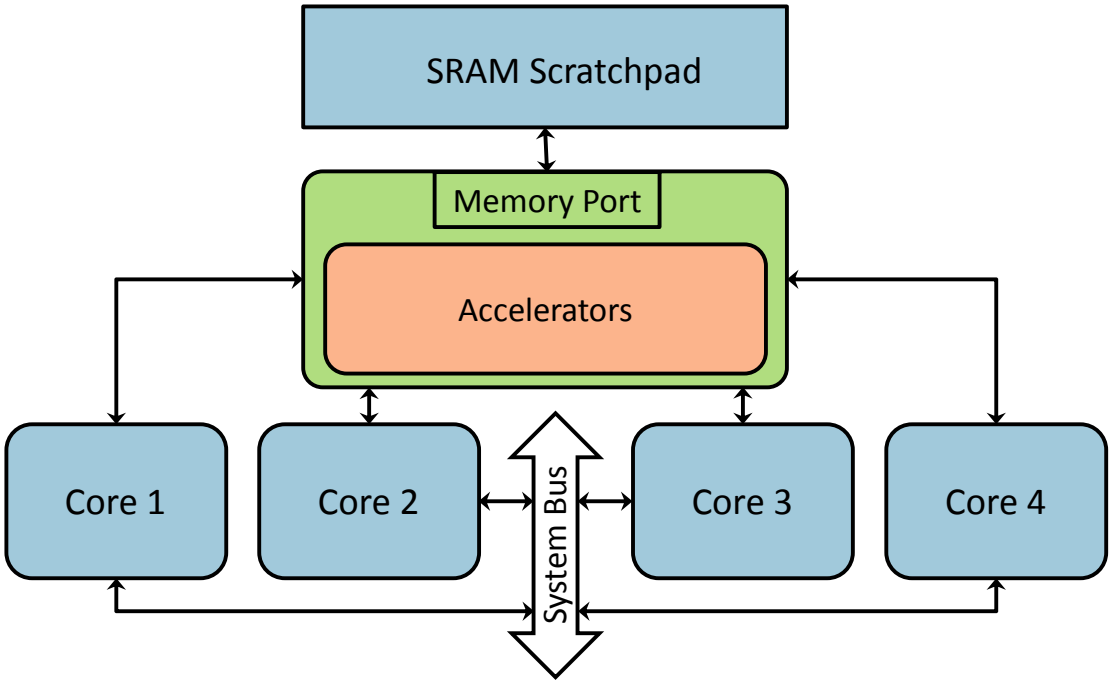


Figure 4.2: Multi-core architecture with one *shared fabric* between all cores.

and hardware components to allow other cores to access the fabric (granting the flexibility of fabric sharing) and the concept and hardware for SI merging, which allows concurrent execution of SIs (as with dedicated fabrics). Multi-core performance can therefore be increased by improving performance of the non-reconfigurable cores ("GPP cores"), while keeping performance of the reconfigurable core as high as possible.

As with multi-tasking (Chapter 3), the COREFAB will not focus on distributing the fabric between different cores. Approaches to do so have been explored in other existing work (see Section 2.3.1), and can be used in conjunction with COREFAB.

4.3 Fabric Under-Utilization

Before addressing fabric under-utilization, this section will formalize the problem.

As described in Section 2.6, the fabric consists of the following resources: reconfigurable accelerator containers (RACs), RAC links, private register files (pRF) and memory ports. Memory ports would seem to be similar either to RAC links (communication) or pRFs (storage), but they behaves more closely to accelerators. The memory ports have the same interfaces as RACs and are connected to the fabric-internal links and they have their own pRFs where they either store read data or where they buffer data to be written. The kernels for which SIs are generated are either memory-bound or computationally bound. Fabric resources related to these

two characteristic will improve kernel performance (e.g. by increasing the amount of fabric area for computation or increasing the bandwidth for fetching input data from memory), thus RACs and memory ports are called *principal resources*. The kernels are not bound by the transfer of intermediate results, which is the purpose of RAC links and pRFs. These latter resources are simply a necessity to support the multi-cycle implementation of SIs, and thus are called *auxiliary resources*.

If the principal resources are not fully utilized, running additional SIs on the fabric will improve their utilization. However, if they are fully utilized, then running additional SIs is not possible (as there are no RACs left for computation and no memory bandwidth to fetch input data/write back results), even if auxiliary resources are under-utilized. Therefore, under-utilization occurs if at any point in time when not all principal resources (RACs or memory ports) are used. Thus, there are two possible reasons for under-utilization: RAC under-utilization and memory port under-utilization. For the following definitions it is assumed that an SI is being executed on the fabric. If no SI is running, the fabric resources are not used at all, and fabric under-utilization is at its maximum.

Synthesizing different kernels will generally result in different area requirements (cmp. [LSC09]), i.e. different amounts of RACs. *RAC under-utilization* occurs if an SI does not use all RACs available on the fabric because the fabric was optimized for larger SIs or because –even though the kernel could use all RACs– it is not beneficial to do so. Most SIs provide diminishing performance improvement as the area available for their implementation increases. In a fine-grained reconfigurable fabric the time to load an SI onto the fabric depends on the area used by its implementation. Thus, even though a small performance improvement may be provided by a larger SI variant, it may be negated by the increased reconfiguration time required to load this implementation onto the fabric.

Formally, consider an SI μProgram consisting of K μOps. Let N be the set of RACs on the fabric and let $a_{k,i}$ equal '1' if RAC $i \in N$ is used in μOp $k \in [1, K]$. RAC under-utilization exists if the constraint in Equation (4.1) is satisfied, i.e. there is at least one RAC that is not used during the entire SI execution.

$$\exists i \in N : \sum_{k=1}^{K} a_{k,i} = 0 \tag{4.1}$$

Memory port under-utilization occurs if at least one memory port is idle at any point during execution of an SI. Table 4.1 shows the memory port utilization for several SIs running on the i-Core fabric (two 128-bit wide memory ports). Many SIs do not consistently exploit the available memory bandwidth to full capacity. [ABCG+06] analyzes memory accesses of some representative algorithmic patterns, concluding that only about half of them are memory bound. SIs implementing non-memory bound kernels cannot benefit from the full memory bandwidth available in the fabric. Another reason is that SIs often follow the following pattern: (i) perform a burst read from external memory to get input data, (ii) process input data using one or

SI	Utilization	Description
fmm	60%	Multiplication of 32×32 floating point matrices
sad	47%	Sum of Absolute Differences on a 16×16 block
sha	43%	SHA-1 hashing
dct	33%	Discrete Cosine Transform on a 4×4 block
adpcm	25%	Encoding of audio data to ADPCM format
aesenc	10%	AES Encryption

Table 4.1: Examples of Memory Port utilization in Special Instructions

multiple accelerators (there can be multiple processing stages, resulting in dozens of cycles for this step), (iii) write back result data. During (ii) no memory traffic is performed, leading to under-utilization of the memory port. Such read-process-write patterns can also occur multiple times during a single SI execution (e.g. in the fmm SI).

Formally, memory port underutilization can be defined as follows: let M be the set of memory ports available on the fabric and $m_{k,j}$ equal '1' if memory port $j \in M$ is used in μOp k. Then memory port under-utilization exists, if the constraint in Equation (4.2) is satisfied, i.e. during execution of an SI, there is at least one μOp where not all memory ports are used.

$$\exists k \in [1, K] : \sum_{j \in M} m_{k,j} < \left|M\right| \tag{4.2}$$

An SI μProgram underutilizes the fabric if Equation (4.1) or Equation (4.2) is satisfied. Reconfigurable architectures and the kernels used on them can be tested for under-utilization using these equations. If no under-utilization exists or if it occurs only to a negligible degree, then the fabric already has high area efficiency. COREFAB is optimized for scenarios with middle to large under-utilization.

4.4 Merging of Fabric Accesses

To reduce fabric under-utilization, COREFAB merges SIs μOps of different cores on-the-fly, to allow their concurrent execution. In this section, the prerequisites for merging will be formalized.

Fabric resources only support exclusive access, i.e. in any cycle a particular resource instance may be used by at most one SI (although during the same cycle other resource instances can be used by other SIs). For merging of SIs, let the fabric resources R be the principal resources RPUs N and memory ports M and auxiliary

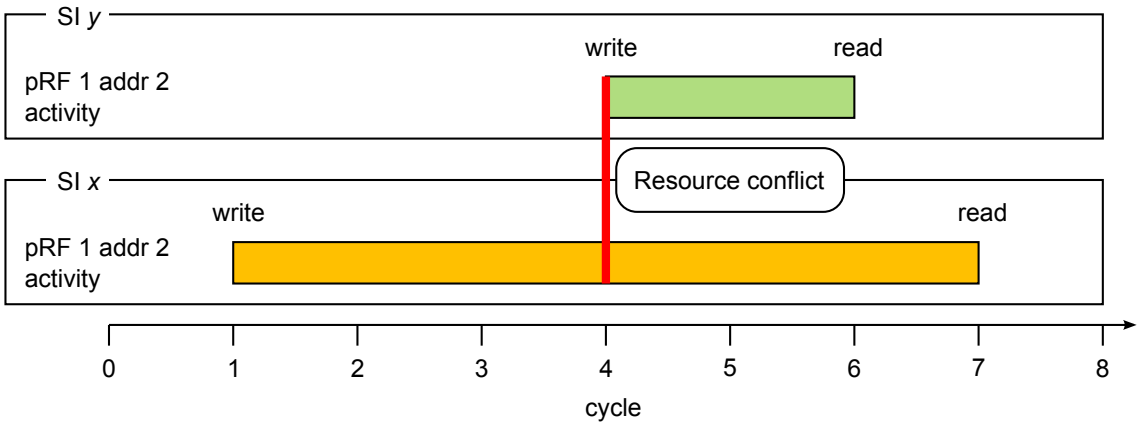


Figure 4.3: Resource conflict due to SIs x and y using the same private Register File (pRF) address.

resource RPU links L (pRFs are not included here and will be handled separately further down):

$$R = N \cup M \cup L \tag{4.3}$$

If multiple SIs shall run in parallel on the fabric, it needs to be ensured that there are no resource conflicts between the SIs, i.e. no two SIs x and y that are executed concurrently on the fabric may access the same resource in a given cycle.

Let the system attempt to execute μOp k_x of SI x in the same cycle as μOp k_y of SI y. Furthermore, let $R_{k_x}, R_{k_y} \subseteq R$ correspond to the set of resources required on the fabric by k_x and k_y, respectively. Then, the resource requirements of the μOps of both SIs need to be disjoint in the current cycle to allow their parallel execution on the fabric:

$$R_{k_x} \bigcap R_{k_y} = \emptyset \tag{4.4}$$

This allows the operations to be combined into one **merged** operation, which is the disjoint union of the requirements of both operations (Equation (4.5)).

$$R_{k_x} \bigsqcup R_{k_y} = \begin{cases} R_{k_x} \bigcup R_{k_y} & \text{if Equation (4.4) satisfied} \\ R_{k_x} & \text{otherwise} \end{cases} \tag{4.5}$$

Additionally, pRF conflicts between x and y may occur. pRF activities, which are reads and writes to pRF addresses, are part of the μOps where this activity occurs. However, conflicts may occur even during μOps where no such activity happens, as long as the intermediate value in a particular pRF address will still be read during a later μOp. This is illustrated in Figure 4.3. SI x writes an intermediate value at RAC 1, pRF 1, address 2 in cycle 1, and will read that value at cycle 7. A different SI y that runs simultaneously, writes to the same pRF 1 at RAC 1, address 2 in cycle 4, leading to a conflict, as SI y would overwrite the value of SI x. However, the cycle 4 the μOp of SI x has no activity for this pRF, as there are no reads or writes

there during this cycle. Therefore, the conflict would be detected by analyzing μOps on a cycle-by-cycle basis.

As conflict detection and merging is done at run-time in each cycle, a fast and therefore simple method is required that avoid pRF conflicts. SIs are annotated with *used address ranges* for each pRF in each RAC, which are the addresses used during execution of the whole SI. An SI x could have the used address ranges of 0–2 for RAC 1, pRF 1, while SI y could have range 3–4 for the same pRF and RAC. In this case there would be no pRF conflict between the two SIs. Formally, let the fabric have N RACs, with S pRFs per RAC and A be the number of addresses in each pRF. Then, given the function $u(z, n, s)$, which returns the set of used addresses for an SI z, a RAC n and a pRF s, let the used address range for SI x be defined as follows:

$$\begin{aligned} \{[l_{x,n,s}, h_{x,n,s}]\} \ \ \forall n \in [0..N], s \in [0..S], \text{where} \\ l_{x,n,s} := \min u(x, n, s) \\ h_{x,n,s} := \max u(x, n, s) \end{aligned} \tag{4.6}$$

No pRF conflict exists between SIs x and y, if

$$\forall s : \forall n : [l_{x,n,s}, h_{x,n,s}] \cap [l_{y,n,s}, h_{y,n,s}] = \emptyset \tag{4.7}$$

Used address ranges are available after μProgram generation and are checked before an additional SI is started on the fabric. This is described in the next section, along with the details of μOp merging.

4.5 Concurrent Fabric Utilization

The SI merging concept presented in the previous section reduces fabric under-utilization by allowing execution of multiple SIs on one fabric simultaneously. COREFAB is based around the merging concept, but also introduces the components and a protocol to connect the non-reconfigurable cores to the fabric to allow them to execute SIs. This section describes the resulting multi-core system, with a focus on the COREFAB contributions.

The resulting multi-core architecture is shown in Figure 4.4. The reconfigurable core (based on the i-Core, Section 2.6) is tightly coupled to the reconfigurable fabric and is called *Primary Core*. SIs executed by the primary core are called *Primary-SIs*. Other non-reconfigurable cores are called *Remote Cores*. SIs executed by remote cores are called *Remote-SIs*. The remote core pipelines need to be extended to recognize SIs as valid instructions (Instruction Decode Stage) and communicate with the fabric to start the SI and transfer operands/results between the pipeline and the fabric (Register Access, Execute stages and Memory Stages). These pipeline extensions are similar to those of the i-Core (which is based on the LEON3 processor) itself. COREFAB will prioritize execution of the primary core, as the goal is to improve

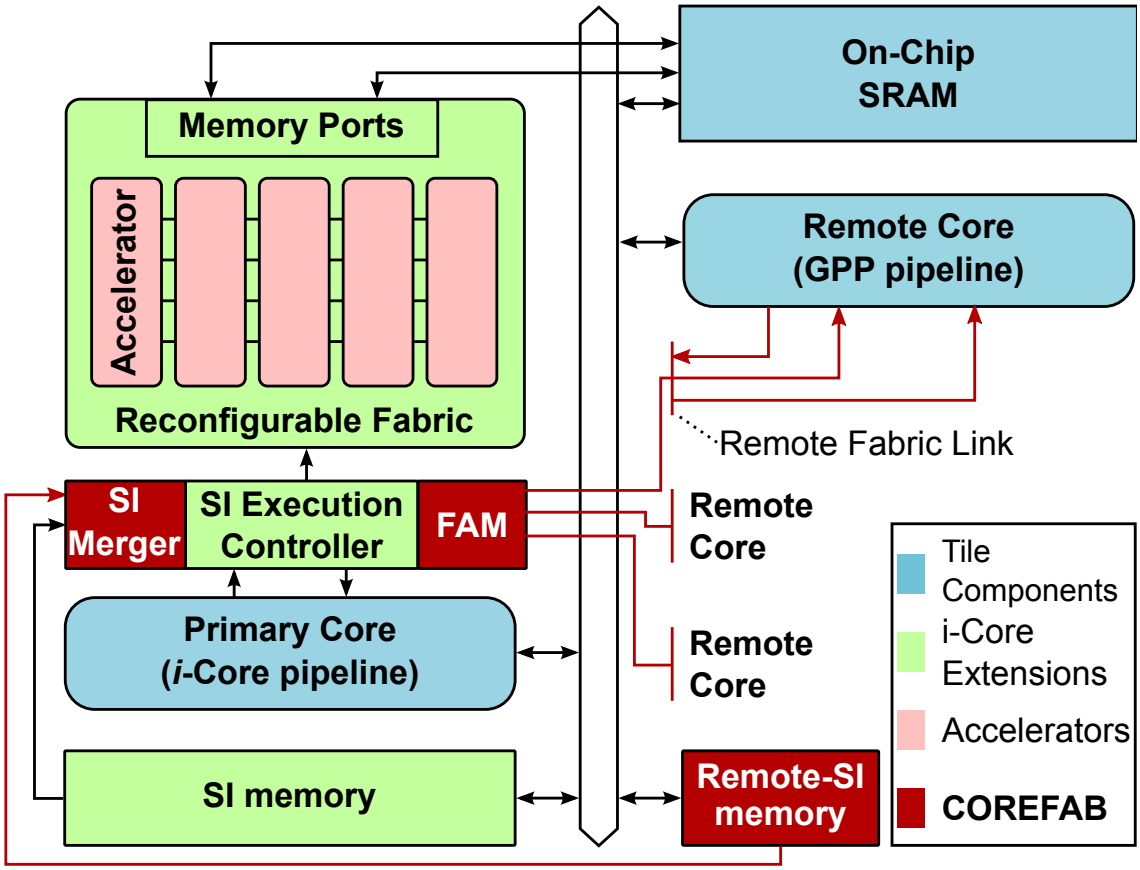


Figure 4.4: Multi-core architecture with shared reconfigurable fabric. COREFAB contributions are highlighted.

performance of remote cores while maintaining high performance on the primary core.

To allow execution of Remote-SIs, additional hardware is required for the reconfigurable multi-core system (red shaded modules in Figure 4.4), i.e. the Fabric Access Manager for providing non-reconfigurable cores access to the fabric, the SI merger for merging μOps of the Primary-SI and Remote-SI, and the Remote-SI memory.

SI μPrograms are stored in a dedicated on-chip memory, which is connected to the system bus and the SI Execution Controller. If the same memory would be used for Remote-SIs, an additional read port would need to be added to allow retrieval of μOps for both Remote-SIs and Primary-SIs in the same cycle. This is costly in terms of area, thus a smaller dedicated μProgram memory for Remote-SIs is used instead. The size of the memory is a system design parameter which depends on the amount of different Remote-SIs executed by the remote cores.

A remote core that wishes to offload a kernel to the fabric, will issue Remote-SI requests to the *Fabric Access Manager* (FAM), which will interact with the *SI Execution Controller* to allow *co-execution* of a Remote-SI and the currently running Primary-SI. Execution of a Remote-SI involves communication between the FAM

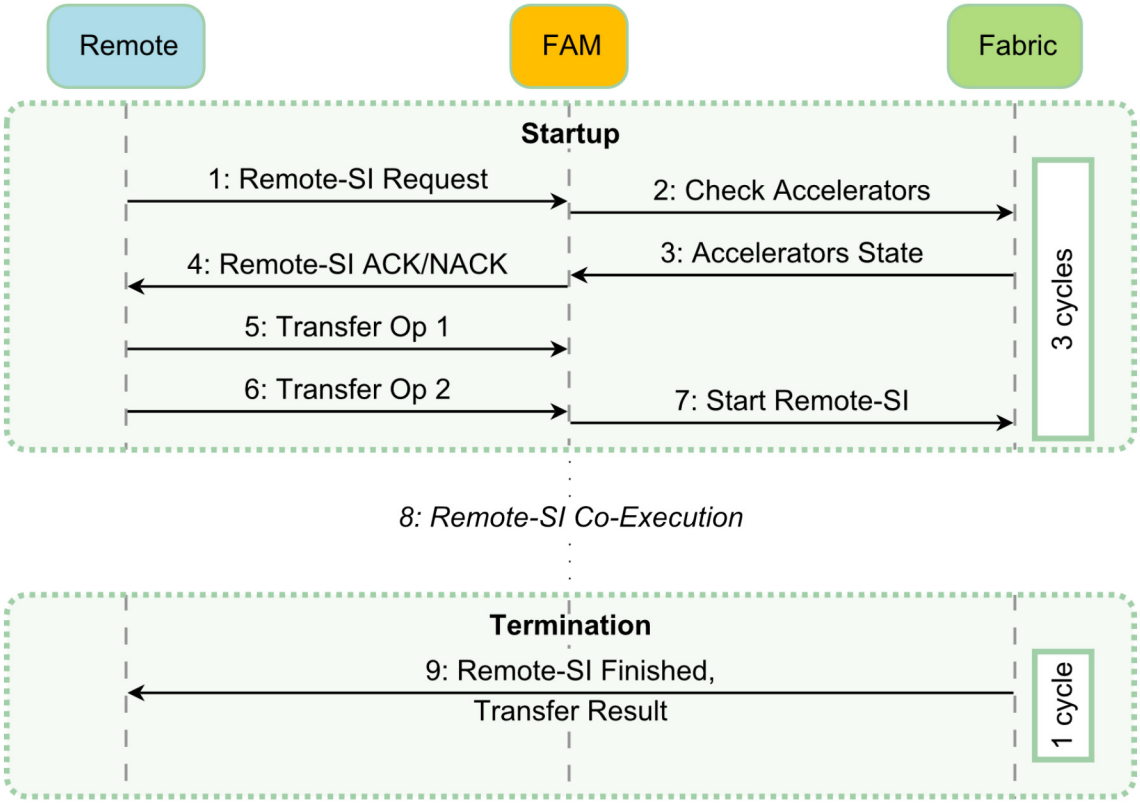


Figure 4.5: Message-Sequence chart for executing a Remote-SI. Communication between the remote core starting the Remote-SI, the Fabric Access Manager (FAM) and the Fabric is shown. The vertical axis is time.

and a remote core for the Remote-SI requests and operand/result transfer. The corresponding protocol is described in the Section 4.5.1.

The *SI merger* merges μOps from Primary-SI and Remote-SI on-the-fly. When no Remote-SI is executed, the SI merger is in *single* mode and forwards the μOps from the Primary-SI μProgram memory to the SI Execution controller. Upon start of Remote-SI execution, FAM sets the SI merger into *concurrency* mode and a Primary-SI μOp k_x and Remote-SI μOp k_y are fetched from the respective memories. Merger operation is describe in detail in Section 4.5.2.

The implementation presented here is limited to two concurrently running SIs, one Primary-SI and one Remote-SI. This is done in order to keep hardware overhead of the FAM and SI merger low. As the SI merger lies on the critical path, increasing merger complexity would reduce fabric frequency (or would require a more complex pipelined implementation), in addition to increasing area overhead.

4.5.1 Fabric Access Manager

Executing a Remote-SI involves communication between the remote core that triggers the Remote-SI, the FAM and the fabric. The message-sequence chart in Figure 4.5

illustrates the protocol used for this communication.

Execution of a Remote-SI proceeds as follows. A remote core encounters an SI in its instruction stream. The remote core sends a Remote-SI request to the FAM ①. The request encodes which accelerators are required at which location. The FAM checks if the SI prerequisites for running on the fabric are met ②, ③. The fabric is queried for the currently loaded accelerators, and they are compared to the accelerators required by the Remote-SI. If not all required accelerators are available, the SI is rejected, as loading a missing accelerator would take too long (the remote core can still start loading the accelerator, but it can execute multiple iterations of the SI-equivalent kernel on its pipeline). At the same time, the FAM checks for pRF conflicts between the running Primary-SI and the incoming Remote-SI. It does so by comparing the used address ranges of the Primary-SI and Remote-SI for each RAC and pRF of the fabric (done in parallel). If at least one address range overlaps, the Remote-SI is rejected.

If the Remote-SI cannot be executed, the remote core is notified ④ and executes the kernel containing the Remote-SI as regular software code. If the required accelerators are available, the remote core is notified ④ and transfers the operands (1 per cycle ⑤, ⑥) for the Remote-SI to the FAM. The FAM starts execution of the Remote-SI ⑦ by sending the operands to the fabric and instructing the SI execution controller to switch into *concurrency* mode. The fabric then proceeds with co-execution of the Primary-SI and Remote-SI ⑧ by merging them (detailed in next subsection).

When the Remote-SI is finished, the FAM notifies the remote core, sets the SI execution controller into *single* mode and transfers the SI result to the remote core ⑨. If the Primary-SI is finished while a Remote-SI is still running, the FAM proceeds as if a Primary-SI was still running, i.e. the SI execution controller stays in *concurrency mode*. The only difference is that as upon termination of the Primary-SI, the SI merger sets the Primary-SI program counter to `NONE`, thus only Remote-SI μOps are fetched until a new Primary-SI starts. This is to prevent another remote core starting a Remote-SI, which could stall the primary core until either of the Remote-SIs are finished.

Figure 4.6 shows the implementation of the FAM. Each remote core has a channel to the FAM with an input multiplexer selecting the remote core which may co-execute on the fabric. If a Remote-SI request arrives while a different remote core is co-executing (i.e. FAM FSM is not in IDLE state), then the Remote-SI request is rejected. In case of simultaneous Remote-SI requests, a fixed priority conflict resolution is performed, where the core ID is used as the priority. The Remote-SI protocol is implemented by the FAM FSM. Transition edges either are annotated with the edge labels from Figure 4.5 or are unlabeled, meaning that the state transition occurs in the following cycle, after the actions of the current state are finished. Communication with the SI merger is performed via the *mode* and *SI state* signals. 'Mode' sets the SI merger into *concurrency* mode when FAM enters state REQ_RCVD and into *single* mode when entering state IDLE. 'SI state' is used to signal that a Remote-SI has finished execution and to transfer result data from the fabric.

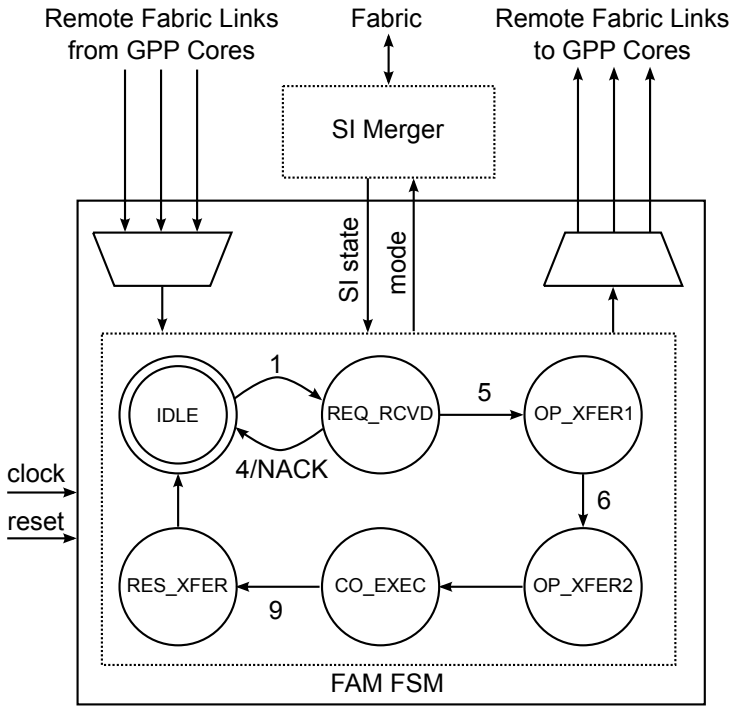


Figure 4.6: Fabric Access Manager

The *Startup* and *Termination* phases from Figure 4.5 would incur an additional 4 cycle latency to each Remote-SI execution if they were performed during the Execute stage of the remote core that issued the Remote-SI. Due to the 7-stage pipeline of the remote core implementation (LEON3), this latency can be reduced as follows. The Startup phase is started when the Remote-SI is detected in the instruction stream of the remote core, i.e. the Decode (DE) stage. The first operand is transferred to the FAM during the Register Access (RA) stage. Assuming that only one 32-bit operand can be transferred at a time to save interconnect area, the remote core stalls for 1 cycle while the second operand is transferred from the remote core to the FAM. Both operands could also be transferred in one cycle over a 64-bit wide channel. When the Remote-SI is at the Execute stage of the remote core, the FAM starts Co-Execution. The termination phase happens during the Memory Stage (ME) of the remote core, thereby no stalling is required. In total, the protocol for executing Remote-SIs is reduced to additional latency of 1 cycle.

4.5.2 SI Merging

Figure 4.7 shows the flow-chart for co-execution of a Primary-SI and a Remote-SI using SI merging.

The μOps for Primary-SI and Remote-SI are fetched from their respective memories (①). The SI merger then tests if the μOps are disjoint (as described in Section 4.4) by checking the bitslice of the μOp of the Primary-SI and Remote-SI that encodes the use of a particular fabric resource (such as a RAC, a memory port, a fabric link).

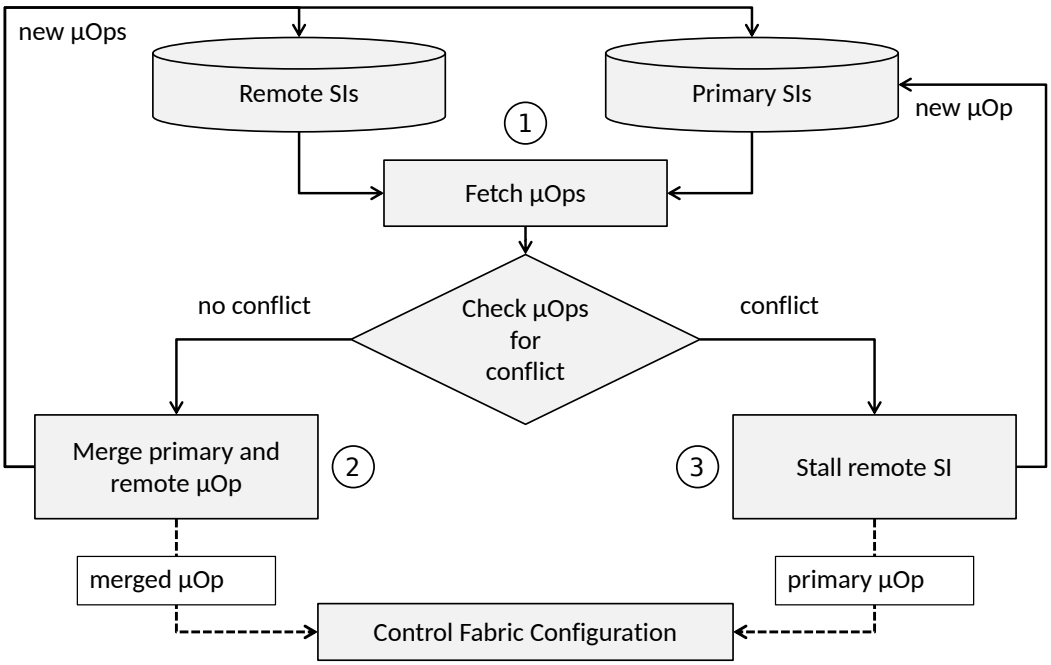


Figure 4.7: Merging of SIs.

If the bitslice is equal to the "unused" bitstring (all zeroes) for at least one of the SI μOps, no conflict occurs for this particular resource. These checks are done in parallel for all fabric resources. If a conflict is detected, the μOps are not disjoint, the Remote-SI is stalled for one cycle, and the merged μOp directly corresponds to Primary-SI μOp (②). If the operations are disjoint, the merged μOp is generated according to Equation (4.5) by OR-ing the μOps of both SIs (③). The merged μOp (whether there was a conflict or not) is then fed to the SI Execution Controller and configures the fabric resources.

If there is a conflict, the program counter for Primary-SI μProgram is incremented, while the Remote-SI μProgram program counter is held, and in the following cycle the merger will attempt to merge a new Primary-SI μOp with the Remote-SI μOp from the current cycle. If there is no conflict, both program counters are incremented, and the merger will attempt merging new μOps for both Primary-SI and Remote-SI in the next cycle.

4.5.3 Reducing conflicts during merging

Merging SIs is not always possible if there are fabric resource conflicts, i.e. if Equation (4.4) is not satisfied. Resource conflicts are handled by stalling the Remote-SI, thus large number of conflicts will reduce the performance improvement of CORE-FAB. As with multi-tasking, RACs are allocated at the start of a task that plans to use the reconfigurable fabric. Section 2.3.1 discusses the RAC allocation strategies

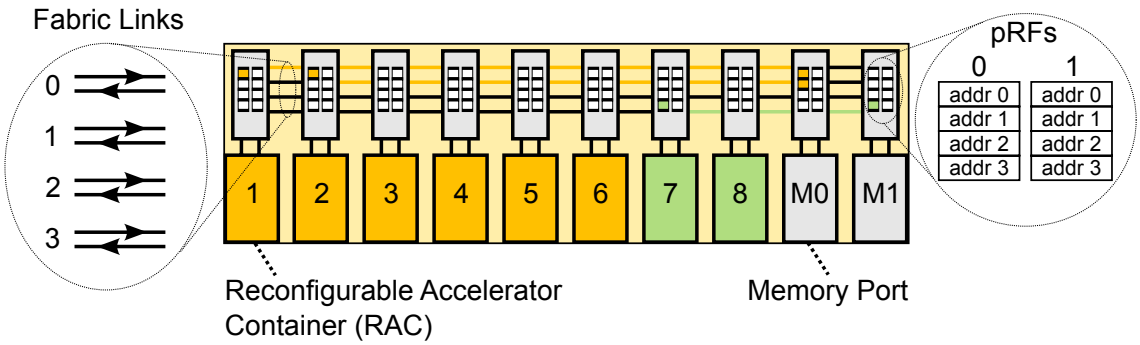


Figure 4.8: Example of fabric resources allocation for Primary-SI and Remote-SI μPrograms.

that can be used for that. As a task can use (just not reconfigure) RACs outside its own share, a RAC conflict could still happen in the rare case that: (i) a task does not have an accelerator reconfigured, but a different task does and (ii) a μProgram is available for exactly this fabric configuration. In this unlikely event the RAC conflict is simply resolved using stalling. To reduce the number of RAC link conflicts and memory port conflicts the SI μPrograms can be generated differently for Primary-SIs and Remote-SIs.

μPrograms determine which fabric resource is used by an SI. A way to reduce resource conflicts is to generate Primary-SI and Remote-SI μPrograms differently, in such a way that the chance for conflict occurrence is low. During allocation of these resources, Primary-SI μProgram generation will use the resource with the lowest ID that is still not allocated (bottom-to-top allocation). For Remote-SI μPrograms the allocation is done in an opposite way: the resource with the highest ID is used (top-to-bottom allocation).

Figure 4.8 illustrates this for an example Primary-SI μOp (orange in the Figure) that performs two data transfers. The first transfer is from RAC 1, pRF 0, address 0 to memory port M0. During μProgram generation, the fabric link with the lowest ID (0), which is still unused is reserved for this transfer, and the pRF address with the lowest unoccupied address is reserved in the target (address 0). The second transfer is from RAC 2, pRF 0, address 0 also to memory port M0. As the fabric link with the lowest ID is already reserved, the next lowest ID is used (fabric link 1), with a similar situation for pRF address reservation (address 1 instead of 0 is used). At the same time the Remote-SI μOp (green in the figure) also performs a data transfer from RAC 7, pRF 0, address 3 to memory port M1. Using top-to-bottom allocation, fabric link 3 and pRF address 3 are used. If both μPrograms were generated using bottom-to-top allocation, then the Remote-SI would have used fabric link 0 as well, leading to a conflict with the Primary-SI and subsequent Remote-SI stalling leading to performance loss by the SI merger.

Technique	Memory Port Utilization
COREFAB	50.1%
shared fabric	42.9%
dedicated fabrics (each core)	14.5%
exclusive fabric	30.4%

Table 4.2: Fabric utilization (Memory Ports).

4.6 Case Study

This section presents in-depth analysis of a COREFAB run and fabric utilization numbers. Performance results for multiple benchmarks, comparison with state-of-the-art and overhead are presented in Section 6.4. The results in this section are from a cycle-accurate simulator (see Section 6.2 for more details) for the *i*-Core reconfigurable processor in a multi-core system with 4 cores.

To measure fabric utilization, the time-span when all applications of a workload are being executed (as in this time-span the potential to reduce under-utilization is highest) is examined. This includes both the time when computation is performed on the remote primary/remote cores or on the fabric. Table 4.2 shows the memory port utilization for a workload consisting of 4 applications (H.264 Video Encoder, SHA hashing, AES encryption, and JPEG decoding) on a fabric with 10 RACs. The comparison partners are the two approaches for using a reconfigurable fabric in a multi-core, discussed in Section 4.2, *shared fabric* and *dedicated fabrics*. Additionally, the *exclusive fabric* approach has the whole fabric available to only one core.

The system with dedicated fabrics has low fabric utilization due to (i) the small fabrics available for each core, thus a large part of the applications runs on the core pipeline instead of the fabrics (for a fabric of 15 RACs, fabric utilization is increased to 26.6% for the same workload) and (ii) a high variation between memory bandwidth requirements of the applications: H.264 performs 33× as many memory accesses from its fabric as JPEG, thus mean utilization of the dedicated fabrics is low. The system with the single exclusive fabric has better utilization, although as for such a system the application with the most demanding kernels is executed on the core connected to the fabric (H.264 in this case). As the SIs of this application use a large portion of the fabric resources, utilization is also better. The shared fabric has even better fabric utilization, as here, when one core is not using the fabric (e.g. due to executing code on the core pipeline), a different core can use the fabric. As COREFAB is the evolution of the shared fabric approach, utilization is improved again, by allowing concurrent execution of SIs, and thus on average using more fabric resources per cycle. The potential for even higher utilization exists, as the COREFAB implementation will run two SIs on the fabric at most (more concurrent SIs would require more complex hardware and reduce fabric frequency).

To show in detail how the fabric is accessed by two different cores, Figure 4.9 provides an excerpt from COREFAB executing a workload consisting of 2 applications (H.264 Video Encoder and SHA hashing) on a fabric with 8 RACs. The green bars in the

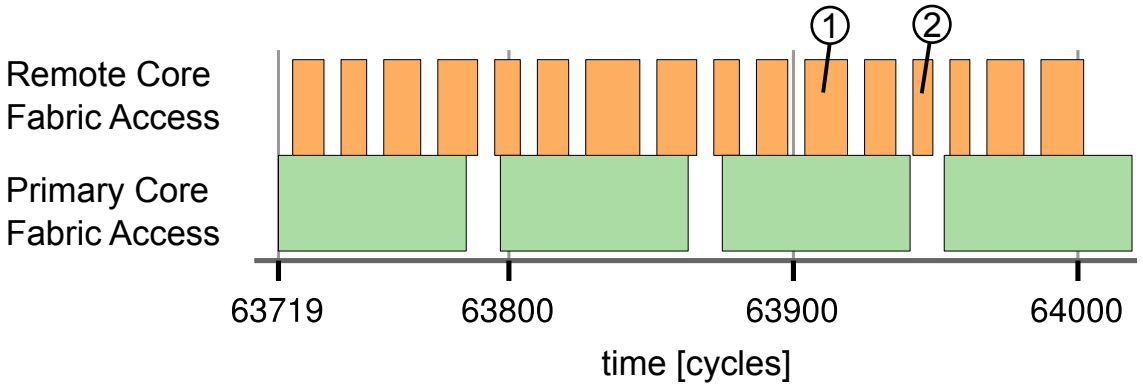


Figure 4.9: Trace of fabric accesses due to SIs from different cores.

lower part show the SI executions on the fabric of H.264 running on the primary core (core 0). This particular kernel (Motion Estimation) of H.264 uses SIs that take approx. 60 cycles on the fabric, with "gaps" in the time between SI executions (which is due to the loop and glue code execute on the core pipeline). In a shared fabric multi-core without COREFAB, only these gaps could be used to execute other SIs.

COREFAB allows the remote core (running SHA – orange bars), access to the fabric even while the SIs of the primary core are running (e.g. at ① the fabric is accessed by both cores at the same time), resulting in a better overall throughput of SIs. The different length of the orange bars is due to unresolved conflicts during concurrent fabric accesses by both cores, and therefore the SIs of the remote core being stalled (longer orange bars). At ②, the SHA SI of the remote core is being executed when the primary core does not use the fabric, thus the remote core has exclusive and thereby conflict-free access to the fabric, allowing the SI to be processed quicker than at ①, when both SIs access the fabric. The number of conflicts depends not only on the SIs, but also on how the SIs are "aligned", thus the variation in the SHA SI latency.

4.7 Summary

This chapter shows that due to varying application demands, a reconfigurable fabric will often be under-utilized, as the less demanding Special Instructions will use less fabric resources (such as reconfigurable area, memory bandwidth). While more demanding applications utilize a much larger share of these resources to meet their performance constraints, they often use predominantly one type of resource, while under-utilizing the others. This kind of fabric under-utilization results in inefficiency and thus wasted performance. The proposed solution is to use the fabric in a multi-core system, and share the fabric between multiple SIs (which are dispatched from different cores). The goal is to reduce this inefficiency and improve the performance of the whole multi-core system.

The proposed solution, COREFAB, combines the advantages of existing approaches to using reconfigurable fabrics in multi-cores, which are (i) the flexibility to achieve a high fabric utilization under varying application scenarios, and (ii) the ability to concurrently run multiple SIs on the fabric.

COREFAB introduces a protocol to provide non-reconfigurable cores access to the fabric, allowing them to use SIs, as if they were regular reconfigurable cores. Once multiple SIs have been started on the fabric, COREFAB uses the novel SI merging concept to run multiple SIs in parallel. To do this, in each cycle the concurrently running SIs are examined and merged into a single fabric configuration on-the-fly in hardware. In some cases merging is not possible due to resource conflicts, as multiple SIs request an exclusive resource simultaneously. The occurrence of conflicts is reduced by a modification to the generation of SI μPrograms, and the remaining conflicts are handled by the merger by stalling one of the conflicting SIs. Due to the fine-grained nature of the cycle-by-cycle merging, a single conflict delays an SI by only one cycle.

Results show that COREFAB improves fabric utilization, and an analysis of a COREFAB run compared to that of a state-of-the-art approach, shows a better throughput of SIs when using COREFAB (a detailed evaluation is performed in Section 6.4).

5 Flexible Fabric Use in Multi-Core and Multi-Tasking Systems

Chapters 3 and 4 have shown how reconfigurable fabric can be used efficiently in multi-core and multi-tasking systems. Both chapters discussed approaches which have in common that different applications share the fabric. Exactly which applications actually use the fabric at run-time is not known at compile-time, as this would limit the system to static workloads and require precise knowledge of application timing (which can depend on user input). The reconfigurable fabric is therefore allocated to tasks or cores at run-time, and SIs must be *flexible* enough to be able to run on whichever part of the fabric was allocated to the application. This poses a challenge, as on one hand synthesizing SIs at run-time would incur too large an overhead to be beneficial (it would require a full high-level synthesis, placement and routing of the SIs). On the other hand, synthesizing SIs at compile-time would constrain the system so much that the benefits from the multi-core and multi-tasking approaches presented earlier would be negated.

This chapter presents how SIs can be synthesized in a way that an application can fully utilize its allocated fabric, no matter how it is distributed, while keeping the overhead low enough so that the benefits of using the fabric are hardly diminished. The approach is called *partial online synthesis* and is based on performing as much of SI synthesis as possible at compile time, and only performing those operations that are directly dependent on the state of the fabric (i.e. which part of the fabric was assigned to which task, which accelerator is loaded into which RAC) at run-time.

In detail, the chapter is organized as follows: A motivation for splitting SI synthesis between compile- and run-time is presented in Section 5.1. Section 5.2 defines the problem and provides an overview of the solution, partial online synthesis of SIs. Binding and Placement decisions can have an effect on SI performance which is discussed in Section 5.3. Section 5.4 presents the approach to placing accelerators on the fabric in a way that SI performance is maximized. In order to allow an SI to run on the fabric, the SI must be bound to a fabric configuration. Algorithms for this are presented in Section 5.5, again with the focus of keeping SI performance as high as possible. Binding of SIs can take a non-negligible amount of time. To reduce this overhead while still retaining the flexibility benefits, a software-cache and cache-aware placement are presented in Section 5.6. Partial online synthesis is evaluated later in Section 6.5 using multiple applications and including overhead analysis.

5.1 Motivation

An SI is described by a data-flow graph of an application kernel (or a part of it). The graph is independent of the architectural details of the fabric. Figure 5.1a shows the graph for the SI "Motion Compensation – Horizontal" (MCHz) from the kernel responsible for encoding a video frame in a H.264 video encoder application. Throughout this section, the focus for this SI will be on the main processing part of the SI, which is shaded gray in Figure 5.1a. In order to run an SI on the fabric, the exact fabric resources that the SI will use throughout its execution must be known, i.e. :

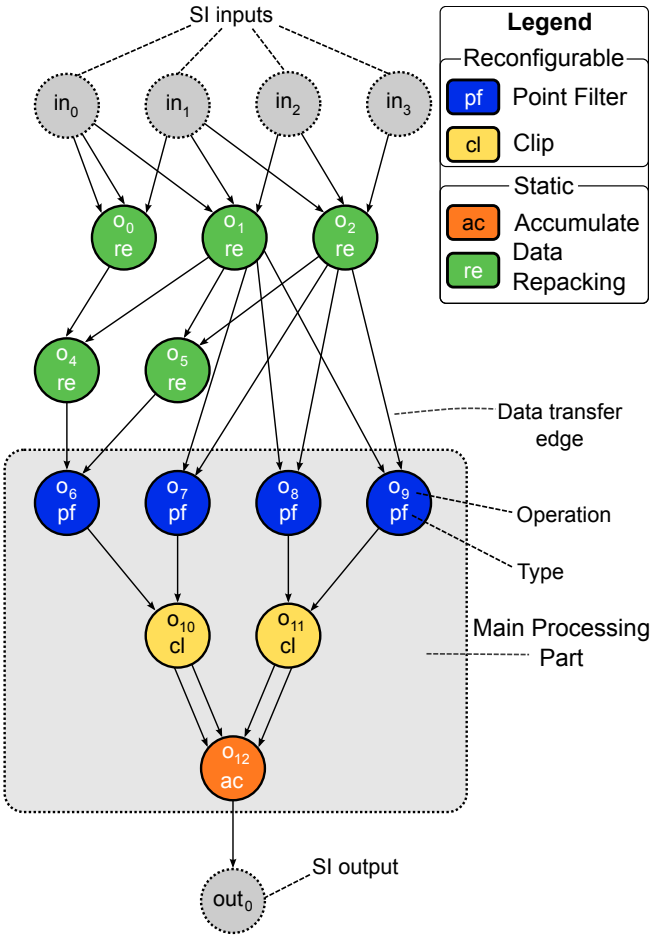
1. how many/which accelerators are used?
2. which operations and data-transfers are performed in a particular cycle?
3. into which RACs are the accelerators loaded?
4. on which RACs are the operations of the SI executed?
5. which links are used to communicate between accelerators/memory ports?
6. to which addresses in the private register files (pRFs) are intermediate results stored?

Item 1 determines how many SI variants will be available, and is a area vs. performance trade-off. These different variants are then scheduled, providing a scheduled DFG for each SI variant (Item 2). For the MCHz SI, 2 variants with their corresponding schedules (main processing part only) are shown in Figures 5.1b and 5.1c. Variant 2 exploits the parallelism of the SI to a larger degree and is therefore faster (less control steps than variant 1). Items 4–6 address μProgram generation, also called *Binding* (details in Section 5.5), as the SI DFG is bound to a particular fabric configuration. Binding depends on Item 3, which determines into which RACs the accelerators of the SI are loaded, also called *Placement* (details in Section 5.4).

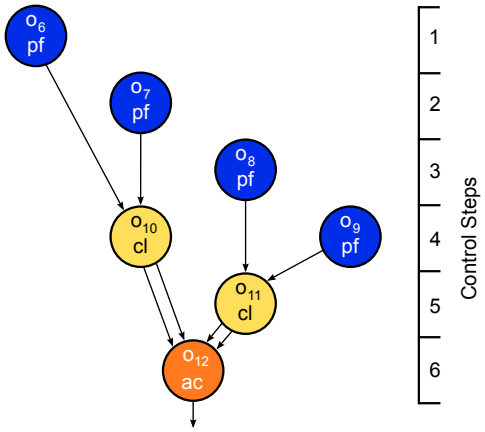
Choosing SI variants and scheduling them is done at compile-time, as the only relevant parameter is the number of RACs available on the fabric, which is a design-time parameter (and does not change at run-time). Items 3–6 (Placement and Binding), can be done both at compile-time, or at run-time. The remainder of this section will show why doing it at run-time is the better choice.

Figure 5.2 shows the configurations of the fabric (i.e. which accelerators are loaded into which RACs) and the allocations of RACs to tasks at different points in time. Similar situations can be encountered during typical workloads. While multiple SIs are used in this example, the focus is on the MCHz SI from Figure 5.1a.

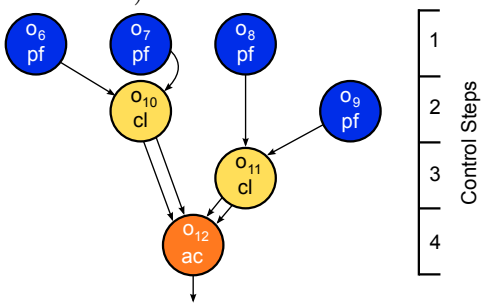
In Figure 5.2a the fabric has been allocated to two tasks, A (allocated 5 RACs) and B (allocated 2 RACs). Fabric allocation is performed whenever a tasks requests the fabric (usually when the task starts), or when it releases it (usually when the task terminates), using the techniques described in Section 2.3.1. The tasks can either both run on the reconfigurable core (see Chapter 3), or one can run on the reconfigurable core and another on a regular core in a multi-core system (see Chapter 4). Task A is an H.264 video encoder and is using the MCHz SI. As A employs other SIs in addition to MCHz, MCHz is using 2 out of 5 RACs and therefore variant 1 is used.



(a) Data Flow Graph for MCHz SI.



(b) MCHz variant 1: scheduled main processing part (grey shaded area in Figure 5.1a) for 2 RACs (accelerators: 1×pf, 1×cl).



(c) MCHz variant 2: Scheduled main processing part (grey shaded area in Figure 5.1a) for 4 RACs (accelerators: 3×pf, 1×cl).

Figure 5.1: Data Flow Graph and schedules for 2 variants for the SI Motion Compensation – Horizontal from the H.264 Video Encoder application.

At some later time, a new task C arrives, while A and B are still running (Figure 5.2b). Fabric re-allocation takes one RAC from A and assigns it to C. C will then load its own accelerator into the newly allocated RAC. In general, a task will use all of the RACs that it is allocated to gain the most performance out of its SIs, thus the RAC that is taken away from A will have an accelerator that is required for one of the SIs currently used by A. In the example the RAC that was taken away from A had the Clip accelerator loaded, which is required for MCHz. To continue running MCHz on the fabric, Clip is reconfigured into one of the remaining 4 RACs in the fabric share of A, replacing an existing accelerator[1]. As Clip is now in a different RAC, the μProgram for MCHz is no longer valid, as the operations o_{10} and o_{11} have

[1]Task A could also decide that the performance loss due to replacing one of its other accelerators would be too great, and MCHz should be executed on the core pipeline.

to be performed in a RAC where Clip is loaded. Thus in order to continue using the MCHz SI on the fabric, the μProgram has to be re-generated for the new fabric configuration.

Figure 5.2c shows a later situation where both tasks B and C have finished, but A is still running due to its larger workload than the other two tasks. At this point A is the only task left, and therefore all fabric has been allocated to A. A can now use the additional RACs (compared to what it was allocated when multiple tasks were running) to speed up its SIs further, in the example by loading more accelerators for MCHz. This allows to use variant 2 of MCHz (Figure 5.1c), which uses 2 additional instances of the Point Filter accelerator to improve SI performance. As before, the fabric allocation has changed at runtime, which affects accelerator placements and the μPrograms for the SI variants that use these accelerators have to be bound to the new fabric configuration.

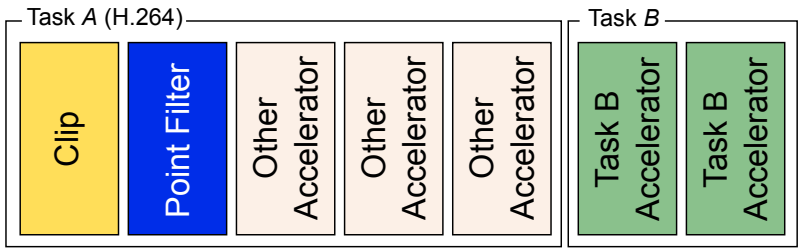
If the fabric allocations (and with that fabric configurations) were known at compile-time, then the μPrograms for the SIs could be generated at compile-time, and at run-time the μProgram for the current fabric configuration would simply have to be retrieved from memory. However, in a system with a dynamic workload (i.e. if and when a task arrives is not known at compile-time), the fabric allocations (and with that the fabric configurations) are not known at compile-time. Therefore either μPrograms have to be generated for all possible fabric configurations, or the μPrograms have to be generated at run-time.

The number of possible fabric configurations is very large, as it depends on (i) which accelerators are already loaded, (ii) which may be replaced, and (iii) which shall be reconfigured. When n RACs are available and an SI requires m different accelerator types with $q(t_k)$ accelerators per type t_k, then the number of different placement possibilities (after writing down the binomial coefficients, bringing them into their factorial form and reducing the terms) is shown in Equation (5.1).

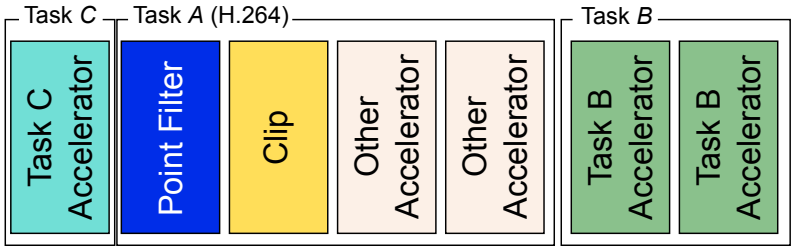
$$\frac{n!}{\prod_{k=0}^{m-1} (q(t_k)!) * \left(n - \sum_{k=0}^{m-1} q(t_k)\right)!} \tag{5.1}$$

In the MCHz examples above, a fabric with 7 RACs was used and MCHz required 2 different accelerator types (Clip and Point Filter). For SI variant 1 there are 604,800 possible fabric configurations, and for variant 2 there are 5,040 fabric configurations. The fully scheduled SI variants (not just the main processing part) consist of 10 control steps for variant 1 and 8 control steps for variant 2. While μProgram size is fabric architecture dependent, for the i-Core reconfigurable processor (Section 2.6) this results in a μProgram size of 1280 bytes and 1024 bytes for each variant. For all possible fabric configurations the storage requirement for both SIs would be 743.2 MB for just one SI. In reality, this number increases significantly, as

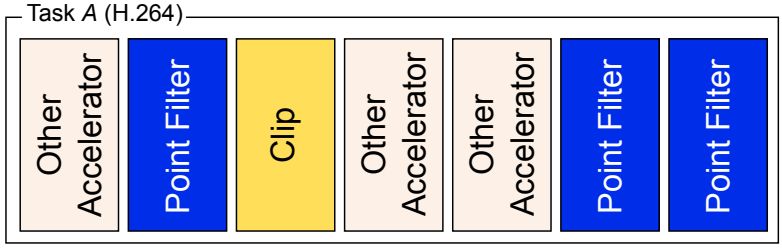
- there are more RACs on the fabric,
- more variants for each SI are available,



(a) Task A is assigned 5 RACs, Task B 2 RACs. The MCHz SI used by Task A uses 2 out of 5 accelerators and is used in variant 1 (Figure 5.1b)



(b) A new Task C enters the system and is allocated 1 RAC, which is taken away from Task A. MCHz is used in the same variant as before, but requires a new μProgram, as the fabric configuration has changed.

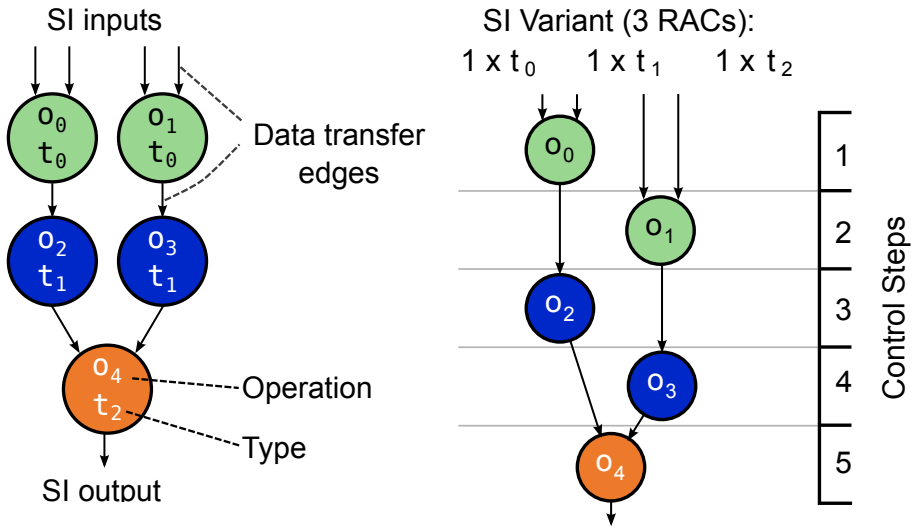


(c) Tasks B and C have terminated, and A is allocated all 7 RACs. This allows using MCHz in the faster variant 2, which requires a new μProgram.

Figure 5.2: Fabric configuration changes over time as new tasks enter and leave the system, changing fabric allocation. To execute an SI (here: MCHz), a μProgram is required, which is specific to a particular fabric configuration and therefore needs to be generated at run-time.

- complex applications have more SIs, and there are multiple applications.

This clearly shows that preparing and storing μPrograms for all possible fabric configurations at compile-time for all SI variants of all applications that shall execute is practically infeasible and therefore μPrograms have to be generated at run-time in order to make use of the flexibility provided by the fabric in multi-tasking and multi-core scenarios.



(a) Data-flow graph (DFG) of an SI.

(b) Scheduled SI variant of the DFG.

Figure 5.3: Data-flow graph and scheduled variant for Special Instructions.

5.2 System Overview and Problem Statement

The techniques presented in this chapter are used to support the approaches presented in Chapters 3 and 4. Therefore, the system used is also the same, a fine-grained reconfigurable processor (see Section 2.6).

An introduction to Special Instructions is provided in Section 2.2. Formally, an SI is described by a data-flow graph (DFG), which consists of $|V|$ *operation* nodes $o_i, 0 \leq i < |V|$ of $T \leq |V|$ different *types* $t_k = t(o_i), 0 \leq k < T$ and directed *data transfers* $d_{i,j}$ between nodes o_i and o_j (Figure 5.3a). Provided that there are $|A|$ accelerator types available to the system, each operation o_i is then implemented by an accelerator $a_m, 0 \leq m < |A|$ of the same type $t(o_i) = t(a_m)$. An SI exists in different SI variants that differ in the number of accelerators that are used to implement the DFG (area vs. performance trade-off). Figure 5.3b shows an SI variant for the DFG in fig. 5.3 for 3 accelerators of types t_0, t_1, t_2 each. The quantity of accelerators of a certain type t_k that are used to implement an SI variant is denoted as $q(t_k) \in \mathbb{N}$. It affects the schedule of the DFG and with that number of control steps that are required to execute the SI variant (and thereby its execution latency in cycles).

SI variants are obtained by scheduling the DFG of an SI using an accelerator allocation $q(t_k)$. The scheduling of the DFG for a given allocation assigns operation o_i to control step $s_l = s(o_i)$ (where the function $s(o_i)$ returns the control step in which operation o_i is scheduled), such that (i) in each control step not more accelerators are required than assigned by the allocation (see Equation (5.2)), (ii) the data dependencies between the operations are considered (see Equation (5.3)), and (iii) the number of control steps (and therefore the SI execution latency) is minimal. SI Scheduling

is done offline using high-level synthesis scheduling algorithms such as LIST or Force-directed Scheduling [PK89].

$$\forall s_l \, \forall t_k : \left| \sum\nolimits_{o_i, t(o_i)=t_k, s(o_i)=s_l} 1 \right| \leq q(t_k) \tag{5.2}$$

$$\forall d_{i,j} : s(o_i) < s(o_j) \tag{5.3}$$

Placement of an accelerator a_m decides into which RAC $c_p = c(a_m)$ it shall be reconfigured. This depends on the fabric share and (as discussed in Section 5.1) needs to be performed at run-time. The placement has to ensure that different accelerators (potentially of the same type) are placed in different RACs (see Equation (5.4)).

$$\forall a_m, a_n : m \neq n \Rightarrow c(a_m) \neq c(a_n) \tag{5.4}$$

Binding is the essential part of μProgram generation, where components of the scheduled SI DFG (operations o_i and data transfers $d_{i,j}$) are bound to the reconfigurable fabric (RACs and links). As it depends on the placement of accelerators, it also needs to be performed at run-time. While one placement decision is needed per accelerator that shall be reconfigured, one binding decision needs to be carried out per SI variant, once all its accelerators are placed. The binding $b(o_i) = c_p$ assigns operation o_i to a RAC c_p such that the types match (Equation (5.5)) and two operations in the same control step are not assigned to the same RAC (Equation (5.6)). Note that multiple accelerators of the same type may be reconfigured to different RACs, so binding has to decide which of them shall execute an operation of that type. An operation o_i that executes on RAC c_p stores its intermediate results in a private register file (pRF) of the connector that is attached to RAC c_p. Binding needs to decide to which address (in the pRF) a result shall be written without overwriting results that are still needed in a later control step. Therefore, during compile-time, a lookup table is created that describes for all operations o_i the control step $s(o_i)$ in which they create a result and after which control step $s_l = max\{s(o_j) : \exists d_{i,j}\}$ it is no longer required. This table is used for binding operations to RACs at run-time to mark pRF addresses as occupied or as free.

$$b(o_i) = c_p = c(a_m) \Rightarrow t(o_i) = t(a_m) \tag{5.5}$$

$$\forall o_i, o_j, s(o_i) = s(o_j) : i \neq j \Rightarrow b(o_i) \neq b(o_j) \tag{5.6}$$

When executing an operation o_j that demands input data from another operation o_i the data transfer $d_{i,j}$ is performed in control step $s(o_j)$ and needs to be bound to a *link*, i.e. a connected set of link segments. To establish the data transfer $d_{i,j}$ on a particular link l_s, the link segments in the interval $[b(o_i), b(o_j)]$ are used, i.e. all link segments on link l_s between the RACs to which o_i and o_j were bound. Multiple

data transfers occurring in the same control step, that are originating from the same source to different targets (e.g. $d_{i,j}, d_{i,y}$) can use the same link l_s. Furthermore, multiple transfers originating from different sources (e.g. $d_{i,j}, d_{x,y}, i \neq x$ or $j \neq y$) can use the same link l_s in the same control step, if eq. (5.7) holds (i.e. if their intervals do not overlap). If they overlap, then different links $l_r, l_s, r \neq s$ need to be used or $d_{i,j}$ and $d_{x,y}$ need to be performed in different control steps.

$$[b(o_i), b(o_j)] \cap [b(o_x), b(o_y)] = \oslash \tag{5.7}$$

A bound control step of an SI variant encompasses all operations and data-transfers that are performed simultaneously. The bound control step corresponds to a μOp in the μProgram, which is stored in the SI μProgram memory (see Section 2.6, Figure 2.19). Once binding is completed and all accelerators are loaded the corresponding SI is marked as "executable" in the SI availability table (Figure 2.19). The SI will now be executed on the fabric using the bound SI variant.

The challenges for placement and binding are that their decisions affect the execution latency of SIs. In the best case, each control step is executed in one cycle, but depending on placement and binding, a control step might require multiple cycles (details in the next section), thus increasing the SI variant execution latency. As SIs are designed to accelerate computationally intensive kernels, delaying their execution directly affects an application's performance.

5.3 Impact of Placement and Binding Decisions

Before presenting the algorithms for placement and binding, the performance of impact of these decisions will be discussed. While ideally a data transfer between two accelerators can be completed in one cycle in the best case, it may take longer depending on the placement of these two accelerators into RACs on the fabric and the binding of DFG operations to the RACs. These additional delays are caused by *Link Saturation Hazards* (LSH) and *Transfer Delays Hazards*, which will be discussed in this section. Figure 5.4a) shows an excerpt from a scheduled DFG of an example SI variant, that will be used to illustrate the hazards.

A link saturation hazard occurs if the number of transfers exceed the number of links between any two neighboring connectors[2]. An example for an LSH is shown in Figure 5.4c) (for simplicity, in the example the fabric only has two links). It occurs when binding tries to reserve a link for $d_{3,5}$ in control step s_1. All link segments between c_2 and c_3 are already used for other data transfers during this control step ($d_{1,4}$ and $d_{2,4}$). Transfer $d_{3,5}$ requires a link segment between c_2 and c_3, and thus exceeds link capacity and causes a LSH. The hazard is resolved by delaying those data transfers (and the corresponding operations) of s_1 that cause a LSH ($d_{3,5}$ is

[2]Each link is uni-directional, therefore only transfers that go into the same direction can cause a LSH.

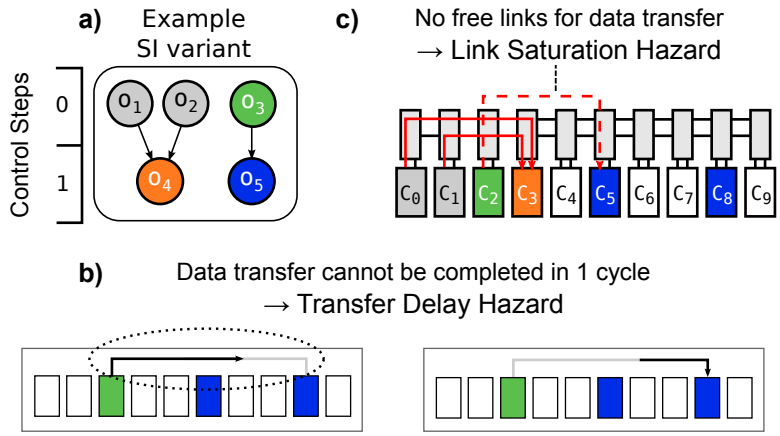


Figure 5.4: Examples for a Link Saturation Hazard (LSH) and Transfer Delay Hazard (TDH).

delayed in the example) by 1 cycle. If these delayed data transfer still cause LSHs, the same method for hazard resolution is applied again (i.e. those transfers that caused LSHs are delayed again by 1 cycle), until no more hazards are caused. Each control step with an LSH causes SI latency to increase by at least 1 cycle. The occurrence of LSHs depends on fabric architecture parameters (number of links), the scheduled DFG (number of data transfers in each control step), placement of accelerators and binding of operations (RAC choice for an operation).

A transfer delay hazard occurs when for a data transfer $d_{i,j}$ the *distance* $|k-l|$ of the RACs c_k (to which o_i was bound) and c_l (to which o_j was bound) is too long to be traversed in one cycle, i.e. $|k-l|$. In the example Figure 5.4b) o_3 was bound to c_2 and o_5 to c_8. If the link speed is below 6 link segments/cycle (i.e. at most 6 link segments can be traversed in one cycle), then transfer $d_{3,5}$ will cause a TDH. To resolve the hazard, the transfer is prolonged for one additional cycle (or longer, if required), by delaying the control step by 1 cycle, or more if required (similar to LSH resolution). The maximum *distance* D that can be traversed in one cycle depends on the technology and operation frequency and different values for D are evaluated in Section 6.5. The required number of cycles to complete a data transfer $d_{i,j}$ is $\lceil |c_k - c_l| / D \rceil$. In addition to fabric architecture parameters, TDH occurrence depends on the placement of accelerators and binding of operations, but it does not depend on the scheduled DFG (unlike LSHs).

As both types of hazard depend on placement and binding decisions, and hazard occurrence causes increased SI latency (and therefore performance loss), the placement and binding algorithms presented in the next sections are designed with the goal to reduce the occurrence of hazards.

5.4 Placing Accelerators

An SI can only be executed once all the accelerators required for the SI variant have been placed onto the fabric. Placement only determines where to load an accelerator, but does not perform the reconfiguration itself (which can only be done after placement).

The location of an accelerator on the fabric can affect the execution latency of all SIs that use this accelerator (see Section 5.3). The goal of a good placement algorithm should be to reduce the latency that occurs due to transfer delay hazards, i.e. placement needs to consider the data transfers between operations. At run-time, the run-time system determines the amount of accelerators that shall be reconfigured to accelerate a computationally intensive kernel (see Section 2.3) and provides the following input to the placement algorithm:

1. C, the current configuration of the fabric (i.e. which accelerators are loaded into which RACs),
2. S, the set of SI variants (which, depending on the kernel can originate from different SIs) that are used in that kernel,
3. t_R, an accelerator type that is already configured on the fabric but that is not required in the current kernel and thus can be replaced, and
4. t_P, an accelerator type that needs to be reconfigured and thus needs to be placed onto the fabric.

In order to reduce the data transfer latencies between accelerators, placement may attempt to load all accelerators of an SI variant into neighboring RACs, i.e. to cluster the accelerators of an SI variant $s \in S$. This approach is called *Cluster Placement*. Let A be the set of accelerators that are required to implement an SI variant s. A_L shall be the leftmost RAC on the fabric and A_R the rightmost RAC that contains an accelerator $\in A$, i.e. $A_L = min\{c(a_m) : a_m \in A\}$ and $A_R = max\{c(a_m) : a_m \in A\}$. Out of all possible data transfers that could be performed by s, none is longer than the distance between A_L and A_R, i.e. $|A_L - A_R|$ which is the *expansiveness* of SI variant s. For example, Figure 5.5 shows a sequence of accelerators that shall be placed to implement an SI variant. A_L is c_3 and A_R is c_9, thus the expansiveness of this SI variant is 6. Cluster Placement iterates over all SI variants s and all RAC candidates c_p that match the replacement type t_R or are empty and computes the expansiveness of s if the accelerator would be placed to c_p. The RAC candidate that results in the smallest expansiveness is selected. If multiple candidates result in the same expansiveness, then the first candidate with that value is selected. The complexity of the Cluster Placement is $\mathcal{O}(|RACs| * |SI\ variants|)$.

The main drawback of Cluster Placement is that it does not consider *how often* a data transfer between two RACs occurs, e.g. there might be no data transfer between A_L and A_R at all in s. For instance, assuming that the most frequent data transfer in Figure 5.5 is between accelerator type t_0 and t_2, it would be better to place the accelerator with type t_0 into RAC 0. This would increase the expansiveness of the SI variant from 6 to 7, but the distance for the most frequent data transfer would

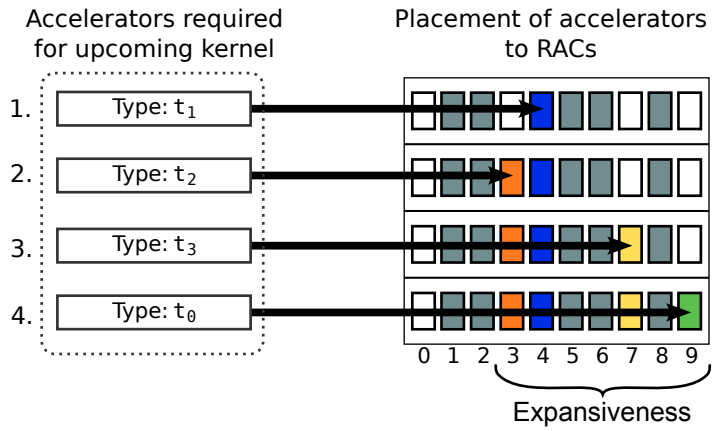


Figure 5.5: Example for placement of accelerators using Cluster Placement. Grey shaded RACs are in use and currently unavailable for placement. The expansiveness of the resulting SI variant is 6.

be reduced from 6 to 3. The resulting SI variant would have reduced latency and thereby improved performance.

Better placement can therefore be obtained by taking into account the communication between accelerators in a SI variant. Communication patterns can be extracted from the scheduled DFG for a SI variant, which is also used as the input for binding (Section 5.5). The *Connectivity Placement* algorithm has been developed to perform communication-aware placement. During scheduling of the SI DFGs at compile-time, all SI variants s are annotated with the *connectivity* between any two operation types t_x and t_y, defined as the amount of data transfer edges $d_{i,j}$ between operations of these types (Equation (5.8)).

$$s_{conn}[t_x, t_y] = |\{d_{i,j} | i : t(o_i) = t_x,\ j : t(o_j) = t_y\}| \tag{5.8}$$

At run-time, Connectivity Placement iterates over all RAC candidates c_p that match the replacement type t_R or are empty, and examines the connectivity for all SI variants $s \in S$ that require the accelerator type t_P. The distance between c_p and all other RACs c_q is weighted with the compile-time prepared connectivity value of s and summed up to a score. The algorithm selects the RAC c_T with the minimal total connectivity score. The complexity of the Connectivity Placement is $\mathcal{O}\left(|RACs|^2 * |SI\ variants|\right)$.

After all accelerators required for an SI variant are placed (but not necessarily already reconfigured), the resulting fabric configuration is used to bind the SI variant. Additionally, after placing each accelerator, the accelerator type and location are pushed to the reconfiguration queue, which is used by the OS to reconfigure the fabric.

One disadvantage when using the placement algorithms presented in this section, is that if the same SI variant needs to be placed at a later time again, it may

result in a different fabric configuration, e.g. when the same SI is used in a different kernel, and RAC candidates are scored differently due to different requirements from other SIs. This non-reproducibility of fabric configurations poses a problem when caching μPrograms (see Section 5.6), a technique for reducing the overhead of generating μPrograms. A cache-friendly placement algorithm is therefore developed in Section 5.6.2.

5.5 Binding Special Instructions

After the placement of accelerators is completed, the *fabric configuration* is available, i.e. the information which accelerator a_m will be loaded into which RAC c_p, with $c_p = c(a_m)$. The compile-time scheduled DFG of an SI variant does not specify concrete resources such as RACs, fabric link IDs or pRF addresses, instead using configuration-independent operations, data-transfers and intermediate results. In order to run an SI variant on the fabric, the configuration-independent DFG has to be *bound* to a fabric configuration (as explained in Sections 2.3, 2.6.3 and 5.1), resulting in a μProgram.

The scheduled DFG of an SI variant consists of control steps, which are bound sequentially. Each control step generates one or more μOps in the μProgram. A μOp encodes all activity that happens in the fabric during one cycle.

In the following it is assumed that the μProgram is generated for a Primary-SI (see Chapter 4) when sharing the fabric in a multi-core. When generating a Remote-SI μProgram the algorithms for data transfer and intermediate result binding are slightly modified, as described in Section 4.5.3. The algorithms for operation binding are the same for both Primary-SI and Remote-SI μPrograms, as each core is assigned its dedicated RACs.

5.5.1 Operations

To bind an operation o_j from control step s_l to a RAC c_p, a list of *RAC candidates* is constructed such that

1. the accelerator types in the RACs match the operation o_j (Equation (5.5)),
2. no other operation is bound to c_p in this control step (Equation (5.6)),
3. the connector that links c_p with other RACs has enough space in its pRF to store the results of o_j.

When the fabric configuration has only one RAC with an accelerator of the required type, then binding o_j is trivial. When multiple such RACs exist, the binding algorithm needs to decide which of the RACs to use for o_j. For instance, in Figure 5.4 operation o_1 can be bound to RACs c_5 or c_8. Binding to c_8 leads to a transfer delay hazard, potentially resulting in increased latency of the SI variant.

The following strategies for binding of operations have been developed:

Algorithm 1: RAC selection in Communication Aware Binding

```
input:
  Reconfigurable Fabric fabric, operation o that needs to be bound
output: Target RAC c_B
c_B := ∅, best_latency = MAX_INT
foreach RAC c_k ∈ fabric.RACs do
    // RAC type matches operation
    if t(C[c_k]) ≠ t(o) then continue
    // No other operation bound to RAC
    if not condition (ii) then continue
    // Enough space for intermediate results
    if not condition (iii) then continue
    fabric.bus.clear(), d_k := 0, saturation := 0
    foreach operand source o_i of operation o do
        // o_i are those operations that are used as input to o
        d_k := max(d_k, ⌈|i − k|/bus_speed⌉)
        line := fabric.bus.reserve(i, k)
        if line = -1 then saturation++
        fabric.bus.clear()
    expected_latency := d_k+ saturation
    if expected_latency ¡ best_latency then
        best_latency := expected_latency
        c_B := c_k
return c_B
```

First Fit Binding (FFB) scans the RACs from left to right and returns the first valid RAC candidate that fulfills the above mentioned constraints. FFB does not consider any impact of RAC choice on hazard occurrence, but is the simplest and thus fastest binding algorithm. The complexity of FFB is $\mathcal{O}(|RACs|)$.

Communication-Aware Binding (CAB) (shown in Algorithm 1) takes the transfer of input data from operation o_i to o_j. o_i occurs at an earlier control step, and therefore already been bound to a RAC. CAB temporarily binds o_j to each candidate RAC c_p. For each such binding CAP examines all data transfers $d_{i,j}$ for Link Saturation Hazards and Transfer Delay Hazards, by temporarily binding each data transfer. If a hazard occurs, the number of cycles that are required to resolve the hazards are computed. The candidate RAC that results in the least number of cycles to resolve the hazards is used to bind o_j. The complexity of CAB is $\mathcal{O}(|RACs| * |data\,transfers|)$.

Communication-Lookahead Binding (CLB) is an extension of CAB. As with CLB, the algorithm scores each RAC candidate for operation o_j according to the additional cycles incurred due to hazards. However, when scoring, in addition to the data transfers $d_{i,j}$ of *input data* from o_i, CLB also considers *output data*,

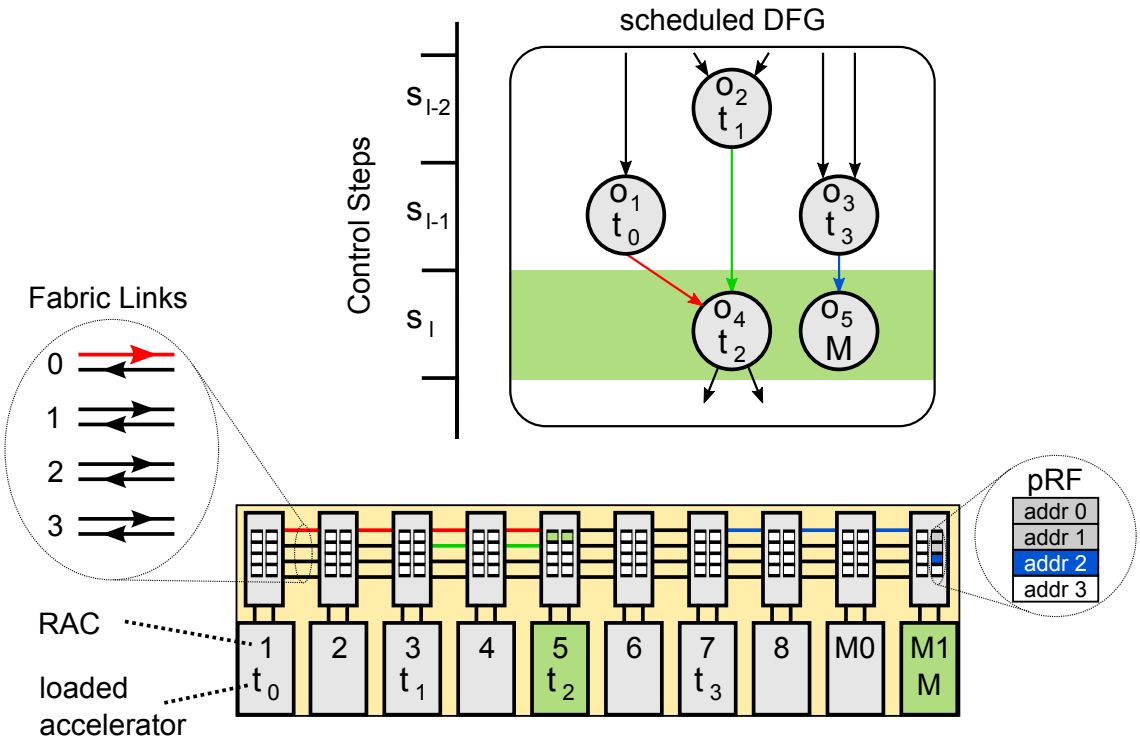


Figure 5.6: Binding of data transfers and intermediate results for control step s_l (highlighted light green) to fabric links and pRF addresses. Binding of operations to RACs has been performed already.

i.e. data transfers $d_{j,k}$ of intermediate results from o_j to other (yet unbound) nodes o_k. To consider these "future" data transfers $d_{j,k}$, CLB needs to know to which RAC o_k is bound. As this information is not yet available at the time o_j is bound, CLB assumes that o_k will be bound to the RAC c_q has the suitable accelerator for o_k and that is closest to the currently examined RAC candidate c_p, i.e.

$$\forall c_q \in \{c_x | t(c_x) = t(o_k)\} : \min |c_q - c_p|$$

CLB selects the RAC that results in the least number of cycles to resolve the hazards for input and output data transfers of o_j. The complexity of CLB is $\mathcal{O}\left(|RACs|^2 * |Data\,Transfers|\right)$.

5.5.2 Data Transfers and Intermediate Results

Passing of operands and intermediate results via edges in SI DFGs corresponds to data transfers on links between connectors on the fabric. Therefore, in addition to binding operation nodes, their input and output edges in the DFG have to be bound to connectors and links. After the operation nodes scheduled in a control step have been bound to RACs, the corresponding data transfers from the scheduled DFG (edges) have to be bound to fabric links. A data-transfer $d_{s,d}$ has a source operation

o_s that has been bound to source RAC c_s and a destination operation that has been bound to destination RAC c_d.

Wasteful use of fabric links can cause Link Saturation Hazards, which would increase SI latency. As fabric links are segmented, a single link can support multiple data transfers, as long as they do not have a conflict, reducing the occurrence of Link Saturation Hazards. There is a conflict between two data transfers $d_{i,j}$ and $d_{s,d}$, if they occur in the same cycle, and both transfers either have the direction left-to-right (Equation (5.9)) and overlap (Equation (5.10)), or both transfers have the direction right-to-left (Equation (5.11)) and overlap (Equation (5.12)).

$$(d > s) \wedge (j > i) \tag{5.9}$$
$$(s < j) \wedge (i < d) \tag{5.10}$$

$$(d < s) \wedge (j < i) \tag{5.11}$$
$$(s > j) \wedge (i > d) \tag{5.12}$$

To bind data transfers, the algorithm goes over the list of unbound data-transfers for the current control step s_l, determines the source and destination RAC (both are known, as the source RAC was bound in a previous control step, and the destination RAC was bound in the current control step before data-transfer binding), and binds them to the link with the lowest ID. In the example in Figure 5.6, the data transfer between o_1 and o_2 is bound to fabric link 0. The transfer from o_2 to o_4 overlaps with the last data transfer and is bound to link 1. Finally, the transfer from o_3 to o_5 does not overlap with either and is bound to link 0. This control step therefore occupies 2 fabric links.

If the transfers require more links than are available, a link saturation hazard occurs, and an additional μOp is allocated for the current control step. Those transfers that do not fit into the number of links available, are performed in the new μOp. This μOp allocation to handle outstanding data transfers is done until no transfers remain.

As most SIs have more than one control step, the intermediate results from operations in earlier control steps are stored within the fabric until they are needed in later control steps. The private register files (pRFs) in each connector allow storing of operands from the fabric links or result data from the RAC attached to the connector. For each operation that finishes in the current control step, the lowest free address in the pRF is reserved. During offline DFG scheduling *liveness* information is annotated to each result of an operation. Liveness consists of the start control step when the result is generated and the final control step when the result is read for the last time. When allocating fabric links to read input operands, this liveness information is checked to see if the operands have exceeded their liveness, and if so, the corresponding pRF addresses are freed and can be reused. To store a value, the lowest unused address in a pRF is allocated.

In the example of Figure 5.6, operation o_4 generates two intermediate values. To store these values in the pRFs of RAC 5 (to which o_4 has been bound), the lowest

addresses for both pRFs are used – address 0. Operation o_5 is a memory port operation (store), which buffers input data in its pRFs before writing it out in the next cycle. Here, addresses 0 and 1 are currently in use (e.g. if this memory port was used earlier to load data for other accelerators), so address 2 is used to buffer the input data.

5.6 Caching μPrograms

Measurement on the FPGA prototype (see Section 6.1) have shown that binding of a typical SI variant (e.g. DCT SI, 5 RACs) takes approximately 4 ms, while reconfiguration of one RAC takes 0.6-0.7 ms assuming a 50% load of the system bus. In general, a kernel prefetch selects at least one SI variant for each RAC that will be reconfigured. If the time to bind an SI variant exceeds the time to reconfigure the RACs required for it, the application will run at reduced performance, as although the RACs have the required accelerators loaded already, they cannot be used until the SI variant is bound and the SI μProgram is available.

Instead of re-generating SI μPrograms each time the same kernel is prefetched, a better approach is caching of SI μPrograms and instead of binding, simply retrieving the cached μProgram. When an SI variant is bound, the run-time system updates the corresponding entry in the SI μProgram cache. Upon future requests to bind this SI variant to the same fabric configuration, the binder will check the cache and return the SI μProgram for this SI variant immediately. During work with the partial online-synthesis system developed in [GBH12], situations could be observed where fabric configurations used to run a particular SI variant would vary during the lifetime of an application, but the RACs required for this SI variant would stay the same. This allows previously generated SI μPrograms to be reused, as long as the fabric allocation for the application does not change, but keeps the system flexible enough to re-generate SIs if a new task arrives (as motivated in Section 5.1), causing parts of the fabric to be re-allocated (and thus invalidating the cached SI μPrograms).

5.6.1 Cache Organization

The SI μProgram cache is a software-managed cache with the cached μPrograms stored in off-chip DRAM memory. If a cache hit occurs, the μProgram is transferred from DRAM memory to the on-chip SRAM-based SI μProgram memory. DRAM memory is used because SRAM memory is expensive (in terms of area, static power), and the SI μProgram memory is not large enough to hold more than a few μPrograms at any given time.

The organization of the cache is shown in Figure 5.7. A cache entry consists of the tag and the variable sized data block.

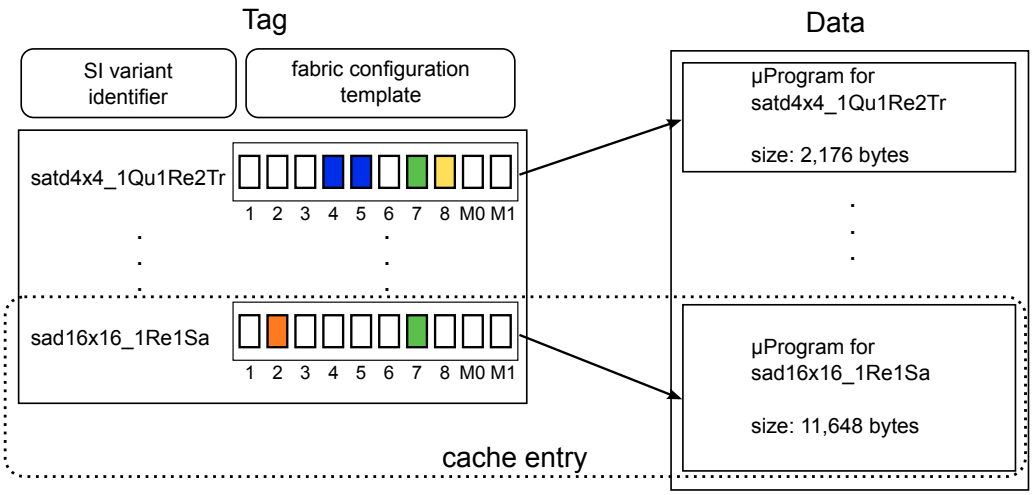


Figure 5.7: Cache for SI μPrograms.

The tag consist of the SI variant identifier and the *Fabric Configuration Template* (FCT), which generalizes the fabric configuration that was used for binding the SI variant. Instead of using the full fabric configuration (which consists of every RAC on the fabric), the FCT consists of only those RACs that are actually used by the μProgram (colored RACs in Figure 5.7). Multiple different fabric configurations can therefore match the same cache entry, as long as they only differ in the RACs unused by the SI. Compared to the tag of a regular CPU-Cache, the RACs of the FCT are similar to the MSBs of the tag, and the RACs that are not used by the μProgram are similar to the LSBs of the CPU-Cache tag.

The SI μProgram is stored in the cache data block. SI μPrograms consist of μOps, which in a CPU-cache would be "words", although μOps are far bigger than a regular 32-bit or 64-bit word (1024 bits per μOp in the *i*-Core reconfigurable processor). Unlike a CPU-cache, it makes no sense to evict single words, as binding is an iterative process, i.e. binding a control step requires that the previous control steps have already been bound. Evicting a single μOp would require binding of the whole associated μProgram. Therefore, μPrograms can only be evicted from the cache as a whole.

The cache is queried before an SI variant s is bound to a fabric configuration F. Unless a cache entry exists where the SI variant identifier matches s and the FCT matches F, a cache miss occurs and the SI variant is bound as described in Section 5.5. Otherwise, a cache hit occurs and the μProgram needs to be transferred to the SI μProgram memory before the SI variant can be run on the fabric[3]. Transfer time depends on the number μOp in the μProgram. As shown later in Table 6.2 (Section 6.2), the average SI latency (and thus number of μOps) is 19. Based on prototype measurements, it takes approximately 690 cycles to transfer SI μProgram

[3]Copying of the μProgram is done via DMA transfer, i.e. the processor is not stalled during this operation, similar to fabric reconfiguration.

with 19 μOps, assuming a bus load of 50%. This is a very small overhead, compared to the time to bind such an SI (multiple 100,000 cycles).

If a cache miss occurred, the newly generated μProgram is inserted into the cache, potentially evicting already present μPrograms, if the cache is full. Different cache replacement strategies and cache sizes are evaluated in Section 6.5.3.

In addition to regular operations such as insertion, eviction, and retrieval, the cache also supports a special operation which is required for Cache-Aware Placement (next section): retrieving all FCTs for a SI variant identifier s.

5.6.2 Cache-Aware Placement

Even if the μProgram of an SI variant s is cached, a future fabric configuration may have the accelerators required for s loaded in different RACs than required for the cached μProgram, resulting in a cache miss. The fabric configuration is determined by the placement algorithm, as discussed in Section 5.4. As the Cluster and Connectivity placement algorithms are cache-agnostic, a placement of accelerators for s may differ from the previous placement (e.g. due to s being used in a different kernel with other SIs).

To improve the hit rate in the μProgram cache and thereby reduce binding overhead, the cache-aware placement (CAP) algorithm has been developed. CAP aims to place accelerators for the SI variants of the current kernel in such a way, that the μProgram cache hit rate is maximized, thus removing the need to bind SI variants unnecessarily. Unlike Cluster and Connectivity placement, CAP is not run before every reconfiguration (as in Figure 2.10), but only once at each kernel prefetch (directly after the *Scheduling* step in Figure 2.10), pre-computing the RACs for as many accelerators as possible. Before a reconfiguration is performed, a simple lookup is made to check if a RAC has been pre-computed for the accelerator to be reconfigured. If so, the accelerator is reconfigured into this pre-computed RAC. In case no μProgram for a SI variant was cached, CAP can not pre-compute the RAC for an accelerator, and before reconfiguring this accelerator the RAC location is computed by using Connectivity placement (Section 5.4).

The idea of CAP is to merge the fabric configurations of those SI variants that are available in the cache into one pre-computed configuration M. The pre-computed configuration will ensure later cache hits for all SI variants that contributed to M (Figure 5.8). The algorithm is shown in Figure 5.9 and consists of 2 parts: Locking and Configuration Pre-Computation. The input of the algorithm are the SI variants that will be used in the next kernel. Just as with Connectivity and Cluster placement, CAP does not perform any reconfiguration, but only computes the RAC locations for the accelerators that will be loaded.

The purpose of Locking is to ensure that those SI variants for the next kernel that are already ready-to-run on the fabric are not hindered by the pre-computed configuration. The algorithm considers all those SI variants that were selected by the run-time

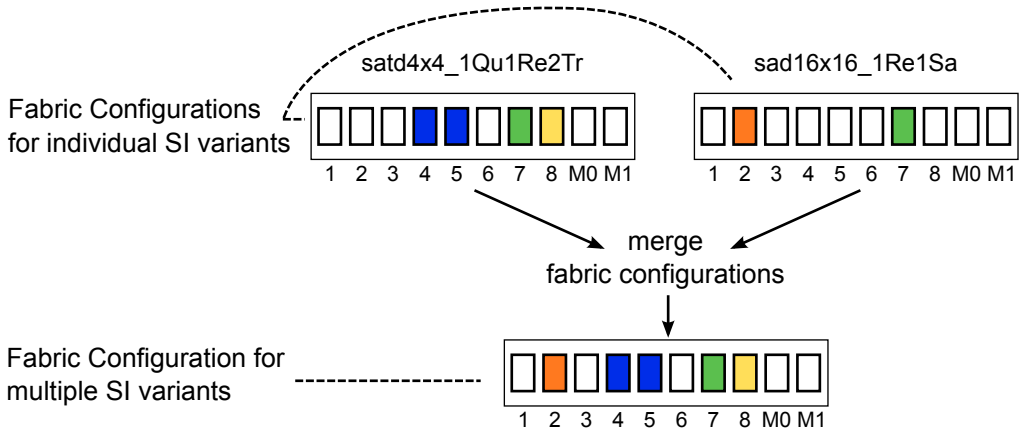


Figure 5.8: Merging of fabric configurations during placement for improved μProgram cache hitrates.

system for the next kernel. If multiple SI variants were selected for one SI, effectively only the "largest" variant (i.e. the one requiring the most RACs) will be locked (if possible). This is due to the fact that in general, for two variants s_1 and s_2 of one SI, the accelerators of s_1 will also be used by s_2 (or vice versa). These SI variants may have been configured during a previous kernel and will be reused in the next kernel. For each SI variant for the next kernel, CAP examines if the SI variant has both a μProgram available and the required accelerators loaded (i.e. "ready-to-run"). If so, the fabric configuration required for this SI variant is merged into a Locked Configuration LC. At the end of Locking, LC is the union of all ready-to-run SI variants. The RACs in LC will not be used during pre-computation.

Additionally, there may be SI variants for which all accelerators are reconfigured on the fabric, but there is no μProgram available. This may be because either already loaded accelerators from other SIs can be re-used for a new SI, or the SI variant was previously ready-to-run, but at some later time its μProgram had to be removed from SI μProgram memory to make space for new μPrograms. To ensure that pre-computation does not replace any of the already available accelerators, these accelerators are added to a set of Locked Accelerators LA. Pre-computation may not change the accelerator of any RAC which has an accelerator from LA.

Configuration Pre-Computation computes the fabric configuration M that will improve cache hitrate when requesting μPrograms. First, M is initialized as an empty configuration and the RACs from LC are marked as "unavailable" (grey in Figure 5.9). CAP then attempts to merge the fabric configuration of each SI variant s into M. To do so, all FCTs for s are retrieved from the cache. Each FCT F is then tested if merging F into M would lead to a conflict. A conflict occurs if either of the following is true:

1. F requires reconfiguration of a RAC that is also present in F (this includes the RACs from LC, which are marked as "unavailable"),

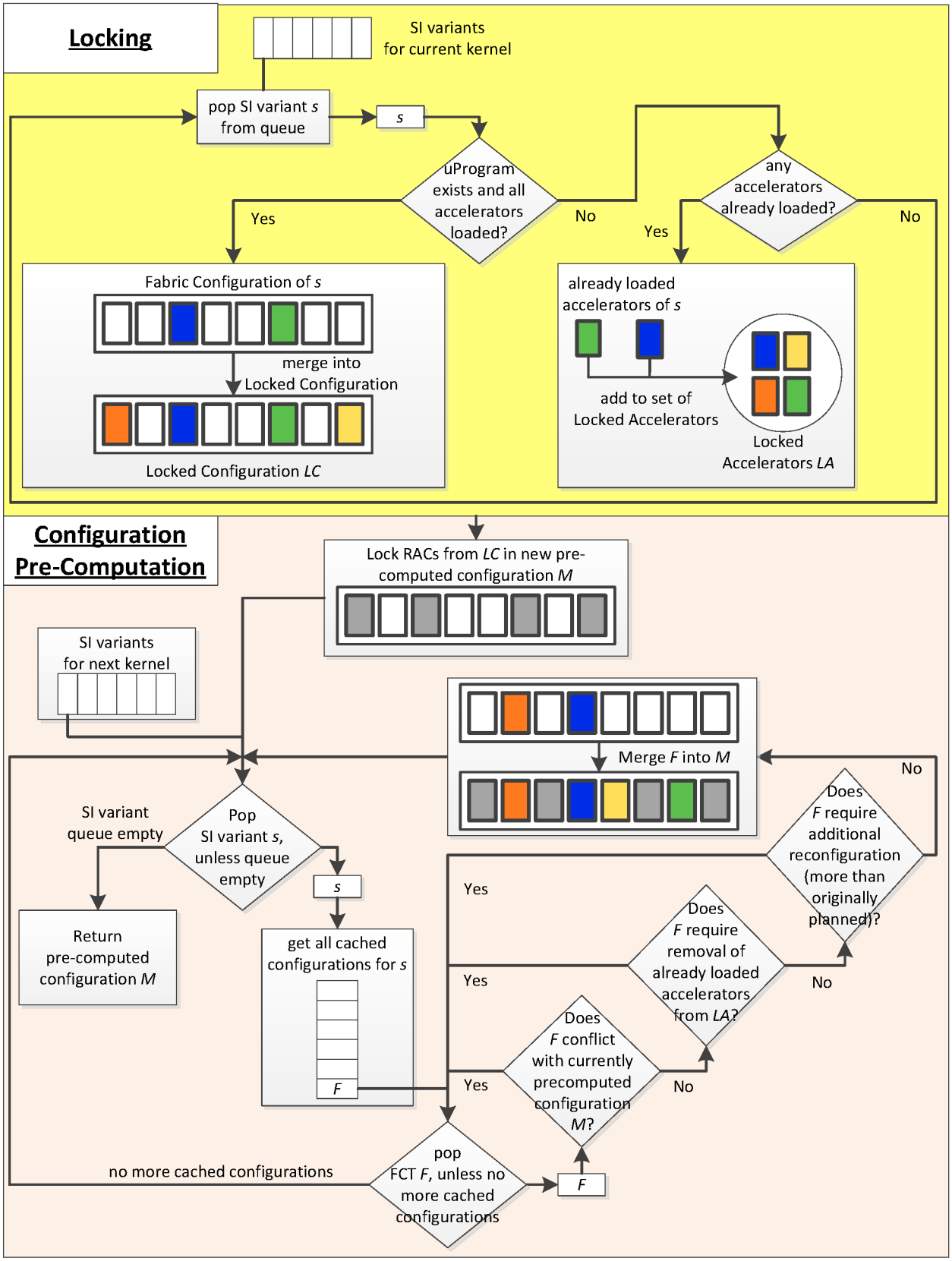


Figure 5.9: Cache-Aware Placement algorithm.

2. F requires reconfiguration of a RAC that currently has an accelerator from LA loaded,
3. F requires reconfiguration of additional accelerators than originally planned (e.g. different fabric configurations require the same accelerator but at different locations)

If a conflict occurs, the next FCT for s is examined. Otherwise, the FCT is merged with M and the accelerators required for the cached entry of s will be part of the pre-computed configuration. The algorithm continues with the next SI variant. After all SI variants for the next kernel have been processed, the algorithm returns M, which containing the pre-computed fabric configuration for the next kernel. For those accelerators of the SI variants that are not part of M (due to conflicts), Connectivity Placement is used as fall-back.

5.7 Summary

The increased system performance achieved when using a reconfigurable processor in multi-tasking scenarios and in multi-core systems (as presented in Chapters 3 and 4) requires the allocating parts of the fabric to different tasks. For dynamic workloads (such as those encountered in typical desktop and mobile systems) fabric allocation is not known at compile-time and thus must be performed at run-time. This requires that the SIs that are run on the fabric are flexible enough to support any fabric share allocated to a task at run-time.

To support this degree of flexibility, the proposed solution is to perform parts of SI μProgram generation (which is required for SI execution on the fabric) at compile-time and the remaining steps of μProgram generation at run-time. The resulting approach is called *partial online synthesis*. Those parts of SI μProgram generation that do not depend on where on the fabric an accelerator is loaded (information which is only available at run-time) are done at compile-time. This includes synthesis and place & routing of accelerator data paths and scheduling of the data-flow graph of the SI. At run-time, the placement of accelerators onto the fabric and the binding of the scheduled data-flow graph to a fabric configuration is performed. The algorithms designed for this have the goal of minimizing performance impact due to fabric architectural details, resulting in low SI latency and thereby high system performance.

Implementation and measurements of placement and binding algorithms on a FPGA-based prototype has shown that the binding step still takes a significant amount of time, allowing for further optimization and performance improvement. A software-cache for SI μPrograms is introduced, and an accelerator placement algorithm is designed that improves hitrate when retrieving μPrograms. The μProgram cache and corresponding placement algorithm maintain the flexibility provided by placement and binding at run-time, while lowering their overhead by reducing the number of times a μProgram has to be bound.

With the partial online synthesis approach presented in this chapter, the techniques for multi-tasking and multi-core support in reconfigurable processors (proposed in Chapters 3 and 4) are fully supported.

6 Evaluation

The contributions of this thesis (Chapters 3 to 5) are evaluated in this chapter using the *i*-Core reconfigurable processor (see Section 2.6). To allow faster design-space exploration and comparison with state-of-the art approaches, the results are obtained using simulations of an architecture model of the *i*-Core and the contributions. However, in the scope of this thesis, a prototype of the *i*-Core was developed as well, which is used to parametrize the architecture model. The following two sections describe the prototype (and measurements obtained on it) and the experimental setup of the architecture model, as well as the applications used for evaluation.

Then, the proposed approaches for multi-tasking in reconfigurable processors, sharing the reconfigurable fabric in multi-cores and the partial online synthesis of μPrograms for running SIs on the fabric are evaluated in detail in Sections 6.3 to 6.5. Section 6.6 shows that all of these approaches can be used together in a reconfigurable multi-core system.

6.1 Prototype

The *i*-Core was implemented as an FPGA-prototype and integrated both into multi-core systems and a many-core system (Invasive Computing many-core, see Section 2.6.4).

Table 6.1 shows the FPGA resource utilization for a dual-core system consisting of an *i*-Core and an unmodified LEON-3, when implemented for the Virtex-5 LX110T FPGA. LUTs are look-up tables (mainly used to implement combinational logic) and BlockRAMs are dedicated memory blocks (used for e.g. register files, on-chip SRAM memory). The fabric requires a significant amount of LUT resources due to the flexibility provided by the fabric links and link connectors. As shown in

	LUTs	18kb BlockRAMs
GPP core (unmodified LEON-3)	4740	17
i-Core GPP core (modified LEON-3)	6478	24
Fabric (5 RACs)	13544	0
SI μProgram memory	0	14
SI Execution Controller	189	0

Table 6.1: Prototype implementation details of the *i*-Core reconfigurable processor on an Virtex-5 LX110T.

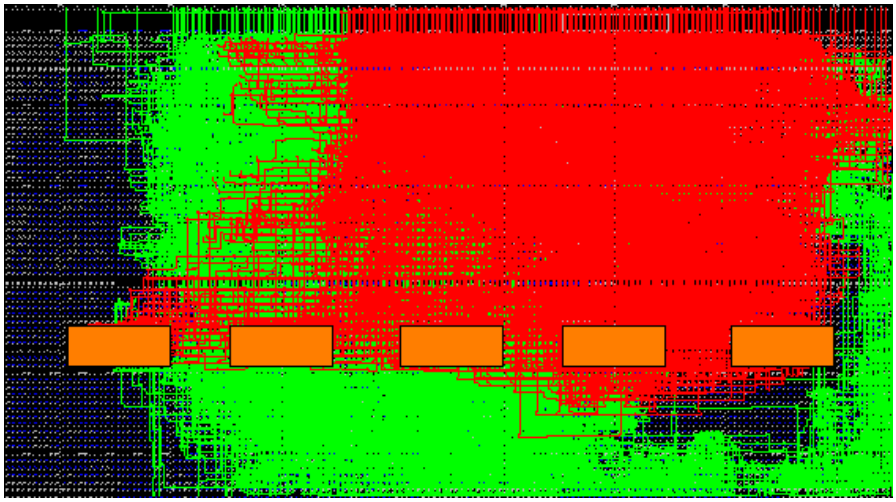


Figure 6.1: Floorplan of the *i*-Core on the Virtex-5 LX110T FPGA with routing resources highlighted. Red: Fabric and Interconnect, Green: GPP core, Orange: RACs.

Figure 2.21, for a fabric with 8 uni-directional fabric links, each link connector has 8 4:1 multiplexers (MUXes) for the outgoing link lines, 2 12:1 MUXes for the RAC inputs and 2 2:1 MUXes for the pRF inputs. As link connectors are used not just for RACs, but also for each memory port and non-reconfigurable module (which are used for operations such as byte rearrangement in a word or simple arithmetic), a large number of multiplexers are required. The large area overhead of the fabric due to MUXes can be expected to be reduced for a chip-tapeout, as link connectors would be implemented as an ASIC (see Figure 2.20), and [EL09] observes that "Multiplexers are expensive in FPGAs and cheap in ASICs". The increased amount of BlockRAMs required for the *i*-Core GPP core (compared to an unmodified LEON-3) is due to a 5-port register file (4 read, 1 write) used in the *i*-Core instead of a regular 3-port register file (2 read, 1 write) used in a LEON-3[1]. Increased LUT requirements in the *i*-Core GPP core are due to the extensions in several pipeline stages in order to support Special Instructions (see Section 2.6.1).

Figure 6.1 shows the floorplan of the synthesized design (generated in Xilinx FPGA Editor), with the routing resources and the RACs highlighted. The achieved system frequency is 50 MHz, with RAC dimensions of 20×5 CLBs and the time to reconfigure one RAC measured at 274μsec (using a bare-metal C program, thus no additional bus-load was present).

When used as a processing element in the heterogeneous Invasive Computing many-core system, the *i*-Core is used to accelerate application parts exhibiting a high degree of instruction-level parallelism, which are implemented as *i*-Core SIs. For example, [PSOE+15] presents the results of using the *i*-Core as part of an object recognition application in a robotic vision scenario. There, the object features recognized are

[1]The register file has been extended, as often SIs require more than 2 inputs (e.g. input data address, input data size, output data address.

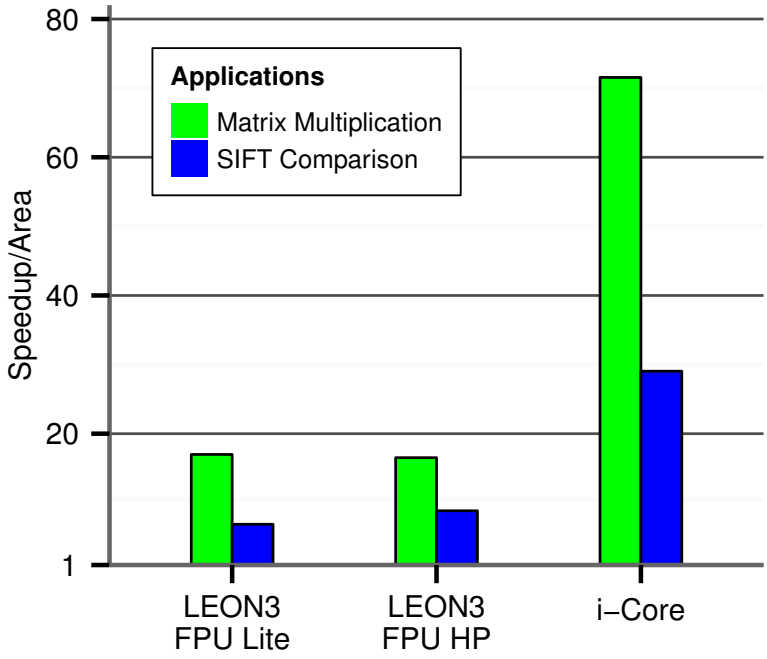


Figure 6.2: Matrix Multiplication and SIFT kernels on the *i*-Core. Comparison of area efficiency of *i*-Core SIs to LEON-3 with two different Floating-Point Unit (FPU) implementations (HP – High Performance).

identified using the "Harris Corner Detection" and "SIFT" (Scale Invariant Feature Transform) algorithms. Using an SI with specialized floating-point accelerators, the *i*-Core accelerates the matching of these features to known features stored in a database. Furthermore, the *i*-Core is also used to speed up applications from the high-performance computing domain. Here, the *i*-Core accelerates multiplication of 32x32 matrices, which are "atomic" operations in an algorithm for multiplication of large matrices. In the Invasive Computing system an application has the choice of using the *i*-Core or other computational elements (such as GPPs), thus area efficiency (i.e. given the same area, which computational resource achieves better performance?) is an important metric. Figure 6.2 shows the area efficiency for SIFT matching and matrix multiplication when performed on the *i*-Core or GPP cores with two different floating-point unit (FPUs) implementations. Here, the area of the *i*-Core is the sum of LUTs required for the modified GPP core, the fabric, and the *i*-Core support hardware. Even with the significant area overhead of the reconfigurable fabric, the *i*-Core provides better efficiency, as multiple accelerators can be used to exploit the parallelism in both matrix-multiplication and SIFT matching kernels.

Figure 6.3 shows a demonstrator setup of a small (2 tiles, 2 cores per tile) Invasive Computing design, where one of the cores is an *i*-Core. The accelerated application is an H.264 Video Encoder modified to immediately re-decode an encoded frame and output it via VGA to the screen in order to provide visual feedback of the encoding process. The application is running on top of the Invasive Computing operation system OctoPOS and an agent-based resource management system.

Figure 6.3: Demonstration setup of the i-Core as part of a small (2 tiles) Invasive Computing architecture configuration. The prototype is implemented on a Virtex-6 FPGA (ML605 board on the lower right) and is connected to a custom interface to allow interaction with each of the 5 RACs. The demo setup is running a modified H.264 Video Encoder.

6.2 Experimental Setup

The results presented in the remainder of this chapter were obtained from cycle-accurate simulations of a reconfigurable processor (in both single- and multi-core setups). The architecture model used in the simulations is a SystemC description of the *i*-Core with the additional extensions required to realize the contributions proposed in this thesis[2].

The applications used for benchmarking were an H.264 Video Encoder and several applications from the MiBench[GREA+01] and MediaBench[LPM97] benchmark suites. These applications cover the following domains: Image Decoding (JPEG), Image Processing (SUSAN corner/edge detection), Audio Encoding (AdPCM), Error Correction Code (CRC), Cryptographic Hashing (SHA) and Encryption (AES). The characteristics of these applications are detailed in Table 6.2. SHA, AdPCM and AES each have one kernel (although of different complexity) that is accelerated by one SI each. The H.264 Video Encoder has a very dynamic execution behavior: to encode a frame, three different kernels (implemented by several executions of altogether 9 different SIs) are used, with the SIs of the second kernel ("Encoding Engine") being highly dependent on the type of frame (input data dependency). JPEG and SUSAN also have multiple kernels, although they exhibit less dynamic behavior than H.264. Table 6.2 also shows the *inter-SI gap* characteristic for each application, which is defined as the time spent running non-SI code between two subsequent SI executions. The ratio of inter-SI gap to the SI Latency is of interest for the multi-core evaluation in Section 6.4. If the ratio is large enough, then fabric accesses by different cores can be serialized efficiently, thus reducing the benefit provided by merging fabric access (as done by COREFAB). For AdPCM, H.264 and SHA the inter-SI gaps are small, i.e. the system spends little time running non-SI code between two subsequent SI executions compared to AES, JPEG and SUSAN. The reason for the large inter-SI gaps is usually that the output of an SI requires control-flow intensive post-processing, which is performed on the GPP.

Figure 6.4 shows the speedup for these applications when run on reconfigurable fabrics of different sizes. While all applications benefit from allocated fabric, speedup saturates at some point. For applications with few (or simpler) SIs, few accelerators are required and this saturation point is reached with 2 or 3 RACs (e.g. AdPCM, AES), while applications with multiple (and/or more complex) SIs require more accelerators and benefit from additional RACs (e.g. H.264 Encoder, SUSAN, JPEG).

For evaluation of the deadline-aware task scheduler from Section 3.4 the applications were modified in order to model tasks with periodic jobs. To do that, a task *yields* after processing a fixed amount of input data, which suspends the task until its next deadline, at which point is is released again. A job of a task is then the

[2]The SystemC model assumes a LEON-2 GPP core instead of a LEON-3 core, as the LEON-2 based model has been calibrated against register-transfer level simulations and is therefore very accurate. The LEON-2 pipeline has 5 stages instead of 7 (for the LEON-3), however, the throughput of both processors is the same.

	AdPCM	H.264	SHA	AES	JPEG	SUSAN
Application time spent on fabric [%]	28.3	47.0	35.2	11.6	3.6	7.9
Number of SIs	1	9	1	1	4	3
Number of different accelerator types	2	10	1	2	5	7
SI Latency on fabric (mean) [cycles]	7.0	17.5	6.0	10.0	21.1	52.0
inter-SI gap (mean) [cycles]	17.7	19.7	11.0	76.3	152.8	583.5
Application type	Audio	Video	Crypto	Crypto	Image	Image

Table 6.2: Characteristics of applications used in evaluation. Data for is a reconfigurable processor with 10 RACs.

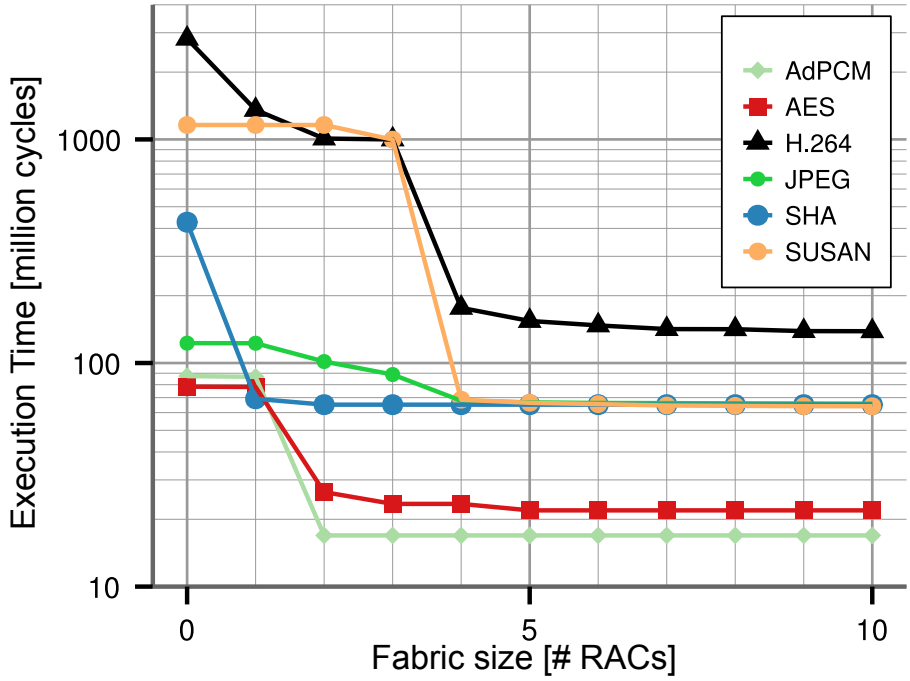


Figure 6.4: Application performance when run as single tasks on a reconfigurable processor. Speedup depends on fabric size and saturates for larger fabrics.

application part between two consecutive yields. For the other evaluations these modified applications were not used.

The tasksets that were used for evaluation were composed of these applications and will be shown in the following sections together with the evaluation of each contribution.

Taskset 1		Taskset 2		Taskset 3		Taskset 4		Taskset 5	
App.	DL	App.	DL	App.	DL	App.	DL	App.	DL
SHA	-	SHA	-	SHA	-	AES	40	H.264	40
H.264	40	H.264	40	AES	8	SUSAN	60	Jpeg	20
H.264	40	Jpeg	20	AdPCM	10	H.264	30		

Table 6.3: Applications and deadlines (in ms) used in the tasksets for evaluation of schedulers with the goal of tardiness reduction.

6.3 Multi-Tasking

6.3.1 Tardiness Reduction

The tasksets used for scheduler evaluation with the goal of tardiness (accumulated time by which all deadlines were exceeded) reduction are shown in Table 6.3. Up to 3 tasks are used per taskset to model the limited degree of parallelism in typical desktop/mobile workloads (see Chapter 1). As stated in Section 6.2, for this evaluation all applications (except for SHA) are broken up into jobs and annotated with a deadline in the tasksets. SHA is an exception, which is used as an aperiodic task (to model a background/maintenance task) without a deadline and can therefore be scheduled in a "best-effort" fashion.

In order to analyze only the effect of scheduling on tardiness, all schedulers use the same fabric allocation strategy. Applications are assigned a static priority (according to the application performance characteristics from Figure 6.4), and assigned RACs proportionally to their priority. Allocation is performed when a task (but not a job within a task) starts or finishes. For this evaluation, SHA does not use any SIs and thus is not allocated any RACs. Each taskset was scheduled on all fabric sizes between 5 and 15 RACs.

The proposed Performance-Aware Task Scheduler (PATS) from Section 3.4 is compared to Rate-Monotonic (RMS), Earliest Deadline First (EDF) and Round-Robin (RR) schedulers. Round Robin is not deadline-aware and is simply used here for comparison, as it also exhibits the effect of reconfiguration hiding (see Section 3.1) to a certain degree. As discussed in Section 2.5, no specialized schedulers (which is the novelty of PATS) are used in state-of-the-art reconfigurable processors with a high degree of flexibility in their SI approach (i.e. multiple variants for each SI, allowing the processor to adapt to dynamic workloads instead of the "all-or-nothing" approach, which provides only one implementation per SI). To obtain the tardiness value, a scheduler runs a taskset for 100 million cycles and monitors the tardiness for each job of each running task. Jobs of a task that can not start in time (i.e. before the next deadline) are not discarded, but instead are started late and therefore contribute to overall tardiness.

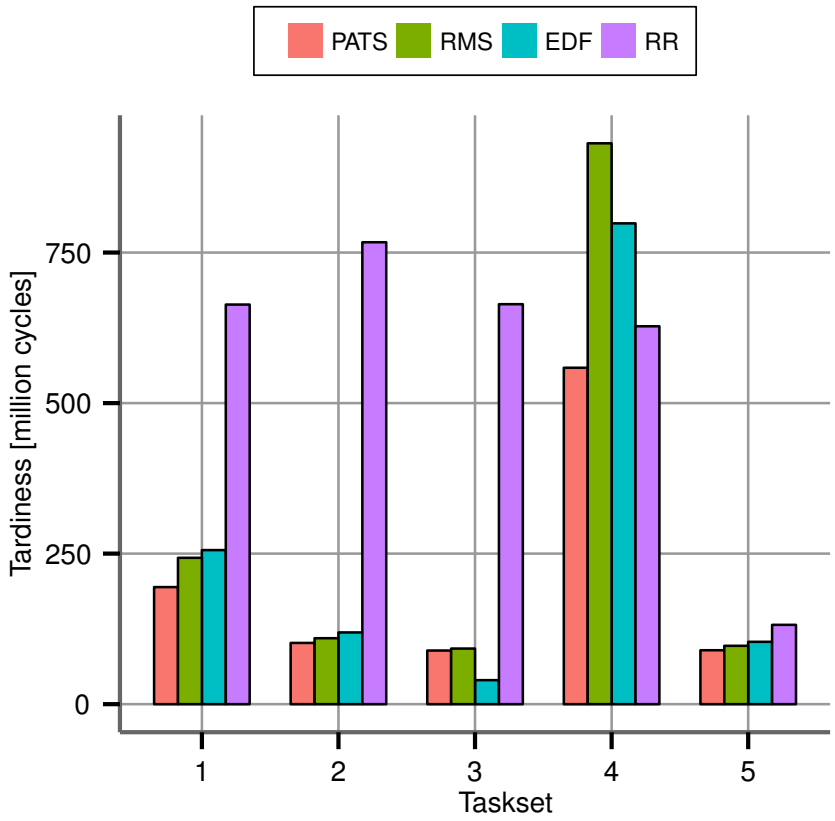


Figure 6.5: Tardiness comparison of PATS (contribution), Earliest Deadline First, Rate-Monotonic and Round Robin Schedulers. The tardiness value is relative to the worst scheduler for a particular taskset.

Over all tasksets, the schedules produced by PATS result in the lowest tardiness, followed by EDF (on average 1.22× worse than PATS), RMS (1.30× worse than PATS) and RR (2.5× worse than PATS). Figure 6.5 shows the relative tardiness of the evaluated schedulers for each taskset. Relative tardiness is computed by using the scheduler that produced the worst tardiness for a taskset as a baseline of 1.0, and then computing the relative tardiness values of the other schedulers based on this baseline. This allows performance comparison across different tasksets (which can have a large difference in absolute tardiness values, depending on task composition and deadlines).

PATS performs best in tasksets where a large degree of reconfiguration hiding (or RiCL reduction) can be performed. Such tasksets contain applications that perform periodic reconfigurations while they are running, e.g. due to switching kernels. H.264 and SUSAN are such tasks, which are both present in tasksets 1 and 4, where PATS provides a 1.25× and 1.12× tardiness improvement over the next best scheduler. On the other hand, in taskset 3 both AES and ApPCM load their accelerators immediately after starting, and do not perform any further reconfigurations. As both tasks reach (and keep) a task efficiency of 1.0 shortly after start, PATS behaves like a regular scheduler (RMS or EDF, depending on how overloaded the system is, see

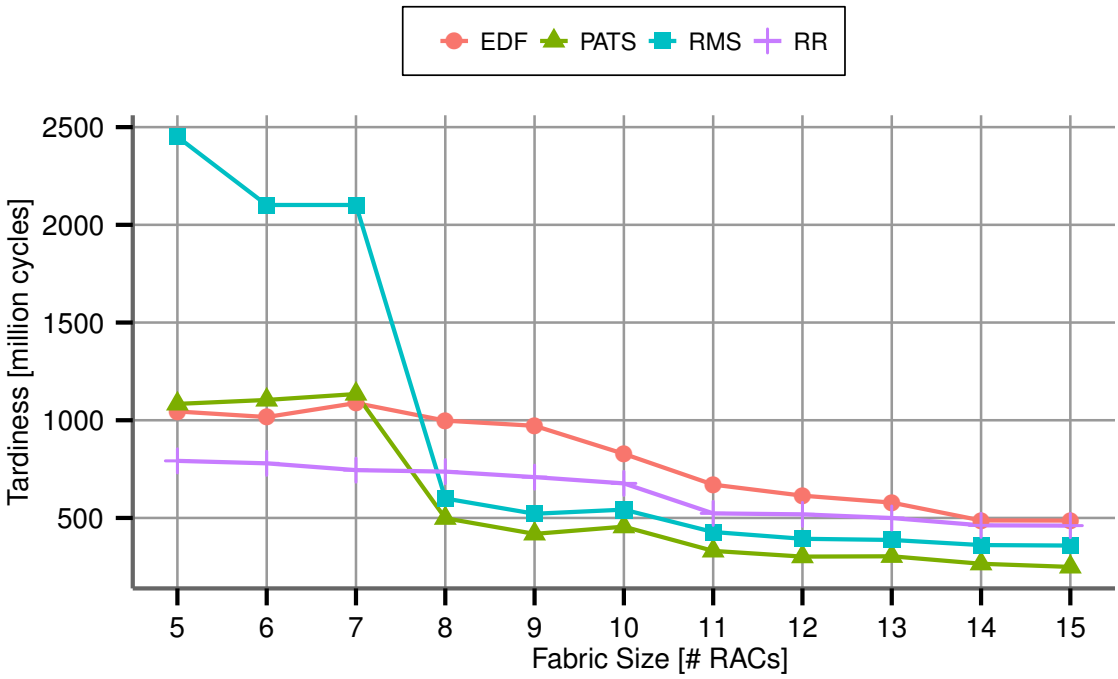


Figure 6.6: Tardiness (absolute value) for different fabric sizes (here shown for Taskset 4). Performance improves with increasing fabric size, resulting in improved tardiness as deadlines are met better.

Section 3.4) and provides no performance benefit.

Apart from the taskset, the size of the reconfigurable fabric has an effect on tardiness. Figure 6.6 shows a detailed plot for the tardiness of Taskset 4 when scheduled for different fabric sizes. For small fabric, few accelerators can be loaded by an application and deadlines are harder to meet, resulting in high tardiness. As fabric size is increased, application performance also increases, resulting in fewer deadline misses and better tardiness.

6.3.2 Makespan Improvement

Table 6.4 shows the tasksets used for evaluation of task scheduling with the goal of reducing the makespan. Tasks are characterized by their kernel execution behavior, as described in Section 3.3: multi-kernel tasks (MKT), single-kernel tasks (SKT) and zero-kernel tasks (ZKT). The version of the SHA application used for the benchmarks in this section does not use SIs (and thus does not use reconfigurable fabric) in order to provide the system with a software-only ZKT. This version of SHA can be substituted by any other application without SIs.

The MORP scheduler uses performance curves for the applications, based on the single-task application results from Figure 6.4. The curves use normalized perfor-

	Taskset 1	Taskset 2	Taskset 3
MKT	H.264 SUSAN	H.264 SUSAN	H.264
SKT		AdPCM Rijndael	
ZKT		SHA	SHA SHA SHA

Table 6.4: Tasksets used for evaluation of schedulers aimed at makespan improvement.

mance (in the range of [0..1]) for each fabric size, based on the execution times from the figure.

MORP is compared to three other task scheduling policies: First Come First Serve (FCFS), Round Robin (RR, used in the reconfigurable processor Molen[SSB09]), MORP and Optimal. As the tasks have no precedence constraints and are assumed to be released simultaneously, the makespan on a non-reconfigurable single-core processor is simply the sum of the completion times of each task in the task set. For that case, FCFS would provide the best schedule. For reconfigurable processors, "Optimal" is defined as FCFS scheduling with a RiCL of 0 and no overhead apart from context switching time. This approach is modeled by configuring the simulator to assume zero-overhead reconfiguration time. However, makespans produced by the "Optimal" scheduler are not achievable in general and should be only regarded as a lower bound.

In addition to scheduling, the allocation of RACs to tasks needs to be determined. FCFS runs a task to completion, therefore during the time a task runs, all RACs are assigned to this task. This policy is not beneficial for RR, as it switches frequently between different tasks and would have to reconfigure potentially all RACs after each context-switch (multiple ms per context switch, which is an infeasible overhead). Instead, for RR it is beneficial to divide the reconfigurable fabric between the currently executing tasks, i.e. a particular task T_i has a certain share of the reconfigurable fabric where it can reconfigure, while another task T_j has its own share that it can reconfigure, but it can not overwrite the share of T_i. RR partitions the fabric equally among the tasks, without exceeding the saturation point at which the task no longer benefits from additional RACs, as illustrated in Figure 6.4. For example, AdPCM would not be provided with more than 2 RACs, as that would not improve its performance. MORP uses its own integrated fabric allocation technique, described in Section 3.5.

For accurate overhead analysis, a C implementation of the major MORP functions was fed with test input data and the execution time (in cycles) of each function was measured using the ArchC SPARC V8 instruction set simulator [ARBA+05]. The SystemC-based architecture simulator counts how often a particular function of MORP is executed. Together this allows to simulate the total overhead of using

MORP for each benchmark. Context switch duration was measured on the FPGA-based prototype running Linux with a hardware cycle counter (counter started when the kernel started a context switch and stopped when the context switch was completed). Over all simulations, FCFS performed 3.6, MORP 272.9 and RR 488.9 context switches per taskset on average. Context switch duration and MORP overhead is included in all of the following results.

Discussion

The makespan results of the benchmarks are shown in Figure 6.7. Figure 6.8 shows the results of as relative speed of MORP, Round Robin; and FCFS when compared to Optimal (which is always at 1.00). Makespans achieved by MORP are on average 2.8% slower than Optimal (median: 2.8% slower), whereas makespans of FCFS and Round Robin are 12.2% and 19.6% slower than Optimal (median: 13.8% and 19.6%), respectively. Scheduler performance varies with the taskset. MORP achieves its best results with Taskset 3, as does Round Robin, while FCFS favors Taskset 2. Over all fabric/taskset combinations, MORP yields better results than FCFS and Round Robin in 35 out of 45 benchmark combinations. In the cases where MORP does not yield the best makespan, its result is on average 1% worse than that of the best scheduler for this particular fabric/taskset combination. Most of the combinations where MORP did not achieve the best result were in Taskset 1, which is disfavorable to MORP, as explained further below. In the remaining configurations in Tasksets 2 and 3 where MORP does not yield the best makespan, RiCL is already very low, thus the RiCL reduction by MORP can not offset the makespan increase due to its overhead.

The following two non-scheduler related effects can be observed: (i) the more reconfigurable fabric is available for accelerators, the faster the taskset is completed – this is due to more SIs being executed in hardware, as more fabric is available. (ii) As the fabric size increases, the makespans of all schedulers converge. The reason is that once a large enough amount of RACs are available, a task will have all accelerators for all of its SIs loaded on the fabric and will no longer perform any reconfigurations. Once this fabric size is reached, RiCL is nearly 0 and the scheduler no longer has an effect on the makespan (apart from its overhead).

For very small fabric sizes (1–2 RACs) MORP and FCFS produce similar results, as reallocating from the already small fabric would slow down the primary task significantly, thus MORP behaves similar to FCFS. Round Robin generally yields a worse makespan for small fabric sizes. If multiple tasks benefit from hardware acceleration, then RR forces some of them to execute in software as not enough RACs are available for all of them[3].

On larger fabrics (8–12 RACs), MORP has more opportunities to reduce RiCL and thus improve the makespan. The potential for improvement also depends on the

[3]This behavior is not specific to the RR task scheduling policy, but rather the fabric allocation policy used for RR.

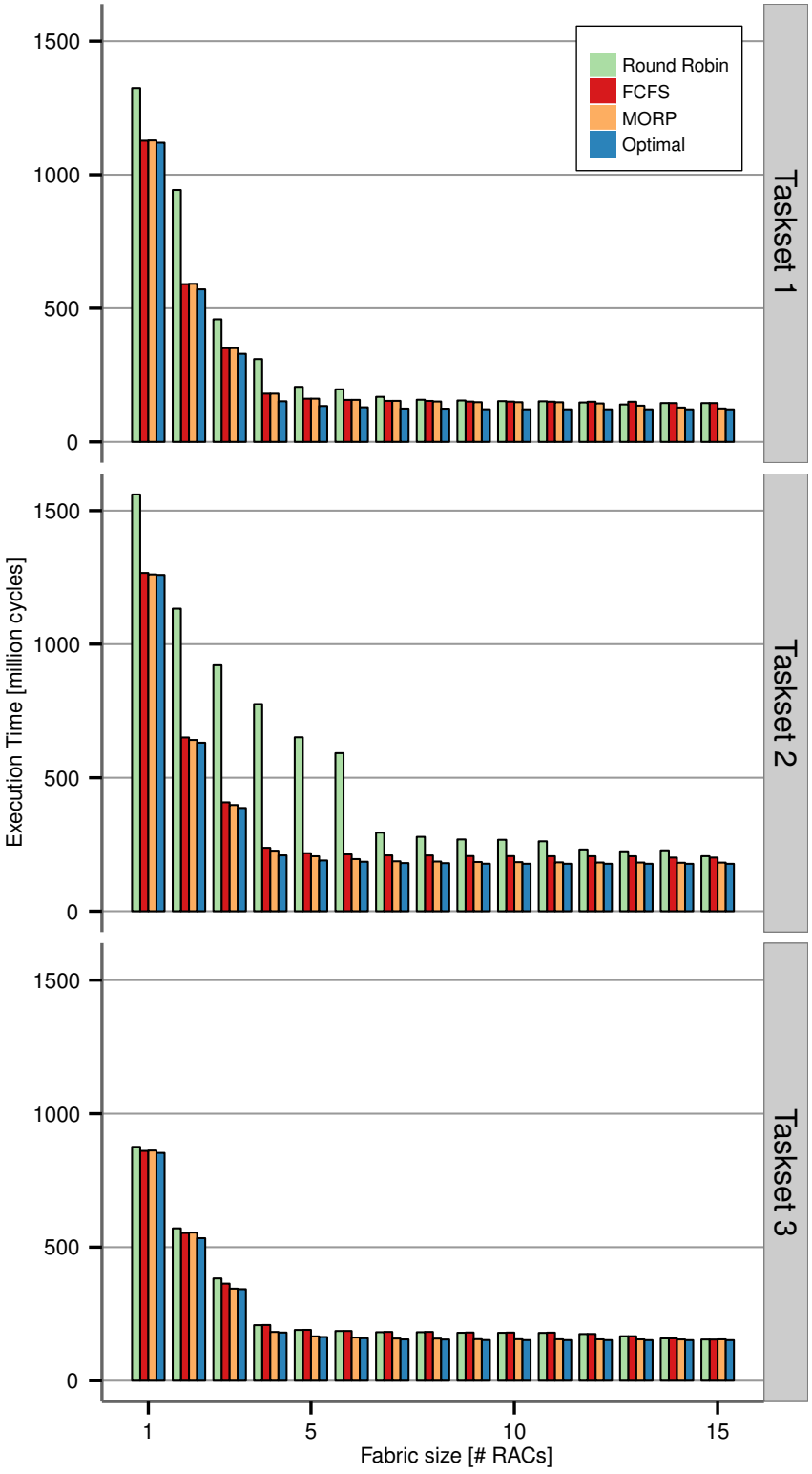


Figure 6.7: Makespan of 3 tasksets when scheduled by different schedulers.

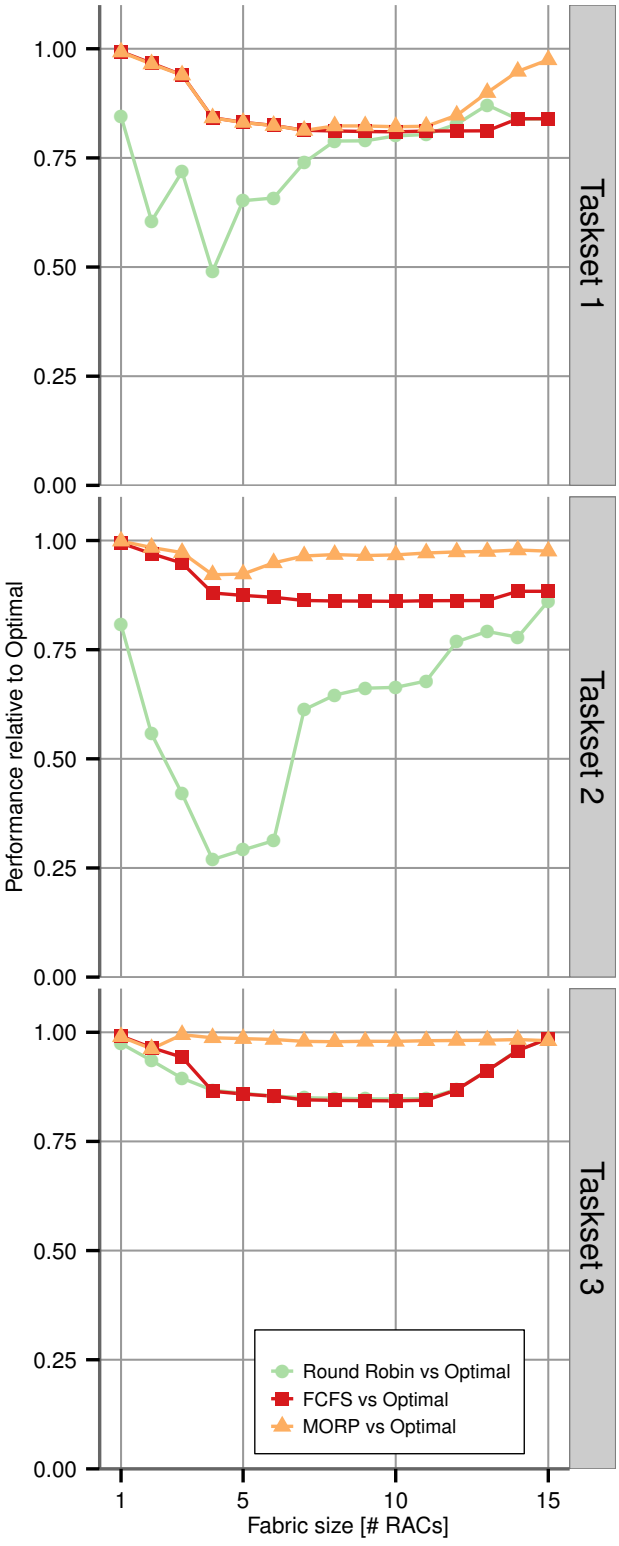


Figure 6.8: Relative speed of MORP, Round Robin and FCFS schedulers compared to Optimal.

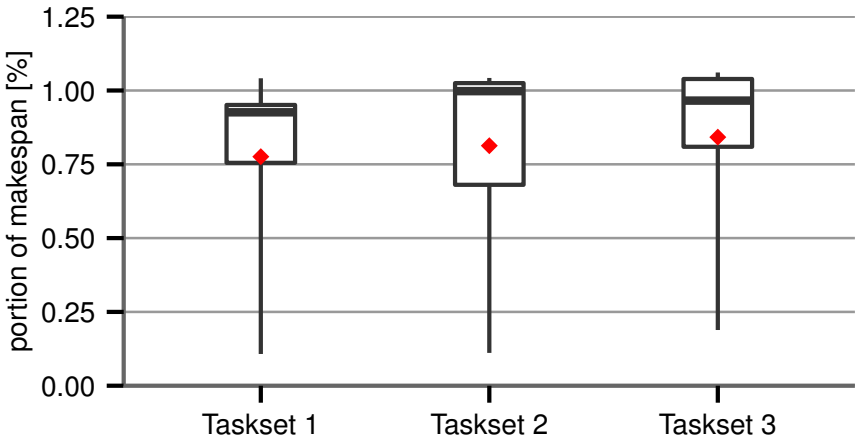


Figure 6.9: Overhead of MORP as share of the total makespan.

Scheduling Strategy	Transferred Configuration Data
Round Robin	24,022 kB
FCFS	25,800 kB
MORP	28,318 kB

Table 6.5: Average amount of reconfiguration data transferred.

set of executable tasks from which secondary tasks can be selected. Taskset 1 has two MKT tasks, which are both good candidates for primary tasks, but not very suitable for secondary tasks. They also have a rather high RiCL. As there is no suitable candidate for a secondary task, no reallocation is performed and MORP behaves comparable to FCFS until the taskset is finished, which is the reason why MORP does not always achieve the best result for Taskset 1. Taskset 2 offers better possibilities, as there are enough SKTs and ZKTs for hiding reconfigurations of both H.264 and SUSAN MKTs. MORP has the best performance when used on Taskset 3, as hiding reconfigurations with ZKTs incurs no performance loss on the primary task, as in this case no reallocation of RACs of the primary task is required. RR also does some amount of reconfiguration hiding (although not intentionally): when it switches to another task, the reconfigurations for the first task continue (if there were any remaining).

The overhead of MORP (already included in the benchmark measurements) is approximately 1% of the total makespan. Figure 6.9 shows boxplots for the overhead for each of the 3 tasksets. The execution time of the MORP functions mainly depends on the number of tasks in a taskset and the size of the reconfigurable fabric. Table 6.5 shows the amount of bytes transferred through the reconfiguration port per simulation run. MORP keeps the reconfiguration port busier than the other schedulers, as it deliberately performs reconfigurations when switching from a primary to a secondary task to reduce the total makespan.

Workload	Core 1 (reconf)	Core 2	Core 3	Core 4	Fabric Use
1	H.264	SHA	-	-	7
2	H.264	AdPCM	SHA	-	9
3	H.264	SHA	AES	JPEG	14
4	SUSAN	JPEG	ADPCM	-	10
5	SUSAN	H.264	-	-	9
6	AES	SHA	-	-	5

Table 6.6: Workloads used for multi-core evaluation.

6.4 Multi-Core

The workloads used for multi-core evaluation are shown in Table 6.6. The "Fabric Use" column shows the number of RACs that allow the workload to achieve near-maximum speedup (using data from offline profiling, see Figure 6.4). This number is not the maximum speedup, due to the diminishing returns of allocating more RACs to an application. For example, the speedup of H.264 saturates at 19 RACs, but the speedup improvement when increasing allocation from 5 to 19 RACs is only 19%. The number of RACs to achieve actual maximum speedup would be very high, and would not show how many RACs would be useful in a realistic scenario, where fabric area is scarce. The values from the table are *not* provided to COREFAB for guiding its operation (fabric allocation is described below), but instead they show what ratio of the fabric should be used to concurrently process a workload.

COREFAB is compared with the following techniques (based on the discussion in Section 4.2):

- *Reconf-Base* – a SoC with one reconfigurable core and three GPPs, without any possibility for sharing the fabric with the GPPs.
- *Dedicated Fabric* – the fabric is split into equally-sized shares (one share per application in the workload) that can be used concurrently, a technique also used in ReMAP ([WA10], called "Spatial Partitioning" in the paper). The number of dedicated fabric shares is assumed to be equal to the number of active cores in the workload, e.g. for Workload 1, the fabric is split into 2 shares, while for Workload 3 it is split into 4 shares. To provide true independent access to the scratchpad memory for each fabric share, each has two dedicated memory ports with a correspondingly reduced bandwidth. Thus for Workload 3, the fabric memory bandwidth is 1/4 compared to the other approaches, while for a Workload 1, the bandwidth is 1/2.
- *Shared Fabric* – the fabric can be accessed by all GPPs, but an SI may only execute if the fabric is not currently used by the reconfigurable core, i.e. fabric accesses are serialized, prioritizing accesses from the reconfigurable core. As this technique allows access by GPPs to the fabric, the Fabric Access Manager (FAM, see Section 4.5.1) module is required, incurring its protocol overhead of

1 cycle per Remote-SI execution. The serialization of fabric accesses is similar to the fabric access technique used in [CM11] (the Remote-SI latency overhead of 1 cycle is not present in [CM11], as their system is limited to sharing the fabric in a dual-core system).

For COREFAB and Shared Fabric, RACs are statically assigned to the cores as follows: Using the application performance characteristics from Figure 6.4, a branch-and-bound solver chose the allocation that resulted in the best average performance. COREFAB can be used with other fabric allocation techniques, such as those discussed in Section 2.3.1.

Discussion

The goal of COREFAB is to improve performance of a reconfigurable multi-core system by increasing GPP performance (i.e. by extending the benefits provided by a reconfigurable core to the non-reconfigurable cores in a multi-core system). Figure 6.10 shows the normalized per-core execution times relative to COREFAB for all approaches and workloads, and Table 6.7 shows the average execution time for the reconfigurable and GPP cores and for each strategy.

On the reconfigurable core, Shared Fabric has the same performance as COREFAB, because the SI merging technique of COREFAB benefits applications running on GPPs. On the GPPs, Shared Fabric has 30% worse performance than COREFAB on average, as it does not provide true concurrent execution. Dedicated Fabric has much worse performance on the reconfigurable core (more than 3× worse than COREFAB) as the memory bandwidth and the fabric area available to one core is only a fraction of the whole reconfigurable resources (e.g. 1/3 of the memory bandwidth and 1/3 of the RACs for each core in for Workload 2) and thus generally the reconfigurable core has a lower fabric share compared to the other approaches. However, the GPPs obtain larger fabric shares and may use them exclusively with independent memory ports, which leads to slightly better performance than COREFAB on the GPPs (2% on average). For workloads where the fabric shares of the GPPs are sufficiently large and the kernels are not memory-intensive (so that the reduced bandwidth has lower impact), Dedicated Fabric achieves even better performance on the GPPs (16% better than COREFAB for Workload 1). However, that does not compensate for the lower performance of the reconfigurable core.

Reconf-Base has equal or better performance on the reconfigurable core than the other methods (12% better than COREFAB on average), because the application running there has access to all RACs, while for the other strategies a part of the fabric is allocated to the other cores. However, Reconf-Base has drastically worse performance on the GPPs (more than 4× worse than COREFAB), as they can not offload their kernels onto the fabric, instead having to run them in software on the GPP pipelines.

The average accumulated execution time for each approach is shown in Figure 6.11. The accumulated execution time is the sum of the times that it took each core to

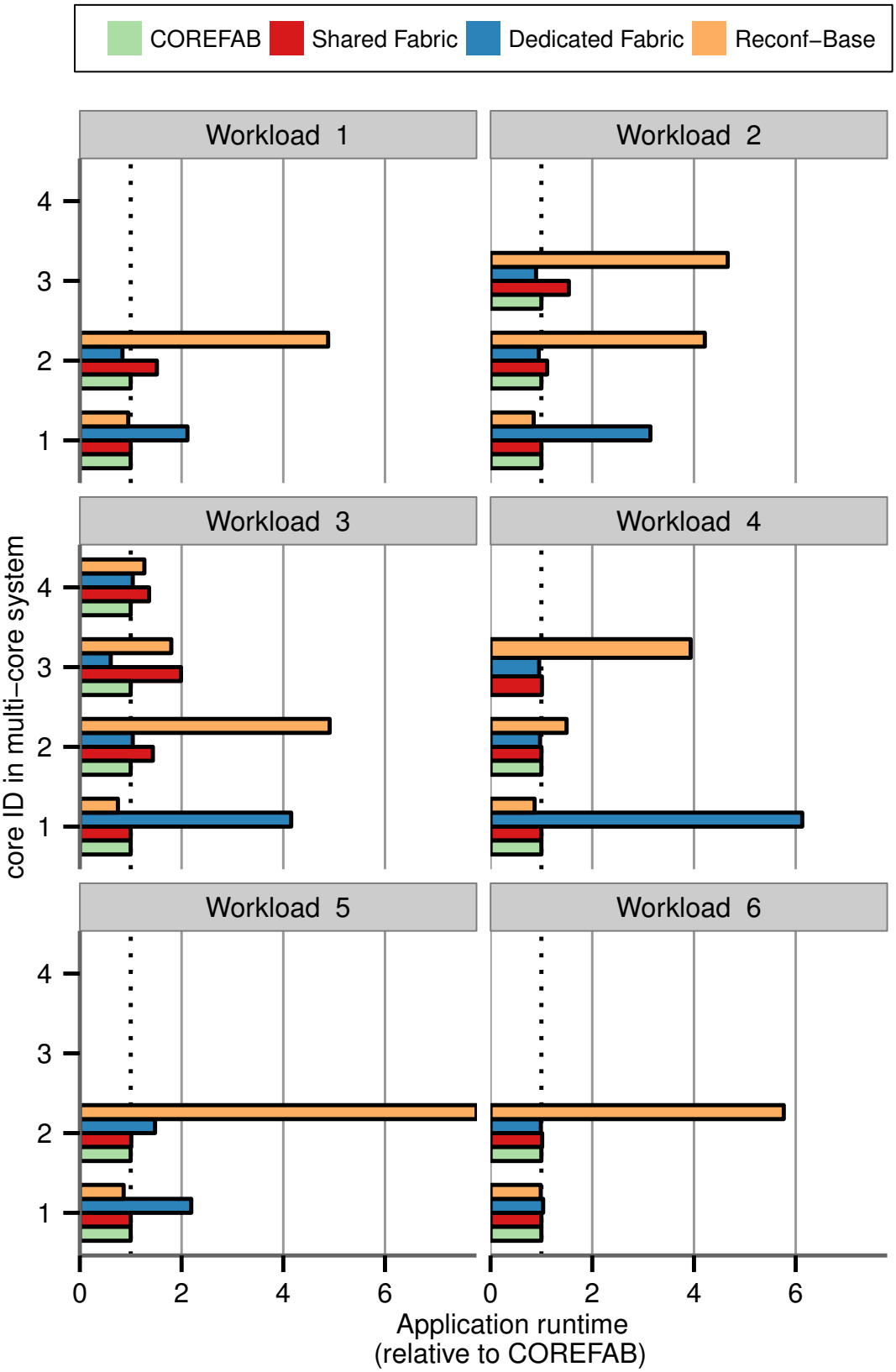


Figure 6.10: Normalized application execution time per-core (lower is better) of state-of-the-art fabric sharing strategies compared to COREFAB (dotted line at 1.0). Core ID 1 is the reconfigurable core, other cores are GPPs.

Core	Technique	Normalized Execution Time (lower is better)
Recon-figurable	COREFAB	1.00
	Shared Fabric	1.00
	Dedicated Fabric	3.13
	Reconf-Base	0.88
GPPs	COREFAB	1.00
	Shared Fabric	1.30
	Dedicated Fabric	0.98
	Reconf-Base	4.74

Table 6.7: Performance summary for different fabric sharing strategies.

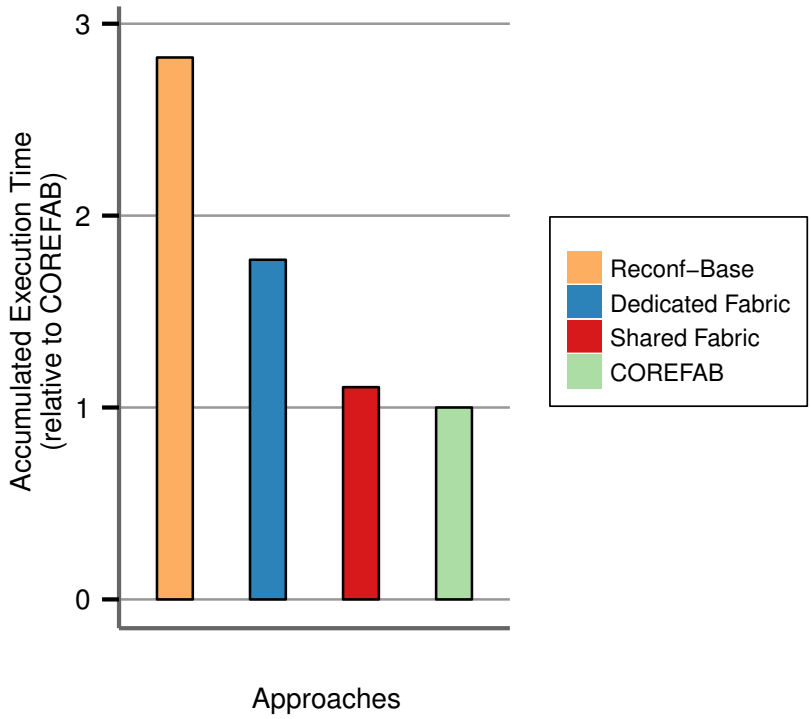


Figure 6.11: Accumulated execution time over all 4 cores (averaged over all workloads).

complete its application (e.g. if core 1 took $2 \cdot 10^6$ cycles and core 2 took $5 \cdot 10^6$ cycles, the accumulated execution time would be $7 \cdot 10^6$ cycles). This value corresponds to the total activity in the multi-core system, which also indicates the energy usage of the system. COREFAB results in the smallest accumulated execution time (on average 10% less than Shared Fabric, the closest competitor).

Performance of the evaluated approaches depends on the workload, e.g. examining only Workloads 1–3, Shared Fabric has 49% worse performance on the GPPs than COREFAB, while for Workloads 4–6 Shared Fabric is only 1% slower. The reason for this is that in Workloads 4–6 the application on the primary core has very large inter-SI gaps, such as for the SUSAN application (see Table 6.2). Such gaps leave

enough "idle time" on the fabric to allow applications running on other cores to run their SIs on the fabric during these gaps, effectively serializing SI execution from multiple cores with (almost) no performance impact. However, COREFAB is designed to improve performance in the presence of concurrent fabric accesses which occur when the applications have small inter-SI gaps (i.e. SIs are frequent and of medium to long latency). If the workload profile is similar to Workloads 4–6, COREFAB provides little benefit (however, COREFAB also does not perform worse than Shared Fabric).

Dedicated Fabric benefits from workloads consisting of tasks that (i) do not have high memory bandwidth requirements (or at least if all applications of the workload have similar bandwidth requirements) and (ii) achieve most of their speedup within the fabric share assigned to the core they run on (or at least if the applications have similar performance characteristics, as shown in Figure 6.4). These requirements are satisfied by Workload 6, where AES and SHA achieve most of their speedup with 1 or 2 RACs, respectively. Of course, the dedicated fabrics can also be customized at design time, e.g. the fabric of core 1 could be assigned more bandwidth (suitable for memory SIs), while another core could be assigned more fabric area (for computationally bound SIs). However, this static assignment does not address variations within an SI in respect to fabric resource usage (e.g. memory-bound at the start and end of the SI, computationally bound in between). Thus, the fabric would be optimized only for a certain type of workload (e.g. must have one memory bound and one computationally bound application to fully exploit the customized fabric resource assignment).

COREFAB achieves the best results due to its ability of merging concurrent fabric accesses, but sometimes concurrent fabric accesses result in resource conflicts, leading to stalling of the Remote-SI. The presented evaluations have shown that the major reason for conflicts is a busy memory port (98% of all conflicts) with insufficient link capacity responsible for the remaining conflicts (2%). No RAC conflicts occurred, as the minimal amount of RACs required for running the application from the workload in hardware was assigned to each GPP by the fabric allocator when an application was started.

For overhead analysis, the SI Merger and FAM modules (the two main components of COREFAB) were implemented as standalone hardware modules and were synthesized using Synopsys Synplify H-2013.03 for a Virtex-5 LX110T FPGA. The area overhead of these COREFAB extensions is 1231 LUTs (98 LUTs for FAM and 1133 LUTs for the SI merger). As shown in Section 6.1, SI μProgram memory consists of BlockRAM, thus an additional 14 BlockRAMs are required for an additional instance of SI μProgram memory used to store Remote-SIs.

6.5 Partial Online Synthesis

In this section, the placement, binding and μProgram caching techniques proposed in Chapter 5 are evaluated in detail, and a reconfigurable system that uses partial

online synthesis of SI μPrograms is compared to a reconfigurable system that uses compile-time generated SI μPrograms.

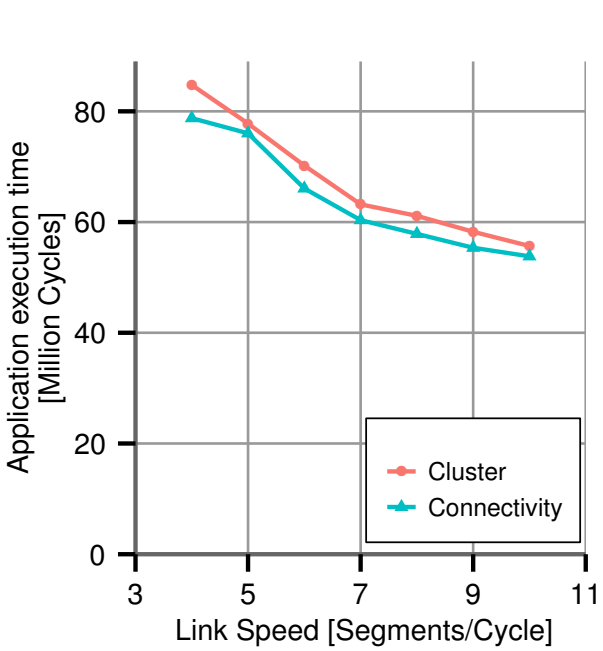
As a proof-of-concept and in order to improve simulation accuracy, a complete run-time system that includes placement and binding was implemented as a user-space application for a version of Linux modified for the *i*-Core, and verified on a Virtex-5 FPGA-based prototype of the *i*-Core (see Section 6.1). For accurate algorithm overhead estimation, cycle measurements of placement and binding algorithms in the Linux-based run-time system were used to calibrate the overhead simulation in the SystemC based simulator.

6.5.1 Placing accelerators

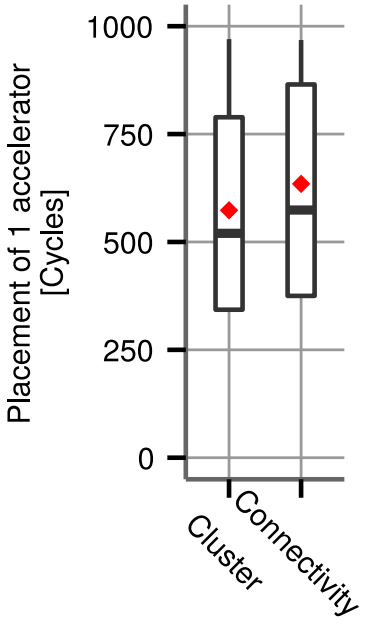
To evaluate the proposed placement algorithms, the completion time of an application is measured when using a particular placement algorithm. Frequent reconfigurations should be performed during execution in order to stress the placement algorithm. This requires an application that performs continuous prefetches. Additionally, the application should use sufficiently large SI variants (i.e. it should use a large number of RACs), so that effects of bad placements become evident (as when few RACs are used no transfer delay hazards occur, no matter what placement is used). Both conditions are met by the H.264 Video Encoder application, which will be used for evaluation here (the full partial online synthesis approach will be evaluated with additional applications). The binding algorithm used was CAB.

Figure 6.12a shows the application execution time for different link speeds (in segments that can be traversed in one cycle) of the reconfigurable fabric (fabric sizes ranged from 8–25 RACs, plotted values are averaged over all fabric sizes). With low link speeds, transfer delay hazards occur more often (especially for bad placements), resulting in longer execution time and thus worse performance. On average, Connectivity Placement results in 5.0% faster performance than Cluster Placement, as it considers the communications between the accelerators during placement.

To measure their overhead, the placement algorithms were implemented as standalone C programs and their execution time was measured using the ArchC SPARC V8 instruction set simulator. Figure 6.12b shows a boxplot of the amount of cycles required to place one accelerator. On average, it takes 574 cycles using Cluster Placement and 635 cycles using Connectivity Placement. Compared to the time it takes to reconfigure one accelerator this overhead is negligible (0.6–0.7 ms correspond to 60,000–70,000 cycles at 100 MHz), therefore the recommend algorithm is Connectivity Placement due to its better performance.



(a) Execution time of an H.264 video encoder using Cluster and Connectivity placements.



(b) Boxplot of the overhead for placing one accelerator (mean marked by red diamond).

Figure 6.12: Performance and Overhead results for different placement strategies.

6.5.2 Binding

Binding directly affects the execution latency of an SI variant, thus for a direct comparison between binding algorithms, the results of binding are compared among each other instead of the execution times of full applications. Various system configurations (summarized in Table 6.8) were used, that reflect different fabric hardware architectures parameters. On each of these configurations each SI variant from the applications listed in Section 6.2 is bound to 100 random fabric configurations (resulting in $\approx$ 1.6 million bindings in total). Table 6.9 shows how often a particular binding results in the fastest SI execution latency. Communication-Lookahead Binding (CLB) leads to the best results, followed closely by Communication-Aware Binding (CAB).

While on average CLB creates the fastest SI variants, it takes over twice as long as CAB and over four times longer than First Fit Binding (FFB). Whether or not the overhead of CLB is acceptable depends on how often the SI is executed. For example, one of the SI variants for the *DCT* SI is bound by CLB resulting in a latency of 35 cycles per execution but with the binding overhead of 461,040 cycles. The same SI variant is bound by FFB (which has the lowest overhead of the other binding algorithms) to the same fabric configuration resulting in an execution latency of 37 cycles with a binding overhead of 74,690 cycles. Therefore, the additional overhead of CLB amortizes if this SI variant is executed for 193,175 times on the same fabric

Parameter	Value
Links	4, 6, 8
Link speed [segments/cycle]	2, 6, 10
RACs	20
Fabric configurations	100
SI variants	453 variants of 29 SIs
Binding Algorithms	Random, First Fit (FFB), Comm.-aware (CAB), Comm.-Lookahead (CLB)

Table 6.8: Parameters for binding algorithm evaluation.

Algorithm	FFB	CAB	CLB	Random
Achieved Best Result	33.5%	70.0%	75.0%	23.2%

Table 6.9: Binding evaluation: Amount of times a binding strategy achieved the best result among all other strategies for a particular SI variant.

configuration (for a 720p frame, the DCT SI is executed 144,000 times). During evaluation no situation was encountered where an SI was executed for such a large number of times for the same fabric configuration, although for larger workloads (e.g. higher resolution) such numbers are feasible. Therefore, for small to medium-sized workloads, even when using μProgram caching (evaluated in the next section), the recommended binding algorithms are CAB or FFB as they provide a better compromise of the obtained result and the overhead than CLB.

6.5.3 Caching

As shown in the previous section, the overhead to bind an SI variant is significant, and can diminish or completely negate the benefits (shown later during evaluation) provided by generating μPrograms at run-time. However, using the software-based caching approach discussed in Section 5.6, this overhead can be reduced without requiring any additional hardware. As an example, the H.264 Video Encoder takes 410.6 M cycles to encode a particular workload using a reconfigurable fabric with 10 RACs and without caching SI μPrograms. With 50 KB of μProgram configuration cache and FIFO replacement, the same workload takes 343.8 M cycles, a run-time improvement of 19%. The effects of cache size, replacement strategy, fabric size and accelerator placement strategy on system performance are investigated in the following. For the following evaluation, the H.264 video encoder was used (as it exhibits dynamic behavior due to its multiple kernels and therefore will require regular re-generation of μPrograms) encoding a workset of 60 frames. The binding algorithm used in the following was CAB.

Cache size determines the amount of SI variants (bound to a specific fabric configuration) that can be stored in the cache. Figure 6.13 shows the hitrate for different cache sizes (25 KB to 100 KB) for different fabric sizes. Hitrates were evaluated for a cold μProgram cache, thus for very large fabric sizes (19 and 20 RACs) μPrograms were generated only the first time a particular μProgram was used. As the accelerators for all SI variants used throughout the execution of H.264 fit onto the fabric together, once the μPrograms for all these SIs were generated, they could be used until the application finished. As with regular caches, a large cache size yields the best hitrates, with 79% for 100 KB, 75% for 75 KB, 59% for 50 KB and 35% for 25 KB. LFU cache replacement provides the best results over all cache sizes with a 68.3% hitrate, followed by MRU and FIFO with 67.5% and 67.3%, respectively. The choice of the accelerator placement strategy also affects hitrates with Cache-Aware Placement resulting in a 69.7% hitrate and Connectivity placement with 54.6%.

A cache size of 100 KB *per task* would be quite large for on-chip SRAM memory. Additionally, the transfer time from main memory to SI μProgram memory is quite small, thus the cache should reside in main memory (DDR) as part of the run-time system. The μProgram cache is only a software entity, and can be easily modified (new replacement algorithms, different per-task cache-size, etc). The fairly high hitrates for an investment of only 100 KB of main memory cache may be seen as an argument against online generation of SI μPrograms, with an alternative being a small amount (e.g. 100 KB) of offline-generated SI μPrograms. While this would be a feasible approach for single-tasking systems, multi-tasking systems which support dynamic tasksets (encountered in desktop systems, smart-phones, etc.) allocate the fabric according to the currently active tasks, which are unknown at compile time (as shown in Section 5.1). As this run-time allocation influences which μPrograms can be used, offline generated μPrograms are infeasible for such systems.

While the hitrate is a good metric for the performance of the cache itself, the deciding metric for the system designer is overall performance improvement due to using caching. Figure 6.14 shows the execution time for the H.264 encoder, a cache size of 50 KB and FIFO replacement policy compared to a system with no caching. SI μProgram caching provides a performance benefit of up to 1.22×, depending on fabric size.

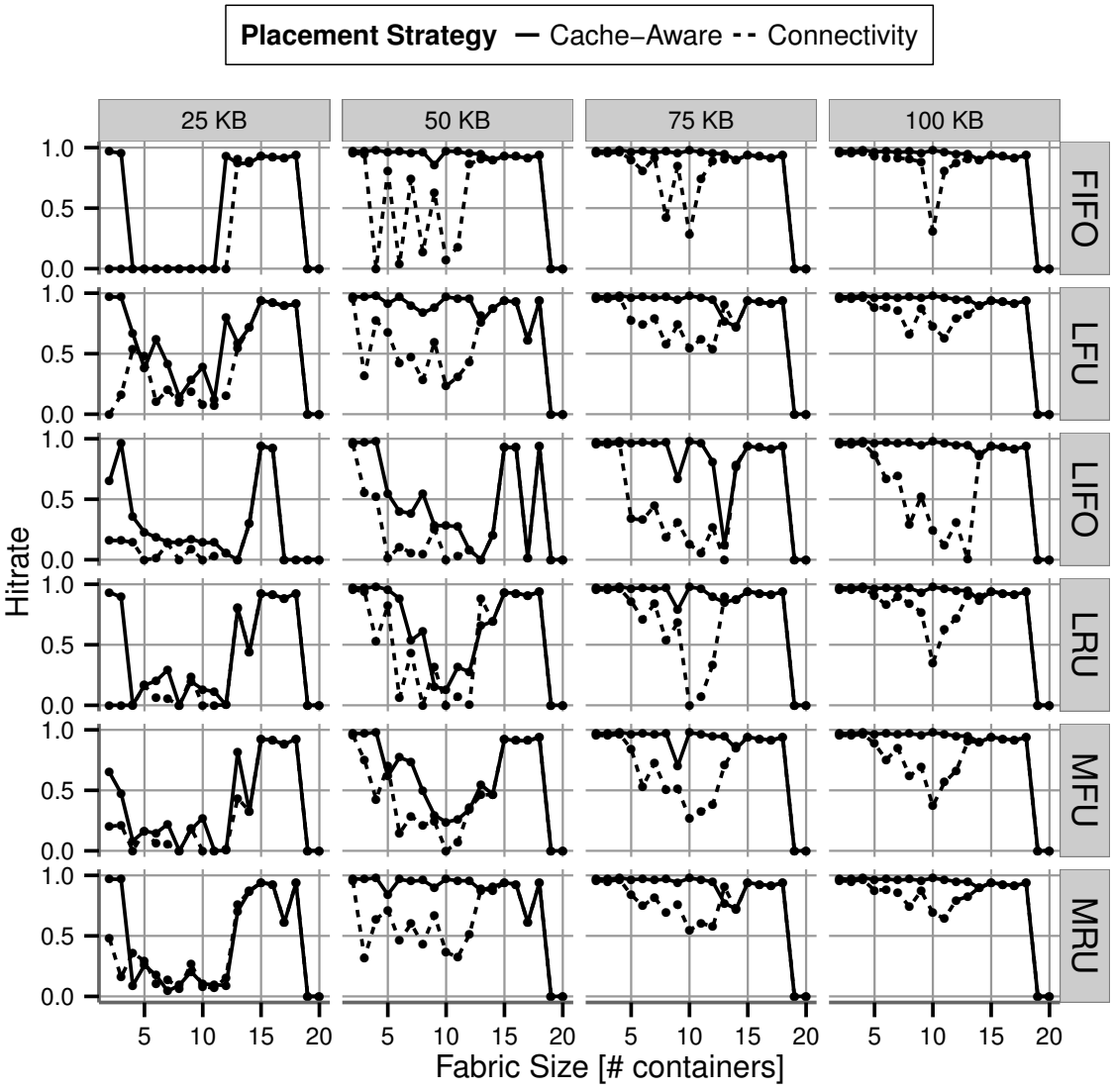


Figure 6.13: Hitrates for different μProgram cache sizes, cache replacement policies and accelerator placement policies. A cold cache was used, thus the hitrates of 0 for 19 and 20 RACs are due to the fact that the cache was not accessed at all (the μPrograms were generated once, and no further reconfigurations were performed due to the large fabric size).

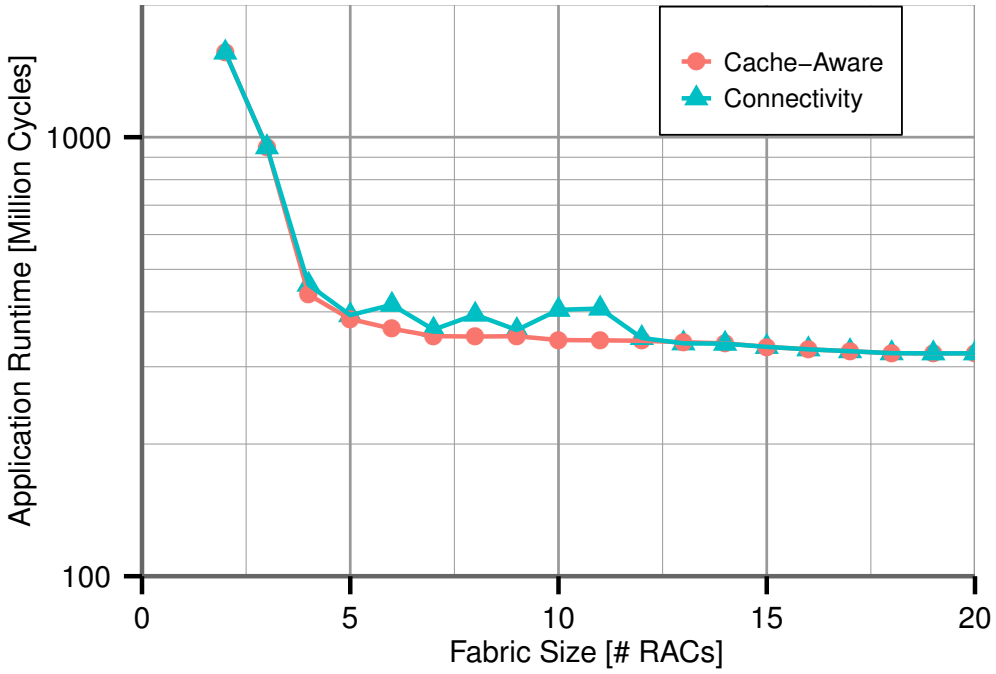


Figure 6.14: Application speedup for different sizes of the μProgram cache using FIFO replacement policy and a cachesize of 50 KB.

6.5.4 Comparison of partial online synthesis vs. offline generated SI μPrograms

To compare a system using partial online synthesis of SI μPrograms against a system with offline generated SI μPrograms (called "static" system in the following), both systems are used in 5 multi-tasking scenarios. The tasksets for these scenarios use the applications from Section 6.2. Each task performs a certain amount of work (e.g. decode an 800x600 pixel JPEG image) and terminates afterwards, i.e. tasks are not periodic and do not have deadlines.

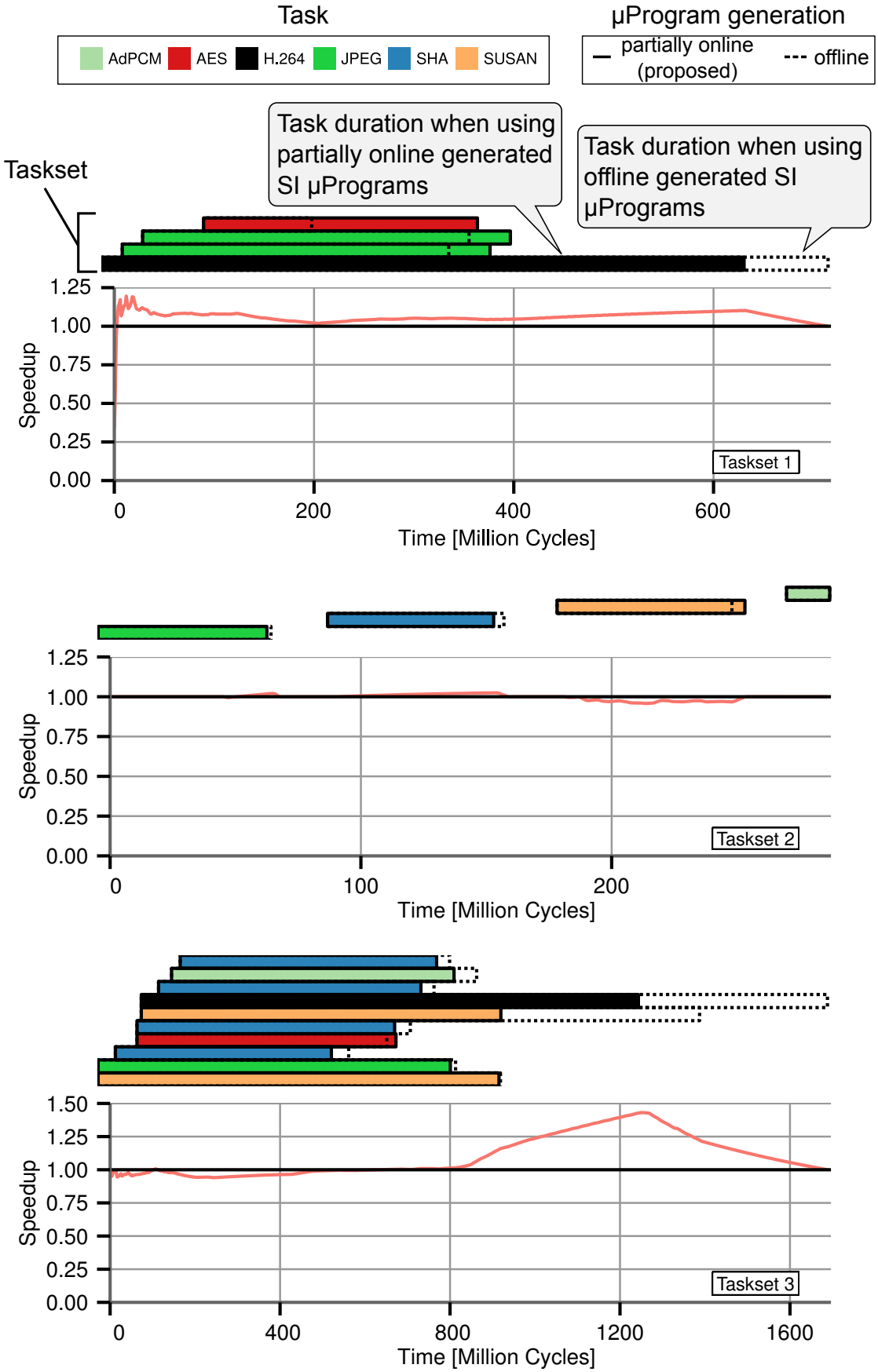
In order to evaluate only the SI μProgram generation, no specialized task schedulers from Chapter 3 are used, and the tasks are scheduled using a round-robin strategy. Figure 6.14 shows the results for each of the 5 tasksets. The tasksets were chosen to cover a diverse set of scenarios that demonstrate in which situations partial online synthesis provides benefits over the static systems, and in which it does not. In each figure, the top part (horizontal bars) shows a Gantt chart for task activity, and the bottom part (line chart) shows the speedup of the proposed partial online synthesis compared to the static system. In the Gantt chart, a solid bar shows the time when the task was active (i.e. runnable or running) on the system with partial online synthesis, while an overlayed dotted bar outline shows when the same task was active when scheduled on a static system.

After a task is started, it obtains a share of the reconfigurable fabric where it can reconfigure its accelerators. The size of that share depends on the task priority (set at compile time) and the amount of other tasks currently using the fabric. Both systems use a reconfigurable fabric with 10 RACs and all other parameters are identical as well. The partial online synthesis system uses Cache-Aware Placement and Communication-Aware Binding strategies, while the static system uses Connectivity Placement and pre-computed SI μPrograms.

As shown in Section 5.1, providing offline-generated μPrograms for all possible SI variants of an SI is practically infeasible. Thus, the static system with the offline generated μPrograms uses a limited set of SI variants. The applications were profiled for different fabric sizes and for each SI the most efficient SI variant was identified, i.e. the variant where the ratio $performance/\#accelerators$ is highest. To ensure

Taskset	Speedup of partial online synthesis	
	on a dedicated core	on a shared core
1	1.13	1.10
2	1.00	0.90
3	1.35	1.28
4	1.15	1.05
5	1.85	1.73
Mean	1.30	1.21

Table 6.10: Speedup of a system using partial online synthesis vs. static system


Task
AdPCM
AES
H.264
JPEG
SHA
SUSAN
µProgram generation
partially online (proposed)
offline
Taskset
Task duration when using partially online generated SI µPrograms
Task duration when using offline generated SI µPrograms
Speedup
1.25
1.00
0.75
0.50
0.25
0.00
0
200
400
600
Time [Million Cycles]
Taskset 1
Speedup
1.25
1.00
0.75
0.50
0.25
0.00
0
100
200
Time [Million Cycles]
Taskset 2
Speedup
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
400
800
1200
1600
Time [Million Cycles]
Taskset 3

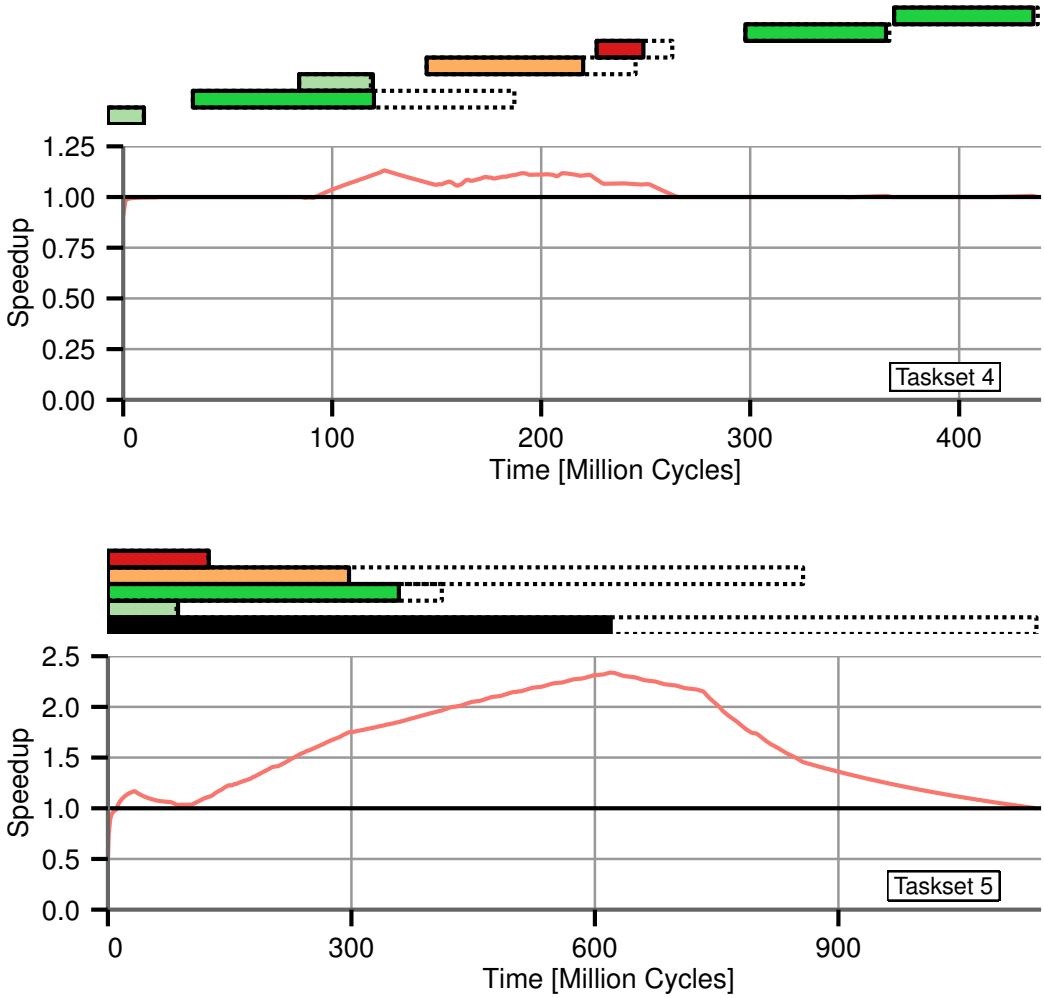


Figure 6.14: Comparison of a reconfigurable processor using partial online generated SI μPrograms (proposed approach) vs. one using offline generated SI μPrograms in a several multi-tasking scenario.

that the static system is not unfairly penalized, it may use *any* fabric configuration for each such SI variant.

The speedup plotted in Figure 6.14 is defined at time t between two systems a and b as shown in Equation (6.1), where $instructions\_executed(i,t)$ is the number of executed instructions for all tasks of system i in the period $[0,t]$. This definition of speedup is used to allow comparison of both systems at any point in time. One consequence of this definition is that assuming system a has finished execution at time t_a, and system b finishes at a later time t_b, then speedup will converge to 1.0 during the time span $[t_a, t_b]$ (this can be observed for every taskset in Figure 6.14).

$$speedup(t) = \frac{instructions\_executed(a,t)}{instructions\_executed(b,t)} \tag{6.1}$$

When a reconfigurable processor is used as part of a multi-/many-core system, the run-time algorithms for SI μProgram generation can be run on a different core than the application ("dedicated core"), or on the same core as the application ("shared core"). This was evaluated and the achieved speedups are shown in Table 6.10. The reason for the difference in speedups is that on a shared core, application execution is suspended while the μProgram for an SI is generated. If the algorithms for partial online synthesis run on a dedicated core, then during the time a μProgram is generated, the application can still be executed on the reconfigurable core, although slower (as while the μProgram for an SI variant is not yet available, the SI will have to be run in a slower variant, or on the core pipeline).

Overall, the speedup of partial online synthesis was lowest for tasksets 2 and 4, where the fabric is used by only 1 or 2 tasks at a time, and thus the offline generated SI μPrograms of the static systems provide very good performance, while not incurring any overhead. However, when there are frequent changes in fabric allocation during execution of a taskset (tasksets 3 and 5), some tasks may not receive the fabric share required for the offline generated SIs used in a static system, and the system has to fall back to executing these SIs on the core pipeline, resulting in slower execution speed. With partial online synthesis, μPrograms can be generated for smaller SI variants, which are still faster than SI execution on the core pipeline.

For tasksets that exhibit a high degree of dynamicity (1, 3 and 5) the mean speedup of partial online synthesis compared to the static system is 1.44× when using a dedicated core for SI μProgram generation and 1.37× on a shared core. For tasksets 2 and 4, where fabric re-allocation is hardly present, partial online synthesis leads to a marginal speedup of 1.08× when used on a dedicated core and a slow down to 0.98× on a shared core. Over all tasksets the mean speedup when using partial online synthesis was 1.3× on a dedicated core and 1.21× on a shared core.

6.6 Joint Approach

After stand-alone evaluations of the approaches for multi-tasking on reconfigurable processors, sharing the reconfigurable fabric in a multi-core system and improved fabric flexibility, a system using all three of these contributions is demonstrated. The proposed system uses COREFAB for fabric sharing (Chapter 4), MORP for task scheduling (Chapter 3) and partial online synthesis of μPrograms with μProgram caching for improved fabric flexibility (Chapter 5). Table 6.11 describes the parameters of the proposed system and a system using state-of-the-art approaches used for comparison. Both systems run one workload, with Core 1 running the H.264 video encoder and a software-only application (SHA without SIs), and Core 2 is running the AdPCM audio encoder. The workload is kept simple to allow analyzing which approach effects the results in which way. The algorithms for partial online synthesis are assumed to be run on an a dedicated core (see Section 6.5.4).

	Proposed	State-of-the-Art
Fabric Size	10	
Links (in one direction)	4	
Link Speed [segments/cycle]	30	
Multi-Core Fabric Integration	COREFAB	Shared Fabric
Task Scheduler	MORP	FCFS
Fabric Allocator	MORP	FCFS
μProgram Generation	Partial Online Synthesis	Offline
Accelerator Placement	Cache-Aware	N/A
Binding Strategy	CAB	
Cache Size	50 KB	
Cache Replacement	FIFO	

Table 6.11: Parameters for Proposed and State-of-the-Art system used for joint evaluation.

Figure 6.15 shows the execution times for the workload when run by both systems. On Core 1 the proposed system is 16% faster, as the MORP scheduler uses the SHA task to hide the reconfigurations of H.264 and thereby reduce its RiCL, improving performance. On the other hand, the FCFS scheduler suffers from a high RiCL (as motivated in Section 3.1) when running H.264. As SHA is a software-only task, it runs with the same speed on both systems. On Core 2 the proposed system is 30% faster, as COREFAB merges the SIs of H.264 and AdPCM to execute them concurrently, while the shared fabric of the state-of-the-art system serialized SIs, resulting in worse performance.

The results include the overhead of binding for partial online synthesis. While this simple workload shown here is not particularly dynamic, and thus even a system using a limited amount of offline generated SIs uses the fabric efficiently, this shows that the overhead can be reduced by caching of μPrograms and using cache-aware accelerator placement.

Overall, this evaluation shows that the three proposed approaches for the use of reconfigurable processors in multi-tasking and multi-core systems can complement each other as part of a combined system.

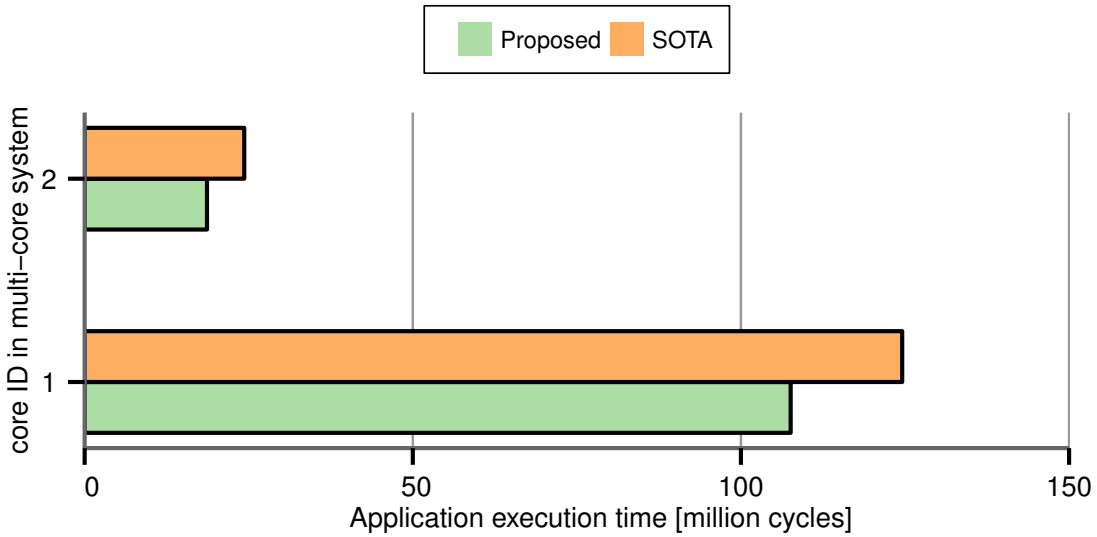


Figure 6.15: Comparison of the proposed Multi-tasking Reconfigurable Multi-Core Processor with a state-of-the-art approach.

6.7 Summary

The concepts for reconfigurable processors in multi-tasking and multi-core systems proposed in this thesis were evaluated in this chapter.

The task schedulers from Chapter 3 were designed around the notion of task efficiency on a reconfigurable processor, and were designed to improve tardiness or makespan. For workloads with periodic tasks with deadlines, PATS (Section 3.4) was able to meet deadlines better than standard schedulers (which are used in similar reconfigurable processors), achieving a 1.22× better performance than the closest competitor. MORP, a combined approach for task scheduling and fabric allocation (Section 3.5) produced schedules with better makespans than existing approaches. Compared to a lower bound (which assumes an architecture than can perform reconfigurations without any delay), the makespans achieved by MORP were only 6% larger, while those of other approaches were 12% and 20% worse. Performance of both proposed schedulers was dependent on the workload, with workloads consisting of a diverse mix of tasks providing the highest optimization potential. Such workloads consisted of both complex tasks that performed frequent reconfigurations, simpler tasks that used fewer accelerators which were reconfigured once at the beginning, as well as software-only tasks than did not use reconfigurable fabric at all.

Sharing the reconfigurable fabric in a multi-core in an efficient way was proposed in Chapter 4 by using a heterogeneous multi-core architecture, consisting of one reconfigurable processor and multiple non-reconfigurable cores connected to its fabric. State-of-the-art approaches fall into two classes: shared fabric and dedicated fabric. Compared to these approaches, the proposed technique COREFAB achieved better performance for the whole multi-core. In particular, COREFAB improves

performance on the non-reconfigurable cores by 1.3× on average (up to 1.49× for workloads with sufficient optimization potential) and matches performance on the reconfigurable core when compared to the shared fabric approach The dedicated fabric approach had worse performance than COREFAB on the reconfigurable core (by 3.1× on average), while having the same or slightly better performance on the non-reconfigurable cores. As with task scheduling, the type of workload affected the performance difference of the approaches, with COREFAB having achieved a high performance for non-balanced workloads that consisted of both complex and simple tasks.

The partial online synthesis approach for μProgram generation (Chapter 5) provides the flexibility required of using the fabric in dynamic multi-core and multi-tasking scenarios. Evaluation of the algorithms has shown that best performance and least overhead is achieved by using the proposed μProgram software-cache, the communication-aware binding algorithm and cache-aware accelerator placement. Compared to a system that uses (a limited amount of) fully offline generated SIs, partial online synthesis provided a 1.3× speedup (1.21× when sharing the core with the application that is accelerated).

Finally, a case study of a reconfigurable multi-core system employing all proposed techniques has shown a performance improvement of 16% on the reconfigurable core, and a 30% improvement on the non-reconfigurable core when compared to a system that used state-of-the-art approaches for task scheduling, fabric sharing and μProgram generation.

7 Conclusion and Future Work

7.1 Conclusion

This thesis presents approaches for the use of highly adaptive reconfigurable processors in multi-tasking scenarios and as part of multi-core systems.

Multi-tasking can be used to offset an important drawback of reconfigurable processors: the long time to reconfigure an accelerator. This can be done by interleaving reconfiguration of one task and scheduling the execution of another task. The notion of task efficiency is introduced to capture this concept, describing how well a task benefits from the current state of the reconfigurable processor. In addition to the accelerators currently loaded onto the fabric, task efficiency depends on which accelerators a task currently requires. By reacting to changes in task efficiency, the system can aim to schedule tasks with a high efficiency, knowing that low-efficiency tasks will improve their efficiency as their reconfigurations complete. Two schedulers based on this concept are presented, PATS, a scheduler for workloads with deadlines and MORP, a scheduler for workloads without deadlines. PATS is an extension of existing schedulers (Earliest Deadline First and Rate-Monotonic) with the notion of task efficiency and the goal of improving tardiness (i.e. the cumulative time by which deadlines are missed). This allows either a higher utilization of the reconfigurable processor, or tighter deadlines than for a system scheduled with a scheduler that does not consider task efficiency. MORP is a combined approach for both task scheduling and fabric allocation, with the goal of reducing the makespan of a taskset. Shortly before an application switches to a different kernel, MORP re-allocates a small part of the fabric allocated to the currently running task to a different secondary task. When the running task switches to a new kernel, its task efficiency drops, and MORP schedules the secondary task, which can run at improved efficiency due to the re-allocated fabric. Once the first task has achieved sufficiently high task efficiency (i.e. when its reconfigurations are nearly finished), the scheduler gives it the whole fabric again, and schedules it. These specialized schedulers are the first that are designed specifically for reconfigurable processors that use a flexible Special Instruction (SI) model which allows for trade-off between performance and area at run-time.

Even when running demanding applications, only a small part of the fabric resources are used at any time during the execution of a single SI, while the remaining fabric resources are idle. These idle resources can be used to run additional SIs in parallel. The proposed approach, COREFAB allow this by sharing the fabric in a multi-core. COREFAB, introduces a heterogeneous multi-core architecture, consisting of one

reconfigurable core with a fabric and multiple non-reconfigurable cores. A protocol is proposed to allow the non-reconfigurable cores access to the fabric, and the novel concept of SI merging allows executing the SIs in concurrently. To do so, the SIs are merged on a cycle-by-cycle basis at run-time in hardware. In the presence of a conflict due to multiple SIs attempting to use the same fabric resource in a single cycle, COREFAB stalls one of the SIs for a single cycle (while the execution of the other SI is not affected), and attempts to merge in the next cycle again. This type of fine-grained merging minimizes the impact of resource conflicts during concurrent SI execution.

This use of reconfigurable processors in multi-tasking scenarios and multi-core systems demands a high degree of flexibility from the fabric. Dynamic workloads run on such systems cause the fabric to be allocated to different tasks or cores in a way that is unknown at compile-time. Thus SIs synthesized for the fabric at compile-time are often not usable, as they generally assume a different fabric allocation than the one that is actually established at run-time. To address this problem, a way to synthesize SIs partially at compile-time and partially at run-time is proposed. Only those steps of SI synthesis that depend on the which accelerators are loaded where on the fabric are synthesized at run-time. The independent parts of SI synthesis (such as graph scheduling and synthesis of single accelerators) are done at compile-time. Implementation and subsequent measurement of this system on a FPGA-prototype of a reconfigurable multi-core processor have shown that the run-time part still induces considerable overhead. To reduce this, a novel software-cache designed for synthesized SIs is introduced, along with an accelerator placement technique, which results in improved hitrates (and reduced overhead).

Evaluation of the approaches show performance improvement for each proposed approach. PATS improves tardiness by 1.22×, and MORP achieves makespans that are only 6% worse than the lower bound (while other schedulers are 12%–20% worse than the lower bound). COREFAB improves performance on the reconfigurable core by 3.1× (with only 2% performance decrease on the non-reconfigurable cores) when compared to one type of state-of-the-art, and 1.3× improvement on the non-reconfigurable cores (while maintaining the same performance on the reconfigurable core) on a different type of state-of-the-art reconfigurable multi-core. The approach for synthesizing SIs partially at run-time allows a more flexible fabric use, resulting in 1.3× better performance compared to a system that fully synthesizes SIs at compile time.

The reconfigurable multi-core processor developed as part of this work was integrated into the heterogeneous many-core architecture of the Invasive Computing project, and deployed as an FPGA prototype.

7.2 Future Work

This thesis focused on using the reconfigurable fabric in multi-tasking scenarios and multi-core systems, but several other promising future research directions can be explored in the future. These focus on two areas: (i) improving usability (*Security* and *Predictability*) and (ii) further improving fabric efficiency (*Fabric-internal Heterogeneity* and *Control-Flow support in Special Instructions*).

Security Adding a reconfigurable fabric to a processor introduces several security issues, especially when used in a multi-tasking environment where not all tasks are trusted. In the following example scenarios, a reconfigurable multi-core processor consisting of a reconfigurable core c_0 and a non-reconfigurable core c_1 is used. The tasks T_A and T_B are non-malicious, while T_E and T_M are malicious.

If an SI used by T_A does not clear its internal state (private register files, accelerator-internal storage), a special SI used by T_E that executes afterwards can be used to read out the data from the accelerator previously used by T_A and store it to main memory, send it via network, etc. Special care needs to be taken when handling sensitive data, such as encryption keys, clear text that needs to be encrypted, etc. Either SI designers have to take care to wipe any storage used at the end of the SI μProgram, or the SI execution controller will have to clear the fabric automatically after each SI. This issue becomes even more important if SIs are made interruptible to allow for faster task switching or to improve real-time behavior. In case an SI executed by T_A is interrupted by a task switch, SIs are even more likely to have sensitive information in fabric-internal storage, and the SI designer cannot prevent this issue. A more severe variant of this attack would be if T_M used an SI to modify the intermediate data left by the interrupted SI of T_A. As the task switch is triggered by OS, it needs to take care of clearing and restoring fabric-internal storage, possibly with hardware support.

Predictability If the fabric is used exclusively by one core, most operations within the reconfigurable fabric happen in a fixed and predetermined amount of cycles, with the only exception being memory accesses from the fabric to the memory hierarchy. When restricting SIs to accessing on-chip memory this unpredictability issue can also be addressed by prioritizing communication between the fabric and the on-chip memory (in case a dedicated connection is used).

When sharing the fabric between multiple cores, as described in this work, several components need to be extended to make SI execution predictable. The Fabric Access Manager could be configured to designate a *predictable core*. In case of a conflict only the SI of this core would be allowed to proceed, while the SIs of all other cores would be stalled for one cycle. A core could program the Fabric Access Manager with a timeout, which if exceeded, would abort SI execution for this particular core. Aborting SI execution would be helpful if the fabric has a high utilization due to SIs from other cores, leading to frequent conflicts and thus stalling. The core would

then re-execute the aborted SI in software, which while slower, would at least place an upper bound on SI latency.

The Fabric Access Manager could also be programmed with a specific *budget*, which would count the cycles of the SI of a predictable core that are lost due to conflicts. Fabric sharing would proceed using a best-effort approach until this budget is reached. At this point the Fabric Access Manager would stall all other SIs until the SI of the predictable core has finished, thus guaranteeing that the latency of this SI does not exceed the user-specified bound.

Fabric-internal Heterogeneity In this work a homogeneous fabric was used, i.e. all reconfigurable accelerator containers were identical. However, not all accelerators require the same amount of logic, leading to internal fragmentation in a RAC, as simpler accelerators use only a fraction of a RAC. Furthermore, some accelerators can be implemented faster or more area-efficient using "hard" IP cores embedded in the fabric: a floating point accelerator can be implemented using embedded multipliers, while cryptographic accelerators can use embedded memory blocks to implement S-boxes. A heterogeneous fabric would consist of RACs of different sizes and some RACs would provide embedded IP cores. More complex accelerators may also process more input data, thus larger RACs could have a wider link to the fabric interconnect. Accelerators would need to be synthesized for each of these different types of RAC, or be constraint only to a subset of RACs.

In addition to these architectural changes, the decisions of the run-time system which SI variant to use and into which RAC to reconfigure an accelerator would become more complex. When multiple tasks compete for the fabric, the run-time system has to optimize the accelerator-to-RAC mapping over multiple tasks. As tasks are not synchronized in their SI execution, a run-time system reconfiguring the fabric for task T_A could have the possibility to evict an accelerator from a RAC currently used by task T_B (e.g. due to the accelerator required for T_A requiring this particular RAC due to its size or embedded IP block). In this case, the run-time system would have to decide if the performance loss of T_B due to losing the accelerator is acceptable.

Control-Flow Support in Special Instructions A kernel can be described by a control-data-flow graph. SIs can only describe the data-flow part, the control-flow part is executed on the core pipeline. If a kernel switches frequently between control- and data-flow (e.g. when searching for a pattern, traversing a tree), speedup will be small, as SIs will be very short and fabric features such as the wide memory ports cannot be fully exploited.

The SI execution controller that processes SI μPrograms could be extended with support for conditional branching, allowing to modify SI control flow depending on results from the accelerators. For example, an SI used for searching could use multiple accelerators to test input data loaded via the memory ports for a specified pattern, and signal the SI execution controller when the pattern was found, terminating the

SI. This feature would allow supporting a whole new class of SIs (those including control-flow), further improving performance.

Bibliography

[ABCG+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[ABP11] G. Ansaloni, P. Bonzini, and L. Pozzi. "EGRA: A Coarse Grained Reconfigurable Architectural Template". In: *VLSI Systems*. Vol. 19. 6. 2011, pp. 1062–1074.

[AC01] Marnix Arnold and Henk Corporaal. "Designing Domain-specific Processors". In: *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*. 2001, pp. 61–66.

[ADÖ05] Kubilay Atasu, Günhan Dündar, and Can Özturan. "An Integer Linear Programming Approach for Identifying Instruction-set Extensions". In: *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 2005, pp. 172–177.

[Aer] Aeroflex Gaisler AB. *LEON 3*. `http://www.gaisler.com`.

[AHKL+14] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. "ReconOS: An Operating System Approach for Reconfigurable Computing". In: *IEEE Micro* 34 (2014), pp. 60–71.

[Alt15] Altera Corporation. *Cyclone V Device Family Overview*. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf`. 2015.

[ARBA+05] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. "The ArchC architecture description language and tools". In: *International Journal of Parallel Programming* 33.5 (2005), pp. 453–484.

[ASBH11a] Waheed Ahmed, Muhammad Shafique, Lars Bauer, and Jörg Henkel. "Adaptive Resource Management for Simultaneous Multitasking in Mixed-Grained Reconfigurable Multi-core Processors". In: *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2011, pp. 365–374.

[ASBH11b] Waheed Ahmed, Muhammad Shafique, Lars Bauer, and Jörg Henkel. "mRTS: Run-Time System for Reconfigurable Processors with Multi-Grained Instruction-Set Extensions". In: *Design, Automation and Test in Europe (DATE)*. 2011, pp. 1554–1559.

[BBDG08] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. "Architecture Enhancements for the ADRES Coarse-grained Reconfigurable Array". In: *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*. 2008, pp. 66–81.

[BBFB07] Sven Bachthaler, Fernando Belli, Alexandra Fedorova, and BC Burnaby. "Desktop workload characterization for CMP/SMT and implications for operating system design". In: *Workshop on the Interaction between OS and Computer Architecture*. 2007, pp. 2–9.

[BC11] Shekhar Borkar and Andrew A. Chien. "The Future of Microprocessors". In: *Commun. ACM* 54.5 (2011), pp. 67–77. URL: http://doi.acm.org/10.1145/1941487.1941507 (visited on 07/28/2015).

[BDMF10] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. "Evolution of Thread-level Parallelism in Desktop Applications". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 2010, pp. 302–313. URL: http://doi.acm.org/10.1145/1815961.1816000 (visited on 08/20/2015).

[BH11] Lars Bauer and Jörg Henkel. *Run-time Adaptation for Reconfigurable Embedded Processors*. 2011.

[BJW07] M. Butts, A.M. Jones, and P. Wasson. "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing". In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM 2007*. 2007, pp. 55–64.

[BRAS11] I. Beretta, V. Rana, D. Atienza, and D. Sciuto. "A Mapping Flow for Dynamically Reconfigurable Multi-Core System-on-Chip Design". In: *TCAD* (2011), pp. 1211–1224.

[BRGC08] A.C.S. Beck, M.B. Rutzig, G. Gaydadjiev, and L. Carro. "Transparent reconfigurable acceleration for heterogeneous embedded applications". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2008, pp. 1208–1213.

[BSB13] H.-P. Bruckner, C. Spindeldreier, and H. Blume. "Energy-efficient inertial sensor fusion on heterogeneous FPGA-fabric/RISC System on Chip". In: *2013 Seventh International Conference on Sensing Technology (ICST)*. 2013, pp. 506–511.

[BSH08a] Lars Bauer, Muhammad Shafique, and Jörg Henkel. "A "=Computation- and "=Communication- Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor". In: *18th International Conference on Field Programmable Logic and Applications (FPL)*. 2008, pp. 203–208.

[BSH08b] Lars Bauer, Muhammad Shafique, and Jörg Henkel. "Run-time Instruction Set Selection in a Transmutable Embedded Processor". In: *Conference on Design Automation (DAC)*. 2008, pp. 56–61.

[BSH09] Lars Bauer, Muhammad Shafique, and Jörg Henkel. "MinDeg: A Performance-guided Replacement Policy for Run-time Reconfigurable Accelerators". In: *IEEE International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*. 2009, pp. 335–342.

[BSKH07] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. "RISPP: Rotating Instruction Set Processing Platform". In: *Conference on Design Automation (DAC)*. 2007, pp. 791–796.

[BSKH08] Lars Bauer, Muhammad Shafique, Stephanie Kreutz, and Jörg Henkel. "Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set". In: *Design, Automation and Test in Europe (DATE)*. 2008, pp. 752–757.

[CELM+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. "The Reconfigurable Streaming Vector Processor (RSVPTM)". In: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2003, pp. 141–150.

[CG13] Jason Cong and Karthik Gururaj. "Architecture Support for Custom Instructions with Memory Operations". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2013, pp. 231–234.

[Cla15] ByDon Clark. *Intel Rechisels the Tablet on Moore's Law*. http://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/. 2015.

[CM11] Liang Chen and T. Mitra. "Shared reconfigurable fabric for multi-core customization". In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2011, pp. 830–835.

[CMM12] Liang Chen, T. Marconi, and T. Mitra. "Online scheduling for multi-core shared reconfigurable fabric". In: *Design, Automation and Test in Europe (DATE)*. 2012, pp. 582–585.

[Dal03] Michael Dales. "Managing a reconfigurable processor in a general purpose workstation environment". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2003, pp. 980–985.

[DP06] Klaus Danne and Marco Platzner. "An EDF Schedulability Test for Periodic Tasks on Reconfigurable Hardware Devices". In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*. 2006, pp. 93–102.

[EL09] Andreas Ehliar and Dake Liu. "An ASIC Perspective on FPGA Optimizations". In: *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications (FPL)*. 2009, pp. 218–223.

[EZc15] EZchip Semiconductor Ltd. *TILE-Gx72 Processor Product Brief*. `http://www.tilera.com/files/drim__TILE-Gx8072_PB041-04_WEB_7683.pdf`. 2015.

[GB11] Carlo Galuzzi and Koen Bertels. "The Instruction-Set Extension Problem: A Survey". In: *ACM Trans. Reconfigurable Technol. Syst.* 4.2 (2011), 18:1–18:28.

[GBH12] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "Partial Online-Synthesis for Mixed-Grained Reconfigurable Architectures". In: *Design Automation and Test in Europe Conference (DATE)*. 2012, pp. 1555–1560.

[GCSB+06] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. "An overview of reconfigurable hardware in embedded systems". In: *EURASIP Journal on Embedded Systems* 2006 (1 2006), pp. 1–19.

[GHPB08] D. Gohringer, M. Hubner, T. Perschke, and J. Becker. "New dimensions for multiprocessor architectures: On demand heterogeneity, infrastructure and performance through reconfigurability - the RAMPSoC approach". In: *International Conference on Field Programmable Logic and Applications*. 2008, pp. 495–498.

[GHSB08] D. Göhringer, M. Hübner, V. Schatz, and J. Becker. "Runtime adaptive multiprocessor system-on-chip: RAMPSoC". In: *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)* (2008), pp. 1–7.

[GMP15] E. Giaquinta, A. Mishra, and L. Pozzi. "Maximum Convex Subgraphs Under I/O Constraint for Automatic Identification of Custom Instructions". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.3 (2015), pp. 483–494.

[GPYB+06] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels, and Stamatis Vassiliadis. "Automatic selection of application-specific instruction-set extensions". In: *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. 2006, pp. 160–165.

[GREA+01] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *International Workshop on Workload Characterization*. 2001, pp. 3–14.

[Hei14] Jan Heißwolf. "A Scalable and Adaptive Network on Chip for Many-Core Architectures". PhD thesis. Department of Electrical Engineering and Information Technology, Karlsruhe Institute of Technology, 2014.

[HHBW+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R.K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and Kobbe S. "Invasive Manycore Architectures". In: *16th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Invited Paper for the Special Session "Design and Prototyping of Invasive MPSoC Architectures". 2012, pp. 193–200.

[HM09a] Huynh Phung Huynh and Tulika Mitra. "Runtime Adaptive Extensible Embedded Processors – A Survey". In: *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2009, pp. 215–225.

[HM09b] Huynh Phung Huynh and Tulika Mitra. "Runtime reconfiguration of custom instructions for real-time embedded systems". In: *Design, automation and test in Europe (DATE)*. 2009, pp. 1536–1541.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HSM08] Huynh Phung Huynh, Joon Edward Sim, and Tulika Mitra. "An efficient framework for dynamic reconfiguration of instruction-set customization". In: *Design Automation for Embedded Systems* 13.1-2 (2008), pp. 91–113.

[HV09] Chen Huang and F. Vahid. "Transmuting coprocessors: Dynamic loading of FPGA coprocessors". In: *46th ACM/IEEE Design Automation Conference*. 2009, pp. 848–851.

[HYMN+09] Wei Han, Ying Yi, M. Muir, I. Nousias, T. Arslan, and A.T. Erdogan. "Multicore Architectures With Dynamically Reconfigurable Array Processors for Wireless Broadband Technologies". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.12 (2009), pp. 1830–1843.

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors". In: *International Symposium on Computer Architecture (ISCA)*. 2007, pp. 186–197.

[ITR11] ITRS. *International Technology Roadmap for Semiconductors – 2011 Edition – System Drivers*. http://www.itrs.net/ITRS%201999-2014%20Mtgs,%20Presentations%20&%20Links/2011ITRS/Home2011.htm. 2011.

[KBSS+10] Ralf König, Lars Bauer, Timo Stripf, Muhammad Shafique, Waheed Ahmed, Juergen Becker, and Jörg Henkel. "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture". In: *Design, Automation and Test in Europe (DATE)*. 2010, pp. 819–824.

[KCCK+12] Changmoo Kim, Mookyoung Chung, Yeongon Cho, M. Konijnenburg, Soojung Ryu, and Jeongwook Kim. "ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications". In: *2012 International Conference on Field-Programmable Technology (FPT)*. 2012, pp. 329–334.

[KD06] Shannon Koh and Oliver Diessel. "Communications Infrastructure Generation for Modular FPGA Reconfiguration". In: *IEEE International Conference on Field Programmable Technology*. 2006, pp. 321–324.

[KMN02] Kurt Keutzer, Sharad Malik, and A. Richard Newton. "From ASIC to ASIP: The Next Design Discontinuity". In: *International Conference on Computer Design (ICCD)*. 2002, pp. 84–90.

[Kob15] Sebastian Kobbe. "Scalable and Distributed Resource Management for Many-Core Systems". PhD thesis. Chair for Embedded Systems, Computer Science Department, Karlsruhe Institute of Technology, 2015.

[LC06] Abelardo Lopez-Lagunas and Sek M. Chai. "Compiler Manipulation of Stream Descriptors for Data Access Optimization". In: *Proceedings of the International Conference Workshops on Parallel Processing (ICPPW)*. 2006, pp. 337–344.

[LCBM+06] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri. "XiSystem: a XiRisc-based SoC with reconfigurable IO module". In: *IEEE Journal of Solid-State Circuits (JSSC)* 41.1 (2006), pp. 85–96.

[LEP13] Adrian Lifa, Petru Eles, and Zebo Peng. "Dynamic Configuration Prefetching Based on Piecewise Linear Prediction". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. 2013, pp. 815–820.

[LEP15] A. Lifa, P. Eles, and Z. Peng. "A Reconfigurable Framework for Performance Enhancement with Dynamic FPGA Configuration Prefetching". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP.99 (2015).

[Leu04] Joseph Y.-T. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. 2004.

[LH02] Z. Li and S. Hauck. "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation". In: *Proceedings of 8th international symposium on Field Programmable Gate Arrays (FPGA)*. 2002, pp. 187–195.

[LMBG09] Yi Lu, Thomas Marconi, Koen Bertels, and Georgi Gaydadjiev. "Online Task Scheduling for the FPGA-Based Partially Reconfigurable Systems". In: *ARC*. 2009, pp. 216–230.

[LP07] Enno Lübbers and Marco Platzner. "ReconOS: An RTOS supporting Hard- and Software Threads". In: *Field Programmable Logic and Applications (FPL)*. 2007, pp. 441–446.

[LP09] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded programming for reconfigurable computers". In: *ACM Transactions on Embedded Computing Systems (TECS)* 9 (1 2009), 8:1–8:33.

[LPM97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems". In: *International Symposium on Microarchitecture (MICRO)*. 1997, pp. 330–335.

[LSC09] Siew-Kei Lam, T. Srikanthan, and C.T. Clarke. "Selecting Profitable Custom Instructions for Area-Time-Efficient Realization on Reconfigurable Architectures". In: *IEEE Transactions on Industrial Electronics* 56.10 (2009), pp. 3998–4005.

[LSKK+11] Sangjo Lee, Joonho Song, Minsoo Kim, Dohyung Kim, and Shihwa Lee. "H.264/AVC UHD decoder implementation on multi-cluster platform using hybrid parallelization method". In: *2011 18th IEEE International Conference on Image Processing (ICIP)*. 2011, pp. 381–384.

[LSV06] Roman Lysecky, Greg Stitt, and Frank Vahid. "Warp Processors". In: *Transactions on Design Automation of Electronic Systems (TODAES)* 11.3 (2006), pp. 659–681.

[LTC03] Andrea Lodi, Mario Toma, and Fabio Campi. "A Pipelined Configurable Gate Array for Embedded Processors". In: *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*. 2003, pp. 21–30.

[LTCC+03] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. "A VLIW processor with reconfigurable instruction set for embedded applications". In: *IEEE Journal of Solid-State Circuits (JSSC)* 38.11 (2003), pp. 1876–1886.

[LWB14] M.J. Lyons, Gu-Yeon Wei, and D. Brooks. "Multi-accelerator system development with the ShrinkFit acceleration framework". In: *2014 32nd IEEE International Conference on Computer Design (ICCD)*. 2014, pp. 75–82.

[MAA15] Sen Ma, Zeyad Aklah, and David Andrews. "A run time interpretation approach for creating custom accelerators". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–4.

[MBHK+12] Shravan Muddasani, Srinivas Boppu, Frank Hannig, Boris Kuzmin, Vahid Lari, and Jürgen Teich. "A Prototype of an Invasive Tightly-Coupled Processor Array". In: *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 23–Oct. 25, 2012, pp. 393–394.

[MBSA05] N. Moreano, E. Borin, Cid de Souza, and G. Araujo. "Efficient datapath merging for partially reconfigurable architectures". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.7 (2005), pp. 969–980.

[MBTC06] C. Mucci, M. Bocchi, M. Toma, and F. Campi. "A case-study on multimedia applications for the XiRisc reconfigurable processor". In: *International Symposium on Circuits and Systems (ISCAS)*. 2006, pp. 4859–4862.

[MGMB+13] Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, and Jörg Henkel. "Hardware Acceleration for Programs in SSA Form". In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2013.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994.

[Mit15] Tulika Mitra. "Heterogeneous Multi-core Architectures". In: *IPSJ Transactions on System LSI Design Methodology* 8 (2015), pp. 51–62.

[MNMB+04] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. "Run-time Support for Heterogeneous Multitasking on Reconfigurable SoCs". In: *Integration, the VLSI Journal* 38.1 (2004), pp. 107–130.

[Moo98] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Proceedings of the IEEE*. Vol. 18. 1. reprint. 1998.

[MSPZ+13] G. Mariani, V.-M. Sima, G. Palermo, V. Zaccaria, G. Marchiori, C. Silvano, and K. Bertels. "Run-time optimization of a dynamically reconfigurable embedded system through performance prediction". In: *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)*. 2013, pp. 1–8.

[MVVM+03] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". In: *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*. 2003, pp. 61–70.

[NK14] T.D.A. Nguyen and A. Kumar. "PR-HMPSoC: A versatile partially reconfigurable heterogeneous Multiprocessor System-on-Chip for dynamic FPGA-based embedded systems". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–6.

[NMBV+03] V. Nollet, J-y. Mignolet, T. A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. "Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems". In: *Engineering Reconfigurable Systems and Algorithms (ERSA)*. 2003, pp. 81–87.

[NSBN08] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll. "Design flow for embedded FPGAs based on a flexible architecture template". In: *Proceedings of the Conference on Design, automation and test in Europe (DATE)*. 2008, pp. 56–61.

[NSMK11] Tomoyuki Nagatsuka, Yoshito Sakaguchi, Takayuki Matsumura, and Kenji Kise. "CoreSymphony: An Efficient Reconfigurable Multi-core Architecture". In: *SIGARCH Comput. Archit. News* 39.4 (2011), pp. 32–37.

[OSKB+11] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "OctoPOS: A Parallel Operating System for Invasive Computing". In: *EuroSys 2011 Workshop on Systems for Future Multi-Core Architectures (SFMA)*. 2011.

[PAAS+06] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews. "Hthreads: A Computational Model for Reconfigurable Devices". In: *FPL*. 2006, pp. 1–4.

[PH11] R. Panda and S. Hauck. "Dynamic Communication in a Coarse Grained Reconfigurable Array". In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2011, pp. 25–28.

[PK89] P.G. Paulin and J.P. Knight. "Force-directed scheduling for the behavioral synthesis of ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1989, pp. 661–679.

[PM12] Mihai Pricopi and Tulika Mitra. "Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (2012), 22:1–22:21.

[PSOE+15] Johny Paul, Walter Stechele, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, Wolfgang Schröder-Preikschat, Manfred Kröhnert, Tamim Asfour, Ericles Sousa, Vahid Lari, Frank Hannig, Jürgen Teich, Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. "Resource Awareness on Heterogeneous MPSoCs for Image Processing". In: *Journal of Systems Architecture*. Vol. 61. 10. 2015, pp. 668–680.

[Pur10] Madhura Purnaprajna. "Run-time reconfigurable multiprocessors". PhD thesis. Paderborn, Univ., 2010. URL: `http://digital.ub.uni-paderborn.de/hs/content/titleinfo/1446`.

[RASS+08] Vincenzo Rana, David Atienza, Marco Domenico Santambrogio, Donatella Sciuto, and Giovanni De Micheli. "A Reconfigurable Network-on-Chip Architecture for Optimal Multi-Processor SoC Communication". In: *VLSI-SoC: Design Methodologies for SoC and SiP*. 313. 2008, pp. 232–250.

[RBC11] Mateus B. Rutzig, Antonio Carlos S. Beck, and Luigi Carro. "CReAMS: An Embedded Multiprocessor Platform". In: *Reconfigurable Computing: Architectures, Tools and Applications*. 2011, pp. 118–124.

[RC12] M. Reshadi and C. Cascaval. "Multidimensional dynamic behavior in mobile computing". In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 2012, pp. 113–115.

[RMAS+09] Vincenzo Rana, Srinivasan Murali, David Atienza, Marco Domenico Santambrogio, Luca Benini, and Donatella Sciuto. "Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems". In: *CODES+ISSS*. 2009, pp. 325–334.

[SBAH11] Muhammad Shafique, Lars Bauer, Waheed Ahmed, and Jörg Henkel. "Minority-Game-based Reconfigurable Fabric Resource Allocation for Run-Time Reconfigurable Multi-Core Processors". In: *Design, Automation and Test in Europe (DATE)*. 2011, pp. 1261–1266.

[SBH10] Muhammad Shafique, Lars Bauer, and Jörg Henkel. "Selective Instruction Set Muting for Energy-Aware Adaptive Processors". In: *28th International Conference on Computer-Aided Design (ICCAD)*. 2010, pp. 353–360.

[SBH14] M. Shafique, L. Bauer, and J. Henkel. "Adaptive Energy Management for Dynamically Reconfigurable Processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.1 (2014), pp. 50–63.

[SNBN06] T. von Sydow, B. Neumann, H. Blume, and T. G. Noll. "Quantitative Analysis of Embedded FPGA-Architectures for Arithmetic". In: *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2006, pp. 125–131.

[SPA] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*. `http://www.sparc.org/specificationsDocuments.html#V8`.

[SSB09] M. Sabeghi, V.-M. Sima, and K. Bertels. “Compiler assisted runtime task scheduling on a reconfigurable computer”. In: *Field Programmable Logic and Applications (FPL)*. 2009, pp. 44–50.

[Str] Stretch Inc. *S6000 Family Software Configurable Processors*. `http://www.stretchinc.com/products/s6000.php`.

[Str13] Timo Stripf. “Softwareframework für Prozessoren mit variablen Befehlssatzarchitekturen”. PhD thesis. Department of Electrical Engineering and Information Technology, Karlsruhe Institute of Technology, 2013.

[SVH13] Muhammad Shafique, Benjamin Vogel, and Jörg Henkel. “Self-Adaptive Hybrid Dynamic Power Management for Many-Core Systems”. In: *Proceedings of Design, Automation and Test in Europe Conference (DATE)*. 18–Mar. 22, 2013, pp. 51–56.

[TCWM+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. “Reconfigurable computing: architectures and design methods”. In: *IEE Proceedings Computers & Digital Techniques* 152.2 (2005), pp. 193–207.

[THHS+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. “Invasive Computing: An Overview”. In: *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*. 2011, pp. 241–268.

[TKBP+07] 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”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2007, pp. 409–414.

[TPD15] R. Tessier, K. Pocek, and A. DeHon. “Reconfigurable Computing Architectures”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 332–354.

[TRC11] Hsiang-Kuo Tang, P. Ramanathan, and K. Compton. “Combining Hard Periodic and Soft Aperiodic Real-Time Task Scheduling on Heterogeneous Compute Resources”. In: *International Conference on Parallel Processing (ICPP)*. 2011, pp. 753–762.

[UMKU06] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. “Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling”. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. 2006.

[VS07] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing*. 2007.

[VWGB+04] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. “The MOLEN polymorphic processor”. In: *Transactions on Computers (TC)* 53.11 (2004), pp. 1363–1375.

[WA10] M.A. Watkins and D.H. Albonesi. “ReMAP: A Reconfigurable Heterogeneous Multicore Architecture”. In: *International Symposium on Microarchitecture (MICRO)*. 2010, pp. 497–508.

[WMGR+10] Jason Williams, Chris Massie, Alan D. George, Justin Richardson, Kunal Gosrani, and Herman Lam. “Characterization of Fixed and Reconfigurable Multi-Core Devices for Application Acceleration”. In: *ACM Trans. Reconfigurable Technol. Syst.* 3.4 (2010), 19:1–19:29.

[WSP03] H. Walder, C. Steiger, and M. Platzner. “Fast online task placement on FPGAs: free space partitioning and 2D-hashing”. In: *Reconfigurable Architectures Worshop*. 2003, pp. 178–185.

[Xil] Xilinx, Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`.

[Xil08] Xilinx, Inc. *Spartan and Spartan-XL FPGA Families Data Sheet, v1.8*. `http://www.xilinx.com/support/documentation/data_sheets/ds060.pdf`. 2008.

[Xil12] Xilinx Inc. *Virtex-5 FPGA User Guide, v5.4*. `www.xilinx.com/support/documentation/user_guides/ug190.pdf`. 2012.

[Xil13] Xilinx, Inc. *Zynq-7000 All Programmable SoC Overview*. `http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`. 2013.

[YWWZ+10] Like Yan, Binbin Wu, Yuan Wen, Shaobin Zhang, and Tianzhou Chen. "A Reconfigurable Processor Architecture Combining Multi-core and Reconfigurable Processing Unit". In: *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*. 2010, pp. 2897–2902.

[ZKG09] Pavel G. Zaykov, Georgi K. Kuzmanov, and Georgi N. Gaydadjiev. "Reconfigurable Multithreading Architectures: A Survey". In: *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2009, pp. 263–274.