Prototyping Methodologies and Design of Communication-centric Heterogeneous Many-core Architectures

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (Dr.-Ing.)

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik,
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Dipl.-Inform. Leonard Jannik Masing
geb. in Heidelberg

Tag der mündlichen Prüfung: 5.3.2020

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr. sc.techn. Andreas Herkersdorf

Abstract

The age of parallel processing and heterogeneous architectures is upon us. Fueled by the end of Dennard scaling, such architectures are the only way to keep the power wall in check until entirely new technologies such as quantum computing or new material breakthroughs might change the entire playing field. High performing heterogeneous many-cores interfaced by scalable networks-on-chips are one promising approach that has emerged in this context. However, these architectures come in many possible variations as the manifold application domains, from embedded and Internet of Things(IoT) to high performance computing, make customized computing architectures desirable. This in turn raises design efforts and complexity. At the same time, design size, complex interconnects and heterogeneous computing nodes are further straining existing techniques and methodologies for debugging, verification and validation.

Inspired by these developments, this work presents novel approaches for design and prototyping of heterogeneous many-core architectures. Three main aspects, not covered sufficiently by the existing state of the art are targeted. These aspects encompass early software development, hardware verification and design automation. A specific scenario that relates to a part in the many-core design process motivates each aspect. An emphasis is put on a novel networks-on-chip extension for low latency interconnects that highlights the major limitation of existing approaches: scalability for large architectures. As a central concept, virtual platforms on the electronic system level are utilized for early software development and verification tasks due to their binary compatibility with the target architecture. This allows for software, low-level function calls, drivers and operating system functionality to be developed in parallel to hardware design and verification tasks. A virtual platform environment is used to model a many-core architecture by extending it towards support for heterogeneous elements, an abstracted interconnect as well as camera and video I/O. Virtual platforms play another crucial role as part of the novel multi-level hybrid prototyping methodology that is introduced to provide a scalable answer for hardware verification tasks. In this approach, a prototype is built consisting of an FPGA part and a virtual part, interfaced by high speed PCIe. It reduces synthesis times and enables prototyping of large architectures that would otherwise not fit on available FPGA-board prototypes. Finally, the hardware/software codesign and design automation are also considered. Specifically, a framework that improves High-Level Synthesis (HLS) flows is introduced. It starts from OpenCL input and automatically generates a virtual platform representation as an intermediate step. This saves costly synthesis time and allows for optimizations on a human readable level in SystemC, as opposed to unintelligible RTL-Code generated by common HLS tools. The framework is enhanced further by providing an automated conversion among real number representations. This data representation forms a major trade-off in the design of customized accelerators, having significant impact on resource and power consumption. In summary, the presented contributions significantly speed up design processes and enable scalable prototyping of large heterogeneous many-core architectures that was not possible before.

Zusammenfassung

Die Ära der Parallelverarbeitung und des heterogenen Rechnens ist angebrochen. Bedingt durch das Ende des Dennard Scalings sind solche Architekturen die einzige Möglichkeit um Leistungssteigerungen trotz der Limitierungen durch die Energiedichte zu erreichen, bis völlig neue Technologien wie das Quantenrechnen oder neue Materialien diese Hürde möglicherweise überwinden. Hochperformante, heterogene Vielkernarchitekturen die durch ein Netzwerk auf einem Chip verbunden sind, stellen einen vielversprechenden Ansatz in diesem Kontext dar. Diese Architekturen existieren jedoch in sehr unterschiedlichen Variationen, um den vielfältigen Anwendungsgebieten wie beispielsweise eingebettete Systeme, das Internet der Dinge sowie das Hochperformanzrechnen gerecht zu werden. Dies wiederum erhöht die Komplexität im Entwurf und der Verifikation. Gleichzeitig überfordern die Designgröße, neuartige Verbindungsinfrastrukturen und die Integration heterogener Rechenelemente zusätzlich die bestehenden Techniken und Methodiken für das Debugging, die Verifikation und die Validierung.

Inspiriert durch diese Entwicklungen werden in der vorliegenden Arbeit neuartige Ansätze für das Design und das Prototyping von heterogenen Vielkernarchitekturen vorgestellt. Der Fokus liegt hierbei auf drei Hauptaspekten, welche bisher im bestehenden Stand der Technik nicht ausreichend betrachtet worden sind. Diese Aspekte umfassen die frühzeitige Softwareentwicklung parallel zur Architekturentwicklung, die Hardwareverifikation und die Designautomatisierung. Jeder dieser Aspekte wird durch ein spezielles Szenario aus der Entwicklung einer Vielkernarchitektur motiviert. Insbesondere wird hier eine Erweiterung der Kommunikationsinfrastruktur hervorgehoben, welche eine Latenzreduzierung ermöglicht. Dieser neue Beitrag zur Leistungssteigerung von Vielkernarchitekturen zeigt ein deutliches Problem bestehender Entwurfs- und Verifikationstechniken: Die Skalierbarkeit für große Architekturen. Als zentrales Konzept werden in dieser Arbeit virtuelle Plattformen auf dem Elektronischen System Level (ESL) eingesetzt. Diese ermöglichen die frühzeitige

Entwicklung und Verifikation von Software dank ihrer Binärkompatibilität mit der Zielarchitektur. Auf diese Weise kann selbst hardwarenahe Software, wie Treiber oder betriebssystemspezifische Funktionen, entwickelt werden während ein fertiges Hardwaredesign noch nicht existiert. In dieser Arbeit wird konkret eine Umgebung für virtuelle Plattformen speziell für die Modellierung von heterogenen Vielkernarchitekturen, eine abstrahierte Verbindungsstruktur sowie Kamera und Video Eingabe/Ausgabe erweitert. Virtuelle Plattformen spielen ebenso eine entscheidende Rolle als Teil der neuartigen hybriden Prototyping Methodik die eingeführt wird um eine skalierbare Lösung für die Hardwareverifikation zu liefern. Dieser Ansatz sieht vor, den Entwurfsprozess von einer virtuellen Plattform ausgehen zu lassen und Teile der Architektur auf einen FPGA zu verlagern. Die virtuelle Plattform wird über hochperformantes PCIe angebunden um eine niedrige Latenz und hohen Durchsatz zu ermöglichen. Der hybride Ansatz reduziert die Synthesezeit und erlaubt die Erstellung von Prototypen für große Architekturen, welche ansonsten derzeit nicht auf reine FPGA-Lösungen passen würden. Als weiteres großes Thema in der vorliegenden Arbeit wird das Hardware/Software Codesign und die Designautomatisierung für heterogene Vielkernarchitekturen betrachtet. Insbesondere wird ein Framework vorgestellt, welches den High-Level Synthese Entwurfsprozess verbessert. Dieser startet auf C Ebene mit OpenCL Quellcode und generiert automatisiert eine virtuelle Plattform als Zwischenschritt, im Gegensatz zur direkten Synthese auf Register Transfer (RT) Ebene bei bisherigen Ansätzen. Dies erspart kostbare Synthesezeit und ermöglicht es, Optimierungen in der gut verständlichen und übersichtlichen Zwischenebene vorzunehmen, anstatt des unleserlichen Codes der auf RT Ebene erzeugt wird. Das Framework wird zudem erweitert durch ein Feature zur automatisierten Konvertierung von reellen Zahlen. Dies ermöglicht die Evaluierung von Tradeoffs für heterogene Beschleuniger im Sinne des approximativen Rechnens. Zusammengefasst beschleunigen die vorgestellten Beiträge den Designprozess signifikant und ermöglichen das skalierbare Prototyping von großen heterogenen Vielkernarchitekturen, was bisher nicht in dieser Form möglich war.

Vorwort

Die vorliegende Doktorarbeit ist in meiner Zeit am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Instituts für Technologie (KIT) entstanden. An dieser Stelle möchte ich mich ganz herzlich bei all jenen bedanken, die mich in dieser Zeit auf meinem Weg begleitet und unterstützt haben.

Mein ganz besonderer Dank gilt zunächst meinem Doktorvater Prof. Jürgen Becker für die Möglichkeit bei ihm am ITIV zu promovieren. Ich bin dankbar für die inhaltlichen Diskussionen, die Freiheit eigenen Ideen folgen zu können und alles was ich auf diesem Weg von ihm lernen durfte. Ebenfalls gilt mein besonderer Dank Prof. Andreas Herkersdorf für die Übernahme des Koreferats, den inhaltlichen Austausch und die gute Zusammenarbeit mit ihm und seinem Team über die Jahre. Nicht vergessen möchte ich auch meine Prüfungskommission, bestehend aus Prof. Ivan Peric, Prof. Laurent Schmalen und Prof. Ahmet Cagri Ulusoy, welchen ich für Ihre Zeit danken möchte.

Mein weiterer Dank gilt insbesondere allen die mich auf meinem Weg begleitet haben. Dies sind zunächst einmal meine ehemaligen und aktuellen Kollegen vom ITIV, mit denen ich spannende Diskussionen, eine gute Zusammenarbeit in Projekten, Lehre, etc. hatte und die mir auch darüber hinaus eine tolle Zeit am Institut ermöglicht haben. Genauso danke ich allen Partnern von anderen Institutionen aus den vielen Projekten an denen ich beteiligt war für den angeregten und guten Austausch. Auch all meinen Studenten bin ich sehr dankbar, denen ich nicht nur etwas beibringen durfte sondern die auch andersherum mich immer wieder auf neue Ideen oder Erkenntnisse gebracht haben. Nicht zu vergessen sind auch alle weiteren Mitarbeiter aus der Verwaltung und insbesondere dem Sekretariat, die immer unkompliziert bei Problemen geholfen haben. Ohne all diese Leute wäre die vorliegende Arbeit nicht möglich gewesen.

Zu guter Letzt gilt mein spezieller Dank meiner Familie, meinen Eltern und meinem Bruder, die stets an meiner Arbeit interessiert waren und mich immer unterstützt haben.

Karlsruhe, im September 2020
Leonard Masing

Contents

1 Introduction

For many years a steady increase in raw processing power of computer architectures has been observed. Figure 1.1 highlights this development in relation to a VAX11-780 machine. For the most time, this increase followed the principle set by Gordon Moore that transistor counts integrated onto a single chip would double roughly every one and a half year [87]. Increasing transistor

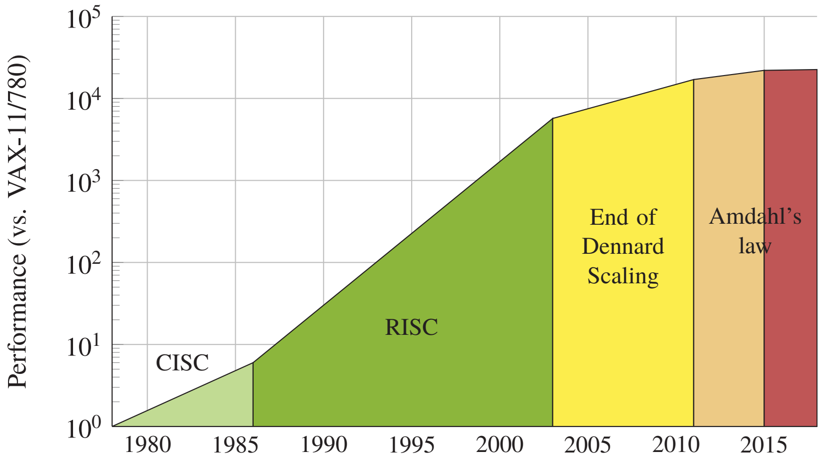


Figure 1.1: 40 years of processor performance [58]

counts was mostly achieved by shrinking the circuitry to ever smaller feature sizes. Smaller feature sizes in turn meant that wire and transistor switching delays could be cut down which, together with deeper pipelines and the move towards Reduced Instruction Set Computers (RISC), helped to increase clock rates and subsequently performance. The transistor shrinking had many positive side-effects, most notably reduced supply voltage required to operate the circuits leading to reduced power consumption. This was also labeled Dennard scaling which claims that power density is constant as transistors shrink since the increased number of transistors per area is offset by lower voltage and current required to operate them [37]. Average power dissipation is calculated according to the following equation [114]:

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{static}$$

$$P_{avg} = \alpha C_L V_{dd}^2 f + I_{sc} V_{dd} + I_{leakage} V_{dd}$$

The dynamic component of power in this equation is $P_{switching}$, which calculates according to the node transition activity factor α, the load capacitance C_L, the supply voltage V_{dd} and the frequency f. Since transistor shrinking allowed lower V_{dd} that is squared in this equation and also reduced C_L, increases to frequency were possible without the risk of rising power density. The second component $P_{short-circuit}$ emerges when both the NMOS and PMOS transistors are simultaneously active. The third component in the equation above has been neglected for a long time. Static leakage power P_{static} results from multiple sources that had rather insignificant impact in larger semiconductor technology nodes but have become a fundamental problem in the latest nodes. With shrinking transistor size, static power dissipation overtakes dynamic power dissipation and will start to dominate the overall impact on power density in the future. On scales below 10nm, physical effects appear or amplify to a degree that they can not safely be ignored anymore. In order to keep a circuit stable when quantum effects start to cause problems, voltage cannot be reduced the same as before on new technology nodes. This amplifies the power problem as the shrinking process continues. The only available answers to this development are either limiting or even reducing clock frequencies and clock/power gating circuits for certain intervals. At the horizon, entirely novel concepts such as quantum computing can be seen that might overcome these issues and propel the performance of processing towards much higher levels again. Yet today no one can say for sure when such a technology will be ready for commercial use and it is even disputed whether quantum computing can replace general purpose processing at all or only fill in a very specific computing niche [98]. There is also hope that breakthroughs on topics such as new materials, optic networks/processing and 3D-stacking will improve the situation yet the outcome is still unclear and often results are not expected to be seen in the near future [107].

In the light of this situation, developments on the architectural level need to provide solutions for further performance improvements. The problem of power densities has been observed even before the more recent difficulties on smaller feature sizes, as frequencies have plateaued at around 3GHz since the year 2002 when the first commercial processor achieved this mark [34]. The approach since then is clear: Limit clock rates and improve performance via more parallelism. This lead to the beginning of the multi- and many-core era [19]. Yet according to Amdahl, parallelism can only improve the fraction of an application that can be parallelized [8]. Furthermore, even when limiting clock rates, the aforementioned difficulties when scaling transistors in recent technology nodes still move circuits closer towards the power-wall. This results in a situation nowadays where only a fraction of a chip can be switched at the same time since otherwise the chip would melt. This phenomenon is labeled dark-silicon, referring to the fact that at all times some parts of a chip need to remain ”dark”, i.e. not active and powered. All these developments substantially increase the need for efficient data processing which can be achieved by very customized and specialized architectures - tailored towards a single task. These specialized optimizations must not necessarily target more performance as some approaches value power efficiency as most important metric nowadays [125]. These heterogeneous architectures are not improved on a technology level but instead by clever optimization towards their intended task. Consequently, progress must come from the architectural level since it can not be provided by technology scaling anymore [41]. Adhering to the concept of dark silicon, designs are not only becoming more parallel, but also more specialized and heterogeneous - only activating parts of the chip that can deal with a task in the most efficient manner.

1.1 Motivation

As far back as the late 90s, surveys discovered a development that was labeled the design productivity gap. As Figure 1.2 shows, design complexity due to more and more transistors integrated into a single chip was rising much faster than the productivity of individual design engineers. More than a decade later, it became obvious that the predictions of a design crisis never really came to pass. An analysis showed that several factors played a role, yet most importantly design reuse through Intellectual Property (IP) cores was

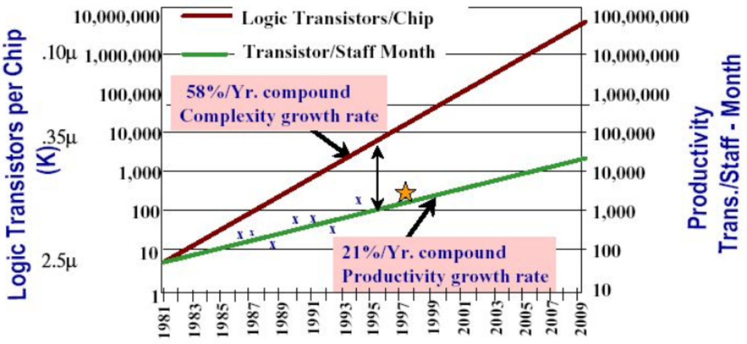


Figure 1.2: The design productivity gap. Source: SEMATECH.

able to keep complexity in check [45]. However, it was also shown that while design efforts could be kept almost at a stable level, verification efforts increased by a large degree, meaning the problem was never fully avoided but more shifted towards a verification problem. Furthermore, even IP reuse and improved design tools[1] are expected to reach their limits, resulting in the need for progress on a methodology level.

The design productivity gap and its solutions are still closely connected to the old ways of transistor scaling, homogeneous designs and general purpose processors. The IRDS report[2] acknowledges this history of semi-conductor industry but notes a shift in the semi-conductor world that occurred in recent years [61]. Information processing is nowadays omnipresent in fields such as mobile communication, server farms, cloud computing or the Internet of Things (IoT). Yet according to the IRDS, the driver for advances in processing power is more and more the application field itself, triggering novel architectures that are increasingly heterogeneous while continuously scaling up number of cores and parallel processing. The growing number of separate application domains together with the limitations and challenges of the power-wall men-

[1] Significant efforts have been spent on parallelization of tools in order to speed up their operation to cope with increasing design sizes

[2] The International Roadmap for Devices and Systems. The report serves as a follow up of the ITRS semiconductor roadmaps

tioned earlier have lead to a large diversity in architectural features and systems that are often tailored for certain fields of application. This growing diversity and the growing number of transistors and processor cores on a single chip have led to a steep increase in design complexity of computing architectures. Integrating multiple IP cores, handling HW/SW trade-offs and design space exploration, developing novel architecture extensions and application specific accelerators all need to be considered when realizing efficient heterogeneous multi- and many-core platforms which are needed to overcome the aforementioned limitations and challenges. Since building a new chip is very costly, designing an architecture that does not meet the requirements, that fails due to errors and bugs or does not solve its tasks efficiently can decide about success or failure in the market for a product or even a whole company. As such, methods and techniques that enable design space exploration, verification, debugging and system validation become more important than ever before. A major role in the initial and intermediate stages of design is played by the concept of prototyping. A prototype represents an early available, functional version of a design. It serves the purpose of steering development of the actual product by observing and evaluating the prototype. Prototyping has been a major driver that helped closing the productivity and verification gap that the rising complexity in circuit design has caused. Yet prototyping must also evolve in order to stay up to date with new developments and changes in the semi-conductor world towards future computing architectures.

The recent trends that affect the prototyping can be considered as a) the move towards multi- and successively many-cores b) the increasing degree of heterogeneity brought by custom architectures and accelerators. These trends formulate the basic question and set the theme of this work: How can such architectures be prototyped successfully? How do the established techniques and methods cope with these developments and will they have to evolve?

Since these questions relate to a huge field, this work will introduce some restrictions and limitations on its scope. Firstly, this work specifically targets the concept of prototyping as it is used for feature development, debugging, verification and validation tasks. Other approaches that try to achieve the same or similar goals by different means, such as formal verification, will not be part of this document. Secondly, some recent developments in the underlying technologies will not be covered, most notably the trend towards 3D stacking. This is mostly due to the fact that real 3D stacking (without an interposer or the like) is currently only feasible for memory chips while Integrated Circuits (ICs) struggle with the power-wall in 3D even more than in 2D. Furthermore, from the architectural perspective 3D does not contain new and interesting design trade-offs and will be constrained by the cooling method which needs to be developed first before considering it in a prototype makes any sense. Similarly, any future technology that is not within reach and is thus not yet well understood, needs their very own, specialized form of prototyping which can not be covered together with the more common and well understood technologies.

1.2 Goals

Based on the motivation, the fundamental goals of the work presented in this document are formulated. The major goals and thus the contribution of this work to the state of the art can be summed up as the following:

- Give an overview and analyze the current state of the art regarding computer architecture prototyping.
- Assess the major requirements for prototyping heterogeneous many-core and analyze the state of the art in regard to these new kind of architectures.
- Extend, adapt, enhance existing and develop novel prototyping concepts and design methodologies towards the major requirements posed by heterogeneous many-core.
- Develop and evaluate these new concepts and methodologies in close relation to novel heterogeneous many-core architecture designs and feature extensions.

Despite not being limited by it, this work shall focus on two aspects that affect performance of future computing architectures significantly: The interconnect, which requires entirely different solutions in a many-core, and the heterogeneous computing aspect which is tightly connected to large many-core or many-accelerator platforms. The presented work shall discuss hardware and software aspects, their codesign and the trend towards abstraction that is realized by increased levels of design automation.

1.3 Outline

The work is organized as follows. In chapter 2, fundamental terms, definitions, tools and techniques are presented that need to be understood in order to follow the later works and contributions. Most notably, the basics about many-core architectures will be introduced including a number of example architectures that are available as of today. Additionally, networks-on-chip, heterogeneous processing elements and a selection of design languages are discussed that will appear in later chapters. In chapter 3, state-of-the-art approaches for the prototyping of traditional computer architectures are introduced. This chapter will highlight recent research works and lay the groundwork for the following chapter 4, that analyses the requirements of a heterogeneous many-core towards prototyping methodologies and also discusses the applicability of the state-of-the-art techniques. The chapter also introduces three examples, highlighting challenges and novel techniques in heterogeneous many-core architectures, that motivate the need for the novel methodologies presented in this work. The individual contributions of this work are explored and explained in more detail in the following three chapters. The first of these in chapter 5 introduces the modeling of many-core architectures in virtual platforms and presents a number of novel extensions and improvements. The focus of this chapter lies on the prototyping of system software and applications in an early design stage. In contrast, chapter 6 introduces techniques that focus on prototyping approaches that enable test, debug and verification/validation tasks of many-core hardware components in a full system environment. This chapter is motivated specifically by a NoC extension for latency reduction in large many-core architectures that requires such a novel approach for enabling its verification. The third of these in chapter 7 deals with high-level design and Electronic Design Automation (EDA) tool supported methodologies for the hardware design that raises the abstraction level of design processes. Finally, the results of this work are summarized and the initial questions answered in a quick an concise way in the concluding chapter 8.

2 Fundamentals

This work presents prototyping approaches and methodologies for future heterogeneous many-core and many-accelerator architectures. In the following, basics for understanding the shape and needs of such architectures shall be laid out and a number of clarifications and definitions will be introduced. Among the topics discussed are many-core architectures which can be considered a focal topic of this work. Such architectures necessitate the employment of a Networks-on-Chip (NoC) since traditional bus-based interconnect schemes do not scale. This on-chip communication is one of the major factors which distinguishes many-core architectures from single- and multi-core execution or compute clusters. Consequently, this topic is also a major topic in this work and the fundamental concepts of NoCs will be introduced and discussed here as well. Afterwards, heterogeneous processing and accelerators will be discussed. These form the second big focal topic of this work and understanding of the basic concepts is required to follow the contributions later on. The topic is highly relevant since future architectures will move towards more custom computing and specialized accelerators integrated into a heterogeneous platform. Finally, some well established languages and tools will be described which were used in some of the contributions and have a direct impact on evaluations and results.

2.1 Many-core

A many-core architecture can broadly be defined as follows:

Definition: *A many-core system is a computing architecture that tightly integrates a large number of general purpose processor cores into a single chip*

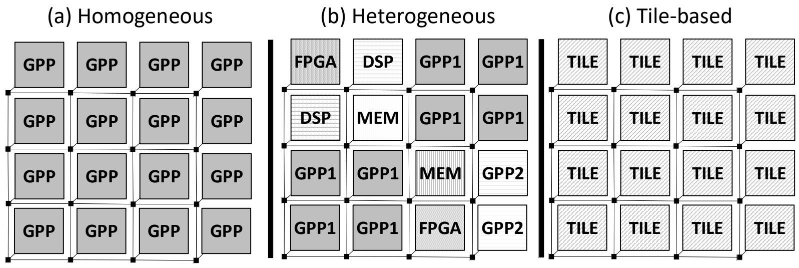


Figure 2.1: Conceptual layout of many-cores. From left to right: (a) homogeneous, (b) heterogeneous and (c) tile-based many-core.

At first glance there seem to be many similarities with multi-cores that integrate a much smaller number of cores. However, a deeper look shows that by increasing the number of cores, many new challenges arise and entirely different architectural solutions are required as the interconnect and memory quickly become a bottleneck. Many-cores have some similarities but also a big differentiation with SIMD-based (Single Instruction Multiple Data) processing that units such as GPU (Graphics Processing Unit) provide: A many-core tightly integrates full scale General-Purpose Processor (GPP) cores instead of simplified minimal cores or functional units that make up the shader cores in a GPU. Thus, a many-core inherits the strength of GPPs for handling control flow dominated tasks. Both Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) based GPP variants are possible. Despite a general trend towards RISC architectures that ease the hardware design, recent studies show that both GPP variants can provide good performance/power ratios and may thus be included in a many-core design [17]. If an architecture contains more than one kind of GPP core, i.e. cores with differing Instruction Set Architecture (ISA), it is called heterogeneous. The differentiation becomes less clear regarding other, non-GPP processing elements that may be part of a many-core. In general, common components, such as Direct Memory Access (DMA) or ISA extensions that are directly integrated into all GPPs or tiles of the system, are not considered as heterogeneous elements. In contrast, if the many-core contains either Digital Signal Processors (DSP), a reconfigurable fabric or standalone computing accelerators in only some parts of the architecture, it can also be considered as heterogeneous.

Based on this broad definition, a large number of variants can be considered as many-cores. Consequently, a number of categorizations and differentiations exist. One such differentiation is according to the alignment, clustering and interfacing of processing elements. A conceptual view of three layouts is highlighted in Figure 2.1. In (a), a homogeneous many-core is shown. The GPP are aligned in a 2D mesh topology, each connected to a global interconnect. Memory and I/O are often at the borders of such a design. In (b), a heterogeneous variant is shown that contains other computing units such as FPGA and DSP or different kinds of GPP. Since memory takes up large amounts of space and close integration with compute logic can be difficult, only small SRAM-based memories can be found within the tiles or close to the cores. However, recent advances in 3D stacking technology also allow the placement of large DRAM-based memories close to logic. Stacking memory on top of a GPP has been investigated, resulting in improved performance due to faster memory access times [75]. Consequently, a memory tile in a many-core could host a memory controller and a network interface, stacking the memory cells in 3D for lower access times. Finally in (c), a many-core is shown that employs so called tiles. These are clusters of compute logic or memory that introduce a level of hierarchy. They consist of multiple processing elements and often contain additional components that are shared among the cores within a tile. The components are interfaced either by a sublayer of a NoC or use more traditional bus-based interconnects. Since a tile typically does not host a large amount of processing nodes, buses are still a valid option on this level of the hierarchy. The tiles may access the global network via a common network interface, yet communication within a tile and accesses to tile-local memories are typically much faster. While a many-core does not need to be aligned in a meshed fashion as presented in the examples above, it eases the hardware design of the interconnect and the placement of components.

A common and essential feature of a many-core are on-chip networks. This kind of interconnect scheme is indispensable since more traditional bus-based or crossbar interconnects do not scale well - an essential property that is required when interfacing a large number of nodes. Since the interconnect plays a major role in the design of a many-core, it will be highlighted in a separate section later on.

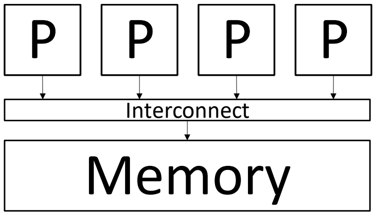


Figure 2.2: Memory access in an architecture with shared memory.

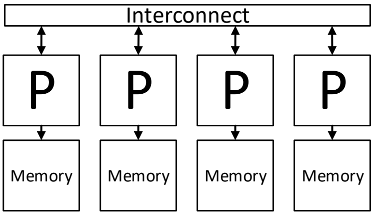


Figure 2.3: Memory access in an architecture with distributed memory.

2.1.1 Shared Memory vs Distributed Memory

A major distinction among many-core architectures can be made according to the memory model they employ. The major variants will be discussed in the following. Traditional single-core architectures provide one large block of main memory with a singular address range. In a multi- or many-core however, the main memory may be physically divided into several blocks, mostly due to space limitations and placement restrictions on a chip. Similarly, memories may also be split on a logical level into multiple individual entities so that cores or tiles may have only limited access to a restricted subset of the architecture. Conceptually, physical and logical separation are independent of each other, meaning a logically combined memory can operate on an underlying separated physical memory and vice versa. The terms shared memory and distributed memory refer to the logical separation of memories, each having their advantages and disadvantages [11].

The concepts are illustrated by two comparative figures. In Figure 2.2, a shared memory architecture is shown, where all processing elements have a direct or indirect access via an interconnect to the same memory. Communication among Processing Elements (PEs) is handled directly via memory, by operating on the same memory locations. Synchronization primitives such as semaphores, barriers and locks need to be employed to guarantee deterministic behavior.

In Figure 2.3, a distributed memory architecture is shown. In this case, each processing element has direct access to their own, private memory that is unreachable for the other processors. Data exchange is only possible via direct communication, typically message based via the global interconnect.

In a shared memory architecture, all processing elements have the same address space and can access the same memories at fixed locations in their memory map. A distributed memory architecture on the other hand provides a separate address space for each processing element. Consequently, there is no way for one processing element to directly access the memory of another. PEs

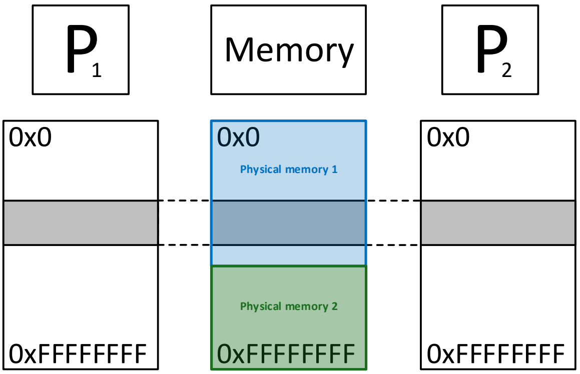


Figure 2.4: The memory map of two processors P_1 and P_2 in a shared memory architecture.

may also be grouped into locally shared, globally distributed architectures. This is often the case in tile-based many-core architectures. Furthermore, there are so called Distributed Shared-Memory (DSM) architectures which provide a singular address space, although the underlying hardware consists of multiple physically separated memories [99]. The memory maps for shared memory and specifically DSM architectures is highlighted in Figure 2.4. Both processors P_1 and P_2 access the exact same physical memory location when they request a certain address, independent of their location or interfacing with the memory. Depending on whether the access latency to the physical memory is the same for all cores, the architectures are called Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) architectures. In DSM, the physical memory is split into multiple (e.g. two in the figure) memories that form a contiguous memory range together. There is no indication or restriction to where those memories reside physically, it only needs to be ensured that processors can directly request an address and the interconnect will handle all steps necessary to retrieve the data on said address. Due to the limited space for memories that are close to the processing elements there are also variants called Cache-Only Memory Architecture (COMA) which refers to the fact that local memories only hold copies of a far away physical main memory.

In contrast to the shared memory approach, distributed memory architectures typically follow a MPI-based (Message Passing Interface) communication scheme [112]. MPI is based on the exchange of messages, either directly between individual nodes or via broadcast/multicast[1] messages. The communication is always push based, meaning a node can never directly access another nodes memory unless it sends a request via a message that is handled on the receiving side accordingly. Shared memory architectures on the other hand allow many styles of communication and synchronization. As a downside, consistency and coherency issues need to be addressed in such systems.

2.1.2 Memory Hierarchy

Since memory cells can easily cover large parts of a chip, their location, size and underlying implementation technology are of critical importance. In order to improve the trade-off between size and locality (which typically translates into access latency) a memory hierarchy can be employed. A typical memory hierarchy in a single- or many-core system is shown in Figure 2.5. Closest to a processor are its registers that operate at full core speed and need to be accessed directly ir implicitly via corresponding instructions. Due to their location, integrated into the core itself, only a very limited amount of registers are available. All following levels in the hierarchy are accessed via addresses. Each directly accessible physical memory block needs to be part of the memory map that links memory blocks to address ranges. Larger physical memories are typically located further away from a processor core and are implemented in more cost efficient technologies due to their capacity. This results in increased access latencies typically proportionally to their size and location. In order to find a balance between size, speed and location, caches are often employed in a memory hierarchy. A cache has no physical address range and does not

[1] Broadcast refers to a stlye of communication that targets all available nodes while multicast targets subset of nodes that needs to be specified

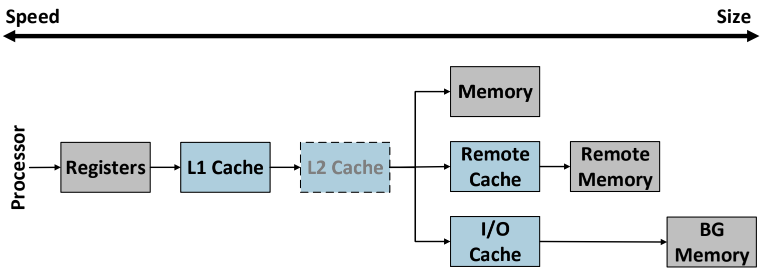


Figure 2.5: Speed versus size tradeoff in memory hierarchies.

hold any unique data but instead "hides"[2] other physical memories by taking in selected copies. The motivation behind this is that a cache is smaller and much faster than the memory it hides but contains only the most relevant data that a processor is currently operating on. Caches are fully transparent to a processor, i.e. the processor is not aware whether a cache is present but will have improved performance when data is fetched much quicker due to being stored in a cache. Caches come in several levels and different variations, again differing in size and access speeds. Common are Level 1 (L1) caches, but some architectures use L2 or even L3 caches in their design. These may also differ by having either shared or separate caches for instruction and data. A DSM architecture that allows direct access to remote memories often also employs a remote cache (sometimes also labeled in the "L" + number format). These avoid costly data transfers over the interconnect network. Similarly, large Background Memories (BG Memory) that are not directly memory mapped, but only accessible via some form of I/O might employ some form of caching.

Even though memory hierarchies are introduced by adding hardware components such as caches which are transparent to a processor, making good use of the underlying memory hierarchy on the software level can have a major impact on performance. As a well known example, a programmer should take good care when implementing a matrix multiplication with two loops since the order of the loops decides whether there will be many cache misses or not. This

[2] The word originates from the french verb "cache", translating to "hide"

lead to the development of special languages that enable a memory-hierarchy aware form of programming [43].

Cache Coherence

If some data is shared among multiple caches, any update on said data needs to trigger a synchronization among the caches and possibly also the memory location that is mirrored in the cache line. This so called cache coherence can be seen in shared memory architectures that employ caches since they need to consider the coherence in order to avoid operating on wrong data. In order to provide a coherent view on the data for all processing elements, a coherence scheme is employed. A multitude of schemes exist, for example bus-snooping protocols as used in the pentium processors [9] or directory based cache coherence protocols [27]. Most coherence protocols do not scale well, causing problems in a many-core architecture. Without coherence, the software layers need to make sure no data corruption may occur, eliminating many of the benefits that a shared memory architecture exhibits. A solution for the problem of scalability is provided by a concept called Region-Based Cache Coherence (RBCC) [116]. In order to avoid the problem that all caches in a system need to be synchronized, RBCC selects only a subset of caches in a system that are kept coherent. This reduces hardware complexity and overhead while keeping limitations for the software layers low. It does however necessitate that the regions do not grow too large since this would again impact scalability negatively.

2.1.3 Existing Many-core Architectures

Several commercial architectures that can be considered true many-cores have been released over the past few years. Among them are attempts to tap into this new form of massively parallel processing from well known processor companies like Intel, but also architectures that were designed by smaller start-ups or initiated by government funded research programs. The presented commercial architectures in the following serve mostly as a motivation for further research in this field and highlight the diversity that is already available.

Furthermore, several research many-core architectures exist in the academic world. Since these are pure research architectures, no physical chip implementation exists. Instead, simulations and FPGA prototypes are used for feature development, performance evaluation and verification. Due to the fact that only prototypes exist, these architectures are not fixed and come in variations containing more or less cores and accelerators or using different configurations for components such as the interconnect. Two such architectures, namely FlexTiles and InvasIC, will be highlighted specifically. The analysis of many-cores in this work was inspired greatly by these architectures and many of the proposed techniques and methodologies introduced later on were developed with these architectures in mind and are based on the experiences during their development process.

Intel SCC and Xeon Phi

As a major manufacturer of single- and multi-core processors, a number of research teams at Intel set out to investigate many-core processing following different angles of approach. Among the more recent architectures are the Single-chip Cloud Computer (SCC) [59] and the Xeon Phi [113]. The SCC is based on the concept of bringing the processing power of a whole server cluster into a single chip. The name stems from its intended use as a hardware backbone for cloud computing in the server market.

In contrast, the more recent Xeon Phi architecture has more similarities with a GPU and their SIMD/vector processing approach. It hosts simplified cores that are better suited for SIMD processing while easing integration of many cores into a chip.

The SCC has 48 physical cores while the Xeon Phi comes in variants with up to 72 cores - both classifying these architectures as many-cores.

Kalray MPPA

Kalray is a company dedicated to the research and development of chips for massively parallel processing. Despite being supported by the french government and public research organizations, the kalray Multi-Purpose Processor Architecture (MPPA) was released including all software support packages as

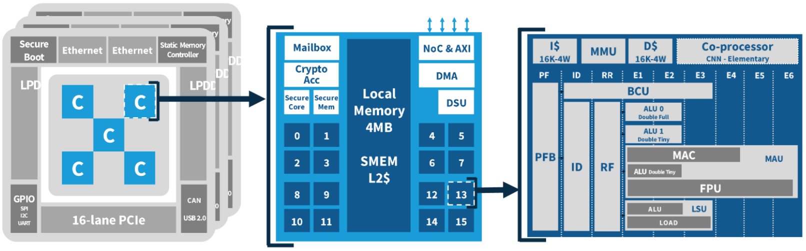


Figure 2.6: A schematic view of the Kalray MPPA3 Coolidge many-core architecture [2].

a commercial architecture. The MPPA consists of a family of architectures that have been released over the past few years [39]. The latest development is the MPPA3 Coolidge architecture as seen in Figure 2.6, which is produced in 16nm FinFet technology. It comprises up to 160 64-bit cores with another 160 co-processors for computer vision and deep learning applications. The chip consists of compute clusters with 16 regular 5-issue Very Long Instruction Word (VLIW) compute cores, some cluster local memory and a number of special accelerators like crypto cores or DMA.

Adapteva Epiphany

In 2016, Adapteva taped out their latest many-core chip: the Epiphany-V [92]. The chip contains 1024 64-bit RISC processors, each with a small amount of SRAM and a network-on-chip interconnect. The chip also hosts custom ISA extensions for deep learning, communications and cryptography. Adapteva released their Epiphany chip as part of a board solution, containing a Zynq FPGA including an ARM A9 dual core running the Linux operating system. The many-core can be programmed either bare metal or with common frameworks and languages such as OpenCL or MPI. Since the on-chip network implementation provides a distributed shared memory environment, even OpenMP programming is possible.

FlexTiles

The FlexTiles architecture was developed within an European FP7 project by the same name introduced in an overview paper [74]. The architecture itself is a pure research architecture based on the CompSoC platform developed by the TU Eindhoven [52]. CompSoC utilizes the concept of virtual execution platforms that provide composable isolation i.e. each platform can be analyzed independently for performance, Worst Cast Execution Time (WCET), etc. since the platforms are executed side-effect free. Composability does not only give upper bound guarantees, instead it is based on the promise that program execution will always happen exactly as if no other tasks or virtual execution platforms are present in the system. A CompSoC platform contains a number of microblaze cores that can be clustered in tiles. The tiles and peripherals such as memory or I/O are interfaced via the Æthereal NoC [50]. In FlexTiles, the CompSoC platform is extended towards a flexible, heterogeneous many-core architecture. The conceptual layout is shown in Figure 2.7, yet the platform may contain an arbitrary number of processing nodes. FlexTiles introduces a generic concept for harnessing the computing power of

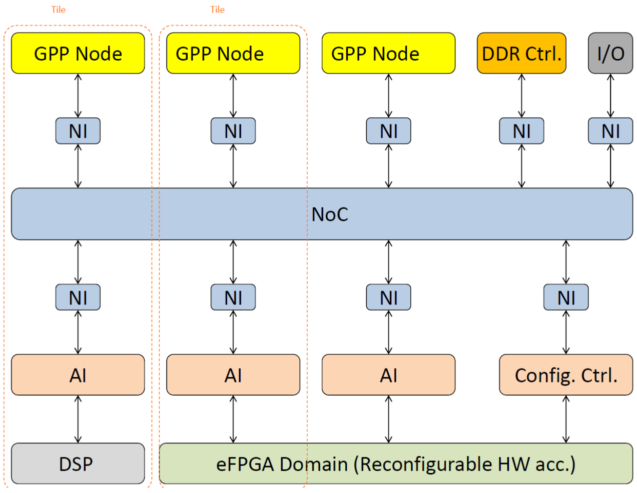


Figure 2.7: A schematic view of the FlexTiles many-core architecture [74]. The number of nodes is scalable and thus not representative of an actual chip implementation.

heterogeneous accelerators. Towards this goal, a flexible Accelerator Interface (AI) is added to the platform that can be accessed and configured by a control GPP core over the NoC. The AI can interface a multitude of accelerator cores, most notably specialized DSP cores and an eFPGA domain that provides a runtime-reconfigurable hardware.

InvasIC

The InvasIC architecture [57] is a many-core design that is developed within the Invasive computing transregional collaboration research center 89 (TCRC 89) [121]. The participants of this DFG funded research cluster come from the Karlsruhe Institute of Technology (KIT), the Technical University Munich (TUM) and the Friedrich-Alexander University of Erlangen-Nuremburg (FAU). InvasIC investigates novel massively parallel programming and processing paradigms on all levels from algorithms and language through software and operating system down to the hardware level.

The hardware architecture of InvasIC is a heterogeneous tile-based many-core architecture that adheres to the Partitioned Global Address Space (PGAS) model by providing a distributed shared memory [126]. A tile in InvasIC can take many shapes, among them are regular compute tiles, memory or I/O tiles, and special tiles containing accelerators such as the Tightly-Coupled Processor Array (TCPA).

Most prevalent is the regular compute tile, which is a multi-core architecture by itself. It typically consists of a number of cores, a Tile Local Memory (TLM) and a number of peripherals. All components within a tile are interfaced by a tile local bus. Most components of a compute tile are taken from the cobham gaisler IP library. This library is an open source collection and Hardware Description Language (HDL) library for building multi-core architectures. It comprises of the LEON3 processor core (a sparc v8 ISA), several peripheral components including caches, bridges, I/O and local memories, all connected via an Advanced High Performance Bus (AHB). On top of the gaisler components, several additional components have been added as part of the InvasIC research efforts. These include the run-time reconfigurable *i*-Core special processor that may replace a regular processing core in a tile or the TCPA accelerator.

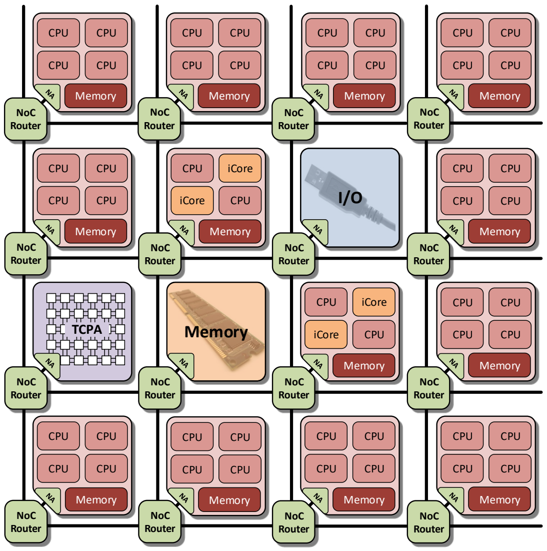


Figure 2.8: A typical configuration of the InvasIC many-core architecture [55].

An example of the InvasIC architecture can be seen in Figure 2.8. All tiles are arranged in a 2D meshed layout. Each tile has a local network adapter that interfaces the tiles via a NoC, called the *i*-NoC, to each other. The *i*-NoC and the tile based approach creates a scalable framework for building many-core architectures. Since the InvasIC architecture is not realized as an Application Specific Integrated Circuit (ASIC), it is not fixed and can come in many variations. However, FPGA prototyping limits the size of the architecture. Recent multi-FPGA prototypes can host up to 16 tiles with a total maximum number of 80 cores.

2.2 Networks on Chip

A Network on Chip (NoC) takes concepts from the Internet infrastructure and High-Performance Computing (HPC) interconnects in order to solve the scalability limitations of more traditional crossbar or bus-based on-chip interconnects [13]. As more and more nodes are interconnected with one another, providing only a single data line that needs to handle all data transfers quickly becomes a bottleneck in any processing architecture. Similarly for the other extreme, all-to-all connections are no solution either, since they do not scale due to exponential growth in the number of resources. Reducing the amount of accesses over shared interconnects, for example by employing caches, can alleviate but not prevent this bottleneck and can thus delay but not solve the underlying issue. A scalable solution that remains is to separate all interfaced nodes logically and physically. Networks on chip provide such a separation that still allows all nodes to communicate with one another. A NoC offers multiple paths between connected nodes. As a single data transfer uses only a part of the whole network, the remaining network resources are free for other data transfers that may happen in parallel. The ability to handle multiple transfers from different source/destination pairs in parallel results in increased throughput that scales with the size and topology of the network. It also enables a large degree of freedom when integrating multiple processing, memory, accelerator or peripheral components [6].

A NoC consists of two parts: the routers which form the communication backbone and a Network Interface (NI), sometimes called Network Adapter (NA), that bridges the gap between routers and tiles. The NI is responsible for handling packet creation and reception and automatically translates requests from the processing elements in the tiles into remote data fetch or data push operations on remote memories and buffers. Although most NoCs operate clock synchronous, some follow the GALS approach (Globally Asynchronous Locally Synchronous) by operating the NoC internals completely asynchronous and leaving only the compute units in a synchronous domain [16]. In this case, the network interface is responsible for splitting the two domains. Multiple routers are interconnected to one another depending on a chosen topology. Routers in a NoC are typically copies of the same architectural template, although optimized designs sometimes introduce different router architectures within the same NoC. This can help reducing bottlenecks for example by increasing the link width[3] close to the memory or near I/O regions.

[3] The link is the physical connection between two nodes. Link width typically refers to the width of the data line, not including any control lines.

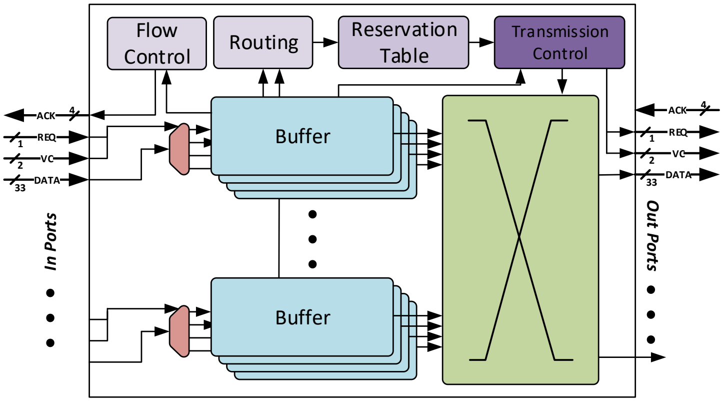


Figure 2.9: Layout of a packet-switched router with virtual channels and flow control based on the *i*-NoC router [55].

An abstracted architecture view of a packet-switched router is shown in Figure 2.9. The figure is based on the *i*-NoC router and will be used to highlight several basic NoC features and concepts [55]. The interface signals and all essential components within the router are shown. In general, a packet-switched router needs to handle the following tasks:

- Buffering of incoming data
- A routing unit that decides the outgoing port for each packet
- A crossbar for connecting all input ports to the output ports
- A Link arbitration if multiple packets compete for the same output port
- Flow control to avoid data loss

Buffering of packets may be handled in different locations of the router, typically either on the input, the output, or both. Using multiple buffers can be beneficial for reducing the critical path, allowing higher clock frequencies in the design. However, it is a trade off and is often discouraged since buffers take up a significant amount of space and power so any additional buffers should generally be avoided. Routing is handled by a routing unit that is triggered when a new packet arrives at a router. Depending on a network protocol, the output port for a given packet is chosen and an entry in a reservation table is made. Based on this reservation table, all active packets in a router can be arbitrated onto the links of the output ports. The transmission control configures the crossbar so that packets from the input are forwarded towards the output port[4]. Flow control is a concept for avoiding overflows in network buffers for traffic flows between routers. It may be realized by some form of handshaking or acknowledgment. It is an imperative feature for NoCs since packet loss must be avoided at all cost as network protocols usually do not consider re-transmission and error detection. Flow control counters are a solution for non-blocking operation since the sender can determine the available buffer space in a neighboring node.

The interface between two routers in Figure 2.9 contains four signals which are duplicated for bidirectional operation. One such interface is shown at the input, yet the same interface is used at the output. This bidirectional interface exists for all connected routers and network interfaces, depending on the topology of the NoC. For example, in a meshed topology this would mean one for each of the following: north, west, south, east, and local. The bit width of the interface signals may vary depending on the actual router implementation. The data line is used for transferring the packets and may contain extra bits for distinguishing the packet headers. The request line (REQ) has a bit width of 1 and indicates that there is valid data on the data lines in the current clock cycle. The virtual channel (VC) line indicates, to which virtual channel the currently active data belongs. The concept of virtual channels essentially splits the input buffers into multiple smaller buffers that can hold independent packets or data streams. This reduces blockage in the routers since a blocked data stream will not hinder another independent data stream. It furthermore enables the implementation of Quality of Service (QoS) features. Finally, an acknowledge line (ACK) signals flow control credits back to the upstream[5] router.

A packet is composed of one or more flits, which in turn consist of one or more phits. A phit refers to the amount of data that can be physically transferred

[4] Depending on the location of buffers in the router, the crossbar can connect buffers and/or ports directly

[5] Upstream refers to a neighboring router that has sent the data to the current router

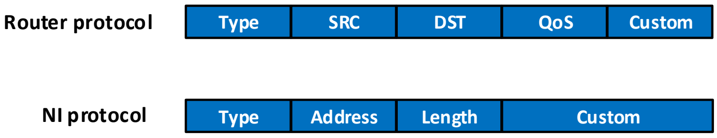


Figure 2.10: Generic components of the router- and the NI-protocol.

between two nodes in a single cycle. It is thus equal to the width of such a physical connection. A flit is a logical unit, referring to a flow control unit, which may consist of several phit. A packet in its simplest form is only a chunk of data that is transferred from one end to another. Yet in order to do so, the routers need to know how to forward the packet. For this task, packets must either include information about their target location or follow a pre-established connection.

Besides the structural layout, a network protocol is required that defines how the end-to-end communication is handled and how each packet is interpreted. A NoC essentially implements the four lower layers of the OSI reference model [60]. The lowest level, the physical layer, is represented by the physical links between two routers. Layer two, the data-link layer, is realized by the flow control mechanism. Flow control is required to prevent any data loss due to overflowing of buffers. The network layer is realized by the routing and switching within each router. Finally, the transport layer, which is responsible for the end-to-end data handling, is realized by the network interface. Transport and network layer are also reflected by a router protocol or network interface protocol respectively. A generic example is shown in Figure 2.10. The NI and router protocol wrap around the actual data that is transmitted. The outermost layer is the router protocol since it is required to handle transmission among the routers. A packet must contain a header flit that specifies its type and either a packet length field or any other indication that signifies the end of a packet such as a tail flit. A destination address within the network is also required for the routing algorithm to make its decision. Additional information for QoS handling or other custom parameters required for the transmission among the routers may also be part of the router protocol. The innermost layer represents the NI protocol and contains information that is required to handle requests in the network interfaces. In a DSM architecture, it must specify the operation (read or write), the address, potentially the length (in case of DMA accesses) and other architecture specific custom extensions.

In a distributed shared-memory architecture, all processing elements need to be able to access every memory and every peripheral that is part of the global memory map. The network interface needs to support this accordingly. Thus, the NI needs to listen on the local bus and react once a remote addresses is accessed. It automatically triggers a remote request over the NoC by creating a packet according to the original request. The decision, to which tile a remote request is sent, can be made based on a part of the address that is accessed. In contrast, a distributed memory architecture does not have no global addresses and thus the NI does not need to automatically create packets for remote requests. Instead, the processors need to configure each remote request manually by writing memory mapped registers.

2.2.1 Topologies

A NoC consists of nodes, called routers, that are physically interconnected to a number of neighboring nodes. The topology of the NoC decides how the nodes are interconnected to one another. In theory, NoC routers can be aligned and connected to form any kind of regular or irregular topology. Yet there are common topologies that are often used in NoC implementations. This is mostly due to the fact that routing schemes (i.e. the algorithm that decides which next neighbor a message is forwarded to) have a strong connection to the topology. Some desirable routing schemes that are simple to implement require a specific topology in order to work. Most notably, the Dimension Order Routing (DOR), also called x-y routing, only works if every node can be reached from every other node by first following the link in direction of the x-axis, then in direction of the y-axis. A topology that allows DOR and is thus among the most commonly used topologies is the meshed NoC. In a mesh, each node is connected to one neighbor in all cardinal directions. Nodes at the mesh boundary do not contain any further connection in direction of their border, leaving them with less physical links than a node in the center. A mesh can be of any dimension, yet 2D meshes are the most common variant since chips today are typically 2D integrated which eases placement of the mesh logic on a chip. However, meshed interconnects work exactly the same in higher dimensions, i.e. when 3D stacking becomes more prevalent new chips might employ 3D meshed topologies.

2.2.2 Switching Schemes

Switching schemes are another distinctive feature in a NoC. There are two major switching schemes available: circuit switching and packet switching. These have a major impact on the performance of a NoC and both come with their advantages and disadvantages. The general concept for both is highlighted in Figure 2.11. Packet switching refers to a switching style that is based entirely

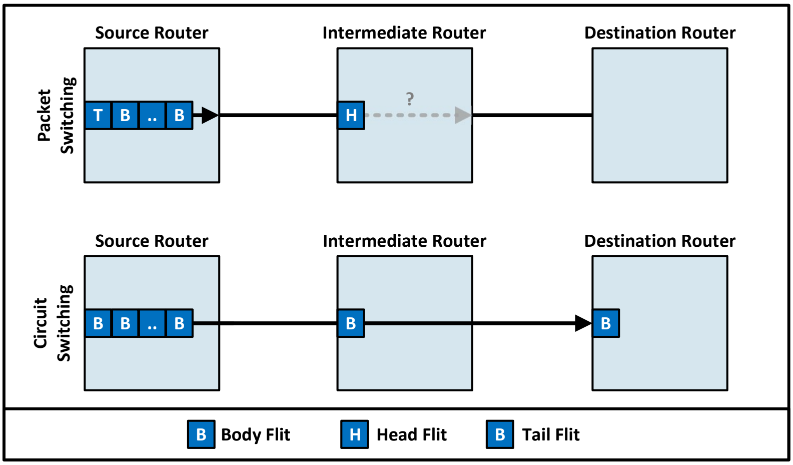


Figure 2.11: Packet versus circuit switching in a NoC.

on packets: single messages consisting of one or more flits that need to be routed independent of other packets towards their destination. In theory, each packet may take a different route through the network if the routing algorithm allows it. This requires routing information in any packet, generating some overhead in form of a header that has to be transported in addition to the actual data called payload or body flit. Whenever a head flit arrives at a router, its next hop[6] needs to be determined by the routing unit. The benefit of this style of communication is that it is easy to implement, does not require any setup or holding of state in the routers, and is very flexible. In contrast, circuit switching refers to a style of communication that is based on end-to-end connections. Whenever data is transported over the NoC, a connection (i.e. circuit) that reaches through all routers from sender to receiver of the message needs to be established first. Afterwards, no further routing information is required and no protocol overhead in form of head or tail flits is required. This means that a circuit does incur less overhead on a continuous stream of packets if the circuit is kept alive, yet keeping a circuit alive means that the associated resources in the routers can not be used by any other packets crossing its path. Thus, circuits may have to be deconstructed in order to free resources for other circuits or packets. Since circuit establishment and deconstruction induces an overhead and requires a setup mechanism that needs additional resources, it is not suited for traffic situations in which many small messages among different nodes have to be transmitted. Although it is more complex due to the setup mechanism and the state holding of the circuit, it also assures high throughput (less bandwidth wasted on headers), low latency on established circuits (no routing/reservation step required), allows bandwidth and latency guarantees, and saves buffer space in the routers.

2.2.3 Existing Networks on Chip

The NoC technology can be found as part of an integrated circuit in multiple forms. There are several companies that provide a NoC as a flexible interconnect that can be tailored towards a specific application when integrating components into a System on Chip (SoC). Among the three biggest competitors are Arteris, Netspeed and Sonics that all offer a commercial product that has been used in a number of IC designs. Furthermore, there are NoCs that are specifically developed for many-core computing. These are either linked to a specific many-core architecture or are developed as independent, generic architectures for NoC research activities. The latter are of special interest in this work about many-cores and one such architecture will be presented in more

[6] A hop is a single step in a network, i.e. between two neighboring nodes.

detail. It was extended by some special features as described later on in this work and was also used as a major target for the prototyping methodologies.

The invasive NoC (*i*-NoC) is a scientific NoC architecture that is developed within the InvasIC Transregional Collaborative Research Cluster. As a scientific architecture it is a feature rich design that comes in many variations [55]. A typical configuration uses a packet-switched layer on a 2D mesh topology with four virtual channels. The router is pipelined with up to five stages and routers operate on a data width of 32 bit. Flow control is realized with counters that track the fill level of the input buffers of neighboring routers. The flow control credits are communicated via a separate acknowledge line. A network adapter serves as interface between the tiles of the invasive many-core and the routers of the NoC. It provides a distributed shared memory access by defining global memory addresses that are automatically routed over the NoC to the correct tile [133]. To enable high-throughput communication among the tiles a special DMA unit is integrated into the network adapter that can push large chunks of data over the network. The *i*-NoC was specifically designed to enable resource invasion, the core concept of InvasIC. This is realized by a time slot mechanism that allows the reservation of a certain amount of link bandwidth in a Time-Division Multiplexing (TDM) manner. These need to be setup in advance and provide a communication style that is similar to circuits even though it is built on top of a packet-switched NoC layer that works as the basic interconnect.

2.3 Heterogeneous Computing

Heterogeneous computing refers to a style of computing that utilizes multiple different processing elements. In a broad sense this includes many special cases, for example specific functional units within a processing core.

Single-ISA heterogeneous multiprocessors can provide much higher performance than comparable regular multiprocessors of similar size [69]. Architectures based on ARMs BIG.little concept that contains cores with two different bitwidths are an example for single-ISA heterogeneous multiprocessors [1]. These can be found in several commercial designs today, such as the Exynos processor from Samsung. They have the benefit of rather simple and already available software support, only requiring scheduling optimizations in the oper-

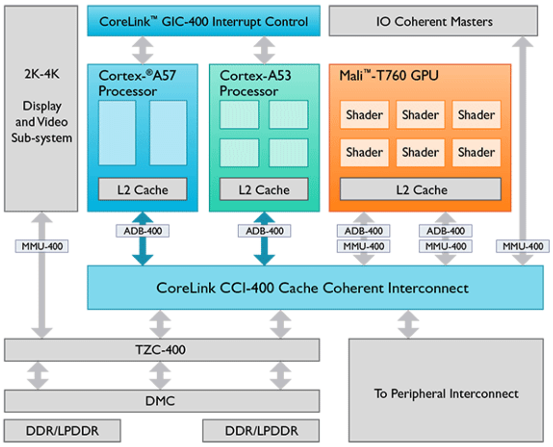


Figure 2.12: ARM big.LITTLE architecture [1].

ating system to reap the benefits of these architectures. The idea of integrating similar processors that differ in their performance and power consumption promises to mitigate the limitations imposed by Amdahl's law. The unavoidable sequential part of an application can benefit from a high-performance processor while the parallel parts can be handled by a large number of power-efficient cores [68]. Using an entirely different ISA or pairing cores with accelerators promises even more performance per watts and will be a crucial part in the fight against the dark silicon problem and the power wall. Such architectures fulfill a more strict definition in the sense that they are heterogeneous or asymmetric as defined in [85] if they either contain multiple GPP cores with differing ISA or they contain GPP cores and detached special accelerator tiles that host either fixed hardware blocks or a reconfigurable fabric. Accelerators in this case are considered to be hardware blocks that are designed to fulfill a predefined function without requiring or allowing any software programming. When building such a heterogeneous computing system, it is imperative to consider all other bottlenecks that may arise when using massively parallel and highly optimized processing elements. Specifically, such architectures often prove very data intensive which results in bottlenecks in the interconnect or the memory controllers that should be considered during the design process [97].

Both scientific many-core architectures presented earlier in section 2.1.3 and section 2.1.3 are heterogeneous. FlexTiles introduced the accelerator interface (AI) that provides a common interface for a multitude of accelerators including DSP cores and eFPGAs. The design exhibits inherent heterogeneity by including the reconfigurable eFPGA and is designed to be scalable to a multitude of nodes, forming a many-core architecture. Similarly, InvasIC utilizes the *i*-Core that extends the processor pipeline with a reconfigurable fabric [56]. Furthermore, the tile based design allows many different computing units including some special accelerators such as the TCPA, making InvasIC designs heterogeneous in nature. Both examples introduce heterogeneity in the form

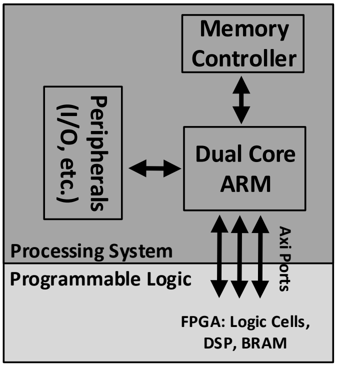


Figure 2.13: Xilinx Zynq introduces heterogeneity by combining an ARM core with a reconfigurable fabric (FPGA) [131].

of FPGA based reconfigurable elements. The *i*-Core is based on a LEON3 core that was extended with a reconfigurable fabric. This fabric allows the instantiation of so called Special Instructions (SI) that extend the regular sparc ISA. It can also be considered as a special case of heterogeneity, since it can adapt its instruction set dynamically, thus providing a different ISA.

Similar to the *i*-Core or the eFPGA, other architectures include FPGA technology in order to enable heterogeneous computing. The Zynq from Xilinx tightly integrates hardwired ARM cores with a reconfigurable FPGA fabric into a multiprocessor system on chip (MPSoC) as shown in Figure 2.13. It enables a multi-core ARM platform to exploit the massive parallelism that an FPGA based accelerator provides. This makes it a prime example of a heterogeneous massively parallel platform although it is typically not labeled as being many-core. Its biggest challenge is providing an efficient interface between the host core and the reconfigurable fabric [110]. Xilinx utilizes an AXI interface that can automatically be instantiated in the reconfigurable part when using their FPGA design tools.

One of the most extreme variants of heterogeneous computing are accelerator-rich architectures. These host a large amount of potentially different accelerators that promise very high performance and efficient execution yet prove difficult to handle [31]. The limitations of fixed accelerators that might not fit well to different applications can be circumvented by including reconfigurable fabrics in such architectures as well [30].

2.4 Design Languages

Hardware design and development has been enabled by the emergence of tools and languages that help in coping with the growing design complexity.

In the following, a selection of relevant languages that are in widespread use as of today will be presented.

2.4.1 VHDL/Verilog

VHDL and Verilog are Hardware Description Languages (HDL) that were developed in an effort to raise the level of abstraction in hardware design. Instead of manually designing circuits on the transistor or gate level and dealing with the placement of logic, HDL typically operate on the Register-Transfer Level (RTL). The assumption behind RTL descriptions is that any digital circuit consists of clocked elements that hold state (i.e. registers) and combinatorial logic that connects such registers. HDLs allow structural descriptions of modules, which are defined by their interface and may hierarchically be composed of other modules down to transistor level. To ease this process, HDLs introduce so called behavioral modeling which allows a style of hardware design that describes the behavior on a functional level instead of specifying the structure of a design. This brings HDLs much closer to traditional computing languages, allowing the use of variables and simple operators. However, hardware still differs from software in that it is inherently parallel, while most software languages assume a strictly sequential form of execution incorporating only explicit forms of parallelism. This has to be considered when designing in HDLs, since reordering of the code may result in entirely different designs.

In todays digital circuit design, hardware blocks are typically written in VHDL/Verilog or one of their extensions (e.g. System Verilog, VHDL-AMS). HDL descriptions may be simulated in HDL simulators that allow quick compilation or serve as input for a number of synthesis tools that target FPGA or ASIC technologies. HDL simulators provide the means for emulating the inherent parallelism of hardware even if they are executed in sequential software. This provides accurate results, yet in some rare cases they produce different results from real hardware. However, this typically only happens in code that does not correctly follow language specifications. Similarly, some tools do a number of automated optimizations if they detect behavior that was probably unwanted by a programmer. While this often improves on the quality of result, it can also hide sources of errors or make debugging more difficult.

2.4.2 SystemC

SystemC is a C library that extends the C language towards system and hardware modeling. Initially, SystemC was designed to mimic HDLs (see subsection 2.4.1) with strong correlations in their syntax. It introduced new logic types and the concept of modules and their interfaces. Furthermore, it provides a simulation core that enforces the parallel execution and evaluation on a clock cycle basis. As such, it allows the hardware design and verification similar to the traditional HDL while following C syntax. This means that a regular C compiler can be used to create the simulator binary. The simulation core is added by including and referencing the SystemC library during compilation. The approach of building on top of a software language has several benefits. Most notably, a large group of people is already familiar with the language and its syntax. Furthermore, compilers are well optimized and testbenches can easily be written in native C. On the other hand, designers must be aware that not all syntactically correct C code that can be simulated is also valid input for synthesis tools.

However, SystemC also enables an even higher level of abstraction from hardware, often called Electronic System Level (ESL). This level is less about implementation details and timings but instead targets functional verification and early Design Space Exploration (DSE). This is achieved by focusing on execution speed and ease of modeling above all else.

2.4.3 OpenCL

OpenCL is an open standard for programming heterogeneous architectures [88]. The goal of OpenCL is to ease the handling of parallelism and heterogeneous architectures at the same time for achieving efficient processing and execution of applications on all available hardware resources. While originally used for harvesting the raw parallel processing power of GPUs in conjunction with the CPU, support by tools and vendors for DSPs and FPGAs has continually increased. Performance-wise, OpenCL can achieve similar results to purely GPU targeted languages such as CUDA [42]. OpenCL is based on a subset of the C language, yet the recent version extends this towards a C++14

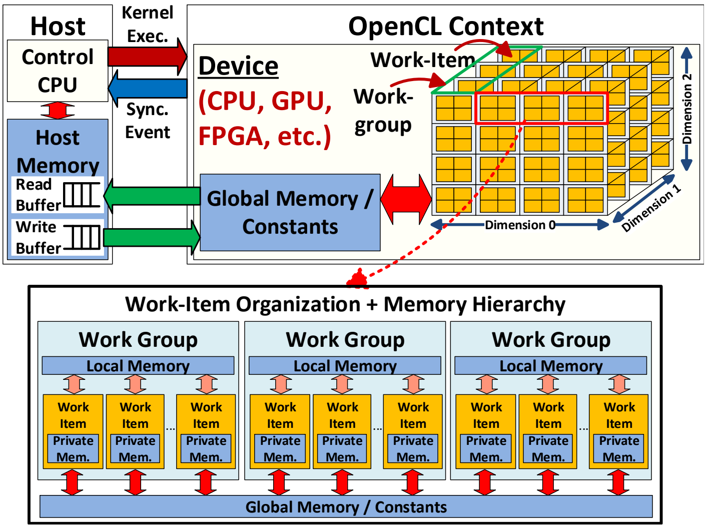


Figure 2.14: The OpenCL execution and memory model as shown in [SXMX+18].

subset. In OpenCL, programs are split into so called ”compute kernels” and a host application as shown in Figure 2.14. The latter controls the execution of the application and triggers the compute kernels that are executed on the available accelerators. Such an accelerator can be anything that is supported by a toolchain that can translate the kernel code and handle the interfacing. GPU have seen the best tooling support and are the most common use-case, yet even cores in a Commercial Off-The-Shelf (COTS) multi-core can be used for executing the kernels. More recently, support for FPGA-based accelerators has been introduced by the major FPGA vendors. Tools that target FPGA have to use some form of High-Level Synthesis (HLS) which translates the kernel code (written in C) into netlists on the FPGA. Each kernel is split into a number of work groups which in turn are split into individual work-items that are executed on a functional unit. OpenCL defines a number of memories that organize and optimize processing and interaction between host and kernels. Among those are global memories and constants, local memories that are shared within a working group, and private memory that is only accessible for a single work-item.

3 State Of The Art in Computer Architecture Prototyping

Prototyping has been a major enabler for the design of computer architectures and information processing systems. A prototype helps in the tasks of design verification and validation from the system level down to the physical level. These tasks are increasingly challenging as the complexity of computer architecture designs is rising in the era of multi/many-cores and heterogeneous systems comprising a multitude of customized accelerators. The term ’prototyping’ or ’prototype’ will be used in this document according to the following definition:

Definition: *A prototype is an early functional version on which the further development of a product is based*

In the context of IC design, a prototype always follows the goal of providing a computing system that can fulfill a predetermined set of computing tasks. Prototyping is by no means the only technique that has been employed for IC design as other approaches such as formal verification can be employed as well. However, todays computing systems are becoming too complex for pure formal approaches and they need to consider both hardware and software, making prototyping one of the best suited options.

In the following, state-of-the-art prototyping methods for computer architectures will be presented. In a first step, an overview is given, highlighting the general angles of approach. Specifically, classifications according to the level of abstraction and the scope are discussed. Afterwards, many individual methods are presented that are underlined by a number of cited research works which either make use of these methods or develop them further. This chapter lays the groundwork for the following chapter and the contributions of this work by giving an overview of relevant existing approaches and techniques. These will be analyzed further in the context of heterogeneous many-core in the next chapter, where the state of the art will be criticized to determine areas that are lacking as of today.

3.1 Classifications

Prototypes may be classified according to their scope or according to their level of abstraction from a physical implementation. Although there are connections and correlations among some prototyping methods based on their classification in these two areas, they are generally orthogonal.

3.1.1 Abstraction Levels

A major distinction among prototypes can be made according to their level of abstraction from a physical implementation. More abstract prototypes serve the purpose of software development rather than hardware design and are suitable for very early coarse-grained design space exploration. Gajski et al introduced their famous Y diagram in Figure 3.1 back in 1983, giving an overview of the levels of abstraction in IC design [47]. The Y diagram contains three axes, splitting the design process into the domains of behavioral, structural and physical. The abstraction levels are represented by circles that cross these axes. The innermost circle depicts the most accurate level while the outermost circle represents the most abstract level. A design flow based on the Y diagram starts from the outer layer and iteratively moves towards the inner layers while switching between the axes as needed.

While these classifications still hold as of today, many of the lower levels of abstraction are fully automated by Electronic Design Automation (EDA) tools nowadays and the separation of the axes is rather inflexible. Reality is much more practical and the trend towards more and more automation and design on higher levels of abstraction continues in order to keep the so called design productivity gap in check. Consequently, state-of-the-art design processes are not strictly bound by the abstraction levels introduced with the Y-diagram.

A more realistic view on the levels of abstraction in todays design processes is shown in Figure 3.2. The individual techniques will be discussed in more detail in the following sections and are only shortly introduced here to highlight their

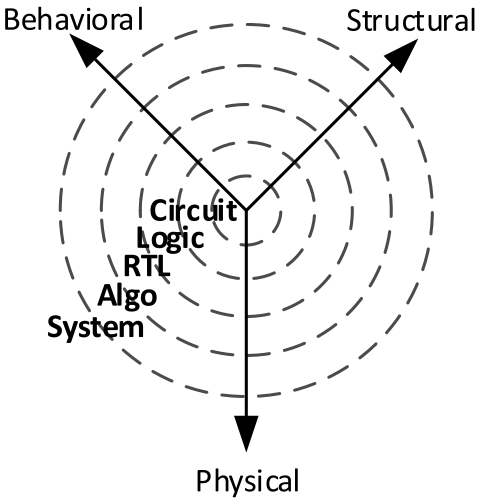


Figure 3.1: Y diagram according to Gajski et al [47].

interactions and interrelations. The figure relates to the level of abstraction from top to bottom as it is also often roughly followed in IC design. The color scheme gives an estimate on the operation speed of the respective technique, red taking comparatively long while green is relatively fast. Yellow represents speeds that often depend on the situation and can vary greatly, however they are typically intermediate in comparison to red and green. Double-ended arrows depict interfacing techniques while single-ended arrows depict design or simply execution steps, the latter of which are colored in black. The figure depicts a simplified view since in reality there are many interrelations and even potential feedback loops or re-spins involved in design processes. Similarly, the operation speeds may depend on many factors and should not be seen as absolute. In any case, the design process needs to be built on some kind of specification that defines the goals and requirements of a design. The specification can be split into a hardware and a software part by means of partitioning. A good design flow will follow a hardware/software codesign

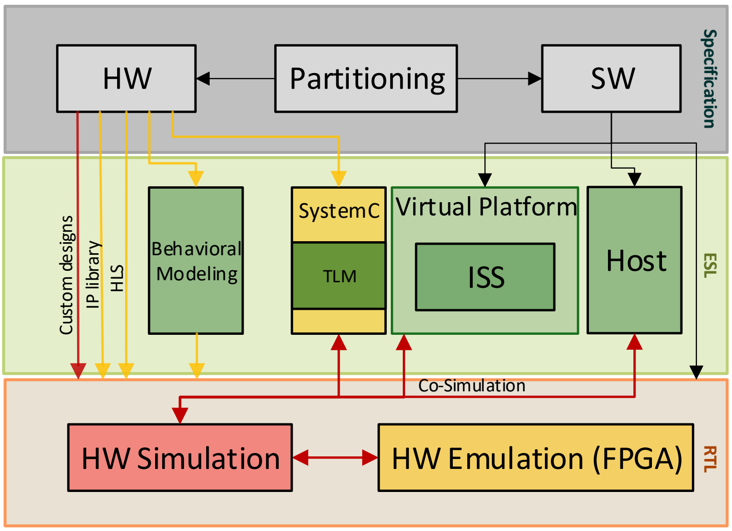


Figure 3.2: Elements and abstraction levels in a state-of-the-art IC design process.

approach that yields a well harmonized and efficient implementation through a carefully selected partitioning. Although this topic will be touched in this work, it is mostly out of scope and a field of research on its own. The two other major levels are the Electronic System Level (ESL) and the Register-Transfer Level (RTL). The ESL evolved in recent years as an overarching concept that includes modeling of hardware on an abstract level for enabling software developments to start early before a finished hardware implementation is realized. Although strict definitions are lacking, ESL revolves around the concept of behavioral modeling, Virtual Platforms (VP) and the interfacing of simulators and components. It is also closely related to design automation tasks that take models on abstract levels as input. A virtual platform may include an Instruction Set Simulator (ISS) that provides binary translation. This enables a VP to execute software that is binary compatible with a yet to be developed hardware architecture of which only the Instruction Set Architecture (ISA) is already defined. SystemC models are also often considered to operate on

ESL, either as independent components or as part of a VP. Abstract interfacing among components in a VP is provided by interconnection methodologies such as Transaction Level Modeling (TLM).

On the other end of abstraction is a very accurate level that is mostly used in hardware simulation and emulation. These prototyping techniques typically serve the purpose of low-level hardware design and verification. On this level, absolute cycle accuracy and correctness plays a crucial role. Accurate designs are closely related to the register-transfer level that serves as input for software based cycle-accurate HW simulators or synthesize into bitstreams for FPGA based HW emulators. As a downside, prototypes on an accurate hardware level are often difficult to implement and debug. Furthermore, accuracy comes at the cost of execution speed, making it often unfeasible to evaluate large designs or applications. Since FPGA allow full parallel operation of hardware resulting in comparatively fast execution, full system prototypes are typically realized on FPGA boards.

A number of attempts have been made to establish intermediate levels between the two, yet none has fully caught on. SystemC introduced approximately timed and loosely timed abstraction levels in order to provide such an intermediate level. The criticism states that intermediate levels can not provide full correctness and can thus not be used for hardware verification tasks while at the same time providing much less execution speed and increased modeling effort compared to the higher abstraction levels.

Hardware design and verification is still a process that often involves a large amount of manual steps. The major factor that kept hardware design efforts under control despite growing design sizes was the increased use of IP cores and libraries. However, whenever required IP blocks are not available or if multiple blocks have to be interfaced, manual efforts have to be made. Well defined interconnect standards help in interfacing, yet might also introduce inefficient structures due to their generic nature.

Despite the abstraction from hardware, there is a move towards tool supported synthesis of ESL-based system descriptions that allow skipping any lower abstraction level design steps in some cases [48]. ESL models or even software on a purely algorithmic/functional level can thus also be used as input for tools that are able to generate hardware layouts or FPGA bitstreams. This recent trend in EDA is closely connected to the term High-Level Synthesis (HLS).

Still, the defining goals for abstract simulators and prototypes are execution speed and ease of use. They allow the definition of interfaces according to fixed specifications, creating a so called golden reference that hardware implementations and more accurate prototypes must later adhere to. This enables software development efforts to start early, even including low level software and driver codes.

A panel discussion at DAC asked the question: "System Prototypes: Virtual, Hardware or Hybrid?" [18]. The panelists share their opinions on this question but seem to agree on the fact that virtual prototypes operating on an abstract level will provide a new addition to the well established accurate hardware prototyping. Hybrids can also be seen as a future development that is triggered by the emergence of virtual prototypes as they may bridge the gap between VP and hardware prototypes. In contrast to intermediate abstraction levels, a hybrid is never on an intermediate level but instead combines accurate and abstract models into one system.

3.1.2 Prototyping Scopes

The purpose of a prototype is directly related to its scope that defines what elements a prototype encompasses. Although there may be an arbitrary number of different scopes, there are a few distinct scopes that are often chosen for a prototype in computer architecture design. A typical prototype can either be restricted to the compute core itself or include other elements that a computer architecture needs to function. Specifically, this includes caches, memories and some form of I/O. The term System on Chip (SoC) has been coined which extends this scope even more. In a SoC, many different IP blocks are integrated into a single chip. Such a SoC may include multiple cores, interconnect networks, heterogeneous accelerators, multiple levels of memory hierarchy, etc. A prototype may represent a full SoC which encompasses all these elements for verification and validation purposes. Yet it can prove useful to employ independent prototypes for individual components as well. This is mostly seen when multiple parties provide a contribution to a full SoC design, since a full system prototype is only available at later stages in the design process. Prototypes that target single components or features may also focus on aspects specific for their design and are typically less complex and thus easier to implement. In summary, there is a distinction between a full system, a subsystem (multiple components) and individual components.

Another major scope for computing architectures are the distinctions between hardware and software. The term "computer architecture" in this document is used synonymous with processor architectures that contain both hardware and software. A prototype may target only the functional or algorithmic level, enable software development on a physical or virtual hardware, deal with the hardware verification and validation, or provide an environment to investigate both software and hardware in full detail.

3.2 Electronic System Level

The term Electronic System Level (ESL) encompasses a large amount of different techniques and approaches [80]. In this document the term will be used for describing techniques and methodologies on the highest level of abstraction, whether they target hardware design and verification, software design or early design space exploration. In the following, a selection of the most relevant ESL related techniques will be introduced and described in regard to latest developments in the scientific world.

3.2.1 Transaction Level Modeling

Transaction Level Modeling (TLM) first appeared in the system language and modeling domain [24]. However it only found widespread acceptance after it was adopted by SystemC [103]. As introduced earlier in subsection 2.4.2, SystemC was originally established as an environment for cycle-accurate hardware simulation of designs written in the C programming language. Yet as of version 2.0, the modeling of the system level came more and more into focus, creating the need for inclusion of methodologies that could handle this higher level of abstraction.

Despite its name, TLM is less a real abstraction level itself and more a style of implementing interfaces between components on an abstract level. The concept in comparison to interfacing in RTL is shown in Figure 3.3. It is used for separation of communication and computation which eases integration of

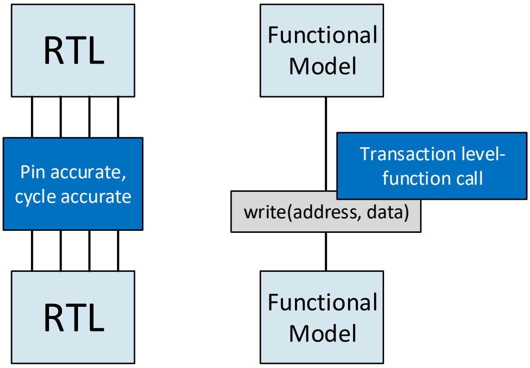


Figure 3.3: RTL versus TLM interfacing.

different components. In RTL designs, every component or module must be interfaced exactly according to a well defined set of pins and any interaction must follow a specified pattern or protocol. Special considerations must also be taken depending on synchronous or asynchronous operation when interfacing components. In contrast, interactions in TLM are handled by so called transactions. These essentially map to well defined and standardized function calls that allow the interfacing of arbitrary blocks written in SystemC or even other languages that adhere to the TLM specification. Data is transferred as so called payloads with any detail of accurate hardware implementation and interfacing hidden in the transactions.

3.2.2 Instruction Set Simulators

Instruction Set Simulators (ISS) represent a CPU on an entirely functional level. Their goal is to allow execution of application code compiled for a certain ISA even though the simulator itself is executed on a physical CPU with an entirely different ISA (typically x86 or x64). The ISS achieve high execution speeds by translating or mapping each instruction of an application onto one or more instructions taken from the hosts ISA. ISS have been an invaluable part of computer architecture design, mostly for design space ex-

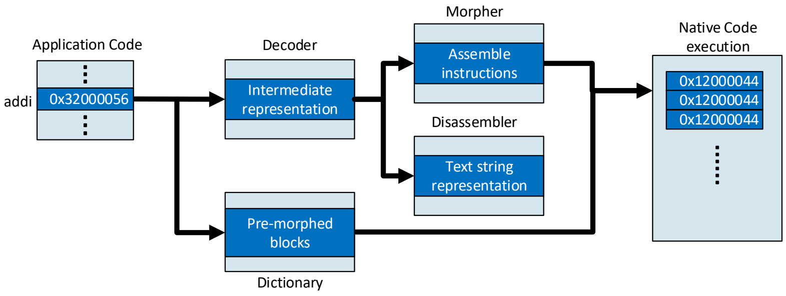


Figure 3.4: Dynamic code morphing in an Instruction Set Simulator (ISS).

ploration, partitioning and feature evaluation in early design stages. ISS can be categorized by their underlying implementation [134]. Interpretation-based simulators provide internal processor state and execute instructions by cycling through fetch, decode, dispatch and execute steps. Compilation-Based simulation basically does all these steps at compile-time already, translating every instruction into a series of native instructions. Lastly, binary translation does this translation dynamically, reusing dynamically translated code in a Translation Lookaside Buffer (TLB) for higher performance. An example for dynamic binary translation is shown in Figure 3.4. The input is a binary representation of application code compiled for a target architecture. If the code segment has not been processed before, it is dynamically translated by first decoding it into an intermediate representation. This serves as input for the text disassembly and the morpher stages. The latter generates a set of instructions in the native ISA of the host processor on which the ISS is executed. Previously translated instructions or code blocks are stored in a dictionary so that a lookup is sufficient in subsequent calls to this code segment. Depending on the modeled architecture, a single instruction may need to be decoded into a sequence of native instructions, affecting performance. The achievable performance also depends on a number of other factors, most notably the closeness between the two instruction sets[1] (simulated ISA and physical host-ISA). While ISS can

[1] Two ISA are close if the mapping of instructions in an application does not increase the required total amount of instructions significantly. This happens if many instructions are similar and can be directly translated into a single host instruction

work stand-alone, they are often part of a larger system and can be integrated in many other modeling or simulation environments such as SystemC [93].

3.2.3 Virtual Platforms

The term Virtual Platform (VP) is a rather generic term that is used in a number of different contexts. Virtualization is an approach that can be used for splitting available hardware resources and provide isolation of the access to these resources among several virtual platforms. This purely software oriented use case is often seen in todays many server clusters where virtualization has become an essential technique, allowing multiple customers to share server resources and dynamically scaling/changing the VP sizes [111]. Having multiple VPs on one physical machine may require some form of arbitration or management, a task that is handled by a so called hypervisor. Virtualization in this regard is closely related to operating system functionalities. So called virtual execution platforms may also be used to allow composable and thus predictable operation of multiple applications [51].

In contrast to the server and operating system domain, virtualization can also be used for prototyping and architecture research. In this work, the term VP is directly connected to this second use case and will be defined as follows:

Definition: *A Virtual Platform is a virtual prototype of a full processing system which exists only virtual and is executed on top of an actual hardware platform*

In this context, a virtual platform makes it possible to investigate non available or even non existing hardware. At the core of such a VP is often an ISS, which allows the execution of applications compiled for their respective ISA. A virtual platform can be adjusted or extended entirely on a software level compared to actual hardware prototypes, making it useful for early evaluations [90]. The prototyping use case is enabled by modeling schemes for new ISS models and peripherals that some virtual platforms such as Open Virtual Platforms (OVP) provide [3]. A VP also typically contains predefined models for common components such as bus and memory. Many commercial virtual platforms target the server market, however VPs such as the The Quick Emulator (QEMU) and OVP are also used for prototyping, especially in the scientific community.

A virtual platform is intended to represent a full computing system at execution speeds that aim to match an actual system. While this is not always achievable, it distinguishes a VP from accurate simulation of hardware models. Despite using a high level of abstraction to gain fast execution speeds, it is possible to extend Virtual Platforms towards more accurate execution. However, providing accuracy never comes for free as the execution speed is slowed down by a factor of 150-170 [106]. Virtual platforms can also be extended with power models that enable early estimates based on their architecture and application patterns although the high level of abstraction inevitably results in some inaccuracies in the predictions [35].

3.2.4 High-Level Synthesis

The rising complexity in hardware design calls for new methodologies in design and verification. Design automation plays a major role in dealing with this complexity that cannot be handled by human developers by themselves anymore. A multitude of tools and frameworks exist for this purpose but it is also reflected even earlier, in the definition of enhanced languages and methodologies. High-level Synthesis (HLS) is a term that describes a methodology that is realized by tools for generating hardware using design descriptions on abstract levels as input. The history and evolution of HLS can be roughly split into three generations [79]. The first generation was more of a by-product of research in the data-path domain. The second generation developed these initial steps into commercial, albeit rather unsuccessful, tools. These EDA tools would take a behavioral model and automatically translate it into a representation on a lower abstraction level such as RTL [81]. Hardware description languages, closely related to RTL, include such behavioral modeling in their language. These languages share many similarities with programming languages, yet there is still a major difference since in hardware, everything is executed in parallel by default while in software everything is expected to happen sequentially. The third generation developed the tools and methodology even further towards the software domain. As such, EDA companies provide new HLS tools which target SystemC, OpenCL or even plain ANSI-C/C++ as input language and can provide HDL code as a result. This C based synthesis has also been promoted as part of a solution for the design productivity gap [118]. However as of today, even though the abstraction level was raised significantly, HLS is still far from the universal tool that enables hardware design for anyone. This is mostly for two reasons, firstly the tools typically only provide good results for data-path driven designs with a high degree of trivial parallelism. Secondly, there is still knowledge about hardware architecture and supported language constructs required when designing with HLS as the tools need to be steered by a developer to produce good results [32]. A panel discussion at DAC highlights the most common input languages and their advantages respectively disadvantages [46]. The panelists stress the fact that if HLS is to succeed, it is not enough to work on synthesizeable subsets. Furthermore, HLS must come as part of a complete solution including virtual platforms for early software development, inclusion of IP cores in the design process and solutions for verification tasks. Solving these issues and supporting control-flow oriented tasks will decide the future and lead to a fourth generation of HLS.

Recent HLS tools that have been used for the design of hardware blocks include catapult-C from mentor [83] and Vivado HLS by Xilinx [130]. The latter is an extension of the Vivado simulation and synthesis framework by Xilinx that is provided together with the FPGA boards produced by the company. In the research community, the LegUp HLS framework has risen to some prominence as an open source alternative. LegUp also describes an intended design flow that profiles an application and subsequently synthesizes heavily used parts into a special accelerator [25]. High-level synthesis has also been investigated in conjunction with virtual platforms [20]. The goal is to use HLS for the full system design so that only a single platform description has to be written, in contrast to the process of developing an abstract virtual platform and later on manually designing a matching hardware design accordingly.

3.3 Hardware Simulation

In contrast to abstract simulators where execution speed and early DSE are the most important goals, there is also a need for fine grained prototyping on a much lower level that allows for verification, validation and debugging of a hardware design under test. Hardware simulators provide these features by taking HDL sources as input and compiling a simulation environment, in which all modules including their signals can be visualized. Since hardware is inherently parallel but a simulator is essentially software that is executed sequentially, these simulators need to provide a means for handling pseudo-parallel execution. Input languages depend on the simulator support, yet VHDL and Verilog are commonly used. SystemC models on their cycle accurate level may also be used as a valid input. Since hardware simulators are typically very slow and show bad scalability with increased design sizes, efforts towards simulator parallelization have been made [29]. Since in hardware simulation, all signals and registers in a synchronous design change in parallel, this task seems promising. However, due to the interfacing of components all signals crossing two partitions require synchronization in each cycle. Furthermore, even powerful server machines only boast a limited number of physical processing cores, limiting parallelization efforts.

3.3.1 Co-Simulation

In some cases such as Cyber-Physical Systems (CPS), several domains such as the electrical and physical need to be covered by a prototype. In an attempt to enable prototypes that encompass multiple domains at the same time, the term co-simulation was coined for standards and techniques that allow the integration and interaction of entirely different simulators. Independent of multiple domains being involved, it allows parallelization and even distributed execution of simulations. It furthermore also allows the mixing of different simulators with a different level of abstraction. High-Level Architecture (HLA) is such a standard that, despite originally not being developed for computer architecture simulation and prototyping, has been applied in this field in recent years [104] [91]. HLA defines three major elements, the Run-time Infrastructure (RTI), the federates and the Federation Object Model (FOM). The RTI is the central component that manages and operates the distributed simulations. Federates are individual instances that interact with the RTI. The FOM specifies the object structure that is used to exchange data.

3.4 Hardware Emulation

Hardware emulation targets a similar goal to hardware simulation: verification and debugging of hardware designs. In contrast to simulation, emulation overcomes the scalability issues since it is based on fully parallel execution of a hardware design. On the downside, this results in a long mapping stage of a design under test onto an emulator. Furthermore, hardware emulators are often extremely costly and require experienced operators to use them efficiently. Despite some core differences, FPGA prototyping fills a similar role and is thus sometimes considered another form of hardware emulation. Since FPGA prototyping is faster and much cheaper, this work will focus on this technique.

3.4.1 FPGA

Field Programmable Gate Arrays (FPGA) are a type of circuitry that can change its function even after production and deployment. While abiding resource constraints, any kind of logic can be mapped onto an FPGA. This enables them to be used as reconfigurable hardware since a design that is mapped onto an FPGA can be changed afterwards. FPGA provide a target technology for hardware designs on the same level as ASICs. Compared to ASICs however, FPGA have the benefit of allowing updates to the hardware design even after production and rollout has finished. Furthermore, the same FPGA can be mass produced but host a multitude of different designs, reducing product cost due to the economies of scale. This leads to FPGA being used as target technology for hardware implementation and deployment in many different application domains [102].

However, FPGA are not only used as a target for hardware designs but also prove to be a prime technology for prototyping of hardware architectures. In this context the term simulation acceleration is sometimes used since both hardware simulation as well as FPGA prototyping typically take the same HDL input for generating their designs but the FPGA executes orders of magnitude faster. Yet despite using the same input languages and hardware design descriptions, there is a major conceptual difference between hardware simulation and FPGA prototyping. A hardware simulator is a piece of code that is executed sequentially on a host PC or a custom architecture for fast simulation. An FPGA on the other hand provides a fully parallel execution, the same that any real hardware integrated circuit implementation exhibits. Due to this fact, FPGA prototyping is also sometimes referred to as hardware emulation.

Generating a design for an FPGA from a HDL description takes up several steps, most notably the synthesis of hardware modules, the placement onto the target FPGA and the routing (i.e. connecting) of modules and pins. Synthesis and place and route can easily take several hours for large designs and has to be re-done for each small change in the architecture. One technique that can improve this situation is partial-reconfiguration which allows the selection of regions which will not be changed in a re-synthesis. However, this technique is more often intended for live updates, i.e. changes to the hardware while it is running. Partial synthesis or checkpoint-based design are further techniques that slowly find adoption.

If a single FPGA does not provide enough resources for a design, multiple FPGA can be interfaced in order to provide a larger multi-FPGA platform. There are several proprietary variants of such multi-FPGA platforms on the market, developed by companies like proDesign, Synopsys or Cadence. Some scientific publications also present custom setups that consist of specifically tailored FPGA boards and generic interconnection schemes, for example FORMIC [77].

3.4.2 FPGA Virtualization

Virtualization is a concept that can be applied to FPGAs in a similar way as a virtual platform is used for a full system. The general goal of virtualization is to allow resource sharing or prototyping and evaluation of architectures which are not physically available. In the context of FPGA there are two major approaches for virtualization:

- Virtual FPGA (VFPGA)
- Resource virtualization

Since an FPGA is reconfigurable hardware it can inherently be used to model any kind of architecture for prototyping reasons as long as resource constraints are not a limiting factor. This means, a physical FPGA can also be used for hosting the logic that makes up an FPGA. The hosted FPGA is called a virtual FPGA or short VFPGA [44]. A VFPGA allows to keep an FPGA design independent of the underlying hardware and also enables FPGA related research since features that may not be supported by the physical FPGA may be emulated in the virtual FPGA. An example for this would be partial reconfiguration, which is not available for all FPGA, especially in earlier versions.

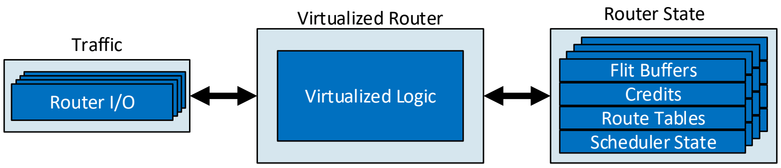


Figure 3.5: Resource virtualization on FPGA for emulation of a NoC design.

Resource virtualization on the other hand tries to virtualize a hardware design itself. This technique can be used when large, reoccurring structures exist in the design. As an example, a Networks-on-Chip (NoC) architecture can be chosen. In a NoC, the routers themselves fit the description of a reoccurring structure, since a NoC typically hosts many routers that are just duplicates of the exact same logic design. Such an approach for a virtualized NoC on an FPGA has been successfully realized in literature, enabling larger NoCs to be prototyped on an FPGA [95]. The concept can be seen in Figure 3.5. The NoC architecture is split into several virtual regions that run on a single physical instance of such a region. Thus, the logic exists just once and only the state holding elements are duplicated to exist physically for each virtual region. A control logic handles the clocking and switches among these memories in a round robin fashion so that all virtual routers are handled and synchronized before the next global cycle takes place. The same approach can also be applied towards other hardware components, as long as they contain large reoccurring logic structures. Besides the NoC, the cores of a Multi-core are an example for such a situation [105].

3.5 NoC Prototyping

NoCs are a recent development for interconnecting the components in SoC designs, mostly when multiple processing cores are integrated into a system. Since the trend towards such architectures continues, many prototypes specialize entirely on the NoC, independent of the system that is built around them when integrating the NoC into a full SoC. In fact, it can be argued that the interconnect is one of the most important aspects of modern computer architectures since it can easily become a bottleneck since the processing elements need to be fed with data from I/O or memories. NoC prototypes can be abstract, cycle accurate or FPGA based implementations. A recent publication shows an overview including a comparison of several FPGA based NoC simulators [64]. Major distinctions among FPGA-based simulators are made based on virtualization techniques (as mentioned earlier in subsection 3.4.2), traffic generation and the decoupling of simulator and network. Traffic generation refers to the method that is used for generating input for a NoC. This input can be generated by application code running on soft-cores on the FPGA [76]. Another approach is to utilize a C-based generator on a host machine [127]. A more software oriented approach that forgoes FPGA implementations allows fast design space exploration and architectural changes. Some software-based simulator frameworks try to achieve accuracy that is close to a real hardware implementation at the same time, yet it always comes at the cost of performance [63]. SystemC-based simulators have also been extended by including OVP models for attaining realistic traffic patterns [128]. This is possible since the OVP processor models allow the execution of unmodified application code, resulting in compatibility with actual benchmarks.

3.6 Full System Simulators/Emulators

A well known example for a widely used full system simulator is the Gem5 framework [15]. Gem5 provides a modular platform for computer architecture research and design space exploration. Historically, gem5 is the results of a merger between the M5 and GEMS simulation frameworks. Together, they provide a large number of CPU models with a differing ISA and furthermore an extension for interconnects and cache coherence protocols. Gem5 is designed to be flexible and extensible, making it useful as a basis for research in the field of computer architecture. The processor models and the memory subsystem can each be executed on different levels of accuracy. For CPU models, the differentiation is made whether full out-of-order execution is modeled, only in-order execution is modeled or whether even more abstractions from the micro-architecture are made. The simulator supports several features for software development and evaluation, such as a system-call emulation mode that emulates most syscalls and devices. Further extensions provide a language or environment for defining cache coherence protocols or novel ISA.

The most accurate configuration of GEM5 can be compared against a real system [23]. The evaluation shows, that even this accurate configuration results in a mismatch regarding execution time that varies between 1.39% and 17.94%. This is due to a correlation between L2 cache misses and the magnitude of the mismatch, pointing towards inaccurate modeling of DDR accesses in GEM5. It highlights the fact that even fully cycle accurate models do not behave exactly as a real system if not every component or aspect of it is modeled correctly.

Several attempts have been made on improving performance of full system simulators since they are typically used for design space exploration, a task that benefits from high execution speeds. One approach is introduced by the Sniper multi-core simulator [26]. It features so called interval simulation that splits the simulation into a functional part and an analytical model that does not necessarily represent the exact order of instructions executed. Instead it relies on the observation that miss events such as branch mispredictions or cache and TLB misses interrupt the otherwise steady streaming of instructions through a processor pipeline. These misses are simulated via special models and divide the flow of execution into intervals. For each interval, the simulated time is then calculated based on the analytical model of the architecture.

QEMU is also often referred to as full system emulator [12]. In this case the name is mostly attributed due to the ability of running the full Linux operating system on top of an emulated machine. However, it can also be considered a virtual platform similar to OVP introduced earlier, yet with less ability to model custom peripherals and architectural features.

3.7 Hardware/Software Codesign

A well designed computer architecture needs to coordinate and synergize its hardware and software components. If the design process splits these fields into two separate procedures, it is difficult to achieve good results. Thus, the term Hardware/Software Codesign (HSC) was coined, describing methodologies that try to achieve a well coordinated development of hardware and software.

A survey of the history and challenges of co-design claims that it evolved over three generations [122]. The first generation mostly dealt with the partitioning problem by designing algorithms that can provide good partitions of hardware and software components for a problem that is described on a functional level. In the second generation, multi-threaded execution and co-simulation extended the rather limited capabilities of co-design in the first generation. The third and current generation covers a broad range of new challenges, mostly the increased heterogeneity of computer architectures and the increased complexity in hardware and software. In this context, ESL and cross-level design have emerged and can be seen as closely related to the HSC. These specifically include some of the topics introduced earlier, namely HLS and virtual platforms. Further, a technique called co-emulation mixes emulators for hardware design with simulation for software design similar to the co-simulation approach [109].

4 Prototyping of Heterogeneous Many-core Architectures

Prototyping of heterogeneous many-core architectures introduces a new set of challenges where the existing techniques and methodologies for prototyping of computer architectures do not suffice. Thus, existing techniques must be adapted under consideration of these challenges and entirely new approaches need to be investigated.

In this chapter, requirements for successful and efficient heterogeneous many-core prototyping will be identified. The requirements will be categorized according to four major aspects that any computer architecture needs to cover. These are: software and programming, processing (elements), data (memories) and finally communication among all elements. Scenarios from the development of heterogeneous many-core platforms are introduced that emphasize the need for novel approaches. The state-of-the-art techniques presented earlier will be discussed in the context of the requirements and scenarios in order to reveal the limitations of existing approaches and show how they can be overcome.

4.1 Requirements

A multitude of prototypes may exist for the same many-core architecture since a single prototype might not fulfill all requirements and goals by itself. In the following, such requirements and goals for heterogeneous many-core prototypes will be discussed. Even though not all of these requirements must be met by a single prototype, they are all relevant towards the successful design of a new computing system.

4.1.1 Programming

One of the major reasons why many-cores have not found widespread acceptance in nowadays computing world yet is the challenge of programming such architectures. The trend towards more and more parallelism is unbroken, yet even at comparably low levels of parallelism, languages and software development tools and efforts often fail. Race conditions and general unexpected behavior caused by interleaved execution are not easy to detect and result in errors that are much harder to resolve since they can be difficult to reproduce and pinpoint their location due to the inherent lack of determinism in execution. Some models of computation or specialized parallel languages exist that alleviate this problem, yet they always bring major limitations, inefficiencies or difficulties along with them. Consequently it is important to be able to investigate such interleaved execution and synchronization mechanisms. Yet even aside such issues, not every program can achieve significant speedups through parallelization in the first place. According to Amdahl's law [8], the sequential part of the application provides an upper bound on the speedup that can be achieved as follows:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

The sequential part r_s is constant, independent of the number of available cores while the parallel part r_p (calculated as $1 - r_s$) is sped up by the factor n, i.e. the number of processors. This equation assumes trivial parallelism, i.e. the overhead due to synchronization or resource contention is not covered. Real performance evaluations will require a prototype that does model these aspects as well. In an extension to Amdahl's work, Gustafson argues that with higher parallel processing power, so does the parallelism in tasks and applications grow [53]. Similarly, if multiple independent applications are executed, it generates trivial parallelism that can easily benefit from a large number of cores. In both cases however, efficient execution depends on many factors, the mapping onto cores and resources, scheduling of processes, etc. These may not be possible in a fully static analysis but may require a prototype.

Programming requirements may also be more fundamental: On the lower software layers the hardware interfaces need to be developed based on specifications that should not be changed afterwards as this may cause major redesign efforts and introduce additional faults. If an early prototype exists however, it may influence the specification process in a positive way. Conversely, driver and low level software development is error prone since there is often no safety net and design mistakes often cause fatal errors.

Considering all these challenges, it is imperative to start working on the software implementation as early as possible. If a physical architecture is available, the software development can take place on such existing system. However, if a novel hardware architecture is designed, a prototype of the hardware is required which is available as soon as possible in the development cycle. Such a prototype needs to mirror the actual hardware as close as possible on a functional level. This means it must provide the correct hardware interfaces according to a fixed specification and also adhere to the specified ISA so compiler customizations or optimization tasks can be handled. As the whole software stack including an operating system needs to be designed, tested, and evaluated, it is furthermore of utmost importance to provide a prototype that allows very fast execution speeds. This can be a challenge due to the size of a many-core. While an ASIC implementation would handle all transistor switching in the platform in parallel, a prototype might have to rely on sequential execution. For extensive testing, hardware components and interfaces for I/O, memory, etc. also must be modeled accurately. Lastly, it is not enough to have a prototype that has a singular output: success or failure. Instead, the progress, state of the prototype, and potential error conditions and situations need to be visible and analyzable.

To summarize the prototyping requirements that come from the programming of many-cores:

- Early availability
- Modeling of fixed/defined interfaces (golden reference)
- High execution speeds
- Visibility and debugging capabilities

4.1.2 Processing

A heterogeneous many-core system contains a large number of processing elements. They range from various GPPs which might exhibit entirely different ISA, GPPs with ISA extensions (e.g. the *i*-core as introduced earlier in section 2.3) and many special accelerators with task specific functionality. They can be clustered and grouped in tiles or work independently as a single entity, i.e. a node in the system that is directly interfaced to an interconnect layer such as a NoC.

Since any processing element in a many-core could also operate on its own or in tandem with a single master, prototyping the individual element can be accomplished entirely by traditional means and will not be discussed in more detail here. Instead, the prototyping requirements when integrating processing elements into a heterogeneous many-core have to be formulated according to their interaction and interfacing towards a full system. Thus, one important aspect is the interaction among processing elements based on contention and limited access to shared resources. Another aspect is handling of heterogeneity and accelerators in general, which adds another level of complexity for the software layers as well as the design-space exploration and architecture optimization tasks.

From a software perspective the requirement mostly lies in functional verification of application mappings or operating system functionality such as schedulers and drivers. Yet from an architecture or hardware/software codesign perspective the performance aspect is of major importance. In order to evaluate the performance of a system, some form of monitoring can be used. On a software level it is typically sufficient to do a profiling or tracing within the application code. On a hardware level however it is important to inspect the operation within and among the processing elements on a much more fine-grain level. Thus, a requirement for the prototyping of processing elements in a many-core is the provision of good visibility of their current state for acquiring and assessing performance metrics. This is of special importance in the development phase, since debugging efforts rely on good accessibility and visibility of the processor state.

To summarize the prototyping requirements that come from the processing elements in a many-core:

- Handling of heterogeneity and accelerators (functional and design process)
- Evaluation of impact on shared resources
- Accessibility and visibility of hardware state

4.1.3 Data

Any form of information processing requires some form of data to operate on. The data can represent either input/output data, configuration data or temporary data that processing elements need for their operation. A processor[1] furthermore requires a program, i.e. a sequence of instructions that can be fetched and executed. In both cases, a memory needs to exist that can store these programs and all required data. A many-core prototype requires memories holding this data that can be accessed by the processing elements. Furthermore it should provide a means of accessing continuous input data not only statically from a memory but also through I/O interfaces towards sensors, actors, or remote systems.

On the most abstract level, it does not matter where the data comes from or where it resides - as long as a sufficient amount of memory is available. If the prototype shall allow accurate evaluations of a real platform however, the exact timing behavior, the memory hierarchy, sizes and access patterns (e.g. burst accesses) need to be considered. Furthermore, memories need to be initialized (typically zeroed) to guarantee a known and well-defined initial state. Afterwards there must be a mechanism for loading the memory with appropriate data and programs before the actual execution starts. These actions need to be considered and provided by a prototype.

All of these mentioned data requirements are the same in any single- or multi-core system. Yet in a many-core, the situation becomes more complex and demanding, making data accesses and memories another major limiting factor for many-core adoption. The tight integration of cores often does not leave much space for memories and the memory resources can quickly become

[1] In this work, a processor is defined by the capability and requirement to execute software. Typically this refers to either a GPP or an ASIP

a bottleneck if multiple cores have to share their accesses. Furthermore, a many-core often introduces additional challenges due to the distributed nature of memories, consistency and coherency issues, or simply the size of the architecture (and thus the prototype).

To summarize the prototyping requirements regarding data handling in a many-core:

- Provide easy to initialize memories of sufficient size (functional)
- Provide means for I/O with sensors, actors, and remote systems
- Allow the investigation of timing details and coherency/consistency issues

4.1.4 Communication

The communication infrastructure is of special importance in a heterogeneous many-core system. In more traditional architectures often only a single master (i.e. the CPU) exists that has full control over its interconnect. The master can decide how and when to use the interconnect in order to access attached slave devices such as memory controllers, I/O, etc. This simple concept was developed further into multi-master systems by adding multiple masters (either several CPU or accelerators like DMA, etc.) on a single bus. In a many-core or many-accelerator system the situation changes even more drastically. Here, a multitude of (often independent) masters require access to a multitude of shared slaves. Many interconnect and communication techniques that have been established in the past (e.g. buses and crossbars) can not be applied to a many-core or many-accelerator system due to their limited scalability. The emergence of networks-on-chip can be seen as an answer towards these scalability issues. These novel interconnects also directly impact the prototyping requirements. Most notably, it is not enough to prototype only small designs and then expect the same behavior when the design is scaled towards larger sizes. A NoC behaves entirely different depending on load scenarios and larger NoC may introduce bottlenecks that need to be considered when designing and parameterizing the interconnect. Thus, a prototype is required to host large designs and scale well with the increased design size.

The location and mapping of processing elements but also the characteristics of application running on these processing elements have a direct impact on the requirements towards the interconnect. This means it is not sufficient to observe basic functionality with artificial test cases. Prototyping the communication in a heterogeneous many-core has to cover the full software stack and use real applications to create realistic scenarios.

To summarize the prototyping requirements regarding on-chip communication in a many-core:

- Allow investigations of large number of nodes
- Allow investigation of interactions among heterogeneous nodes
- Enable fast evaluation of real application patterns including the full software stack

4.2 Motivating Scenarios

In the following, three motivating scenarios are presented that highlight the requirements shown previously by providing real examples of novel designs and features of heterogeneous many-core as developed in various projects. These examples highlight specifically the major challenges that either emerge or which are much more critical in the design process of heterogeneous many-cores compared to more traditional computing architectures.

4.2.1 Dynamic Task Mapping and Runtime System

Heterogeneous many-core adoption still suffers from an abundance of possible bottlenecks and inefficiencies. These make such platforms often perform worse than more traditional architectures, despite having much more throughput and performance potential due to the increased number of processing cores. One such major bottleneck is on the system software level. A many-core requires some kind of runtime system that provides a means for loading the application code, initializing the hardware and starting the execution. However, the full potential of a many-core can only be realized by providing more features than this since a purely statical execution is not able to adapt according to application patterns or environment changes. For example, the image processing of a camera-based obstacle detection on a moving vehicle might require a different framerate depending on the velocity. A more elaborate runtime system is thus desirable that can handle dynamic task mapping among tiles as well as scheduling and threading on individual cores. This in turn requires a management unit that is implemented either centralized, decentralized or as a mixture of both. The management unit needs several basic functionalities that have to be provided by the runtime system: the ability to communicate among its distributed instances, the ability to gather status information potentially also including an API to the application code, and in general the ability to schedule, move and control application execution. Developing such a runtime system including a management unit is a major software development task that is closely correlated with the underlying hardware architecture. Specifically the number of cores, the interconnect, the memories including their hierarchy and beyond all that fundamental design decisions such as the model of computation play a role. Waiting for hardware design teams to implement a set specification and finish all integration tasks wastes valuable time in the design process, requiring approaches that allow system software development to start early and in parallel to hardware design and integration tasks.

4.2.2 Providing a Low-latency on-Chip Interconnect

Increasing the amount of processing elements that are tightly integrated as in a many-core system impacts both the interconnect and memory subsystems[2]. For instance, interconnects have evolved from bus to on-chip networks to promote scalability. The memory hierarchy also needs to evolve as the processing elements beyond tile borders require a coherent view over their caches. Coherence must be enforced by a coherence scheme and supported by fast interconnects. This claim is based on observations reported at NOCS conference that show a significant impact of network latency on coherence messages and schemes [62].

[2] This motivating scenario was described in a publication by the author of this work in [MSK$^+$18]. In this subsection, some parts of the text are taken from said publication with or without modification and will not be identified further.

Traditional inter-tile cache coherence schemes offer global coherence that does not scale. Authors in [116] proposed a novel region-based cache coherence that uses a divide-and-conquer methodology to provide scalable and efficient inter-tile coherence for many-core systems. Due to inherent NUMA patterns of tile-based systems, these coherency regions need to be confined spatially to achieve optimal performance. Consequently, a NoC that can not only alleviate the spatial constraints of region-based cache coherence, but also accelerate the inter-tile coherence messages is desirable for increasing the overall performance of a many-core. While lower communication and memory access latencies will always be a design goal, the potential for gains depends on several factors. In a NoC based many-core, the largest benefits can be expected when a large amount of nodes are interfaced by the NoC since long distances yield the biggest potential for improvements on latency. Long distances impact the amount of hops in the NoC, directly increasing the amount of cycles spent for the communication. At the same time, circuits that enable a direct way of communication can not be kept active since large parts of the communication channels and resources would be blocked. Since future many-cores are expected to grow in size and amount of cores that need to be interconnected, it is imperative to find scalable solutions for this development. However, the design and verification of techniques that could handle or improve this situation and provide low-latency interconnects creates a challenging environment. Large many-core platforms hosting many nodes and a large meshed NoC do not fit in available FPGA boards while taking immense amounts of time in hardware simulation. This calls for novel approaches in the design and verification of such architectures.

4.2.3 High-Level Design Flows for Accelerators and Interconnects

On software level, the runtime system is the most challenging part for heterogeneous many-core while on the hardware side the interconnect and memories have a large impact on performance. However, as motivated in the introduction and backed up by the IRDS roadmap, until entirely new technologies may disrupt the field, future performance gains in computing architectures will emanate from the applications and application domains. These trigger the development of specialized architectures that still show growth potential when progress on the technological level has nearly come to a halt. Since such architectures will make use of optimized interconnects, memory hierarchies and especially custom accelerators, they can no longer be implemented entirely by IP-based reuse of existing components. This leads to more abstraction and automation which will gain ever more importance. Consequently, the design processes and methodologies will follow a top-down approach with the electronic system level at its core. Yet, these methodologies still struggle in many regards as they often produce sub-par results, calling for novel contributions and improvements.

4.3 Analysis and Conceptual Approaches

Based on the requirements and motivating scenarios, approaches towards a holistic solution for the challenges in design and prototyping of heterogeneous many-core are formulated.

4.3.1 Early Prototypes for Software Development

To overcome the challenges of many-core programming, threading libraries such as pthread or specific programming languages (e.g. IBMs x10) that target parallel execution in their language as a core concept, have been devised [84]. Similarly, heterogeneous architectures can be programmed by languages such as OpenCL, which has also been further developed in literature for handling distributed systems that include accelerators [65]. Research towards support of more common matlab/scilab or C language for automated parallelization are also ongoing [38]. On a more basic level, the underlying hardware architecture may support concepts like shared memory programming in order to simplify the distributed nature of memories in a many-core.

Despite all these approaches to ease programming, the process remains error prone and requires prototyping to help in debugging and verification tasks. While software can be developed hardware independent by adding a hardware abstraction or intermediate layer [132], this often results in lower performance and compute efficiency. In recent years, a number of full-system simulators have emerged to help in this task [15]. These are often built for fast execution speeds and early design space exploration, yet some also operate on more accurate levels. Even though not natively targeted towards it, simulators such as Gem5 have been extended to enable many-core prototyping [21] [22]. However, their focus typically lies on micro-architectures, making the environment less suited for interconnect simulation and multiple core interactions. At the same time, they typically do not provide the same execution speeds as true Electronic Systems Level (ESL) prototypes. Specifically virtual platforms that incorporate an ISS have mostly been overlooked in the context of many-cores, despite having some appealing characteristics. Most notably, they provide the highest execution speed possible, nearly at the same speed as a native execution on a host-PC. Performance is only limited by the amount of physical cores on the host system, which is typically less than the prototyped virtual platform[3]. This makes multiplexing necessary, resulting in performance degradation depending on how many virtual cores and other virtual components need to share a physical core. A virtual platform that uses an ISS at its core also enables heterogeneous architectures to be build. If a modeling scheme for arbitrary components such as interconnect or accelerators is present, a VP can provide all the tools necessary for simulating a heterogeneous many-core. This allows the software development efforts to start early, reducing overall development cycles and also enables the hardware/software codesign (i.e. the software development can influence the development of the hardware architecture and vice versa). Abstract prototypes also offer further benefits that can make them useful even when a physical architecture already exists: they can be executed on regular and widely available COTS processors, they can be better observed and instrumented non-intrusively and performance can even be higher (when the target is a low clock rate embedded device for example). Finally, a virtual prototype allows to introduce different timings, thus triggering race conditions and improper coding more easily. A prototype that enables design, verification and validation of software and programming needs to be fast enough to handle high levels of parallelism and large applications and data sets. However, these platforms also need to provide real world data input from sensors or I/O devices such as cameras to enable the efficient prototyping of the full software stack in real scenarios.

[3] This is due to the fact that the prototyped design represents a many-core while the simulation host in most cases is a COTS system

4.3.2 Hardware Verification and Validation

While virtual platforms are deemed to be suitable for software development and early prototyping, by itself they cannot be used for hardware design and verification. Instead, hardware simulation is the common approach for the verification, validation and debugging in integrated circuit design. Its benefits are relatively fast recompilation after design changes even in large designs, good visibility of signal and register states, and overall cycle-accurate behavior. In a many-core architecture, hardware simulation still plays an integral role, yet its weaknesses are even more prevalent. The sheer size of a many-core makes hardware simulation incredibly slow since it has to scale with the increased amount of logic. Parallelizing the simulators can alleviate the situation and is investigated in many scientific works. Yet there is still the limitation of available physical cores and synchronization that needs to be applied on a cycle-by-cycle basis in order to guarantee accurate and deterministic behavior as required for verification tasks. The slow speed and lacking scalability makes it infeasible to use hardware simulations for more than some debugging tasks of individual components and logic blocks. Overall integration into a full system and performance evaluations need to be carried out another way. While it is possible to simulate memories and put real binaries in them, the requirements regarding programming can not be fulfilled by hardware simulation simply because it is far too slow. Similarly, even the other requirements can barely be fulfilled for the same reason, since complex interactions of heterogeneous processing elements often only show in extended use-cases triggering corner cases.

FPGA based prototyping does provide scalability due to the fully parallel nature of FPGA execution. Yet at the same time, resources are limited and large many-core designs do not fit onto todays FPGA chips. Virtualization of resources can help, yet it reduces performance significantly while increasing design complexity. Multi-FPGA solutions on the other hand can host reasonably large many-core designs, yet there is extra complexity due to the mapping step onto the individual FPGA. Furthermore, these prototyping systems are very expensive and come with a number of limitations, such as the number of pins on each FPGA that not only limit I/O but also inter-FPGA connections. Additionally, synthesis times grow with design size, resulting in extensively long iterations for design changes.

Instead of a full system, single components of a many-core can also be prototyped. Many individual components of a many-core do not require adaptations in their prototyping approach. However, the situation is entirely different for the network-on-chip interconnect. This core component of a many-core has a large impact on performance, non-functional properties and provision of QoS guarantees. A NoC is a highly flexible component that may come in many variations that perform better or worse depending on several factors such as network load, network size, topology and application patterns. Consequently, there is a selection of pure interconnect simulators, typically targeting network-on-chip interconnects. However, most of these simulators only use application traces or synthetic traffic as input. Some simulators use an interface to the host processor for executing application code and feeding live input data through a wrapper into the interconnect simulator. This approach however discards any timing synchronization or accuracy in the process. Furthermore, a custom interface to the simulator is required which has no relation to the physical interface that a full system uses after the NoC has been integrated into a many-core platform. This means low level driver code will not be compatible and even the end-to-end network protocol stack might be different in the prototype compared to a physical implementation. Similarly, the software will also not be binary compatible, making the critical task of driver and low level software development impossible. While unit-testing is an important and established design technique, the full system view on an abstract and on a detailed level is required during the design process as well. Thus, these approaches are not sufficient for a many-core prototype since neither the performance, nor the functionality can be evaluated by isolated prototypes.

4.3.3 Design Automation and Abstraction

Coping with rising design complexity not only means the target technology for a prototype needs to keep up with this development but also the design tools and the design methodology itself needs to do so. This is mostly achieved by novel languages and advances in Electronic Design Automation (EDA). Introduction of the Register Transfer Level (RTL) and support for it by synthesis and hardware simulation tools was a major step in this direction. Yet the productivity ceiling needs to be raised again by introducing even more abstract modeling and descriptions that are accordingly supported by EDA tools. The electronic system level can be considered as an overarching title for many different approaches towards more abstraction. Yet many of the techniques on that level still do not work well as Quality of Result (QoR), input language support and design cycle times are still far from optimal. In this context, it is important for the many-core design and development process to investigate which technologies and languages promise increased productivity and ease in the design process towards a large degree of parallelism, heterogeneity, optimized interconnect and a well crafted memory hierarchy.

4.4 Summary

The following conclusions are drawn from the analysis of the requirements and state of the art:

Many traditional techniques for IC design are still going strong for heterogeneous many-cores. Yet in the face of sheer design size, they reach their limit and start to struggle due to scalability issues. Even though there are ways to improve scalability such as simulation parallelization or multi-FPGA platforms, these cannot fully solve the situation and often come with additional overhead, limitations or simply added cost and complexity. Novel approaches such as virtualization, known mostly in the server domain, are already in use for single core design prototyping but promise further potential also for many-cores. Lastly, raising the abstraction level and increasing design automation is a constant ongoing process, yet many-cores might push the need for further advances in this process forward due to their high level of complexity.

In summary, three major topics for improving the process of many-core design are determined as follows:

- Provide an early available Virtual Platform representation that allows the definition of clear hardware interfaces, allowing software development to run in parallel to the HW design and enable HW/SW codesign techniques.
- Enhancing HW design and verification that is traditionally based on hardware simulation and emulation by moving towards hybrid multi-level prototyping.

- Fighting the complexity by raising the abstraction level and automation in many-core design by enhancing and extending existing languages and toolflows for design space exploration and automated hardware generation.

These three topics set the theme for the following chapters and will each be discussed in more detail in a separate chapter. For each it will be described how they can be realized, what challenges must be overcome, and what can be added to improve the current state of the art. The second topic will also introduce a novel concept for reducing latency in a NoC, which motivates the novel hardware prototyping approach that is the central element of this chapter. The final chapter will also include a description of the envisioned design and verification methodology that is realized by a specialized framework.

5 Virtual Platforms for Heterogeneous Many-core

In this chapter, the prototyping of many-core architectures on an entirely virtual level is presented. The focus of this chapter will be on prototyping methods that enable early available models for software development and interface specification. Based on the requirements and analysis of existing approaches, Virtual Platforms (VP) have been determined as the most promising method for achieving these goals. Thus, in the following it will be described how a VP can be efficiently used for many-core prototyping, what extensions and enhancements are necessary and how the resulting platforms perform.

5.1 Building a Basic Many-core Prototype in OVP

In this section, the extension of a virtual platform towards a full many-core architecture will be described. The Open Virtual Platforms (OVP) framework is selected for this task [3]. OVP supports a wide range of different ISA in form of Instruction Set Simulators (ISS) and is available in a free of charge version. It also contains a special modeling environment for peripherals that is executed by the Peripheral Simulation Engine (PSE) and furthermore allows the definition of custom ISA processor models. Native host execution through semihost functionality that can be included as part of a platform enables even more functionality and performance. All these features result in full flexibility and extensibility as it is needed for modeling a heterogeneous many-core while still providing the fast execution speeds of a fully boosted virtual platform utilizing binary translation at its core.

When designing a many-core in OVP, the first step is to instantiate a CPU and connect it to a local bus. Similarly, multiple CPU can be connected onto the same bus system, creating a multicore design. The bus serves more as a logical connection, since no real bus contention is modeled in OVP. It does however provide a separation of the memory maps which allows the modeling of tile-based architectures with distributed or distributed shared-memory. The CPU are chosen from a list of available models in the OVP library, among which are ARM, or1k, MIPS and Microblaze models. In a multi/many-core VP, a mix of designs with different ISA can be chosen to introduce heterogeneity. In this case, each processor must load a separate binary that is cross-compiled for the respective ISA. It is also possible to extend the OVP library by using the binary translation scheme of OVP. This means writing a CPU model and the according mapping of all instruction from its ISA to the host ISA (x86). Such a custom CPU model enables the investigation of any architecture for scientific purpose.

In the most simple architecture, there are a number of CPUs connected to a memory via the bus. The memory can be directly loaded with an application code and the so created multi-core tile can be executed by the OVP simulator. In order to build a full many-core in OVP, at least a custom peripheral needs to be added that can take over the shape and functionality of a NoC. Such a peripheral is introduced in [MWB15] and briefly mentioned in [JSK$^+$15]. There, a so called ”communication device” represents an interconnect peripheral. It can be designed according to a hardware specification of a NoC. The goal is to provide the exact same interface to the software as the actual hardware implementation. Thus, no extra glue code or an additional hardware abstraction layer is required when executing applications on either platform (VP or hardware). Even drivers that handle communication can be operated on the virtual prototype exactly as on real hardware. For a more detailed NoC implementation beyond a functionally correct interface, the OVP environment can also be interfaced to a SystemC based NoC simulator [100].

5.2 Accelerator Modeling in Virtual Platforms

Future computing architectures are expected to become more heterogeneous, hosting a variety of processing elements and accelerators. A major difficulty is the handling of these accelerators, i.e. when to map and execute which parts of an application on what accelerator, how to handle low level interfacing and data exchange efficiently and how to integrate accelerators into the HW platform and the SW stack. Thus, there is a need for prototyping of accelerator-rich many-cores and many-accelerator systems. Many of the difficulties can be investigated on the SW level independent of hardware details, making virtual platforms a suitable prototyping environment. As described previously, OVP can easily be extended towards many-core prototyping due to their peripheral modeling environment. Accelerators are not specifically foreseen in the OVP framework, yet they can also be introduced nonetheless as described in the following. In this case, the approach will be investigated on a much more detailed level, to highlight the various options and possibilities of virtual platforms. The presented work was published in [MWB15][1].

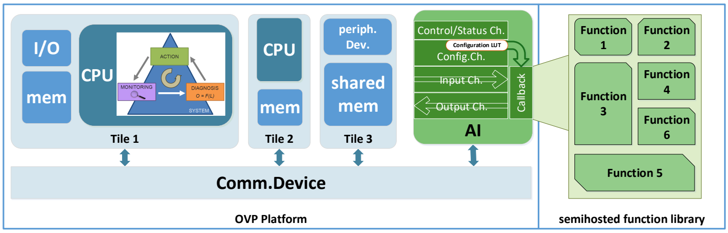


Figure 5.1: Modeling of the FlexTiles architecture including the AI in a virtual platform.

In Figure 5.1, an excerpt of a many-core architecture based on FlexTiles as introduced in section 2.1.3 is shown that is modeled within OVP. Depicted are a monitoring tile (Tile 1), a regular computing tile (Tile 2) and a memory tile (Tile 3), all interconnected by the communication device. The architecture may contain many more computing or memory tiles yet these are not relevant in this example. To the right is the Accelerator Interface (AI), which serves as a configurable interface to a reconfigurable fabric. This fabric hosts a number of run-time reconfigurable accelerators. The AI provides a well-defined interface to the NoC and is thus accessible remotely by the software on the regular computing tiles. The interface consists of a control and status channel, a configuration channel, an input and an output channel.

In OVP, the AI can be modeled as a peripheral that is hooked up to a dummy tile. The dummy tile is then connected to the rest of the platform via the

[1] Extracts from [MWB15], which were completely written by the author of the work in hand, are used verbatim in this section without further identification

communication device. The AI peripheral models its interface including all channels according to the hardware specification. To model the actual reconfigurable fabric, either the PSE environment or the semihosted functionality of OVP can be used. However, PSE execution blocks the platform and is typically much slower than native execution in the semihosted environment. Thus, the semihosted implementation is preferable.

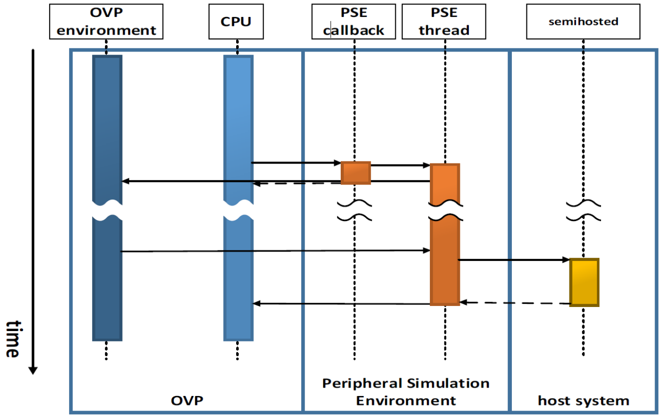


Figure 5.2: Sequence diagram highlighting the execution order of the different OVP environments.

The sequence of execution in an optimized implementation is shown in Figure 5.2. In the OVP environment, the application code that is executed on a virtual CPU calls an AI driver function to start a calculation on a specified accelerator. This first triggers the callback of the communication device in the PSE which listens to a memory range which is accessed by the function call. The communication device in turn handles the request of the triggering function and forwards the trigger to the AI peripheral in the PSE. The AI peripheral then instantiates a PSE thread and returns operation to the OVP simulation to continue its execution. The PSE thread has a library of pre-compiled functions (seen in Figure 5.1) that are executed in the semihosted environment. When the accelerator execution is finished, the PSE thread provides the result back to the CPU and destroys itself afterwards.

In Table 5.1 an investigation of several algorithms is carried out to determine the speeds when they are executed either on an ARM model, a microblaze

Algorithm	Variant	Sim time in s	Instructions	Speedup factor
Susan	Semih	0.04	<1000	45x
	PSE	0.23	<1000	8x
	ARM	0.23	176,408,769	8x
	uB	1,81	508,252,089	1x
Quick-sort	Semih.	0.44	<1000	14x
	PSE	2.23	<1000	3x
	ARM	2.24	2,512,431,935	3x
	uB	6.34	2,670,876,722	1x
LZ77	Semih.	0.1	<1000	17x
	PSE	0.63	<1000	3x
	ARM	0.55	702,362,833	3x
	uB	1.69	662,720, 033	1x
DCT	Semih.	0.15	<1000	49x
	PSE	0.48	<1000	15x
	ARM	0.37	381,480,972	20x
	uB	7.28	2,702,877,555	1x

Table 5.1: Execution time measurements comparing various implementation alternatives in OVP.

model, a PSE accelerator or finally the optimized semihosted accelerator. The results show that the ARM model is superior to the microblaze model in OVP, yet this is also partially due to a better compiler support which results in less instructions necessary for the same task. The PSE accelerator achieves similar speeds to the ARM model. The semihosted accelerator however achieves much faster speeds, up to 49x faster than the microblaze model in the DCT algorithm. This means by employing a semihosted implementation, the execution of the VP is actually sped up when using accelerators. It does not mean however that a direct correlation between total execution time and performance of a hardware implementation exists. Instead, the goal is to provide a means of fast debugging and functional verification. Still, similar to other approaches that bring some level of accuracy to OVP, the accelerators can also be annotated or extended by delay models that provide a means for assessing their performance.

5.3 Real-world I/O for Virtual Platforms

Any useful information processing system needs to operate on some form of input data and produce some output data. In High-Performance Computing (HPC), the data sets are often provided through a streaming interface or they are fetched ahead of execution into a large memory block. In a CPS or embedded multi/many-core however, the input typically comes from a user, sensor or camera while the output can be visualized to a user or serve as an input trigger of another connected device or system. The previous chapters described how OVP can be used and extended towards heterogeneous many-core architectures but it never touched the question any real system needs to answer: which data does it operate on and where does that data come from. In OVP, programs and data can be loaded directly into a virtual memory in the platform. For simple test cases this form of I/O is sufficient, since a VP can be observed via debugging channels and the memories can be read out manually. A real system however needs to operate on live data - a prototype without this capability is severely limited by often artificial input data. This necessitates a mechanism for connecting physical I/O devices to a virtual platform. The work presented in the following was published in [WMLB15][2].

The major difficulty when solving this issue is the encapsulation of a virtual platform. The situation when using OVP can be seen in Figure 5.3. Any data

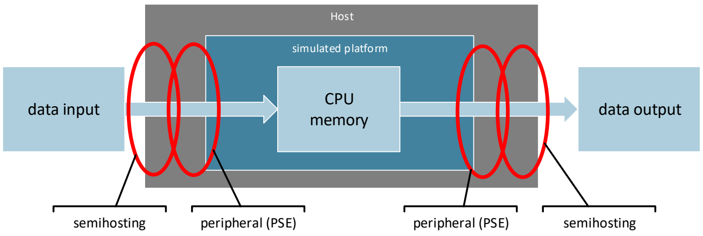


Figure 5.3: Encapsulations in an OVP platform. The red circles mark the borders that must be crossed to access host resources such as connected I/O devices.

[2] Extracts from [WMLB15], which were completely written by the author of the work in hand, are used verbatim in this section without further identification

input and output must at least cross two interfaces, i.e. one towards the host system and one towards the virtual platform. When designing a real-world I/O device for OVP, an essential requirement is that the I/O behaves exactly as it would on a physical hardware, at least from a processor and software perspective. For this task, the Peripheral Simulation Engine (PSE) in OVP can be used to model an interface exactly according to hardware specifications. The PSE environment however is still an encapsulated part of the simulator and does not have direct access to the host's file system. Therefore, the semihosting functionality of OVP needs to be employed. This feature works with function interception on function calls in the PSE and gives access to all the features of the hosts operating system. When a function call is intercepted, the simulator executes a corresponding function available in the semihost library, which runs natively on the host and can access its resources like files stored on the host by using standard C functions, for instance fopen() and fgets(). The PSE together with the semihost functionality can be used to model a custom peripheral that provides I/O data from the hosts file system. However, it is also possible to extend this feature by using the ”everything is a file” principle of Linux/Unix to include device files as a source. These kind of files represent real hardware devices in the Linux file system and allow similar access to every type of character device available on Linux as on text files

OVP allows three methods for realizing the access to the host's file system. These will be presented and discussed in the following.

5.3.1 PSE Callback Method

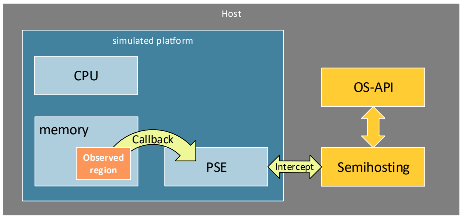


Figure 5.4: Callback method.

The first method is based on PSE callbacks as highlighted in Figure 5.4. In OVP, the common way peripherals work is through callback functions that are associated with a certain memory range. When a bus request takes place (typically initiated by a CPU) it is checked whether the memory range of a peripheral is accessed. In such a case, the registered callback is triggered, passing the address and assigned value as a parameter. Since the callback is still within the encapsulated PSE environment, the use of additional simulator API functions or a function interception is required for getting access to host resources. Such a semihosted function that intercepts a PSE call runs outside the simulated platform but all parameters of the call from within the simulator are passed to it.

A big advantage of the callback approach is, that some functions like endianess conversion can be implemented to automatically apply during operation. Furthermore only copies are transferred and thus the simulation environment cannot be corrupted. However, the ”call-by-value”-fashion leads to a high overhead as well. On one hand the memory requirement doubles, on the other hand additional intercept functions are needed. Another disadvantage is the missing option of remapping existing callback regions. Thus, the method is only suitable if little amount of data is transferred or if the memory region does not change.

5.3.2 VMI Runtime

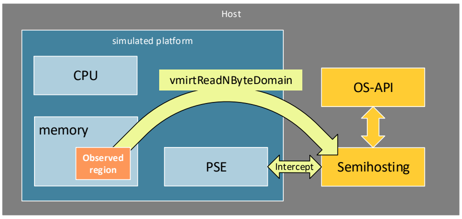


Figure 5.5: Special semihost function call.

By accessing the Virtual Machine Interface (VMI) runtime, the semihost has access to the virtual memory management of the OVP simulator. The function vmirtReadNByteDomain() for instance can be used to transfer data from memory of the virtual platform into an array which is allocated in the semihosting. Compared to the callback method, transferring large memory ranges is possible and remapping is trivial, since only the source address has to be adapted. A potentially big disadvantage is the fact that the whole memory area must be copied resulting in a big overhead if there were only little changes since the last transfer. Furthermore, the memory use is doubled because the semihost holds a copy of the guest memory. Transfers with VMI-runtime functions are fast and a good choice for large data blocks, which are often modified and have to be transmitted as a whole. Figure 5.5 depicts the corresponding schematic.

5.3.3 Native Mapping

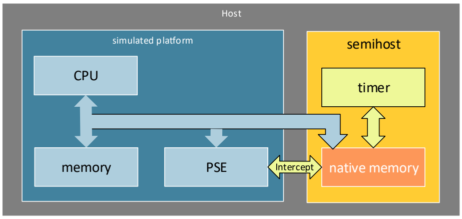


Figure 5.6: Native mapping.

A final method is able to completely avoid any memory copy operations by providing a single memory that is directly accessible from within and from outside of the simulated platform. This method is called ”native mapping” and is shown in Figure 5.6. In this approach, part of the memory map in the simulated platform is mapped onto a memory range on the host system, circumventing the memory management of the simulator entirely. This allows the direct access of host resources from within the virtual platform. However, this also means that a developer needs to ensure that there is a valid mapping during the simulation. To perform a remapping, it is needed to unaliase a previously mapped memory region using vmirtUnaliasMemory() and configure it again as regular memory which is managed by the simulator using vmirtMapMemory(). Since the (re)mapping is performed during runtime and vmirtMapNativeMemory() does not copy data, values stored at the location to be mapped must be copied into the native memory manually. Otherwise, the data will be lost. Similarly, data stored in the native buffer must be copied in the internally managed memory when the buffer is unmapped. This approach has the benefit of requiring less memory (no copies need to exist) and remapping of memory regions at runtime is possible.

5.3.4 Synchronization and Overhead Reduction

There are two methods to synchronize the communication of the virtual platform with its environment. The simplest one is to implement waiting time within a peripheral using bhmWaitDelay() to force the PSE to wait and perform a callback after a predefined period of time. The waiting time is proportional to the simulated time. Another option is the use of multi-threading in the semihost to allow using POSIX threads. These threads are executed independently of the simulation. This can be used to implement waiting times relative to the host machine clock by calling standard functions like usleep(). This is useful when interacting with hardware devices connected to the host or working with fixed frequency device like cameras which update the framebuffer every 40ms (25fps). Additionally, this method allows performing tasks in the semihosting without triggering an intercept function.

Since external I/O devices may run at specified frame rates or provide new inputs only after long delays in real time, an application may spend a long time idle. On a host system with a real I/O device attached this can not be prevented, however sleep calls at least allow other processes to take over the available processing resources. However, this situation changes in a virtual platform when using files with prerecorded data as input. Here, the wait times and the performance can be optimized with a tuning factor. The concept is shown in Figure 5.7. The top chart shows the default implementation that is calling bhmWaitDelay() according to the input delay of the I/O device. This delay can be reduced with the tuning factor, effectively providing new input updates much quicker to the platform. It can be seen as fast-forwarding a simulation when the physical processing power is potent enough to allow running faster than real-time. Although this is not possible in a live system, it does speed up the evaluation process in a prototype, especially if the modeled hardware is an embedded system without much processing power.

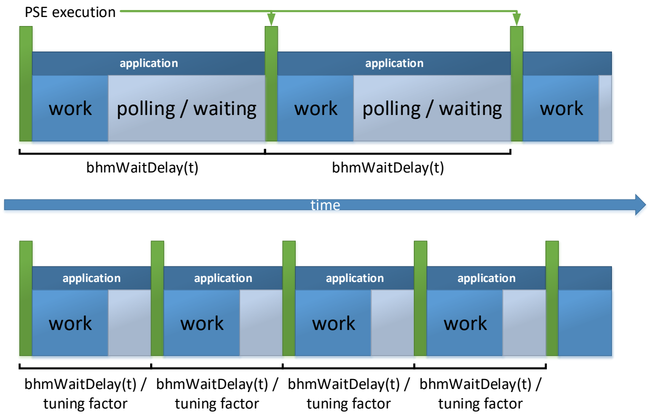


Figure 5.7: Thread interleaving and tuning factor.

Method	Data rate (native)
Callback frequency	1,579,778 Callbacks/s
Initializing internal memory	32.18 MiB/s
Writing in internal memory	345.72 MiB/s
Callback copy	5.78 MiB/s
vmiRtReadNByteDomain()	355.80 MiB/s
Native mapping	481.73 MiB/s

Table 5.2: Throughput measurements of OVP methods.

A performance evaluation of the different methods allowing data exchange between the virtual platform and the host is shown in Table 5.2. A minimized virtual platform consisting of a MicroBlaze system and a peripheral with an internal buffer of 64 MiB is used to perform the data transfers and measure throughputs. The application executed by the MicroBlaze fills the buffer ten times with random numbers. To determine and remove the overhead caused

Done by	Output mode	Framerate [fps]	Data rate [MiB/s]
PSE	SDL	242	283,59
	DLO	20.2	23,67
Microblaze (calculated)	SDL	61.88	72.52
	DLO	71.33	83.59
Microblaze (output)	SDL	22.83	26.75
	DLO	12.20	15.46

Table 5.3: Performance measurements of video output.

by the callback mechanism itself from the measurement of the throughput, the application calls 10,000 times a dummy function in the semihosting after start up. Besides the performance of the three methods, the throughput of the memory management of OVP is measured as well, to serve as a reference.

Using memory natively allocated in the semihosting provides the highest throughput of the proposed methods. Due to avoiding any overhead caused by the simulator it is even faster than memory accesses within the virtual platform. Another advantage is, that no additional memory is needed. However, using the native mapping of memory allocated in the semihosting into the virtual platform, the developer is responsible for managing the memory. He must ensure its validity during the simulation, since OVP cannot perform any checks. The biggest benefit of this method is the possibility to allocate memory from the Linux kernel space using mmap() in the semihosting and making it available to the simulated system in the virtual platform. Compared to regular memory allocations in user space as performed by the OVP simulator, this memory is physically contiguous as needed by network adapters and graphics cards usually used in conjunction with DMA devices, connected to the host. After measuring the pure throughput, the native memory method is implemented in a peripheral simulating the DVIoutput IP-core provided by Xilinx which is added to the minimized virtual platform mentioned before. With this setup the maximum performance of the video output is measured and serves as a reference to determine a possible slow down caused by image processing applications executed by a virtual platform. The DVI-out peripheral is connected to a virtual screen emulated by Simple Directmedia Layer (SDL) and to an external graphics card using DisplayLink (DLO). For evaluation purposes an application is used which draws a contiguous moving rainbow. This ensures that each pixel is changed compared to the previous frame and no compression side effects caused by external driver codes affect the measurements since all data has to be transferred in each iteration. The results are shown in Table 5.3. At first, this application is implemented directly in the peripheral to determine the maximum performance possible. When using the virtual screen, the virtual platform writes the rainbow with more than 240 fps. When the peripheral is connected to the external DisplayLink device, the achievable framerate is lowered to about 20 fps. In contrast to the SDL connection, this method does not load the internal graphics card of the host machine and the CPU load is at 5%. The slow down can be explained by the fact that the implementation uses the open source version of the DisplayLink drivers only allowing uncompressed data transfer via USB 2.0. The achieved framerate equals an achieved data rate of 23.67 MiB/s. In contrast to the native semihosted execution performed by the peripheral, the maximum output performance is measured when the rainbow application is executed on the simulated processor as well. In this scenario two framerates are measured for SDL and DLO output. The first framerate indicates the maximum framerate the virtual platform can provide without any interaction with the host system, since it is determined by measuring the time needed to write 1,000 frames in the framebuffer. The second framerate shows the number of frames effectively written by the DVI-out peripheral. As expected, the output on the virtual screen is faster than the output via external graphics card (22 fps to 13 fps). An interesting effect is, that the number of frames calculated by the virtual platform (”MB (calculated)”) is higher when using the DisplayLink adapter due to less CPU load, while the number of frames written (”MB (output)”) is less due to the overhead for interacting with the USB subsystem.

5.4 OVP Parallelization

The binary translation used by OVP allows very fast simulation of CPU models. In the previous sections it is shown that also accelerators and peripherals for modeling of other vital parts of a many-core platform can be realized in a high-performing manner. However, when scaling up the number of cores and elements in a many-core, the simulation still has to share the underlying host resources of the physical machine it is running on. This creates a physical limitation of the performance in memory and processing for large architectures. Selecting a host machine with higher processing power helps to alleviate this problem but it also means the simulation needs to make use of this higher processing power. Since such increased performance is typically achieved by parallelization, the simulator must also be able to parallelize its execution onto the available physical cores on the host.

Parallelization of a many-core platform in OVP requires splitting of the architecture into several independent parts. The available cores of a host system are a reasonable target number of partitions that a platform is split into since it reduces context switching while employing all available compute resources. However, it may also make sense to partition virtual cores according to some other metric in order to improve load-balancing. Each partition can be executed either by a separate thread within an OVP wrapper or as an individual linux/unix process. When splitting an architecture, the cuts must be considered in all parts. Specifically, in a tile-based architecture the cut can be either within a local bus or within the NoC interconnect. Since the former is only a logical construct, the cut implies that any memory and any peripheral that is connected to this bus must be shared among all partitions that contain a CPU connected to the same bus. In both cases, the semihosted functionality of OVP is required to implement the sharing of resources among partitions. Semihosting allows access to host resources and thus the interfacing of independent processes and threads. For memories, the semihosting is used to map native memory into the memory map of each partition. This allows a seamless shared memory approach while giving full independent execution of the CPU split into their respective partitions. Peripherals on the other hand have two options: Either duplication in each partition or a shared implementation in the semihost. Duplication is the trivial approach that gives full functionality and full execution speed in each partition. However, it also means accesses to the peripherals are completely isolated. Furthermore, peripherals may hold physical resources on the host machine that cannot be shared. If either of these conditions collide with the intended functionality of a peripheral, duplication cannot be used. Instead, a shared implementation in the semihost has to be utilized. In such a case, each partition is extended by a dummy peripheral that is instantiated in their respective platform. This peripheral then triggers a shared single entity of this peripheral in the semihost. In order to provide synchronization, a client-server approach is used between dummy peripheral and the shared master peripheral. Each dummy registers as a client with the shared master and can post requests towards the master. Since the master is a single instance, it is able to hold non-shareable resources of the host system, enabling access for all partitions to such resources.

5.5 Summary

In this chapter, a virtual platform based on OVP was extended towards the modeling of a heterogeneous many-core platform. Specifically, a communication device, an accelerator interface and a real-world I/O interface were introduced and described. By incorporating these new components, virtual platforms can be used as early available and high performing prototypes that are required for early design space exploration and software development. The communication device abstracts the NoC implementation but provides the exact same interface and behavior of a hardware design, making the software binary compatible. The accelerator interface allows the modeling of generic accelerators that behave the same as hardware IP blocks while even speeding up simulation due to the native code execution that is utilized. The real world I/O interface allows direct access to input data from files on the host including device files as used by cameras and video screens, providing live data for evaluating the virtual architecture and the software stack. Virtual platforms are not only fast, but also enable and ease debugging and evaluation tasks. In order to retain scalability in the context of many-cores, approaches for parallelization of the VP were presented.

6 Scalable Hardware Design and Verification

In this chapter, prototyping methodologies for hardware design and verification are presented. According to the analysis earlier in subsection 4.3.2, hardware simulation does not provide the execution speed to handle many-core architectures and can thus only be used in the development process of small sub-components and in debugging efforts when the root cause has been narrowed down significantly. Consequently, hardware emulation techniques need to be employed. However, specialized hardware emulators induce large overheads and come at immense cost. More generic and easily available FPGA based prototypes are the remaining solution that does not suffer from bad scalability regarding execution times. However, the massive design size and complexity of many-cores even pushes FPGA prototyping to its limits, requiring multi-FPGA solutions that come alongside additional limitations, challenges and costs. In order to overcome this situation and provide a different approach that is cost efficient and easily available while being scalable and high performing, a novel multi-level prototyping approach is introduced in this chapter. To give a further motivation of this approach, a novel hardware feature for many-core architectures is introduced in the following that emphasizes this challenge. The feature extends a NoC to enable low latency communication that unfolds its potential only in larger architectures with many interconnected nodes or tiles. While first evaluations were carried out in a cycle accurate but non-synthesizable SystemC simulation, the RTL design, debugging and verification required the support of FPGA prototyping. Since state-of-the-art FPGA methodologies did not suffice, the aforementioned novel hybrid prototyping approach was developed and successfully used in this context.

6.1 The In-NoC-Circuits

In this section, a novel architecture for low latency NoC interconnects based on shortcuts in a hybrid NoC is introduced. This work was published in [MSK+18][1].

State-of-the-art hybrid NoCs have in common that they are always employed for taking advantage of the positive aspects of both circuit and packet switching. Circuit switching can offer strict QoS guarantees, low latency (on established circuits) and lower area/power consumption, mostly due to less buffer space that is required in the routers. Packet switching on the other hand typically has a better link utilization, simple implementation and no circuit setup/tear down delay. Contrarily, these are the major downsides of circuits since in many situations they have to be established and teared down often, leading to higher latency. Alternatively, they are kept alive even when data is not continuously transferred which in turn decreases link utilization since each end-to-end circuit that is kept alive blocks its share of the link (i.e. either time slots in case of TDM or link resources in case of SDM).

The advantages and disadvantages of packet, respectively circuit-switched on-chip interconnects are well understood and have led to the publication of numerous techniques trying to alleviate the disadvantages of each approach. Alongside specific optimizations for packet or circuit switching individually, hybrid architectures try to combine both approaches in a best of both world scenario towards the perfect interconnect. In one such early work the authors analyze optimal on-chip interconnects and then present a packet-switched architecture which is capable of also establishing circuits, called Express Virtual Channels (EVC) [67]. These EVC may circumvent the stages of the router pipeline in the packet-switched network and thus improve latency. Another approach of a hybrid NoC focuses on the splitting of link resources between a so called Pnet and a Cnet [86]. Packets may travel on one such network and switch between them multiple times in order to reach their destination. The same authors later present shortcut paths in [124] as well as partial circuits [123], which can be seen as extensions to their initial work. The shortcut paths include a traffic monitoring unit to help in the construction of shortcut

[1] Extracts from [MSK+18], which were completely written by the author of the work in hand, are used verbatim in this section without further identification

flows. The partial circuits introduce packets, that constantly request circuits towards their destination while being transmitted over the packet-switched network.

These works have in common that they specifically have a synchronous packet-switching architecture in mind, which uses synchronous circuit switching in order to reduce cycles in the routers by skipping pipeline stages. Such techniques are rather simple to implement, yet do not allow for large reductions in overall latency. A major goal of these works is thus the low power design of the NoC, instead of a focus on latency as it is needed for cache coherence messages and remote read operations. Furthermore, the works are purely based on common interconnect simulators and do not portray a real HDL implemented and FPGA prototyped many-core architecture.

Cache coherent many-core architectures have also been investigated in regards to a networks-on-chip interconnect. Significant research effort is put into solving the challenge of cache miss prediction and the design of advanced coherence protocols. One approach is to characterize communication behavior based on synchronization points and use a small hardware unit to observe and predict the target of each coherence message at run-time [36]. In [62], the authors reaffirm the significant impact of network latency on performance in a cache-coherent chip multiprocessor (CMP) system. In order to improve latency, they propose to use circuit switching in combination with a specially tailored coherence protocol that allows for a better prediction of coherence traffic. Their circuits start from injecting routers and improve latency due to a pipeline bypass that achieves better results especially in high-load scenarios. In [7], the authors specifically target remote memory access latency improvements by using circuits that are established ahead of time. Their circuit setup is triggered by a memory request and is aligned with the delay in the memory controller, achieving circuits that are only active in the exact cycles when the data is provided by the memory controller. The authors in [66] present a new approach for single-cycle multi-hop circuits within a synchronous router with a single link. The idea is based on an asynchronous setup network that will configure a multi-hop path from a source router to a destination that uses asynchronous repeaters within the crossbar of each router to transmit the data in a single cycle. This approach can provide near instant communication even over several hops even when only a single link is available. However, there is still a small setup delay for the circuits since they are not kept alive and the PS-links on the circuit path are blocked for any other traffic during transmission.

In the following, a novel concept for low latency interconnects called In-NoC-Circuits (INC) is introduced. The INC concept blends well with multiple regions/clusters of cache coherent tiles as it can service two adjacent yet different coherency regions as opposed to more costly all-to-all circuits. The INC were developed as part of the InvasIC many-core architecture introduced in section 2.1.3 that consists of a tile based MPSoC where every tile houses several processing elements coupled with their private L1 caches connected via a bus interface. The intra-tile L1 caches are kept coherent using a bus snooping protocol. The tile also contains a shared L2 cache along with a Tile Local Memory (TLM). Region-based cache coherence is used to maintain coherence beyond tile borders, i.e. between inter-tile L2 caches. The primary goal of the INC concept is to monitor the inter-tile coherence traffic within and/or between different coherency regions to dynamically create dedicated INCs that accelerate the performance of the overall system.

The INCs are intended to enhance and improve specifically hybrid networks-on-chip (NoC) interconnects for many-core architectures. Hybrid NoCs have been proposed in many different variations, however in this work, the general definition of a hybrid NoC as a combination of packet-switching (PS) and circuit-switching (CS) schemes in the same on-chip network is assumed. In a hybrid NoC, these two schemes may be implemented by sharing the same physical link utilizing sub-links based on TDM or SDM techniques. Alternatively, they can be implemented based on a separate physical link (i.e. a separate layer) for each network. There are further variations based on synchronous or asynchronous control and datapath in the routers. In this case and in contrast to the state of the art, a separate physical link and a synchronous router architecture together with asynchronous transfers on the datapath in the circuit-switched network are used. This is due to the fact that the proposed technique shows its largest potential in an architecture where transmission delay in the circuits is only limited by the wire delay. The concept however is applicable to all such variations and will only differ in the magnitude of its latency gains.

As discussed earlier, a common approach to counter the downside of circuit switching is to employ predictive techniques to reduce setup and tear-down delay. The in-NoC-circuits follow a different approach with the goal of setting

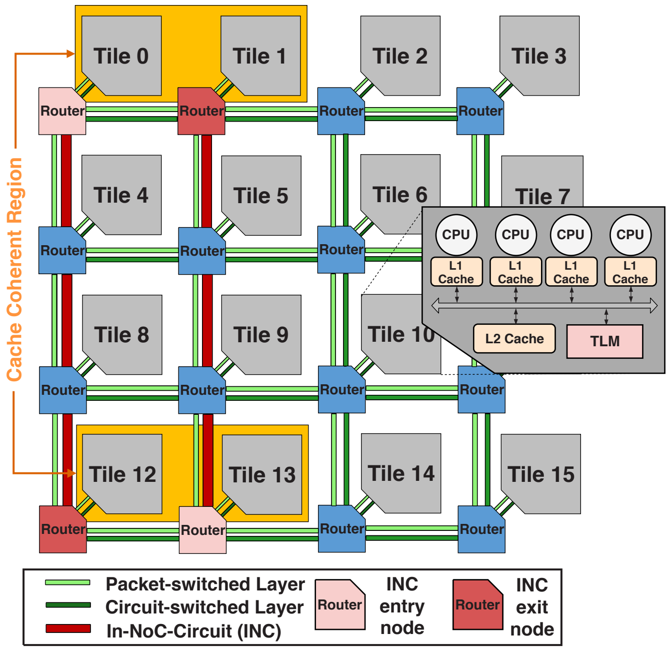


Figure 6.1: An example 4x4 architecture with a packet and circuit layer as well as two INCs that form a direct one-cycle connection between the entry node and its connected exit node.

up circuits that are not constantly replaced. Instead, they increase the utilization of existing circuits by making them sharable among communication streams and partners. To allow this sharing of circuits, they are established directly between routers in the NoC and not between tiles in the many-core architecture. An example 4x4 architecture including two INCs is shown in Figure 6.1.

The packet-switched layer is primarily used for forwarding regular data packets that do not need any form of QoS guarantees (i.e. best-effort traffic). The circuit-switched layer is used for asynchronous and bufferless end-to-end connections, setup by messages in the packet layer. The traditional end-to-end circuits give applications a means of communicating with fixed latency and throughput bounds. However, many applications on a general-purpose many-core do not need such strict guarantees and can be served by the packet-switched layer alone. In such a case there are unused resources in the circuit layer, which can be used by the INC in order to improve overall performance of the network. The INCs are circuits that start and end in a router, yet they can span multiple hops, skipping all nodes in between. An INC is not limited in length but its usefulness entirely depends on utilization and hop count reduction for the traffic transported by it. The fundamental part of an INC is the link between the PS and CS layer to inject flits coming from the packet-switched layer into the circuit-switched layer and eject the flits at the endpoint of the INC vice versa. The second aspect is the circuit establishment procedure which in turn consists of the traffic monitoring to find relevant communication patterns and the actual circuit establishment process. The final element is the extended routing unit, which needs to accommodate to the fact that packets may be forwarded faster either over the INC or the regular ports in all cardinal directions.

6.1.1 Extended Router Architecture

The extended router architecture is shown in Figure 6.2. It consists of a synchronous packet-switched layer and an independent circuit-switched layer. The router is a variant of the *i*-NoC router presented in subsection 2.2.3 that is developed as part of the InvasIC architecture introduced in section 2.1.3. Since the InvasIC architecture exists as a (reconfigurable) prototype, its components do not have a fixed design but come in several variations. Most notably, the NoC can be configured to solely use the packet-switched layer, enable the circuit-switched layer and use a reduced packet-switched layer only for circuit setup, or use a fully hybrid approach which utilizes a mixture of both. For the INCs, the last option is assumed. This is due to the fact that a fully hybrid architecture benefits the most from the proposed technique.

The packet-switched layer features wormhole switching, multiple virtual channels (VCs), round robin arbitration, dimension order routing and is implemented in a pipelined fashion allowing synthesis tools to achieve high frequency targets. The pipeline stages of the router consist of the following: Input Buffering, Routing, Reservation, VC Scheduling and Output Buffering. Using all five pipeline stages reduces the critical path in physical synthesis below 1ns in a TSMC 45nm worst-case setup, yet it increases latency when the design is operated in lower frequencies [55]. Thus, some stages are optional and may be merged or simply omitted, leaving a two stage pipeline at its min-

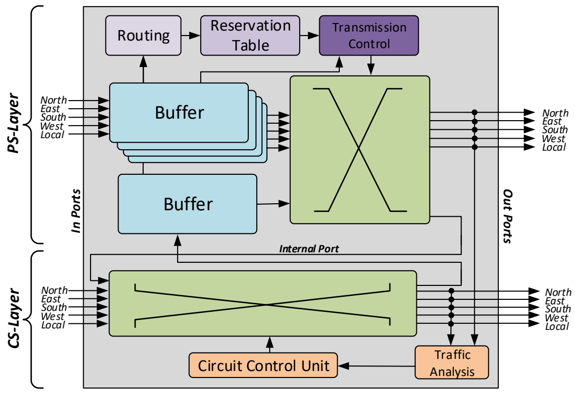


Figure 6.2: The router architecture including the INC extension. Not shown are the virtual channels and the configuration interface of the circuit control unit which can be accessed via regular packets or a separate, lightweight layer.

imum configuration since input buffering and the reservation step that assigns input virtual channels to output virtual channels are mandatory.

In the packet layer, a packet may arrive at a router through one of the input ports, i.e. in a meshed NoC from one of the cardinal directions and furthermore from the local input port of the connected tile. If a head flit is received, the routing unit performs the routing decision and forwards its results to the reservation table which handles all the free virtual channel allocations between input and output virtual channels. The transmission control then arbitrates the crossbar between all active requests in the reservation table for each output port. The circuit layer on the other hand is rather lightweight since it does not contain any buffers and does not require routing. Instead, it only contains the crossbar that is configured by setup packets transmitted over the packet-switched network. It typically operates on two channels per port, allowing full duplex operation but it can also be used for establishing two circuits in the same direction.

For realizing the INC, an additional internal port is added to both crossbars in the router. Additionally, an input buffer is added for this port on the PS side so a packet that is transmitted over an INC is buffered in the exit node. The reservation table now contains an additional entry for the PS crossbar to switch any input buffer of the router onto the port towards the CS layer. When a packet gets transmitted over an INC, it is buffered in the destination router and can take up slots in the regular reservation table of the packet layer for further transmission towards its destination tile. Instead of being forwarded in the packet layer, a packet may also take multiple INCs till its final destination is reached.

6.1.2 Traffic Monitoring and Analysis

The resources in the circuit layer are limited since any unnecessary wires, logic and buffer resources in an on-chip network should be avoided. As such, a limited amount of INC may be established depending on the available resources. The selection of the best entry and exit point of an INC plays a major role for the performance and efficiency of the approach. Consequently, a special hardware unit is proposed to automate the selection process. This unit can detect dynamic effects such as traffic hotspots and establish suitable INCs accordingly. To enable scalability, this process must be handled in a distributed manner.

In order to select the best INC at a current time frame, each router is extended by a monitoring unit which tracks the outgoing packets and tries to find communication patterns that could benefit from an INC. The monitoring unit evaluates the head flits of each outgoing packet, since these contain the coordinates of their destination. The trivial approach simply counts all packets going to a specific destination and will establish an INC to the destination that is most commonly targeted by the packets passing through the router. To avoid frequent circuit flipping, a threshold needs to be employed, which will enforce that a new INC is set up only if it holds significantly more promise than an existing INC. Furthermore, multiple INCs may contend for resources (i.e. links) in the circuit layer, so establishing an INC might fail if the priority of the new connection is lower than an existing connection that is using conflicting link resources anywhere on its path. To reduce complexity and help the scalability of the traffic monitoring, the meshed topology can be split into virtual sub-regions. An example is shown in Figure 6.3, where a 10x10 meshed NoC is split into 25 sub-regions of size 2x2. Choosing larger sub-regions will improve scalability, yet yields more imprecise results from the traffic analysis.

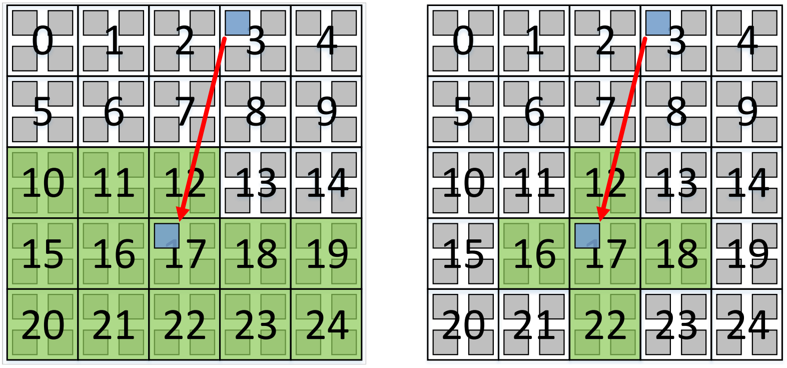


Figure 6.3: The sub-regions and the targets benefiting from an INC on the left and a more scalable implementation on the right.

The INC selection can be improved by utilizing a metric that takes into account all regions that would benefit from a certain INC. An example of such a region is shown in Figure 6.3 on the left. Any packet that arrives at the marked router in region 3 would benefit from a direct connection to region 17, if its final destination is in one of the marked green regions. Weights can be utilized to rate the impact of an INC on the path of a packet, giving a higher weight to an INC that connects a node very close to its final destination. Since such an evaluation is rather complex, a simplified version only checks the direct neighbors of a target node as shown in Figure 6.3 on the right.

6.1.3 Circuit Setup

Once the traffic analysis has found a destination for which the expected gain is above a certain threshold it will try to initiate the establishment process of an INC. First, a priority for the INC is calculated and afterwards a setup packet is created and transmitted towards its destination. For this task, either a lightweight packet-switched control layer or the regular packet-switched network can be used. The procedure for each router is shown in Figure 6.4. In each node along the path of the setup packet, the router will perform a lookup whether there are free link resources remaining or whether all resources are blocked by existing circuits. In the latter case, a new request may also trigger a

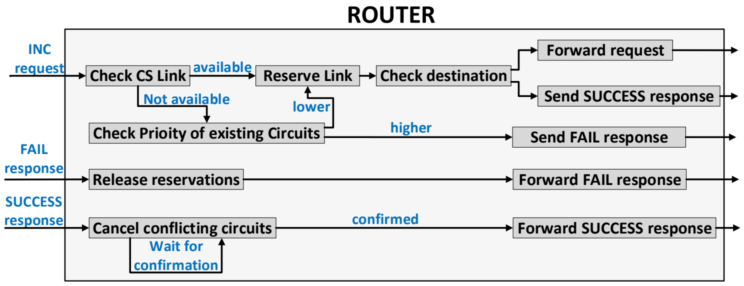


Figure 6.4: The process of establishing a circuit in a router.

cancellation of existing circuits that have significantly lower priority. Statically constructed circuits with the purpose of strict end-to-end QoS guarantees have the highest priority and may never be canceled other than by the creator (i.e. the application) itself. This is according to the goals of the INC: Improve utilization by making use of unused resources without interference with hard application requirements. Other INCs of lower priority may be canceled however, in which case a cancellation packet is generated and sent along the way of its circuit. To avoid unnecessary cancellation of existing circuits, a new circuit request will only trigger a reservation, the actual activation must wait until a response packet from the downstream router indicates a successful route to the destination or a blocked link along the path. The new circuit must also delay its establishment until each canceled circuit is fully deconstructed, ensuring that no packet loss may occur.

6.1.4 Routing

By adding the additional port connecting the packet and circuit layer, the routing unit has an additional degree of freedom for its routing decision. Thus, the dimension order routing (DOR) is extended by a closeness calculation to select the preferred output port. If the destination router of an INC is closer[2] to the destination tile of a packet compared to the router selected by DOR routing,

[2] Regarding the Manhattan distance

it will be chosen as output port. A further extension is able to arbitrate between the INC and the best output port according to DOR routing in case that the INC is blocking due to congestion, thus improving throughput. This new routing scheme is simple to implement, yet it also means the previously guaranteed order of packets is not necessarily enforced anymore. This situation occurs when either the routing utilizes the extra arbitration in the case of congestion or it can happen during setup of the INC. Transmissions that rely on the order of packets must thus enforce it by other means which can typically be handled by a suitable protocol in the network adapters.

6.1.5 Evaluation

In order to demonstrate the benefits of the INC concept, a MPSoC architecture with a relatively high tile count is required. This is limited by regular hardware prototyping capabilities. Therefore, SystemC simulations are used to evaluate the concept in the following.

A two-step simulation approach is used that consists of two different SystemC simulation frameworks viz. a high level system simulator and a cycle accurate network simulator. The system simulator models the scientific DSM architecture with region-based cache coherence support and the network simulator models the NoC with the INC extension. The system simulator contains a trace-based processor that is fed with memory traces obtained from gem5 [15] running in full system mode. The system simulator is modified to additionally generate specific network traces with coherence markers during the course of its execution. These network traces are then fed to the network simulator which simulates the effects of the INC.

A subset of workloads from the PARSEC [14] (blackscholes, canneal, fluidanimate, swaptions) and SPLASH-2 [129] (fft, lu, radix) benchmark suites is chosen for the evaluation. The chosen workloads are a mixture of entire applications and kernels that cover domains such as financial analysis, data mining, animation, integer sort and matrix operations.

Two sets of evaluations will be presented for the proposed technique and the motivating scenario. In the first set, the impact of different configuration options in the router is investigated. Since most of these options only affect the behavior of the INC in scenarios with high contention for INC resources,

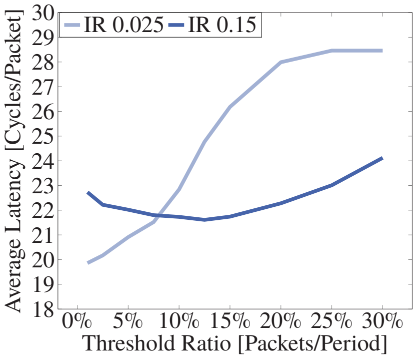


Figure 6.5: The ratio of packets in a time interval, that need to target the same area in order to trigger the generation of a circuit request.

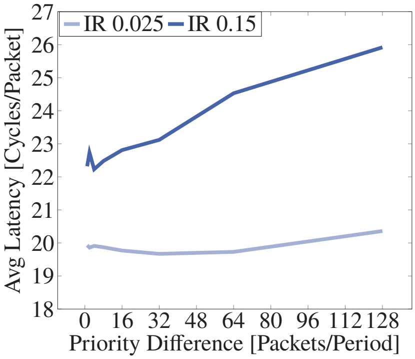


Figure 6.6: The priority difference necessary for a new circuit request to replace an existing connection.

random traffic can be applied for the traffic generation. The second set of evaluations uses the benchmarks running within cache-coherent regions in the meshed architecture.

6.1.6 INC Parameters

The setup for the first set of evaluations is as follows: 10x10 NoC, 3-stage pipeline routers and a 256 cycle period for the traffic analysis. Two major parameters that affect the performance of the INC are the threshold, which controls at what point a circuit request will be generated and the delta between two circuit weights (called the priority difference), that decides whether a new INC request will replace an existing circuit.

In Figure 6.5, the establishment of circuits based on different thresholds is highlighted. The threshold is given as a ratio of packets within a time frame of 256 cycles that have the same target region. A threshold of 10% for a certain destination is thus met if at least 26 cycles within an evaluation period were used for transmitting a packet towards that destination. In a low traffic scenario with Injection Rate (IR) 0.025 Flits/Cycle/Node, the best results can be achieved by a very low threshold since there is only little contention among circuit requests. The curve saturates at 20% and above since in the low traffic scenario such a threshold is never met, leading to an average delay that is identical to the delay in the regular packet-switched network without the INC. In a scenario with a higher load of IR 0.15 this is no longer the case and INC requests are triggered even at much higher thresholds. However, low ratios may also impact performance negatively since too many INC requests cause contention, reducing average delays.

In Figure 6.6 the impact of the minimal priority difference that a new circuit request needs to fulfill in order to replace an existing connection is shown. In a low load scenario there is barely any difference between settings of this parameter. In such a case, the choice of INC is not so important and there are few "bad" circuits, which are replaced at a later time. In the scenario with IR 0.15 it can be observed that a priority difference which is chosen too high will have a negative impact on average latency. Choosing a value that is too low however always bears the risk of "circuit-thrashing", i.e. constant construction and deconstruction of circuits, resulting in worse overall performance.

6.1.7 Benchmarks

When evaluating the INC concept based on real traffic as in the presented benchmarks, an essential part is the placement of the cache coherent region. Any number of tiles can be joined to form a coherent region, yet scalability limits regions from getting too large. Thus, a typical region can be assumed as consisting of 4 regular computing tiles and furthermore a special memory tile containing larger data sets that do not fit into local memories. The memory tile is located either in the center of the mesh, in order to allow short distance to all other tiles, or at one of its sides due to placement constraints that a physical chip would exhibit. An example placement is selected to showcase their impact, yet the INCs help in providing low latency to any placement, even completely arbitrary regions. The regular tiles of a region are placed as either of the following:

- **Reference:** All tiles of a region are placed right next to each other in a 2x2 area

- **Clustered:** Two tiles in the top left and two tiles in the bottom left (as highlighted in Figure 6.1)
- **Maximum spread:** The tiles are located in all 4 corners of the mesh, e.g. Tile 0,3,12,15 in Figure 6.1

Since INCs serve the purpose of reducing the latency of multi-hop messages while also allowing multiple data streams to profit from the same INC, they cannot give any latency improvement to packets that are sent from one tile to its direct neighbor. As such, a reference placement of a coherence region with minimal hop distances among the tiles will be used as the baseline. However, if regions get bigger or the placement is conflicting with other regions, if dynamic scheduling and mapping occurs, hardware resources like I/O or memory tiles are required, such a mapping is not possible anymore and the INCs can provide their benefits. It can be assumed that clustered and maximum spread benefit most from an INC, yet depending on overlapping traffic, other setups could provide even more latency gains when multiple coherent and concurrently active regions are evaluated.

In a first step, a single benchmark is analyzed in more detail to show the impact of the NoC size and potential background traffic on the performance of the INC. In Figure 6.7 the results based on different NoC sizes are shown. The traffic pattern is based on the "canneal" benchmark using the clustered coherence region mapping with the memory tile in the center. In this example, the delay increases linearly with the size of the mesh when the INC are not in use. This is due to the fact that the clusters and the memory tile are placed further apart in a larger mesh. In the variant with INCs enabled, only a slight increase between a 4x4 and 6x6 mesh and even less increase between 6x6 and 8x8 can be observed. This is close to the optimal usage of the INC, which will bridge the gap between the two clusters entirely, allowing the same access latencies between the tiles independent of their distance. In a 10x10 mesh, there is a noticeable increase in average latency. Investigation of this result shows that in the 10x10 mesh there is a contention for link resources between the messages going towards the memory tile and the messages between the two coherence clusters. This leads to a higher priority in the intermediate nodes, resulting in a shorter and thus sub-optimal INC. Nonetheless and aside from such variations, the average latency reduction is typically much higher in larger meshed architectures.

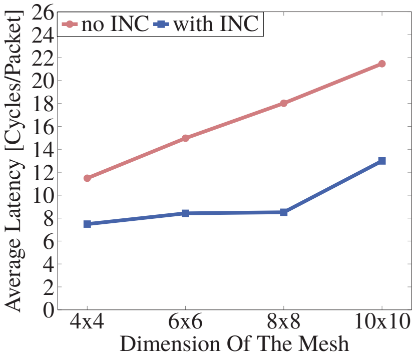


Figure 6.7: Latency gains of the INC depending on the size of the mesh.

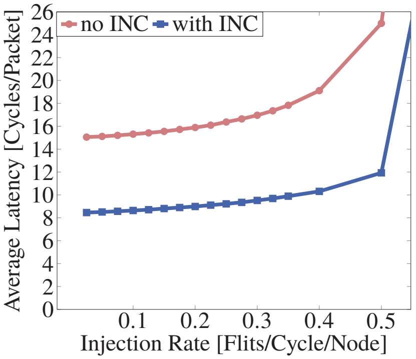


Figure 6.8: Impact of random traffic with varying injection rates on the performance of INCs.

In Figure 6.8 it is shown how the INC traffic performs under different load scenarios. As before, the traffic pattern is based on the "canneal" benchmark with a clustered coherence region while the size of the mesh is now set to 4x4. Additionally to the network traffic from the benchmark, all compute tiles now inject some extra load into the network in a random pattern. The added traffic is not allowed to compete for INCs but will congest the packet layer. Here, the latency of coherence messages slowly increases with higher injection rates at a the same rate for both, the variant with INCs and without. Since there is rarely contention and congestion in the network in low load scenarios, the result matches expectations. This changes however in high load scenarios, where the coherence traffic over the INC is staying stable much longer than the traffic without the INC. As a result, although overall performance of the network starts to break down, the latency sensitive traffic that is allowed to use the INC retains its low latency longer before collapsing.

In Figure 6.9 the full set of benchmarks is evaluated in a 4x4 mesh with a clustered coherence region. The values are normalized towards the reference placement, since the goal of the INC approach is to have similar delays as in the reference, but increase the freedom of choice in placing or adjusting the coherence regions. The results show that the INC can achieve a latency reduction between 3%-12% compared to the reference placement. This improvement is mostly due to synergy effects among the tiles within a cluster, which can use a

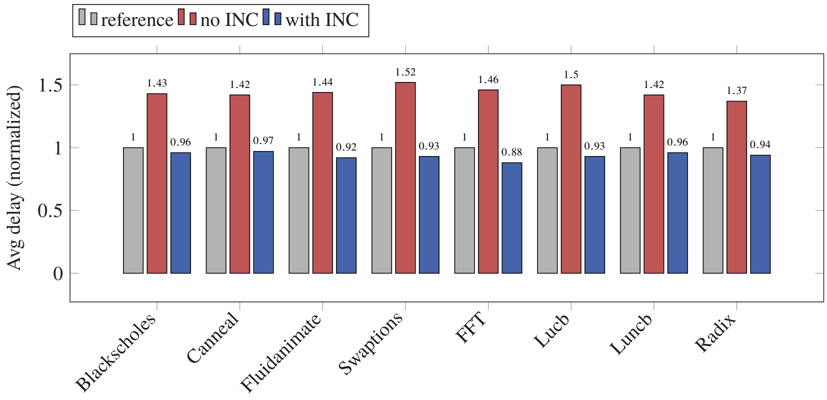


Figure 6.9: The normalized average delay for coherence messages with the following setup: 4x4 NoC, clustered coherence region.

shared INC towards the memory tile and shared INCs between the two clusters. The latency gains are even more significant (up to 40%) when compared to the clustered placement without the INCs, highlighting the new flexibilities for coherence regions.

In Figure 6.10 a similar evaluation is shown, yet this time the maximum spread placement of the coherence region is used. In this case, a performance degradation compared to the reference placement which stems from contention for limited INC resources can be seen. Specifically, it is not possible to have a direct connection between all tiles at the same time. Furthermore, there are less synergy effects of shared INC among the coherence region and the memory tile. Degradation can also happen when the traffic pattern frequently changes or when they bridge too small distances, reducing their impact. However without using INCs, the latency is reduced even more, almost doubling the delays compared to the reference. As such, the INCs can increase the latency by up to 45% in a maximum spread placement coherence region.

An observation about the actual runtime of the investigated benchmark showed that the INC did not provide a significant impact despite their large latency reductions. Analysis of this phenomenon hinted towards the issue that in the selected benchmarks, the largest part of the runtime was due to the processing in the tiles and not the network traffic. Even more, since for the evaluations, only

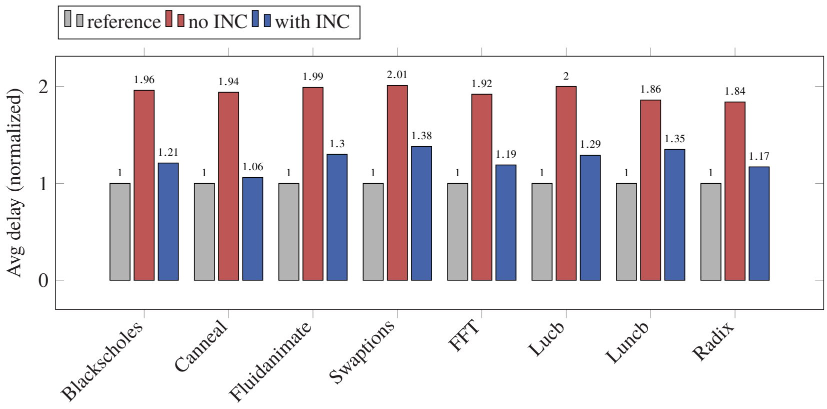


Figure 6.10: The normalized average delay for coherence messages with the following setup: 4x4 NoC, maximum spread coherence region.

coherence traffic was allowed to construct INC, all other traffic was banned, reducing possible performance gains. As a concluding insight based on these observations it can be stated that parameters like the size of a local memory, compiler and mapping optimizations, and architectural details play a major role for actual performance. Furthermore, the automated traffic analysis requires multiple data streams and dynamic effects due to e.g. a dynamic scheduler in order to be superior to static placements of the INCs. As such, for each architecture and each use-case it has to be evaluated separately whether the latency gains of the INC warrant the extra cost in area and power for the added logic.

As the evaluations were carried out in a non-synthesizable SystemC model, no hardware implementation details were available. In order to attain numbers on resource consumption for the INC concept and investigate the challenges during hardware implementation and system integration, a HDL description was developed. Hardware simulation tools were able to help in low-level verification of the basic functionality, but could not be used for testing and verifying large networks and real traffic scenarios due to their lacking scalability. Consequently, multi-FPGA prototyping was investigated for this purpose yet in the end, a hybrid prototyping approach was devised that provides the means necessary for the challenges at hand.

6.2 Multi-FPGA Prototyping

Using FPGAs for prototyping is sometimes also seen as an enhancement or extension to hardware simulation, labeled simulator acceleration. Field programmable gate arrays have a long history in hardware prototyping besides also being a target technology for hardware implementation and deployment in many different application domains [102]. FPGA prototypes allows clock frequencies of up to a few hundred MHz and provide fully parallel hardware execution, in contrast to sequential execution in any software simulator. This allows high performance and good scalability with growing design size. However, this scalability is limited by the available resources on an FPGA. In fact, a single resource such as LUT, FF or BRAM that exceeds its limit prevents the placement of the whole design. Additionally, the more resources are in use, the less likely it is for the tools to provide good routing to achieve high clock frequencies required for high performing designs.

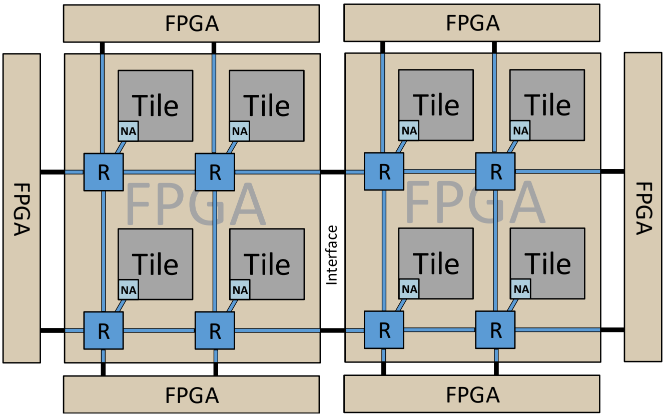


Figure 6.11: Many-core partitioning onto a multi-FPGA system.

Even though FPGAs are growing in size each generation, no single FPGA can currently host a full many-core design with hundreds of cores on its own. A common approach for prototyping a many-core architecture is thus based on a multi-FPGA prototyping platform. Such prototyping systems typically host a fixed number of FPGAs connected via high-throughput and low-latency interconnection cables or boards. Partitioning a many-core architecture onto multiple FPGAs as shown in Figure 6.11 needs to minimize cut-signals, i.e. the amount of signals that cross FPGA boundaries. As many-cores are based on networks-on-chip interconnects, the most promising cut uses the NoC-links between routers (R) to separate the design. For generating the least amount of delay, implementation on the link-level is recommended. However, alternatives exist that introduce extra logic to form a bridge and handle the cut on the transport protocol level [89]. Links which cross FPGA boundaries cannot be operated with the same delay as FPGA-internal links without slowing down the entire design significantly. These different delays result in restrictions for the design and potentially even different timing behavior. Furthermore, there is always an overhead when connecting pins for the FPGA interconnect and doing delay calibration among all signals on such a link. Another difficulty is the limited amount of pins on any FPGA, leading to an FPGA I/O bottleneck [120]. The pin count on recent FPGA is rather large and may be sufficient for most FPGA designs, a partitioned many-core however presents the amount of cut-signals on the link level as follows:

$$Pins = linkwidth * NoC\_links_{cut}$$

The $linkwidth$ typically only refers to the data width. However for counting the amount of interfacing pins, the control signals need to be considered in this case as well. A typical configuration of the *i*-NoC operates on 33bit data bit width, 4 bit ack signal, 1 bit request signal and 2 bit for virtual channels. The number of such links depends on NoC_links_{cut}, i.e. the amount of routers that are interfaced to a neighboring router on another FPGA and the amount of ports in those routers that are used to cross the boundaries.

The full amount of physical pins is typically not available for the inter-FPGA links since there are reserved pins of a multi-FPGA system and further pins required for extension boards. Thus, scaling a many-core architecture and partitioning it onto a multi-FPGA system might use all available pins before all internal logic resources are depleted. In order to overcome this limitation, pin-multiplexing can be utilized. A simple implementation connects multiple NoC-links that cross FPGA boundaries onto a set of pins with the size of a single link. Such an approach has the downside of skewing the clock cycles any transmission takes when crossing such a boundary. In order to avoid such skew, the clock of the FPGA design can be decoupled from the FPGA-FPGA interfacing. This allows to clock the FPGA-link at much higher clock rates than the design-under-test. In a best case scenario, all multiplexed signals can be handled and transported within a single clock cycle of the FPGA design. Measurements on the proFPGA prototyping system [4] show that the inter-FPGA links add 4 extra clock cycles for transmitting the data in a 50MHz design. Clocking the many-core prototype at higher rates would increase the extra clock cycles required for the multiplexing since the cycle overhead is kept low by operating the inter-FPGA connections at much higher clock rates that cannot be increased further. For attaining these numbers, clock counters were inserted and the request lines of the routers were connected to an ILA core. This allows precise delay measurements for packets sent to targets within one FPGA and between different FPGA. Modern FPGAs also provide additional means for fast I/O, as some works incorporate GTX transceivers in their design [40]. Yet there is always a level of delay that introduces inaccuracies in such multi-FPGA approaches.

6.3 FPGA-Host Interface

The previous section described the partitioning of a many-core architecture onto a multi-FPGA system and discussed the required FPGA-FPGA interconnects. A partitioned many-core architecture however also needs a way to interact with its environment. This includes:

- Initializing and loading of memories with program code and data
- Providing live input/output data
- Allowing debug access and enhancing visibility of the internal state of the architecture

The first two items are required in any phase of a many-core development, from an early prototype until deployment in a finished product. The last item is mostly required during prototyping phase itself but can prove useful for maintainability even in a deployed architecture. The second item is typically handled by a number of interfaces to peripherals or other information processing systems. There are numerous interfacing standards and specifications available for this task. FPGA boards as well as multi-FPGA systems provide IP blocks for integrating such I/O into a design. These either make use of components directly integrated onto the FPGA-board or utilize extension boards (in the latter case). The availability of live data can be an important aspect of a prototype for evaluating the performance or other non-functional properties of a many-core design in a real use-case scenario. However, the other two items are even more important: A many-core prototype can only work when its memories are correctly initialized. Furthermore, debugging capabilities are essential to the successful design of new architectures and architecture extensions. Some architectures may allow handling of debugging and program load via the regular I/O methods that are used for live data. This has to be supported by the I/O core or a low level system software that can handle the incoming data appropriately. A much more convenient method however creates a direct connection to a memory and/or processor. In tiled many-core architectures this is achieved by hooking a hardware block that contains a new bus master onto each local bus of all tiles. Such an approach is often called ”transactor” based access to the prototype. The transactor is interfaced either directly to a host system via a connector (such as USB, PCIe) or indirectly through a motherboard on a multi-FPGA system. In both cases, there needs to be a protocol and a low level software driver that allows simple read and write commands. These are transmitted to the transactor which guarantees valid bus handling when executing the commands. A transactor may occupy an address in the memory map and can thus even be configured to handle common I/O functions or primitives such as printf by redirecting the output stream to the host system. If multiple tiles are connected via a central interface component, multiplexing and arbitration of this resource is required.

Depending on the architecture, it is also possible to interface a NoC link instead of the local buses of each tile. Such an approach requires a fully established and connected NoC with a network interface that contains a bus master in each tile to provide full access to the memory or processors on a tile.

6.4 Multi-Level Hybrid Methodology

As central topic in this work, a novel multi-level hybrid methodology for prototyping of many-core and many-accelerator architectures is presented. The goal of a multi-level hybrid approach is to gain the benefits of several prototyping domains by combining or using them in an interleaved fashion.

Specifically, the goal is to combine tools, languages, simulations and FPGA-based prototyping for improving metrics towards successful design, debugging and evaluation of many-core and many-accelerator systems. This work was published in [MLB19] and [MLB20][3].

In a perfect world scenario, many-core prototypes run at full speed of virtual platforms while still providing the accuracy and debugging capabilities of hardware simulation and scalable execution speeds of FPGA-based prototyping. While it is impossible to build a single prototype that provides all of this for an entire hardware unit or system the size of a many-core, it is possible to create a prototype that implements a part of its prototyped architecture as a virtual platform, while another part is implemented on a fully accurate level. Historically, a similar challenge emerged several years ago where it was impossible to fit a single large processing core onto the available FPGA. As a solution back then, the functional model and the timing model were split into two parts [28]. This allowed fitting the timing model which required accuracy onto an FPGA, while the functional model could be executed on a host processor either natively or in a QEMU full system simulator. Special computing systems that included hardcore processors and a tightly integrated FPGA similar to the Xilinx ZYNQ eased this approach and enhanced their performance. Nowadays, fitting a single processing core onto an FPGA is not an issue anymore, making this split obsolete. This is beneficial since it avoids a number of added challenges and downsides that come along with this technique. However, combining FPGA prototyping with a prototyping environment on a different level of abstraction is still a promising approach. Since hardware simulation itself is extremely slow, virtual platforms are the most promising solution for combining with an FPGA-based prototype. The fusion of a software based virtual platform (VP) and a FPGA prototype is called multi-level hybrid prototyping. The application of a hybrid prototype is motivated by the observation that full level of detail is not always needed in every component when developing a many-core. The approach is similar to the concept of testbenches for small sub-components when designing hardware in cycle accurate simulators. In this case however, the goal is to execute real applications and prototype the full many-core architecture, yet not every component at the same level of detail. As such, the idea is to implement a part

[3] Extracts from [MLB19] and [MLB20], which were completely written by the author of the work in hand, are used verbatim in this section without further identification

	LUT	FF	Mux	BRAM
1 LEON3 Core	8797	2583	96	16
1 Tile (5 Cores)	61572	27949	733	132
1 Æthereal Router [50]	2658	N/A	N/A	N/A

Table 6.1: Resource cost comparison of example components present in a many-core architecture.

of the architecture on an FPGA, while modeling the remaining part in a virtual platform. This allows to have more FPGA resources for critical components under observation while creating a prototype that encompasses a much larger many-core architecture in total. As a recent development in the EDA industry, Synopsys released an extension to their multi-FPGA platforms that interfaces with their Virtualizer tool [5]. This development highlights the importance and potential of a hybrid approach. In contrast to the Synopsys tool, this work describes the concept and implementation as part of a VP-based design and verification methodology, independent of an expensive special multi-FPGA platform. Also, the presented approach provides a synchronization mechanism and is targeted towards many-core architectures since these specifically need novel approaches due to their design size. Many-cores show high suitability for a hybrid approach since design components such as the NoC are prime examples for a useful design split between VP and FPGA. The only other remaining solution that can handle prototypes the size of a many-core are specialized emulation platforms such as Palladium, Veloce or ZeBu. While they provide good debugging capabilities and scalability, they are very expensive and can realistically only reach speeds of ~1 MHz in recent designs [108]. Cadence also follows a hybrid approach as extension of their Palladium emulation platform. Since the platform is software based, it can easily integrate with a virtual platform and speed up execution of large software parts, such as booting an operating system. However, execution speed of the hardware components that are targeted for verification is still limited in the 1 MHz range. In contrast, the approach described in the following uses cheap and easily available FPGA boards and can allow designs with full FPGA speed of 100 MHz and more.

An analysis of the area consumption of a CPU that is available as synthesizeable HDL description and a multi-CPU tile in a typical configuration, as seen in the InvasIC architecture in section 2.1.3, is shown in Table 6.1. The core is a LEON3 processor from Gaisler Research including 32KB/64KB I/D L1 caches. A typical tile consists of 5 cores, local bus, network interface, Tile Local Memory (TLM) and debug link. As it can be seen, the LUT requirements are the limiting factor for the cores and tiles. Based on the resources available on the XC7V2000T (1,222M LUTs and 2,443M FFs), theoretically a full design consisting of 17 tiles and routers would fit onto such an FPGA. In comparison, there are resource utilization numbers of the Æthereal NoC available as provided in [50] which hint towards a theoretical number of 459 routers fitting on the XC7V2000T. These theoretical numbers can't be achieved in an actual synthesis due to placement constraints and other limitations, yet this estimation demonstrates the impact of mapping only a subset of an architecture to the FPGA: The full many-core (even without the routers) could host only a 4x4 design while focusing purely on the NoC, a 21x21 mesh is possible.

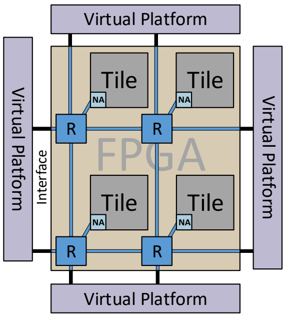


Figure 6.12: Scenario 1: A subsection of the Manycore is implemented in the FPGA.

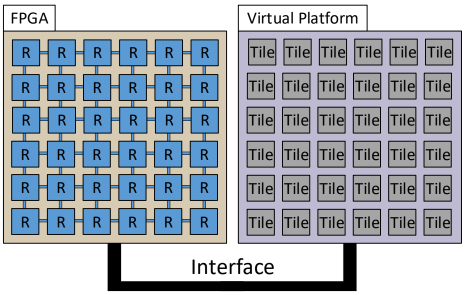


Figure 6.13: Scenario 2: The NoC is implemented physically, tiles reside in the virtual platform.

Based on these area observations, there are two major scenarios for many-core prototyping that motivate a hybrid approach. The first scenario implements a spatial fragment of the architecture (i.e. several tiles including their NoC routers) on the FPGA, while modeling the remaining tiles and routers in a virtual platform. This approach is shown in Figure 6.12. The connection between FPGA and virtual platform is located between two routers in the NoC, i.e. a physical router on the border of the FPGA that is connected to a neighboring router in the virtual platform. The interface between the physical router on the FPGA and the virtual router on the host PC will be described in detail later. In this setup, the focus is on the investigation of the tiles, which are prototyped with full FPGA accuracy but can send and receive their data to respectively from a much larger architecture. This allows the modeling of different load scenarios, access latencies over the NoC, I/O and large background memory on the host system.

The second scenario (Figure 6.13) is focused on the network-on-chip interconnect itself. In this scenario, a full NoC and all its physical links are implemented on the FPGA while the tiles are modeled in the virtual platform. The network interface can either be included in the FPGA or the VP part. In the former case, the connection between FPGA and VP is at the local port of each router at which point the data is transferred to the host and processed by a virtual network interface and the virtual tiles. Since the "cut" between FPGA and VP in this case is on regular NoC links and the local port/link typically behaves exactly the same as the links between two routers, both scenarios can be handled by the same physical interface implementation and only require different modeling on the VP side. Shifting the NI into the VP part requires a custom tile interface for this scenario but opens up additional prototyping and evaluation opportunities. In any case, this scenario allows the investigation of much larger NoC architectures under real application traffic, provides resources (e.g. memory, I/O) that might not be available on the FPGA itself and enables modeling of different (end to end) network protocols and paradigms in the virtual platform (e.g. shared memory versus message passing).

The hybrid approach is applicable independently of these two scenarios for any architecture and partitioning of virtual and FPGA part. The reduction in synthesis time benefits any design and verification effort. The approach can also be used in more traditional sense in the process of hardware/software codesign. Partitioning approaches may be used for optimizing the design into a software and a hardware part [10] [115]. However, in the context of design verification the partitioning is based on the accuracy needed for the relevant design component. Thus, the DUT would be mapped onto the FPGA unless resource constraints or performance bottlenecks force a different partitioning. For a many-core, the hybrid approach has the added benefit of allowing prototypes of a size not possible otherwise while retaining FPGA accuracy at least on a part of the architecture. Furthermore, the two presented

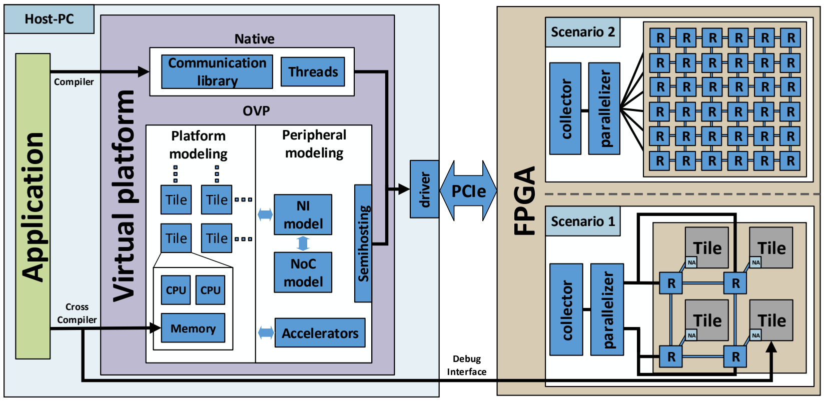


Figure 6.14: The full hybrid prototyping architecture.

scenarios share the same conceptual interface between FPGA and VP (i.e. the NoC links). Any other scenario would require a custom adaption of the interface to the DUT.

Figure 6.14 shows the full hybrid prototyping scheme. On the left a Host PC is shown, on which the virtual platform and the application code running within the platform reside. A virtual platform can either be abstracted on a functional level by the use of a communication library that handles the access to the FPGA or it is provided by a framework such as OVP. The virtual platform uses a direct access to a low-level driver for the physical interface to the FPGA (in this case PCIe). On the right the FPGA part is shown, split into three components: The fragment of the many-core architecture that is prototyped (i.e. the two scenarios), a NoC Interface which collects/distributes and prepares all the data from/to the NoC, and the host interface that includes the PCIe.

6.4.1 Interface Stack

The complete multi-level hybrid prototyping methodology consists of a VP part and an FPGA part. In order to join these very distinct prototyping worlds together, a number of interfaces need to be handled and suitable components need to be implemented. This interface stack consists of multiple parts that will be introduced and discussed in the following. From a bottom-up view there are the DUT interface (which corresponds to the NoC interface as motivated earlier), the host interface, the SW interface and the VP interface. Most of these interfaces are well defined, e.g. the NoC design defines the NoC interface while the virtual platform (in this case OVP) defines the VP interface. The remaining undefined interface is the host interface. The requirements for this physical interface between FPGA and host are low latency data transfers and high potential throughput so that large many-core architectures with high network load can be handled. Multiple FPGA/host interfaces are available on typical FPGA evaluation boards. PCIe is a preferable choice for this task since it offers the best performance in comparison to any other typically available interface between FPGA and host PC. The PCIe interface is clocked with 250MHz and offers 128 bit width for the data transfers. Data is transferred via DMA into a ringbuffer in the main memory on the host PC. The main memory hosts two separated ringbuffers, one for the receiving and one for the sending side.

The NoC interface as shown in Figure 6.15 is responsible for collecting all the data from the NoC towards the VP, and distributing the data from the VP to the NoC. For this task a Dummy Network Interface (DNI) is introduced that is able to handle the communication with the routers. All dummy NIs are connected to one of the parallelizers, which bundle the data streams of multiple NIs into 128bit packets that are used by the PCIe Interface. A configurable amount of dummy NIs can be connected to a parallelizer, yet with a typical NoC Link bandwidth of 32bit that are converted to 128bit, the default configuration contains four dummy NIs for each parallelizer. The parallelizer also adds a header to each packet which indicates the ID of the router and the virtual channel on which a packet arrived. The parallelizers are connected to a collector. This unit contains dual clock FIFOs which allow an arbitrary configuration of the clock frequency of the NoC, while keeping the PCIe interface at maximum clock speed for best performance. The collector is then connected to the Host Interface, that contains a Xilinx PCIe core.

When data is copied from the FPGA into the host memory via DMA, an interrupt is raised to notify the user-space software. A specific driver handles the low-level tasks for communication of user-space software with the FPGA design. The driver can reserve memory ranges in the host memory, initialize the PCIe device and setup DMA and interrupts. It creates a character device

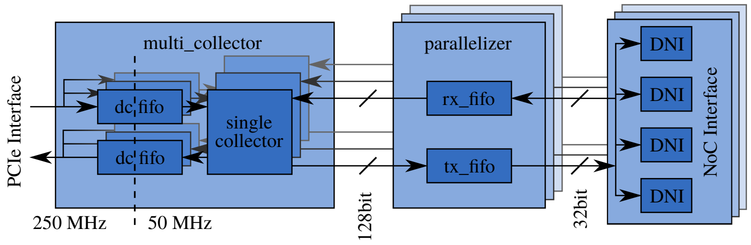


Figure 6.15: Components of the NoC Interface.

with the following syscalls: open, close, read, write, ioctl, poll. The driver makes sure that only valid data is read and supports both blocking as well as non-blocking calls. This interface allows the creation of user-space libraries and applications that directly work on the driver in order to send and receive data to, respectively from the FPGA.

In order to prototype a full many-core system, all the components that are not placed on the FPGA need to be modeled on the host. As an example, according to scenario 2 the NoC interconnect is placed on the FPGA while the network interface and the tiles are modeled in a virtual platform. In order to be fully binary compatible, an Instruction Set Simulator (ISS) is required. Furthermore, a modeling and platform generation framework for arbitrary components like the network interface, memories and local interconnects within a tile is needed. One existing framework that fulfills these goals is Open Virtual Platforms (OVP) that was used earlier in chapter 5 for software design and verification. OVP contains processor models for many different ISA, allows platform and system generation and the generation of peripherals that can be modeled to behave exactly like real hardware with fully compatible interfaces to the software. These peripherals also allow the use of a semihosting functionality, which gives access to host resources including drivers. Through semihosting we can access the low level SW-Interface and our PCIe driver, process the data within the modeled peripheral and make load/store accesses to data in the memories of the virtual platform.

6.4.2 Parallelized Host Execution

In a multi-level prototype, efficient execution relies on a balanced design that avoids bottlenecks in either domain including their common interface. On the FPGA, design speed is mostly limited by the DUT, which has to be considered a parameter that cannot be affected on a methodological level. Speeds of 50MHz or 100MHz will be assumed in the following as the complexity of full many-core FPGA designs often prevents achieving higher clock rates. However, due to the fully parallel nature of FPGA execution, these frequencies can be achieved independent of design size as long as it can be placed on the FPGA[4]. The interfacing introduced with the hybrid prototyping is separately analyzed afterwards since it is closely related to the issue of synchronization between the two domains. This leaves the host part containing the virtual platform as the remaining potential bottleneck. Improving performance can mostly be achieved via parallelization by using all available physical cores of the host system and allowing scalability on more powerful host machines that provide a large number of cores. Section 5.4 highlights steps and considerations towards a parallel OVP platform. Since peak performance can only be achieved for simulation of a single core, it must be divided by the number of virtual cores that are simulated. Furthermore, some additional processing power is required for the peripherals in the VP.

Yet this leaves one issue that arises in a hybrid multi-level prototype, namely the fact that hardware resources can only be held by a single entity at the same time. This means that accesses to the host-FPGA interface need to be arbitrated in some way. There are two possible approaches for solving this issue. Firstly, each process may individually request the resource, handle their operation, and release their control over the resource afterwards. Yet there are several severe problems that appear when following this approach. Most notably, fair or ordered arbitration is not guaranteed, possibly resulting in starvation or generally unfair execution with a high level of indeterminism. Furthermore, overhead for claiming and releasing resources is typically high, hindering performance and increasing latencies. A different approach is based on a client-server scheme. Here, the device driver runs as a server that owns

[4] This statement is not fully accurate since the placement tools may struggle under high resource utilization. In such cases, long routes for the wiring may impact achievable clock rates.

the resource and handles arbitration of the access to the physical resource. Every client (i.e. every process) initially connects to the server that assigns a handler. This handler can then be used to send requests to the server. Overhead in recent client/server implementations is low, making this approach the preferable solution that is implemented in the framework.

6.4.3 Timing Accuracy and Synchronization Mechanisms

Fully cycle accurate operation as in a HDL simulator is only provided on the FPGA part of the hybrid prototype. The host part including the virtual platform works on instruction accurate processor models and fixed delay peripherals. This means that the hybrid prototype is inherently not a fully timing accurate model of a real hardware-synthesized many-core, since it does not consider processor pipelines, memory access times and any other elements that are part of the VP. As motivated earlier, this is often perfectly fine when focusing on a single aspect of the whole architecture, using the hybrid prototype for functional testing, validation and debugging. However, there are situations in which a certain level of timing accuracy is desirable for the whole prototype, mostly considering design space exploration and performance evaluations. In these cases, synchronization of the two domains (i.e. FPGA and VP) of the hybrid prototype is of special importance. The term synchronization as it is used in this document is defined as follows:

Definition: *Synchronization is the coordination of events to operate a system in unison*

Synchronization sets the two domains in relation to each other and can be achieved by various means, depending on the envisioned goal. In the following, timing delays induced by the interfacing of the two different domains are investigated and approaches for bringing certain levels of timing accuracy into the hybrid prototype are described.

Four different modes of operation regarding synchronization of the two domains of the hybrid prototyping can be identified:

- Asynchronous

- Fully synchronous
- Pseudo synchronous
- Delta-based

The first mode of operation is the default setting for a hybrid prototype. In this most basic variant, no synchronization mechanism is employed at all. Since the motivation for such a mode of operation is based on purely functional verification as presented earlier, it will not be discussed here and this option is simply listed for completeness.

The fully-synchronous stepping refers to a mode of operation in which all elements in both domains progress one cycle and synchronize before the next cycle is processed. It means that there must be a common notion for the smallest interval of time that is comparable in both domains (such as a cycle) and the values on any signals or wires that cross the domains must be exchanged after each cycle. This mode has been investigated in many publications when coupling HDL simulators, typically used for testbenches, with FPGA prototypes that run the design under test [70]. Similarly to the existing techniques that use HDL simulators, fully-synchronous operation on a cycle basis can also be achieved for virtual platform execution. Depending on the level of detail and accuracy in the VP, there is still a common smallest time interval, i.e. a cycle. Instruction accurate models provided by OVP lose some accuracy since they assume that each cycle one instruction will be completed. This does not match with reality where pipelines, predictive execution and misses, bus contention, etc. exist. However, it is possible to bring accuracy to instruction accurate virtual platforms such as OVP at the cost of execution speed that is lowered by a factor of 150-170 [106]. This shows that accuracy can only be achieved by sacrificing performance, yet a VP that is slowed down by a factor of 150 is still far superior regarding simulation speed to any comparable technique at fully accurate levels. Extending these techniques for the synchronization of the two domains of a hybrid prototype generates a huge amount of overhead. For understanding the source of this overhead, a delay measurement of the hybrid prototype is presented in Table 6.2. FPGA to Host and Host to FPGA both encompass the parallelizer, collector and PCIe interface as described earlier. They represent the delay between a packet being sent to the local dummy NI via the NoC, till the processed PCIe packet is located in the ringbuffer in the host memory. The software delays encompass the interrupt and packet pro-

Domain	Mean	SD	Min	Max
FPGA to Host	0.52µs	0.04µs	0.39µs	1.49µs
Software	4.46µs	0.74µs	3µs	24µs
Host to FPGA	1.54µs	0.33µs	1.28µs	7.24µs
Sum	6.52µs	1.11µs	4.67µs	32.73µs

Table 6.2: Delay measurements for the domain crossing.

cessing on the host, including the access to the virtual platform. Not part of the measured time is any processing time within the virtual platform. This would represent a scenario that does not trigger additional operations, such as accessing a memory in the VP. The processing time will be considered in the calculations later on, yet it is not relevant for measuring the access times between VP and FPGA. The Linux RT (real time) patch on the latest kernel is used to avoid large delays due to the regular linux scheduler. The results show that the software is the major source of delay. However, the biggest issue is not the average delay but instead the fact that there are some outliers with a much larger delay than average. These outliers have been optimized and improved by the use of the RT patch, yet they are still four times larger than the mean delay. Using a bare-minimum operating system providing fully static resource allocation could improve this situation even more. Similarly, using polling instead of interrupts may also improve performance and reduce outliers at the cost of processing power. In order to provide in-time requests over the domain crossing in accordance with the presented findings, the VP and the FPGA design can be run at lower frequencies. This does not affect the performance of the host interface on the FPGA since it operates in its own clock domain.

Considering the presented delay measurements, the overhead for a synchronization on every cycle can be calculated as follows:

$$Overhead = \frac{T_{cyc,Hyb} + Max(T_{del}, H2F + T_{msg,H2F}, T_{del}, F2H + T_{msg,F2H})}{T_{cyc,Hyb}}$$

This formula consists of the average delay T_{del}, the time for transmitting the full synchronization message T_{msg} and the reference execution time for one cycle in the hybrid prototype $T_{cyc,Hyb}$. T_{del} and T_{msg} depend on the direction of the transfer, that is either from host to FPGA (H2F) or from FPGA to host (F2H). Since PCIe works fully duplex, both directions can be transmitted at the same time. However, as the amount of traffic may differ according to the traffic pattern, the maximum metric for the data transmissions of both directions is relevant for the overhead calculations. The values for T_{del} can be taken from Table 6.2 where the mean values for each direction can be seen.

T_{msg} is defined as follows:

$$T_{msg} = \frac{message\ size}{bandwidth}$$

It consists of the message size, i.e. the amount of data that needs to be transferred for synchronizing one cycle, divided by the available bandwidth. Message size depends on the protocol overhead, as well as the amount of NoC flits that have to be transmitted. It has an upper bound in the sense that in each cycle, at most one flit may be transmitted over each NoC link that crosses the boundary between VP and FPGA. It is thus based on the amount of routers in the design (in scenario 2), or the active links (in scenario 1).

The reference execution time $T_{cyc,Hyb}$ is defined as follows:

$$T_{cyc,Hyb} = Max(T_{cyc,vp}, T_{cyc,fpga})$$

It depends on the execution speeds in both domains. Since the slower speed will determine the overall performance, the formula chooses the maximum value. $T_{cyc,fpga}$ denotes the clock frequency of the design on the FPGA. It refers to the DUT clock and is independent of the interface clocking. $T_{cyc,vp}$ depends on the virtual platform size and the available physical processing resources on the host machine.

In light of the issues that fully-synchronous operation faces, alternatives are required that offer better performance while still providing some level of synchronization and accuracy. This is a difficult task, since there are a number of sources for delays and indeterminism throughout a multi-level prototype. It starts on the driver level that can only provide single threaded access to a hardware resource (i.e. DMA/PCIe). Handling this requires arbitration and results in sequentialisation even if the VP part is fully parallelized, creating a potential bottleneck. Next, the DMA unit and PCIe of the host system is also liable to indeterminism by reordering/grouping requests or due to contention for resources. Finally, packets arrive sequentially on the FPGA and need to be unpacked one after another as they come, even though the final injection into the NoC may happen fully in parallel. These sources of delay and indeterminism may not only result in situations where accurate operation is not possible, but instead also situations where no NoC contention happens at all. This situation may arise if the delays between subsequent traffic injections is too large, allowing a packet on the FPGA to be routed all the ways to its destination before another packets is injected. Such behavior is undesired since it will never trigger corner cases that appear under heavy loads in a NoC.

One alternative approach to cope with this situation is balancing the speeds of both domains and taking the sources of delays and indeterminism into account in order to allow similar execution patterns compared to the fully synchronous mode. This pseudo-synchronous operation does not use any direct synchronization, avoiding the issues that come from the interface delays completely. However at the same time it does not provide the same behavior as a fully cycle-accurate execution and is more intended to give statistically sound results instead of fully accurate ones.

There are two imbalances that this mode of operation tries to minimize:

- The FPGA running too fast, resulting in situations where a request towards the FPGA is handled before another request takes place, resulting in loss of resource contention scenarios that might happen in synchronized execution.
- The VP running too fast, resulting in request clogging in the FPGA which might not have happened in a synchronized execution.

Realization of this mode requires that the hybrid prototype is running at defined speeds in both domains, for example by scheduling the VP elements so that they are sequentially executed to match the FPGA clock speed. This can be achieved by introducing time slots and synchronization points at the end of each time slot. If virtual components advance too fast, they are stopped at a synchronization point and put to sleep. By selecting appropriate time slots, the accuracy/execution speed trade-off can be adjusted. Another issue that needs to be addressed is the delay that is introduced by crossing the domains between FPGA and VP. The shortest possible request that crosses the domain and triggers a response is the critical part in this regard. Taking the invasive many-core architecture as an example, the shortest request that crosses boundaries is a remote read operation. In a full hardware implementation it takes 9 cycles for input processing in the network interface, 5 cycles for bus and memory access of a single word, and another 9 cycles for output processing. Providing an accurate remote read thus requires that within 23 cycles on the FPGA, the dummy NI can transfer the request to the host and retrieve the data in time. Considering the delay measurements presented earlier in Table 6.2, one can calculate the maximum clock frequency on the FPGA so that the hybrid prototype would behave the same as a full FPGA prototype:

$$f = \frac{cycles_{max\_op}}{delay_{FPGA,Host} + delay_{Software} + delay_{Host,FPGA}}$$

Using the values for the individual mean and max delays from the table, a frequency of 3.5MHz for the mean case and 0.7MHz for the max delay case can be determined.

Other common operations in a network are remote writes or DMA transfers which are essentially clustered write operations. Since these typically do not require a response to complete, they are unaffected by the interface delays. Otherwise, similar calculations can be done for any other request-response operation.

The last mode retains a global synchronization that is valid in any concurrent task driven Model of Computation (MoC) like Kahn Process Networks (KPN) [49], synchronous dataflow [73], or the actor model [54]. These models have in common that communication is only handled via messages and no other influence on their local data is possible. Execution is based on producing and consuming of data and a fixed graph that defines the interrelations of the concurrent parts of an application. While this limits execution patterns, it also means that for each part of the code it is clearly defined when and on what data it operates. Without KPN or a similar MoC, an application may operate on some data that is remotely changed at any time. This creates many difficulties, even more so in a hybrid prototype. If execution has already passed the time when a remote operation would change some data and the update of said data was delayed due to the utilized prototyping method, old data is used for the operation, resulting in a divergence of results. Providing accuracy in such cases can only be handled by a fully synchronous prototype as described earlier.

If program execution and data exchange can be guaranteed to follow the above mentioned rules for KPN and similar MoC, synchronization can be achieved as described in the following. The general assumption is that the FPGA part of the prototype is inherently accurate while the VP part is inherently inaccurate. In fact, since the VP is inaccurate anyways, it does not need to keep track of its current time. Instead, the VP side is only required to calculate a delta, such that it can determine how many clock cycles it takes to produce some data. This procedure may either be periodic or depend on input data which triggers the execution. Since the FPGA part is fully accurate it is extended by a clock counter that signifies the elapsed clock cycles since the last reset. The bitwidth of the counter can be configured as 32bit or more (such as 64bit), depending on whether an absolute clock count is required or an overflow is acceptable. When a packet is injected into the NoC from the VP, it expects a timestamp which is used to delay a packet if it was processed too fast in the VP and would thus be injected too early. The packets are buffered in the dummy NI until the clock counter matches the timestamp of a packet. In case that the timestamp is already passed when the packet arrives, it can either be sent directly or an error flag is raised to indicate inaccurate behavior. Concerning the other direction when data from the NoC is collected by the dummy NI, the global clock counter is sampled and attached to the PCIe packet towards the host. When a consuming task is triggered by that packet, the VP starts processing and takes the timestamp as a baseline, adding its processing time to this timestamp. When processing in the VP triggers another event (i.e. message over the NoC), the new timestamp will be added to the data transfer towards the FPGA. While this mode does come with some limitations as to the model of computation and requires the FPGA part to be slowed down so that the VP is always able to run ahead to avoid a situation in which a request is sent to the FPGA that should have been injected at a global clock cycle earlier than the current clock cycle, it also provides a fast and accurate execution aside from the inaccuracies of the virtual platform itself, which are out of scope here.

6.4.4 Bandwidth Considerations

The final synchronization mode does assume that the bandwidth of the host-FPGA interface is never a limiting factor. If this can not be guaranteed, higher delays may occur that are difficult to quantize and locate. In the following, some bandwidth considerations are discussed.

Bandwidth becomes the limiting factor if the worst-case amount of traffic is generated (i.e. the maximum possible data) and the implementation cannot handle it. Consequently, an estimate of the maximum traffic that gets generated in a platform is required. It can be assumed that worst-case Injection Rate (IR) is at around 0.5 packets/cycle in every router at the same time. This assumption is based on the observation that independent of the application, NoC performance can not handle more than this IR and will start to break down at higher rates, causing large delays. In reality, the IR depends on the applications and benchmarks as well as their mapping and scheduling onto the tiles of a many-core. In case that an overall average IR higher than 0.5 is observed, it hints towards an unbalanced system that requires optimization so this value can be seen as a theoretical upper bound. Although an application might communicate in bursts that inject one packet in every cycle (i.e. the theoretical physical maximum), it will not happen in all routers at the same time or if it does, it can be considered a design flaw on scheduling or algorithmic level. Assuming a DUT/NoC clock rate of 50MHz, 32bit NoC link width and the aforementioned IR of 0.5, the throughput generation at the local port for a single router is 100MB/s.

In comparison, PCIe provides 8GB/s with 8 lanes in the common 3.0/3.1 standard used in many COTS host-PC and FPGA boards, but can go up to 64GB/s in its latest iterations. This means in theory, that it is possible to create a hybrid prototype containing 80 tiles and routers (for PCIe 3.0/3.1 with 8 lanes) or 640 tiles and routers (for PCIe 5.0 with 16 lanes) that is still not bandwidth limited. In reality however, the situation is a bit more complicated since the theoretical throughput numbers of PCIe can not be achieved on an actual FPGA board. This is mostly due to protocol overhead that is always present, even if there is no contention for PCIe resources. The magnitude of this overhead varies to a large degree, depending on many factors. A white paper by Xilinx discusses several scenarios and factors that impact PCIe performance [72]. The paper calculates two realistic throughput scenarios based on write transactions

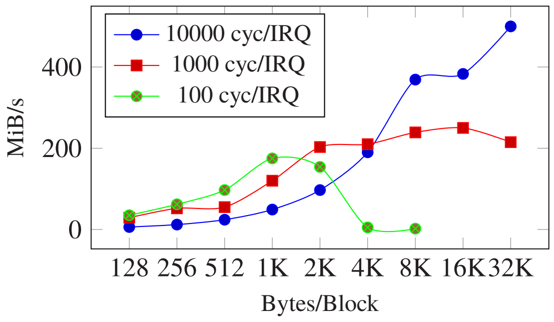


Figure 6.16: Throughput against block size and IRQ frequency as shown in [MLB20]

over PCIe 3.0 with 8 lanes, resulting in either 1.689GB/s or 1.912 GB/s. A third scenario is presented that looks at read transactions that only achieves 0.5GB/s due to an assumed delay in a memory controller. However, the hybrid prototype uses FIFO buffers that do not trigger such a delay. Evaluations on the hybrid prototyping framework using a VC707 board (which is limited to PCIe 2.0 8x), highlight that there is also a trade-off between latency and throughput in the shape of IRQ frequency. The results are shown in Fig. 6.16. High throughput can be achieved if interrupts are sent sparsely, since this allows filling of transfer buffers without constantly interrupting the operating system. Conversely, too many interrupts can impact VP performance negatively. The interrupt frequency should be chosen according to the intended usage pattern of the hybrid prototype. As such, workloads with many small transfers or low latency requirements should select a high interrupt frequency, while large bulk transfers for tasks such as video streaming benefit from a lower interrupt frequency. The system may even lock up due to interrupt storm. This may happen if block sizes are too large and interrupts happen too often, since the system cannot handle such situations anymore. This explains the missing results for 16K/32K block sizes at 100 cycles per IRQ.

Based on all these sources and observations, realistic PCIe throughput is thus assumed as at most 8x less than the theoretical throughput. This means that (depending on PCIe version), up to 64 tiles can be safely handled even in the worst case. Since an average IR of 0.5 over all nodes at the same time is often unrealistic, lower injection rates leave more room for additional routers that can be fed with data. Furthermore, the initial calculations assumed an FPGA clock rate of 50MHz. This rate needs to be adjusted according to the synchronization mechanism presented in 6.4.3, reducing the stress on the PCIe connection even more and allowing increased number of routers. The amount can be increased by a factor that is proportional to the factor the clock rate is reduced by (e.g. for a 1MHz clock there may be 50x more routers). While this results in reduced performance of the prototyping approach, it is still on a completely different level than hardware simulation while allowing large architectures that do not fit onto existing FPGA board solutions.

6.5 Hybrid Prototype for In-NoC-Circuit Design and Verification

The INC concept was initially prototyped and evaluated in a SystemC environment which allowed fast development of the functionality on a cycle-accurate level. However, due to limitations and simplifications in the SystemC design it is not synthesizeable directly into hardware. Thus, after the results seemed promising and the desired features were selected based on the SystemC prototype, a HDL implementation was devised. The HDL implementation used hardware simulation for the early testing and debugging. After stable operation on small testbenches was achieved, the design was included in a hybrid prototype of the InvasIC architecture. First, the hardware resource consumption in an FPGA design of an extended router are shown. The design was synthesized with Vivado version 2018.1. In Table 6.3 the first column shows the resource cost of the basic *i*-NoC router, consisting of a packet-switched layer and a circuit-switched layer. The next column shows the extra cost for implementing

	Basic Router	INC extension	Traffic Analysis
LUT	8008	1460	1042
FF	4988	796	237

Table 6.3: LUT and FF resource cost of the basic router and the extensions.

the basic INC functionality. In the final column, the additional cost for the dynamic traffic analysis is shown. This extra column for the traffic analysis is included since it is an optional feature and the INC could also be configured based on design-time knowledge at compile time or by an operating system component at runtime. The added resource cost of the INC extension including the PS/CS-Interface and the circuit control amount to about 18% LUT and 16% FF of the basic router architecture. The traffic analysis adds another 13% LUT and 5% FF of the basic router. Since all buffers are implemented in logic (LUTRAM), there are no extra cost in BRAM. The extra resource cost mostly comprise of unavoidable extra logic for the additional ports in the crossbar of both layers and the added buffer at the input port towards the packet layer. Other elements like the traffic analysis and the logic for keeping track of active circuits however contain elements that have to scale with larger mesh sizes and could be further optimized. The critical path is not impacted by the extensions since the INCs are implemented with an asynchronous data-path and the traffic analysis and setup mechanism are independent of the other router components.

With the proposed hybrid prototyping it is possible to bring the implementation onto a single FPGA and use it for verification, bug detection and feature optimization.

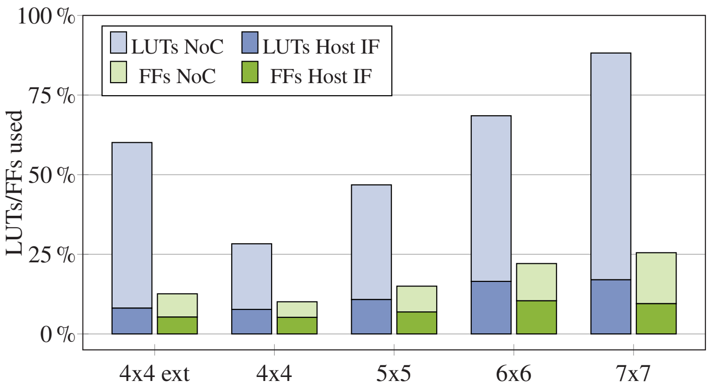


Figure 6.17: The resource consumption of the NoC (the design under test) and the host interface including the NoC interface.

In the example setup a Xilinx Virtex-7 XC7VX485T evaluation board is connected to a regular Intel PC with 8xPCIe 2.0 as a host system. The resource overhead that accompanies the hybrid prototyping can be seen in Figure 6.17. In a 4x4 NoC design 7.7% of the total available LUT and 5.2% of total FF on the FPGA were used by the host interface, compared to 20.6% of LUT and 4.9% of FF used for the NoC itself. The host interface thus adds an overhead of 37% LUT to the design under test in a 4x4 meshed NoC. Since the figure shows that LUT are the limiting factor, the overhead in FF can be ignored. A comparison with the more complex, multi-layer design of the extended NoC router (INC extension) in a 4x4 NoC shows that in more complex routers the overhead is even less significant. When looking at larger NoC sizes, the NoC IF and the PCIe part scale better than the NoC itself, resulting in an overhead of 24% in a 7x7 design.

Hybrid prototype	SystemC sim	HDL sim
179s	1d 2h	10d 8h

Table 6.4: Run time of the blackscholes trace on NoC prototypes from different prototyping domains, each representing a 4x4 NoC.

For determining the prototyping speeds, the runtime of a simple benchmark application is analyzed on the available implementations: SystemC, HDL simulator and FPGA. A trace of the network traffic is chosen that was generated from the blackscholes algorithm, mapped onto several tiles of a 4x4 architecture. A 4x4 architecture is used since the HDL simulator does not scale well and would take an unreasonable amount of time for a larger mesh. For a fair comparison the exact same input trace is used in all three variants. However, by including a suitable VP, the hybrid prototype can also natively execute the algorithm in contrast to the other methods. The following tools were used: SystemC version 2.3, ModelSim SE-64 10.0d for the HDL simulation and Vivado 2017.3 for FPGA synthesis. The results are shown in Table 6.4. The FPGA was able to execute the whole trace via the communication library and the PCIe interface in 179 seconds. The SystemC simulator took 93 678 seconds (1day 2hours) based on a cycle accurate model of the invasive NoC that contains some simplifications and is not synthesizeable. The Modelsim simulation took a total of 927 047 seconds (10days 8hours). The design was based on the same HDL description that was synthesized onto the FPGA, minus the PCIe interface and the collector logic but instead with a testbench that reads the traces of the blackscholes benchmark. The run time difference is enormous and highlights the benefits of hybrid prototyping, besides the ability to execute the same code as on a physical platform and the modeling advantages of the VP.

The gap will widen even more to a point where the pure simulations become completely unfeasible when the designs grow larger or include processors, memory, etc. in the HDL and SystemC models.

Prototyping platform	Tile count	Build duration
Multi FPGA	4	5h51m
	16	23h24m
Hybrid	4	20m
	16	42m
	49	1h19m

Table 6.5: Build duration for a commercial Multi-FPGA platform and the proposed hybrid prototype.

Compared to a HDL simulator, the hybrid prototype requires synthesis of the FPGA part and thus induces some extra time for re-synthesis when design changes are made. The measured times for a hybrid platform are shown in Table 6.5. To put these numbers in comparison, the build duration[5] required to re-synthesize an architecture containing the same amount of tiles for a large multi-FPGA prototyping platform is also shown. As it can be expected, these numbers are much larger since the multi-FPGA platform hosts a design containing the full tiles and thus much more logic needs to be synthesized. Depending on the mapping onto multiple FPGA, the build durations may be decreased by utilizing more parallelism (i.e. synthesizing all FPGAs in parallel), if enough computing power is available. However, this will improve the build duration even in the best case scenario (a perfectly weighted mapping with no extra overhead) at most by a factor that is equivalent to the amount of FPGA boards in such a system. Considering this, the hybrid prototype can be re-synthesized much more quickly and allows higher tile counts even on a smaller FPGA board (Virtex-7 XC7VX485T for the hybrid prototype versus the Virtex-7 XC7V2000T of the multi-FPGA platform).

[5] Build durations may vary depending on the machine on which it is executed, however for comparison it is sufficient that the same machine was used for all builds.

Using the hybrid prototype for developing the NoC extension may also help to detect bugs and design issues that would go unnoticed without. Any observed issues can be reproduced in small HDL testbenches after analyzing the behavior on the FPGA. However, some issues do not show without a hybrid approach at all. One such case was a wrongly written assert statement, that is ignored in HDL simulation but causes issues in the FPGA design. Since asserts are deleted by the tools in synthesis, some critical assignments can be incorrectly deleted due to a missing semicolon. The synthesis gives a warning about ignoring the assert but gives no hint that actually a large chunk of code is deleted. The HDL simulator on the contrary simply evaluates the assert as true and schedules the assignments correctly.

6.6 Summary

In this chapter, FPGA based hardware prototyping approaches for many-cores are investigated. Since traditional techniques are not sufficient, a novel multi-level hybrid methodology is introduced. The novel methodology is specifically motivated by a NoC extension that enables scalable low latency communication in many-core architectures spanning large amounts of tiles or nodes. This is achieved by introducing shortcuts in the network that utilize circuits within the network which can serve multiple data streams at the same time. The shortcuts work well in conjunction with coherence messages and remote accesses to large memories that can easily become a bottleneck. The design, debugging and verification of this NoC extension was enabled by a hybrid prototyping approach. This hybrid approach combines an FPGA part that contains a DUT that is prototyped with full hardware details and a virtual platform that allows modeling the remaining components of a many-core. This enables prototyping of a subset of tiles or a large NoC on the FPGA while providing a fully functional many-core environment. Combining FPGAs and VPs matches well since both allow high performance and good scalability despite being on different levels of abstraction in the design process. Going beyond purely functional prototypes, the challenges of synchronization between the prototyping levels are handled by appropriate schemes that either operate on a pseudo-synchronous level or even allow fully synchronized behavior as long as the model of execution is based on KPN or similar schemes. Bandwidth is determined not to be a limiting factor since the synchronization mechanisms require slowing down the FPGA part before the bandwidth limit is reached in a realistic application scenario.

7 High-level EDA supported Design

In this chapter, a novel approach for increasing the level of automation in many-core and many-accelerator design is introduced. Design processes typically follow a top-down approach that is based on an initial functional and non-functional design specification. From such a specification, a hardware and software implementation is derived in a codesign fashion. Two advancements for handling the complexity of such design processes are abstraction and automation. Abstraction is essential for enabling the top-down approach by introducing intermediate levels of design instead of directly targeting a hardware and software implementation at their lowest level. Automation on the other hand tries to skip the lower levels of design in a top-down approach entirely. Such approaches do not operate on the de-facto standard level for hardware design (i.e. RTL) but instead rely on latest advances in EDA tools towards abstract system modeling and descriptions with automated hardware generation. Consequently, they relate directly to the Electronic System Level (ESL) and the virtual platforms that were utilized in the previous chapters for software development and the hybrid prototyping for hardware design and verification.

At the heart of this chapter is one concept: High-Level Synthesis (HLS) as introduced in subsection 3.2.4. The major goal of HLS is an increase in design productivity. This metric can be defined based on the time a design process takes and the Quality of Result (QoR) of the produced design [96]. Although HLS is often seen as the next step in system design towards higher levels of design productivity, it is currently still rather limited in its features and QoR as supported by a comparative study about commercially available tools [82].

This chapter first presents a design, prototyping and verification methodology that is based on the techniques and approaches introduced in the previous chapters. Additionally, it highlights the missing steps for a fully holistic view on the topics targeted in this work. Specifically, this relates to investigations on the applicability and potential of HLS in the design and verification process of heterogeneous many-cores. As a major contribution, a framework is introduced that improves the hardware synthesis process of HLS by generating a SystemC intermediate representation from OpenCL input. An automated conversion towards approximate computing extends the presented framework, covering another important aspect of future heterogeneous architectures. The presented methodology has been published as part of [MLB20] and the framework was published in [SXMS$^+$16] and extended in [SXMX$^+$18][1].

7.1 A Virtual Platform Centered Design and Verification Methodology

As motivated in the analysis, a holistic view on the design, prototyping and verification of heterogeneous many-core has not yet been formulated. Earlier in this document, Figure 3.2 highlighted techniques used in typical computer architecture design processes. It included the specification level and the two major levels on which prototypes are built for verification, debugging and evaluation tasks. The two previous chapters introduced novel approaches for heterogeneous many-core prototyping on ESL level and on a more accurate RT-Level via a hybrid prototyping framework. In this chapter, these contributions are fused into a holistic design, prototyping and verification methodology. Towards this goal, the original figure is redesigned and extended by a number of improvements to this design process. Based on this figure, the missing links that have not been touched in this document yet are highlighted. These will be discussed in the following sections of this chapter, introducing further novel contributions to the state of the art.

The envisioned methodology can be seen in Fig. 7.1, spanning three levels of abstraction. In a first step, an initial specification needs to be devised that describes the functional, algorithmic level as well as non-functional constraints. DSE frameworks and partitioning algorithms are used to create an initial abstract hardware and software architecture. In recent years, specifically multi-objective optimization strategies have proven successful for this task [94]. Yet other, more specialized approaches such as scenario-based DSE,

[1] Extracts from [MLB20], [SXMS$^+$16] and [SXMX$^+$18] which were completely written by the author of the work in hand are used verbatim in this chapter without further identification

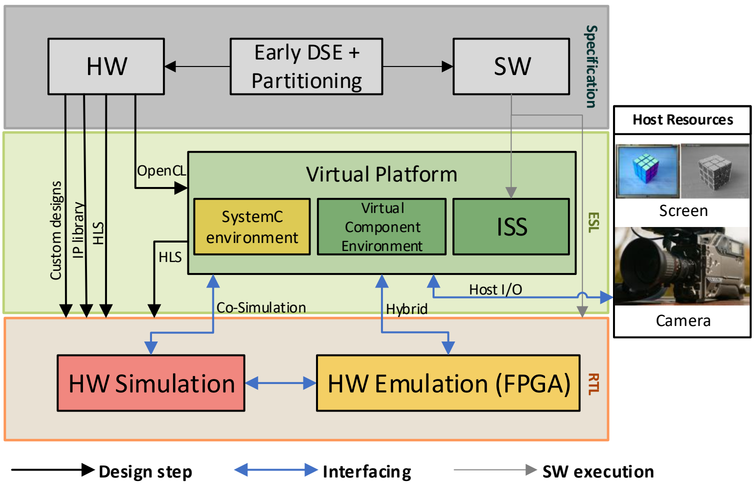


Figure 7.1: VP based design methodology using virtual platforms as central element.

show promising results [117]. HLS frameworks such as LegUP [25] may additionally help in identifying code that is suitable for creating a custom hardware accelerator. The proposed VP based design and verification flow defines the VP as the central element for further developments and optimizations. The VP is developed based on the hardware specification and serves as development platform for the software implementation. Additionally, it is used as a reference that the hardware implementation needs to adhere to. Since most components such as the GPP models are readily available in frameworks such as OVP, there is a direct mapping from specification to VP for common components. GPP extensions or entirely new custom ISA cores may be implemented by writing the corresponding morpher code that translates the custom instructions into host code. The architecture is built from IP cores that cover the regular processing elements as well as the interconnect and peripherals. The VP also allows direct access to host resources, enabling the use of I/O for real world data such as video or sensor input as presented earlier in section 5.3. Hardware integration and verification are envisioned as a multi-step approach via the hybrid prototyping approach as described in the previous chapter. Successively, parts of the architecture are selected for hardware integration and verification. This selection is handled manually and may produce different mappings at the same time for verification of individual parts, depending on the availability of RTL code. The HW components are prototyped on an FPGA board that is interfaced with the virtual platform on the host system. This enables a fully functional system view that provides accuracy and speed of FPGA based prototyping for the integrated components. The design loop is intended to operate on ES- and RT-Level, yet flaws and oversights in the initial specification may also be found and fed back so that future design iterations may converge more quickly. The hybrid prototyping approach provides a synchronization mechanism that can be deactivated in favor of faster yet purely functional verification and validation efforts. As system design progresses, the pseudo-synchronous mode or even better (if applicable) the delta-based synchronization can be utilized for performance estimations.

The described methodology builds upon the hybrid prototyping approach and the extended virtual platforms that were introduced in the previous chapters. However, the toolflow and design automation have not been discussed yet, even though these aspects need special attention as they can not be handled by the current state of the art approaches appropriately. Specifically, the handling of custom accelerators from specification to VP and the inclusion of a VP in the High-Level Synthesis process play a vital role for the proposed methodology. The first aspect is a required step in the proposed methodology that has to be handled manually as of now. The second aspect relates to the current HLS tools that do not make use of the VP centered design methodology. In order to cover both aspects, a design flow based on OpenCL is envisioned that improves on state-of-the-art HLS flows. A traditional OpenCL design flow operates on the original OpenCL code, invoking FPGA compilation tools that can only validate simple testbenches. In the next step, High-Level Synthesis is used for generating bitstreams that can be evaluated on FPGA boards. This flow leads to very long design cycles due to long synthesis times and restricted debugging and evaluation capabilities that FPGAs exhibit. When the synthesized design on the FPGA does not meet expectations or specified criteria, the original OpenCL description must be adapted and the design flow starts from the beginning. In contrast, the design flow when using a SystemC VP intermediate steps is as follows: A designer specifies and develops an initial OpenCL application or provides an existing legacy OpenCL application. The application code is then fed to the framework which starts by converting the OpenCL description into a virtual platform. Now all evaluations and further steps can be done on the generated virtual platform without any interaction with the original OpenCL description. The VP allows investigations on untimed or timed levels and later refinement of the design onto lower abstraction levels such as RTL, making use of the hybrid approach.

In the following, the general use of HLS for heterogeneous many-core is analyzed. Afterwards, a novel framework is introduced to provide the OpenCL based design flow described before.

7.2 HLS for Heterogeneous Many-cores

Despite the promise of increased productivity, less errors, shorter verification periods and overall ease in access to hardware design even to non-experts in the field, High-Level Synthesis also comes with several limitations and caveats. Even though the first HLS tools have been introduced more than a decade ago and by today all major EDA companies have included HLS solutions in their product portfolio, it is still mostly used for generating simple accelerators while adoption in other fields such as control blocks is nearly non-existent. There are a number of reasons why this is still the case, most notably the steps required to come from a functional, sequential description as in ANSI-C, down to a fully parallel and possibly even pipelined design require major algorithmic efforts. These efforts increase with complexity and size of the input functional description, resulting in decreased quality and increased time for compilation and synthesis. Another major issue comes from the fact, that even a small design change in the functional description results in a complete re-execution of all compilation, mapping and synthesis steps. Especially in early stages where a lot of changes may happen, this severely limits the benefits of HLS which specifically rely on the fast design cycles this methodology provides. A possible solution for this issue is incremental compilation and synthesis, as investigated in some scientific works [71]. Finally, generating intermediate representations such as HDL code or even low level representations are typically very difficult to read and understand for a human. If the tools work perfectly and generate optimal and verified designs this is not an issue, however if custom optimizations or low level debugging are required, it becomes a real problem.

Based on these observations, using HLS for the design of a full heterogeneous many-core architecture is not desirable. However, it may help in the design of some components of such a system. The regular processing units are typically better taken as IP-blocks such as the open source LEON3 from Gaisler or the RISC-V design. The complexity, pipeline intricacies and control flow heavy aspect of general purpose processors do not fit in with the capabilities of to-days HLS tools. If IP cores are not sufficient and a custom ISA processor is desired, special languages and tools such as LISA (Language for Instruction Set Architectures) exist [136]. Despite struggling with some of the difficulties mentioned above, accelerators are the main strength and target of HLS. Improving on these difficulties in the context of accelerator design, verification and integration will be discussed in a later section when an OpenCL framework is presented that introduces an intermediate step in the HLS process. Aside of the computational elements, memory and interconnect are the remaining major components in a heterogeneous many-core. While memory is handled by HLS tools often implicitly, the interconnect has not been investigated yet. HLS tools are able to automatically generate an interface to common interconnects such as AXI, yet it has no support for a network on chip. Still, NoCs exist in many variations and architectural layouts that could benefit from customized designs based on traffic patterns or depending on the interfaced components. Consequently, this topic is of interest in this work and will be analyzed in the following.

7.2.1 HLS for Networks-on-Chip Design

A network on chip forms the communication backbone of any many-core architecture. It consists of a number of routers and a network adapter or network interface that must exist in any tile of the architecture. The routers are interfaced among each other based on the chosen topology while a network interface connects a tile with a router. Much of the functionality is placed in the network interface since it decides the end-to-end protocol and can take over additional tasks such as DMA transfers or special queuing and triggering mechanisms. The routers on the other hand are often kept rather simple in order to save resources and reduce the impact on power. The basic tasks a router needs to provide include buffering, routing, crossbar interconnect, link arbitration and flow control as described earlier in section 2.2. There are a large number of additional features, extensions and techniques that improve on these basic functionalities. There may also be some special architectures that do not need to fulfill all of the above listed items. However, the list indicates that a basic router implementation is not control flow heavy and shows some potential for HLS supported development.

As an individual item, the routing unit is the most promising part of a router for a HLS based design flow. It is mostly independent of the other parts of a router, only requiring the packet as input and a port selection as its resulting output. Potentially there may be additional inputs such as buffer fill levels and performance counters but these are easy to specify and integrate even without the help of HLS. Consequently, routing algorithms are a prime example for an HLS enhanced flow in NoC design or for evaluating novel features. However, the C developer must make sure that the packet handling is correctly matching the protocol followed by the network interface which generates all packets. In a direct comparison, implementing a simple yet often used Dimension Order Routing[2] (DOR) algorithm shows that on C level especially the testbench generation and functional verification can be handled much faster. The generated result on RT level is logically the same. However, the netlist is ordered slightly different. Contrary to the expectation, this seems to have a minor impact on resource consumption. While a standalone build of both implementations takes the same amount of resources, the HLS generated variant takes slightly more when integrated into the full router architecture[3].

The remaining elements of a NoC listed earlier do not show much promise for HLS based design flow individually. This is mostly because they have very fixed structures with little room for variation aside of few design parameter which are easily handled by generics without the help of HLS. However, due to the large degree of parallelism and rather simple control flow in a routers execution, generating a simple router entirely in HLS is possible. In order to compare such a HLS based router implementation, the design goal was to reach functional compatibility with the iNoC router that was written in SystemVerilog as part of the InvasIC architecture. This allows a direct evaluation of the design

[2] Often also called x,y-routing since the algorithm strictly follows first x then y direction towards its target

[3] For this test the existing implementation of the iNoC was taken and the routing algorithm replaced with the one generated via HLS

efforts required and the challenges/difficulties involved. The implementation process shows the benefits but even more so the issues and difficulties that exist for designing a router on a functional level. Firstly, while HLS tools have been developed over the course of many years, giving them time to mature - they are still mostly intended for accelerators that implement a simple logic that is interfaced to and controlled by a well known hardware unit. Typically, the automatically generated AXI interface can easily be integrated with an existing processor system. This allows direct control of the accelerator via software. In contrast, interfacing two HLS generated units as in the case of two routers is not foreseen. In this case, there is no well defined master but the HLS generated design must interface with another HLS generated design. Since HLS takes care of the timings, it is not inherently suited for cycle-dependent operation with custom interfacing. Consequently, the interface must be either modeled on a cycle accurate level or it must be able to cope with asynchronous behavior since the amount of time (or cycles) the other component takes is not known in advance. This means none of the automated interface generators of the HLS tools can be used since these can not be controlled manually by a master component (such as the CPU). If a design without any manual cycle-level implementations on a purely functional level is desired, this leaves only manual handshaking protocol implementation as a viable option. Yet even aside of the challenges on the interface, the logic within a router is not easily translated into a resource efficient and fast design. Knowledge about hardware design and architectures is required to steer the tool into the right direction. This includes topics such as array handling, arbitrary data types and oversights when hardware is generated that handles unwanted or unneeded situations. Array handling in Vivado HLS will infer a BRAM instance that only allows dual ported access. This means it is not suited for buffering multiple independent data streams as they occur in a router. Data types will only be implemented efficiently if they are chosen as arbitrary types with a selected bit-width. While this generates directly into very efficient hardware, it increases design complexity as some operations are not possible on such types in C. Yet even considering such optimizations which reduce the latency and resource consumption by a factor of 10, the result is still far beyond an optimized RTL design. The resource consumption is higher by a factor of 2 while the amount of cycles required are at 10 compared to a 2-5 cycle SystemVerilog *i*-NoC router, depending on the active pipeline stages.

7.3 OpenCL Based Framework for Many-accelerator Architectures

The trend towards more and more abstraction and design automation will continue since complexity is rising ever more. Tools and abstraction allows designers to cut down development times and reduce verification efforts in a field where time-to-market is an imperative metric, especially when a design needs to be ”first time right”. The evolution towards more abstract levels of system design has been going on for many years. One of the more recent developments was the introduction of high-level synthesis that starts with a high-level software language like C as input and generates suitable hardware designs for either FPGA or ASIC targets. Tools like catapult or VivadoHLS promise to deliver such functionality, yet many issues and limitations still remain. One fundamental issue that cannot be easily overcome are the large synthesis times, especially when targeting large architectures. While this is fine for a final synthesis run of a verified design, it causes issues for the earlier steps of product development that rely heavily on prototyping and design space exploration. Thus, utilizing a purely FPGA based prototyping approach is bad for design space exploration and parameter optimization. Instead, virtual prototyping on multiple levels of abstraction can fill the gap between FPGA based prototyping and early design space exploration.

Previously it was discussed how an OVP based virtual platform can be extended for use in many-core prototyping and how a hybrid prototype with a VP + FPGA part is achieved. As a limitation, the hybrid prototype required manual architecture modeling and implementation on both VP and FPGA side. Now, a solution will be presented how an architecture can be described and prototyped based on a single source input. A desirable input would be an abstract language for functional description of a task. The C-language is a very suitable choice due to its widespread proliferation and powerful expressions. However, C-synthesis tools are still rather limited regarding support for the full language specification. Even more, without expert knowledge it is difficult to use C-synthesis to its full potential, resulting in bad quality of the generated results. Language extensions and standardization proves helpful in this situation. One such example is OpenCL, which targets specifically accelerator rich systems which are a prime example of systems where HLS can produce good results.

Yet challenges remain and call for novel approaches for design of accelerator rich and massively parallel heterogeneous computing.

In the following an approach will be described that takes an OpenCL program as input and automatically generates a virtual platform for design space evaluation before promising designs are finally synthesized for FPGA prototyping. This work was published in [SXMS+16] and extended in [SXMX+18]. It is motivated by a gap between OpenCL and virtual platforms that currently exists. Using OpenCL for a hardware design significantly limits the target designs due to the available tools. As of today, only GPP/GPU and GPP/FPGA (such as the Xilinx Zynq) targets are supported. However, the language would also be suitable for exploration of other targets, such as many-cores or pure FPGA. A further limitation is that FPGA targets require a slow synthesis and place & route step, which is detrimental to early design space exploration or architectural evaluations that trigger frequent design changes. These deficits can be overcome by utilizing an intermediate step in the shape of a virtual platform. Using OpenCL as input step instead of directly working on a virtual platform ensures compatibility and portability via the existing tools.

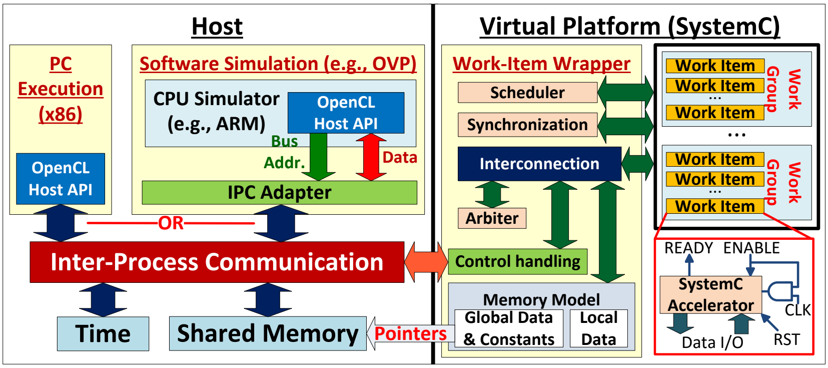


Figure 7.2: The OpenCL prototyping framework layout as introduced in [SXMX+18].

In Figure 7.2, the layout of the OpenCL based prototyping framework is shown. The figure contains all the elements of an OpenCL application and how they are represented in the framework. The framework consists of two parts: the host on the left and a virtual platform on the right. The VP is implemented in SystemC which gives full flexibility of the abstraction levels. Since SystemC is a C extension just the same as OpenCL, conversion schemes can reuse large parts of the source code.

On the host side, the sequential control logic of an OpenCL application is executed. The host can be represented by a COTS x86 PC or a software simulator that provides virtualization. The latter enables the benefits of virtual platforms as highlighted in earlier chapters while the former provides higher performance. Consequently in the context of prototyping novel architectures, the use of a virtual platform based on OVP is promoted in the framework. In either case, the OpenCL host API is used to access the shared memory and the kernels via the Inter-Process Communication (IPC). For the software simulator, a specific adapter must emulate the physical components that enables the IPC. The IPC mechanism itself is adapted by the framework to interface with the SystemC virtual platform instead of a GPU or FPGA. Data exchange is handled via direct pointers in a shared memory that can be accessed from host side and from VP side. Furthermore, the control handling is realized by forwarding the API calls to a work-item wrapper in the VP. This work-item wrapper is the core element of the framework on the VP side, as it provides a scheduler, a synchronization unit and interconnect arbitration to the memory and the IPC. Besides the wrapper, the actual work-items that contain the kernel code (i.e. the computational part) of the OpenCL application are instantiated in the virtual platform. These kernels are modeled in SystemC and represent simple accelerator units the are controlled via enable and ready signals while operating on a predefined data I/O interface.

Creating a Virtual Platform in SystemC from an input OpenCL application requires several steps. These steps include a syntax check, the code conversion, the construction of the Virtual Platform and finally the regular compilation of the SystemC code. The first and last step use regular available tools for syntax checking (clang) and compilation (gcc), linking the SystemC library to the generated VP. The other two steps do require some extra effort. Although both OpenCL and SystemC are based on ANSI-C, some of the features and expressions of OpenCL are not supported in SystemC C++. This make a conversion of such expressions necessary, most notably the following.

- **Vector processing:** OpenCL uses vector expressions that need to be converted to a custom vector class in C++

- **I/O transactions:** In SystemC, access to globals, constants, and locals must be handled via input/output method calls to memories. Any OpenCL access through array expressions must be converted.

After the conversion is completed according to the necessary steps above, the kernel code is placed in a work-item template. These are used to build the VP by including the wrapper and putting all components together. Special compilation flags can configure the process and outcome (specifically the wrapper) further. Finally, the outcome of the previous steps is a SystemC compliant source code that can be compiled with a regular gcc compiler into an executable SystemC simulation.

The design flow and the framework described so far are intended to reduce overall design cycles of the high-level synthesis methodology. Although not all benefits can be quantized, the time that a full iteration of the design process takes can be measured accurately. For this purpose, benchmarks from the Rodinia suite [101] of OpenCL applications is used as input. The OpenCL code is fed to both, the regular available toolflow from major FPGA vendors and the SystemC design flow.

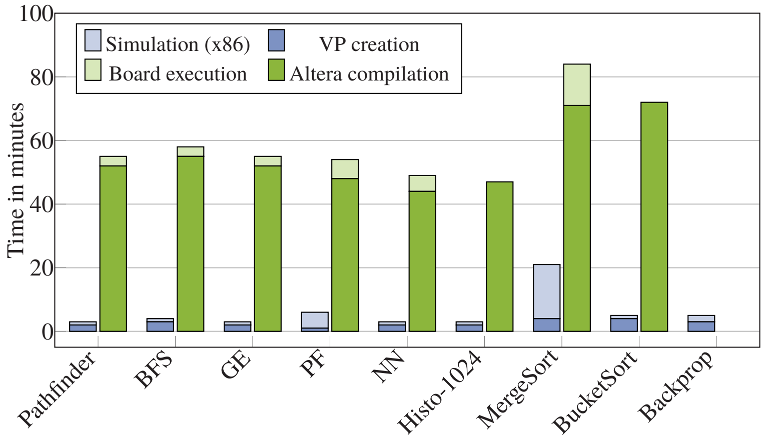


Figure 7.3: The OpenCL to SystemC conversion framework compared to an Altera OpenCL flow, as presented in [SXMX+18].

In Figure 7.3 the results are shown. The Altera flow consists of a compilation step that takes up most of the time. Depending on the input, this step takes between 44 and 72 minutes. The execution on the evaluation board takes comparably little time, with a maximum of 13 minutes. The results show, that there is no direct correlation between compilation time and run time on the board. In comparison, the conversion flow takes time for creating the VP in SystemC and executing the generated simulator afterwards on the host PC. The total time for both steps is around 3 to 4 minutes for most benchmarks, with only MergeSort peaking to a total of 21 minutes.

7.4 Automated Conversion for Approximate Accelerators

Computer architectures will become increasingly heterogeneous in order to provide efficient computations that alleviate the dark silicon phenomenon. Approximate computing is another approach following the same goal by providing more efficient computing when some level of errors or inaccuracies are allowed. Approximate computing can be realized either on the algorithmic, the physical or the data representation level. In the context of heterogeneous accelerators, the representation of data in memory or during processing is a very promising field. The OpenCL framework presented in the previous section was extended to support this form of heterogeneity by allowing the selection of different real number representations at design time that trigger the automated insertion of optimized conversion modules and utilization of corresponding arithmetic libraries. In the following, the reasoning behind this feature is motivated, the implementation details are discussed and evaluation results are shown. This work was published in [SXMX+18][4].

[4] Extracts from [SXMX+18], which were completely written by the author of the work in hand, are used verbatim in this section without further identification

7.4.1 Real Number Representation

An important factor when developing a new computing architecture is its design size, especially for FPGAs, since resources are often limited and designs may need to shrink. In addition, using less resources leads to cheaper designs and can improve metrics such as power and performance. The OpenCL framework presented earlier benefits greatly by providing simple means of changing different parameters that affect the design size. One such parameter concerns data accuracy, especially regarding computations that use real numbers. A regular CPU contains a special Floating-Point Unit (FPU), so algorithms are free to use floats without much concern. However, a full-fledged FPU consumes a very large amount of resources and thus the designer may need to think twice about whether floating-point support is really needed for an accelerator or a new processing architecture. For this reason, a different concept may be followed for application specific hardware accelerators. In particular, one possible solution that was initially promoted by the GPU industry is to use a half-precision 16-bit floating-point representation, according to the IEEE 754-2008 standard. Compared to 16-bit integers, this representation has the benefit of a large dynamic range that is used mostly in computer graphics while using fewer resources than the single-precision float. Another solution is the use of a fixed-point representation, often seen in DSPs. Fixed-point arithmetic is typically cheaper to implement, but at the expense of accuracy, since it does not offer the same dynamic range as floats. The choice between the above real number representations depends on the required algorithmic accuracy, the dataset that is processed and the resource utilization constraints. Although there are general rules of thumb, e.g., a predictable dataset with a small dynamic range can benefit from fixed-point without much loss of accuracy, it is important to accurately examine in early stages whether a design meets the data accuracy requirements. Also, resource utilization highly depends on the arithmetic operation: Figure 7.4 depicts a utilization comparison among float, half-float and fixed-point representations for basic arithmetic operations. According to this comparison, fixed-point representation allows very efficient additions/subtractions while exhibiting huge costs for division, as very complex division circuits are required, contrarily to 16-bit or 32-bit floats where A/B is equivalent to A * 1/B, thus only a multiplier is required as 1/B requires a simple masking on the exponent. Thus, the simulation-based accuracy val-

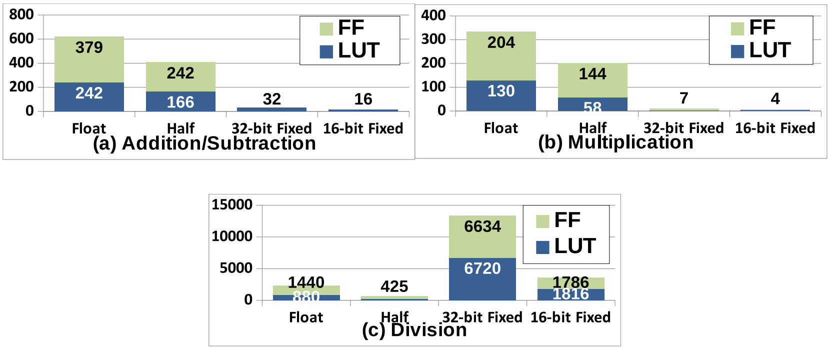


Figure 7.4: Resource usage of basic arithmetic operations, when using different real number representations.

idation and the utilization analysis are very important stages for a framework that enables early evaluation of different parameters and realization options.

7.4.2 Realization in the Framework

To achieve the most efficient trade-off between resource utilization and data precision, different floating-point and fixed-point real-number representations can be utilized. They are applied on 32-bit float variables and input/output of the original OpenCL source. The selection among representations is done at compile time and does not require design modifications. To reach a decision on an efficient representation, they can be iteratively configured with different representations. This allows the evaluation of accuracy and thus enable the selection of the most efficient representation. The approach can be combined with methodologies in literature that explore optimal representations. Such a work is presented in [78]: The authors present a framework that takes as input a floating-point expression and searches for a solution that fulfills a set of numerical precision constraints while minimizing the resource usage. In [33], the authors focus on fixed-point arithmetic, building abstract representations of an algorithm and modeling noise propagation and quantization decisions. Once a representation is selected at compile time, the conversion from/to the alternative representation is committed at runtime on the VP side: During a

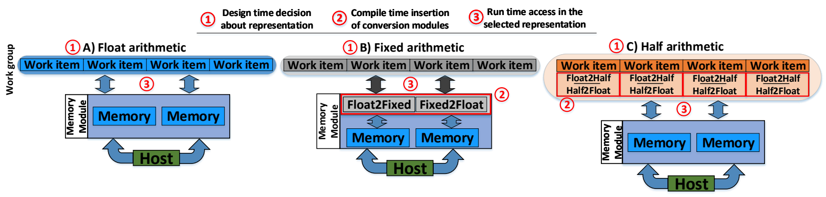


Figure 7.5: The supported real-number arithmetic including the conversion scheme.

memory fetch, a 32-bit float is converted into the alternative representation, while during a memory store the variable is converted back to a 32-bit float. The benefit of this approach is the unified data representation in the memory, for both the host and the OpenCL kernels, thus the host does not need to know which representation is utilized by each kernel. Also, this enables the use of different representations for each kernel. The fixed-point representations use the ”sc_fixed” data type, which is part of the SystemC library, while the supported floating-point ones are the native 32-bit and a ”half” 16-bit representation. For ”half,” the proposed framework utilizes an external library (a.k.a. hls_half), acquired from Xilinx Vivado HLS, which provides arithmetic and logical operations among half-floats, as well as complex mathematical functions. Typical HLS tools support the generation of optimized hardware based on these fixed-point and floating-point libraries. In the following, customized conversion functions between the 32-bit floating-point type and the target types (i.e., half-float and fixed-point) are introduced. The custom conversion functions are fully synthesizable by any HLS tool and are optimized in terms of resource utilization and performance. For hls_half, the custom conversion functions replace the original conversion methods of the library. Hence, the runtime conversion infrastructure imposes minimum utilization and performance overhead in a typical many-accelerator system. To reduce even more the utilization overhead, the extra logic for the conversion to/from fixed-point is not included into the work-items, but instead it is inserted in the memory controller: In a typical many-accelerator system there are much more work-items than memory modules, thus the conversion logic will be replicated less times in the design, while retaining full performance. However, the conversion to/from half-floats is implemented inside the work-items, as hls_half by default implicitly converts the input and/or output of the arithmetic operations to/from half-point representation. Nevertheless, it is noticed that the conversion to/from half-float does not occupy a very large amount of resources, thus the overhead is low. Figure 7.5 depicts how the different representations and the required conversions are handled in the proposed framework. The color scheme denotes the representation for data storage and arithmetic operations: Blue matches to 32-bit float, grey to fixed-point, and orange to 16-bit half. The figure also highlights the steps for the real number conversion: First, at design time, a decision is taken about the selected real number representation for each kernel. This decision is expressed as a set of compilation flags in the framework, to instantiate at compile time the respective arithmetic operations and insert the required conversion modules either in the memory modules (case B) or in the work-items (case C). When the work-items are executed at runtime, every memory access is converted automatically by the conversion modules. More in detail, in case of 32-bit float operations (case A), the memory accesses do not contain any conversion. For fixed-point arithmetic (case B), during a memory fetch the 32-bit float is first converted to the fixed-point representation inside the memory module and then is sent to the work-item. In the same way, during a memory store, the value to be stored is sent to the memory module, which converts the fixed-point number to a 32-bit float and writes it to the memory. For half-floats (case C), a memory fetch acquires a 32-bit float from the memory to the work-item, which afterwards converts this value to a half-float. In the reverse direction, before a memory store, the work-item converts the half-float value to a 32-bit float and then sends it to the memory module. The optimized conversions between full and half-precision float are based on the formula presented in [135]. The general layout of both float and half is the same, consisting of a sign bit, the exponent and the mantissa. However, the exponent in half-precision float uses less bits (5 bits for exponent and 10 bits for mantissa) and thus exhibits a different bias (i.e., offset value to put the exponent into an unsigned range). In particular, the bias is equal to 127 for 32-bit floats and 15 for half-floats. Thus, the bias of the source representation is subtracted and the bias of the target one is added. Also, exponent and mantissa are appropriately masked and shifted to be placed in their correct position. The fixed-point representation is entirely different from floating-point and thus requires several transformations. Fixed-point numbers resemble the integer representation, which also explains why some operations like additions can be realized with a very efficient implementation. Fixed-point numbers consist of a fractional and an integer part (i.e., the parts on the right and the left of the radix point, respectively) and can be configured to use any combination of sizes regarding both parts. The conversion to fixed-point is efficiently achieved by eliminating the exponent and shifting the mantissa according to (a) the difference in the radix point position between the float (i.e., bit 23) and the fixed-point representation, as well as (b) the exponent value. In addition, extra logic is utilized for the two-complement representation, which is necessary for mathematical operations.

7.4.3 Evaluation

In this section, the supported real number representations of the OpenCL framework are evaluated and their impact discussed based on (1) the benefit of using the optimized conversions between float, half-float and fixed-point compared to the available state-of-the-art conversion implementations, and (2) the impact of different real number representations in a selection of benchmarks that utilize floating-point operations. To analyze the efficiency of the optimized conversions, each conversion is synthesized as stand-alone function and the results are compared with the existing conversion mechanisms provided by the state-of-the-art libraries hls_half of Xilinx and sc_fixed of SystemC. The synthesis results including the resource usage and the minimum possible clock periods are shown in Table 7.2. Each comparison is reported for the supported real-number representations, grouped by the direction of the conversion, i.e., conversion to and from 32-bit float on the left and on the right, respectively. For fixed-point representation, it is noticed that changing the position of the radix point has no impact on utilization or performance. The results show a large improvement in required resources and expected timings of the proposed library in most cases, especially regarding the conversions from float to the target representation. In particular, the improvements in flip-flops range from 40% up to 100%, while in LUTs, except for some cases with slight overhead, the maximum gain reaches up to 70%. Also, the timing improvements range from 0% up to 54%. By analyzing the generated HDL files for both state of the art and the proposed approach, it is noticed that the conversion mechanisms of the state-of-the-art libraries utilize multi-functional IP cores with additional interface and status signals, which do not allow for further optimizations in timing and resource utilization. Thus, the proposed implementation leads to less conversion overhead. Moreover, the fixed-point conversion in Xilinx Vivado HLS instantiates a 64-bit float IP core. On the contrary, the proposed conversion libraries utilize simple shift and add operations of 32-bit width.

Benchmarks where all kernels use only integer operations

Algorithm	Application Domain	Kernel Name[a]	Work-Items Invoked	Work-Items Available	Workgroup size	Input
Pathfinder	Graph/tree traversal		256	64	64	2×128 matrix
BFS	Graph traversal	Bfs1	1024	128	1	1024-node graph
		Bfs2	1024	128	1	

Benchmarks where all kernels use floating-point operations

Algorithm	Application Domain	Kernel Name[a]	Work-Items Invoked	Work-Items Available	Workgroup size	Input
K-Means	Data Mining		100	100	1	100 elements
Gaussian Elimination	Linear Equations	Fan1	64	32	1	16×16 matrix
		Fan2	64×64	32×1	1	
Particle Filter	Signal Processing		256	64	1	256 particles[b]
Nearest Neighbor	Pattern Recognition		42816	64	1	42764 records
Back-Propagation	Machine Learning	Layer-forward	16×64	16	16	64-input ANN[c]
		Adjust-weights	16×64	16	16	
Hotspot	Thermal Simulation		32×32	32×32	16×16	2×16 elements[d]
Speckle Reducing Anisotropic Diffusion (SRAD)[e]	Image Processing	Extract, Prepare, Reduce, SRAD1, SRAD2, Compress	All kernels 512	All kernels 512	All kernels 256	Medical image sample (20×18 pixels)
Histogram-1024[f g]	Data management		384	48	48	1000 elements

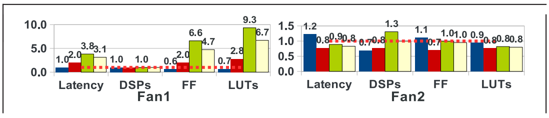
Benchmarks with kernels using floating-point operations or only integer operations

Algorithm	Application Domain	Kernel name[a]	Float Operations	Work-Items Invoked	Work-Items Available	Workgroup size	Input
MergeSort[f]	Stored data management (e.g. Databases) – Algorithms from Hybrid-Sort application	Merge-first	Yes	3072	32	32	1000 elements
		Merge-pass	Yes	3120	26	1	
		Merge-pack	No	3072×1024	32×1	32×1	
BucketSort[f]		Bucket-count	Yes	32	32	32	
		Bucket-prefix	No	1024	64	64	
		Bucket-sort	No	32	32	32	

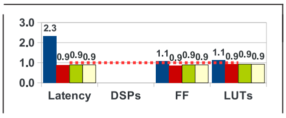
[a] Necessary only when using more than one kernels.
[b] In 10-frame 32×32 video
[c] Neural network with 64 inputs, 1 hidden layer with 16 neurons and 1 output
[d] 16 temperature and 16 power consumption values.
[e] The kernels of SRAD use mathematical functions which only support 32-bit floats.
[f] The algorithms are parts of Hybridsort, however they are considered as autonomous benchmarks.
[g] Histogram-1024 belongs to the second group as it does not have mixed kernels.

Table 7.1: OpenCL benchmarks for the experimentation setup, taken from Rodinia suite.

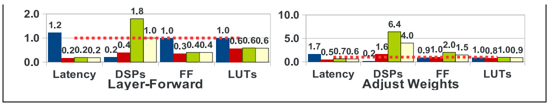
🟦 Half 🟥 Fixed <8,8> 🟩 Fixed <16,16> 🟨 Fixed <24,8> ····· Float



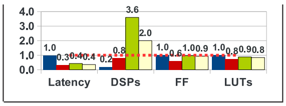
(a) Gaussian Elimination



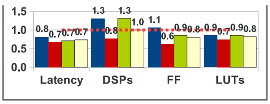
(b) Particle Filter



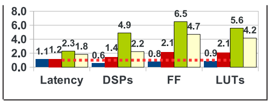
(c) Back-Propagation



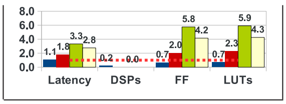
(d) Nearest Neighbor



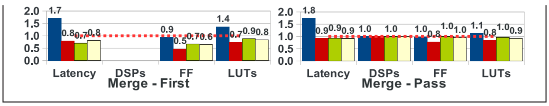
(e) Kmeans



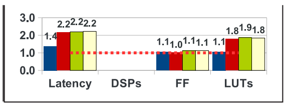
(f) Hotspot



(g) Histogram-1024



(h) Merge-Sort[a]



(i) Bucketsort[b]

[a] Merge-pack kernel does not include floating-point operations.

[b] Bucket-prefix and Bucket-sort kernels do not include floating-point operations.

Figure 7.6: Evaluation of the total latency (estimated by HLS) and the resource utilization for different real number representations. The results are normalized, using as reference the 32-bit float. Fixed-point representation scenario $< i, f >$ uses i and f bits for the integer and the fractional part respectively. In this analysis, Pathfinder and BFS are excluded, as they have only integers, while SRAD is excluded as the utilized mathematical functions support only 32-bit floats.

		Target: `float`			Source: `float`		
		FF	**LUT**	**Clock Period**	**FF**	**LUT**	**Clock Period**
Half	`hls_half` lib	55	9	1.79 ns	92	25	1.89 ns
	Optimized	0	31	1.60 ns	0	24	0.84 ns
32-bit fixed	`sc_fixed`	489	439	3.88 ns	1130	756	2.52 ns
	Optimized	283	563	2.02 ns	201	223	2.52 ns
16-bit fixed	`sc_fixed`	489	434	3.88 ns	989	643	2.52 ns
	Optimized	180	322	2.55 ns	175	190	2.52 ns

Table 7.2: Comparison between the conversions provided by `hls_half` and SystemC `sc_fixed` and the proposed conversion libraries.

In Figure 7.6, the impact of different real number representations on resource utilization and total latency is explored (as estimated by HLS). This analysis excludes Pathfinder, BFS and the kernels ”Merge-pack,” ”Bucket-sort,” and ”Bucket-prefix,” as they do not comprise floating-point operations, according to Table 7.1. Also, SRAD is excluded, because it utilizes mathematical functions that support only 32-bit floats. All numbers are normalized and compared to the baseline 32-bit float. A first observation is that there is a great variation in resource usage among the benchmarks: In some of them, fixed-point representation gives a huge improvement in resource usage and timings while others may even exhibit much worse results. This underlines the necessity of evaluating multiple real-number representations at design-time, to explore the optimal realization for each individual use-case, as provided by the proposed framework. Having a closer look at the benchmarks that have the worst results when using fixed-point representations (i.e., Fan1 kernel of Gaussian Elimination and Hotspot), it can be noticed that those results were due to the division operations of those kernels: As described earlier in this article, division for fixed-point is very resource-consuming and as such, any kernel that uses this operation will accrue a large overhead. However, the half-precision float, as compared to the regular 32-bit one, may induce some timing overhead: In both representations, the operations are performed in the same way, simply using fewer bits in case of half, which results in less resource usage by a fixed factor. However, due to inefficient implementation of non-basic operations (e.g., comparisons) provided by the initial version of the employed hls_half library, conversions may be required from float to half and vice versa, using the proposed conversion library that induces some timing overhead. In

	K-means	Gauss. Elimin.	Part. Filter	Nearest Neighbor	Back-Prop.	Hot-spot	Hist. 1024	Bucket Sort[a]	Merge Sort[a]
Half	0.13	1.07	0.59	0.19	0.0	4.51	0.33	0.50	0.001
Fixed<8,8>	0.27	88.68	0.65	69.06	0.0	255.80	9.99	0.53	0.001
Fixed<16,16>	0.22	101.13	0.59	39.77	0.0	0.20	0.11	0.54	0.000
Fixed<24,8>	0.27	99.90	0.65	612.86	0.0	0.20	9.99	0.53	0.001

[a] Bucket-sort and Merge-sort incorporate mixed kernels with either floating-point or only integers. However they have been included in this analysis in order to evaluate Hybridsort application.

Table 7.3: Mean absolute error for different alternative representations. Pathfinder and BFS are excluded, as they have only integers, while SRAD is excluded as the utilized mathematical functions support only 32-bit floats.

conclusion, the overall gains depend on whether the more efficient operations can outweigh the additional overhead added by the conversion library. Employing a more timing-/resource-efficient real-number representation typically also leads to a loss of accuracy. Table 7.3 shows the mean absolute error of the evaluated benchmarks in total (i.e., not separately for each kernel) for the different representations. Each representation is applied for all the kernels of the benchmark. To calculate the mean absolute error, the absolute output difference between the alternative representation and the 32-bit float is taken, for each output sample. Those differences are summed up and an average is taken. Half-precision offers a much wider dynamic range representing very small and very large numbers, as compared to fixed-point, which typically leads to less deviation from the 32-bit float, as mainly seen in Gaussian elimination and Nearest Neighbor. However, in some benchmarks, which do not require real-number accuracy, fixed-point can provide better results (e.g., Histogram-1024), while using different bits for the fractional and integer parts can also have a significant impact on accuracy.

7.5 Summary

In this chapter, a novel design methodology was discussed that builds on the virtual platforms and the hybrid prototyping presented in the previous chapters. To provide a holistic approach, design automation via High-level

Synthesis was envisioned as essential for said methodology, yet the current state of the art was deemed lacking. Thus, a framework was presented that introduces an intermediate step in the design flow for generating hardware as promoted by Xilinx and Altera/Intel tools. The framework converts the input OpenCL code into a SystemC representation that is used for all following design steps. This reduces design cycles since the costly synthesis step is only executed once at the end of the design flow. It also avoids some of the issues HLS still has: Any design change in the source code must run through the whole toolflow, may result in an entirely different design and produce a computer generated and thus mostly unreadable low level description. All these issues are alleviated by the intermediate step, benefiting from the flexibility of SystemC. The framework is further extended by a feature that introduces approximate computing in an OpenCL design flow. It allows the selection of a desired representation of real numbers which triggers the insertion of an optimized conversion library, allowing evaluation of performance, area and loss of accuracy. The aforementioned issues of HLS also limit their use mostly to the design of simple accelerators. Still, in this chapter the use for many-core design was discussed. Specifically in the area of networks on chip, their use for the design of networks on chip was evaluated.

8 Conclusion

The computing world is continually moving forward in search of more performance and computational efficiency[1]. In the past, this was achieved mostly by technology scaling towards ever smaller feature sizes of semiconductor structures. However, this development has slowed down significantly in recent times due to the power wall and will come to a stop almost completely due to physical and monetary limitations. In order to still generate growth and performance improvements, two major trends have emerged that are expected to become even more prevalent in the future. According to these trends, computing architectures will become more and more parallel and heterogeneous in nature. Consequently, there will be less generic and generalized architectures but instead, architectural features, compositions and layouts will be designed and optimized according to the end products requirements and goals. This increases the design space and overall design complexity significantly, requiring major improvements and progress in the area of design tools, languages, verification and validation.

The presented work investigates heterogeneous architectures and many-core computing with a special focus on prototyping approaches for these novel forms of computing architectures. Heterogeneity raises the complexity of software and runtime systems which require prototypes that mirror the interfaces and behavior while being early available and providing peak execution speeds. Loss of accuracy or even completely functional verification is acceptable for such tasks. In hardware design, interfacing and optimization of heterogeneous architectures are challenging tasks. Many-cores further increase those challenges in being defined by their sheer design size and added complexity due to optimized interconnect networks and memory hierarchies required for achieving high performing systems.

[1] Considered here as performance per watts

In this context, the presented work investigates novel approaches for both software and hardware in regards to design and verification. The novel contributions are centered around the concept of Virtual Platforms (VP). VPs offer a promising solution for early available and fast prototypes and, as described in this work, can be extended for prototyping heterogeneous many-core architectures. Most notably, these platforms can incorporate accelerator modeling, provide access to real world I/O from sensors/actors attached to the host machine and retain their high speed thanks to parallel process or thread based execution.

Hardware simulation will still play a role in hardware design and verification of heterogeneous many-core, yet the design size makes simulation extremely slow up to a point where it can only be used for simple test cases and small scale debugging. FPGA emulators are still going strong, yet they also face the challenge of design sizes that comes alongside of many-core architectures. Some solutions exist, namely multi-FPGA prototyping and FPGA-based design-virtualization techniques, yet they all exhibit weaknesses and undesired behavior. This situation is highlighted and emphasized by a core contribution in this work: a novel networks-on-chip extension that enables scalable low latency interconnects. This In-NoC-Circuits called extension introduces circuits that start and end within routers, allowing multiple data streams to share a circuit over a low latency secondary network. This feature is designed for interfacing a large number of nodes. Since existing approaches do not suffice, a novel methodology is introduced that develops the concept of a multi-level hybrid prototype. In this methodology, a prototype is built of components on multiple levels of abstraction. As most promising levels, virtual platforms on a host PC together with FPGA-based designs are introduced and investigated. The approach enables prototyping of large many-core architecture including heterogeneous elements such as accelerators at full FPGA speed levels. While the approach does introduce some levels of inaccuracy, these can be reduced via synchronization techniques at the cost of performance. Still, it is possible to achieve prototypes that are orders of magnitude faster than hardware simulations.

On top of the introduced prototyping techniques and extensions for hardware and software design and verification, the trend towards more design abstraction is unopposed and will continue in the future. Consequently, this work provides a contribution in presenting an enhanced OpenCL toolflow that is improved by an automated SystemC virtual platform generation and evaluation framework. It shows the benefits of fast design evaluation and directly interacts with high-level synthesis, a methodology that is expected to see large growth and widespread acceptance in the future. In fact, it can be claimed that as HLS is adopted more and more, the current state-of-the-art RTL design will become like assembly programming today: still required but done only by a small group of experts [119]. A further extension of the HLS framework allows evaluation of approximate accelerators by automatically inserting optimized conversion modules in the generated hardware design.

All presented novel contributions are part of a holistic design methodology for heterogeneous many-core. This methodology is centered around the concept of virtual platforms and makes use of the OpenCL based HLS framework and the hybrid prototyping approach.

Looking ahead, there is still much room for further improvements in the design and verification of heterogeneous many-cores. Yet even beyond that, the same topics covered in this work are also relevant in other computing areas. Most notably, the trend in recent years towards machine learning and optimized architectures in this field, labeled neuromorphic computing, introduces a similar set of challenges. These focus even more on the memory aspect, yet hardware/software tradeoffs, design size and complexity as well as improvements in design automation will play a major role in their advance as well.

List of Figures

List of Tables

Acronyms

AHB Advanced High-Performance Bus

AI Accelerator Interface

API Application Programming Interface

ASIC Application-specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

AXI Advanced eXtensible Interface Bus

BRAM Block Random Access Memory

COTS Commercial Off-The-Shelf

CPS Cyber-Physical System

CPU Central Processing Unit

CS Circuit Switching

DAC Design Automation Conference

DDR Double Data Rate

DMA Direct Memory Access

DOR Dimension Order Routing

DSE Design Space Exploration

DSM Distributed Shared Memory

DSP Digital Signal Processor

DUT	Design Under Test
DVI	Digital Visual Interface
EDA	Electronic Design Automation
eFPGA	embedded Field Programmable Gate Array
ESL	Electronic System Level
FF	FlipFlop
FIFO	First In - First Out
flit	flow control digit
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
HSC	Hardware Software Codesign
I/O	Input/Output
IC	Integrated Circuit
IDRS	International Roadmap for Devices and Systems
ILA	Integrated Logic Analyzer
INC	In-NoC-Circuits

IoT	Internet of Things
IP	Intellectual Property
IR	Injection Rate
IRTS	International Technology Roadmap for Semiconductors
ISA	Instruction Set Architecture
ISS	Instruction-Set Simulator
ITIV	Institut für Technik der Informationsverarbeitung
KIT	Karlsruher Institut für Technologie
KPN	Kahn Process Networks
LUT	Look Up Table
MoC	Model of Computation
MPI	Message Passing Interface
MPPA	Multi-Purpose Processor Architecture
MPSoC	Multiprocessor System on a Chip
NA	Network Adapter
NI	Network Interface
NoC	Network on Chip
NUMA	Non-Uniform Memory Access
OS	Operating System
OVP	Open Virtual Platforms
PCI	Peripheral Component Interconnect

PCIe	Peripheral Component Interconnect express
PE	Processing Element
PGAS	Partitioned Global Address Space
PS	Packet Switching
PSE	Peripheral Simulation Engine
QEMU	Quick Emulator
QoR	Quality of Result
QoS	Quality of Service
RAM	Random Access Memory
RBCC	Region-Based Cache Coherence
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Level
SCC	Single-chip Cloud Computer
SDM	Spatial Division Multiplexing
SoA	State of the Art
SoC	System on Chip
SPARC	Scalable Processor Architecture
SRAM	Static Random-Access Memory
TCPA	Tightly-Coupled Processor Array
TDM	Time Division Multiplexing
TLB	Translation Lookaside Buffer
TLM	Tile Local Memory

USB	Universal Serial Bus
VC	Virtual Channel
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VMI	Virtual Machine Interface
VP	Virtual Platform

External Literature

[1] *ARM big.LITTLE*. https://www.arm.com/why-arm/technologies/big-little,

[2] *KALRAY MPPA3 Coolidge*. http://www.mpsoc-forum.org/archive/2017/files/proceedings/Benoit_Dinechin.pdf,

[3] *Open Virtual Platforms (OVP)*. http://www.ovpworld.org/,

[4] *proDesign proFPGA*. https://www.profpga.com/,

[5] *Synopsys Hybrid Prototyping Solution*. https://www.synopsys.com/verification/virtual-prototyping/virtualizer/hybrid-prototyping.html,

[6] Agarwal, Ankur ; Iskander, Cyril ; Shankar, Ravi: Survey of network on chip (noc) architectures & contributions. In: *Journal of engineering, Computing and Architecture* 3 (2009), Nr. 1, S. 21–27

[7] Alonso, M. G. ; Flich, J.: PROSA: Protocol-Driven Network on Chip Architecture. In: *IEEE Transactions on Parallel and Distributed Systems* (2017), S. 1–1. – ISSN 1045–9219

[8] Amdahl, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, ACM Press, 1967

[9] Anderson, D. ; Shanley, T. ; Inc, MindShare: *Pentium Processor System Architecture*. Addison-Wesley, 1995 (Mindshare PC System Architecture). – ISBN 9780201409925

[10] Arato, P. ; Jahasz, S. ; Mann, Z.A. ; Orban, A. ; Papp, D.: Hardware-software partitioning in embedded system design. In: *IEEE International Symposium on Intelligent Signal Processing, 2003*, IEEE

[11] Baillie, Clive F.: Comparing shared and distributed memory computers. In: *Parallel Computing* 8 (1988), oct, Nr. 1-3, S. 101–110

[12] Bellard, Fabrice: QEMU, a fast and portable dynamic translator. In: *USENIX Annual Technical Conference, FREENIX Track* Bd. 41, 2005, S. 46

[13] Benini, L. ; Micheli, G. D.: Networks on chips: a new SoC paradigm. In: *Computer* 35 (2002), Nr. 1, S. 70–78

[14] Bienia, Christian: *Benchmarking Modern Multiprocessors*, Princeton University, Diss., January 2011

[15] Binkert, Nathan ; Sardashti, Somayeh ; Sen, Rathijit ; Sewell, Korey ; Shoaib, Muhammad ; Vaish, Nilay ; Hill, Mark D. ; Wood, David A. ; Beckmann, Bradford ; Black, Gabriel ; Reinhardt, Steven K. ; Saidi, Ali ; Basu, Arkaprava ; Hestness, Joel ; Hower, Derek R. ; Krishna, Tushar: The gem5 simulator. In: *ACM SIGARCH Computer Architecture News* 39 (2011), aug, Nr. 2, S. 1. – ISSN 0163–5964

[16] Bjerregaard, Tobias: The MANGO clockless network-on-chip: Concepts and implementation. In: *IMM, Danmarks Tekniske Universitet* (2005)

[17] Blem, Emily ; Menon, Jaikrishnan ; Vijayaraghavan, Thiruvengadam ; Sankaralingam, Karthikeyan: ISA Wars. In: *ACM Transactions on Computer Systems* 33 (2015), mar, Nr. 1, S. 1–34

[18] Borgstrom, Tom ; Haritan, Eshel ; Wilson, Ron ; Abada, David ; Dauman, Andrew ; Chandra, Ramesh ; Mielo, Olivier ; Cruse, Chuck ; Nohl, Achim: System prototypes: Virtual, Hardware or Hybrid? In: *Proceedings of the 46th Annual Design Automation Conference - DAC '09*, ACM Press, 2009

[19] Borkar, Shekhar: Thousand Core ChipsA Technology Perspective. In: *44th ACM/IEEE Design Automation Conference*, IEEE, jun 2007

[20] Burgio, Paolo ; Marongiu, Andrea ; Coussy, Philippe ; Benini, Luca: A HLS-Based Toolflow to Design Next-Generation Heterogeneous Many-Core Platforms with Shared Memory. In: *12th IEEE International Conference on Embedded and Ubiquitous Computing*, IEEE, aug 2014

[21] Butko, Anastasiia ; Gamatie, Abdoulaye ; Sassatelli, Gilles ; Torres, Lionel ; Robert, Michel: Design Exploration for next Generation High-Performance Manycore On-chip Systems: Application to big.LITTLE Architectures. In: *IEEE Computer Society Annual Symposium on VLSI*, IEEE, jul 2015

[22] Butko, Anastasiia ; Garibotti, Rafael ; Ost, Luciano ; Lapotre, Vianney ; Gamatie, Abdoulaye ; Sassatelli, Gilles ; Adeniyi-Jones, Chris: A trace-driven approach for fast and accurate simulation of manycore architectures. In: *The 20th Asia and South Pacific Design Automation Conference*, IEEE, jan 2015

[23] Butko, Anastasiia ; Garibotti, Rafael ; Ost, Luciano ; Sassatelli, Gilles: Accuracy evaluation of GEM5 simulator system. In: *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, IEEE, jul 2012

[24] Cai, L. ; Gajski, D.: Transaction level modeling: an overview. In: *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, ACM

[25] Canis, Andrew ; Choi, Jongsok ; Fort, Blair ; Lian, Ruolong ; Huang, Qijing ; Calagar, Nazanin ; Gort, Marcel ; Qin, Jia J. ; Aldham, Mark ; Czajkowski, Tomasz ; Brown, Stephen ; Anderson, Jason: From software to accelerators with LegUp high-level synthesis. In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, IEEE, sep 2013

[26] Carlson, Trevor E. ; Heirman, Wim ; Eeckhout, Lieven: Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, ACM Press, 2011

[27] Chaiken, D. ; Fields, C. ; Kurihara, K. ; Agarwal, A.: Directory-based cache coherence in large-scale multiprocessors. In: *Computer* 23 (1990), jun, Nr. 6, S. 49–58

[28] Chiou, Derek ; Sunwoo, Dam ; Kim, Joonsoo ; Patil, Nikhil A. ; Reinhart, William ; Johnson, Darrel E. ; Keefe, Jebediah ; Angepat, Hari: FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, IEEE, 2007

[29] Chopard, B. ; Combes, P. ; Zory, J.: A Conservative Approach to SystemC Parallelization. In: *Computational Science – ICCS 2006*. Springer Berlin Heidelberg, 2006, S. 653–660

[30] Cong, Jason ; Ghodrat, Mohammad A. ; Gill, Michael ; Grigorian, Beayna ; Huang, Hui ; Reinman, Glenn: Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity. In: *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. Piscataway, NJ, USA : IEEE Press, 2013 (ISLPED ’13). – ISBN 978–1–4799–1235–3, 305–310

[31] Cong, Jason ; Ghodrat, Mohammad A. ; Gill, Michael ; Grigorian, Beayna ; Reinman, Glenn: Architecture support for accelerator-rich CMPs. In: *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, ACM Press, 2012

[32] Cong, Jason ; Liu, Bin ; Neuendorffer, Stephen ; Noguera, Juanjo ; Vissers, Kees ; Zhang, Zhiru: High-Level Synthesis for FPGAs: From Prototyping to Deployment. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30 (2011), apr, Nr. 4, S. 473–491

[33] Deest, Gael ; Yuki, Tomofumi ; Sentieys, Olivier ; Derrien, Steven: Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In: *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, nov 2014

[34] Deleganes, D. ; Douglas, J. ; Kommandur, B. ; Patyra, M.: Designing a 3 GHz, 130 nm, Intel Pentium 4 processor. In: *2002 Symposium on VLSI Circuits. Digest of Technical Papers (Cat. No.02CH37302)*, IEEE

[35] Delicia, G Shalina P. ; Bruckschloegl, Thomas ; Figuli, Peter ; Tradowsky, Carsten ; Marchesan, Gabriel ; Almeida, Juergen B.: Bringing accuracy to Open Virtual Platforms (OVP): A safari from high-level tools to low-level microarchitectures. In: *International Journal of Computer Applications* 975 (2013), S. 8887

[36] Demetriades, S. ; Cho, S.: Predicting Coherence Communication by Tracking Synchronization Points at Run Time. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012. – ISSN 1072–4451, S. 351–362

[37] Dennard, Robert H. ; Gaensslen, Fritz H. ; Rideout, V L. ; Bassous, Ernest ; LeBlanc, Andre R.: Design of ion-implanted MOSFET's with very small physical dimensions. In: *IEEE Journal of Solid-State Circuits* 9 (1974), Nr. 5, S. 256–268

[38] Derrien, Steven ; Puaut, Isabelle ; Alefragis, Panayiotis ; Bednara, Marcus ; Bucher, Harald ; David, Clement ; Debray, Yann ; Durak, Umut ; Fassi, Imen ; Ferdinand, Christian ; Hardy, Damien ; Kritikakou, Angeliki ; Rauwerda, Gerard ; Reder, Simon ; Sicks, Martin ; Stripf, Timo ; Sunesen, Kim ; Braak, Timon ter ; Voros, Nikolaos ; Becker, Jurgen: WCET-aware parallelization of model-based applications for multi-cores: The ARGO approach. In: *Design,*

Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE, mar 2017

[39] Dinechin, Benoit D.: Kalray MPPA®: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor. In: *2015 IEEE Hot Chips 27 Symposium (HCS)*, IEEE, aug 2015

[40] Dorai, Atef ; Sentieys, Olivier ; Dubois, Helene: Evaluation of NoC on multi-FPGA interconnection using GTX transceiver. In: *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, IEEE, dec 2017

[41] Esmaeilzadeh, Hadi ; Blem, Emily ; Amant, Renee S. ; Sankaralingam, Karthikeyan ; Burger, Doug: Dark silicon and the end of multicore scaling. In: *2011 38th Annual international symposium on computer architecture (ISCA)* IEEE, 2011, S. 365–376

[42] Fang, Jianbin ; Varbanescu, Ana L. ; Sips, Henk: A Comprehensive Performance Comparison of CUDA and OpenCL. In: *2011 International Conference on Parallel Processing*, IEEE, sep 2011

[43] Fatahalian, Kayvon ; Horn, Daniel R. ; Knight, Timothy J. ; Leem, Larkhoon ; Houston, Mike ; Park, Ji Y. ; Erez, Mattan ; Ren, Manman ; Aiken, Alex ; Dally, William J. ; Hanrahan, Pat: Sequoia: Programming the Memory Hierarchy. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA : ACM, 2006 (SC '06). – ISBN 0–7695–2700–0

[44] Figuli, Peter ; Hubner, Michael ; Girardey, Romuald ; Bapp, Falco ; Bruckschlogl, Thomas ; Thoma, Florian ; Henkel, Jorg ; Becker, Jurgen: A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion. In: *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, IEEE, jun 2011

[45] Foster, Harry D.: Why the Design Productivity Gap Never Happened. In: *Proceedings of the International Conference on Computer-Aided Design*. Piscataway, NJ, USA : IEEE Press, 2013 (ICCAD '13). – ISBN 978–1–4799–1069–4, 581–584

[46] Gajski, Dan ; Austin, Todd ; Svoboda, Steve: What input-language is the best choice for high level synthesis (HLS)? In: *Proceedings of the 47th Design Automation Conference on - DAC '10*, ACM Press, 2010

[47] Gajski, Daniel D. ; Kuhn, Robert H.: New VLSI tools. In: *Computer* (1983), Nr. 12, S. 11–14

[48] Gerstlauer, A. ; Haubelt, C. ; Pimentel, A.D. ; Stefanov, T.P. ; Gajski, D.D. ; Teich, J.: Electronic System-Level Synthesis Methodologies. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (2009), oct, Nr. 10, S. 1517–1530

[49] Gilles, KAHN: The semantics of a simple language for parallel programming. In: *Information processing* 74 (1974), S. 471–475

[50] Goossens, Kees ; Bennebroek, Martijn ; Hur, Jae Y. ; Wahlah, Muhammad A.: Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In: *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, IEEE, apr 2008

[51] Goossens, Kees ; Nejad, Ashkan B. ; Nelson, Andrew ; Sinha, Shubhendu ; Azevedo, Arnaldo ; Chandrasekar, Karthik ; Gomony, Manil D. ; Goossens, Sven ; Koedam, Martijn ; Li, Yonghui ; Mirzoyan, Davit ; Molnos, Anca: Virtual execution platforms for mixed-time-criticality systems. In: *ACM SIGBED Review* 10 (2013), oct, Nr. 3, S. 23–34

[52] Goossens, Sven ; Akesson, Benny ; Koedam, Martijn ; Nejad, Ashkan B. ; Nelson, Andrew ; Goossens, Kees: The CompSOC design flow for virtual execution platforms. In: *Proceedings of the 10th FPGAworld Conference on - FPGAworld '13*, ACM Press, 2013

[53] Gustafson, John L.: Reevaluating Amdahl's law. In: *Communications of the ACM* 31 (1988), Nr. 5, S. 532–533

[54] Haller, Philipp: On the integration of the actor model in mainstream technologies. In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions - AGERE! '12*, ACM Press, 2012

[55] Heisswolf, Jan: *A Scalable and Adaptive Network on Chip for Many-Core Architectures*, Diss., 2014

[56] Henkel, Jorg ; Bauer, Lars ; Hubner, Michael ; Grudnitsky, Artjom: i-Core: A run-time adaptive processor for embedded multi-core systems. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)* The Steering Committee of The World Congress in Computer Science, 2011, S. 1

[57] Henkel, Jorg ; Herkersdorf, Andreas ; Bauer, Lars ; Wild, Thomas ; Hubner, Michael ; Pujari, Ravi K. ; Grudnitsky, Artjom ; Heisswolf, Jan ; Zaib, Aurang ; Vogel, Benjamin ; Lari, Vahid ; Kobbe, Sebastian: Invasive manycore architectures. In: *17th Asia and South Pacific Design Automation Conference*, IEEE, jan 2012

[58] Hennessy, J.L. ; Patterson, D.A.: *Computer Architecture: A Quantitative Approach,6th edition*. Elsevier Science, 2017 (The Morgan Kaufmann Series in Computer Architecture and Design). – ISBN 9780128119051

[59] Howard, J ; Dighe, S ; Vangal, S R. ; Ruhl, G ; Borkar, N ; Jain, S ; Erraguntla, V ; Konow, M ; Riepen, M ; Gries, M ; Droege, G ; Lund-Larsen, T ; Steibl, S ; Borkar, S ; De, V K. ; Wijngaart, R Van D.: A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. In: *IEEE Journal of Solid-State Circuits* 46 (2011), jan, Nr. 1, S. 173–183

[60] International Telecomunication Union: *Open Systems Interconnection - Basic Reference Model: The basic model*. https://www.itu.int/rec/T-REC-X.200-199407-I/en, 1994

[61] *International Roadmap for Devices and Systems 2018 Edition, Executive Summary*. https://irds.ieee.org/editions/2018,

[62] Jerger, N. D. E. ; Peh, L. S. ; Lipasti, M. H.: Circuit-Switched Coherence. In: *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, 2008, S. 193–202

[63] Jiang, Nan ; Balfour, James ; Becker, Daniel U. ; Towles, Brian ; Dally, William J. ; Michelogiannakis, George ; Kim, John: A detailed and flexible cycle-accurate Network-on-Chip simulator. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, apr 2013

[64] Kamali, Hadi M. ; Hessabi, Shahin: AdapNoC: A fast and flexible FPGA-based NoC simulator. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, aug 2016

[65] Kegel, Philipp ; Steuwer, Michel ; Gorlatch, Sergei: dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IEEE, may 2012

[66] Krishna, T. ; Chen, C. H. O. ; Kwon, W. C. ; Peh, L. S.: Breaking the on-chip latency barrier using SMART. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013. – ISSN 1530–0897, S. 378–389

[67] Kumar, Amit ; Peh, Li-Shiuan ; Kundu, Partha ; Jha, Niraj K.: Express Virtual Channels: Towards the Ideal Interconnection Fabric. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2007 (ISCA ’07). – ISBN 978–1–59593–706–3, S. 150–161

[68] Kumar, R. ; Tullsen, D.M. ; Jouppi, N.P. ; Ranganathan, P.: Heterogeneous chip multiprocessors. In: *Computer* 38 (2005), nov, Nr. 11, S. 32–38

[69] Kumar, R. ; Tullsen, D.M. ; Ranganathan, P. ; Jouppi, N.P. ; Farkas, K.I.: Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: *Proceedings. 31st Annual International Symposium on Computer Architecture*, IEEE, 2004

[70] Kwon, Young-Su ; Kyung, Chong-Min: Performance-driven event-based synchronization for multi-FPGA simulation accelerator with event time-multiplexing bus. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24 (2005), sep, Nr. 9, S. 1444–1456

[71] Lavagno, Luciano ; Kondratyev, Alex ; Watanabe, Yosinori ; Zhu, Qiang ; Fujii, Mototsugu ; Tatesawa, Mitsuru ; Nakayama, Noriyasu: Incremental high-level synthesis. In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, jan 2010

[72] Lawley, Jason: Understanding Performance of PCI Express Systems. In: *Xilinx. Rev* 1 (2014)

[73] Lee, E.A. ; Messerschmitt, D.G.: Synchronous data flow. In: *Proceedings of the IEEE* 75 (1987), Nr. 9, S. 1235–1245

[74] Lemonnier, Fabrice ; Millet, Philippe ; Almeida, Gabriel M. ; Hubner, Michael ; Becker, Jurgen ; Pillement, Sebastien ; Sentieys, Olivier ; Koedam, Martijn ; Sinha, Shubhendu ; Goossens, Kees ; Piguet, Christian ; Morgan, Marc-Nicolas ; Lemaire, Romain: Towards future adaptive multiprocessor systems-on-chip: An innovative approach for flexible architectures. In: *2012 International Conference on Embedded Computer Systems (SAMOS)*, IEEE, jul 2012

[75] Loh, Gabriel H.: 3D-Stacked Memory Architectures for Multi-core Processors. In: *2008 International Symposium on Computer Architecture*, IEEE, jun 2008

[76] Lotlikar, Swapnil ; Pai, Vinayak ; Gratz, Paul V.: AcENoCs: A Configurable HW/SW Platform for FPGA Accelerated NoC Emulation. In: *2011 24th Internatioal Conference on VLSI Design*, IEEE, jan 2011

[77] Lyberis, Spyros ; Kalokerinos, George ; Lygerakis, Michalis ; Papaefstathiou, Vassilis ; Tsaliagkos, Dimitris ; Katevenis, Manolis ; Pnevmatikatos, Dionisios ; Nikolopoulos, Dimitris: Formic: Cost-efficient and Scalable Prototyping of Manycore Architectures. In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, apr 2012

[78] Mahzoon, Alireza ; Alizadeh, Bijan: OptiFEX: A Framework for Exploring Area-Efficient Floating Point Expressions on FPGAs With Optimized Exponent/Mantissa Widths. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25 (2017), jan, Nr. 1, S. 198–209

[79] Martin, G. ; Smith, G.: High-Level Synthesis: Past, Present, and Future. In: *IEEE Design & Test of Computers* 26 (2009), jul, Nr. 4, S. 18–25

[80] Martin, Grant ; Bailey, Brian ; Piziali, Andrew: *ESL design and verification: a prescription for electronic system level methodology*. Elsevier, 2010

[81] McFarland, M.C. ; Parker, A.C. ; Camposano, R.: The high-level synthesis of digital systems. In: *Proceedings of the IEEE* 78 (1990), Nr. 2, S. 301–318

[82] Meeus, Wim ; Beeck, Kristof V. ; Goedemé, Toon ; Meel, Jan ; Stroobandt, Dirk: An overview of today's high-level synthesis tools. In: *Design Automation for Embedded Systems* 16 (2012), aug, Nr. 3, S. 31–51

[83] Mentor: *Catapult High-Level Synthesis*. https://www.mentor.com /hls-lp/catapult-high-level-synthesis/c-systemc-hls,

[84] Milthorpe, Josh ; Ganesh, V. ; Rendell, Alistair P. ; Grove, David: X10 as a Parallel Language for Scientific Computation: Practice and Experience. In: *2011 IEEE International Parallel & Distributed Processing Symposium*, IEEE, may 2011

[85] Mittal, Sparsh: A Survey Of Techniques for Architecting and Managing Asymmetric Multicore Processors. In: *ACM Computing Surveys* 48 (2016), 02

[86] Modarressi, M. ; Sarbazi-Azad, H. ; Arjomand, M.: A hybrid packet-circuit switched on-chip network based on SDM. In: *2009 Design, Automation Test in Europe Conference Exhibition*, 2009. – ISSN 1530–1591, S. 566–569

[87] Moore, Gordon E. u. a.: *Cramming more components onto integrated circuits*. 1965

[88] Munshi, Aaftab: The OpenCL specification. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*, IEEE, aug 2009

[89] Nejad, Ashkan B. ; Molnos, Anca ; Martinez, Matias E. ; Goossens, Kees: A hardware/software platform for QoS bridging over multi-chip NoC-based systems. In: *Parallel Computing* 39 (2013), sep, Nr. 9, S. 424–441

[90] Ni, Yi ; Mong, Wai S. ; Zhu, Jianwen: On virtual prototyping of embedded system-on-chips. In: *2011 9th IEEE International Conference on ASIC*, IEEE, oct 2011

[91] Oliveira, H. F. A. ; Bucher, H. ; Brito, A. V. ; Araújo, J. M. F. R. ; Melcher, E. U. K. ; Duenha, L.: Power-aware design of electronic system level using interoperation of hybrid and distributed simulations. In: *2015 28th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2015, S. 1–7

[92] Olofsson, Andreas: Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip. In: *CoRR* abs/1610.01832 (2016). http://arxiv.org/abs/1610.01832

[93] Oussorov, I. ; Raab, W. ; Hachmann, U. ; Kravtsov, A.: Integration of instruction set simulators into SystemC high level models. In: *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, IEEE Comput. Soc

[94] Panerati, Jacopo ; Sciuto, Donatella ; Beltrame, Giovanni: Optimization Strategies in Design Space Exploration. Version: 2017. http://dx.doi.org/10.1007/978-94-017-7267-97. In: *Handbook of Hardware/Software Codesign*. Springer Netherlands, 2017. – DOI 10.1007/978–94–017–7267–97, S. 189–216

[95] Papamichael, Michael K.: Fast scalable FPGA-based Network-on-Chip simulation models. In: *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, IEEE, jul 2011

[96] Pelcat, Maxime ; Bourrasset, Cedric ; Maggiani, Luca ; Berry, Francois: Design productivity of a high level synthesis compiler versus HDL. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, jul 2016

[97] Pham-Quoc, Cuong ; Ashraf, Imran ; Al-Ars, Zaid ; Bertels, Koen: Heterogeneous Hardware Accelerators with Hybrid Interconnect: An Automated Design Approach. In: *2015 International Conference on Advanced Computing and Applications (ACOMP)*, IEEE, nov 2015

[98] Preskill, John: Quantum Computing in the NISQ era and beyond. In: *Quantum* 2 (2018), S. 79

[99] Protic, J. ; Tomasevic, M. ; Milutinovic, V.: Distributed shared memory: concepts and systems. In: *IEEE Parallel & Distributed Technology: Systems & Applications* 4 (1996), Nr. 2, S. 63–71

[100] Real, Maria M. ; Wehner, Philipp ; Rettkowski, Jens ; Migliore, Vincent ; Lapotre, Vianney ; Gohringer, Diana ; Gogniat, Guy: MPSoCSim extension: An OVP simulator for the evaluation of cluster-based multi and many-core architectures. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, jul 2016

[101] *Rodinia: A Benchmark Suite for Heterogeneous Computing*. http://lava.cs.virginia.edu/Rodinia/, 2016

[102] Rodriguez-Andina, J.J. ; Moure, M.J. ; Valdes, M.D.: Features, Design Tools, and Application Domains of FPGAs. In: *IEEE Transactions on Industrial Electronics* 54 (2007), aug, Nr. 4, S. 1810–1823

[103] Rose, Adam ; Swan, Stuart ; Pierce, John ; Fernandez, Jean-Michel u. a.: Transaction level modeling in SystemC. In: *Open SystemC Initiative* 1 (2005), Nr. 1.297

[104] Roth, Christoph ; Almeida, Gabriel M. ; Sander, Oliver ; Ost, Luciano ; Hebert, Nicolas ; Sassatelli, Gilles ; Benoit, Pascal ; Torres,

Lionel ; Becker, Jurgen: Modular Framework for Multi-level Multi-device MPSoC Simulation. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, IEEE, may 2011

[105] Saboori, Ehsan ; Abdi, Samar: Hybrid Prototyping of Multicore Embedded Systems. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, IEEE Conference Publications, 2013

[106] Schreiner, Soren ; Gorgen, Ralph ; Gruttner, Kim ; Nebel, Wolfgang: A quasi-cycle accurate timing model for binary translation based instruction set simulators. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, jul 2016

[107] Shalf, John M. ; Leland, Robert: Computing beyond Moore's Law. In: *Computer* 48 (2015), dec, Nr. 12, S. 14–23

[108] Shankar, S. S. ; PinXing, L. ; Herkersdorf, A. ; Wild, T.: BiSME: A Hardware Coprocessor to Perform Signature Matching at Multi-Gigabit Rates. In: *2018 IEEE 29th Int. Conf. on Appl.-spec. Syst., Arch. and Proc. (ASAP)*, 2018, S. 1–9

[109] Shim, Kyuho ; Kim, Woojoo ; Cho, Kwang-Hyun ; Min, Byeong: System-level simulation acceleration for architectural performance analysis using hybrid virtual platform system. In: *2012 International SoC Design Conference (ISOCC)*, IEEE, nov 2012

[110] Silva, Joao ; Sklyarov, Valery ; Skliarova, Iouliia: Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip. In: *IEEE Embedded Systems Letters* 7 (2015), mar, Nr. 1, S. 31–34

[111] Singh, Aameek ; Korupolu, Madhukar ; Mohapatra, Dushmanta: Server-storage Virtualization: Integration and Load Balancing in Data Centers. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA : IEEE Press, 2008 (SC ’08). – ISBN 978–1–4244–2835–9, 53:1–53:12

[112] Snir, Marc ; Gropp, William ; Otto, Steve ; Huss-Lederman, Steven ; Dongarra, Jack ; Walker, David: *MPI–the Complete Reference: The MPI core*. Bd. 1. MIT press, 1998

[113] Sodani, Avinash ; Gramunt, Roger ; Corbal, Jesus ; Kim, Ho-Seop ; Vinod, Krishna ; Chinthamani, Sundaram ; Hutsell, Steven ; Agarwal, Rajat ; Liu, Yen-Chen: Knights Landing: Second-Generation Intel Xeon Phi Product. In: *IEEE Micro* 36 (2016), mar, Nr. 2, S. 34–46

[114] Srinivasan, Jayanth: An overview of static power dissipation. In: *CiteSeer public search engine and digital libraries for scientific and academic papers in the fields of computer and information science* (2011), S. 1–7

[115] Srinivasan, V. ; Radhakrishnan, S. ; Vemuri, R.: Hardware software partitioning with integrated hardware design space exploration. In: *Proceedings Design, Automation and Test in Europe*, IEEE Comput. Soc

[116] Srivatsa, A. ; Rheindt, S. ; Wild, T. ; Herkersdorf, A.: Region based cache coherence for tiled MPSoCs. In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, S. 286–291

[117] Stralen, Peter van ; Pimentel, Andy: Scenario-based design space exploration of MPSoCs. In: *2010 IEEE International Conference on Computer Design*, IEEE, oct 2010

[118] Sullivan, Chris ; Wilson, Alex ; Chappell, Stephen: Using C Based Logic Synthesis to Bridge the Productivity Gap. In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA : IEEE Press, 2004 (ASP-DAC ’04). – ISBN 0–7803–8175–0, 349–354

[119] Takach, Andres: High-Level Synthesis: Status, Trends, and Future Directions. In: *IEEE Design & Test* 33 (2016), jun, Nr. 3, S. 116–124

[120] Tang, Qingshan ; Mehrez, Habib ; Tuna, Matthieu: Multi-FPGA prototyping board issue: the FPGA I/O bottleneck. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, IEEE, jul 2014

[121] Teich, Jürgen ; Henkel, Jörg ; Herkersdorf, Andreas ; Schmitt-Landsiedel, Doris ; Schröder-Preikschat, Wolfgang ; Snelting, Gregor: Invasive computing: An overview. In: *Multiprocessor System-on-Chip*. Springer, 2011, S. 241–268

[122] Teich, Jürgen: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. In: *Proceedings of the IEEE* 100 (2012), may, Nr. Special Centennial Issue, S. 1411–1430

[123] Teimouri, N. ; Modarressi, M. ; Sarbazi-Azad, H.: Power and Performance Efficient Partial Circuits in Packet-Switched Networks-on-Chip. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013. – ISSN 1066–6192, S. 509–513

[124] Teimouri, Nasibeh ; Modarressi, Mehdi ; Tavakkol, Arash ; Sarbazi-azad, Hamid: Energy-Optimized On-Chip Networks Using Reconfigurable Shortcut Paths. In: *Architecture of Computing Systems - ARCS*, 2011. – ISBN 978–3–642–19137–4, S. 231–242

[125] Venkatesh, Ganesh ; Sampson, Jack ; Goulding, Nathan ; Garcia, Saturnino ; Bryksin, Vladyslav ; Lugo-Martinez, Jose ; Swanson, Steven ; Taylor, Michael B.: Conservation cores. In: *ACM SIGPLAN Notices* 45 (2010), mar, Nr. 3, S. 205

[126] Wael, Mattias D. ; Marr, Stefan ; Fraine, Bruno D. ; Cutsem, Tom V. ; Meuter, Wolfgang D.: Partitioned Global Address Space Languages. In: *ACM Computing Surveys* 47 (2015), may, Nr. 4, S. 1–27

[127] Wang, D. ; Lo, C. ; Vasiljevic, J. ; Jerger, N. E. ; Steffan, J. G.: DART: A Programmable Architecture for NoC Simulation on FPGAs. In: *IEEE Transactions on Computers* 63 (2014), mar, Nr. 3, S. 664–678

[128] Wehner, Philipp ; Rettkowski, Jens ; Kleinschmidt, Tobias ; Gohringer, Diana: MPSoCSim: An extended OVP simulator for modeling and evaluation of Network-on-Chip based heterogeneous MP-SoCs. In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, jul 2015

[129] Woo, Steven C. ; Ohara, Moriyoshi ; Torrie, Evan ; Singh, Jaswinder P. ; Gupta, Anoop: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1995 (ISCA ’95). – ISBN 0–89791–698–0, S. 24–36

[130] Xilinx: *Vivado High-Level Synthesis*. https://www.xilinx.com /products/design-tools/vivado/integration/esl-design.html,

[131] Xilinx: *Zynq*. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000,

[132] Yoo, Sungjoo ; Jerraya, A.A.: Introduction to hardware abstraction layers for SoC. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*, IEEE Comput. Soc

[133] Zaib, Muhammad A.: *Network on Chip Interface for Scalable Distributed Shared Memory Architectures*. München, Technische Universität München, Dissertation, 2018

[134] Zhu, Jianwen ; Gajski, D.D.: An ultra-fast instruction set simulator. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10 (2002), jun, Nr. 3, S. 363–373

[135] Zijp, Jeroen. van d.: Fast half float conversions. (2012)

[136] Zivojnovic, Vojin ; Pees, Stefan ; Meyr, Heinrich: LISA-machine description language and generic machine model for HW/SW co-design. In: *VLSI Signal Processing, IX*, IEEE

Supervised Student Works

[Bar17] Baranska, Joanna: *Cross-Domain Prototyping of NoC-Based Many-Core Platforms*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2022, January 2017

[Bor15] Borgmeyer, Hendrik: *Parallelization of the High-level Simulation (OVP) of a Multicore Platform*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-1957, March 2015

[Gra15] Gramlich, Georg: *Intelligent Camera based Driver Assistance Systems for a Roadtrain Scenario*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2056, September 2015

[Hel16] Held, Felix: *Parameterization of the InvasIC Architecture for Prototyping Purposes on a FPGA Evaluation Board*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Master's Thesis ID-2117, June 2016

[Kre17] Kress, Fabian: *Dynamic Circuit Switching in the Invasive NoC*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2217, March 2017

[Les14] Lesniak, Fabian: *Interaction of an OVP Simulation with Real World Devices*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-1909, December 2014

[Les18] Lesniak, Fabian: *Development of an Efficient and Scalable NoC Prototyping Environment*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Master's Thesis ID-2356, May 2018

[Li16] Li, Hui: *Connection of an Instruction Set Simulator to the iNoC Simulation Framework*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2166, October 2016

[Lu17] Lu, Tianyu: *Concept, Evaluation and Implementation of a Real-Time Capable Memory Infrastructure for Many-Core Systems*, Karlsruhe

Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Master's Thesis ID-2263, September 2017

[Ung16] Unger, Kai L.: *Implementation and Evaluation of a Circuit Switching Extension of the Invasive NoC in SystemC*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2142, June 2016

[Xia19] Xiao, Fan: *High-Level Synthesis for the Design of Networks-on-chip*, Karlsruhe Institute of Technology, Institut für Technik der Informationsverarbeitung (ITIV), Bachelor's Thesis ID-2557, July 2019

Own Conference Articles

[AKM+19] Anantharajaiah, N. ; Kempf, F. ; Masing, L. ; Lesniak, F. M. ; Becker, J.: Dynamic and scalable runtime block-based multicast routing for networks on chips. In: *Proceedings of the 12th International Workshop on Network on Chip Architectures - NoCArc*, ACM Press, 2019

[HFM+16] Heisswolf, J. ; Friederich, S. ; Masing, L. ; Weichslgartner, A. ; Zaib, M. A. ; Stein, C. ; Duden, M. ; Teich, J. ; Herkersdorf, A. ; Becker, J.: A Novel NoC-Architecture for Fault Tolerance and Power Saving. In: *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, 2016, S. 1–8

[HWZ+15] Heisswolf, Jan ; Weichslgartner, Andreas ; Zaib, Aurang ; Friederich, Stephanie ; Masing, Leonard ; Stein, Carsten ; Duden, Marco ; Klopfer, Roman ; Teich, Jurgen ; Wild, Thomas ; Herkersdorf, Andreas ; Becker, Jurgen: Fault-tolerant communication in invasive networks on chip. In: *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, IEEE, jun 2015

[JSK+15] Janssen, Benedikt ; Schwiegelshohn, Fynn ; Koedam, Martijn ; Duhem, Francois ; Masing, Leonard ; Werner, Stephan ; Huriaux, Christophe ; Courtay, Antoine ; Wheatley, Emilie ; Goossens, Kees ; Lemonnier, Fabrice ; Millet, Philippe ; Becker, Jurgen ; Sentieys, Olivier ; Hubner, Michael: Designing applications for heterogeneous many-core architectures with the FlexTiles Platform. In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, jul 2015

[KAJB19] Kempf, Fabian ; Anantharajaiah, Nidhi ; Juergen Becker, Leonard M.: A Network on Chip Adapter for Real-Time and Safety-Critical Applications. In: *System on Chip conference (SOCC)*, 2019

[MLB19] Masing, Leonard ; Lesniak, Fabian ; Becker, Jurgen: Hybrid Prototyping for Manycore Design and Validation. In: *Intelligent Information and Database Systems*, Springer International Publishing, 2019, S. 319–333

[MSK$^+$18] Masing, Leonard ; Srivatsa, Akshay ; Kress, Fabian ; Anantharajaiah, Nidhi ; Herkersdorf, Andreas ; Becker, Jurgen: In-NoC Circuits for Low-Latency Cache Coherence in Distributed Shared-Memory Architectures. In: *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, IEEE, sep 2018

[MWB15] Masing, Leonard ; Werner, Stephan ; Becker, Jurgen: Virtual prototyping of heterogeneous dynamic platforms using Open Virtual Platforms. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, IEEE, jun 2015

[PKMW11] Pankratius, Victor ; Knittel, Fabian ; Masing, Leonard ; Walser, Martin: OpenMPspy: Leveraging Quality Assurance for Parallel Software. In: *Euro-Par 2011 Parallel Processing*, Springer Berlin Heidelberg, 2011, S. 124–135

[RMB$^+$18] Reder, Simon ; Masing, Leonard ; Bucher, Harald ; Braak, Timon ter ; Stripf, Timo ; Becker, Jurgen: A WCET-aware parallel programming model for predictability enhanced multi-core architectures. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, mar 2018

[SXMS$^+$16] Sotiriou-Xanthopoulos, Efstathios ; Masing, Leonard ; Siozios, Kostas ; Economakos, George ; Soudris, Dimitrios ; Becker, Jurgen: An OpenCL-based framework for rapid virtual prototyping of heterogeneous architectures. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, jul 2016

[WMLB15] Werner, Stephan ; Masing, Leonard ; Lesniak, Fabian ; Becker, Jurgen: Software-in-the-Loop simulation of embedded control applications based on Virtual Platforms. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, sep 2015

Own Journal Articles

[MLB20] MASING, L. ; LESNIAK, F. ; BECKER, J.: A Hybrid Prototyping Framework in a Virtual Platform Centered Design and Verification Flow. In: *IEEE Embedded Systems Letters* (2020)

[SXMX+18] SOTIRIOU-XANTHOPOULOS, Efstathios ; MASING, Leonard ; XYDIS, Sotirios ; SIOZIOS, Kostas ; BECKER, Jrgen ; SOUDRIS, Dimitrios: OpenCL-based Virtual Prototyping and Simulation of Many-Accelerator Architectures. In: *ACM Transactions on Embedded Computing Systems* 17 (2018), sep, Nr. 5, S. 1–27. `http://dx.doi.org/10.1145/3242179`. – DOI 10.1145/3242179