Scalable System-on-Chip Design

Paolo Mantovani

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2017

ABSTRACT

Scalable System-on-Chip Design

Paolo Mantovani

The crisis of technology scaling led the industry of semiconductors towards the adoption of disruptive technologies and innovations to sustain the evolution of microprocessors and keep under control the timing of the design cycle. Multi-core and many-core architectures sought more energy-efficient computation by replacing a power-hungry processor with multiple simpler cores exploiting parallelism. Multi-core processors alone, however, turned out to be insufficient to sustain the ever growing demand for energy and power-efficient computation without compromising performance. Therefore, designers were pushed to drift from homogeneous architectures towards more complex heterogeneous systems that employ the large number of available transistors to incorporate a combination of customized energy-efficient accelerators, along with the general-purpose processor cores. Meanwhile, enhancements in manufacturing processes allowed designers to move a variety of peripheral components and analog devices into the chip. This paradigm shift defined the concept of *system-on-chip* (SoC) as a single-chip design that integrates several heterogeneous components. The rise of SoCs corresponds to a rapid decrease of the opportunity cost for integrating accelerators. In fact, on one hand, employing more transistors for powerful cores is not feasible anymore, because transistors cannot be active all at once within reasonable power budgets. On the other hand, increasing the number of homogeneous cores incurs more and more diminishing returns. The availability of cost effective silicon area for specialized hardware creates an opportunity to enter the market of semiconductors for new small players: engineers from several different scientific areas can develop competitive algorithms suitable for acceleration for domain-specific applications, such as multimedia systems, self-driving vehicles, robotics, and more. However, turning these algorithms into SoC components, referred to as *intellectual property*, still requires expert hardware designers who are typically not familiar with the specific domain of the target application. Furthermore, heterogeneity makes SoC design and programming much more difficult, especially because of the challenges

of the integration process. This is a fine art in the hands of few expert engineers who understand system-level trade-offs, know how to design good hardware, how to handle memory and power management, how to shape and balance the traffic over an interconnect, and are able to deal with many different hardware-software interfaces. Designers need solutions enabling them to build scalable and heterogeneous SoCs. My thesis is that *the key to scalable SoC designs is a regular and flexible architecture that hides the complexity of heterogeneous integration from designers, while helping them focus on the important aspects of domain-specific applications through a companion system-level design methodology.* I open a path towards this goal by proposing an architecture that mitigates heterogeneity with regularity and addresses the challenges of heterogeneous component integration by implementing a set of *platform services*. These are hardware and software interfaces that from a system-level viewpoint give the illusion of working with a homogeneous SoC, thus making it easier to reuse accelerators and port applications across different designs, each with its own target workload and cost-performance trade-off point. A companion system-level design methodology exploits the regularity of the architecture to guide designers in implementing their intellectual property and enables an extensive design-space exploration across multiple levels of abstraction.

Throughout the dissertation, I present a fully automated flow to deploy heterogeneous SoCs on single or multiple field-programmable-gate-array devices. The flow provides non-expert designers with a set of knobs for tuning system-level features based on the given mix of accelerators that they have integrated. Many contributions of my dissertation have already influenced other research projects as well as the content of an advanced course for graduate and senior undergraduate students, which aims to form a new generation of system-level designers. These new professionals need not to be circuit or register-transfer level design experts, and not even gurus of operating systems. Instead, they are trained to design efficient intellectual property by considering system-level trade-offs, while the architecture and the methodology that I describe in this dissertation empower them to integrate their components into an SoC.

Finally, with the open-source release of the entire infrastructure, including the SoC-deployment flow and the software stack, I hope I will be able to inspire other research groups and help them implement ideas that further reduce the cost and design-time of future heterogeneous systems.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Luca Carloni. He has been supporting my Ph.D with endless enthusiasm, brilliant ideas, and continuous dedication to research. He has been a real mentor and I am honored to be able to say that I am one of his alumni.

I also want to thank all of my outstanding colleagues and friends, who shared with me the dynamism of our Computer Science Department. The endless technical discussions, as well as the not-so-technical conversations, during the past five years have been source of inspiration, relief from the pressure of the deadlines and an incredible learning experience.

Special thanks to Emilio, who led me through the intricacy and the beauty of the "art" of scripting and taught me most things that I know about the Linux operating system. To Giuseppe, who discussed with me about so many topics that I can't even enumerate. I am really glad and thankful for having him in our team. To Christian, who has been sharing with me some of the trouble of making things work for real. To YoungHoon, who always had an answer to my questions and the kindness to explain it to me.

I can't thank enough my loving parents, who supported me in any possible way and never doubted my ability to succeed in completing this Ph.D. If I made it, that is also because of you.

To my joyful son Gabriel I want to say that he changed my life like no Ph.D program could ever do. Since he arrived, almost one year ago, I knew that for him I would have done every possible thing and now I am delighted to have him share with me and my wife this achievement.

Most importantly, though, my deepest gratitude goes to my wonderful wife. She followed me far from home and she endured with endless patience and love the long weeks and weekends of work, when I was just a Ph.D student with no time for anything else. She has always fueled my motivation and her great effort to the success of this experience was as crucial as my commitment. I will never be able to thank you enough Sara, but I will try my entire life to make it up to you!

Dedicated to my wife Sara.

Chapter 1

Introduction

Over the last decade the semiconductor industry has faced the crisis of constant-power-density technology scaling, which has made energy dissipation become the limiting factor for the performance of microprocessors. Designers started replacing single-core designs with multi-core and many-core architectures to exploit parallelism and recover the necessary speedup across product generations. As geometry scaling has progressed towards most advanced technology nodes, the ratio between the number of transistors on a chip and the number of transistors that can switch simultaneously has been decreasing, to the point that entire functional blocks must be intermittently powered off to prevent overheating. As a result, the benefits of using transistors to implement specialized hardware, a.k.a. *accelerators*, have rapidly increased, leading current state-the-art designs to integrate several distinct fixed-function accelerators next to general-purpose multi-core clusters.

Thanks to this paradigm shift, the *system-on-chip* (SoC), which stands for a single-chip design integrating many heterogeneous components, has emerged as the most important computing platform across many application domains from embedded systems to data centers [Mair *et al.*, 2015; Park *et al.*, 2013; Pyo *et al.*, 2015]. This evolution opens the market of *intellectual property* (IP) for domain-specific applications to new small players, who have in-depth knowledge about specific scientific areas and are capable of developing efficient algorithms amenable to acceleration. The know-how of SoC integration, however, is still privilege of a few experts and big semiconductor companies with a legacy of years of development in hardware design. IP developers may not be willing to share their algorithms with SoC designers. At the same time, however, the non-recurring engineering costs for building the first prototype of an SoC are now estimated to be on the order of

tens of million dollars [Goering, 2009]. This sets an entry point that is too high for small companies and startups. Furthermore, if not addressed, the SoC complexity, design time and cost are destined to rapidly increase in the future, together with the number of heterogeneous components that are getting integrated to chase efficiency and performance.

My thesis is that *the key to enable scalable SoC designs is a regular and flexible architecture that hides the complexity of heterogeneous integration from designers, while helping them focus on the important aspects of domain-specific applications through a companion system-level design methodology.*

In order to allow IP developers to manage the complexity of large heterogeneous systems, I propose a design platform consisting of a scalable architecture that balances heterogeneity and regularity, paired with a companion system-level design methodology. The architecture hides the complexity of integration by creating the illusion of a homogeneous system, thanks to IP encapsulation and a set of *platform services*. The latter are a combination of hardware components and software interfaces that relieve the designers from managing shared-system resources and creating complex and non-standard hardware-software interfaces. By imposing some regularity constraint and defining the interfaces to platform services, the architecture promotes IP reuse and guarantees software portability across different SoCs, each with its own target workload and cost-performance trade-off point. The methodology exploits the regularity of the architecture to let designers focus on implementing their IPs and enables an extensive design-space exploration across multiple levels of abstraction.

By leveraging modern field-programmable-gate-array (FPGA) devices, I present an automated SoC generation flow that is based on the proposed architecture and methodology. This allows IP developers to seamlessly integrate many different accelerators on a real system, run their application with the final software stack, perform full-system design-space exploration, and tune power management and memory-allocation policies. This infrastructure can be leveraged to prototype and optimize SoCs before committing for fabrication on silicon technologies, or even to deploy the final system on a target FPGA. Recent advances in the features and logic fabric of FPGAs, in fact, brought them to be considered as competitive alternatives to application-specific integrated circuits (ASICs), especially to reduce the risks and the cost of the final design. Meanwhile, heterogeneous platforms that combine general-purpose processor cores with FPGA components are

emerging across different application fields, from cloud computing to embedded computing. This is the case of Xilinx, for instance, who pioneered the coupling of FPGAs with embedded processors with the ZYNQ programmable SoCs [Crockett *et al.*, 2014; Kathail *et al.*, 2016]. Companies like Microsoft, with Project Catapult [Putnam *et al.*, 2014], are in the process of coupling an FPGA device to nearly every processor in their servers, thus offering faster and more efficient cloud services, while at the same time being able to update or change completely algorithms and applications at need [Metz, 2016]. The combination of FPGAs with traditional processors and graphics-processing units is enabling server farms to run new search engines, innovative graph algorithms and artificial intelligence applications based on deep-neural networks. Even further, Amazon is creating a revolutionary marketplace of FPGA-based IP blocks through their EC2 F1 Instances [Amazon, 2017]. Given the growing interest in reconfigurable logic for cloud services, Intel, the leading company in the market of microprocessors, launched the HARP program [Schmit and Huang, 2016], which couples an FPGA fabric with a server-class processor. Across two generations of the HARP program, this has evolved from board-level to in-package coupling and it is reasonable to predict that in the near future the FPGA fabric will be integrated on chip, thus creating a server-class alternative to the embedded ZYNQ SoC.

FPGAs bridge the gap between the too expensive ASIC and the necessary specialization required to achieve energy-efficient computation. Programming FPGAs, however, is also a complex task, much more difficult than programming software [Metz, 2016]. This complexity adds up to the challenges of integrating IP blocks on heterogeneous platforms. Arguably, my thesis applies to such platforms as well and the entire work described in the dissertation is a step towards empowering software engineers and algorithm developers to directly implement and build efficient computing systems.

1.1 The Challenges of System Integration

Improving design productivity by reducing the complexity of SoC integration is fundamental to enable future generations of computational platforms beyond the crisis of technology scaling [Horowitz, 2014]. Designers must be able to raise the level of abstraction above register-transfer level (RTL) and make *system-level design (SLD)* become a viable and mostly automated solution [Borkar, 2009;

Dally *et al.*, 2013; Sangiovanni-Vincentelli, 2007; ITRS-2.0, 2015]. Complexity arises also from the intertwined issues of managing *shared resources*, because traditional solutions are designed for homogeneous architectures and must be adapted to a new mix of computing elements. In this regard, my dissertation addresses key challenges of SoC integration that are related to these three resources: *memory* allocation, *power* control, and on-chip *communication* services.

Memory. SoCs for high-performance embedded applications integrate *high-throughput loosely-coupled* accelerators [Cota *et al.*, 2015], which implement complex application kernels. Further, they have a dedicated, highly-customized *private local memory (PLM)* and fetch data from DRAM through direct memory access (DMA). The PLM is the key to achieve high data-processing throughput. It consists of many independent static-random-access memory (SRAM) banks whose ports can sustain multiple concurrent accesses from both the highly-parallelized logic of the accelerator datapath and its DMA interface. Recent studies of many accelerators confirm the importance of the PLM, which occupies 40 to 90% of the accelerator area [Cota *et al.*, 2015; Lyons *et al.*, 2012]. Consequently, researchers have started investigating methods to share the PLM SRAMs across accelerators [Cong *et al.*, 2014; Lyons *et al.*, 2012; Pilato *et al.*, 2014b] and with the processor caches [Cota *et al.*, 2014; Fajardo *et al.*, 2011] to improve the utilization of the growing portion of silicon area that is dedicated to memory. Despite such trends, the data-set sizes of embedded applications are increasing at a higher rate than the capacity of on-chip memory. While caches have successfully closed this gap for general purpose processors, a traditional memory hierarchy is not sufficient to preserve the performance and efficiency of accelerators when operating on a large amount of data.

Power. Power management in multi-core chips consists mainly in power and clock gating of idle components, coupled with dynamic voltage-frequency scaling (DVFS), which is a consolidated technique to adjust the operating supply voltage point of a circuit to its changing workloads [Isci *et al.*, 2006; Macken *et al.*, 1990; Semeraro *et al.*, 2002; Simunic *et al.*, 2001]. As the count of processing elements grows, increasing the number of these voltage-frequency domains helps designers meet performance and power targets by enabling a more fine-grained application of DVFS. This trend is visible across many classes of integrated circuits, from server processors to SoCs for embedded applications. According to a global survey performed by Synopsys among its costumers in 2011, about 30% of the respondents used more than ten clock domains and between four and ten voltage domains in their designs [White, 2012]. Such trend is supported by the emerging class of *integrated voltage*

regulators (IVRs) [Andersen *et al.*, 2015; Burton *et al.*, 2014; Lee *et al.*, 2015; Sturcken *et al.*, 2013; Tien *et al.*, 2015], which are expected to improve the performance of currently-available on-board regulators by enabling tens of independent voltage-frequency domains on the same chip and by reducing the time to switch between two operating points from microseconds to nanoseconds. Traditional power management implemented at the operating system level is too slow to exploit the capabilities of IVRs. Furthermore, the behavior of different accelerators may vary and is unlikely to match the activity profile of a general-purpose processor.

Communication. The challenges of integration, memory management and resource sharing cannot be solved without considering the physical interconnect and the communication services that the SoC must offer to the integrated components. These affect the scalability of a system at all levels, from system level to physical design. In order to be able to integrate profoundly different components, the system must implement a set of mechanisms to support communication and control of the shared resources in a practical and flexible way. The main challenge is the need of interfacing with a variety of non compatible standards, such as bus or IP-specific protocols, and optimize communication for many different data access patterns, while satisfying all timing requirements.

1.2 Research Contributions

Guiding system-level designers to build future SoCs requires to streamline an effective methodology supported by a flexible architecture. The methodology must address all of the challenges of the integration process and hide such complexity from the designers. The goal of building a scalable design platform for SLD is progressively approached in the dissertation through the contributions listed in this section.

The first contribution that I present consists in defining *embedded scalable platforms* (ESP) which is a design methodology for SLD combined with a flexible and regular architecture. I then tackle the three main integration challenges, listed in the previous section, by implementing the ESP *platform services* that manage memory, power and communication and define the necessary hardware and software interfaces. The last contribution is *Open-ESP*, the SoC deployment flow for FPGA, which embeds all the techniques and design solutions described in my dissertation. This infrastructure is a concrete step towards enabling a scalable design for future SoCs.

ESP. With *embedded scalable platforms* I refer to the combination of an SoC architecture and a companion design methodology that together aim to simplify the design, integration, and programming of the heterogeneous components in a complex system. The architecture is scalable because it relies on a regular tile-based structure and its interconnect consists of a multi-plane network-on-chip (NoC). A set of interfaces, wrappers and proxies implements sockets for IP integration and platform services that mask the complexity of resource management, while providing access to the interconnect. The ESP companion design methodology decouples system integration from accelerators design, raises the level of abstraction and guides designers in the application of high-level synthesis tools. High-level synthesis enables a more efficient design of accelerators with a focus on their algorithmic properties, a broader exploration of their design space, and a more productive reuse across many different SoC projects. The ability to quickly integrate different sets of accelerators and to generate SoC instances, with any mix of Pareto-optimal implementations for each component, makes application and system-level design space exploration a practical process, which happens independently from the IP design [Mantovani *et al.*, 2016b].

Accelerators for large data sets. I define the large data set problem for accelerators as the problem of a widening gap between the size of on-chip dedicated storage and the memory footprint of the application to accelerate. While memory hierarchy has long solved a similar problem for general-purpose programmable cores, the case of accelerators requires alternative techniques to avoid performance and efficiency degradation. I propose a hardware-software solution to preserve the accelerator speedup when scaling from small to large data sets, without giving up the use of shared memory across all processing elements. The design is transparent to user applications, thus it doesn't increase the complexity of programming and it is generally applicable to any SoC hosting high-throughput accelerators [Mantovani *et al.*, 2016c].

Fine-grained DVFS. Fine-grain power tuning is an ESP platform service that combines the capability to emulate frequency scaling with accurate power models and a distributed probing mechanism [Mantovani *et al.*, 2016a]. Designers can run full-system workload scenarios and let the system collect statistics about power dissipation, traffic over the interconnect and activity of hardware accelerators. With this platform service I enable an incremental design-space exploration of power management settings to achieve near-optimal energy efficiency. The synergistic interaction of a fast

hardware controller and a software daemon implements power-management policies that take into account both local conditions and system-level metrics. In addition, policies can be incrementally adapted at run-time simply by tuning the behavior of the software daemon.

Efficient communication. There exist three main approaches to support shared data among processing elements. The first method is to have private memories be coherent with the system memory hierarchy. The opposite approach is to maintain a shared buffer, pinned to DRAM, without involving the cache hierarchy. The third method consists in having explicit message passing among components, which potentially reduces memory accesses. The optimal solution may depend on the access patterns and the size of data sets of the given accelerators. Furthermore, when multiple accelerators are arranged in a coarse-grain pipeline to speed up a particular application, the size of the data-tokens they exchange has also a direct impact on which communication mechanism should be chosen. Therefore, I support both message-passing and through-memory communication as platform services that can be specialized for the given accelerators. The interconnect plays a fundamental role in guaranteeing throughput and scalability. Hence, I designed the architecture of ESP with a multi-synchronous approach, where each tile can have an independent source clock. To prevent performance penalty in communicating across tiles, however, the interconnect is based on a fully-synchronous NoC for up to a dozen of integrated IP blocks. For larger systems I choose, instead, a hierarchical approach, based on asynchronous high-throughput bridges, which are placed strategically during the floor-planning of the IP blocks.

Open-ESP. The inherent complexity of SoCs and the growing size of workloads demand accurate, yet fast, analysis tools. Using FPGA prototypes is therefore necessary to avoid simulation shortcomings in terms of run time and accuracy [Arvind, 2014]. Indeed, FPGA-based prototyping is a standard practice in the industry to run full-system test scenarios. Furthermore, FPGA technology has been increasingly adopted in data centers, where reconfigurable logic offers lower cost acceleration for dynamically evolving workloads [Putnam *et al.*, 2014; Schmit and Huang, 2016; Amazon, 2017]. Therefore, I created Open-ESP, an FPGA-based development platform that allows designers to leverage the flexibility and scalability of the ESP architecture and design methodology. Thanks to my work, students at Columbia University were able to integrate dozens of accelerators with an embedded processor, two memory controllers and several I/O peripherals on

state-of-the art FPGA-development boards. In a few simple steps, designers can integrate their IPs and investigate most of the trade-offs in developing new SoCs by deploying instances of Open-ESP on single or multiple FPGAs. For example, designers can decide which application kernels are worth being accelerated, test different Pareto-optimal implementation of each accelerator, tune power management, and setup different communication mechanisms across processors and accelerators. Through Open-ESP, design-space exploration is carried out at all levels of abstraction: from single-component to full-system, taking into account all complex hardware-software interactions, including non deterministic behaviors of external memory and of the operating system [Mantovani *et al.*, 2016a].

1.2.1 Structure of the Dissertation

The first part of the dissertation is dedicated to the presentation of background information, including major motivating factors and related work. **Chapter 2** dives briefly into the history of microprocessors and analyzes their performance trends. In particular, Section 2.1 focuses on the breaking point that was marked by the end of Dennard scaling. Next, Sections 2.2 and 2.3 discuss the advent of multi-core and general-purpose graphics processing units, followed by the rise of heterogeneous systems enhanced by specialized hardware accelerators. The concept of accelerator is not well defined in the literature. Therefore, Section 2.4 discusses some of the models that have been adopted for the design of accelerators and to estimate their efficiency. The last segment of this section talks about accelerator-processor coupling and shows how this is perhaps the most distinctive characteristic for accelerators, which can help designer classify them and choose which model is suitable for a specific target application.

Chapter 3 describes state-of-art tools and techniques that have been proposed to raise the level of abstraction in the context of hardware design. In particular, Section 3.1 presents the flow of high-level-synthesis and an overview of the high-level languages that have been invented to support it. Section 3.2 presents a few interesting frameworks for SLD from both industry and academy, each using some flavor of high-level synthesis as a starting point.

The central part of my dissertation is dedicated to ESP and dives into the details of the research contributions, while discussing the ESP architecture and methodology. **Chapter 4** presents the full design methodology and gives an overview of the underlying infrastructure that supports the SoC

generation flow and design-space exploration (DSE). Specifically, Section 4.1 defines the model of an ESP hardware accelerator and shows the methodology for component-level DSE, based on high-level synthesis, through a real-application test case. Section 4.2 describes the ESP socketed architecture and the basic set of platform services that are necessary for the integration of IPs. Finally, Section 4.3 discusses the ESP software stack, which provides designers with the ability to transparently interact with the accelerators, without the need to implement low-level hardware-software interface protocols.

Chapter 5 focuses on the memory-access platform service and explains how ESP closes the gap in the domain disparity that exists between processors and accelerators. Section 5.1 defines the large data set problem and presents a motivating example that highlights the importance of addressing it to preserve the speedup of accelerators. Section 5.2 illustrates how ESP solves the large data set problem, without sacrificing ease of programming the accelerated applications. The experimental results, presented in Section 5.3, report the FPGA-based evaluation of complex test scenarios involving multiple concurrent accelerators.

Chapter 6 discusses power management with fine-grain DVFS. The premise, stated in Section 6.1, is that integrated-voltage regulators are available and their quality is improving to the point that on-chip voltage regulation is feasible, efficient and capable of adapting faster to the system workload, when compared to traditional off-chip regulators. Section 6.2 describes the design of the power controller and the combination of hardware and software policies for power management that are implemented in ESP as a platform service. In addition, this section explains how fine-grained DVFS is emulated on FPGA for accurate power estimation of concurrent accelerator runs. Experimental results are reported in Section 6.3, with three full-system test scenarios.

Chapter 7 deals with communication services, which make ESP a scalable design. Specifically, Section 7.1 describes the benefits of creating pipelines of accelerators with point-to-point communication, while Section 7.2 explains the details of the ESP interconnect and how IPs can communicate across the network-on-chip. These communication services enable complete portability of software written for a traditional bus-based system. Finally, Section 7.3 describes how ESP can scale the size of the network by leveraging a hierarchical approach and asynchronous communication, even across multiple chips or FPGAs.

The last part of the dissertation includes the conclusive chapter and the Appendix. Section 8.1 in **Chapter 8** lists some relevant projects that have spawned from this research, as well as future work that I envision being integrated in Open-ESP. Section 8.2, describes the course *System-on-Chip Platforms*, which has been influenced by my research work. Finally, Section 8.3 concludes the dissertation by summarizing its main contributions.

Appendix A describes the structure of Open-ESP and serves as a documentation to use the open-source infrastructure, whereas **Appendix B** explains how ESP can be exploited to generate processor-less architectures on FPGAs that are connected to a host system via standard protocols.

Part I

Background

Chapter 2

The Rise of SoCs

The word *system-on-chip* (SoC) is used to refer to most modern large-scale integrated systems. Nowadays, engineers involved in the semiconductor industry and in digital design tend to refer even to processors using the acronym SoC. Despite its popularity, the concept of SoC is the result of a fairly recent technology evolution and sits at the basis of the research presented in this dissertation. Therefore, this chapter goes through the crucial steps of technology evolution that led to the rise of SoC as a computing platform. This path starts from the crisis of technology scaling that was sustaining the skyrocketing improvements of microprocessors. Then, it evolves with an overview of the computing platforms that appeared as disruptive solutions to overcome such crisis. These platforms include multi-core processor architectures and general-purpose graphics processing units, as well as a whole variety of specialized hardware accelerators.

2.1 The Crisis of Technology Scaling

Looking back at the history of semiconductors and digital design means to observe the evolution of processors. Driven by the popular Moore's law stated in 1965, the semiconductor industry has been able to grow the number of devices integrated on a single processor-chip approximately by a factor of 2×, at fixed cost, every eighteen to twenty four months [Moore, 1965]. Fig. 2.1 captures the effects of such empirical law, which brought processors to be the powerful and hidden motor of everyday-life objects. Even if the evolution of transistors did not always show a perfect match with Moore's law, the trend reported in Fig. 2.1 has largely followed the 1965 prediction until very

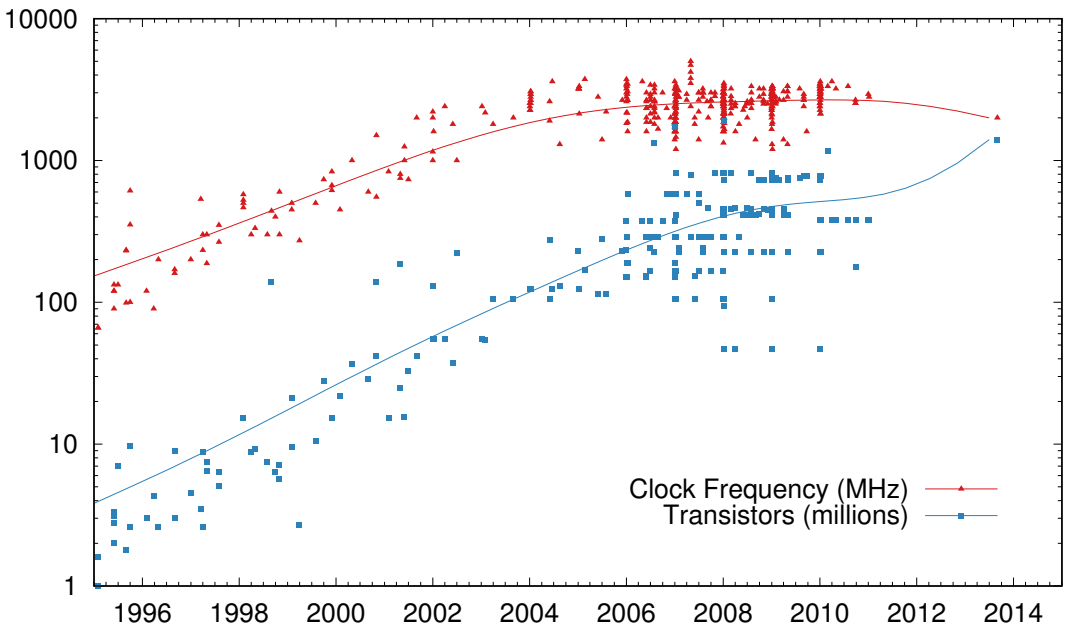


Figure 2.1: Evolution of microprocessors in terms of transistor count and nominal clock frequency (data from [Danowitz *et al.*, 2012]).

recently. The chart reports information about a database of microprocessors designed over almost two decades [Danowitz *et al.*, 2012]. The blue curve shows the count of transistors integrated on a single chip in millions. On a logarithmic scale the growth of the curve is almost perfectly linear, which means that designers could integrate an exponentially increasing number of transistors, enabling the realization of more and more functionality in each subsequent processor generation. Admittedly, the cost of doubling the number of transistors has not remained constant over time. But, thanks to commodity electronics, the growing user base of integrated circuits allowed the semiconductor industry to cope well with the increased manufacturing costs. Furthermore, the operating frequency of transistors kept increasing as a consequence of the reduction in their feature size. Hence, designers could fit more functionality in a single clock cycle. Alternatively, they could increase the clock frequency of the entire chip. The evolution over time of the latter is represented by the red curve in Fig. 2.1, which reports a steady growth of the clock frequency until the year 2005.

The correlation between Moore's law and performance improvements can be explained with

another popular observation, made a few years later by Robert Dennard. By analyzing the relation among the size of transistors, their capacitance and the resistance they pose to the current flow, he devised a law known as "Dennard scaling" [Dennard *et al.*, 1974]. The starting point is the equation of the dynamic power dissipated by a switching transistor in any CMOS logic gate [Ahrons *et al.*, 1965], which is the primitive building block of any digital circuit.

$$P_{dyn} = \alpha \times C \times F \times VDD^2 \tag{2.1}$$

Dynamic power is proportional to the capacitance C of the transistor, the square of the supply voltage VDD and the product between clock frequency F and switching activity α. The latter represents the probability of the transistor to change its state from non conductive (logic zero), to fully conductive, or saturated (logic one). Such probability, in general, depends on the circuit and on the stimuli applied to it. As the transistor geometry shrunk, from one generation to the next one, their capacitance used to decrease linearly. Furthermore, manufacturers were able to reduce VDD as well, thanks to the improved conductivity of smaller transistors. By combining this with Equation 2.1, Dennard predicted that every new technology node would have allowed to integrate more transistors, operating faster, while maintaining constant the power (and heat) dissipated per unit of area [Dennard *et al.*, 1974].

Interestingly, Fig. 2.2 shows that while the die size for microprocessors has not changed significantly in the past twenty years, the average power has increased over the same period of time. This trend shows how designers took advantage of Dennard scaling to improve processors' performance even beyond constant-power scaling. Through the early 2000s, the general approach in designing processors was to make every transistor count towards improving the performance of a single stream of executed instructions, a.k.a. single-threaded performance. As long as scaling permitted, in fact, engineers followed a "90/10 optimization" principle [Borkar and Chien, 2011]. This means that all transistors are active at least 90% of the time. The result was the steady and fast improvement of processors' performance that semiconductor industries used to deliver with each new technology node. Dennard's model, however, ignores leakage power and the threshold voltage of transistors. The former was a negligible term, with respect to dynamic power, in old technology nodes. With geometry scaling, however, more and more current is leaked by transistors when they are inactive, thus static power has become predominant in sub-micron technologies. Furthermore, threshold voltage, which is the minimum voltage to make a transistor conductive, imposes a limit to the scaling

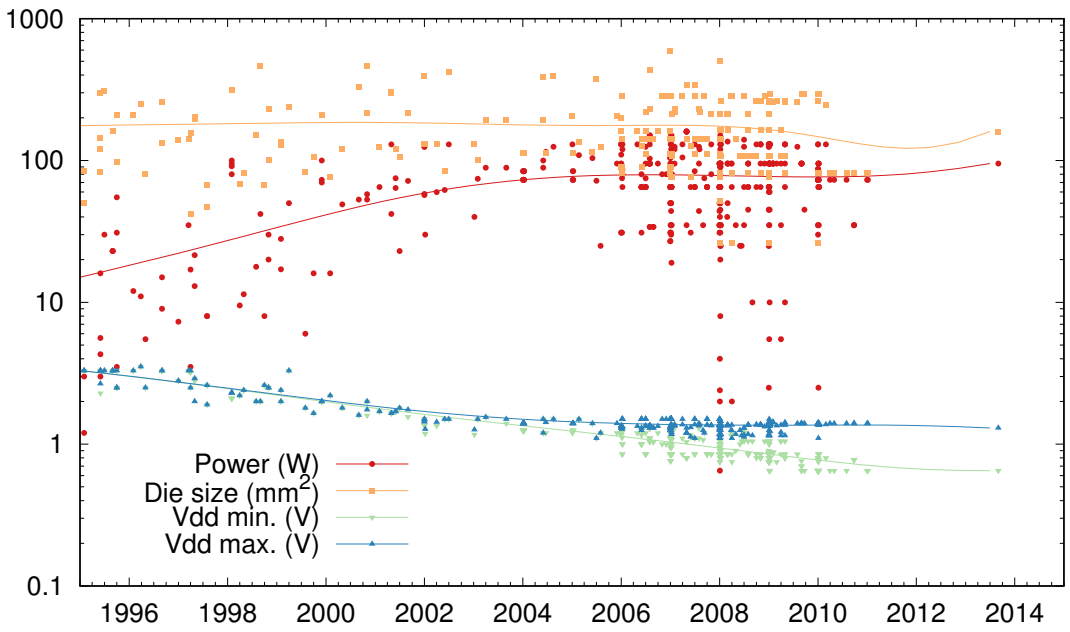


Figure 2.2: Evolution of microprocessors in terms of power, total die area and supply voltage (data from [Danowitz *et al.*, 2012]).

of VDD. The closer VDD is to the threshold voltage, the smaller the available voltage sweep to fully activate or de-activate a transistor. Beside reducing the robustness of the circuit, reducing the supply voltage VDD at a value close to the threshold voltage may lead transistors to operate in a transitioning region, referred to as "linear region", which incurs even higher power dissipation. Despite recent efforts in developing techniques for near-threshold-voltage operation [Kaul *et al.*, 2012; Kim and Seok, 2015; De *et al.*, 2017], leakage and threshold voltage impose a lower bound to the transistor power that does not scale with geometry.

The sudden trend shift in the curves of Fig. 2.1 and Fig. 2.2 are the natural consequence of the end of Dennard scaling. Power and voltage are no longer scaling with the size of transistors and even if more transistors can still be integrated per unit of area, they cannot operate faster, nor they can be switching all at the same time. At constant frequency and voltage, in fact, more transistors per unit of area imply a higher power density. When the resulting heat cannot be appropriately dissipated, the only way to prevent damage to the circuit is to turn off part of it. This phenomenon, which is referred to as *dark silicon* [Esmaeilzadeh *et al.*, 2011], led first to the advent of multi-core architectures and

then to a paradigm shift from a "90/10" to a "10×10 optimization approach" [Borkar and Chien, 2011]. This means that instead of having most transistors active at the same time, designers should now consider to have, for instance, 10% of them switching at any given time. Depending on the workload, however, a different 10% of the design shall be activated.

The introduction of multi-core architectures has been the first reaction to the crisis of technology: by reducing the frequency of each core, in fact, two or more of them can run parallel workloads without exceeding the system power budget. Unfortunately, not all workloads can run in parallel on multi-core processors. Hence, designers were pushed to look for alternative ways of improving performance within a limited power envelope. As a result, some form of specialization appeared first in embedded systems with the combined integration of microprocessors and of digital-signal processors on the same chip, together with a few accelerators. Furthermore, specialized multi-core devices were developed in the context of applications for graphics. These devices, known as graphics processing units (GPUs), brought the concept of multi-core architecture to the extreme, by integrating hundreds, or even thousands, of small simple cores optimized for performing multiply-and-accumulate types of operations in parallel on matrices of pixels. The advent of mobile devices, however, boosted even more the ever growing demand for energy efficiency, to the point where the latter replaced performance as the main objective of system optimization. Consequently, aggressive hardware specialization was the only possible solution to further sustain the evolution of new computing platforms [Horowitz, 2014]. Therefore, custom accelerators, consisting of hardware dedicated to the target applications, were designed and integrated on chip to achieve two to three orders of magnitude more efficient computation [Hameed *et al.*, 2010]. In the past, the cost of accelerators, measured in silicon area, was not affordable because they can only speedup specific parts of a workload. With the advent of *dark silicon*, however, accelerators became more appealing, as each of them can perform a specific task very efficiently and then turn off, thus letting other portions of the chip use the available power budget.

In conclusion, the crisis of technology scaling favored the adoption of new design paradigms, but none of these is capable of covering the entire spectrum of requirements in terms of functionality, performance and efficiency. As a result, modern computing platforms typically combine several different components that operate in a synergistic way to achieve higher energy efficiency. These complex and mainly heterogeneous systems are named SoCs.

2.2 Facing the Crisis with Multi-Core Architectures

The failure of Dennard scaling started showing its symptoms around 2005. Processor chips hit the limit of power density and improvements of single-threaded performance began to slowdown. Therefore, semiconductor companies were forced to find a new driver that could justify Moore's law and the economics of technology scaling. Designers started implementing processors running at lower frequencies and with less power-hungry pipelines. The additional transistors, provided by Moore's law, were used to integrate multiple copies of these more efficient cores. Hence, performance could continue to improve by exploiting workloads' parallelism [Esmaeilzadeh *et al.*, 2011]. Each core can sustain one or more thread of execution so that the aggregate number of instructions committed per cycle can grow with the number of available cores, which is proportional to the maximum number of concurrent threads that can be executed in parallel. In practice, limited parallelism of the workload and system-level factors, such as memory and interconnect bandwidth, contribute to setting an upper-bound to the performance improvements obtained with multi-core architectures. Such upper-bound is computed by applying Amdahl's law [Hennessy and Patterson, 2011] and its variation for parallel programs on multi-core architectures [Hill and Marty, 2008]. Nevertheless, multi-core architectures have been the first response to the crisis of technology scaling and it is interesting to analyze their impact on processors' performance. This section reports published results about performance trends over time and shows how multi-core architectures recently affected such results by supporting more concurrent threads of execution, with respect to single-core processors.

Single-threaded benchmarks. Fig. 2.3 plots the published results about running SPEC benchmarks [SPEC, 2017] on different processor architectures. These were released over a time-range that spans from 1995 to present days. Each point in the chart is the normalized benchmark result for a particular system. The types of processor range from mobile and desktop computers to data-center servers. Hence, the number of supported threads can vary significantly. The color of each point corresponds to the number of available threads, according to the gradient palette shown on the right. Note that the benchmarks SPEC int® and SPEC FP® were developed to measure single-threaded performance and, therefore, increasing the number of cores and threads does not return a performance improvement in these cases. This is clearly visible on the top chart, which shows results for the integer benchmarks: after the years 2005 and 2006, the number of systems support-

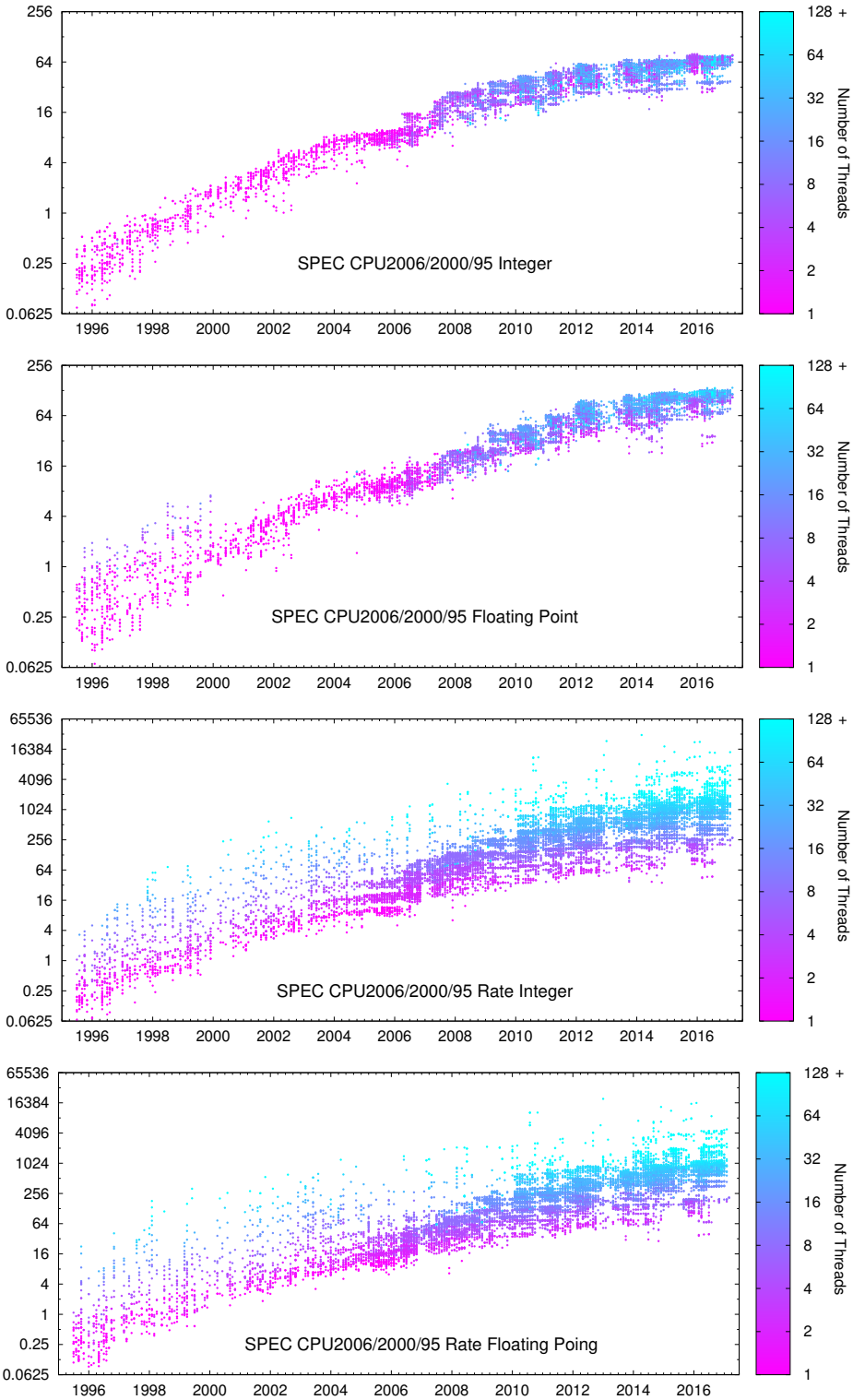


Figure 2.3: Performance and computational capacity (rate) of microprocessors over time for single-threaded benchmarks (data based on SPECint® and SPECfp® [SPEC, 2017]).

ing the execution of more concurrent threads drastically increases; at the same time, the results for multi-core processors, which naturally support more threads of execution, lie in the same range of single-core ones. Moreover, the top chart is an evidence of the reduced rate at which integer performance improves for single-threaded applications, starting around the time when more public results on multi-core processors become available. A similar trend can be seen in the second top chart in regards to floating-point performance, except that the improvement rate seems to slow down later in time with respect to the chart for integer performance. This fact is probably an indicator that floating-point pipelines were not as optimized as the integer units; hence, architectural improvements could be exploited to partially recover the speedup that technology scaling wasn't providing any more. Interestingly, multi-threading support on most recent systems yields small, but noticeable benefits. The chart for SPEC floating-point, in fact, shows that machines with more threads (blue to light-blue points), in the period between 2013 and 2017, have higher scores than machines with one, or few, threads. Most likely, this is due to the capability of compilers to automatically parallelize code, whenever data dependencies allow them to do so. Given that floating-point instructions have typically longer latency, the code is likely to offer more opportunities to exploit instruction-level parallelism (ILP) [Hennessy and Patterson, 2011].

Single-threaded benchmarks replicated. The two charts at the bottom of Fig. 2.3 show the published results from the "Rate" version of the SPEC benchmarks. In this case, multiple copies of the same benchmark are ran concurrently on the systems under test. The resulting score is a metric of the computational capacity of each processor, not of how well they handle multi-threaded applications. The concurrent processes run independently on the system, therefore Amdahl's law [Hennessy and Patterson, 2011] does not apply to the code running on the processors. Beside the processor computational capacity, the limiting factors are mainly memory and interconnect bandwidth. As a consequence, SPEC results are much better with respect to the top charts, for machines that allow multiple concurrent threads to be in flight. Still, performance capacity improvements, on average, started slowing down since 2006. By excluding super-computers, which are the few sparse points at the top of the curve, there is a narrowing gap between the score of many-core architectures and the score of processors with a smaller core count. On one hand, these results reflect the trend of single-threaded performance; on the other hand, they also indicate that system-level factors, such as memory, are now as relevant as the internal features of the processor or, perhaps, even more.

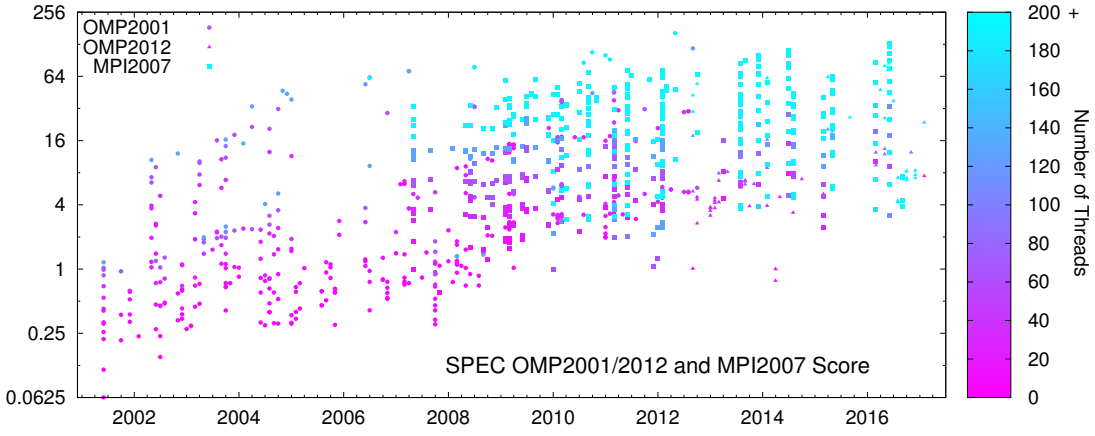


Figure 2.4: Performance of microprocessors over time for multi-threaded benchmarks(data based on SPEC OMP® and SPEC MPI® [SPEC, 2017]).

Multi-threaded benchmarks. Even though multi-core architectures can process faster and more efficiently a set of concurrent but independent tasks, many interesting workloads consist of a single application. Hence, the real challenge is to speedup the execution of one application by exploiting its potential parallelism with multiple threads of execution. In order to help programmers who are used to a single-threaded programming model write programs for multi-threaded execution, software engineers have developed sets of primitives, libraries and compiler macros, also known as application-programming interface (API). An API hides the complexity of handling operations such as thread spawning and synchronization on shared variables, or critical code regions, which should be executed in sequence. Among the most popular APIs for multi-threading programs there are: `Open Multi-Processing` (OMP), which relies on shared memory for communication among threads, and `Message-Passing Interface` (MPI), which implements an explicit communication mechanism across cores. Depending on the target application, the particular processor and the memory hierarchy, one of the two approaches may have better performance or reduce the overhead to maintain coherence across the processor caches and satisfy the memory consistency requirements [Hennessy and Patterson, 2011]. The chart in Fig. 2.4 reports the results of executing multi-threaded benchmarks from SPEC on a set of processors released from 2001 to present days. These benchmarks are based on OMP (points represented with circles and

triangles), and MPI (points represented with squares). The reported results for the OMP benchmarks have been scaled with respect to the results of the MPI benchmarks, so that they can be compared on a single chart. The color of the points indicates the number of threads used by the benchmark. Even though this is related to the number of available cores and threads on the underlying system, it is possible that some architectures with fewer cores are capable of exploiting more parallelism on a single application and spawn more threads than architectures with higher core counts machines. Furthermore, using more threads doesn't necessarily lead to an overall better performance. Unless the application is embarrassingly parallel, in fact, the more threads are used, the more overhead the system incurs for cache coherence (in the case of OMP) or explicit communication and synchronization among cores (in the case of MPI). Each additional thread obtains diminishing returns, following Amdahl's law, up to the point when the overhead increases more than the speedup of the parallel code-regions. This explains why systems using fewer threads (purple and violet points) are able in some cases to compete with systems that allow a higher thread count (light-blue points). Furthermore, this reinforces the thesis that multi-core architectures alone cannot sustain the desired performance scaling in the future [Esmaeilzadeh *et al.*, 2011; Taylor, 2012].

2.3 GPUs for Computation

Next to traditional processors, computing systems used to integrate at board level video-graphics arrays (VGAs) which were effectively used as accelerators for two-dimensional user interface. During the 90s, the first tri-dimensional graphics accelerators enabled the era of 3D gaming and applications. In order to support the ever growing complexity of image processing and rendering required for gaming, these accelerators progressively evolved to become first micro-programmed processors and then fully-programmable systems, which are called graphics processing units (GPUs). Rendering high-definition graphics scenes is a problem with tremendous inherent parallelism. In most cases a program written to render one pixel has to be replicated for the entire frame and then for several frames per second to support, for instance, interactive gaming [Nickolls and Dally, 2010]. Such parallelism led GPUs to evolve as massively parallel architectures, integrating on a single chip a growing number of small cores. The pipelines of these cores are optimized for a few operations

typical of graphics computation, such as multiply-and-accumulate. Moreover, they have limited control logic, because in most cases they simply need to process a stream of input pixels, in the form of vectors or matrices.

The massive parallelism of the GPUs, in turn, pushed software engineers to create programming languages and models that transparently express the necessary parallelism required for efficient graphics computation. These models enabled programmers to migrate intensively parallel tasks, not strictly related to image rendering, onto the GPU cores. An example of these tasks are the simulations needed to compute the position of moving objects in a video game. As a result, GPUs quickly evolved from specialized devices to more general-purpose architectures. The introduction of the CUDA computing platform [NVIDIA, 2006] and the corresponding programming model in C++ from NVIDIA put GPUs in competition with general-purpose cores, as every programmer could leverage them as accelerators for massively parallel workloads. Alongside CUDA, other programming frameworks, such as OpenCL [Group, 2009], were developed to provide programmers with an API for using general-purpose GPUs as computing systems, which in present days integrate thousands of cores. The flexibility of the programming framework allows a single program to scale in the number of spawn threads depending on the available underlying hardware. Hence, a new GPU with a higher core-count can be used without editing the source code.

When a program gets compiled for GPU, the compiler creates a hierarchy of threads that matches the hierarchy and the distribution of channels to external memory, local memory, caches, processor clusters, register files and cores. The two main advantages of GPUs are, in fact, very large memory bandwidth and fine-grained organization of local memory. This guarantees quick access to data, which is the enabler for the single-instruction multiple-threads (SIMT) architecture that GPUs rely on [Nickolls and Dally, 2010]. One of the most successful consequences of general-purpose GPUs is the quick flourishing of frameworks for machine learning applications [Abadi *et al.*, 2015; Theano, Development Team, 2016; Collobert *et al.*, 2011; Jia *et al.*, 2014]. These tasks apply specific functions to process huge sets of data and in many cases perform the same instructions over and over for multiple inputs. Such tasks map well on vector-processors with multiple pipeline lanes, also referred to as single-instruction-multiple-data (SIMD) [Hennessy and Patterson, 2011], and even better to the SIMT architecture of GPUs, which bring the number of parallel threads to the extreme for a given technology node.

Even though high-memory bandwidth is key to the performance improvements, the need of having dedicated DDR-memory nodes is also a major drawback of discrete GPUs. This implies that data must be copied from the system main memory to the GPU dedicated one. Despite the high speed of modern serial buses, such as PCI Express, the overhead of data transfers affects the overall system performance and power dissipation [1]. The success of GPUs in providing more efficient high-performance computation for parallel workloads culminated with a major innovation in the design of processor chips: the integration of a GPU and a multi-core system on the same die, for which Intel's *Sandy Bridge* [Yuffe *et al.*, 2011] is a paradigmatic example. By having the GPU sitting next to processor cores, there is no need to transfer data from the system memory to a dedicated DDR. Since I/O and memory accesses dissipate hundreds of times more power than moving data on-chip, this approach guarantees noticeable power and energy savings at both chip and system-level. In terms of performance, however, studies have shown that discrete GPUs are still performing better than integrated ones [Daga *et al.*, 2011], because of higher memory bandwidth and larger number of threads supported, at the expenses of silicon area on a dedicated chip. Not surprisingly, the break down between active computation and communication overhead proves that, while integrated GPUs are slower at computation, discrete GPUs spend more time waiting for data transfers to complete over PCI Express, rather than computing.

Despite their success, and the large adoption in data centers, where more money can still buy enough power, GPUs are orders of magnitude below the energy-efficiency requirements expected for future systems. Specialization and heterogeneity, instead, hold the promise to fill the efficiency gap by coupling processors and GPUs with hardware accelerators [Horowitz, 2014; Taylor, 2012].

2.4 The World of Accelerators

SoCs were first introduced in the context of embedded systems, where the integration of DSPs and accelerators for multimedia applications was necessary to comply with the limited power envelopes. The popular OMAP open platform for multimedia applications [Song *et al.*, 2003] is the most representative example of such systems, that had to seek for energy efficiency much earlier than the corresponding processor chips for server-class applications. Nevertheless, modern general-purpose

[1] Note that in this case power dissipation refers to the power of the entire system, and not just the processor chip.

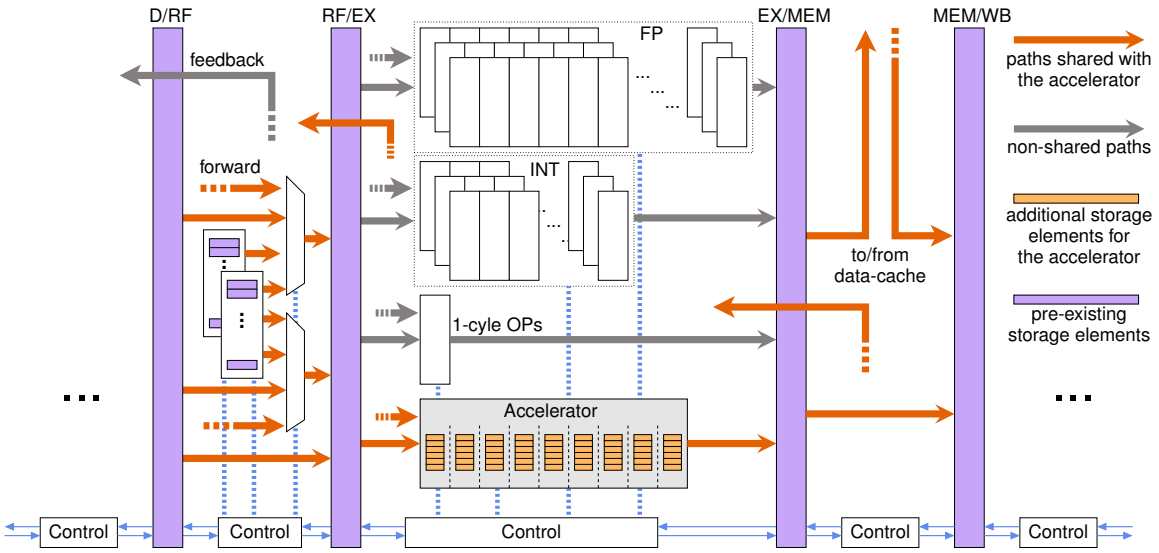


Figure 2.5: The most tightly-coupled accelerator model, integrated within the processor's pipeline and sharing the data-path with other functional units.

multi-core architectures have now evolved to systems-on-chip (SoCs), which integrate on a single-chip several heterogeneous components, including hardware accelerators. There is, however, no standard definition of an accelerator; in fact, a large variety of accelerator-based architectures has been proposed, each specifying a model of hardware accelerator that differs from others with respect to memory management, power control and coupling with the general-purpose processor cores.

2.4.1 Accelerator-Processor Coupling

The coupling of specialized hardware with the general-purpose cores of an SoC is perhaps the most distinguishing feature for an accelerator, as it impacts fundamental design choices. These choices include whether the accelerator's local storage is memory-mapped or transparent to the processor; how accelerators access external memory; at which granularity accelerators operate with respect to the data set and target application; whether accelerators are controlled in hardware, through software, or both; and whether the processor's instruction set needs to be extended.

ISA extensions. Starting from a processor-based design, it is natural to think about an accelerator in terms of instruction set architecture (ISA) extension. For instance, encryption extensions and vector

extensions provide instructions that are not part of the original ISA and are specific to the domains of cryptography and vector processing, respectively. This means that the pipeline of the processor is enhanced with specialized functional units that are responsible to execute the additional instructions. These specialized functional units are actually very tightly-coupled accelerators (TCAs) which share most of the processor's resources [Cota *et al.*, 2015]. Fig. 2.5 shows the model of such TCA, placed within a generic processor pipeline. The execution stage of the pipeline is composed of several functional units, including the TCAs. Each unit has a different number of issue ways and a different latency, corresponding to the computation stages of each block [Hennessy and Patterson, 2011]. Shared data-paths are highlighted in orange and include register-file access, immediate operands, forwarded data, read and write paths to the data cache. Shared, but pre-existing, registers are shaded in purple, while additional storage elements are shaded in light orange. Note that the pipeline control logic is also partially shared, because it is responsible to configure and activate the accelerator, based on the decoded instruction that is currently executed.

This strict coupling imposes several constraints on the accelerator's design. First, the area of the accelerator should be approximately in the same range as the area of the other functional units. Even though latency can be different, in fact, it is important to maintain some degree of regularity to reduce the complexity of placing and routing the processor. A consequence of this requirement is that static random-access memories (SRAM) are unlikely to be used as storage elements in the accelerator, because their physical constraints might negatively affect the resulting pipeline layout. Therefore, only a limited amount of storage elements, implemented as registers, should be used within the accelerator. To summarize, a TCA that is so closely integrated into the processor must be tailored to the characteristics of the pipeline, including its bit-width and throughput. Finally, the accelerator's designer must be allowed to extend the ISA with a special instruction for each TCA. The popular Tensilica extensible processor is an example of a commercial IP intended for ISA extension [Leibson, 2006], while the Rocket Chip Generator is an academic processor that allows accelerators to be integrated within the pipeline and has an open ISA, which can be freely extended [Asanović *et al.*, 2016]. In general, however, processors are extremely optimized IPs and vendors don't give access to any internal pipeline interface enabling the insertion of a functional unit. Similarly, since the ISA is typically fixed and proprietary, special instructions cannot be added to control the accelerator.

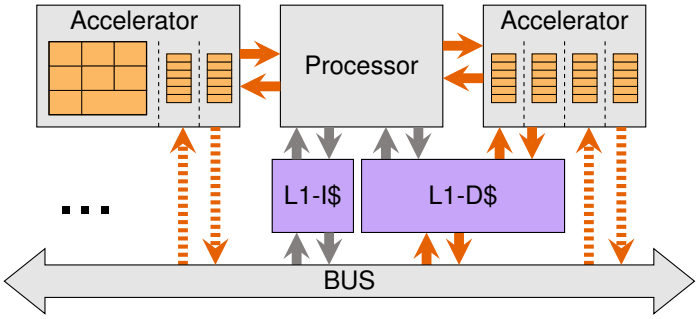


Figure 2.6: Tightly-coupled accelerator model, integrated as a co-processor with optional sharing of the level-1-data cache.

Co-processors. When the target accelerator performs a task significantly more complex than a single instruction and needs to operate on more data than a few operands, integration into the processor's pipeline is not a viable option. Instead, the accelerator must be implemented as a separate entity which can still share some processor's resources. This is the case of accelerators integrated as co-processors, which communicate with the general-purpose processor core through a dedicated interface. Fig. 2.6 shows two different flavors of co-processor integration with optional interfaces represented by dashed lines. The accelerator on the left is controlled through the dedicated co-processor interface, which can also be used for data transfers. Conversely, the accelerator on the right shares the data cache with the processor, and, therefore, data can be accessed directly from the cache. As co-processors, these accelerators are still both tightly coupled to the processor, because of the dedicated interface which is typically activated through special instructions, including load to and store from co-processor. At the same time, the coupling can be relaxed by implementing a direct interface to the bus for the co-processors. Data access through the co-processor interface or by sharing the private cache guarantees coherence with no additional hardware support. However, this choice may generate some undesired resource contention if the co-processor runs too often. When, instead, data are accessed through the bus, at least two options are possible: direct-memory access (DMA) and load/store-bus transactions that leverage the cache hierarchy of the system. The former can be either coherent or non-coherent, while the latter is necessarily coherent. For cache-coherent transactions, the co-processor is required to implement the coherence protocol adopted by the general-purpose cores in the system. As an example, the Rocket Chip Generator implements a co-processor interface for accelerators with optional coherent data-access through a bus interface [Asanović *et al.*, 2016].

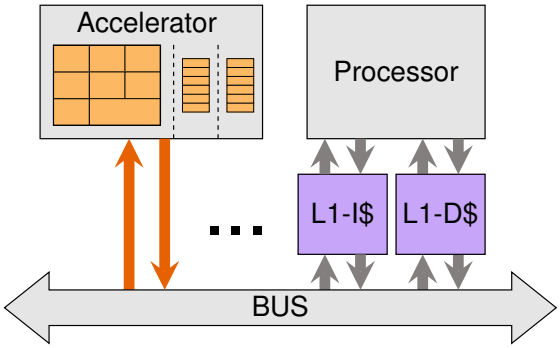


Figure 2.7: Loosely-coupled accelerator model, integrated over the system interconnect.

A similar approach proposes a model of co-processor accelerator that shares the data cache with the processor, but implements a dedicated memory management unit [Vo *et al.*, 2013]. Conservation cores are another interesting example of co-processors [Venkatesh *et al.*, 2010]. These relatively small engines are generated with high-level synthesis and are meant to improve the efficiency of complex instructions, which are likely to be useful for several software generations. Conservation cores are not programmable, but enable a certain degree of reconfiguration; they are controlled through special instructions and share the private cache with the processor [Venkatesh *et al.*, 2010].

Loosely-coupled accelerators. A tight coupling of accelerators with processors limits the achievable speedup when an accelerator is used to completely offload complex tasks processing a large amount of data. Therefore, it is advisable to completely decouple complex high-throughput accelerators from the general-purpose processor cores [Cota *et al.*, 2015]. High-throughput accelerators gain most of their speedup from a specialized and highly-parallel data-path. Parallelism, however, must be sustained by an adequate bandwidth in accessing data, which the private cache cannot guarantee. Loosely-coupled-accelerators (LCAs) benefit from being instantiated separately from the processor, to which they do not have dedicated interfaces. Therefore, designers of an LCA can afford to implement customized private-local memories with a multi-bank and multi-port structure capable of sustaining bandwidth required by the data-path [Cota *et al.*, 2015]. Configuring and running LCAs does not require ISA extensions, nor special instructions. Instead, they are configured with device drivers, similarly to any other SoC device or peripheral. The private memory can be mapped to the system memory to allow general-purpose processor cores to access it directly and manage data transfers for the accelerator. Alternatively, an LCA can leverage DMA and not expose

its private memory to the system.

Following this approach, some architectures have been proposed in the past. For instance, the accelerator-rich architecture refers to a sea of accelerators, implementing computational kernels at different degrees of granularity, all orchestrated by a centralized hardware controller [Cong *et al.*, 2014]. Sharing a similar view, other designers proposed the convolution engine as an architecture composed of arrays of specialized units for convolution, which can be used to accelerate map-reduce type of operations [Qadeer *et al.*, 2013]. Some exotic examples of fully decoupled accelerators are brain-inspired IPs that leverage neural networks and have been proven to be very efficient for approximate-general-purpose computation [Esmaeilzadeh *et al.*, 2012] or machine-learning applications [Chen *et al.*, 2014].

The list of research works related to LCAs shows how different their structure can be. This is possible because LCAs are not limited by the ISA, nor by any particular bus or co-processor interface protocol. As I show in Chapter 4, the integration of an LCA can be achieved through encapsulation, thus the design of the accelerator itself is decoupled from the rest of the system. Furthermore, since I contributed to the classification of TCAs versus LCAs [Cota *et al.*, 2015], I believe that LCAs are a key enabling factor of efficient computation and scalability of the workloads for many different application domains. Therefore, I set as one of the goal of this dissertation *to define a clear model and a simple interface for LCAs that promote IP reuse across different SoCs and are widely applicable to a variety of target applications*.

2.4.2 Accelerator-Memory Interaction

Alongside defining and choosing the ideal types of accelerator and system architecture for a target application, addressing memory aspects of specialized accelerators is a fundamental step to design efficient SoCs. In fact, even if the private memories of LCAs can reach up to 90% of their total area, the amount of data that can be stored on chip is usually limited to few MBs [Lyons *et al.*, 2012], while the memory footprint of applications is orders of magnitude larger. Sharing on-chip memory across accelerators [Cong *et al.*, 2014; Lyons *et al.*, 2012; Pilato *et al.*, 2014b] and processors [Cota *et al.*, 2014; Cota *et al.*, 2016; Fajardo *et al.*, 2011] is a promising approach to mitigate the shortage of storage close to the accelerators, but ultimately the limitations in accessing external memory must be addressed to cope with the ever growing data sets of applications. Prefetching, latency reduction

techniques and bandwidth optimization have been proposed in the past [Winterstein *et al.*, 2015; Lim *et al.*, 2013]. However, there has been no comprehensive analysis of the effects of multiple accelerators processing concurrently large amounts of data accessed through the off-chip memory. This analysis is made in Chapter 5, where I propose an approach to preserve the speedup of LCAs that scale the size of their data sets from a few kilobytes to hundreds of megabytes.

Programmability issues. Previous work has focused on optimizing memory access for concurrent accelerators with many different access patterns [Li *et al.*, 2011], while assuming that the accelerators are tightly coupled with the memory controller of the system. This implies that the accelerators are aware of the system memory mapping and allocation policies, thus reducing their portability and complicating the integration process. Chapter 5 addresses a more general problem that decouples the system memory hierarchy and memory-allocation policies from the accelerators' design. This approach leaves the software and the operating system in charge of memory management and eases programmability and portability of code across different platforms.

Direct-memory access. Other approaches, where accelerators are unaware of the memory subsystem [Yang *et al.*, 2016; Vogel *et al.*, 2015], address a type of accelerator that accesses memory with small transactions. This is similar to what general or special purpose processors do when loading cache lines. High-throughput LCAs, instead, tend to process larger data sets and access them at a coarser granularity. Hence, they benefit from long DMA transfers with DRAM [Cota *et al.*, 2015]. Various solutions have been proposed to expose the accelerator memory to the operating system and transfer large data through scatter-gather DMA mechanisms [Shukla *et al.*, 2013]. However, these solutions are usually implemented inside the operating system and data transfers are executed by the processor, thus degrading the speedup of the accelerators.

Multi-channel memory. The impact of multiple memory controllers has been studied for multi-core architectures [Awasthi *et al.*, 2010], especially to manage data placement, handle the effects on memory-access latency and avoid conflicts while scheduling accesses to the same physical memory [Liu *et al.*, 2012b]. In contrast, the experiments in Chapter 5 involve a multiple-channel memory system, implemented on FPGA, that can effectively parallelize the accesses to different physical memories. These results show for the first time the effects of the interaction between on-chip and off-chip memories in complex SoCs when many accelerators are processing simultaneously very

large data sets. In addition, techniques based on application profiling that further optimize the placement of data across many memory channels can be integrated into the proposed infrastructure [Muralidhara *et al.*, 2011].

2.4.3 Accelerators and Power Management

Dynamic voltage and frequency scaling (DVFS) is a well-studied technique for reducing the power consumption of SoCs, particularly in the context of processors [Herbert *et al.*, 2012; Herbert and Marculescu, 2007; Kaxiras and Martonosi, 2008] or application scheduling [Kornaros and Pnevmatikatos, 2014; Li and Martinez, 2006; Rangan *et al.*, 2009]. DVFS consists in dynamically adjusting the voltage-frequency pair, a.k.a. operating point, of a digital circuit based on the current workload. For instance, when a load operation misses in all levels of the cache hierarchy and needs to retrieve data from the external memory, or even from the hard drive, the thread of execution can stall for a long time. If all the other active threads are not interactive and have low priority, during this waiting time the logic of the pipeline can run in a low-power mode, thus trading off performance for energy savings. Depending on the number of clock and voltage domain on the chip, DVFS can be applied to the entire system, to a group of cores, or to other components, such as I/O peripherals. A study correlating on-chip regulators with power management of multi-core processor systems highlighted the importance of increasing the spatial granularity of DVFS to control power more efficiently [Kim *et al.*, 2008]. An extension of such results was made by combining coarse-grained and fine-grained DVFS to reduce the energy consumption in multi-core chips [Eyerman and Eeckhout, 2011]. Furthermore, machine learning techniques have been proposed to make power-management policies adapt better to the workload characteristics [Juan *et al.*, 2013].

There exist also analytical studies on voltage regulators that try to determine their dynamics and overheads. For instance, a model to analyze the overhead of DC-DC converters has been proposed in the literature [Park *et al.*, 2010], alongside with comparative studies between on-chip and off-chip regulators [Wang *et al.*, 2014b]. Furthermore, both switched-capacitor [Jevtic *et al.*, 2015] and inductor-based converters [DiBene *et al.*, 2010; Sturcken *et al.*, 2013; Tien *et al.*, 2015] have been studied as potential enabler for fine-grained DVFS. Indeed, steps towards fine-grained DVFS have already been made with state-of-the-art chip prototypes targeting superior energy efficiency. For example, the Intel Single Chip Cloud (SCC) research prototype features 48 cores with two

off-die voltage regulators [Salihundam *et al.*, 2011]. They supply eight voltage islands that are independently controlled by software.

DVFS has also been applied to accelerators by leveraging Razor flip-flops [Ernst *et al.*, 2003] to scale clock frequency and voltage according to the behavior of the environment surrounding the accelerator [Dasika *et al.*, 2008]. Nevertheless, there is a lack of research examining the effects of fast on-chip voltage regulators on SoCs that integrate many accelerators. In Chapter 6, I present a complete infrastructure to evaluate and effectively tune fine-grained power management on multi-accelerator SoCs, combining global system policies with local actuators.

Chapter 3

Raising the Level of Abstraction

System-level-design (SLD) methodologies have been advocated for years [Bailey and Martin, 2006; Densmore *et al.*, 2006; Gerstlauer *et al.*, 2009]. The 2011 international technology roadmap for semiconductors, however, still lamented that CAD tools provide little support for SLD and that this situation must change if necessary advances in productivity are to be achieved [ITRS, 2011].

On one hand, following a simple bottom-up approach, which does not consider any system-level metric in the early design stages, may lead to sub-optimal designs. In fact, components that are optimal when evaluated standalone, are not necessarily as optimal when combined with the other system components. As a simple example, consider an accelerator for computing the Fast Fourier Transform (FFT) that almost saturates the external memory. If considered standalone, its design is optimal, because it is capable to fully utilize the memory bandwidth without saturating the interconnect. Hence, the accelerator does not waste energy consuming data faster than they can be streamed into the system. However, if the accelerator for FFT is one among many others, its bandwidth requirements must be adjusted to account for the traffic on the system interconnect generated by all accelerators.

A top-down approach, on the other hand, forces designers to begin the implementation process of an SoC by specifying requirements and constraints for the entire system and progressively sort out the details of the components in a hierarchical fashion. However, designing the entire system architecture of large SoCs is extremely challenging without leveraging an SLD methodology. Furthermore, complexity of integrating an increasing number of heterogeneous components leads to to prolonged design and validation cycles.

My research work embraces this challenge and combines state-of-the-art commercial software with in-house tools to enable SLD of complex systems. Recently there has been some progress in the commercial production of SLD tools, particularly for high-level-synthesis (HLS) [Cong *et al.*, 2011; Coussy *et al.*, 2009; Coussy and Morawiec, 2008a; Fingeroff, 2010; Gupta and Brewer, 2008]. HLS allows designer to raise the level of abstraction from the traditional RTL-design flow and use higher-level languages as a starting point for the development of IP blocks.

Nevertheless, the adoption of HLS is not widespread and its usage is still limited to small portions of the designs [Martin and Smith, 2009]. In part this is due to a natural inertia of companies that have well-established sign-off points and verification procedures based on readable hand-written RTL code. However, I believe that the lack of expertise with the HLS flow is the main reason for not upgrading a design flow that struggles to cope with the increasing complexity of SoCs. My dissertation describes an SLD methodology that combines existing commercial software with in-house tools and provides guidelines to exploit HLS for increased productivity in heterogeneous SoC design. As illustrated in Section 3.1, HLS enables a much wider design-space exploration (DSE) by setting scheduling directives, which guide the HLS tool in creating the desired micro-architecture. Therefore, the design of an IP block can be decoupled from the others because, from a single source code, designers can derive a family of RTL implementations and choose at a later time those that satisfy the system-level constraints. Furthermore, input languages for HLS, such as SystemC, implement the concept of communication-based design, which evolves from latency-insensitive design [Carloni, 2015]. Specifically, a set of primitives that model queue-based data transfers decouple functionality from communication and let the designer focus on the functionality of the target IP blocks independently from their micro-architectures, latency and throughput. Moreover, IP blocks that can tolerate by construction latency variations at their interface ease the integration process by allowing the system interconnect to exert back-pressure in case of contention for accessing any shared resource.

Related efforts based on hardware-function generators aim at a simpler and modular design methodology [Nikolic, 2015]. This combines an SoC generator [Asanović *et al.*, 2016] based on the implementation of a RISC-V processor [Asanović and Patterson, 2014] with the description of specialized IPs in Chisel [Bachrach *et al.*, 2012]. Chisel is a high-level language, based on Scala, which still describes cycle-accurate circuit designs, similarly to RTL. Conversely, HLS promises

to boost productivity and promote IP reuse by allowing designers to re-implement their design for different technologies and for different performance-cost trade-off points without rewriting the original source code. Instead of proposing a new language for hardware design, my research work leverages SystemC, an IEEE-standard, which represents a valid input for most of the available HLS tools [Panda, 2001; Black *et al.*, 2009].

Design-space exploration is also a crucial step of the design process that should be brought to the highest possible abstraction level. DSE is typically carried out by using simulators [Cong *et al.*, 2014; Li *et al.*, 2011; Lyons *et al.*, 2012]. However, existing full-system simulators, also referred to as "virtual platforms" [Aarno and Engblom, 2014], are either very accurate, but unable to handle the growing complexity of SoCs in a reasonable time, or not accurate enough to capture the effects of subtle interactions across many heterogeneous hardware components and software [Arvind, 2014]. While accurate simulators that enable early power-performance analysis are useful and necessary for modeling individual accelerators [Shao *et al.*, 2015] and virtual platforms are widely adopted for early software development, this dissertation advocates the use of SLD methodologies and HLS to build FPGA prototypes that enable accurate full-system DSE.

The rest of the chapter provides the reader with an overview of existing tools, languages and techniques for HLS. In addition, it presents a few relevant state-of-the-art-design infrastructures and heterogeneous SoCs that share some of the motivations at the basis of my research work.

3.1 High-Level Synthesis

The common practice in industry for SoC projects in the early stages is to write some form of high-level specification for the system and its components. In particular, the functional specification is typically implemented as an executable program using languages such as C or C++. This program includes no details about the target hardware and its purpose is simply to validate the intended behavior of each component [Fingeroff, 2010]. Later, hardware architects explore the design space in search for suitable implementations and decide how that behavioral specification should be implemented. Choosing the system architecture is an extremely delicate process, which has direct impact on performance, area, and power consumption of the design. From the architectural specification, the design is manually coded by using a hardware-description languages, such as VHDL, Verilog or

the more advanced SystemVerilog. When the first RTL implementation is ready, the design undergoes several cycles of testing, debugging and code refinement. This process only terminates because engineers have to meet project deadlines [Fingeroff, 2010], but it is unlikely to formally prove that all bugs have been found and corrected.

Since the invention of traditional hardware-description languages, computing systems have become more sophisticated and complex. Verification engineers have tried to keep up with innovative techniques to perform complex corner-case tests and formal analyses, while the RTL creation process has not changed [Fingeroff, 2010]. Furthermore, the increasing size and complexity of modern SoCs exacerbates the design and verification problems, leading to sub-optimal designs, delayed sign-off points and potentially unresolved critical bugs. The challenge is double: first, finding an optimal architecture in an ever growing design-space is extremely unlikely if each micro-architectural variation requires to manually update and optimize RTL code; second, the manual process required to move across different layers of abstractions is a guaranteed source of bugs, which become harder and harder to uncover in complex designs. High-level synthesis (HLS) addresses these challenges by providing an automated flow to translate functional specifications into RTL. Specifically, HLS promises to speedup the design flow with an error-free methodology that allows designers to decouple the specification of the functional behavior from the micro-architectural decisions for its implementation.

Fig. 3.1 shows the typical flow of HLS. The designer provides a high-level description of a hardware component by using C, C++, or SystemC. The source code can be either purely functional, or loosely timed. SystemC, for example, is a run-time environment built as a C++ library, which introduces the notion of clock and allows designers to define special threads that are activated at every edge of such clock. This representation is called "loosely-timed", because it doesn't specify a precise scheduling of operations at each clock cycle. In fact, a designer may code the entire functionality of the target IP within a single thread, which can execute in SystemC in the span of a single clock cycle. Optionally, additional timing specifications can be coded in SystemC. For instance, a call to the built-in function `wait` causes the caller thread to suspend its execution until the next clock edge. A designer can leverage this feature to implement protocols, or simply to impose timing constraints. As shown in Fig. 3.1, the HLS tool parses the source code and builds the control-data-flow graph (CDFG) of the design. The latter, which may or may not be exposed to the designer,

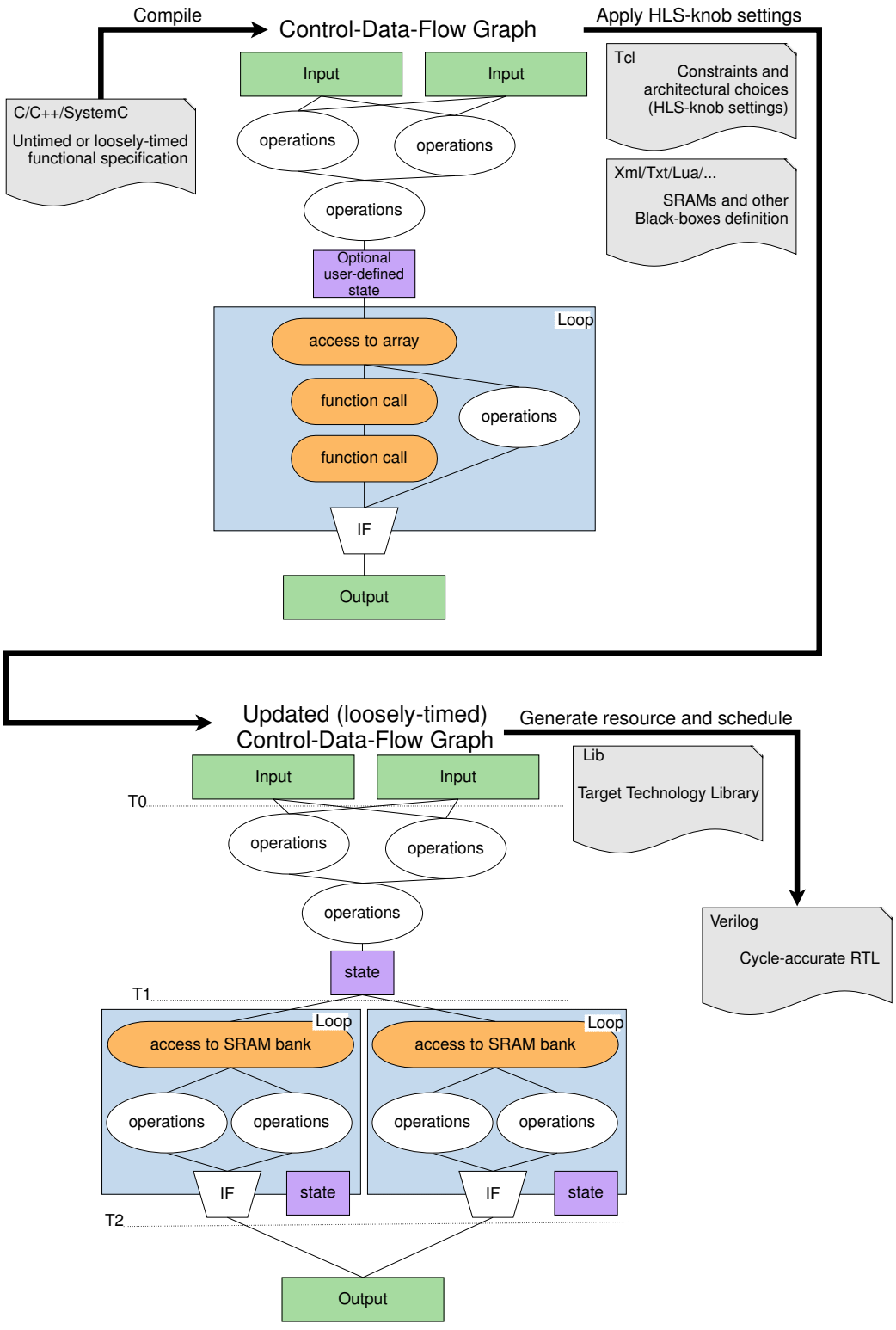


Figure 3.1: Overview of a typical high-level synthesis flow.

defines inputs, outputs, simple operation nodes, function calls, access to data structures, such as arrays, and loops with their exit conditions. Hard-coded timing specifications translate into states in the CDFG, which are present before running the HLS engine. Following the flow in Fig. 3.1, the HLS tool reads the definitions of the black boxes, the constraints and the micro-architectural choices set by the user. Black-boxes can be SRAM banks, which are used to implement large data structures and arrays, or pre-existing RTL IPs. Writing timing constraints for HLS usually consists of setting a clock period. Differently from logic synthesis, however, a different clock period may drastically change the resulting RTL. In fact, setting an aggressive target clock period for HLS doesn't simply imply that the tool will choose larger transistors to reduce the delay of the combinational path between registers. Instead, it implicitly determines the number of clock cycles that are required to execute in hardware the specified behavior. In practice, by specifying the clock period, the designer has indirect control on the number of states that will be encoded in the resulting RTL design. Micro-architectural choices, referred to as *knob settings* or HLS directives, explicitly drive the HLS tool in the process of scheduling the high-level code into a cycle-accurate hardware implementation.

Example. *The flow in Fig. 3.1 shows a possible transformation of the CDFG after applying the HLS-knob settings. In particular, let us focus on the loop transformation: while the original CDFG has one single loop, the updated CDFG has two loops, both positioned within the same coarse-grain time step. This can be achieved by assigning the loop unrolling directive to the loop present in the source code. The HLS tool interprets the directive and generates a functionally-equivalent CDFG where the control logic and the body of the loop are replicated. Then, by applying another directive, known as* loop breaking*, the tool inserts a state in the body of each replicated loop. Thus, every iteration of those loops will take at least one clock cycle. Additional states might be added during the scheduling process to meet timing constraints.* □

Other typical HLS knob settings include *function inlining* which breaks function calls into a flat hierarchy of simple operations; *function sharing* which creates, instead, a shared module for the function and the corresponding control logic to access it; *loop pipelining* which breaks the body of the loop into multiple stages and automatically implements the necessary control logic to enable multiple loop iterations to run concurrently; *latency constraint* that allows designers to specify the exact number of cycles that a specific set of operations should take; *array flattening* which maps the array to a set of registers; and *memory allocation* which, instead, maps an array to a specified SRAM bank.

The final steps of the flow, shown in Fig. 3.1, are parsing the technology library to generate the necessary hardware resources and scheduling the design into synthesizable RTL. The step of generating the hardware resources consists in running logic synthesis on every primitive operation present in the source code. A primitive operation could be an addition, a multiplication, a comparison or an conditional statement. Based on the provided technology library, this step returns a characterization of each operation in terms of estimated latency, area and power. With these information, the HLS tool schedules all operations such that the resulting RTL meets the timing constraints given the specified clock period. The resulting cycle-count is due to a combination of clock period, HLS-knob settings, source code and scheduling algorithm. The latter may change depending on the HLS tool, but it is typically a *list scheduling* algorithm [De Micheli, 1994; Coussy and Morawiec, 2008b], which attempts to minimize the number of cycles given the available resources. Unless the designer sets a scheduling policy that increases resource sharing at the expenses of performance, the number of resource is not limited, and the HLS tool can replicate any resource from the generated pool in order to produce a correct RTL implementation that meets timing and has optimal latency.

Note that, depending on the specific tool, many other knob-settings may be available, in addition to those mentioned above. As a result, from a single high-level description of a hardware component, a designer can generate several different RTL implementations, each characterized by specific latency, throughput, area and power consumption. Some combinations of these characteristics are typically used to compute the performance and cost metrics relevant for the designer. Hence, each implementation can be represented as a point in a bi-objective design space representing a trade-off between cost and performance. In such space, a Pareto-optimal design point corresponds to an implementation whose performance-cost trade-off is such that no higher performance implementation exists for the given cost and no lower cost implementation exists for the same performance [Shacham *et al.*, 2010; Zyuban and Strenski, 2002]. The set of Pareto-optimal points, or Pareto set, defines the *Pareto curve* in the bi-objective design space. The problem of finding an optimal implementation, therefore, comes down to writing sets of knob-settings, running the HLS tool to derive the corresponding RTL implementations, and selecting the design points that lie on the Pareto curve. The ranges, or spans, of cost and performance covered by the Pareto curve in the bi-objective design space are directly correlated to IP reuse: designs with larger cost and

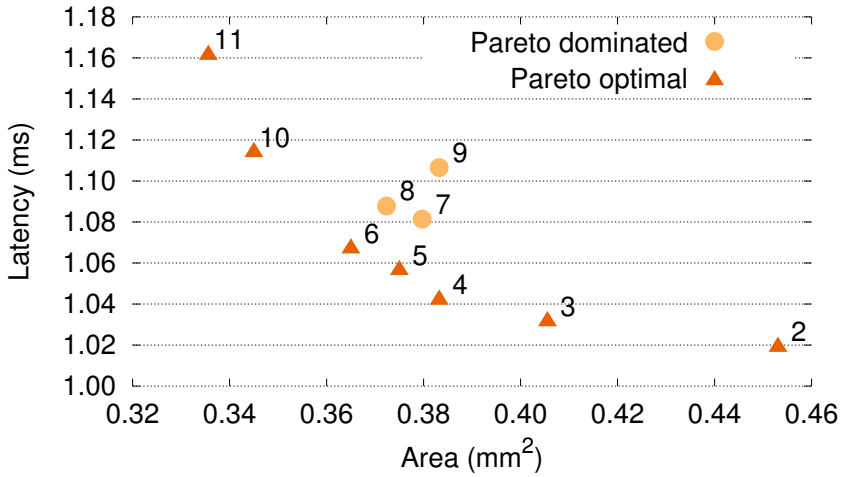


Figure 3.2: Example of design-space exploration with HLS. The chart reports both Pareto-optimal and Pareto-dominated design points.

performance spans are, in fact, more likely to be integrated in different target SoCs.

Example. Let us consider the example of GRADIENT*, a kernel of computation that processes a gray-scale image to compute the variation of luminance across pixels.* GRADIENT *reads pixels from memory in a streaming fashion and computes a result in the form of one floating-point number per pixel. The body of the computation is a loop with independent iterations: the output computed during the* i^{th} *iteration of the loop is independent from the output of any other iteration. In this favorable scenario, it is possible to compute concurrently more than one output, as long as there are enough computational resources to perform all parallel operations. For example, by instrumenting the HLS tool to unroll the body of the loop two times, the resulting RTL implementation of* GRADIENT *computes two output pixels in parallel. Therefore, assuming that the latency for executing the body of the loop for a given target clock cycle is* x *and the number of pixels is* N*, the total latency for this implementation of* GRADIENT *is* $\frac{x \times N}{2}$*. As a counterpart, the number of resources to implement this loop is expected to double. In theory, a large number of unrolled iterations should always lead to RTL implementations having smaller latency and larger area. However, when the implemented functionality is not trivial, the HLS tool has the capability to perform an optimization, based on heuristics, which lead sometimes to unexpected results. Fig. 3.2 shows the results of scheduling* GRAYSCALE *with a number of unrolled loop iteration ranging from two to eleven. The chart shows that three of the design points are not Pareto optimal, because for each of them there exists at least another design point with at most*

same latency, but smaller area occupation, or at least same area occupation, but smaller latency. These are referred to as Pareto dominated and should be excluded from the DSE. □

HLS represents a necessary step toward raising the level of abstraction in designing SoCs. There are still some limitations, however, that need to be addressed. First, HLS tools can only instantiate memories as black boxes and they cannot generate customized memory sub-systems by combining the available SRAM banks. Second, while HLS tools offer a sophisticated design flow to generate optimized RTL for a single component, they lack a complete SLD flow. The latter must guide HLS to achieve the system Pareto set of design points, resulting from the composition of multiple components. The ESP design methodology, presented in Chapter 4 tackles these limitations and leverages HLS to perform system-level DSE.

3.1.1 Languages Proliferation

C and C++ languages are popular input formats for HLS tools for several reasons: a large number of existing algorithms are written in these languages; they facilitate hardware/software co-design since most embedded software is written in C; and C-level functional execution is much faster than RTL simulation. Over the years, however, they have also manifested some proven limitations because they inhibit the specification, or the automatic inference, of important properties of hardware related to concurrency, timing, data formatting, and communication. Arguably, one goal of HLS is to hide such details from the designers and let them focus only on functionality while writing the source code. Nevertheless, sometimes it is required to specify protocol behaviors, for which timing cannot be ignored. Moreover, even when timing could be ignored, the ability to explicitly express parallelism in a suitable way for HLS can unlock a region of the design space otherwise unreachable by simply tuning the knob settings. Other limitations of C and C++ are due to the original intent of many programs that were written with a focus on software performance without considering how they would map onto specialized hardware. For instance, applications written for multi-core architectures use multi-threading libraries that make the code difficult to be refined for HLS. The large semantic gap between software and hardware specification led to the proliferation of a variety of languages and tools. Each of them expresses programs at a different level of abstraction in a spectrum that goes from a low-level RTL specification up to a software specification that doesn't include any explicit detail about hardware.

SystemC. Among the proposed languages, SystemC [Panda, 2001] is the IEEE-standard that has become a *de-facto* standard for HLS and is accepted by all available commercial HLS tools. It consists of a C++ library and an event-driven simulator which behaves similarly to an RTL simulator, thus exposing the effects of concurrency. Functions that are supposed to execute in parallel are declared as one of three types of SystemC threads, which are scheduled by the simulation engine based on events. Events are typically the activation of the reset and the rising edge of the clock, but designers could also specify a sensitivity list that includes other signals, similarly to RTL languages. In addition, SystemC implements hardware-specific data types, such as bit vectors, and distinguishes between native C++ variables and SystemC signals. Variables follow the standard C++ semantics, while the behavior of signals is similar to that of wires in Verilog; hence a write operation on a signal updates its value only when the SystmeC thread is rescheduled. Considering the current state-of-the-art of HLS tools, all the hardware-oriented features of SystemC are necessary to write code amenable to HLS. Nevertheless, this makes the SystemC implementation of an application specification (or a portion of it) a complex and time consuming task. The task gets even harder if the goal is to enable aggressive DSE that produces implementation points with large performance and cost spans. The ESP methodology, described in Chapter 4, is based on SystemC templates and HLS guidelines devised to enable an efficient DSE and ease the process of porting algorithms, and software in general, to a description amenable to HLS.

Domain-specific languages. As an alternative to C, C++ and SystemC, designers may decide to use languages developed for a specific application domain. Being optimized to describe specific types of computational kernels, these languages can benefit from a compact and high-level syntax, which is very efficient for representing the target code. Furthermore, applications from a specific domain tend to have similar structures, therefore it is possible to derive constraints that help the compiler figure out how to map the source code onto different computing platforms. This has been the case for OpenCL, a domain-specific language for parallel programming whose compiler has different back-ends capable of generating executable code for general-purpose processors, GPUs, and even FPGAs [Group, 2009]. Another interesting example is Halide, a functional language specialized for image processing that is very efficient for stencil computation [Hamey, 2015]. Similarly to OpenCL, the Halide compiler's back-end is capable to produce code for different devices, including FPGAs.

Generating code for FPGA is a synthesis operation similar to HLS. In practice, however, scheduling a behavior from domain-specific languages is simpler than with HLS, because specialization can be exploited to ease the hardware-optimization process, at the expenses of general applicability.

Bluespec. Moving to a lower-level of abstraction, Bluespec is a proprietary language that provides designers with the ability to specify hardware with the same accuracy of RTL, but in a more compact way [Arvind, 2003]. Similarly to Verilog, describing hardware components with Bluespec implies to define modules, wires, blocking and non-blocking assignments. Making a step toward HLS, however, Bluespec replaces processes with atomic rules from which control logic is inferred automatically. For instance, let us assume that two rules assign a value to the same wire based on different conditions. If the two conditions were to be satisfied in the same clock cycle, the compiler automatically generates the logic to avoid the conflict, which would otherwise translate into a double-driven signal. In practice, the Bluespec compiler can perform a partial schedule of the code to decide which rules run concurrently and which don't. However, the specification and the timing of the behavior of a rule is still responsibility of the designer. Arguably, Bluespec enables a faster DSE, with respect to RTL, but the implementation of the micro-architectural choices is not an automatic process.

Chisel. Even closer to the lower end of the spectrum of abstraction layers there are languages that allow designers to retain control on all details of the generated hardware. Differently from RTL, however, they leverage more complex constructs and more powerful semantics to enable a compact and less error-prone description of the target design. Similarly to SystemC, these languages typically offer a simulation environment much faster than RTL, but, rather than aiming at HLS, they still describe cycle-accurate hardware. That is the scheduling of all operations is written in the source code. Therefore, a translator, and not a scheduler, produces the RTL usually in the form of synthesizable Verilog. A popular example of these languages is Chisel [Bachrach *et al.*, 2012], which defines the necessary data types and constructs to embed hardware description in Scala. Clock and reset are not present in the source code. Instead, they are automatically inferred by the translator, but in a deterministic way. While Chisel is not moving toward HLS, the advanced features inherited from Scala allow an extremely compact hardware representation that removes most of the verbosity of traditional hardware-description languages.

3.2 Design Frameworks for Heterogeneous Systems

Next to emerging CAD tools for HLS and languages that aim at raising the abstraction level for hardware design, there have been efforts from both industry and academia to help build SoCs.

A very recent and successful example from academia is the "Agile" methodology [Lee *et al.*, 2016] that leverages the Rocket Chip generator [Asanović *et al.*, 2016] to speedup the design of RISC-V-based SoCs with a complete flow from the hardware description language Chisel to the chip layout. The success of "Agile" confirms part of my thesis, which advocates the need of combining a design methodology with an underlying architecture that hides the complexity of system integration. Differently from my approach, however, the "Agile" methodology only allows designers to implement specialized IPs in the form of tightly-coupled accelerators, because the underlying architecture is inherently processor-centric. Furthermore, the Rocket Chip architecture relies on a traditional bus-based interconnect, which limits the scalability of the design. Finally, since all IPs, including the custom TCAs, are designed in Chisel, "Agile" does not offer an automatic flow for DSE.

From industry, the case of the ZYNQ programmable SoCs [Crockett *et al.*, 2014; Kathail *et al.*, 2016] is a representative one for all design platforms that combine FPGA fabric with an embedded processor cluster. Users of the ZYNQ can develop and package their IPs to be interfaced with a cluster of ARM processors through the AMBA® AXI® open-bus standard from ARM. The design flow doesn't require users to elect a particular method to create their IPs; in fact, they could be manually coded in RTL, or generated with HLS. The only requirement is that the interface must be compliant with the bus protocol. In this case, the flow eases the process of integration, but fails to help the designer to run an efficient DSE. Furthermore, integration is still based on a bus which shares similar limitations with the Rocket Chip in terms of scalability.

The "ARA Prototyper" [Chen *et al.*, 2016], which stands for "accelerator-rich-architecture [Cong *et al.*, 2014] prototyper, is also an FPGA-based approach that leverages the ZYNQ system to instantiate a sea of accelerators controlled by the processor cluster. This project shares with my dissertation the target of enabling fast system-level DSE through FPGA prototyping. However, the "ARA Prototyper" architecture is missing a companion SLD methodology that guides the designers in performing DSE at multiple levels of abstraction. Furthermore, this architecture is bound to the ZYNQ system and the management of shared resources, including memory access, is centralized,

thus limiting the scalability of the target design.

Moving the focus from embedded systems to servers and workstations, the recent efforts of Intel with the HARP program promise to ease the integration of custom accelerators with server-class processors [Schmit and Huang, 2016]. Similarly to the ZYNQ platform, HARP provides designers with the specification of a bus interface that has to be used to implement specialized accelerators. A partial bitstream implements the low-level bus and cache-coherence protocols, while it exposes the IP designer to a simpler interface. In addition, HARP provides templates and examples for implementing accelerators using HLS, OpenCL and traditional RTL. Even though HARP shares some of the motivations of my thesis statement, it is still a processor-centric platform and provides no clear SLD methodology to scale the number of heterogeneous components in the system. Perhaps, a future implementation of the HARP platform should interface multiple independent FPGAs with the processor cluster through a more scalable interconnect.

Part II

Embedded Scalable Platforms

Chapter 4

Embedded Scalable Platforms

Motivated by heterogeneous SoC design challenges, my research work develops a concrete implementation of the concept of *Embedded Scalable Platforms* (ESP), which brings together a new platform architecture with a companion SLD methodology. The architecture addresses the complexity of IP-block integration by balancing hardware specialization and design regularity with a tile-based approach. The result is a flexible design that promotes IP reuse and facilitates full-system DSE. The methodology seeks to increase productivity by moving the bulk of the engineering effort to the system level and reducing the gap between hardware design and software programming. Thanks to ESP, the focus of SoC architects can shift from low-level details of the system infrastructure to selecting the best mix of components for a given application-driven design. Further ESP enables designers to analyze the limits imposed by the interaction of all system components, choose the number of IP blocks to instantiate and pick the best suitable implementation.

Fig. 4.1 illustrates the relation between the ESP methodology and architecture. An SoC is an instance of an ESP architecture obtained by specifying a mix of tiles. Each tile may implement a processor, a hardware accelerator, or an auxiliary functionality like I/O access. The number and mix of tiles of a particular ESP instance depends on its target application domain. The choice of a specific tile combination is the result of an application-driven *design-space exploration (DSE)* that is guided by the ESP methodology and supported by a combination of state-of-the-art commercial CAD tools and in-house developed tools [Di Guglielmo *et al.*, 2014; Liu *et al.*, 2012a; Pilato *et al.*, 2014a]. The premise of this approach is that the target workloads must drive both the software-programming and the hardware-design efforts throughout all stages of the system realization.

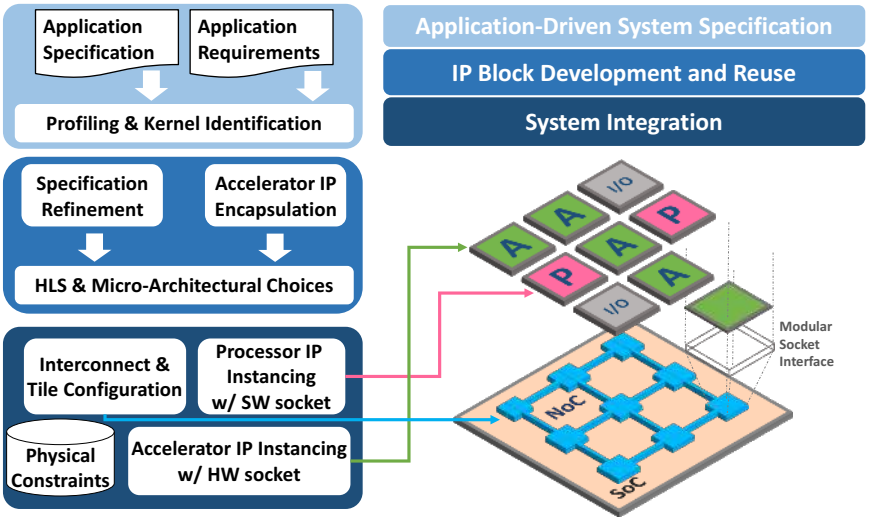


Figure 4.1: The SLD methodology for Embedded Scalable Platforms.

The system specification involves the definition of the application requirements and the development of the application software. Software profiling identifies those critical computational kernels that deserve to be implemented in hardware. The corresponding accelerators can be either developed from scratch or acquired as reusable IP blocks. Critically, for both cases the ESP methodology advocates and promotes the use of *high-level programming languages* such as SystemC [Black *et al.*, 2010] and *high-level synthesis (HLS) tools* [Cong *et al.*, 2011; Coussy *et al.*, 2009; Fingeroff, 2010] to design parameterized IP blocks and provide a richer spectrum of power/performance trade-off points. This augments reusability because architects of very different SoCs can synthesize those IP-block implementations that are more suitable to their purposes.

At the level of individual IP block, the task of DSE consists in deriving a set of alternative implementations, each offering a particular cost-performance trade-off point. The broader are the cost and performance ranges spanned by the Pareto curve, the higher is the reusability of the IP block. At the system level, the DSE is inherently a component-based design effort, as the choice of a particular RTL implementation for a module must be made in the context of the choices for all the other modules that are also components of the given system. A particular set of choices leads to a point in the multi-objective design space for the whole SoC. So, the process of deriving the diagram of Pareto-optimal points repeats itself hierarchically at the system level [Liu *et al.*, 2012a].

The ESP methodology mitigates the complexity of integrating heterogeneous components by

providing a regular but flexible socket-based template and a set of *platform services*, including: accelerator reservation and configuration, data transfers, performance counters, and diagnostics. In particular, the accelerator tiles contain high-throughput accelerators that are loosely coupled with the processor: each accelerator typically works on large (e.g. $300MB$) data sets by leveraging a private local memory that is tailored to its specific needs and exchanges data with main memory through a private DMA controller [Cota *et al.*, 2015].

The platform services are supported by: (1) a *scalable communication and control infrastructure*; (2) a set of *configurable hardware sockets* that interface the components to the interconnect and are designed for modularity and flexibility by following the Protocols and Shells Paradigm of latency-insensitive design [Carloni, 2015; Carloni *et al.*, 2001]; and (3) a set of *software sockets* that convey the illusion of a simpler homogeneous architecture to the programmer. The interconnect can be realized either with a bus or a network-on-chip (NoC), depending on the needs in terms of bandwidth and platform services. Designs that have a larger number of components typically rely on an NoC, which can be scaled up by adding more virtual channels or physical planes [Yoon *et al.*, 2013].

The system-integration phase is completed by the validation of the given ESP instance through its emulation with an FPGA board. This prototyping effort is strongly simplified by the adoption of SLD methods. For example, HLS tools provide an immediate way to re-target an accelerator implementation from an ASIC to an FPGA technology. FPGA emulation is critical not only to validate the correctness of the design but also to obtain an accurate analysis of its performance. For instance, as illustrated later in this chapter, it enables to assess how the performance of a given hardware accelerator is affected by its interaction with all the other system components.

4.1 ESP Accelerator Flow

This section defines a simple, yet flexible, model and interface suitable for most loosely-coupled accelerators. This model is the premise to show how to leverage HLS for component-based and compositional DSE. To illustrate some of the aspects of this process, I present the design of an ESP instance implementing `WAMI-App`, an accelerated version of the Wide-Area Motion Imagery (WAMI) application [Porter *et al.*, 2010]. WAMI is an image processing application used for aerial

FUNCTION	LOC	IF	LOOP	ASSIGN	FCALL	ARROP
Debayer	195	4	24	70	12	56
Grayscale	21	2	2	8	0	4
Warp	88	12	0	51	3	11
Gradient	65	7	4	34	0	54
Subtract	36	7	2	13	0	3
Steep.-Descent	34	0	3	21	0	3
SD-Update	55	9	4	20	0	5
Hessian	43	0	6	18	0	4
Matrix-Invert	166	33	8	59	8	20
Matrix-Mult	55	7	5	20	0	5
Reshape	42	11	1	15	0	2
Matrix-Add	36	7	2	13	0	3
Change-Detect.	128	12	9	62	3	41
Total	964	111	70	404	26	211

Table 4.1: WAMI-App source code profiling.

surveillance. It processes a sequence of input frames to extract masks of "meaningfully changed" pixels. For example, it can detect and track vehicles moving on the ground, while discarding environmental noise, e.g. shadows, surface reflections, etc. In this case, the design starting point is a software specification available in the PERFECT Benchmark Suite [Barker *et al.*, 2013]. This is a collection of applications and kernels that target energy-efficient high-performance embedded computing. The suite distribution includes source files and test applications, both written in C, together with some samples of input sensory data.

Application profiling. The `WAMI-App` specification is coded using a subset of the C language: e.g, it has a limited use of pointers and makes no use of dynamic memory allocation and recursion; also, with the exception of few mathematical functions (e.g. `exp()`, `sqrt()`, etc.), no external library functions are called. This simplifies the task of porting this specification from C into a subset of SystemC that can be synthesized effectively with HLS tools. Before starting implementing the SystemC code, it is recommendable to profile the initial program to identify critical regions of code and function calls. Some interesting metrics are the time spent in functions or code regions that will become accelerated computational kernels; the number of accesses to arrays, i.e. storage elements; the amount of control instructions with respect to other operations; the number of loops

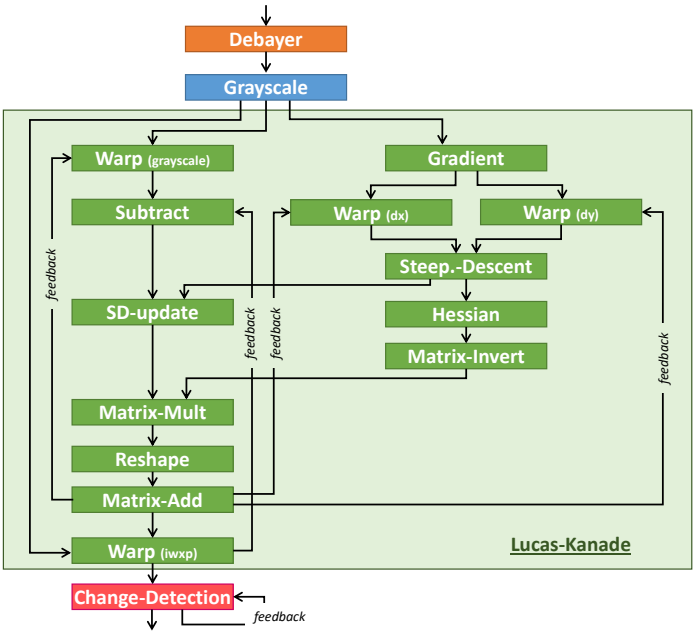


Figure 4.2: WAMI-App dependency graph.

which are amenable for manipulation during HLS; and the sensitivity to cache size and associativity. Standard profiling tools can be used to gather these metrics. For example, statistics about types of instructions, cache misses, branch prediction and function calls can be collected in Linux with `perf`, `gprof`, `ftrace`, `strace`, et cetera. However, if customized or more detailed information is needed, simulators, such as Pin [Luk *et al.*, 2005] or SESC [Ortego and Sack, 2004], can be employed. These allow the user to define callback functions that are invoked every time a particular instruction is executed, thus enabling custom profiling.

Table 4.1 lists some of the characteristics of the WAMI specification. In particular, for each main function, it reports the number of: lines of C code (LOC), conditional statements (IF), loops (LOOP), assignments (ASSIGN), function calls, including both functions provided with the source code and functions from the `C math` library (FCALL), and read/write operations on arrays (ARROP).

Extracting parallelism. The WAMI specification in C consists of four main computational kernels: the DEBAYER filter, the RGB-TO-GRAYSCALE conversion, the LUKAS-KANADE image alignment, and the CHANGE-DETECTION classifier. After porting the WAMI code into SystemC, the LUKAS-KANADE kernels are partitioned into nine SystemC processes to have the option of synthesizing an accelerator

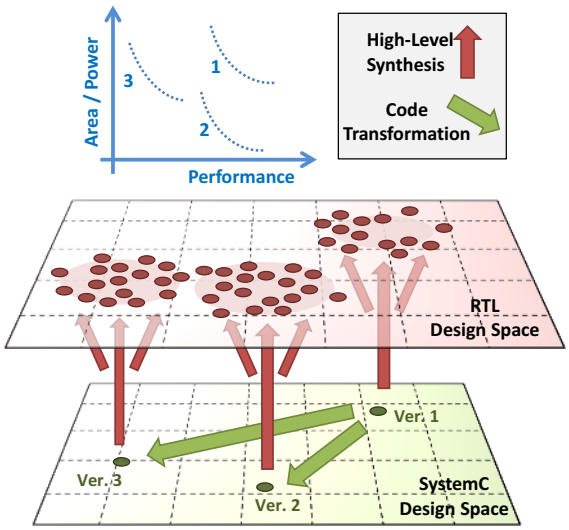


Figure 4.3: The relationship between the SystemC and the RTL design spaces.

for each of them, thereby increasing parallelism at the architecture level. The block diagram of Fig. 4.2 is a data-dependency graph highlighting the relations among the computational kernels of WAMI when processing one frame. Since not all kernels depend on each other, there is *potential* for parallel execution: e.g., MULT must run after SD-UPDATE and INVERT-GJ, which instead can run concurrently. In addition, overlapping the processing of multiple frames in a pipeline fashion would allow for more accelerators to execute in parallel.

Refinement and micro-architectural choices. HLS tools provide a rich set of configuration knobs [Cong *et al.*, 2011; Coussy *et al.*, 2009; Fingeroff, 2010] that can be applied to synthesize a variety of RTL implementations. These implementations are based on different micro-architectures and provide different cost-performance tradeoffs points. Fig. 4.3 shows the relationship between the SystemC design space and the RTL design space. The micro-architectural knobs are "push-button" directives to the HLS tool represented by the red arrows in Fig. 4.3. In addition, the engineer may perform manual transformations (represented as green arrows) to obtain revised versions of the SystemC specification: these transformations preserve the functional behavior but extend the RTL design-space exploration. For example, they can expose parallelism, remove false dependencies, increase resource sharing etc. In this way, the engineer may either reduce the area/power ($ver.\ 1 \rightarrow 2$) or improve the performance ($ver.\ 1 \rightarrow 3$).

```
/* ******************** */
/* before transformation */
/* ******************** */
for (y = 1; y < nRows - 1; y++)
 for (x = 1; x < nCols - 1; x++)
 {
  z = y * nCols + x;
  Xgrad[z] = (  Iin[y*nCols + (x+1)] -
                Iin[y*nCols + (x-1)]) >>1;
  Ygrad[z] = (- Iin[(y-1)*nCols + x] +
                Iin[(y+1)*nCols + x]) >>1;
 }
```

```
/* ******************** */
/* after transformation */
/* ******************** */
float cntrl_diff(float a, float b)
 return ((a - b) >>1);
for (y = 1; y < nRows - 1; y++)
 for (x = 1; x < nCols - 1; x++)
 {
  z = y * nCols + x;
  Xgrad[z] = cntrl_diff(Iin[y*nCols + (x+1)],
                Iin[y*nCols + (x-1)]);
  Ygrad[z] = cntrl_diff(Iin[(y+1)*nCols + x],
                Iin[(y-1)*nCols + x]);
 }
```

Figure 4.4: Code transformation by function encapsulation.

Example. A simple HLS directive is function inlining. *It allows HLS to further optimize the body of the function in the context of the caller and it removes performance degradation due to inter-module communication; but the complexity of the synthesized hardware may increase due to resource replications. The function-inlining knob is provided with most HLS tools. In contrast,* function encapsulation *is a manual code transformation that identifies frequent patterns in the C-like implementation and encapsulates them with new functions. Fig. 4.4 shows a portion of the* GRADIENT *kernel of WAMI-App: the engineer may be able to identify the pattern* `(a - b) >>1` *in the before-transformation code (on the left-hand side) and encapsulate it in the function* `cntrl_diff()`. *This function is then used at any occurrence of the pattern. Function encapsulation allows HLS to reduce the large number of states in the main module that may produce an inefficient circuit with a long critical path delay due to the complicated control; in addition, the body of the functions can be highly optimized and reused. On a final note, this code transformation is reversible: the automatic inlining of the function restores the original RTL datapath.* □

ESP accelerator model. With ESP, I propose a model for the accelerators that is *loosely-coupled* with the processor [Cota *et al.*, 2015]. The accelerator is located outside the processor core and interacts with it and the off-chip memory via DMA, through the on-chip interconnect. This model was defined after implementing many different accelerators for high-performance embedded application kernels. Each kernel has distinctive characteristics that influence the design of the corresponding accelerator. These include the degree and granularity of computational parallelism, the ratio of computation versus communication with main memory, and the memory access patterns to read and write data. Fig. 4.5 presents three examples of memory access patterns from WAMI: DEBAYER has a strided pattern, WARP has a data-dependent pattern, while GRADIENT has a sequential-access pattern. In spite of differences among accelerators, it is possible to identify some common aspects in the

```
/*
 * Debayer
 */
for (row = PAD; row < NUM_ROWS-PAD; row++)
  for (col = PAD; col < NUM_COLS-PAD; col++)
  {
    // Access a 5x5 stencil
    input[row-2][col-2] = mem[(row-2) * (NUM_COLS-PAD) + col-2];
    input[row-2][col-1] = mem[(row-2) * (NUM_COLS-PAD) + col-1];
    // ...
    input[row+2][col+2] = mem[(row+2) * (NUM_COLS-PAD) + col+2];
  }
/*
 * Warp
 */
// Check for missing data and access memory based on precomputed information:
if (missing_pixel_count > 0)
  for (uint32_t col = 0; col < missing_pixel_count; col++)
    input[col] = mem[f(missing_pixels) * NUM_COLS + g(missing_picexl) + col];
/*
 * Gradient
 */
// Stream data from memory in order
for (pix = 0; col < NUM_PIX; col++)
    input[pix] = mem[pix];
```

Figure 4.5: Example of three different memory access patterns from DEBAYER (top), WARP (middle) and GRADIENT (bottom).

behavior of loosely-coupled accelerators. Such commonalities lead to the definition of the accelerator's model for ESP and of a corresponding configurable interface that can encapsulate different kernels.

As shown in Fig. 4.6, the accelerator behavior is organized in four main phases. First, there is the *configuration* that entails the interaction between software and hardware. By accessing state and command registers, a device driver checks the status of the accelerator, configures it, and starts a new execution. The configuration phase takes a negligible time with respect to the others. When invoked, the accelerator iterates over three main phases: *input*, *computation* and *output*. These repeat for portions or blocks of data until the entire input is processed. During the input phase, the accelerator issues a DMA request for a block of data from main memory. Such request is autonomous and not controlled by the processor. The accelerator transfers the block of data from main memory to a properly-sized *private local memory (PLM)* using transaction-level modeling (TLM) primitives. When data are available in the PLM the accelerator performs the actual computation specified by the synthesized functionality. Finally, upon completion, an autonomous write request is issued to store the results back into main memory. Depending on the PLM structure and capacity, these three phases are either serialized or overlapped to improve the efficiency of the accelerator. Hence, the input and output phases are split according to the size of the PLM. For most accelerators, the organization of the PLM is critical to make these components operate concurrently. At a higher

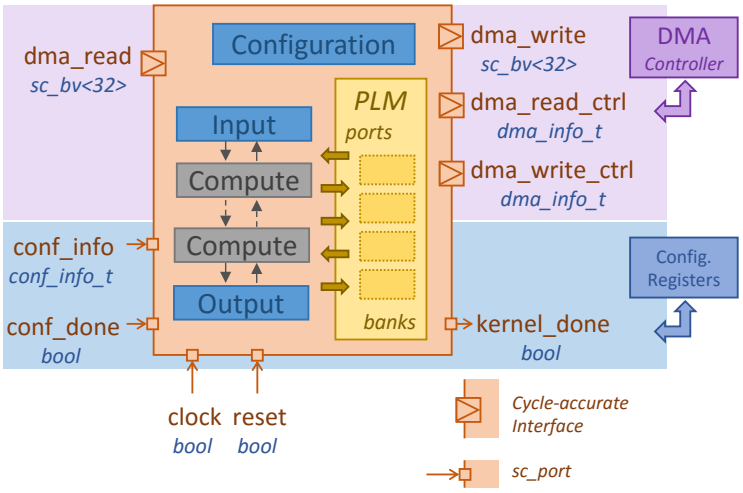


Figure 4.6: ESP Model for accelerator design and integration.

level, the PLM implementation as circular and ping-pong buffers may enable the pipelining of computation and DMA transfers with the off-chip memory. At a lower level, this is possible only by providing multiple memory banks and ports. An in-house tool takes the specification for the PLM made by the designer and generates the accelerator's memory subsystem by using available primitive SRAM blocks [Pilato *et al.*, 2014a]. The resulting multi-port interface is exposed to the HLS tool that employs it to schedule the necessary concurrent accesses to the PLM.

The WAMI-app has been a driving test case for refining the model of accelerators for ESP and the system-level design methodology. In fact, twelve computational kernels from the WAMI-app were implemented as accelerators and were subject to an exhaustive DSE involving tuning the HLS-knob settings, refining the SystemC specification and generating several implementations of the PLM. These twelve accelerators target an industrial $32nm$ ASIC technology. Fig. 4.7 reports the result of the DSE, including the Pareto sets for each implemented accelerator. Design points are plotted on a bi-objective design space, where *area* is the cost function and *effective latency* is the performance metric. The latter is computed as the product of the *latency* required to process the entire data set, which is measured in clock cycles, and the target clock period.

Example. Let us consider DEBAYER*, an image-processing application that takes as input an image in "Bayer" format, which is typical for digital cameras using charge-coupled devices (CCDs) to capture a picture. Each pixel of the "Bayer" image stores information for only one of the three typical channels red, green and blue (RGB). The* DEBAYER *application restores the missing information by interpolating available*

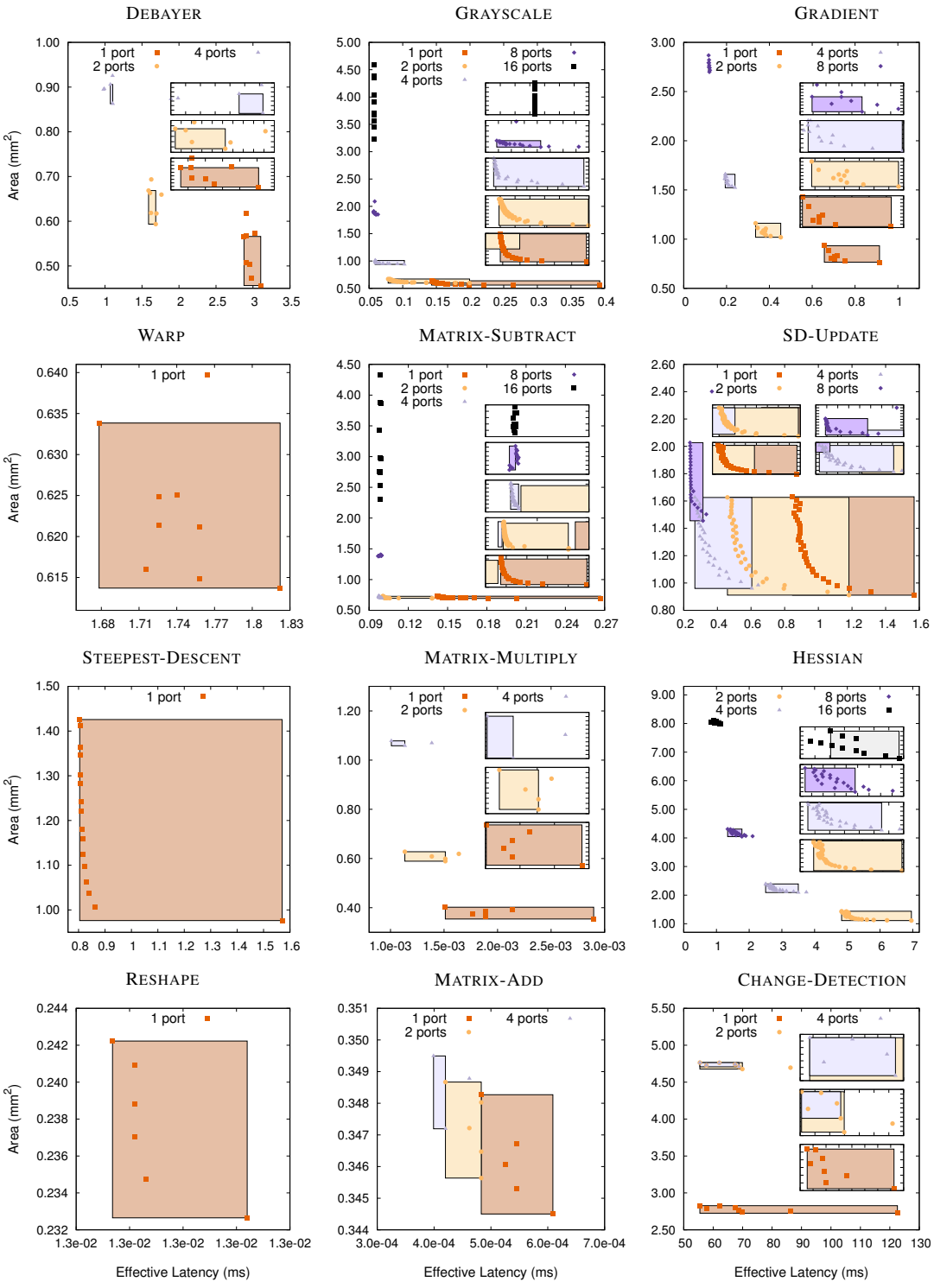


Figure 4.7: Component-based DSE for the twelve accelerators of the WAMI-App.

data on sliding stencils, each sized 5×5 *pixels. Given a specification in C of* DEBAYER*, the first step consists in writing the SystemC code and separate I/O from computation, according to the model of Fig. 4.6. This is the step in which designers should put their best efforts, as implementing a good SystemC specification may unlock larger areas of the design-space. For instance, given the particular access pattern of* DEBAYER*, its PLM is implemented as a circular data buffer, which can hold at least five plus one rows of the input image. For* DEBAYER*, circular-buffering improves throughput by enabling an overlap in time of computation and communication. After reading the first five lines from memory, in fact, the computation process starts interpolating input data using the stencils and then it produces the first output row of the reconstructed RGB picture. In the meanwhile, a new row can be pre-fetched and, therefore, be ready for computation of the second output row. The resulting RTL implementation has a better effective latency (*9% *improvement) with respect to a baseline with no circular-buffering, but a slightly bigger area (*0.5%*) due to the increased size of the PLM.*

By focusing on the DEBAYER *memory access pattern, other Pareto-optimal implementations can be obtained. Each* 5×5*-interpolation mask is centered on the pixel of interest, corresponding to the output pixel to be computed. Depending on the color and the position of such pixel, the algorithm must read and interpolate 9 to 11 neighboring pixels. By limiting the HLS-driven DSE to use traditional vendor SRAMs, with only two read/write ports, the HLS tool will only be able to schedule up to two concurrent memory accesses to the PLM. Since interpolation is a relatively simple operation, the latency to access local memory cannot be hidden through the data-path. Hence, tuning standard HLS knobs, such as loop unrolling, results in a limited performance span. On the other hand, the area occupation increases significantly as a consequence of requesting the HLS tool to use more computational resources. Conversely, the ESP accelerator-flow integrates the generation of a dedicated memory subsystem based on multiple SRAM banks, which produces the equivalent of multi-port vendor SRAMs. Thanks to multiple concurrent interfaces with the PLM, the HLS tool can better exploit the intrinsic parallelism of* DEBAYER *and cover larger regions of the design-space. The top-left chart in Fig. 4.7 plots the design points obtained for* DEBAYER*. By increasing the number of read ports assigned to the computation phase, several new distinct implementations can be found. These are represented as yellow circles (2 ports) and light-purple triangles (4 ports). Five of these points are Pareto optimal and expand the performance span of the Pareto curve to more than 3X. Note that all design points on the chart for* DEBAYER *are clustered based on the number of ports set for the PLM. The gap, in terms of effective latency, between a cluster and the next one cannot be covered by tuning HLS knobs, proving that memory access is the major factor in determining the performance of a* DEBAYER *accelerator.* □

4.1.1 HLS-Driven Design-Space-Exploration

The example of DEBAYER shows how important it is to account for the interfaces to the PLM, while selecting the appropriate micro-architecture for the datapath of accelerators. HLS-driven DSE becomes even more challenging if considering that all CAD tools are based on heuristics, which result in non-deterministic correlation between knob-settings and scheduled RTL characteristics.

Looking again at the chart for DEBAYER in Fig. 4.7, let us consider the shaded-colored regions of the bi-objective space and their zoomed version on the right. Each color corresponds to a different number of allotted ports to the PLM. The lower-right corner of a shaded region corresponds to the slowest Pareto-optimal point, obtained by setting the HLS knobs to the default values, which do not apply any transformation to the control-data-flow graph. Such design point favors resource sharing over performance, hence they exhibit smaller area occupation, but larger effective latency. Conversely, the top-left corner of a shaded rectangle is obtained by choosing performance-oriented knob-settings, as long as increasing resource allocation keeps returning noticeable improvements. Usually these designs take advantage of the maximum feasible loop unrolling to exploit the parallelism exposed by the given number of PLM interfaces. In practice, this correlation between knob-settings and area-performance trade-offs doesn't always hold. Increasing the number of unrolled iterations in the loops should result in design points that take more area, but have improved latency. Nevertheless, this is not always true, as highlighted by the magnified graphs that show how multiple points fall off the Pareto curve and move to unexpected directions. In fact, some combinations of HLS-knob settings have a bad effect on both latency and area due to the application of the tool heuristics. These non-deterministic behaviors of HLS are observable on most charts of Fig. 4.7. For instance, for the WARP accelerator there are four Pareto-optimal design points and four more points deviating from the curve, even though all eight points are the result of a progressive increment of the number of unrolled loop iterations. Furthermore, the exhaustive DSE in Fig. 4.7 shows other interesting facts that must be taken into account. For instance, both GRAYSCALE and MATRIX-SUBTRACT have peculiar Pareto curves, which can be approximated with a flat horizontal segment and a vertical one. The horizontal portion of the Pareto curve derives from the fact that these algorithms are *(i) embarrassingly parallel* and have a *(ii) sequential* access pattern to data. A sequential access pattern allows for data distribution across SRAM banks by using the least significant bits of the address to select the target bank [Pilato *et al.*, 2014a]. Therefore, duplicating data and storage is not required

to implement a multi-port PLM and the area overhead is limited to the area of the additional SRAM ports. Moreover, the high-degree of exploitable parallelism, allows the HLS tool to perform spatial computation without generating complicated control blocks to orchestrate the functional units in the datapath. As a result, the colored regions are stretched along the latency axis, because improving performance has little impact on area. The magnified charts on the right show the typical trend of Amdahl's law: when the parallelism of the datapath cannot be sustained any longer by the available memory ports, the design starts getting diminishing returns. Only when the PLM is allowed more ports, the design points move to the next design-space region, towards smaller latency.

The vertical segment of the curves, instead, is caused by two different factors. On one hand, Amdahl's law effects have a stronger impact when fewer portions of the computational kernel remain to be scheduled in parallel. On the other hand, the interaction with the system, and in particular the DMA controller, becomes the bottleneck of the application. Such condition is modeled by a templated ESP testbench in the form of base classes that implement the accelerator interface for ESP and implement common platform services, such as DMA and configuration. Fig. 4.6 shows the accelerator encapsulation and the SystemC data types at the interface. Specifically, the drawing shows that DMA occurs at the granularity of 32-bit data words, but this is just an example, as the actual width of the DMA channel is determined by the interconnect of the instance of target instance of ESP. Without considering the access to external memory, the larger the DMA bit-width, the larger the bandwidth the accelerator's input and output phases can sustain. Using 32-bits DMA and pushing HLS-knob settings for performance, the accelerators for GRAYSCALE and MATRIX-SUBTRACT saturate the DMA bandwidth in the design-space region corresponding to 4 PLM ports. By further increasing the number of ports, Amdahl's law also affects significantly the shape of the Pareto curve, as most parallelism has been exploited already. Hence the chart shows a dramatic increase in area occupation for almost no performance improvement in the cases of 8 and 16 ports.

Another interesting scenario is the case of WARP, STEEPEST-DESCENT and RESHAPE. These algorithms are characterized by heavy computational tasks performed on smaller amounts of data. Therefore, augmenting local memory parallelism gives little to no advantage. In addition, WARP initially streams a sequence of data, but then accesses the remaining information based on the result of a partial computation. In this scenario, not only pre-fetching is penalized, but the additional dependencies in the kernel data flow limit the ability of the HLS scheduler to exploit parallelism.

This leads to fewer distinct implementations, most of which are not Pareto optimal, as reported in Fig. 4.7. In the extreme eventuality that the HLS tools fail to obtain a set of Pareto-optimal implementations within an acceptable region of the design-space, it is recommendable to execute that part of the application in software. The latter can take advantage of cache hierarchies to exploit at least some temporal locality, if any. Considering the WAMI-App, this occurs for MATRIX-INVERSION, which is not accelerated in hardware.

Fig. 4.7 shows an exhaustive search that helps understanding the challenges of HLS applied to an ESP accelerator. Nevertheless, it is not recommendable to run exhaustively all possible schedules resulting from combining all available knob-settings, DMA widths and number of PLM interfaces. Component characterization for an ESP accelerator, instead, focuses on finding first the regions that include the Pareto-optimal implementations. This is done by sweeping the number of allotted ports to the PLM and the DMA bit-width with no other HLS optimization to find the lower-right corner of each region. Afterwards, the top-left corner is estimated by pushing HLS-knob settings to achieve maximum performance. Note that finding these corners is not always trivial, especially considering the adversarial non-deterministic behaviors of the CAD tools. As highlighted by the colored regions in Fig. 4.7, sometimes the component characterization misses areas of the design-space that include Pareto-optimal points. However, the approximation of these corners is typically accurate enough to enable application-level and system-level DSE, which are described in the next section.

4.1.2 Compositional Design-Space Exploration

Analytic DSE. The system is modeled with the data-dependency graph, assuming an ideal environment, i.e. capable to sustain the maximum throughput of each accelerator. By applying the integer-linear programming method proposed in [Liu *et al.*, 2012a], the DSE determines which components have most impact on the application overall performance and it estimates the Pareto curve of the application, assuming that the complete set of design points for such critical components is available. Working backwards from the estimated curve for the application, the constraints on effective latency for the components are computed. For every critical accelerator, if there is at least one region of the design space covering a latency range that includes the target latency, HLS is invoked to obtain an actual implementation that satisfies such constraint. Given the knob settings

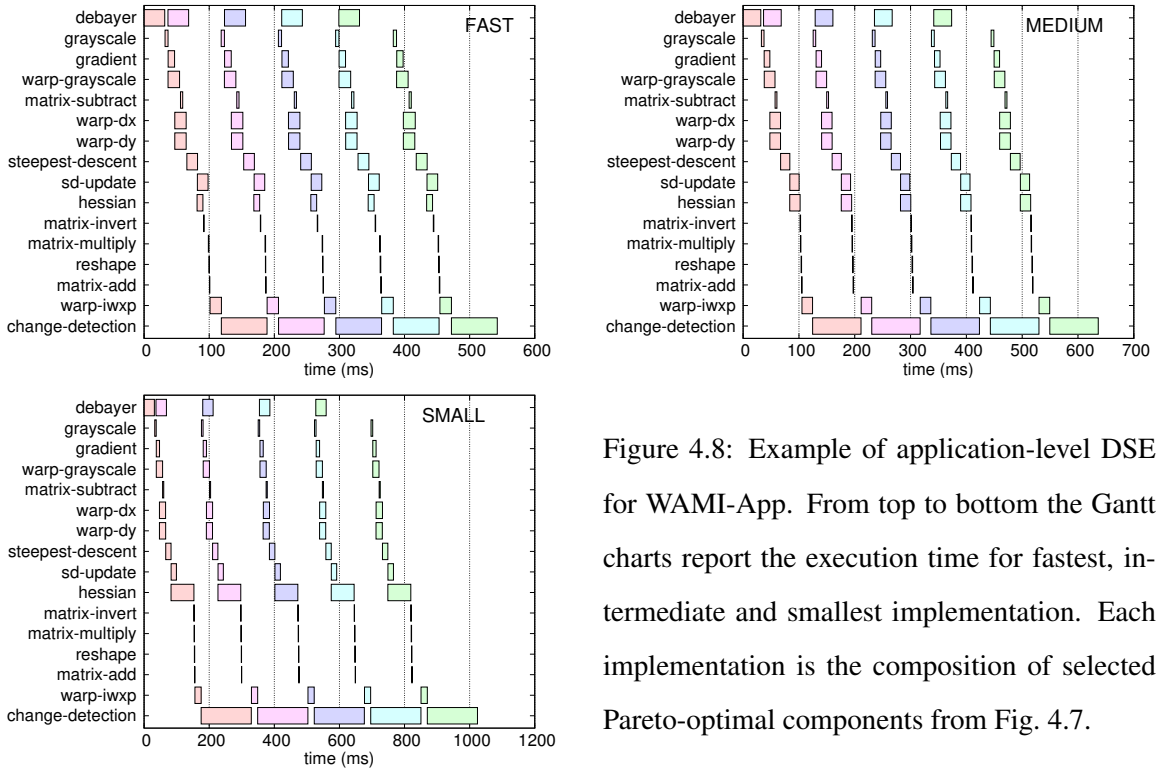


Figure 4.8: Example of application-level DSE for WAMI-App. From top to bottom the Gantt charts report the execution time for fastest, intermediate and smallest implementation. Each implementation is the composition of selected Pareto-optimal components from Fig. 4.7.

corresponding to the corners of a design-space region, HLS knobs are computed by interpolating every parameter. This is done by using the inverse of Amdahl's law as an interpolation function. By excluding non-deterministic behaviors, in fact, the law of diminishing returns applies well to all micro-architectural manipulations done through HLS. Finally, since this flow could hit a non-Pareto-optimal design point, additional scheduling passes are needed to slightly tune HLS-knob settings and search the neighborhood. Conversely, when the system-level constraint for a component falls in a gap where no implementation can be found, the only option is to pick the lower-right corner of the next region. This design point is, indeed, the smallest implementation that satisfies (in general with significant slack) the given constraint. If really needed, DVFS or a slower nominal clock period can help covering gaps in the design space.

Application-Level DSE. The previous step helps designers collect useful implementations for each component. The estimated application-level Pareto curve, however, must be refined by considering other aspects of the actual SoC. The relations among the WAMI-app kernels shown in Fig. 4.2 translates into data dependencies across the corresponding accelerators: e.g., some kernel-level

parallelism can be exploited by supporting multiple concurrent executions of the four WARP kernels. These dependencies are captured with the analytic approach, but without any notion of what enables data exchange and synchronization across accelerators. In a real system, a software application is responsible for allocating necessary buffers in memory, configure all accelerators and orchestrate their runs. To account for the interaction with software, a simulation based on the data- dependency graph (such as the one in Fig. 4.2) is necessary. Hence, the flow for application-level DSE includes a simulator that takes as input the data-dependency graph of the application, the available physical memory to implement data queues for accelerators, and the back-annotated effective latency of each accelerator. The simulator emulates the execution of the application and estimates the waiting time due to interrupt handling and software-thread control. The simulator behaves as an ideal software orchestrator that runs concurrently as many accelerators as possible. This allows designers to properly determine the memory footprint of the real application, which should exploit all the potential of the accelerators. Furthermore, these simulations could potentially reveal that some of the Pareto-optimal implementations are actually not feasible, or non-optimal, at the application level. Fig. 4.8 reports the output of a typical simulation in the form of Gantt charts. For these charts three combinations of Pareto-optimal implementations were selected for each accelerator, thus obtaining the fastest design (labeled `fast`), the smallest one (`small`) and a third one with medium performance and cost (`medium`). In each chart, a bar corresponds to a time interval where a software thread is active. For example, the first line of each chart shows the active time intervals for the thread controlling DEBAYER. The color of the bars identifies the frame number (1 to 5) processed by the accelerators. Note that the active interval includes the waiting time for the accelerator interrupts, as well as additional waiting time due to software-thread control. Analyzing the output of this simulation, gives additional insights on how accelerators interact with each other and with software. For instance, the bars on the Gantt charts for the `medium` and the `small` implementations show similar parallelism, because CHANGE-DETECTION dominates the execution time. Even if the accelerators process five independent frames, they can only proceed when the allocated memory queues have space to store their temporary results and outputs. As a result, in this case, the `medium` scenario represents a better trade-off than the `fast` one.

Fig. 4.9 reports the exhaustive search at the application-level, obtained by simulating all combinations of Pareto-optimal implementations from the component DSE. The colored triangles identify

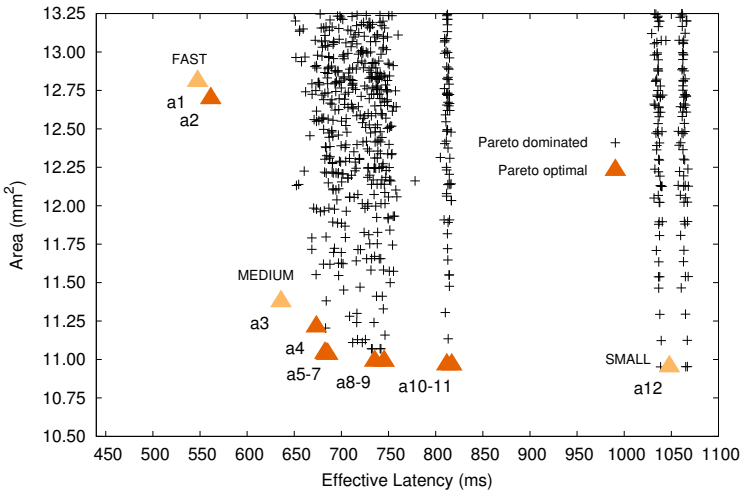


Figure 4.9: Application-level DSE for WAMI-APP.

Pareto-optimal points for the application, while the black crosses represent Pareto-dominated designs. Note that some of these are omitted because the axis ranges for area and effective latency are set to zoom on the Pareto curve. Further, it is not recommended to simulate all combinations, which are only reported for the sake of completeness. Instead, only the combinations recommended in the analytic step should be simulated to filter out non-optimal points and refine the metrics for the remaining designs in the Pareto set. While analyzing the chart in Fig. 4.9, it is worth mentioning that the fastest implementation (*a1*) is not the result of composing the fastest implementations for each accelerator; in fact, only HESSIAN, CHANGE-DETECTION and WARP fastest implementations are included in the design point *a1*. Intuitively, this corresponds to accelerate as much as possible the application bottlenecks, while keeping other accelerators, such as DEBAYER and GRADIENT smaller, in order to save some area.

System-Level DSE. With the available information at this stage, the compositional DSE still ignores contention for shared resources as well as the overhead of the interconnect and other common system components, such as the DMA controller. In the ideal scenario of simulation, each accelerator starts its computation phase as soon as the predecessors have completed theirs. Conversely, deploying the accelerators on FPGA, as part of an ESP instance, enables a further refinement of the Pareto set, accounting for the non-deterministic effects of the operating system and DRAM access, all overheads, as well as resource contention. Furthermore, ESP flexibility can be leveraged not

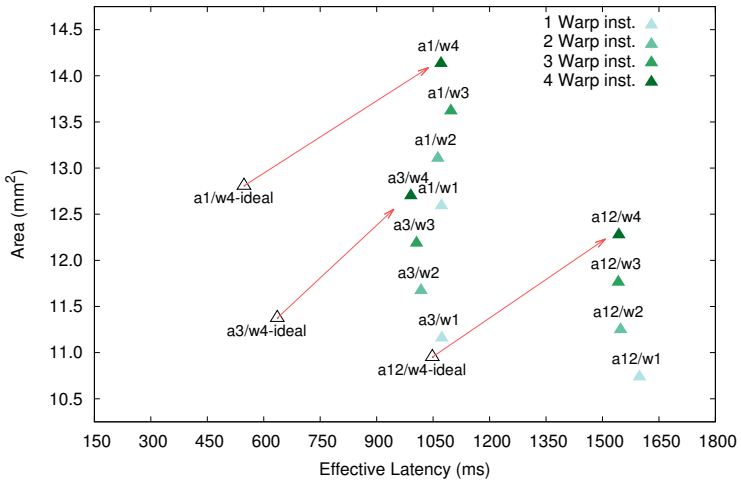


Figure 4.10: Pareto-point migration from ideal to full-system scenario.

only to select the desired mix of implementations among components, but also to pick the number of instances per accelerator. For example, WAMI-App invokes WARP four times within the LUCAS-KANADE loop and it is part of the DSE effort to decide whether to allocate an instance of the WARP accelerator for each occurrence in the dependency graph, or to share in time fewer instances of this accelerator.

Generating the actual SoC implementation is the final step of the ESP methodology in Fig. 4.1. The case study presented here consists of twelve SoCs for WAMI-app: each SoC is an ESP instance featuring one processor tile, two I/O tiles connected to DRAM banks, a multi-plane NoC, and a set of twelve to fifteen accelerator tiles. Each accelerator tile maps to one kernel of WAMI-app except from MATRIX INVERSION, which is executed in software. The target FPGA for the experiment is a `Xilinx Virtex7` with clock frequency set to 100 MHz.

Fig. 4.10 plots the results of deploying the application-level Pareto-optimal designs (*a1*), (*a3*) and (*a12*). Each design point in Fig 4.10 is also labeled with the number of WARP instances available (*w1* to *w4*). The first thing to notice is that all points move further away from the origin in the bi-objective optimization space. After integration, in fact, every component is encapsulated into an ESP tile and the area of a tile is the result of adding the area of the accelerator with the necessary wrappers and platform-service components. Latency also increases because it is measured by the software application, which is equivalent to record the actual latency experienced by the user.

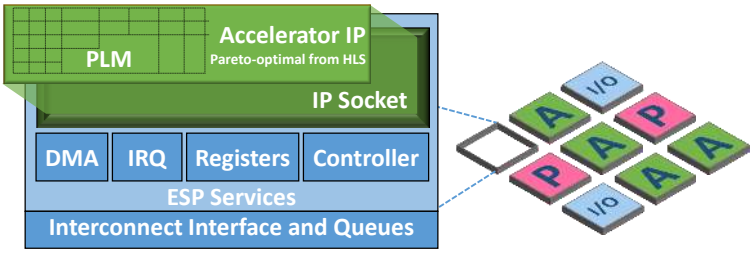


Figure 4.11: Configurable ESP accelerator tile with hardware socket.

While looking at the chart, notice that the design point (*a1/w4*) has both larger area and effective latency, with respect to all the other translated points. This leads to conclude that such implementation is actually not Pareto optimal at the system level. Finally, results from system-level DSE in Fig. 4.10 demonstrate that instantiating more than one accelerator for WARP has little impact on the overall performance. In this particular case, resource contention in accessing the external memory limits the parallelism exposed by multiple instances of WARP, thus suggesting that a single instance is a better trade-off between area and performance.

4.2 ESP Socketed Architecture

HLS enables quick tuning of accelerator IP blocks in isolation. In addition, application-level analysis can be performed by combining the results from each accelerator with a dependency graph. As shown with the example of system-level DSE for the WAMI-App, however, the effective cost and performance of a design can only be estimated by considering the interaction across multiple accelerators, the interconnect, memory and software. The combination of the architecture and methodology of Embedded Scalable Platforms enables fast system-level DSE because it simplifies the integration of heterogeneous IP blocks into an SoC. In particular, I designed the ESP tile-based architecture to strike the right balance between heterogeneity and regularity. It consists of a set of templated hardware and software sockets that enable the generation and programming of a complete SoC by assembling a configurable infrastructure with off-the-shelf processors and accelerators. The latter are designed following the flow described above in Section 4.1.

ESP Accelerator Sockets. A configurable tile, whose high-level view is shown in Fig. 4.11, supports accelerators' execution. An ESP tile can be easily customized to encapsulate a given acceler-

ator and decouple its design and optimization from the rest of the system. An accelerator tile hosts a hardware socket, whose signal-level interface matches the accelerator interface described in Section 4.1. The socket interface exposes the following platform services that provide access to system resources.

- Input read and output write requests from the accelerator; these are in the form of DMA bursts with configurable bit-widths. The burst length and accelerator virtual address are set at run-time by `load_input()` and `store_output()` SystemC threads. The socket implements a latency-insensitive protocol [Carloni *et al.*, 2001] matching the behavior of the TLM point-to-point channels used during the accelerator design. Relaxing interface requirements with a latency-insensitive protocol enables a seamless replacement of an IP implementation with any other alternative one taken from its Pareto-optimal set [Carloni, 2015].

- A set of common configuration registers is used to activate the DMA engine, so that every load and store request issued by the accelerator is served without processor intervention. Start or reset commands and error condition checks are mapped to the ESP command register and status register, respectively. In addition, a set of optional registers allows for dynamic voltage and frequency scaling if the accelerator runs on a dedicated voltage and clock domain. When frequency scaling is enabled, the ESP tile is automatically configured to include dual-clock FIFOs at the interface with the interconnect. Besides the common command and status registers, all user-defined registers and their memory mapping are accelerator specific and are generated based on the data structure `conf_info` defined by the interface of Fig.4.6.

- Interrupt notifications; the accelerator triggers an interrupt request when it completes or in case of error.

The ESP hardware sockets implement the TLM abstraction used during the HLS of the accelerators. In this way the IP designer can be completely unaware of the SoC interconnect specifications. Transactions are translated into platform-dependent messages directed to either a processor or an I/O tile. Similarly, all environment requests are forwarded to the accelerator according to the interface specified in Fig. 4.6.

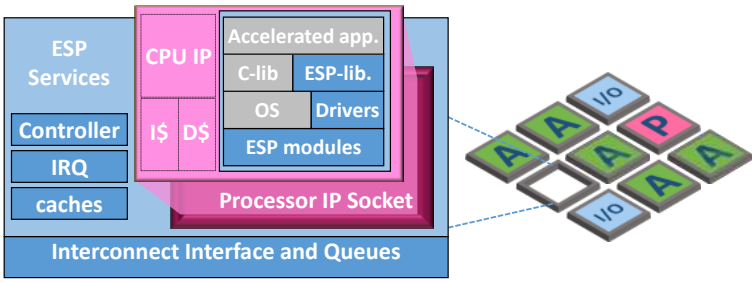


Figure 4.12: ESP Processor tile with software socket.

ESP Processor Sockets. ESP eases the process of integrating the accelerators in a heterogeneous SoC by pairing hardware sockets with software sockets running on processor. Similarly to accelerators, a processor is also encapsulated in a tile which implements the ESP platform services. In this case, the DMA engine is replaced by a cache, which gives the illusion of a traditional homogeneous system and decouples the processor bus from the rest of the SoC. Hence, legacy software can transparently execute on the processor. Similarly to the accelerator tile, a set of optional registers can be instantiated to enable DVFS on the processor. The processor socket is completed by three software layers as shown in Fig. 4.12.

- *ESP-core Linux driver.* This low-level software allows Linux to recognize the accelerators in the ESP tiles and manage memory allocation. It implements interrupt registration/handling and primitives to configure all ESP common registers with one `io_ctl` system call. The modules relieve the IP designer from writing complex low-level routines for each accelerator.

- *Linux device drivers for accelerators.* Since ESP accelerators are seen by the operating system like any other peripheral, they need a device driver. The use of ESP template drivers requires the programmer only to implement the behavior of user-defined control registers. Lower-level routines, in fact, are provided by ESP modules. For instance, across all accelerators for WAMI-App, the "accelerator specific" code that is user-provided represents on average less than 2% of the entire device driver.

- *ESP user-level library.* This library has two main purposes: (1) it implements an API to invoke accelerators within user-level applications and (2) it provides a multi-threaded infrastructure to perform a DSE of multi-accelerator applications. For instance, an application

```c
/* esp.c (Linux module esp.ko) */
/* ... */
static int esp_access_ioctl(struct esp_device *esp,
           struct esp_file *file, void __user *arg)
{
  struct esp_access desc; int rc;
  if (copy_from_user(&desc, arg, sizeof(desc))) return -EFAULT;
  /* Check if configuration is legal (accelerator specific) */
  if (!esp_access_ok(esp, &desc)) return -EINVAL;
  /* Reserve the accelerator */
  if (mutex_lock_interruptible(&esp->lock)) return -EINTR;
  /* Initialize completion state variable */
  INIT_COMPLETION(esp->completion);
  /* Configure registers: accelerator specific section BEGIN */
  iowrite32(file->dma_handle, esp->iomem + ESP_REG_SRC);
  iowrite32(desc->size, esp->iomem + ESP_REG_SIZE);
  iowrite32(desc->mode, esp->iomem + ESP_REG_MODE);
  /* Start the accelerator */
  iowrite32(0x1,esp->iomem + ESP_REG_CMD);
  /* accelerator specific section END */
  /* Sleep and wait for completion */
  rc =  wait_for_completion_interruptible(&esp->completion);
  /* Release the accelerator */
  mutex_unlock(&esp->lock);
  return rc;
}
/* ... */
/* main.c (user space application) */
void main(int argc, char **argv)
{
  /* ... */
  /* Open device and initialize memory */
  *fd = open(device_name, O_RDWR, 0);
  contig_alloc(policy, size, mem);
  init_buf(hw_buf, data, buf_size);
  /* Configure accelerator descriptor and make a system call to run */
  desc.size = size;
  desc.mode = mode;
  if (ioctl(fd, MY_IP_IOC_ACCESS, &desc) < 0) perror("ioctl");
  /* ... */
}
```

Figure 4.13: Snippet from core ESP driver and corresponding `syscall` in user-space.

like WAMI-App consists of multiple kernels, each potentially implemented by an accelerator. Thanks to the library, each accelerator is controlled by a `Pthread` and a hidden queue-based mechanism synchronizes its execution.

4.3 ESP Software Stack

The combination of ESP hardware and software sockets allows designers to bring up a system in a very short time and to perform a DSE from a system-level viewpoint. Similarly to TLM, which decouples computation from communication, the ESP software library decouples accelerators management from the application data-flow.

Single Accelerator. Let us now dive into the details of the ESP software stack starting from the simple scenario of an application that invokes only one accelerator. Fig. 4.13 shows a snippet of

code taken from the core ESP driver, which should be used by any ESP accelerator. Specifically, this is the implementation of the common portion of the `io_ctl` system call to run an accelerator. The driver copies the device descriptor from user space, collects configuration parameters and performs a sanity check to make sure the configuration is legal. Afterwards, the driver initializes a waiting queue, which is used to pause the calling thread until the accelerator raises an interrupt. Finally, after the configuration parameters are written to the hardware socket registers, the accelerator starts running and the thread is suspended. Note that any accelerator-specific register should be configured in the device-dependent templated driver. For each register, the user should specify an entry in the device descriptor and perform a corresponding `iowrite32` call. The user-defined portion of the `io_ctl` call runs before the code in Fig. 4.13 so that the accelerator is fully configured before writing to the command register.

A few system calls are sufficient to interact with the device driver from user space. The `main` function at the bottom of Fig. 4.13 shows how to do so. First, a function call to `open` returns a handle for configuration. Then, by using a memory allocator, a buffer is initialized in memory for DMA transfers. This example includes the call to the ESP-specific allocator that is designed to improve the accelerator performance, as explained later in Chapter 5. With the pointer to the DMA buffer, the application code can create or copy the input data. Finally, a system call to `io_ctl` locks, configures and runs the accelerator. When the registered interrupt line is activated, the thread resumes, the lock on the accelerator is released and the accelerator status is returned to the calling application.

Multiple Concurrent Accelerator. To build an ESP application running multiple concurrent accelerators, such as WAMI, a designer can leverage the ESP `kernel-thread` software library. Using the ESP library, requires to consider that a kernel of execution corresponds to a task in the data-dependency graph, which may or may not map to an accelerator. Furthermore, since more than one kernel could map to the same accelerator, the number of kernels is always larger or equal to the number of available accelerators. For instance, in the case of WAMI-App there are a total of 16 kernels, of which 15 are accelerated and 4 may share the same accelerator instance. Remember, in fact, that during the component DSE MATRIX-INVERT was chosen to be executed as software kernel. Also, recall that as an example of system-level DSE, the count of WARP instances was swept from 4 down to 1.

```
typedef struct kthread_t
{
  pthread_t          handle;
  pthread_mutex_t    mutex;            /* avoid race conditions                                  */
  pthread_cond_t     ready;            /* waiting condition is satisfied                         */
  unsigned           parents;          /* number of threads that prouce an input                 */
  struct kthread_t **parent_threads;   /* pointers to parent threads                             */
  bool               *feedback;        /* not forward / feedback dependency                      */
  bool               reset;            /* true if this is the root kernel in the dep. graph      */
  unsigned           iter_max;         /* maximum number of iterations                           */
  unsigned           iter_count;       /* iteration counter                                      */
  fifo_t             fifo;             /* kernel FIFO                                            */
  /* ... */                            /* Other helper variables                                 */
} kthread_t;

void lock_kernel(...);                 /* acquire mutex lock                                     */
void unlock_kernel(...);               /* release mutex lock                                     */
void enable_feedback(...);             /* mark a thread as a feedback                            */
void disable_feedback(...);            /* unmark a feedback thread                               */
unsigned is_feedback(...);             /* check if a thread is a feedback                        */
unsigned can_put(...);                 /* return first available slot (may sleep if full)        */
void put(...);                         /* reserve available slot for accelerator (push)          */
unsigned can_get(...);                 /* return head of the queu (may sleep if empty)           */
void get(...);                         /* consume head of the queue (pop if no more consumers)   */
void* thread_function(...);            /* Check I/O queues and runs accelerator-specific code    */
/* ... */                              /* Other thread helpers...                                */
```

Figure 4.14: API of the ESP `kernel-thread` library.

The main data structure and the basic API of the `kernel-thread` library is reported in the code snippet of Fig. 4.14.

- The handle to a corresponding `pthread` is used to enable parallelism.
- The `mutex` is used by the API functions `lock/unlock_kernel`, while accessing information about the status of the kernel input and output queues. These queues implement the directed arcs of the dependency graph, but avoiding data duplication, therefore some queues are shared among multiple kernels, when the fan-in or fan-out of some of them is different from one.
- The flag `ready` is used as a trigger for the waiting condition to wake up threads waiting for a kernel to complete.
- The number of predecessors, or `parents`, in the dependency graph and the pointer to such kernel-threads are used to determine which queues must be peeked before executing the kernel.
- The array of Boolean flags `feedback` contains true entries if the corresponding parent threads implement a feedback path in the dependency graph. These flags can be accessed through the API function `is_feedback` and allow a kernel to ignore the corresponding input queue if the

```
void* thread_function(void* kernel_ptr)
{
  kthread_t* kernel = (kthread_t*) kernel_ptr;
  /* ... */
  /* Thread lock and synchronization */
  /* ... */
  /* Wait for reset */
  pthread_mutex_lock(&kernel->mutex);
  while (!kernel->reset) pthread_cond_wait(&kernel->ready, &kernel->mutex);
  pthread_mutex_unlock(&kernel->mutex);
  /* Run accelerator when I/O queues allow it */
  while (1)
  {
    /* Termination condition */
    if (kernel->iter_count >= kernel->iter_max) break;
    /* Check if input is available and output queue is free */
    for (index = 0; index < kernel->parents; index++) {
      kthread_t* source = kernel->parent_threads[index];
      kernel->fifo_get_index[index] = can_get(source, kernel, index);
    }
    kernel->fifo_put_index = can_put(kernel);
    /* ... */
    /* Accelerator-specific: configure and run */
    gettime(&t_start); // Thread local time
    kernel->run_hw(kernel_ptr);
    /* Consume input data */
    for (index = 0; index < kernel->parents; index++) {
      kthread_t* source = kernel->parent_threads[index];
      get(source, kernel, index);
    }
    /* Mark output slot as valid (push) */
    put(kernel);
    gettime(&t_start); // Thread local time
    ++kernel->iter_count;
  }
  return NULL;
}
```

Figure 4.15: Snippet from the ESP multi-threaded library. The kernel-thread main function provides synchronization with memory I/O queues and calls the accelerator-specific configuration function.

`disable_feedback` function has been called. The feedback is ignored until the next call of `enable_feedback`. These API functions are typically useful during the first iteration of an application, when inputs from the feedback paths are not available and reading a feedback queue would lead to a deadlock condition.

- The flag `reset` tells whether the kernel is the root of the dependency graph.
- The maximum number of iterations `iter_max` and the iteration counter `iter_count` are used to determine when the thread has processed all available inputs and should terminate.
- The queue `fifo` is a synchronization data structure that stores information about the current state of the output memory buffer. Each kernel owns an output queue of configurable, but finite size, which holds the output of one or more runs of the kernel. This queue can be peeked through the API by calling `can_get` to see if there are valid data at the head of the

```
void run_warp_dy_hw(void *ptr)
{
  struct kthread_t * thread_data = (struct kthread_t*) ptr;
  /* ... */
  d_warp_dy.esp.contig   = contig_handle(contig_wami_app); /* pointer to allocated memory */
  d_warp_dy.esp.run      = true;                           /* Run accelerator             */
  d_warp_dy.offset_dst   = queue_dst_offset;               /* Offset to out queue         */
  d_warp_dy.offset_src_1 = queue_src_1_offset;             /* Offset to first in queue    */
  d_warp_dy.offset_src_2 = queue_src_1_offset;             /* Offset to second in queue   */
  d_warp_dy.n_frames     = N_FRAMES;                       /* Number of frames            */
  d_warp_dy.n_cols       = N_COLS;
  d_warp_dy.n_rows       = N_ROWS;
  d_warp_dy.pad          = N_PAD;
  /* ... */                                               /* Other accelerator parameter  */
  gettime(&ts_start); // Accelerator execution time (including device driver)
  rc = ioctl(fd_warp_dy, WAMI_APP_IOC_ACCESS, desc_warp_dy);
  gettime(&ts_end);
  /*  ... Validation and Profiling may occur here during debugging ... */
}
```

Figure 4.16: Snippet from the accelerator-specific code to invoke one instance of WARP.

queue or `can_put` to know whether there is available memory to store the output of the computation. Once the execution of the kernel has completed, the state of the queues can be updated with `get` if the input data are no longer needed and `put` if output data are available for the successors in the dependency graph. Calling `put` is equivalent to perform a push to the queue. Conversely, calling `get` may result in a pop from the queue, but only when all kernels depending on the queue have consumed the data at the head of it.

This API provides the user with direct access to the queues, thus enabling the implementation of a custom synchronization protocol. In most cases, however, the default kernel-thread behavior implemented in the ESP library can be adopted. The function `thread_function` in Fig. 4.15 shows how queues should be accessed. At every iteration the thread should check whether input queues have new valid data (`can_get`) or not and if there is available memory to store the output of the computation (`can_put`). Should any check fail, the thread will be suspended before the function call returns because no useful computation can be done until the state of the queues changes. When all checks are successful, the kernel can be executed by calling the application-specific function. The latter should be implemented by the user and can include the system calls to run an accelerator. For instance, Fig. 4.16 shows a possible implementation for the kernel `warp_dy` of the WAMI-App invoking the WARP accelerator. The descriptor in the example contains the base address to the memory buffer allocated for the entire WAMI-App and the offsets to source and destination queues. In addition, there are accelerator-specific parameters, such as the number of frames to process and their size in term of pixel rows and columns. Similarly to the single-accelerator scenario, this

```
/* Declare data structures */
static kthread_t kthreads[KERNELS_NUM];
static contig_handle_t contig_wami_app;
/* ... */
struct wami_app_descriptor d_warp_dy;
/* ... */
/* WAMI-App. main() */
int main(int argc, char **argv)
{
  /* Initialize application */
  /* Open devices and allocate memory */
  open_devices();
  contig_alloc(policy, size, &contig_wami_app);
  /* Initialize input (application specific) */
  fread(&contig_wami_app, input_size, 1, input_file);
  /* ... */
  /* Topology */
  /* ... */
  kthreads[WARP_DY_ID].id = "warp_dy";
  kthreads[WARP_DY_ID].run_hw = &run_warp_dy_hw;
  kthreads[WARP_DY_ID].fifo.capacity = QUEUE_CAPACITY;
  kthreads[WARP_DY_ID].fifo.consumers = 1; /* steepest_descent */
  kthreads[WARP_DY_ID].parents = 2; /* gradient, add */
  kthreads[WARP_DY_ID].parent_threads = (struct kthread_t*[])
            { &kthreads[GRADIENT_ID], &kthreads[MATRIX_ADD_ID] };
  kthreads[WARP_DY_ID].feedback = (bool[]) {0, 1};
  /* ... */
  /* Prepare to run: initialize counters, mutexes and conditional variables */
  /* ... */
  /* create pthreads */
  for (i = 0; i < KERNELS_NUM; i++)
    pthread_create(&kthreads[i].handle, NULL, thread_function, &kthreads[i]);
  /* ... */
  /* Run threads */
  gettime(&t_start); // Global application time
  start(kthreads, 0);
  /* wait and dispose of all of the threads */
  for (i = 0; i < KERNELS_NUM; i++)
    pthread_join(kthreads[i].handle, NULL);
  wrap_up(kthreads, KERNELS_NUM);
  gettime(&t_end); // Global application time
  /* Finalize application, gather reports and free memory */
}
```

Figure 4.17: Snippet from WAMI-App user-space code: device open, memory allocation, kernel-threads initialization and execution.

function calls `io_ctl` to start the accelerator. Note that device open and memory allocation should occur before launching the kernel threads and are not included in the function of Fig. 4.16.

With the ESP `kernel-thread` library described above it is possible to build a multi-kernel application and run concurrently several accelerators. At this stage it is also appropriate to set the size of memory queues, as part of the system-level DSE. The size of the queues has a direct impact on the parallelism that can be exploited by the multi-threaded ESP application. In general, larger queues enable more concurrency among the accelerators. In practice, power caps, contention for shared resources, data dependencies, and diversity in the accelerator execution times may impose unexpected limitations to the benefits of parallelism.

The snippet of code in Fig. 4.17 shows what to add to the user application. First, a call to `open` for each accelerator, initializes the device-drivers that the application will use (i.e. the drivers for all

twelve accelerators of WAMI). Then, a buffer in DRAM is reserved, similarly to what is done for the case of a single accelerator. Since the WAMI accelerators share several input data and a training set, allocating a single buffer for the entire application is advisable to reduce the total memory footprint. In other conditions, users may prefer to allocate separate buffers. The buffer initialization is an application-specific operation which, in this case, corresponds to reading an input file through a call to `fread`. The details on how accelerators access this buffer are given in Chapter 5.

Next, all threads are configured by specifying the kernel name, a pointer to the kernel-specific function, the capacity of the output queue and its fan-out (i.e. the number of consumers), the parent threads and which of these threads is on a feedback path. After configuration, threads are spawned and the computation begins. Each thread executes and invokes its accelerator when all of its input queues have received at least one set of valid data, the output queue has room to store the result, and the requested device is not already in use.

Once all threads reach the termination condition, the parent process disposes of them by calling `pthread_join` and `wrap_up` to de-allocate the data structures. Profiling, reporting and error checking may happen at this point according to the specifics of the user application.

Chapter 5

Handling Memory in ESP

The previous chapter highlighted the importance of dedicated and highly-customized *Private Local Memory (PLM)* for accelerators, while describing the model of loosely-coupled accelerators, which leverage DMA to fetch and store data in DRAM. The PLM itself is key to achieving high data-processing throughput: by integrating many independent SRAM banks whose ports can sustain multiple memory operations per cycle, it enables concurrent accesses from both the highly-parallelized logic of the accelerator datapath and the DMA interface to main memory. Not surprisingly, recent studies confirm the importance of the PLM, which occupies 40 to 90% of the accelerator area [Cota *et al.*, 2015; Lyons *et al.*, 2012] and contributes, together with the processors' caches, to the growing fraction of chip area dedicated to memory. Nevertheless, despite this trend, the data-set sizes of embedded applications are increasing much faster than the available on-chip memory.

As an example, Fig. 5.1 illustrates the growing gap between the aggregate size of the SoC on-chip caches (accounting for L1 and L2 caches, and, starting with iPhone 5s, a 4MB L3 cache) and DRAM size, across eight generations of the Apple iPhone. Thanks to Moore's Law, the cache size has grown by a factor of 228×, from 32KB to 7,296KB. In the meantime, the DRAM size has grown only by a factor of 16×. Still, Relative to the first product generation, the difference between the two sizes has grown by a factor of 16×, to reach almost 2GB[1]. The growth in DRAM size reflects the need for supporting applications with increasingly large footprints, which pose new

[1] Admittedly, the cache numbers do not include the aggregate sizes of the PLMs of the Apple SoC accelerators, which are likely to be large but are not publicly known. Still, even assuming that their contributions could double or triple the reported figures, the on-chip memory sizes would remain very small compared to DRAM.

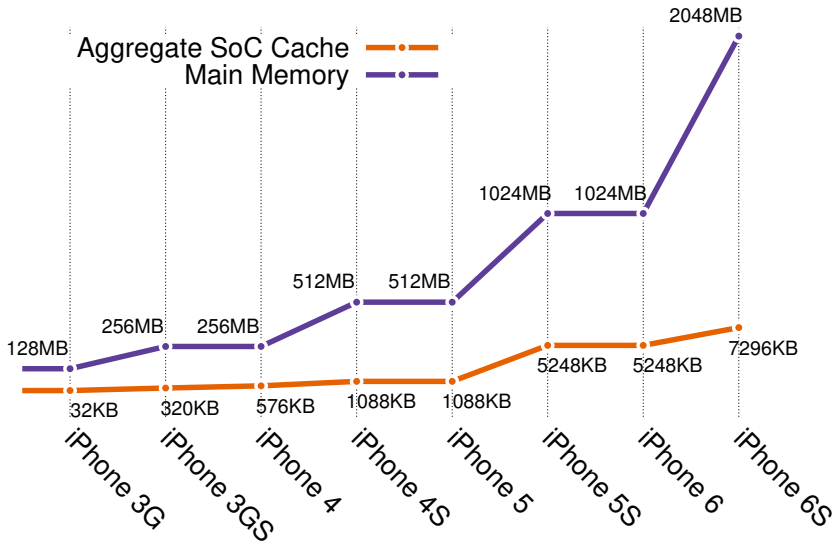


Figure 5.1: The growing gap between the aggregate size of the SoC on-chip caches and the main-memory size, across seven years of Apple iPhone products.

challenges for high-throughput accelerators. Differently from microprocessors, in fact, accelerators cannot rely on a cache-based memory hierarchy for their specific purposes. In order to achieve high performance, an accelerator must be able to continuously access data in parallel from its highly-customized PLM [Cota *et al.*, 2015].

5.1 The Large Data Set Problem

A loosely-coupled accelerator has a twofold nature: while it is similar to an on-board peripheral, in that the processor core can offload a specific task to it, the accelerator does not have a large and private storage system (e.g. a dedicated off-chip memory), and therefore shares the external memory with the processor core. At the same time, the accelerator's computation modules are unaware of the physical memory allocation, which can be even on multiple physically-separated DRAM banks [Yang *et al.*, 2016]. Thus, an accelerator can be likened to a software thread, where physical memory is abstracted by virtual memory and multiple levels of caches. In an accelerator, the PLM gives the illusion of a contiguous address space, allowing the accelerator to perform concurrent random accesses on data structures. This contiguous address space, however, is limited by the size

of the PLM, and processing large data sets necessarily involves multiple data transfers between DRAM and PLM.

I define *the Large Data Set (LDS) Problem for SoC Accelerators* as the problem of finding a high-performance and low-overhead mechanism that allows hardware accelerators to process large data sets without incurring penalties for data transfers. A possible solution to the LDS Problem is to have the accelerators share the virtual address space of the processor in a fully coherent way. This is obtained by replacing the PLM with a standard private L1-cache and sharing the higher levels of the memory hierarchy and the memory-management unit with general-purpose cores [Yang *et al.*, 2016]. This approach, however, is not effective for high-throughput accelerators because it degrades their performance by depriving them from their customized PLMs. Moreover, as the data set grows, the overhead of maintaining coherence further limits the accelerator speedup over software [Benson *et al.*, 2012; Qadeer *et al.*, 2013]. Alternatively, one could expose the PLM to the processor and let it manage data transfers across separate address spaces in suitable small chunks [Komuravelli *et al.*, 2015]. However, as shown in Section 5.1.1, this increases software complexity and forces accelerators to stall while waiting for the software-managed transfers, thus wasting most of the speedup offered by the dedicated hardware.

The memory access platform service for accelerators in ESP is designed to solve to the LDS Problem, while avoiding most common shortcomings of accelerators coupled with embedded processors. The solution is a combination of the following hardware and software features:

- a low-overhead *accelerator virtual address space*, which is distinct from the processor virtual address space, to reduce the processor-accelerator interaction;
- direct sharing of physical memory across processors and accelerators to avoid redundant copies of data;
- a dedicated DMA controller with specialized translation-lookaside buffer (TLB) per accelerator to support many heterogeneous accelerators coexisting in the same SoC, each with its specific memory-access pattern;
- hardware and software support for implementing run-time policies to balance traffic among available DRAM channels.

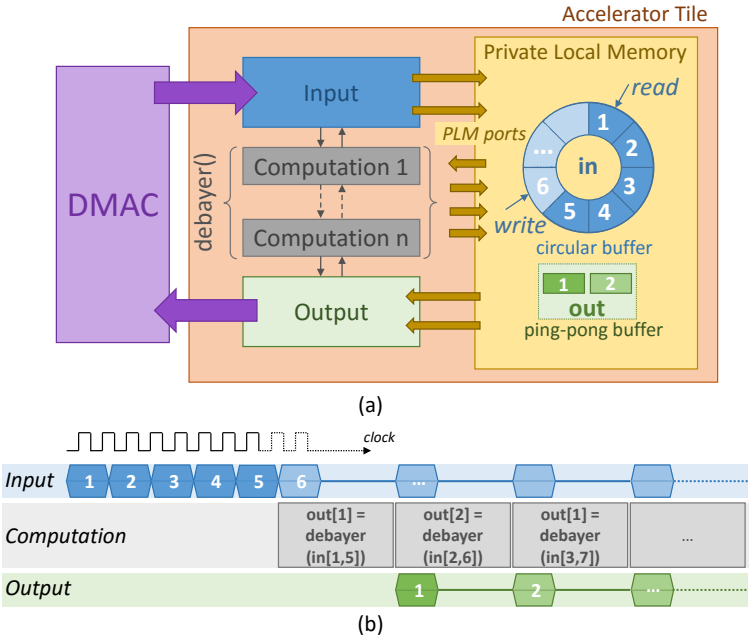


Figure 5.2: (a) The DEBAYER accelerator structure. (b) Overlapping of computation and communication. each I/O burst is marked with the number of the transferred rows of an image, while each computation step shows the rows on which the accelerator operates. The behavior of the main components is shown as a waveform: it alternates I/O bursts and computation.

5.1.1 Preserving Accelerators' Speedup

A loosely-coupled accelerator executes very efficiently as long as it keeps its computation and communication phases balanced. As an example, Fig. 5.2(a) shows the high-level block diagram of a high-throughput accelerator for the DEBAYER kernel [Barker *et al.*, 2013]. Recall from Section 4.1 that this kernel takes as input a Bayer-array image with one color sample per pixel and returns an image with three-color samples (red, green and blue) per pixel, where the missing colors are estimated via interpolation. The accelerator consists of a *load* module (to fetch data from DRAM), one or more *computation* modules, and a *store* module (to send results to DRAM). These modules communicate through a PLM, which is composed of multiple banks and ports. Such PLM architecture allows the computational modules to process multiple pixels per clock cycle. Additionally, circular and ping-pong data buffers support the pipelining of computation and DMA transfers with the off-chip DRAM. This choice derives directly from the functional specification of the kernel: the

DEBAYER interpolates pixels row-by-row and uses 5×5-interpolation masks centered on the pixel of interest. To start the computation, the accelerator needs at least the first five rows of the input image in the circular buffer (input bursts from 1 to 5 in Fig. 5.2(b)). Then, while the computation modules run, the input module can prefetch more rows for future processing (input burst 6 in Fig. 5.2(b)). As soon as a computation step completes, an interpolated row is stored in the first half of the ping-pong buffer so that it can be transferred back to DRAM (output burst 1). Meanwhile, the computation modules can start processing the additional row in the circular buffer and storing the result in the second half of the ping-pong buffer (output burst 2). This behavior represents well many high-throughput accelerators. However, the specifics of the micro-architecture of any given accelerator, including the PLM organization, may vary considerably depending on the particular computation kernel. The timing diagram in Fig. 5.2(b) shows a hypothetical scenario, where the communication (i.e. input and output) and computation phases are overlapping, and the latency of DMA transfers is hidden by the local buffers. Intuitively, if such latency becomes larger than processing time, then the accelerator must be stalled until new data are available for computation. This can limit the efficiency of the accelerator, reducing its advantages over software execution.

The experiment presented next demonstrates that, when loosely-coupled accelerators process large data sets, traditional memory handling for non-coherent devices leads to such undesirable scenario. This example consists of an ESP instance, implemented on FPGA, that integrates one embedded processor with a 32-bit architecture, which runs the Linux Operating System, and two loosely-coupled accelerators. These two accelerators implement the DEBAYER and SORT computational kernels [Barker *et al.*, 2013]. The virtual memory available to user-level applications is 3GB, while the actual physical memory is 1GB. For this experiment let us considered a memory footprint of 32MB for DEBAYER, which elaborates one 2048×2048-pixel Bayer-array and the corresponding bitmap image (16-bit colors), and of 4MB for SORT, which processes in place 1024 vectors each containing 1024 single-precision floating point numbers. The two accelerated applications share the processor in time multiplexing according to the Linux scheduler. Each of them can invoke the appropriate accelerator through the ESP device driver presented in Section 4.3.

Fig. 5.3 shows the memory layout of one application: the physical memory is usually allocated in 4KB pages and remapped to a contiguous virtual memory area where the program stores the application's data. To allow a non-coherent device to access these data, the driver typically im-

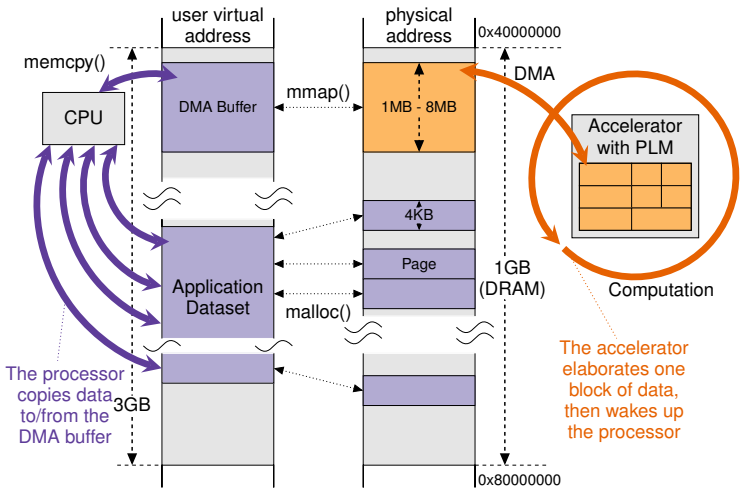


Figure 5.3: Traditional software-managed DMA.

plements a memory-mapping function that serves three main tasks. First, it requests the operating system to reserve a contiguous area in physical memory for DMA and pins the corresponding pages (orange-shaded memory area in Fig. 5.3). Then, it passes the physical address to the device, referred to as `dma_handle` of the allocated buffer. Finally, it remaps the *DMA buffer* to the virtual memory (purple-shaded area) and returns a pointer to the user-level application. The exact amount of contiguous memory that the operating system can allocate depends on the target processor architecture, but it is usually limited to a few megabytes. When setting the size of the DMA buffer to 1MB, the DEBAYER computation can be easily split into 32 parts, each processing a different portion of the input image. Similarly the input vectors for SORT can be divided into 4 sets of 256 vectors each.

The bars of Fig. 5.4 show how hardware acceleration (orange) and software execution (purple) interleave over time. The orange segments include the time for fetching the input data from DRAM via DMA, elaborating them, and transferring results back to DRAM also via DMA. The purple segments, instead, correspond to the time spent by the user application in saving results from the DMA buffer to another virtual memory area and copying the next block of input data into the DMA buffer. Note that the first and the last segments of each bar are always orange, because the application setup and wrap-up phases, which take constant time across all scenarios, are omitted. The experiment was repeated four times varying the size of the DMA buffer from 1MB up to 8MB. As the size of the DMA buffer increases, the data processed by the accelerators are split into fewer blocks and the overhead of interleaving hardware and software decreases. Further, the execution of

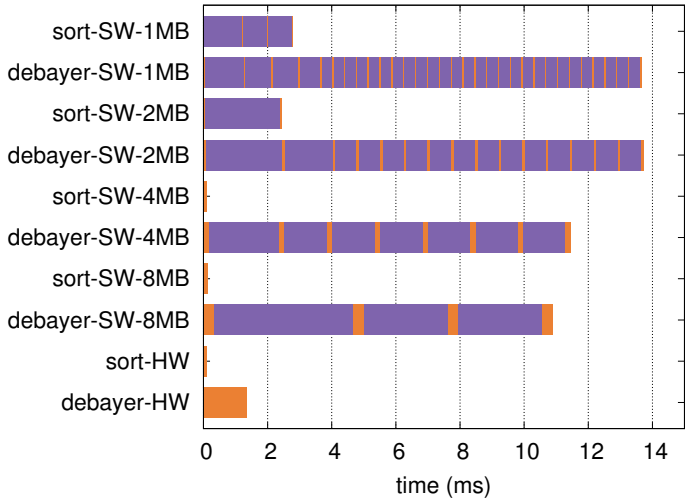


Figure 5.4: Software-managed DMA versus hardware-only DMA execution time breakdown. Orange segments correspond to accelerator DMA and computation, while purple segments represent software-handled data transfers.

SORT benefits from having a DMA buffer large enough for its memory footprint: the accelerator is able to complete the entire task without the intervention of the processor, thus obtaining a speedup of 21× over the test case with a 2MB DMA buffer. For DEBAYER, however, the software-based data management is always responsible for the largest part of the execution time, because its memory footprint never fits into the DMA buffer. Additionally, the execution of multiple accelerators creates contention on the processor, which must handle multiple concurrent transfers between each DMA buffer and the virtual memory of the corresponding application. This is shown by the purple bars that become longer when the two accelerators execute at the same time.

Following the intuition that avoiding the intervention of the processor core in DMA transfers (except from the initial setup) benefits the accelerated application, the experiment was executed again using a Linux patch known as *big-physical area*. When enabled, this patch forces the Linux operating system to reserve a region of contiguous memory configurable in size up to a few tens of megabytes. Fig. 5.5 shows the updated memory layout made possible by the patch: the entire application data set for both SORT and DEBAYER can be mapped to contiguous physical memory. Hence, the accelerator needs only the base address of the buffer to process all data, while the processor

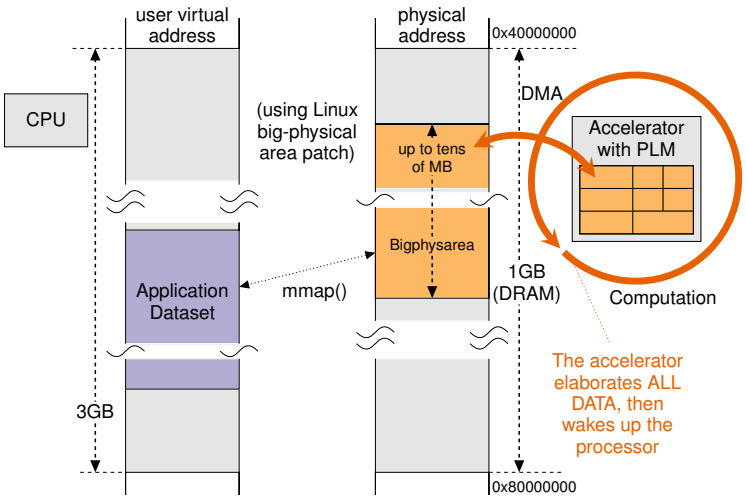


Figure 5.5: Hardware-only DMA using Linux *big-physical area* patch to reserve up to tens of MB of contiguous memory.

can remain idle or perform other tasks. The result is reported in the last two bars at the bottom of Fig. 5.4: the accelerator for DEBAYER achieves a speedup of 8× with respect to the scenario with an 8MB DMA buffer. This experiment proves the benefits of reducing the processor intervention when loosely-coupled accelerators move data with DMA transactions. The *big-physical area* patch, however, is only viable for applications with medium-sized memory footprints. As the number of accelerators and the size of data sets grow, it is necessary to adopt a more scalable and flexible approach.

5.2 DMA Platform Service for Accelerators

In the context of general purpose processors, cache hierarchy and virtual memory are typically used to give user applications the illusion of accessing the entire address space with low latency. As the number of accelerators integrated in SoCs keeps growing, designers need a similar efficient solution dedicated to special-purpose hardware components. This section describes a combination of hardware and software techniques that gives accelerators the illusion of accessing contiguous physical memory. Each accelerator can therefore issue memory references using an *accelerator-virtual address (AVA)*, equivalent to a simple offset with respect to its data structures, without requiring any information about the underlying system memory hierarchy. Combined with a lightweight dedicated

DMA controller, this makes all transactions occur across the entire data set without intervention of the processor, thus allowing the accelerators to preserve the speedup they were initially designed for.

Scatter-Gather DMA and accelerators. For off-chip peripherals and non-coherent devices, the standard Linux API provides routines to create a list of pages reserved for any virtual buffer. This list, called *scatterlist*, represents the *page table* (PT) for the buffer. This name refers to *scatter-gather DMA*, which is a common technique mostly applied to move data between main memory and the dedicated DRAM of on-board peripherals. To reduce the size of the PT, Linux tends to reserve blocks of contiguous pages whenever possible so that it is sufficient to store the base address and length of each block. A typical transaction to an external peripheral implies transferring all data stored in the area pointed by the PT. Hence, the scatter-gather DMA controller must simply walk the PT and gather data from all memory areas in order. Conversely, on-chip accelerators must deal with PLMs having limited size. Therefore, they have to issue several random accesses to memory, following a pattern that is highly-dependent on the implemented algorithm. Since the blocks may have different sizes, the access to a scattered memory buffer with a random offset requires the addition of every block length until the requested data is effectively reached. Moreover, long DMA transfers may easily span across multiple blocks, incurring further overhead to complete the transaction.

Alternatively, Linux can guarantee a set of equally-sized blocks, each consisting of one page (typically 4KB). However, considering a data set of 300MB, the number of required entries in the PT is 76,800, equivalent to 300KB on a 32-bit address space. A traditional TLB, holding only a few of these PT entries, would incur high miss rates. In fact, high-throughput accelerators do not typically reuse the same data multiple times and very little spatial locality can be exploited. The ESP platform service for DMA transfers solve this issue with a Linux module, named `contig_alloc`, implemented to enable optimized scatter-gather memory accesses for accelerators. A companion user-space library is also provided to replaces the standard `malloc` interface. Fig 5.6 shows the request data structure for `contig_alloc`, the `io_ctl` interface and the corresponding user-space function implemented as part of the ESP software library. A request to `contig_alloc` includes the size of the requested memory area (*size*), the desired size of each contiguous physical block

```
/* Data structure for contig_alloc */
struct contig_alloc_req {
 size_t size;                          /* aggregate size required              */
 size_t chunk_size;                    /* size of one chunk                    */
 struct contig_alloc_params params;    /* DRAM allocation policy               */
 unsigned int n_chunks;                /* number of contiguous chunks          */
 contig_khandle_t khandle;             /* handle for the device driver         */
 unsigned long __user *arr;            /* chunks physical addresses (PT)       */
 void __user **mm;                     /* user-space mapping of the chunks     */
};

/* Kernel module */
static long contig_alloc_ioctl(...) {
 /* ... check request, get lock  and allocate chunks ...*/
 if (!contig_alloc_ok(&req.params))
    return -EINVAL;
 mutex_lock(&contig_lock);
 switch (req->params.policy) { /* allocate memory blocks() */ }
 mutex_unlock(&contig_lock);
 /* bookkeeping and copy data structure to user space */
 list_add(&desc->file_node, &priv->desc_list);
/* return codes */
}

/* User hspace library */
int contig_alloc(contig_alloc_params params, size_t size, contig_handle_t *handle) {
 struct contig_alloc_req *req;
 /* prepare request, allocate arrays and system call */
 req->n_chunks = DIV_ROUND_UP(size, chunk_size); /* ... */
 ioctl(fd, CONTIG_IOC_ALLOC, req) /* ... */
 /* remap buffer to user space */
 for (i = 0; i < req->n; i++)
  req->mm[i] = mmap(NULL, chunk_size, flags,
          MAP_SHARED, fd, req->arr[i]); /* ... */
}
```

Figure 5.6: Snippet from `contig_alloc` Linux module for memory allocation with ESP accelerators.

(*block_size*) into which the memory region will be divided, and some allocation policy parameters. These parameters are intended for load balancing in case of multiple DRAM banks. By specifying only the parameter *size*, as typically done for `malloc`, the default values are used for the other parameters. The Linux module generates a DMA handle for the accelerator's driver, the resulting number of equally-sized blocks (also called *accelerator pages*), the corresponding PT, and the virtual-memory mapping for the user-space application. Fig. 5.7 shows the memory layout after calling `contig_alloc`. Note that only the calling process is allowed to access the allocated memory region and the user-level application can still operate transparently on the data in its virtual address space (purple-shaded area). However, differently from standard allocation mechanisms, the corresponding physical pages have larger size (orange-shaded regions in physical memory). For very large data-sets, a medium size for the accelerator pages (e.g. 1MB) cab be set, so that the resulting PT has a size on the order of a few KBs and can be thus stored contiguously in memory, as shown in Fig. 5.7.

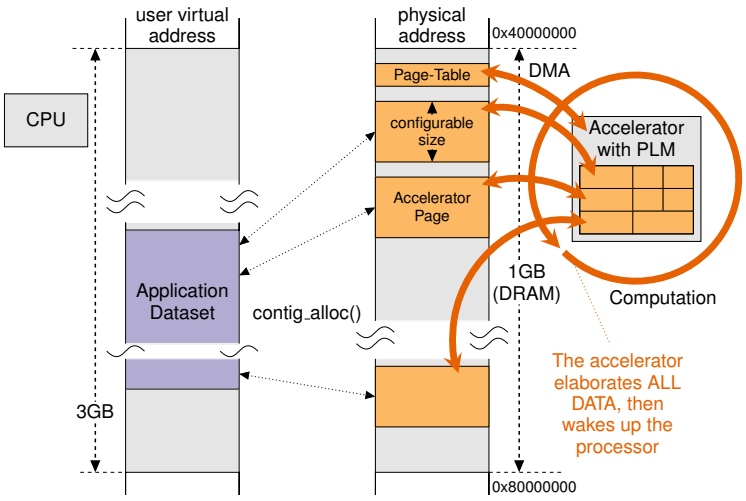


Figure 5.7: Memory layout after calling `contig_alloc` to enable low-overhead scatter-gather DMA for accelerators.

This approach enables a low-overhead version of scatter-gather DMA specialized for loosely-coupled accelerators. Moreover it maintains shared memory across processors and accelerators without requiring the PLMs be coherent with the processors' caches. Differently from Linux `huge_pages`, `contig_alloc` supports dynamic allocation of blocks which can have variable sizes, trading off PT size for memory fragmentation. Furthermore, `contig_alloc` can be used with any target architecture, because its implementation is not dependent on a specific processor core.

TLB and DMA controller for accelerators. Once the data are ready in memory, laid out as shown in Fig. 5.7, the application can run the accelerator by invoking the device driver through the traditional `io_ctl` system call. The driver takes the configuration parameters from the user application and passes them to the accelerator through memory-mapped registers. Such parameters include application-specific variables to be used directly by the accelerator kernel (e.g. the size of the image for the DEBAYER application), and the information for the DMA controller (e.g. the memory address where the PT is stored). Note that user application code is not expected to include complex routines to program the accelerators. On one hand, `contig_alloc` returns a buffer mapped in user-space, similarly to standard dynamic memory allocation. On the other hand, the ESP device driver handles the flushing of all necessary data from the cache and starts the accelerator. This technique avoids the

performance overhead of coherence [Cota *et al.*, 2015], without giving up shared memory, which eases the task of interacting with accelerators from software and avoids redundant copies of data across separate address spaces. Flushing the cache is required to guarantee consistency between processor's caches and accelerator's PLM. This operation, however, is completely transparent to the user-level applications and incurs negligible performance overhead.

After configuration, the DMA and computation are entirely managed by the accelerator. The accelerator requests are composed of a set of control signals to: distinguish memory-to-device from device-to-memory transfers, set an offset with respect to the data structure to process (corresponding to the AVA), and determine the transaction length. To serve such requests the ESP hardware sockets, introduced in Section 4.2, embed a DMA controller (DMAC) and a parametrized TLB for each accelerator. These components autonomously fetch the PT through a single memory-to-device transaction, and store it inside the TLB. Once the TLB is initialized, every accelerator request is translated in only four cycles. When operating on large data sets with very long DMA transfers, this address translation overhead is negligible. The TLB is configured to match the requirements of a given accelerator in terms of number of supported physical memory pages. In fact, thanks to `contig_alloc`, the number of pages is kept under control and set according to the size of the required memory area. Therefore, it is possible to have the number of PT entries match the TLB size. This not only simplifies the design, but it also minimizes the performance degradation due to scatter-gather DMA. Indeed, filling in the TLB can be done with one single transfer before activating the computational blocks. Results, reported in Section 5.3, confirm that preparing and using the TLB has a negligible impact on the overall execution time of the accelerators. Across the analyzed workloads, the accelerator page size is set to 1MB, which is a reasonable trade-off between the complexity of the memory allocation performed by the operating system and the PT size, resulting in few hundreds entries. Should an application require more entries, in order to handle even larger data sets, the TLB can be parametrized to hold more pointers in exchange for silicon area. Note that the relative performance overhead would not increase, because transactions and computation would also scale with the data set.

Fig. 5.8 shows the organization of the accelerator, the DMAC, the dedicated TLB, and the bank of configuration registers. Note that one of the registers stores the DMA handle generated by `contig_alloc`. This corresponds to the PT base address and is used to initialize the TLB. The

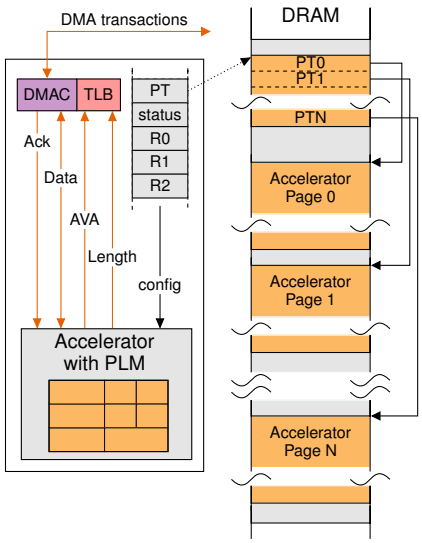


Figure 5.8: DMA interface and accelerator's page table

DMAC and TLB behaviors are described by the finite state machine in Fig. 5.9(a) and Fig. 5.9(b), respectively. As soon as the PT register is written by the device driver, the DMAC engine initiates an autonomous transaction to retrieve the PT, as shown by the transition from *idle* to *send_address* in Fig. 5.9(a). The request includes the PT address and the number of entries to fetch. Then, following the control flow of read requests (i.e. *MEM_TO_DEV* path), the DMA waits for the response of the memory controller before transferring the received pointers to the physical blocks into the TLB. The operation terminates when all entries are received: this corresponds to the transition from *rcv_data* to *idle* in Fig. 5.9(a), where the signal *tlb_empty* is de-asserted. This also corresponds to the transition from *tlb_init* to *idle* in Fig. 5.9(b). After this TLB initialization, the DMAC steps through the states *config* and *running*, and starts its execution. Whenever the accelerator needs to perform a read or write request to DRAM, it sends a request to the DMAC through its DMA interface, as shown in Fig. 5.8. Specifically, the AVA and the length of the data transfer are sent to the TLB, which initiates the address translation, while the DMAC starts a handshake protocol with the DMA interface of the accelerator. The TLB determines whether the transaction needs to access one or multiple pages in memory, and computes the length of the transfer for the first accelerator page. In four cycles the TLB is ready to provide the physical address and the DMAC initiates a transaction over the interconnection system, following either the *MEM_TO_DEV* or the *DEV_TO_MEM* paths, for read or write operations, respectively. In the case of read requests, the acknowledge signal (*Ack*) shown

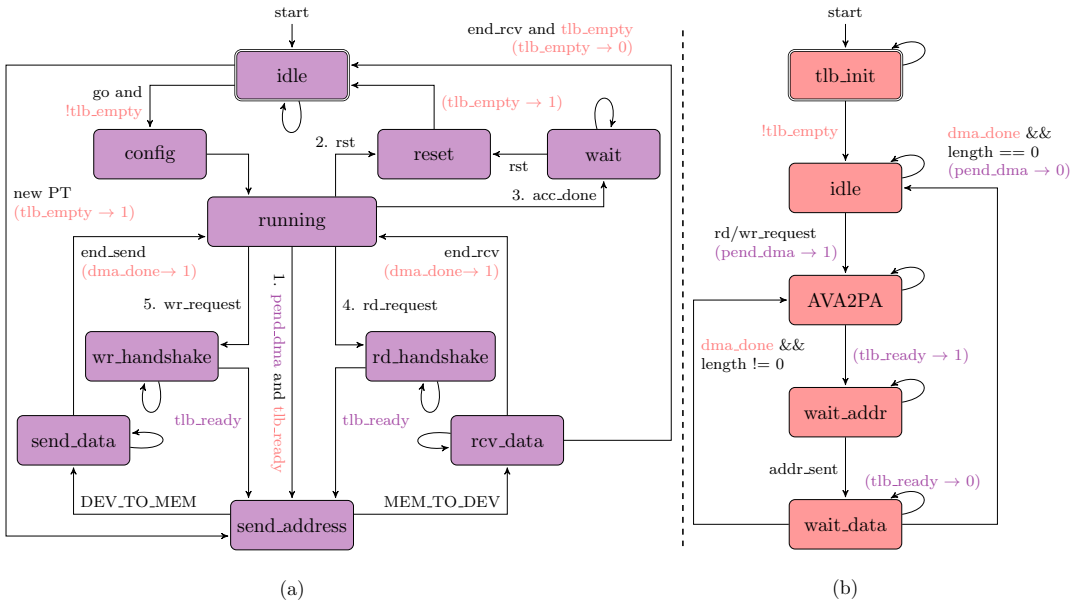


Figure 5.9: DMA controller (a) and TLB (b) finite state machines.

in Fig. 5.8 is set when valid data are available. Conversely, in the case of write requests, the signal *Ack* is set when an output value (*Data*) has been sent to the DMAC. This simple latency-insensitive protocol [Carloni, 2015] ensures functional correctness, while coping with congestion and DRAM latency. After the request has been sent to the interconnect, the TLB controller steps to a second waiting state (i.e. *wait_data* in Fig. 5.9(b)). In this state, if the current transfer length does not match the actual length requested by the accelerator, the controller reads the physical address of the next page in the TLB and initiates another transaction skipping the handshake with the accelerator. When the DMAC returns to the state *running*, it checks first for pending transactions, then it reads the command register to check for a reset from software, and finally waits for the accelerator to raise another request or for completion (i.e. signal *acc_done*). Note that, even considering the DMAC initialization, the delay introduced by each accelerator request is negligible when compared to the lengths of typical burst data transfers, which is of the order of thousands of words.

5.2.1 Main Memory Load Balancing

As the number of accelerators grows, the system interconnect and the I/O channels to the external memory are responsible for sustaining the increasing traffic generated by many long DMA transac-

tions. The ESP tiled architecture pairs naturally with a network-based interconnect and a network-on-chip (NoC) offers larger throughput and has better scalability than traditional bus-based interconnects [Dally and Towles, 2001]. Nevertheless, given that all accelerators need access to external memory, it is necessary to optimize the traffic on the NoC to minimize congestion. The availability of multiple memory channels and DDR controllers on modern systems improves the NoC traffic by balancing the data allocation among such controllers. The optional parameters of `contig_alloc` enable the user to distribute traffic through different paths to the DRAM banks. In addition to such parameters, when loading the kernel module, it is possible to specify the region of the physical address space where `contig_alloc` is allowed to request accelerator pages. The presence of multiple channels to the external memory allows additional control of the interconnect traffic. A sensitivity analysis of the proposed design with respect to load balancing is performed by implementing three allocation policies:

1. `POLICY PREFERRED` returns the first available accelerator pages from the most affine DDR node (i.e. the closest one in the SoC layout to the accelerator owning this set of pages). Some pages may be picked from other DDR nodes if the preferred one has not enough free memory.

2. `POLICY LEAST-LOADED` returns all pages from the least loaded DDR. This policy can be tuned with a user-defined threshold parameter that biases the priority among DDR nodes. For example, if the kernel module is allowed to allocate pages on half of the address space corresponding to the first DDR node (namely `DDR0`), it is convenient to set a threshold for this policy. For instance, by setting the threshold to 16, the policy will allocate the requested pages to `DDR0` only if all other DDR nodes have at least 16 more allocated pages than `DDR0`. Note, in fact, that the region of memory exclusively managed by the operating system is accessed more frequently by the processors. Hence, the system incurs higher contention between accelerators and processors when many accelerator pages are located on `DDR0`.

3. `POLICY BALANCED` returns sets of pages with specified cardinality. Each set of pages is alternatively allocated on the available DDR controllers. This policy accepts the same threshold as `POLICY LEAST-LOADED` to select the first DDR node. Additionally, the number of pages per set is also specified by the user and determines the granularity for balancing the allocation.

5.3 Multi-Accelerator Test Scenarios

The proposed solution to the LDS problem is evaluated with full-system-test cases that integrate different mixes of eight loosely-coupled accelerators. These are implemented to accelerate a set of computing kernels from the PERFECT Benchmark Suite [Barker *et al.*, 2013]. The selected kernels are are very heterogeneous as they process a variety of input/output data sets, with different memory-access patterns and communication vs. computation ratios. Consequently, the corresponding accelerators share the general structure shown in Fig. 5.2 but have major differences in terms of the micro-architecture of the computational blocks and the PLM structure.

Accelerated kernels. The eight selected computational kernels operate on input data sets that are provided as part of the PERFECT suite in three different sizes: SMALL, MEDIUM and LARGE. The actual sizes vary across kernels and range from 1MB to 300MB. Except for SORT, which is executed in-place, the applications must allocate additional data structures to store output and temporary data. Thus, their memory footprint grows up to 345MB, as shown in Table 5.1.

All the selected kernels execute heavy computation tasks, but they are very heterogeneous in terms of data access patterns. SORT, for instance, reorders iteratively and in-place N arrays of 1024 floating-point elements, where N can be 256, 512 or 1024, depending on the data set. FFT2D performs the Fast Fourier Transform (FFT) on each of the input rows of length 2^N, then it transposes the resulting matrix and finally it performs FFT on the transposed-matrix rows. Thus, it requires an additional workspace of the same size of the input matrix, i.e. $2^N \times 2^N$, where N is at most 12. The DEBAYER kernel takes as an input an $N \times N$-pixel Bayer-array image (with N ranging from 512 to 2048 pixels) with one color sample per pixel and returns an image with three-color samples per pixel. Thus, the resulting output is three times bigger than the input. Moreover, the algorithm interpolates pixels row-by-row and uses 5×5-interpolation masks centered on the pixel of interest. LUCAS-KANADE performs image alignment. The algorithm has a multiply-accumulate nature that stores the results in the Hessian output matrix. This has a fixed size (6×6) independently from the size of the input images. Differently from what presented in Section 4.1, for these experiments the entire task of LUCAS-KANADE is implemented as a single accelerator. Its memory footprint grows significantly with the larger data sets due to the amount of intermediate results that the algorithm allocates on the memory stack. Indeed, it has the highest growth rate among all kernels. Further-

KERNEL	SW APP. FOOTPRINT MB	PLM KB	FPGA RESOURCES LUT	FF	BRAM	CMOS AREA μm^2
Sort			36,868	31,300		281,045
−Mem.	18.2	24.00			6	74.95%
FFT2D			3,965	2,190		834,147
−Mem.	292.3	128.00			48	94.13%
Debayer			4,446	1,968		796,920
−Mem.	42.3	95.86			32	98.53%
Lucas-Kan			5,329	3,210		319,109
−Mem.	173.4	20.28			8	84.42%
Change-Det.			16,274	6,378		596,029
−Mem.	345.4	63.00			18	90.57%
Interp.1			20,836	9,119		492,647
−Mem.	109.4	48.05			12	69.65%
Interp.2			20,908	8,623		575,561
−Mem.	137.2	64.05			16	76.67%
Backproj.			14,040	5,588		782,263
−Mem.	329.3	99.00			81	91.61%

Table 5.1: Characterization of the implemented accelerators for the set of experiments dealing with large data sets.

more, each iteration of its computation phase requires two independent memory-read operations: the access pattern of the first one is data dependent, while the second transaction has a behavior known ahead of computation. Since the accelerator can be implemented without considering the SoC memory subsystem, these irregular memory accesses do not exacerbate the complexity of the accelerator. CHANGE-DETECTION takes as input a sequence of frames and an initial training set, and it returns a new training set and a "ground-truth mask": certain portions of each frame are labeled as background. Both frames and training set are represented as a set of N×N-pixel matrices where N is at most 2048. This results in the biggest data set among the kernels (300MB). On the target platform, this application has a memory footprint of 345.4MB. The other three kernels are part of a radar-based imaging application that produces high-resolution imagery by composing data from relatively small images. Two alternative methods of image formation exist: polar format algorithm

(PFA) and backprojection algorithm. The INTERPOLATION-1 and INTERPOLATION-2 kernels are the most computational intensive portions of PFA. All three operate on large matrices, but INTERPOLATION-1 reads them row-by-row, INTERPOLATION-2 accesses them column-by-column, and BACKPROJECTION has a data-dependent access pattern.

This variety of accelerators is integrated in multiple ESP instances, created through the methodology described in Chapter 4. Each instance has four types of tiles. A *CPU tile* integrates a LEON3 embedded processor [Gaisler, 2004] that runs the Linux operating system and the embedded software stack, including the `contig_alloc` module, the ESP device drivers, and the user applications. Each *DDRx tile* has a memory controller offering one independent channel to the external memory. A *MISC* tile implements all other I/O channels and peripherals that are responsible for booting the system and supporting a debug interface. Lastly, each *accelerator tile* encapsulates a given accelerator together with an instance of the components of Fig. 5.8, which provide a simple *network interface* between the guest accelerator and the system interconnect. The flexibility of this interface allows designers to easily swap or replace tiles to create different memory mappings and test scenarios, for which the corresponding routing tables are automatically generated.

The tiles are interconnected through a packet-switched multi-plane NoC. Accelerators rely on two NoC planes that are dedicated to DMA transactions (one for memory-read and one for memory-write transfers) and guarantee deadlock avoidance. The accelerator DMA does not interfere with the NoC planes dedicated to the processor cache request-and-response transfers until the packets reach the memory. Non-cacheable register operations, control messages, and interrupts are delivered through a fifth plane, which is accessed by all tiles. While the size of the SoC instances are ultimately limited by the available resources on the target FPGA, the ESP infrastructure is inherently modular and scalable: it allows for more tiles and NoC planes as the number of integrated accelerators and memory controllers increases.

Probes and performance counters. A set of accurate performance counters are placed at the interface of each DMAC and NoC router. They serve as probes to gather statistics during system execution. In particular, the probes placed between each accelerator and its DMAC measure the total number of cycles in which the accelerator is active, along with the cycles spent in communication (i.e. when DMA transfers are occurring) and in TLB access. The probes placed at each router port

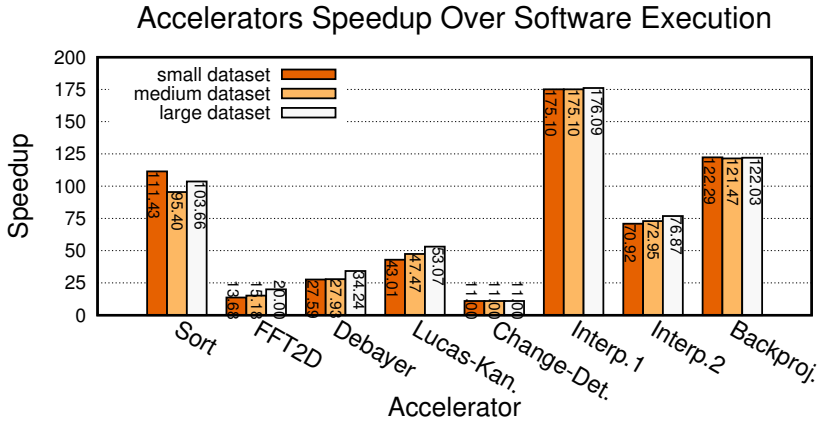


Figure 5.10: Speed-up of each accelerator with respect the corresponding software executions for three data-set sizes.

measure the number of cycles when a flit traverses each link. This information are useful to quantify the contention for shared resources across the different scenarios.

Implementation details. For all SoC instances the target clock frequency is 80MHz. The implemented designs are mapped on a proFPGA Prototyping System [ProDesign, 2014], equipped with a Xilinx Virtex-7 XC7V2000T FPGA and two DDR-3 extension boards. This provides the system with dual-channel access to memory (namely `DDR0` and `DDR1`). The total addressable off-chip memory is limited to only 1GB by the LEON3 default mapping, which is however sufficient to execute all selected workloads. This address space is split into two partitions, each of size 512MB. The lower portion is mapped to `DDR0` and includes 128MB of memory exclusively reserved for the operating system that cannot be used by `contig_alloc`. The rest of the address space, instead, is dynamically shared between the processor and the accelerators.

Hardware solution overhead. The components for translating the requests from each accelerator to the corresponding NoC interface require about 600 look-up-tables (LUT) and 600 flip-flops (FF). Without the logic to support `contig_alloc`, the same DMA engine would require 350 LUTs and 400 FFs. One additional block RAM (BRAM) is needed to store the TLB for each accelerator. This little overhead in terms of resources, however, is negligible when compared to accelerators (see Table 5.1). The aggregated performance overhead to access the TLB is just a few hundred cycles,

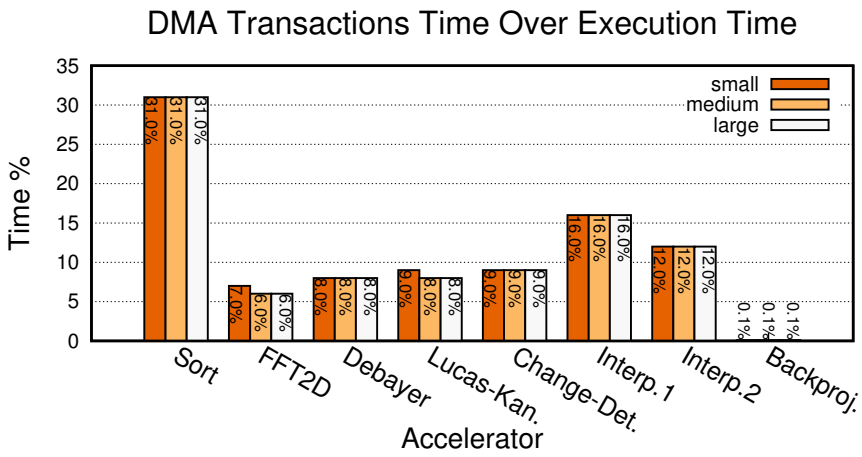


Figure 5.11: Time spent in data transfers expressed as a fraction of the total execution time of accelerators.

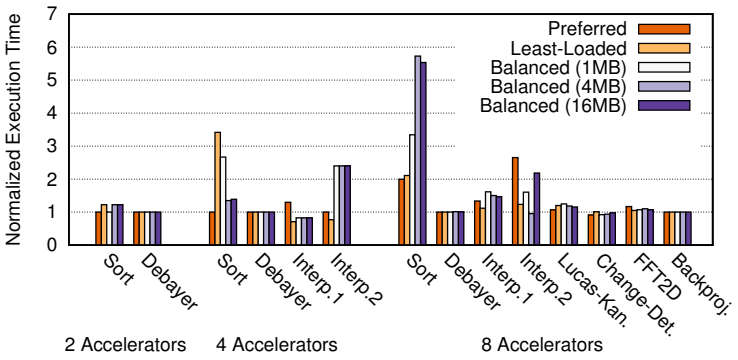
which are negligible across all workload scenarios if compared to a total execution time that ranges in the hundreds of millions cycles. Address translation and TLB initialization time is indeed eight orders of magnitude smaller than the total accelerator execution time.

Evaluation of single accelerators. As a first step, each accelerator presented above is tested standalone in order to assess the speedup over the corresponding software implementation. For this set of experiments, data are allocated on `DDR1`, while `DDR0` is reserved for the processor. This allocation minimizes the contention between the processor and the accelerator. The results are reported in Fig. 5.10. The speedups range from 11× (for CHANGE DETECTION) to 175× (for INTERPOLATION-1). These tests were run on all data sets and show the scalability of the proposed solution for up to 300MB of input data. The speedup is almost constant for five accelerators out of eight, while it shows a slight increase on larger data sets for the others. Average speedup across all input data sizes is about 70× and grows to almost 75×, when excluding smaller test sizes.

Fig. 5.11 reports the percentage of execution time during which each accelerator is involved in a data transfer. This percentage includes both the time where useful data reach or leave the accelerator tile and the waiting time caused by DDR latency. Such metric is a key characteristic of the accelerator, which depends primarily on the ratio between computation time and communication time and on how much these two phases are allowed to overlap. Higher percentages of communication time

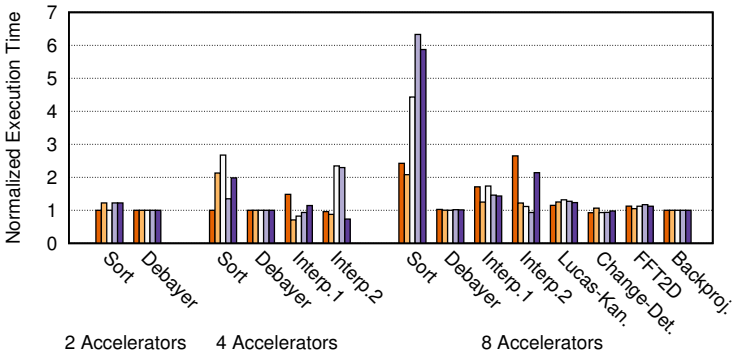
correspond to a larger sensitivity to system congestion and memory bandwidth, because the accelerator tends to perform less operations on each byte of data brought to the PLM. For this reason, it can be easily stalled when varying the latency of memory transfers. As an example, the DMA controller for SORT is active for more than 30% of the execution time, indicating a high sensitivity to variations of the available bandwidth. Conversely, the BACKPROJECTION accelerator performs an extremely long computation after bringing the data in the PLM. Therefore its communication time is less than 1% of the total execution time and traffic over the interconnect has a smaller impact on the achievable speedup. Note that applications performing very little computation on each data token are not suitable for loosely-coupled accelerators[Cota *et al.*, 2015].

Multi-accelerator workloads. The interaction of multiple accelerators is analyzed by sweeping the number of concurrent accelerators and changing the memory allocation policy, chosen among those described in Section 5.2.1. The LEON3 processor limits the amount of addressable DDR to 1 GB, therefore not all accelerators can execute *concurrently* with large data sets at the same time. However, throughout the experiments the total amount of allocated pages reaches a maximum of 768 MB, corresponding to as many accelerator pages. The first SoC instance integrates one copy of each accelerator implemented, and has two memory channels located at the corners of the NoC, as shown in Fig. 5.12. The second test case, reported in Fig. 5.13, is similar to the previous one, except for the location of the memory controllers. These are now positioned in the central tiles to investigate the sensitivity of the design to the placement of the most contended shared resources. The third SoC, shown in Fig. 5.14, integrates two copies of five different accelerators, for a total of ten accelerator tiles. Using two copies of each accelerator reduces the degree of heterogeneity and affects the traffic over the interconnect because there are more components with the same access patterns to memory. The last test case integrates twelve accelerators for the FFT2D kernel (Fig. 5.15) and stresses the system with homogeneous traffic patterns generated from all accelerator tiles. The bar charts next to each SoC layout report the execution time for every accelerator, across several experiments. Each bar is normalized against the corresponding single-accelerator execution time. Each group of clustered bars corresponds to a workload scenario with multiple accelerators running at the same time. For instance, the chart in Fig. 5.12 reports three workloads, running two, four and eight accelerators, respectively. For every workload, the experiment were repeated for five



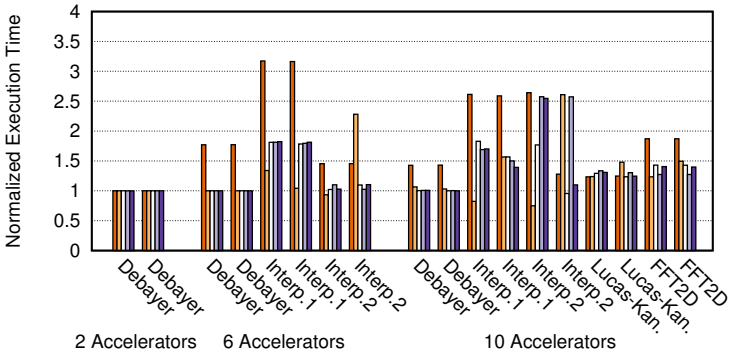
DDR0	INT1	INT2	LK
DB	CPU	MISC	BP
FFT2D	CD	SORT	DDR1

Figure 5.12: Scenario (a): heterogeneous accelerators with memory controllers at opposite corners.



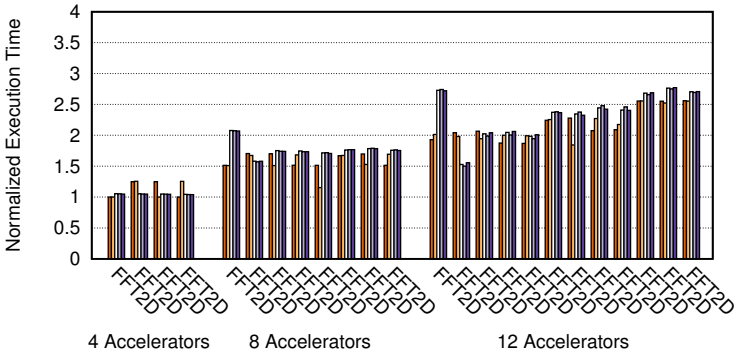
LK	INT1	INT2	CPU
DB	DDR0	DDR1	BP
FFT2D	CD	SORT	MISC

Figure 5.13: Scenario (b): heterogeneous accelerators with memory controllers at central tiles.



DDR0	MISC	CPU	
FFT2D	FFT2D	DB	DB
LK	LK	INT1	INT1
	INT2	INT2	DDR1

Figure 5.14: Scenario (c): heterogeneous pairs of accelerators with two empty tiles.



DDR0	MISC	CPU	FFT2D
FFT2D	FFT2D	FFT2D	FFT2D
FFT2D	FFT2D	FFT2D	FFT2D
FFT2D	FFT2D	FFT2D	DDR1

Figure 5.15: Scenario (d): homogeneous accelerators.

different allocation policies. The leftmost bar in each cluster corresponds to `POLICY PREFERRED`, which has no configuration parameters. The second bar (in yellow) shows the results for `POLICY LEAST-LOADED` configured with a penalty of 32MB for `DDR0`, so that `DDR1` is preferred when both banks are similarly loaded. Finally, the three bars in different shades of purple correspond to `POLICY BALANCED` with sets of 1, 4 and 16 pages, sized 1MB each.

The first conclusion that can be drawn is that for small sets of accelerators the execution time is mostly unaffected by concurrency. This is shown for heterogeneous workloads with two accelerators in Fig. 5.12, 5.13 and 5.14, and for the case of four FFT2D in Fig. 5.15. The little fluctuations reported are due to unpredictable behavior of the system, where the operating system is constantly running and generating "noise" in terms of memory utilization. By increasing the number of accelerators running concurrently, the effects of contention for the shared resources starts affecting the execution time. For instance, SORT is heavily penalized by the higher ratio between communication and total execution time, as already noted for Fig 5.11. Nevertheless, the aggregate performance of multiple concurrent accelerators keeps improving, even if with diminishing returns. For example, by considering the second group of bars in Fig. 5.15, it is possible to distinguish eight clusters for as many instances of parallel FFT2D accelerators. The chart shows that they not only perform better than a single accelerator running in series, but they also exceed the performance of the scenario with four FFT2D. The average execution time, across all policies, for eight FFT2D, in fact, is below the break-even point of $2\times$. From this viewpoint, even better results are shown for the heterogeneous SoCs. Having different accelerators, in fact, leads to the interaction of heterogeneous data access patterns, which tend to reduce contention for shared resources. These results show that, as long as the interconnect can sustain the bandwidth requirements of the accelerators, the design scales well with limited impact on performance. Furthermore, the comparison between Fig.5.12 and 5.13 leads to the conclusion that the proposed solution to the LDS problem is robust to variations of the SoC layout (i.e. position of tiles). In particular, moving the memory controllers from the corners to the central tiles has no impact on the system behavior, even though the traffic distribution on the NoC changes significantly.

Finally, by looking at the results for all workloads, notice that the allocation policy has little to no impact on the performance in most cases and for most accelerators. Such behavior is highly desirable, because it does not constrain the operating system to use one specific load balancing tech-

nique for memory. Note that there are few exceptions to this observation. For instance, results for SORT in Fig. 5.12 and 5.13 show significant variations in the execution time based on the allocation policy. Indeed, on one hand, accelerators like SORT that have a higher ratio between communication and total execution time (see Fig. 5.11) require higher bandwidth with the memory. On the other hand, when multiple accelerators' buffers are scattered across the two DDR nodes, there are on average more packets colliding on the NoC and this can affect the performance. In fact, the probes located inside the NoC routers measured on average 3× more packets traversing the links around the tile for SORT when changing allocation policy from `PREFERRED` to `1MB-BALANCED`. Hence, the reported performance loss is not directly correlated with the DMA and address translation logic. Instead, it is a natural consequence of a higher NoC traffic.

Chapter 6

Fine-Grain Power Management

Emerging technologies allow for a larger number of voltage/frequency (VF) domains on modern SoCs. In addition, on-package and fully integrated voltage regulators (IVRs) allow for much faster transient times in comparison to discrete regulators. How should the hardware and the operating system of SoCs adjust to exploit new extremely fine-grained DVFS capabilities?

This chapter addresses this question by extending the ESP architecture and its FPGA-based emulation infrastructure to analyze the impact of fine-grained DVFS and explore the design space of power-control management. The infrastructure combines actual frequency scaling emulation of multiple independent domains with RTL power-estimation flow to explore the impact of several DVFS policies while varying workload, number of VF domains and temporal granularity. When many high-throughput accelerators execute simultaneously, they compete to gain access to the DRAM memory through the on-chip interconnect. Hence, congestion may naturally arise and the accelerators could make little progress and waste the two most important shared resources: power supply and communication bandwidth.

ESP flexibility allows designers to quickly build different scenarios by mixing heterogeneous accelerators, each coupled with an instance of a *dedicated hardware controller* to enforce a local DVFS policy. Configuration parameters can be set via software, with a *daemon process* running in the Linux kernel that has a global view of the system. In summary, this infrastructure enables pre-silicon tuning and the design exploration of DVFS policies.

6.1 Background on Voltage Regulators

Typically, power management is part of the operating system services: the slow response time of on-board regulators allows the software implementation of sophisticated DVFS policies. In the future, more complex chips with larger numbers of heterogeneous cores and VF domains will require even more sophisticated policies involving the continuous execution of a sequence of tasks such as: scanning many status registers across the cores, monitoring the current workloads, checking the overall power envelope, and picking an optimal operating point for every domain. Meanwhile, the cost of building IVRs, which is mainly area penalty when fabricated on chip and I/O limitations when stacked with 3D integration, must be capitalized by leveraging their faster transient-response time and finer spatial granularity. For instance, a dedicated IVR could be used to promptly reduce the VF operating point of an accelerator whose performance is temporarily throttled by the delayed arrival of data due to a congestion of the on-chip interconnect. Software-only-based solutions seem inappropriate to handle this kind of scenario and call for new power management policies that involve dedicated hardware controllers. Continuous progress in the technology and design of voltage regulators [Wang *et al.*, 2014b] holds the promise of enabling this fine-grained power management both in space (with multiple distinct voltage domains) and in time (with faster transient response).

On-chip voltage regulators can be divided roughly into two categories: linear regulators and switching regulators. *Linear regulators* implement a voltage divider-type structure, using feedback control to control the resistive division. This mechanism restricts the efficiency of such schemes to at most the ratio of the output voltage to the input voltage, e.g. a 2:1 conversion has an ideal efficiency of 50%. This precludes naive deployment of linear regulators in DVFS schemes. *Switching regulators* store energy in capacitors or inductors and deliver that energy at a potential controlled by a switching signal. Voltage conversion through this (ideally) lossless energy transferral can thus take place with markedly higher efficiency, limited only by parasitic resistances and dynamic switching losses. Linear regulators have the benefit of being significantly easier to deploy on chip, as they generally require only resistors in addition to the active circuitry. Switching regulators, instead, require capacitance or inductance values that need significant area and complex fabrication steps to be deployed. This has led to schemes involving many small linear voltage regulators in a grid-like fashion for applications targeting fast transient response [Toprak-Deniz *et al.*, 2014].

Die-integrated switched-capacitor regulators boast similarly fast transient response, high peak

efficiency (>90%) and minimal technology requirements, but suffer from relatively low power densities (<1 W/mm^2), precluding domain-number scaling for high-performance processors [Sanders *et al.*, 2013]. Efforts have been made to address this by using high-density deep trench and/or ferroelectric capacitor technologies, with several demonstrations of >85% conversion efficiency with densities greater than 2 W/mm^2 [Andersen *et al.*, 2014; Andersen *et al.*, 2015; Andersen *et al.*, 2013; Chang *et al.*, 2010]. Such systems are very promising for enabling DVFS due to the relative maturity of the requisite technologies. With the progress in controller design, modern switched-capacitor regulators can offer output voltage ranges with reasonable (5% to 20%) worst-case efficiency degradation over output voltages of interest.

Switched-inductor regulators boast higher power densities. Indeed, switched-inductor regulators are used in most discrete, board-level power management infrastructures and can be designed to have >90% efficiency. However, they remain difficult to build into die-integrated systems due to the low quality factor and correspondingly low efficiency available in die-integrable inductor technologies [Wang *et al.*, 2014a]. Efficiency in the 70%-85% range has been reached with state-of-the-art inductor technology [DiBene *et al.*, 2010; Sturcken *et al.*, 2013; Tien *et al.*, 2015]. The integration of inductors at the package level is less aggressive and allows for use of high quality factor air-core inductors, with corresponding increases in efficiency to the 85-90% range [Burton *et al.*, 2014; Sturcken *et al.*, 2012]. This approach has also succeeded in pushing scaling boundaries for DVFS applications up to thirty domains [Burton *et al.*, 2014]. Enabling efficient die-level integrated regulators in the 10 W/mm^2 power-density range promises further increase of the VF domains.

6.2 Fine-Grain Dynamic-Voltage-Frequency Scaling

Assuming that IVRs will continue to improve in the near future, it is necessary to analyze the impact of fine-grained DVFS on SoC architectures with loosely-coupled accelerators, such as any instance of ESP. Accelerators implement computational-intensive tasks with significant benefits in terms of performance (two-to-three orders of magnitude [Hameed *et al.*, 2010]); also, they can be completely turned off when inactive, reducing the number of transistors switching in the SoC (i.e. they become *dark silicon* [Esmaeilzadeh *et al.*, 2011; Taylor, 2012]).

These SoC architectures present a high degree of heterogeneity since the accelerators have dif-

ferent characteristics in terms of area, performance, and power consumption. However, as the number of their components increases, it is critical to introduce some sort of regularity to keep the SoC design and testing process manageable. Once again, ESP's modular and tile-based architecture, with a NoC as interconnect fabric, offers a desirable balance between flexibility and regularity[1]. A modular architecture can also reduce significantly the complexity of physical design and route, clock distribution, and power grid layout [Angiolini *et al.*, 2006; Dally and Towles, 2001]. Moreover, the packet-switched NoC naturally decouples the run-time operations of the various tiles because flow-control protocols regulate independently the access to the network for each tile through *back-pressure* mechanisms: a stall signal on the local port of the router indicates to its corresponding tile that it has to wait for some clock cycles before the network can accept more packets. Finally, an NoC offers a natural synchronization barrier among clock domains. The local ports of the NoC routers, in fact, may feature dual-clock FIFO buffers which preserve the throughput of transfer bursts and prevent data loss across clock domains [Strano *et al.*, 2010]. Indeed, NoC-based architectures have already been implemented in prototyping chips together with voltage regulators [Salihundam *et al.*, 2011].

The top of Fig. 6.1 shows an instance of ESP that is based on a 4×4 2D-Mesh NoC and features four types of tiles: one *CPU tile* with a general purpose processor running the Linux operating system, twelve *ACC tiles*, each hosting a distinct accelerator, one *I/O tile* with communication peripherals (e.g. Ethernet, UART, JTAG...), and two *MEM tiles*, each hosting a memory controller to access a distinct DRAM storing one half of the overall shared main memory (see Chapter 5). In this example, the twelve accelerators are distributed across four VF domains, labelled D0, D1, D2 and D3. Each domain is enclosed by the dotted lines. A voltage regulator is associated with each VF domain to control the supply voltage of its components. When all the components of a domain are inactive, they can be turned off. Otherwise, they still dissipate static power. Only one tile per domain (colored in light blue) contains the DVFS controller with a PLL, while the other accelerator tiles in the same domain region share the same controller. The drawing at the bottom of Fig. 6.1 shows in more detail an accelerator tile that contains also the DVFS controller (DVFS CTRL) for its VF domain. The accelerator is composed of multiple hardware blocks that interact

[1]While all tiles are not necessarily required to have exactly the same size, this helps the physical design process and can be achieved by combining multiple smaller accelerators within a tile.

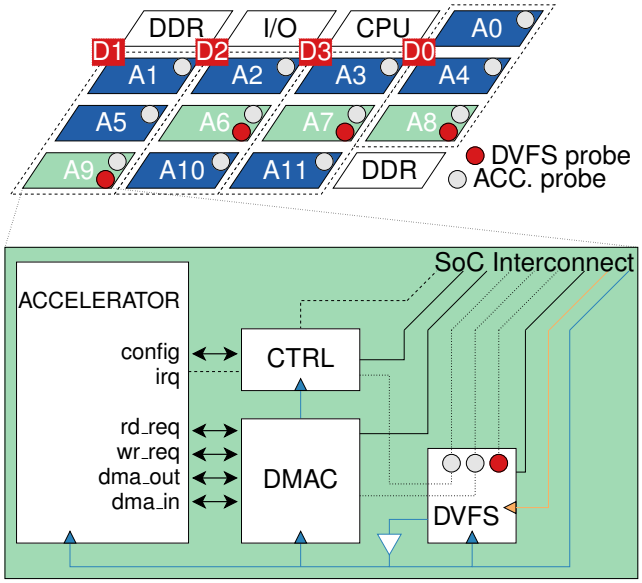


Figure 6.1: Tile-based SoC high-level view (top) and block diagram of an accelerator tile with DVFS controller (bottom)

through the PLM (see Section 4.1). The control interface (CONFIG) exposes the accelerator's configuration parameters to the Linux operating system as memory-mapped registers to be set by the device driver. The accelerator uses direct memory access (DMA) to exchange data between its private memory and one of the two DRAM banks. The DMA controller (DMAC) translates the accelerator's read/write requests into transactions over the NoC. These data transfers are initiated directly by the accelerators during input and output phases. Synchronization signals coordinate the activities of these blocks with those of on or multiple computation blocks that can start the execution only when the data are available in the PLM. Typically, in an efficient accelerator design, computation and communication are well balanced and all blocks work in parallel (see Chapter 5): as a result, when active an accelerator can inject/eject data to/from the NoC at the rate of one packet per cycle per direction.

If many accelerators are simultaneously active in the SoC and work at this rate, it is not uncommon that the NoC becomes congested, with high contention to accessing the DRAM controllers. Things may be further complicated in the case of data dependencies among accelerators, e.g. before starting its execution an accelerator may require that another accelerator terminates its execution and stores its results in DRAM. NoC congestion naturally leads to the occurrence of back-pressure

for some accelerators that end up having to stall their execution waiting for the arrival of new data. These situations offer opportunities to exploit fine-grained DVFS, which can reduce the wasting of supply power while contributing also to alleviate NoC congestion. To dynamically detect these situation, a tile is equipped with probes for the accelerator activity (shaded gray circles in Fig. 6.1). These probes detect: (*i*) whether the accelerator is enabled; (*ii*) if it is computing, transferring data, or both; and (*iii*) if the tile is receiving back-pressure from the NoC, either due to congestion or temporarily unavailable access to main memory. This information is also provided to performance counters and exposed to an external Ethernet interface for application profiling. Finally, each tile hosting a DVFS controller includes also a probe to monitor its behavior (red circle in Fig. 6.1). The DVFS controller receives the information from the probes of all accelerators contained in its VF domain and combines them in order to apply the desired power management policy, as explained in the next section.

6.2.1 Local Power Control Platform Service

This section provides structural and behavioral details of the implemented DVFS controller.

Configurable DVFS Controller. Fig. 6.2 shows the block diagram of the DVFS controller. Specifically, the component DVFS CTRL is a finite state machine (FSM) responsible for regulating voltage and frequency for its local domain by adjusting the VR's reference voltage and dynamically reconfiguring the PLL. The signal `vctrl` translates into a voltage reference adjusting the regulator feedback path to obtain the desired output. The PLL control logic may vary depending on the specific implementation. The simplified scheme of Fig. 6.2 refers to a run-time reconfigurable PLL provided as an IP block together with the 32 nm CMOS commercial standard cell library used for the energy estimation flow in Section 6.2.3.

The FSM logic must be simple enough to guarantee that the power management is performed in a timely manner with respect to the transient time of the VRs. A set of registers, mapped to system memory, allows reconfiguration from software, which can override the local decisions in favor of a new system-level policy. In Fig. 6.2, note the need for synchronization flip flops on the paths between the DVFS controller and the PLL state machine. The latter, in fact, must be clocked by the external reference clock (`refclk`), to make sure all PLL configuration pins are

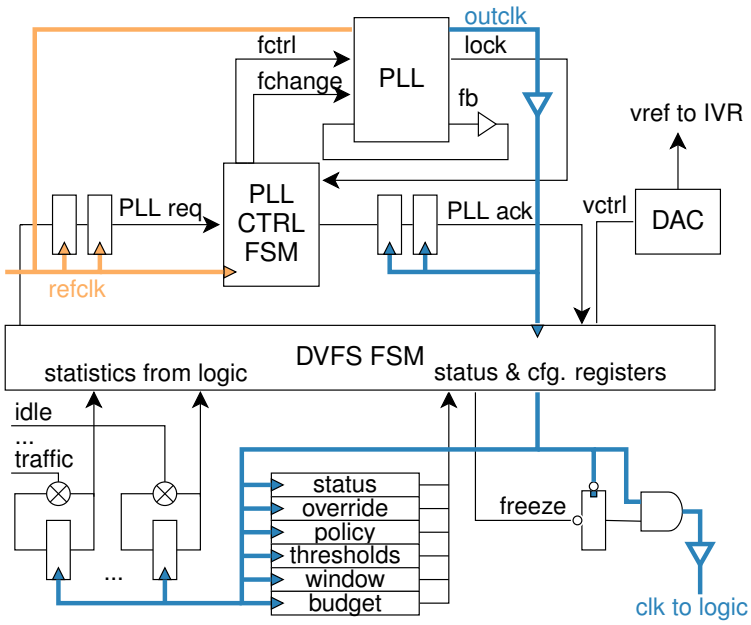


Figure 6.2: DVFS controller block diagram.

driven correctly while its output clock is transitioning from one operation point to the other. The main DVFS controller, instead, shares the same clock frequency with the rest of the logic in the VF domain. This solution reduces the number of synchronization points, as the interface towards the rest of the system is more complex and has a higher bit count. It is also worth noting the feedback compensation clock `fb`, required to obtain a correct phase locking, together with the clock buffers, which are represented as triangles along clocks' paths. Such buffers are the entry point to the clock distribution network and deserve special attention when trying to perform frequency scaling on an FPGA, as discussed in Section 6.2.3.

DVFS state transitions. The lower-left portion of Fig. 6.2 shows two counters, which are incremented every time a specific condition holds within the context of the local VF domain. These represent some of the probes mentioned above, which are used to determine whether a transition of the operating point is required. Details on how this decision is taken are presented in Section 6.2.2. Ultimately, a transition is asserted by setting the FSM input signals `vup`, `vdown`, `fup` and `fdown`, following a typical transition scheme as shown in Fig. 6.3: when a VF pair is stepping up to run logic faster, the VR output must be brought to higher voltage, before increasing the frequency to

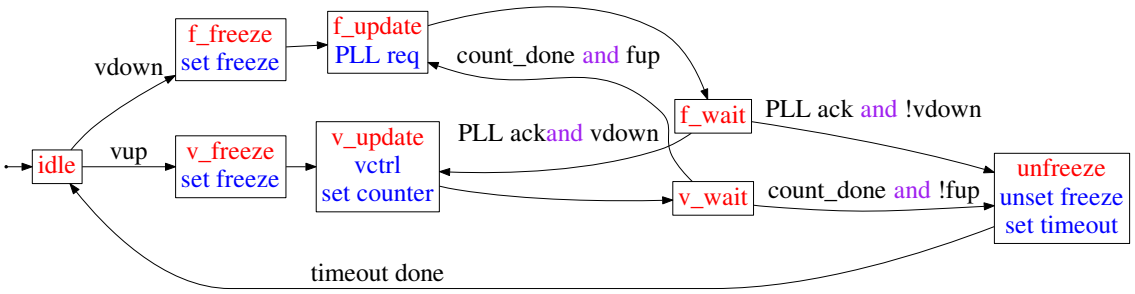


Figure 6.3: Finite-state machine for DVFS control.

avoid timing violations. For the same reason, when stepping down, frequency has to be decreased first. Hence, the FSM steps are inverted.

Note the clock gating logic in the lower-right part of Fig. 6.2. Gating is activated on any transition to preserve functional correctness of the accelerator. The DVFS is designed, instead, to be robust to the transition of the clock frequency, as freezing this logic would lead to a deadlock condition. During the actual transients, corresponding to the FSM states `v_wait` and `f_wait`, a watchdog is set to go off after the transient time of the VR. In addition, when both frequency and voltage have been updated, a configurable timeout is set to allow a sweep of the temporal granularity. The minimum timeout is set to 64 cycles, which corresponds to a conservative transient time for IVRs. To improve the robustness of the design, a request-acknowledge protocol between the PLL controller and the FSM ensures that the transient time has elapsed before disabling clock gating.

6.2.2 Global Software Supervision

High-throughput Accelerators rely on long and frequent data transfers. The experimental results of Section 6.3 show that mitigating resource contention enables significant energy savings when considering the aggregate system workload. In addition, the ability to adapt to low-level dynamics to regulate traffic improves efficiency by reducing the execution time while dissipating on average less power. The DVFS controller supports four main policy settings and each policy is configured with parameters that determine the conditions under which the operating point must change. Table 6.1 summarizes all the settings for the DVFS policies applied during the full-system experiments.

POLICY	OPERATING POINT	WINDOW (CYCLES)	THRESHOLDS TRAFFIC	THRESHOLDS BURST
PN0	1.0V 1.0GHz	-	-	-
PN1	0.90V 0.9GHz	-	-	-
PN2	0.80V 0.8GHz	-	-	-
PN3	0.75V 0.6GHz	-	-	-
PT[4-14]	variable	131,072-64	4,096-32	-
PB[15-25]	variable	131,072-64	4,096-32	114,688-56

Table 6.1: Set of policies used for design space exploration. Each policy results from combining different settings for each configuration parameter.

1. `POLICY NONE` (PN) simply imposes to maintain a fixed VF pair for the entire execution of the accelerator. Each domain still benefits from the VFs capability to quickly switch between on and off states. However, no operating point transition can occur, independently from the information recorded through the probes.

2. `POLICY TRAFFIC` (PT) is based on the observation of back-pressure signals at the interface between a tile and the interconnect. When DMA transactions are held from accessing the NoC for more than a configurable number of cycles, within a time frame, the controller issues a step-down command for the entire VF domain. The speed and dissipated power of accelerators decrease together with the packets injection rate for all the tiles belonging to the domain. When congestion clears, voltage and frequency are stepped back to the fastest operating point to improve performance. While the hardware controller enforces the policy based on local conditions, the software application invoking the accelerators can update the threshold values and time frame length to tune the policy for the current task.

3. `POLICY BURST` (PB) combines the observation of the traffic at the local interconnect interface with the information on the accelerators status. In particular, there are three distinct high-level conditions: (i) computation, (ii) data transfer and (iii) overlapping computation and data transfer. An ideal accelerator would achieve maximum efficiency by remaining in state (iii) for the entire execution time. This is prevented by a non-ideal interconnect and by the contention for memory access. Furthermore, even without considering back-pressure from

the environment, data dependencies and complex communication patterns make a perfect match between communication and computation time be unfeasible. Hence, the accelerators always spend part of their active time in states (i) and (ii). PB observes for how long, within a time frame, the accelerators are in these two non-ideal states, while monitoring traffic as well. A step-down command is issued if back-pressure occurs too often or if the ratio between communication and computation time grows beyond a configurable threshold. This scenario usually corresponds to the activation of state (ii). When, instead, accelerators spend more time in state (i), thus computing at a lower rate than the available bandwidth, a step-up command is asserted. Note that all thresholds are configurable by software to tune the policy for each target application scenario.

4. `POLICY LIMIT` (PL) can be combined with the other policies to ensure a fair distribution of the power envelop across multiple accelerators. It consists in a DVFS supervisor daemon which prevents all accelerators from running at maximum speed and power dissipation at the same time, according to a configurable aggregated power envelop. Most likely, in fact, only a portion of the available accelerators can be allowed to run at full power at the same time. Considering the example of multiple independent FFT2D kernels, running on different data sets, two scheduling approaches can be adopted. One approach is to run at high speed as many FFT2D accelerators as allowed by the power budget and schedule sequentially the remaining ones. While this approach may have some benefits, a balanced scheduling is preferred to maintain fairness among all running accelerators. This second approach aims at running concurrently as many FFT2D tasks as possible, while guaranteeing not to exceed the overall power budget. The daemon scans the system, checking for active accelerators. When the supervisor detects that the system is close to exceed the allotted power budget, it starts to forbid some DVFS controllers from running at the fastest operating point. This limitation extends to the other points as the number of active accelerators increases or the power budget decreases. To ensure fairness, at every scan the daemon updates the VF domains' priority. Hence, during the next system scan, an accelerator that was previously throttled is allowed to speed up. Note that if PT or PB are enabled, they are not allowed to step through all available operating points.

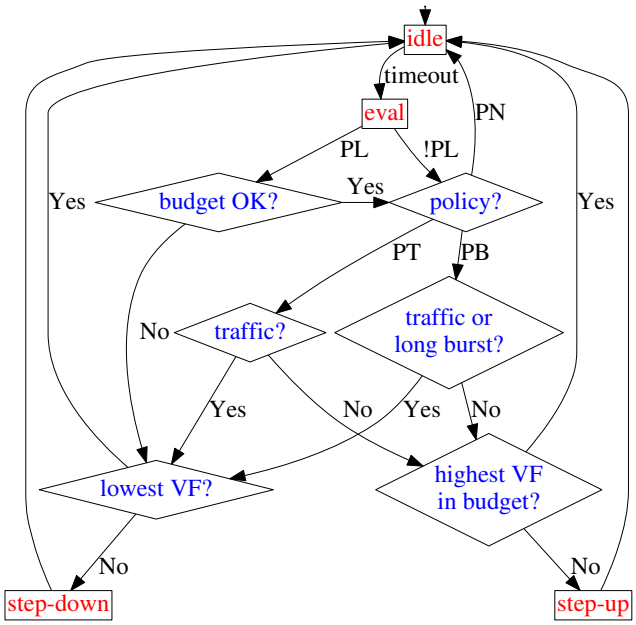


Figure 6.4: DVFS policies flow chart.

The flow chart in Fig. 6.4 provides an overview of the interaction of the policies with the DVFS actuation logic. If the controller ends either in the state "step-down" or "step-up", then a VF transition is initiated, following the mechanism described in Section 6.2.1.

6.2.3 DVFS Emulation for Design-Space Exploration

Thanks to the ESP infrastructure, the analysis of the fine-grained DVFS policies can be performed by deploying several SoC instances on FPGA. The emulation of frequency scaling, combined with the power estimates from a standard RTL flow, allows designers to determine the total energy consumption of the accelerators and tune the power-management policies.

FPGA-Based Frequency Scaling. Modern FPGAs feature several clocking resources which are typically required to support a wide variety of I/O protocols, such as PCIe, Gigabit Ethernet, DDR, etc. For example, the Xilinx Virtex-7 XC7V2000T FPGA [Xilinx, 2016] can use up to 24 PLLs. Each PLL is capable of generating six different frequencies, all related to the input reference clock multiplied by a configurable $\frac{m}{n}$ factor. For example, let us consider the four operating points reported in Table 6.2 for an industrial 32nm CMOS technology. The fastest VF domain operates at

ACCELERATOR	AREA (um^2)	ENERGY PER CLOCK CYCLE (pJ) 1.0V 1.0GHz	0.9V 0.9GHz	0.8V 0.8GHz	0.75V 0.6GHz
FFT2D	828,641	75.1940	64.9284	56.7607	55.8396
D-FILTER	694,050	28.8244	23.6830	19.6212	19.4185
IMAGE-WARP	362,945	36.0161	29.9815	25.7994	25.5095
PFA-INTERP1	478,905	198.6252	155.0892	119.2728	108.2894
PFA-INTERP2	565,417	169.9127	133.1610	103.0760	94.3365
MATRIX-ADD	87,446	3.8950	3.0112	2.3365	2.2897
GRADIENT	1,782,904	42.0945	31.9909	24.3755	25.0221
DEBAYER	698,765	22.8176	17.4281	13.7933	13.7859
GRAYSCALE	418,337	13.3011	10.1874	7.9643	8.0831
HESSIAN	1,251,278	34.9816	27.0928	21.1181	21.5305
MATRIX-MULTIPLY	123,329	5.8928	4.5782	3.5961	3.5816
RESHAPE	58,939	3.0950	2.3965	1.8703	1.8045
SD-UPDATE	1,457,124	38.1158	29.4261	22.7630	23.3643
STEEPEST-DESCENT	1,595,080	39.1712	29.6789	22.6179	23.3072
MATRIX-SUBTRACT	600,697	1.0564	11.8947	9.1569	9.4114
WARP	451,177	25.5707	20.0071	15.7157	15.3780
CHANGE-DETECTION	1,463,797	146.3342	113.3166	86.7009	81.8636

Table 6.2: Energy estimates for each accelerator at different operating points in a 32nm industrial CMOS technology.

1.0V and 1.0GHz. In order to model the scaling of the accelerator for the operating point at 0.8GHz, the multiplying factor should be $\frac{16}{20}$.

Differently from the ASIC version, the Virtex-7 PLL does not have native support run-time setting of its output frequency without making use of the partial reconfiguration capabilities of the FPGA [Xilinx, b]. However, this may lead to a time overhead much larger than the dynamics that need to be emulated for frequency scaling. Hence, instead of reprogramming the PLL division factors, multiple clock outputs are configured at design time, where each of them matches the frequency ratio of a different operating point for the target CMOS technology. For example, considering the experimental setup reported in Table 6.2, there are four clock outputs per region. As a consequence, some logic must be instantiated to dynamically switch the clock frequency of the VF domain. Unfortunately, most of FPGA clocking resources are not directly exposed to the users. For instance, simple components (e.g. clock buffers) are usually automatically instantiated and placed by the FPGA synthesis tool. More advanced basic blocks (e.g. PLLs) are instead wrapped in black box

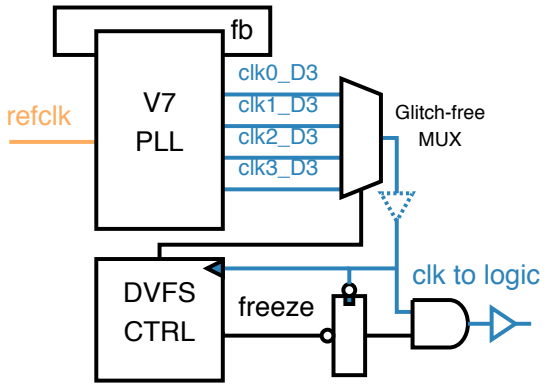


Figure 6.5: FPGA clock generation for frequency scaling.

IPs, together with the low-level physical layer of the instantiated I/O protocol [Xilinx, a]. In order to enable FPGA frequency scaling, both the PLL and the clock-buffer primitives must be manually instantiated. Fig. 6.5 shows a high-level block diagram of the clocking logic that implemented on the FPGA. Besides the PLL, the major building blocks of this circuit are a *glitch-free clock multiplexer* and the *clock buffers*. The clock multiplexer is a simple component that is directly provided as a primitive in the FPGA library. It allows the circuit to switch between two clocks with the guarantee that the period from one rising edge to the other will always be at least as large as the period of the slower clock. In addition, the high-level pulse width is always preserved. This is achieved first by disabling the clock that is currently driving the output on its high-to-low transition and then by enabling the newly selected clock after the transition from high to low level. The provided primitive includes a global clock buffer at the output, which drives the clock tree of a section of the FPGA. From a functional viewpoint, a tree of such multiplexers would enable switching among four clocks. However, two major limitations apply: (1) a limited number of global buffers can be placed in a single design (32 in the case of the XC7V2000T [Xilinx, a]); and (2) cascaded clock multiplexers must be placed in adjacent sites. The first constraint is what determines the maximum number of clock domains that can be obtained on a single FPGA, which would be severely reduced by placing three buffers for each clock multiplexer. Furthermore, the second constraint causes the design to fail place and route. Note, in this regard, the clock buffer placed after the clock-gating logic in Fig. 6.5. This buffer is mandatory, because the AND gate is mapped to a look-up table (LUT) on the FPGA, which cannot drive the clock tree. Using such signal to clock the accelerators in that domain would lead to severe timing violations. Since there is no sequential element between the multiplexer and

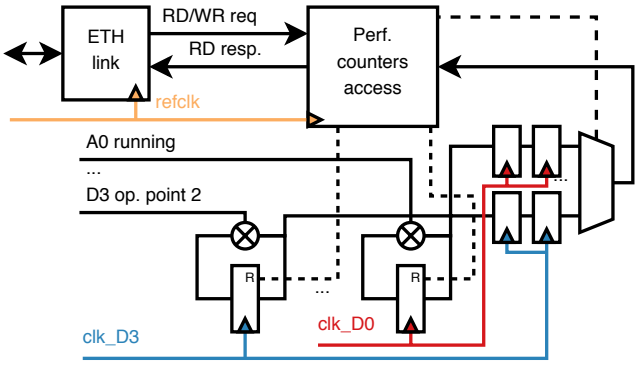


Figure 6.6: Access logic for probes and performance counters.

this buffer, however, using three primitive clock multiplexers determines a scenario in which three different buffers are cascaded to a fourth one. Under this condition the design is destined to fail the implementation step because a legal placement for the global buffers does not exist. This issue can be solved with a custom clock multiplexer featuring a single optional output buffer, which is represented with a shaded line in Fig. 6.5. This is critical to achieve timing closure for some paths within the DVFS controller. Removing such buffer from a few DVFS controller instances, however, can still produce a design that meets timing. By replicating this block, it is possible to implement an ESP instance with up to twelve domains in the target FPGA where the DVFS controller can correctly emulate the frequency scaling. Thanks to a user-guided placement of the clock buffers, the design closes at 100 MHz as the fastest frequency. Other PLL frequencies are accordingly set to match the ratios for the different operating points.

Energy Estimation Flow. Performance statistics from the DVFS controller are exported via an Ethernet interface. Fig. 6.6 shows the corresponding logic. These counters contain information about the number of cycles spent by each accelerator in each operating point. Energy consumption is determined by combining these data with the corresponding power models of each accelerator. Specifically, the RTL implementation of the accelerators is obtained by following the methodology described in Section 4.1. The HLS tool is configured to run two different scheduling tasks and generate the RTL implementations for both ASIC (industrial 32nm CMOS technology) and FPGA (Xilinx Virtex-7 XC7V2000T FPGA) with nominal operating points of 1.0 GHz and 100 MHz respectively. The RTL for ASIC is used for determining the power consumption of the accelerator in the target SoC, while the FPGA implementation is integrated in ESP for emulation.

The energy consumption of each accelerator is computed as: $E = \sum_{i=0}^{N} E_i^c * C_i$, where N represents the number of operating points, while E_i^c and C_i represent the energy consumption for clock cycle and the number of clock cycles spent by the accelerator in the operating point i, respectively. The total energy is then obtained by aggregating the energy consumption of all accelerators. For each point, the energy E_i^c per clock cycle spent by the accelerator is evaluated by combining estimates of the average power consumption P_i with the clock period T_i as $E_i^c = P_i * T_i$, while the power consumption is composed of a dynamic and a static part as $P_i = P_i^{dyn} + P_i^{leak}$. In this way it is possible to estimate both static (i.e. leakage) and dynamic power of the accelerators. The former mostly depends on the cells that are selected during logic synthesis, while the latter depends on the activity of the accelerator. For this reason, the switching activity is extracted with a fully-annotated simulation of the gate-level netlist at each operating point. Note that, due to the computational time issues of this task, the simulation is only performed on a subset of the data on which the accelerator will actually operate on the FPGA. However, since these accelerators have a fairly regular behavior and data access pattern, this simulation is sufficient to extract the average power consumption. The power characterization of each accelerator is then extended to all operating points. First, for each voltage value, the standard-cell and the SRAM libraries are re-characterized with Synopsys SiliconSmart ACE [Synopsys, Inc., b] to obtain the corresponding power and timing information. Representative numbers are extracted from detailed SPICE-level simulation to generate power consumption information in a Liberty NLDM format. The Liberty power information can then be used by power analysis tools (e.g. Synopsys Power Compiler [Synopsys, Inc., a]) to provide estimates of the overall power consumption, including information about the clock frequency related to the corresponding operating point. Timing analysis is also performed to verify that the circuit meets the timing constraints when operating at each lower voltage point.

The test scenarios include seventeen accelerators for various computational kernels from the PERFECT Benchmark Suite [Barker *et al.*, 2013], including the accelerators for the WAMI-App described in Section 4.1. Table 6.2 reports the energy estimates for each accelerator at four different operating points. These correspond to stepping down the voltage from 1V by 0.1V. This stepping value allows the voltage regulator to achieve high power conversion efficiency ($\sim 90\%$) [Kim *et al.*, 2008]. For the last operating point (0.75V), voltage is reduced by 0.5V only to be able to run at 0.6GHz without incurring timing violations. Lower operating points are not used since they may

introduce errors in the computation, especially for SRAMs [Kumar *et al.*, 2009]. The values of Table 6.2 report an energy reduction of about 20% for each step across the operating points. The last operating point, however, has a gain of less than 10% with respect to the previous point, mainly due to the smaller stepping value.

Differently from accelerators, the NoC is assumed to be always running at the fixed nominal frequency. In fact, applying DVFS to the interconnect on a system integrating many high-throughput accelerators, each relying on the NoC for DMA transactions, does not improve the system energy efficiency: even though scaling the supply voltage would reduce both the instant and average power of the NoC, the resulting slow down of the accelerators would lead to much longer execution time, resulting in higher energy consumption.

6.3 Multi-Accelerator Test Scenarios

Three case studies, based on ESP, are presented. Each of these is built using a composition of tiles similar to what shown in Fig. 6.1. Each ESP instance has 1 CPU, 1 I/O tile, and 2 DRAM controller tiles. The three case studies differ for the number and types of accelerators, specifically: (1) MIX features 5 independent heterogeneous accelerators (each duplicated); (2) TWELVE FFT2D features 12 independent homogeneous accelerators (12 copies of the FFT2D design); and (3) WAMI-App features 12 heterogeneous accelerators that depend on each other to provide a complete implementation of a computer-vision application. For each case study a similar set of experiments are repeated for all the policies of Section 6.2.2. The goal is to understand the impact of fine-grained DVFS in terms of both spatial and temporal granularity. This design-space exploration is done by considering all the configurations of the policies listed in Table 6.1. For policy PN all operating points are applied, one at a time, as a fixed VF pair to a domain. For the DFVS policies PT and PB the parameter *window* is swept to obtain different temporal DVFS granularity. Parameters *traffic* and *burst* are also varied across the experiments within the ranges shown in Table 6.1.

MIX**: heterogeneous independent accelerators.** This case study features 10 accelerators (2 FFT2D, 2 D-FILTER, 2 IMAGE-WARP, 2 PFA_INTERPOLATION1 and 2 PFA_INTERPOLATION2) that run continuously and concurrently, each presenting a different and specific pattern in terms of interleaving/overlapping computation (with the data in its local memory) and communication with the off-chip DRAMs.

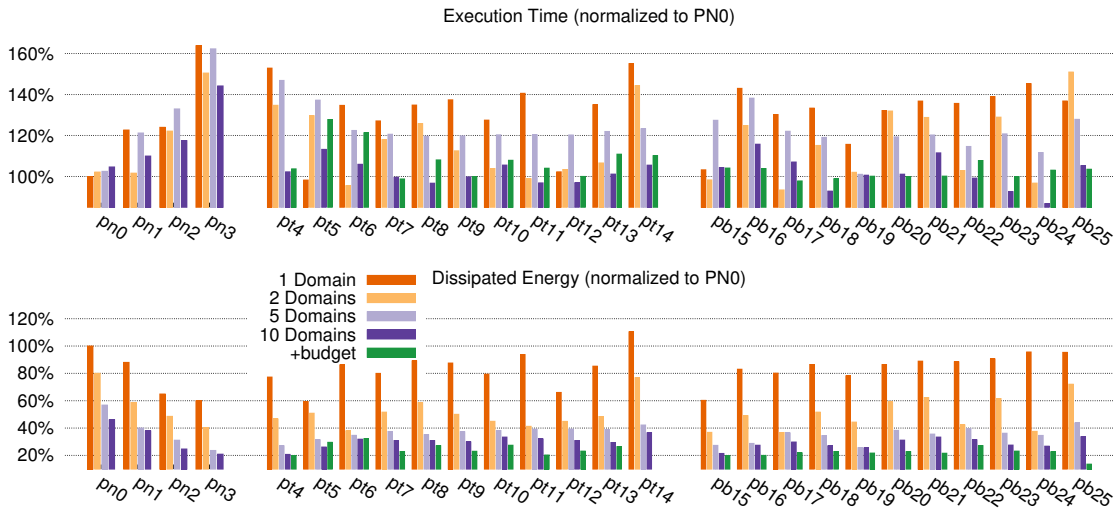


Figure 6.7: Scenario MIX: Normalized dela Normalized delay and energy savings for different DVFS policy and VF domain settings.

Fig. 6.7 collects the experimental results in terms of delay and energy savings for different VF domains and DVFS policies. The two bar diagrams have a similar structure: they are grouped based on the policies and labeled with an acronym that is a combination of the first two columns of Table 6.1: e.g. the first four groups correspond to the application of PN with the four different operating points in Table 6.1, respectively; similarly, the next two sets of eleven groups correspond to all possible applications of DVFS policies PT and PB, respectively. Within each group, the bar color corresponds to the application of the policy for a given number of VF domains: e.g. the first bar (dark orange) of the group labeled *pn0* corresponds to the application of policy PN to all accelerators that are part of a single VF domain; in contrast the fourth bar (violet) of this group corresponds to the application of the same policy to every accelerator, each stand-alone in a dedicate VF domain[2]. Each group of the DFVS policies PT and PB presents a fifth bar (black) that corresponds to the application of policy PL on top of the case of the fourth bar (violet), as explained in Section 6.2.2: i.e., PL combines the DVFS supervisor daemon with either PT or PB for the case of 10 VF domains.

[2] Since *pn0* keeps the same fixed VF pair (1V, 1GhZ), the different behaviors are due to the fact that the decision of turning off an accelerator can be taken only when all accelerators in a VF domain are ready to be turned off.

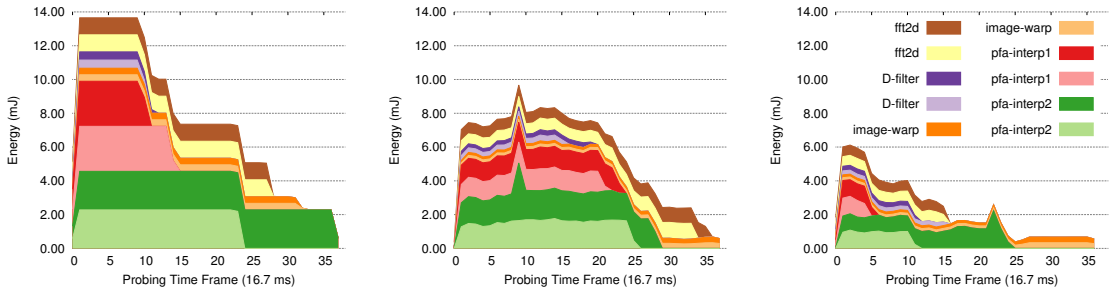


Figure 6.8: Scenario MIX: energy breakdown over time for *pn0* (left), *pb25* (center) and *pb25* with PL (right).

The height of every bar is normalized with respect to a baseline, which is always the first bar in the diagram (i.e. policy *pn0* applied with one single VF domain).

The results of Fig. 6.7 do not show a clear trend while increasing the temporal granularity for DVFS policies PT and PB. The reason is that the specific data-transfer pattern of each accelerator directly affects the statistics measured by the DVFS controller. When temporal granularity and policies thresholds are not properly configured for the accelerator's specific traffic signature, both energy savings and delay are penalized. On the other hand, there is a clear correlation between energy and spatial granularity of DVFS: independently on the policy (bar group), increasing the number of domains yields usually a reduction of the delay and *always* considerable (more than 50%) energy savings; this is the case even if DVFS is not used (PN policy). The reason is that the DVFS controller can take better decisions when its work is dedicated to a single tile.

Across all policies, *pb24* delivers the best delay improvement (10 times less than *pn0*) while *pb25*, i.e. combining the PB policy with the supervisor daemon, achieves the largest energy saving (i.e. about 15% of the energy spent with *pn0*). Fig. 6.8 shows how the combination of the local fine-grained hardware policy and the software supervisor can achieve this result: each chart displays the aggregated energy that is spent over time for the execution of an experiment with a particular policy. Each colored area shows the energy-delay product for one accelerator. The units on the horizontal axis are probing time frames of 16.7ms, which is the time allowed to the Ethernet interface to collect statistics from all probes. The chart on the left (policy *pn0*) highlights that all accelerators dissipate almost the same amount of energy at every time frame, until completion. Conversely, the central figure shows how the DVFS controller under policy *pb25* modulates the energy dissipation during the execution. Interestingly, however, the variation of the energy over time is very similar across

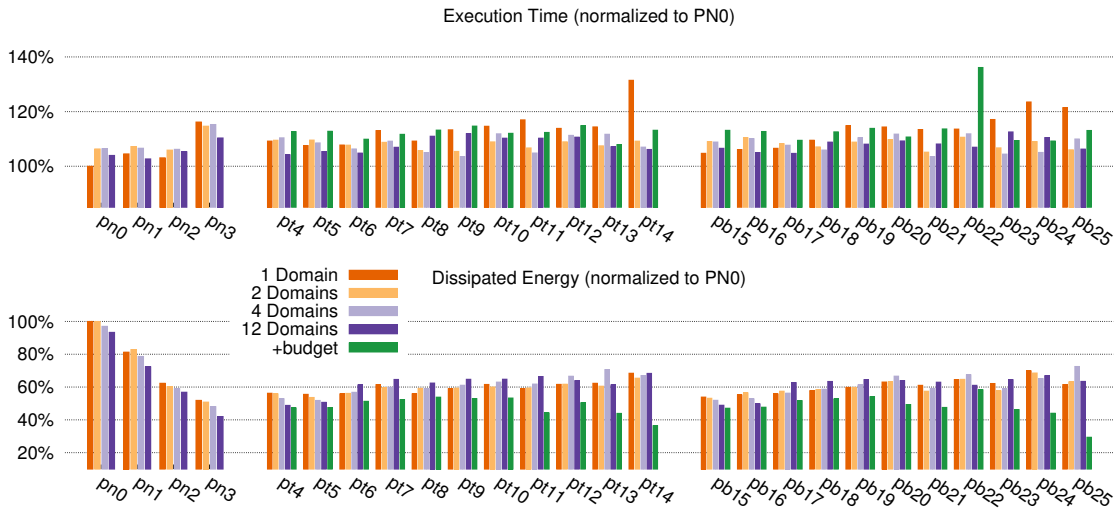


Figure 6.9: scenario TWELVE FFT2D: Normalized delay and energy savings for different DVFS policy and VF domain settings.

most of the domains. Note, in fact that the thickness of the filled lines remains visibly constant over time for each accelerator until completion. This scenario suggests that the decisions of a DVFS controller, based on the traffic at the local interconnect, may be suboptimal if taken simultaneously by all other controllers in the system. Finally, the right chart of Fig. 6.8 confirms the benefits of activating policy PL, by means of a daemon that supervises the hardware controller based on the information obtained scanning the system every 10 ms. All areas shrunk considerably leading to a major decrease of the energy-delay product. The unbalanced bias that the daemon gives to the DVFS controllers reduces the interference across the accelerators' traffic patterns and, therefore, the accelerators spend less time dissipating power while waiting for a transaction to complete. The result is an energy savings of more than 50% with respect to the same policy with no software supervisor.

TWELVE FFT2D**: homogeneous accelerators.** In the second case study, an accelerator for the ubiquitous FFT2D kernel is replicated twelve times. Compared to the results of the previous case study, Fig. 6.9 and Fig. 6.10 present much less variation across the experiment runs with respect to both temporal and spatial granularity. In particular, by excluding the black bars, that correspond to

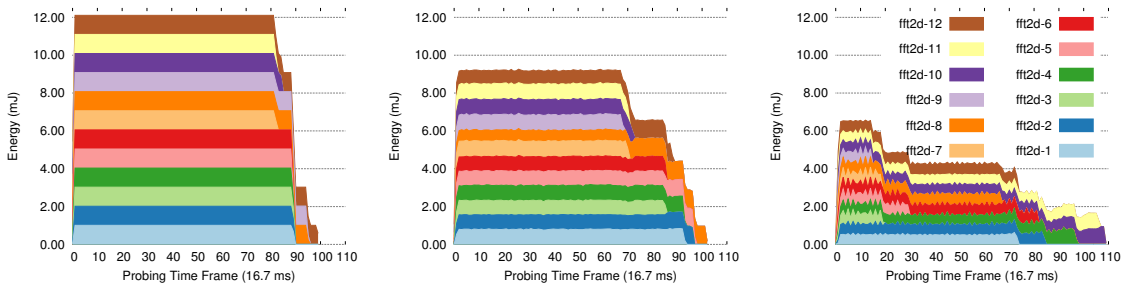


Figure 6.10: Scenario TWELVE FFT2D: energy breakdown over time for *pn0* (left), *pt14* (center) and *pt14* with PL (right).

the activation of PL, these runs show similar energy savings for the cases of two, four and twelve domains. Measuring the NoC injection rate and the traffic at the two memory controller tiles helps understanding this behavior: as soon as more than two FFT2D accelerators are activated, the queues at the memory tiles interfaces get quickly filled up and all tiles start receiving back-pressure from the NoC. This condition of extremely high congestion forces all regulators to slow down, thus giving more slack to the DRAM and the NoC to complete the pending transactions. As soon as the traffic decreases below the configured threshold, however, all accelerators tend to speed up again, thus bringing back the congestion. Such cyclic behavior is confirmed by the comparison between the chart on the left and the one in the middle of Fig. 6.10. Policy *pt14*, in this case, is adding noise to the energy distribution over time, as confirmed by the (slightly visible) ripple in the central chart. On the other hand, as in the MIX case study, the activation of policy PL with the daemon supervising the hardware controller, brings more than 50% extra energy savings and the twelve FFT2D accelerators complete their execution consuming 38% of the baseline energy.

WAMI-App: accelerators with data dependencies. The previous experiments focused on applying the DVFS policies to concurrent, but *independent* accelerators. Complex SoC applications, however, are usually implemented as the composition of many interacting accelerators with data-dependency relations: e.g. one accelerator produces input data for other accelerators, which can start to execute only after the first terminates. The twelve accelerators for the WAMI-App presented in Chapter 4 and the ESP kernel-thread library discussed in Section 4.3 are used as a case study to analyze how *inter-dependent* accelerators *directly* affect each other.

Recall that WAMI-App consists of four main algorithms: the DEBAYER filter, the RGB-TO-GRAYSCALE

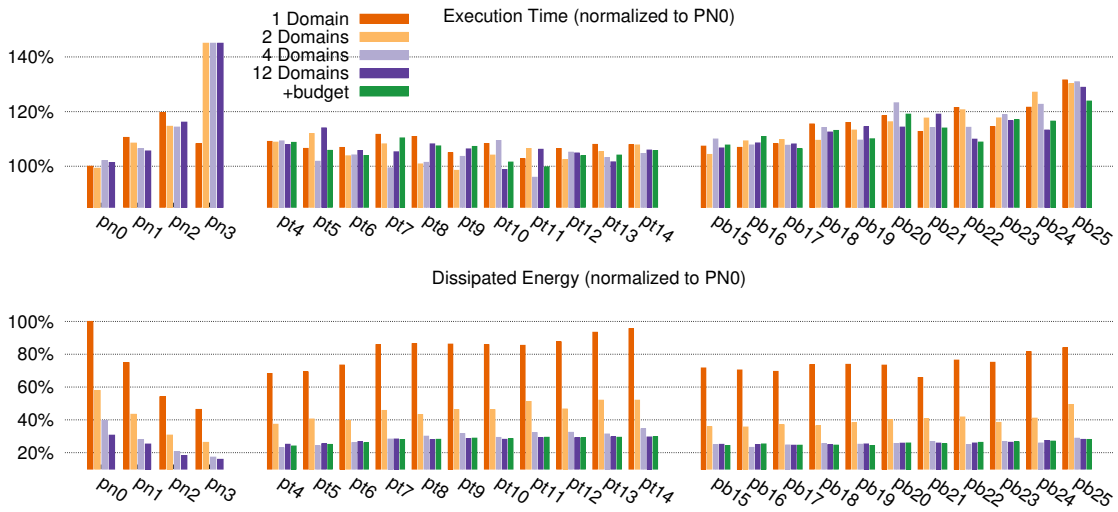


Figure 6.11: Scenario WAMI-App: Normalized delay and energy savings for different DVFS policy and VF domain settings.

conversion, the LUKAS-KANADE image alignment and the CHANGE-DETECTION classifier. To maximize the available parallelism LUKAS-KANADE is implemented with nine different accelerators [3]. The other three algorithms, instead, are implemented with a single accelerator each. The block diagram of Fig. 4.2 highlights the data-dependency relations among the WAMI-App accelerators and the *potential* for parallel execution: e.g., MATRIX-MULT must run after SD-UPDATE and INVERT-GJ, which instead can run concurrently. The data-dependency relations apply to the processing of a single input frame. Overlapping the processing of multiple frames in a pipeline fashion allows more accelerators to execute in parallel. The ESP multi-threaded library allows designers to exploit such parallelism by having each thread invoke a distinct WAMI-App accelerator through its driver. Still, independently on the size of inter-thread queues (see Section 4.3) allocated in memory, the resulting parallelism of the WAMI-App case study remains somewhat limited due to: (i) the fact that the queues have always a finite size and (ii) a heavily unbalanced distribution of the execution time. The latter also corresponds to an unbalanced distribution of the energy consumption: Table 6.2 confirms that CHANGE-DETECTION accounts for almost 40% of the energy spent per cycle by all accelerators. For this reason, the results of Fig. 6.11 show that most policies have a modest impact on energy sav-

[3]The function INVERT-GJ was implemented in software.

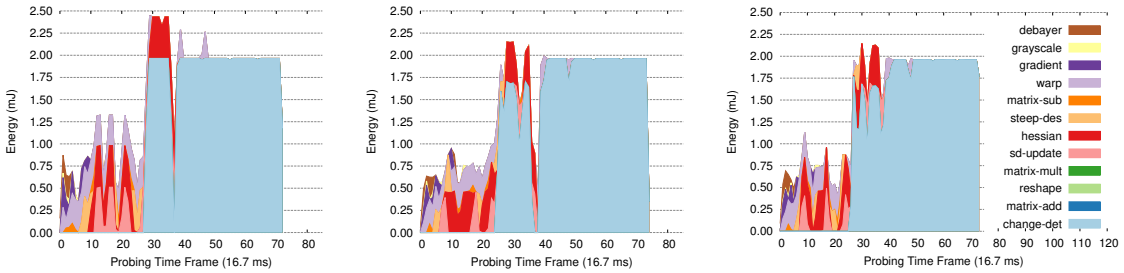


Figure 6.12: Scenario WAMI-App: energy breakdown over time for *pn0* (left), *pt14* (center) and *pt14* with PL (right).

ings, with little variations when moving towards finer temporal granularity. With respect to spatial granularity, for most policies the largest improvement is obtained when moving from two to four VF domains. Notice that four is the largest number of accelerated threads that can perform useful computation in parallel. For the WAMI-App, not even the software supervisor is capable of achieving higher energy savings, while its delay is similar to that of other policies. As for the previous case studies, the shape of the energy dissipation over time, reported in the three charts of Fig. 6.12, changes visibly for many accelerators when PL is enabled. The overall behavior, however, is dominated by CHANGE DETECTION, which not only is responsible for most of the power budget, but it is also running longer than any other accelerator.

The three sets of experiments above demonstrate the applicability of the ESP fine-grain power management service and how it can benefit the efficiency of heterogeneous SoCs. More importantly, the presented FPGA-based emulation infrastructure facilitates the rapid prototyping of heterogeneous SoCs, which vary in number and type of integrated accelerators. Furthermore, it enables the analysis of the impact of fine-grained DVFS, by combining frequency-scaling emulation with energy characterization data of the accelerators at different VF operating points. In summary, the ESP platform service for DVFS assists designers with pre-silicon analysis, tuning and design exploration of fine-grained power management of heterogeneous embedded systems.

Chapter 7

Scalable Interconnect and Communication

Some state-of-the art systems already integrate accelerators together with processors and GPUs, but the number of integrated components and the complexity of SoCs is expected to grow, as a response to the ever increasing demand for energy efficiency paired with the reluctance to sacrifice performance. It is therefore mandatory for a design framework to guarantee scalability at all abstraction levels.

The methodology, presented in Chapter 4 is inherently scalable, because it allows designers to focus on implementing one component at a time, while providing a flow to optimize and integrate components with a system-level approach.

The architecture that supports the methodology is also scalable, thanks to sockets that provide clean interfaces for the integration process and decouple the design of the IP blocks from each other. Through these interfaces, every IP can leverage the platform services to get seamless access to the shared resources of the SoC.

At a lower level, the interconnect is the backbone of the SoC and it is responsible for sustaining design scalability in terms of number of components, size of data sets, bandwidth and latency. As discussed in the previous chapters, a network-on-chip (NoC) enables a better distribution of the traffic towards memory, offers a natural barrier across clock domains, and is the ideal interconnect for scaling the size of the design and the number of integrated components.

Section 7.1 describes the details of the available ESP communication services and how to manage communication among long chains of accelerators, when memory access is the main bottleneck. Then, Section 7.2 talks about details of the ESP communication infrastructure. Finally, Section 7.3 explains how to keep complexity of place and route under control by adopting a hierarchical approach, which splits the interconnect into multiple independent NoCs, each of reasonable size, compatible with either fully synchronous or multi-synchronous design. Connections between different NoCs do not require shipping the clock with data and rely, instead, on asynchronous communication. This approach offers the opportunity to either implement a globally-asynchronous-locally-synchronous (GALS) system on the same chip or to split the design into multiple chips. The asynchronous communication has been tested on a multi-FPGA setup that enables the measurement of latency and throughput over the ESP NoC bridge.

7.1 Communication Platform Services

A typical application runs multiple computational kernels and each of these could be accelerated by dedicated hardware. These kernels may interact directly or require additional code that further processes the output of one kernel before having another kernel elaborate such data. The method shown so far in all test scenarios supports both cases: every accelerator interacts directly with main memory to load input data and store output data. Furthermore, communication across accelerators is implemented with a queue mechanism in memory, as presented in Section 4.3. When additional processing is needed in between two accelerators' runs, a software-only thread can take care of the intermediate computation. Alternatively, one of the two kernel-threads invoking the accelerators can include the additional code before or after the `io_ctl` system call. This approach creates a virtual pipeline of accelerators through memory and, as long as network and memory bandwidth are not saturated, the performance overhead is limited to the driver run time and the interrupt handling procedures.

Nevertheless, when there exists a chain of accelerators directly interacting among each other, allocating queues in memory for data transfers is a sub-optimal communication mechanism. In this case, memory queues not only incur needless power dissipation, but also increase execution latency and can potentially limit the throughput of the accelerators, especially under high resource

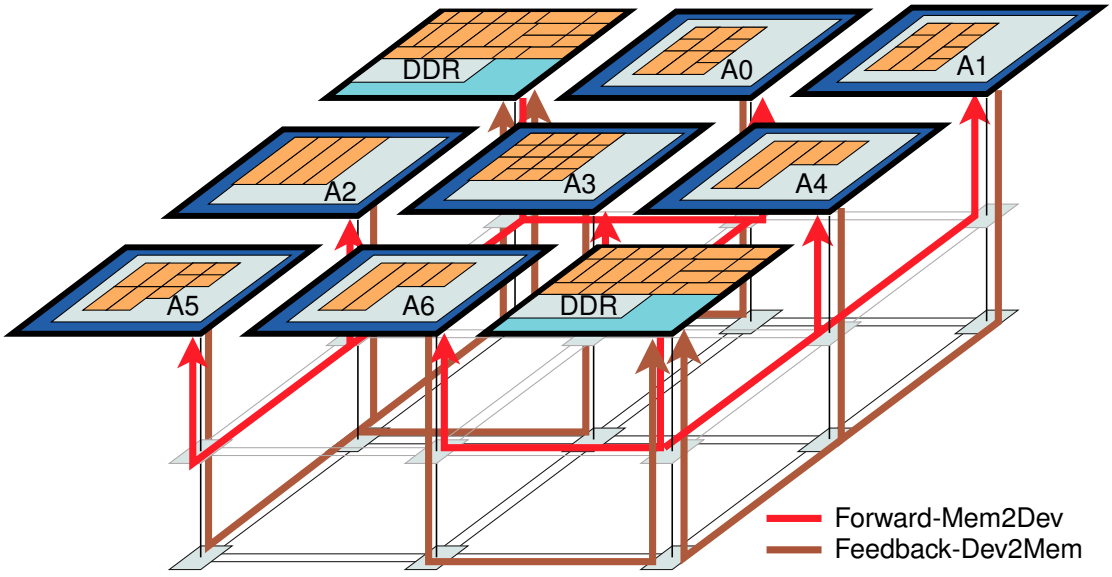


Figure 7.1: Communication through memory only.

contention. As an example, the drawing in Fig. 7.1 shows a hypothetical instance of ESP with two memory controllers and a mix of accelerators. The figure highlights all active communication paths, assuming the current workload is using all accelerators. Each accelerator initiates a DMA read transfer for every input data token and a DMA write transfer for every output data token. The forward memory-to-device (Mem2Dev) packets flow, shown in red, and the write-back, or feedback, device-to-memory (Dev2Mem) packets flow, in brown, travel on separate planes to avoid deadlock. In fact, when there are concurrent accelerators interacting with multiple memory controllers it is possible to create a waiting loop and cause a protocol deadlock [Concer *et al.*, 2010]. Virtual channels could also solve the deadlock issue. But, considering the length of the accelerators' DMA transfers and the throughput requirements, separate planes are the most suitable solution [Yoon *et al.*, 2013].

Even though the NoC can provide the accelerators with significant bandwidth, the two DRAM controllers might quickly saturate. Exploratory tests on FPGA show that with more than two to four accelerators, depending on their bandwidth demand, one DRAM channel saturates. When latency is not an issue and some throughput degradation is acceptable, saturated memory channels give the opportunity to save energy with DVFS, as discussed in Chapter 6. On the other hand, if performance is the primary design target, an alternative communication mechanisms is necessary

to relieve memory contention and improve the latency of accelerators. Chaining the execution of two or more accelerators is an effective way to do so, as it spares several transactions with memory, which are replaced by *point-to-point* (P2P) communication. The latter has lower latency and higher throughput, because it is only limited by the NoC, and not by memory. In addition, once the transaction has started, latency for P2P communication is deterministic on a packet-switched network, whereas external memory access may incur unexpected delays due to DDR conflicts in accessing a row, or to the necessary refresh cycles. Although chaining is a promising solution, it adds a strict requirement on the accelerators involved in P2P transactions: the output of the parent kernel in the dependency graph must match the input of the successor. This is generally true at a coarse granularity (i.e. the entire output of an accelerator run is typically the input for the next accelerator in the graph). The requirement for chaining, however, is stronger: since PLMs cannot usually store the entire data set of an accelerated kernel, chaining implies that every output-data token of the upstream accelerator matches the input-data token of the downstream accelerator.

Example. *The first two kernels of WAMI are* DEBAYER *and* GRAYSCALE. *The former produces as output an RGB image that the latter converts to "gray scale". With the ESP multi-threaded library, multiple frames are elaborated in parallel. Specifically,* DEBAYER *interpolates the first image; afterwards* GRAYSCALE *converts it, while* DEBAYER *interpolates a second frame. This coarse-grain pipeline is supported by memory that holds at least one entire input "Bayer" image and one output RGB image for* DEBAYER, *as well as one output grayscale-image for* GRAYSCALE. DEBAYER *and* GRAYSCALE, *however, can also be chained through P2P communication to create a finer-grain pipeline in hardware. Recall that* DEBAYER *reads stencils of* 5×5 *pixels to produce one pixel of the output image at a time. In the proposed implementation the DMA write transaction initiates after a full row of the output RGB image is stored in the PLM. In particular, the output memory is implemented as a ping-pong buffer to allow the overlapping of computation with write transactions.* GRAYSCALE, *on the other hand, reads one RGB row at a time and writes one row of the grayscake image. With this condition satisfied, it is possible to transfer directly the RGB row from* DEBAYER *to* GRAYSCALE *and then store only the output of* GRAYSCALE *into memory.* □

In principle, implementing a finer pipeline granularity is possible, even using memory queues, by having the accelerators trigger an interrupt for every transaction, rather than only once, after the entire data set has been processed. However, this scenario leads to the performance drawbacks shown in Section 5.1.1 with the example of SORT and DEBAYER being supported by a DMA buffer

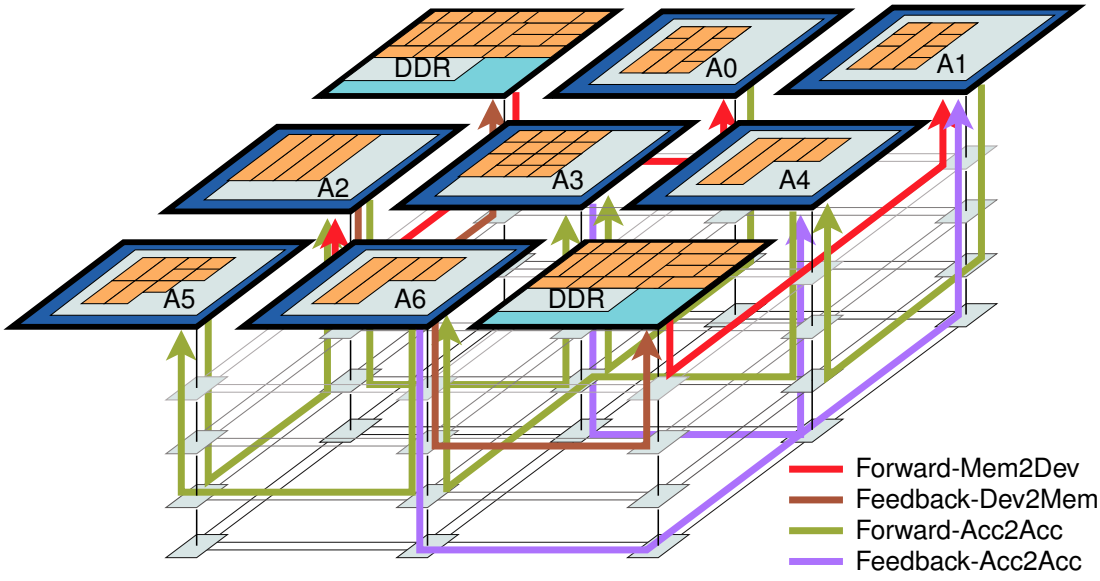


Figure 7.2: Communication through memory and point-to-point (P2P).

of insufficient size. That example suggests that software synchronization of concurrent accelerators should happen at a coarse granularity, such that the accelerator's execution time masks the latency of dealing with the operating system, the device driver and `pthread` primitives.

If chaining accelerated kernels is possible, ESP allows a user to replace the software pipeline, based on memory queues, with an actual pipeline of accelerators, based on P2P communication. The communication method does not need to be chosen at design time; instead, two special configuration registers are used to overwrite the default DMA controller (DMAC) behavior as follows. If the SOURCE-OVERWRITE register is different from zero, the DMAC ignores the accelerator virtual address and length from the accelerator read-request interface and waits for data to arrive from the accelerator specified in the special register. When the data arrive from the NoC, they are forwarded to the accelerator, as if they had been requested from memory. When the DESTINATION-OVERWRITE register is different from zero, the DMAC ignores the accelerator virtual address of the requested write transaction and reroutes the NoC packet to the accelerator specified in the special register, instead of memory. When the accelerator begins the actual transaction, data are forwarded to the NoC through the P2P service queues, similarly to the case of a standard DMA transaction.

Fig. 7.2 continues the example of Fig. 7.1 by updating the active links in the case of P2P communication. In this case, the most important thing to notice is that there are fewer accelerators

interacting with memory. In this example only A0, A1 and A2 read from memory (red Mem2Dev paths) and only A2 and A6 write to memory (brown Dev2Mem paths). In general the number of accelerators interacting with memory is application specific and depends on the data-dependency graph. Nevertheless, in most cases, P2P communication relieves significantly memory contention.

The "accelerator-to-accelerator" (Acc2Acc) paths correspond to the directed arcs in the dependency graph. Forward and feedback Acc2Acc packets must be differentiated to avoid potential protocol deadlock. In addition, it is necessary to mark feedback messages, because they won't typically be delivered during the first execution of the accelerators pipeline. A similar restriction on feedback paths was discussed in Section 4.3 when describing the API for the ESP multi-threaded library. Note that for the sake of simplicity, the example of Fig. 7.2 represents a physical NoC plane for each class of messages. When the fan-in of each accelerator is limited to one, however, three planes or virtual channels are sufficient to avoid protocol deadlock. Note, in fact, that while communicating through memory requires a "request-response" interaction between the accelerator and memory for read operations, P2P communication does not: the accelerator's read requests are ignored because input data are shipped through a write transaction, which requires no acknowledgment.

P2P communication for non-trivial data-dependency graphs. It is likely that some kernels in a data-dependency graph have fan-in or fan-out different from one. Both scenarios must be handled to guarantee proper functionality of the P2P platform service. The case of fan-in different from one is managed by having the special register SOURCE-OVERWRITE store the identifier of multiple upstream accelerators in the order their input is requested by the local accelerator. Informing the DMAC about the order of expected inputs is necessary, because with no flow-control, different accelerators may ship their result out-of-order. To keep the accelerator interface simple, as presented in Section 4.1, the DMAC uses the information in the special register to send credits on the feedback plane to the tiles of the upstream accelerators. When at least one credit has been received, the DMAC is allowed to inject the packet corresponding to DMA write into the NoC. Alternatively, the receiving tile would have to store incoming packets, until it's time to forward them to the local accelerator. While this is a viable solution, local memory is a precious resource, hence, in ESP it is not allocated for supporting out-of-order P2P communication.

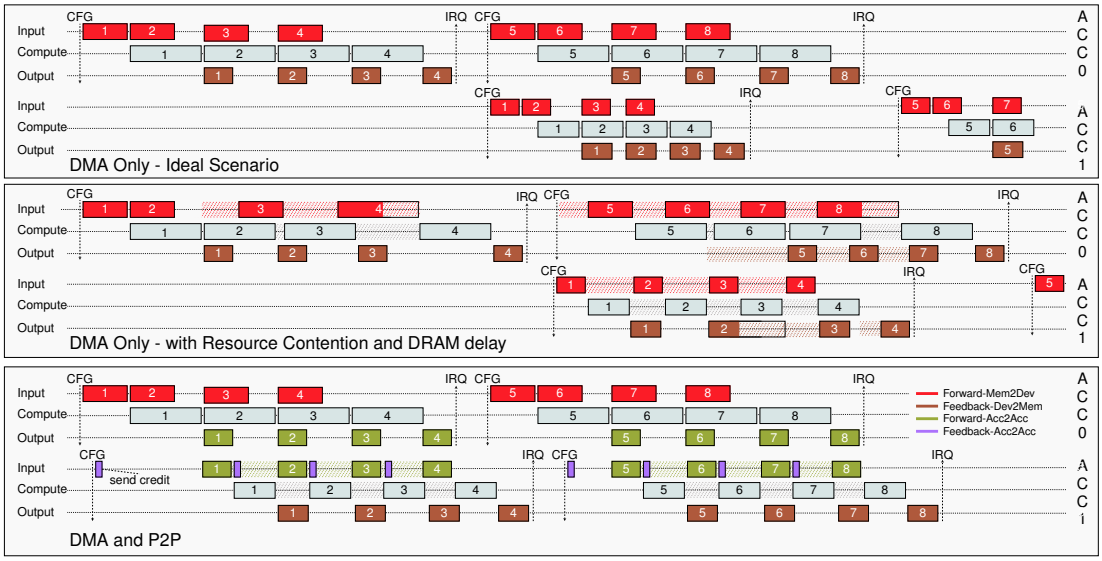


Figure 7.3: Sample execution of a pipeline of two accelerators; ACC0 and ACC1. Communication through memory in an ideal scenario (top); Communication through memory in case of resource contention and memory delay (middle); Communication through point-to-point service (bottom).

When fan-out is different from one, the DESTINATION-OVERWRITE register is used to store the identifiers of all downstream accelerators. Let us assume there are three different recipients listed in the special register. For every DMA write request, the DMAC ships two single-flit packets and the actual DMA write packet to the first destination. When the DMAC at destination receives the first single-flit packet, it stores the identifier of the next recipients and waits for the other packets. The second single-flit header is forwarded immediately to the second recipient. The DMA write packet, instead, is forwarded to both the local accelerator and to the second recipient. The latter will do the same for the third, and last accelerator in the chain. As an alternative, it is possible to leverage a network that supports multi-casting, but this falls beyond the scope of this work and therefore has not been implemented.

7.1.1 Accelerators Pipeline with P2P Communication

Beside relieving memory contention, P2P communication can actually improve performance of multiple dependent accelerators. This can be shown with a simple example of two accelerators, whose data tokens are compatible for chaining. The top Gantt chart in Fig. 7.3 reports accelerator

phases for ACC0 and ACC1 while they run in pipeline. The two accelerators are synchronized in software, where each kernel-thread configures the accelerators by invoking the driver and handles the interrupt once the accelerated task completes. The completion of the configuration phase and the interrupt request are marked with CFG and IRQ in Fig. 7.3. Data exchange happens trough memory queues at the granularity of the whole accelerator data set: as soon as ACC0 terminates its first run, ACC1 begins processing using the entire output from ACC0. This diagram is similar to the Gantt charts shown for WAMI-App in Chapter 4, except that they include the break-down of accelerators' phases.

The top chart in Fig. 7.3 shows an ideal scenario, in which memory is never delayed. Ping-pong buffering on the PLM for both input and output allows the overlap of computation and I/O phases. In addition, input and output phases from both accelerators are allowed to overlap. This is only possible by assuming to have dedicated DDR channels for each accelerator. Such condition, unfortunately, becomes harder and harder to satisfy as the number of concurrent accelerators grows beyond a few.

The middle chart still shows communication through memory, but it also shows the effects of memory delay and contention. For instance, the third input transaction of ACC0 should have initiated together with the second computation phase. However, memory access delay and the interference with the first output transaction result in some unexpected waiting time. This leads to a minor delay on the third computation phase as well. Similarly, the fourth input transaction gets delayed. In addition, this access lasts longer than the others, as the example shows the effect of a row-buffer conflict in memory. In this case, the fourth computation phase is stalled for a long time, impacting the overall performance of the accelerator. Note that writes are less likely to hit a long latency access, because they can be buffered. Nevertheless, under severe congestion, write buffers may get full, together with the NoC queues. Hence, even write transactions could be unexpectedly stalled. With two accelerators running, in addition to unpredictable delays, the chart shows how input and output transactions can only run interleaved, because they contend for the same DDR channel. Not surprisingly, even the DMA write operations are stalled because of high memory contention.

The bottom chart in Fig. 7.3 shows how P2P communication can drastically reduce computation latency. First, notice that both accelerators are configured to start roughly at the same time. ACC1 is configured to wait for data from ACC0, therefore there isn't a DMA read transaction. Instead, when

ACC1 requests a read transaction, the DMAC sends a credit to ACC0. The latter can then transfer data through a forward path as soon as the first computation phase completes. Because of ping-pong buffering, while the computation phase of ACC1 begins, another credit is sent backward to ACC0. Unfortunately, since the ACC0 computation lasts longer than the ACC1 computation, ACC1 cannot exploit pre-fetching to sustain its maximum throughput. This scenario suggests that when creating a hardware pipeline of accelerators, it is advisable to choose their implementation from the Pareto set, such that computation phases have similar latency. For more complex dependency graphs, this requirement translates into finding the maximum sustainable throughput, given by the slowest loop (i.e. the critical cycle) in the graph, and matching the throughput of the critical cycle to that of the others. If the component-level DSE doesn't produce sufficient Pareto-optimal implementations to satisfy this requirement, DVFS can be used to slow down those accelerators that are on the non-critical cycles, thus saving energy.

Despite the mismatch in execution time, the bottom chart in Fig. 7.3 shows that P2P communication eliminates contention for read and write transactions between ACC0 and ACC1, enables overlapping of output phases for ACC0 and input phases for ACC1, reduces the latency of the pipeline, and improves its throughput. Let us imagine to scale this example to a pipeline of multiple accelerators: contention would increase with the number of accelerators if only DMA transactions are enabled. Conversely, memory contention does not necessarily increase when more accelerators are added to the P2P-based pipeline [1]. Furthermore, complex chains of accelerators lead to more opportunities to reduce latency: indeed, each arc of the dependency graph corresponds to a potential performance improvement.

7.2 ESP Communication Infrastructure

This section dives into the details of how the ESP communication is handled over the NoC and how synchronization is achieved in a scalable manner. Fig 7.4 shows the block diagram of an entire ESP system, including the multi-plane NoC and all types of tiles that can be found in any instance of

[1] There might be circumstances where many, or most, accelerators need to interact with memory, despite the availability of the P2P service. For instance, if kernels share a large-size training set, even if they can transfer the main input and output directly with P2P communication, they still need to fetch and update training data in memory. In this scenario P2P communication still relieves memory contention, but with diminishing returns.

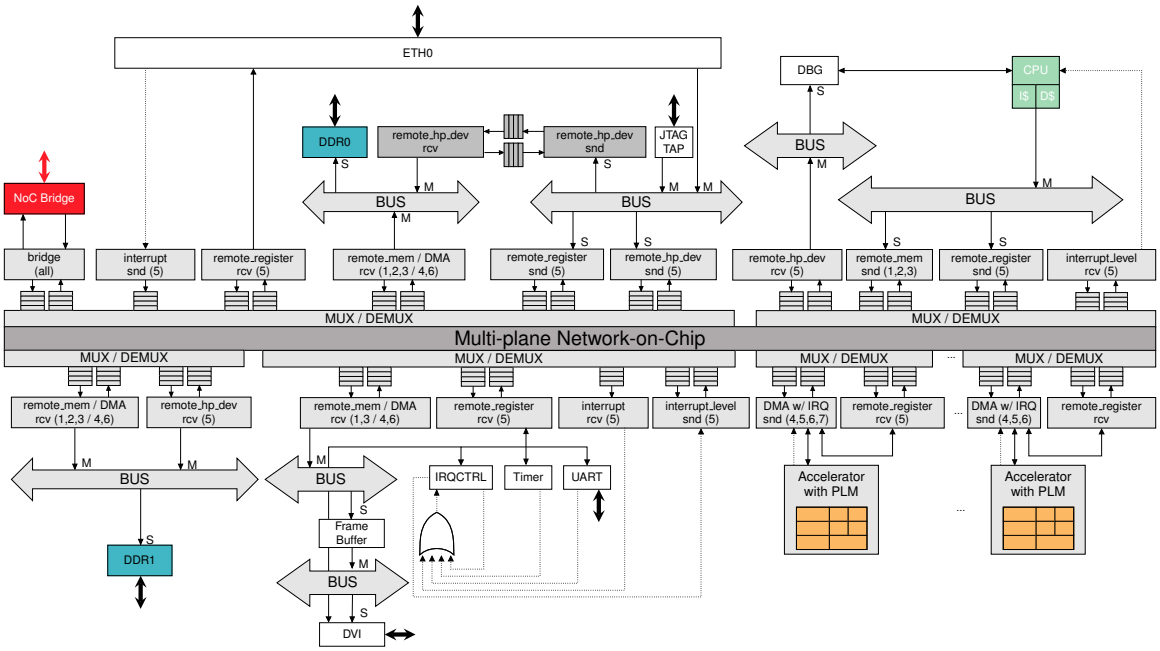


Figure 7.4: Details of the interconnect infrastructure for an instance of ESP.

ESP [2]. The multi-plane NoC consists of seven independent 2D-mesh networks, each assigned to a different set of message classes. The specific assignment is chosen to prevent protocol deadlock and to maximize the bandwidth for the accelerators. The latter is also the reason for preferring multiple physical planes to a single network with virtual channels [Yoon *et al.*, 2013]. A separation in the MUX/DEMUX blocks of Fig. 7.4 marks the boundaries of a tile. For instance, the top of the figure shows a memory and debug (MEMDBG) tile on the left and a processor (CPU) tile on the right, whereas the bottom portion shows, from left to right, a memory (MEM) tile, a miscellaneous (MISC) tile and two accelerator (ACC) tiles.

Tiles are composed of one or more IP blocks, which can be accelerators, processors or I/O peripherals, such as a DDR controller, an Ethernet PHY, a UART, or a video output. Alongside the IP blocks, each tile hosts the sockets, which in turn implement the platform services. Furthermore, when a service implies communicating with an IP located in a different tile (referred to as "remote"), the tile hosts a "proxy" that interfaces local components with the NoC. Each proxy implements a

[2]DVFS-related components are omitted.

thin layer of logic, acting as an adapter, which complies with the communication protocol of the IP that uses the corresponding service. This adapter translates IP-specific messages into packets for the NoC and handles request and response transactions according to a latency-insensitive protocol [Carloni, 2004]. Given the variability in terms of traffic and contention over the NoC links, in fact, it is not always possible to predict the latency of a round-trip transaction over the NoC.

Each proxy has one or more corresponding FIFO queue to interface with the NoC. Queues are used to separate message classes and hold packets, or a portion of them, before they can be injected to the NoC, or right after they have been received from the NoC. Queues are connected to a layer of multiplexers (MUX) and de-multiplexers (DEMUX) that enforce ordering among message classes and route them to and from the NoC plane they have been assigned. Both distribution across planes and ordering are necessary to ensure liveliness of the SoC. The actual access to a network plane occurs through the local port of the NoC routers. This port consists of two FIFO queues: one for input and one for output. When the power management service is enabled, these queues are implemented as dual-clock FIFOs that take care of synchronization between the clock domains of the network and the tile.

MEMDBG tile. Let us consider the MEMDBG tile first. The red box in this tile represents the inter-NoC communication service, implemented through a high-speed bridge that is discussed later in this section. Since the bridge must potentially forward all message classes from one NoC to the other, there are a tile-to-NoC queue and a NoC-to-tile queue connected to each plane of the network. Moving to the right, there is a proxy for interrupts, which receives interrupts from the Ethernet and forwards them to the remote interrupt controller in the MISC tile. This proxy only requires one queue from the tile to one NoC plane, i.e. Plane 5. Remote register access and transactions involving memory-mapped high-performance (HP_DEV) devices occur on Plane 5 as well. The main difference between register and HP_DEV is that the latter can transfer more than one data word per transaction. The last proxy connected to the network in the MEMDBG tile serves remote memory accesses from processors and accelerators. If the memory request originates from a processor, the proxy receives it from Plane 1. In case of loads, response messages return through Plane 3. Moreover, this proxy can send forward-request messages on Plane 2 to support a coherency protocol

across processors [3]. Conversely, if the memory request is a DMA transaction from an ACC tile, the proxy receives a packet from Plane 6. In case of DMA read, the response is sent through Plane 4 [4].

Note that proxies are marked as sender (snd) or receiver (rcv), depending on who is the initiator of the transaction. For instance, the Ethernet acts as a slave on the register-access interface, hence the proxy that supports this service is a receiver on the NoC and a master on the bus. On the other hand, the Ethernet has also a master-initiator interface to transfer data, and so does the JTAG "test-access point". The corresponding proxy is a slave on the bus and a sender on the network. Ethernet and JTAG share both the proxy and the bus, while the DDR controller slave interface sits on a second bus in the same tile. The bus is split to avoid deadlock while guaranteeing support for remote transactions with master initiators and slave targets that are potentially located in any tile.

Example. Assume that the CPU executes a load for an address that misses in cache and maps to `DDR0`*. In order to fetch the requested cache line, a remote memory read travels through Plane 1 over the NoC towards the* MEMDBG *tile. Simultaneously, a JTAG request could be issued to check on the CPU status. In such scenario, if both* CPU *and* MEMDBG *tiles had a single bus, a protocol deadlock would likely occur. The CPU or its cache, in fact, are holding the bus on the* CPU *tile, while waiting for the proxy to receive a reply on the network. Similarly, the JTAG holds the bus on the* MEMDBG *tile, while waiting for the debug unit to return the status of the processor. When the receiver proxies get the packets from the NoC, they request access to the bus, which is never going to be granted because neither the cache miss nor the debug access can be served. The buses remain locked with pending transactions that cannot complete.* □

To prevent protocol deadlock on every bus in the system, it is sufficient to split buses such that each of them can only be connected to one or multiple masters and a single slave proxy (i.e. a sender), or to one or multiple slaves and a single master proxy (i.e. a receiver). Exceptions are allowed for proxies, as long as proxies sharing the same bus do not share the same NoC plane. For instance, this is the case of the MEM tile on the bottom left corner of Fig. 7.4: the remote memory or DMA proxy is used to let processors and accelerators access the portion of the address space mapped to `DDR1`, while the HP_DEV proxy allows Ethernet and JTAG to access the DDR node.

[3] Three planes or virtual channels are sufficient to prevent deadlock with any directory-based cache-coherence protocol [Sorin *et al.*, 2011].

[4] As explained in Chapter 5, two DMA planes are necessary to prevent deadlock when multiple concurrent accelerators contend for more than one channel to external memory and each channel is located in a different tile.

When communication is needed between a master and a slave on the same tile, a pair of send-receiver proxies are used as a bridge across the two buses. This doesn't break the requirements for liveliness, because such proxies do not share a network plane with any other proxy. The MEMDBG tile hosts one of these bus-to-bus bridges to allow the Ethernet interface to access addresses mapped to DDR0.

CPU tile. The CPU tile at the top-right corner of Fig. 7.4 hosts two buses as well, however, direct communication through the bus is not required between the processor and the debug unit. The latter is a slave that responds to messages coming from the HP_DEV proxy on Plane 5, whereas the former is a master and issues both memory accesses and remote register operations. In addition, the CPU tile has a receiver proxy that forwards interrupt levels from the interrupt controller to the processor and sends back a message of acknowledgment, based on how the processor handles pending interrupts.

MISC tile. At the bottom of Fig. 7.4 there are the MISC tile, next to the MEM tile. This tile is where IP blocks that are shared among all processors in the system are located. Specifically, the figure shows a UART interface, a timer and the interrupt controller that act as slaves in the system by exposing a register-access interface. These three components, together with memory, are necessary to be able to boot an operating system, such as Linux: the UART allows the user to have a console interface; the timer enables periodic tasks, such as scheduling, by raising interrupts with a known time interval; the interrupt controller allows peripherals, including the timer, to inform the processor about relevant events in the system. The MISC tile hosts also two optional components that enable video output: a digital-visual interface (DVI) [DDWG, 1999] and a dedicated memory used as a frame buffer. The DVI slave interface receives configuration messages in the form of register accesses. In addition, a master interface is used to read the content of the frame buffer, i.e. the image to display. If the frame buffer were mapped to external memory, the DVI would generate a periodic traffic on the NoC to read the frame buffer and refresh the screen at the rate imposed by the digital-visual interface standard. The on-chip buffer, placed next to the DVI avoids unnecessary traffic when the image to display does not change. In addition, thanks to a dual-port interface, the frame buffer allows processors and accelerators to update the image through a remote-memory proxy without incurring delays in the screen refresh operation.

ACC tile. The last set of tiles on the bottom-right corner of Fig. 7.4 illustrates the structure of the accelerator tiles. As presented in the previous chapters, accelerators are configured by writing to a set of memory-mapped registers; hence there is a proxy for remote-register access on each ACC tile. In addition, there is an integrated DMA and interrupt proxy that forwards DMA requests and interrupts to the appropriate recipients in the system: Dev2Mem messages travel on Plane 6; Mem2Dev messages return trough Plane 4; and interrupts are delivered using Plane 5. If the P2P service is enabled, forward Acc2Acc and credits for feedback packets share Plane 6 with packets directed to memory. Consider, in fact, that most of the traffic on Plane 6 is generated by DMA write; with P2P communication, part of this traffic is diverted to an accelerator, rather than memory, but it can still travel on the same NoC plane. Moreover, note that credits towards accelerators that communicate on a feedback path are actually requests, which from the NoC viewpoint are considered forward messages. Feedback Acc2Acc packets, and credits for forward messages, instead, must be routed on a separate channel [5] (i.e. Plane 7).

7.3 Scaling the Size of the NoC

The first goal of the ESP methodology is to reduce the engineering effort in creating heterogeneous systems and integrating a wide variety of IP blocks. In order to achieve this target, scalability must be guaranteed at all levels of abstraction, including physical design. Scaling an ESP system from a few dozens of tiles to several tens or hundreds of them, however, requires additional support at the interconnect level. The communication infrastructure described in the previous section relies on fully-synchronous NoC planes. They connect various tiles, each placed in a dedicated clock domain. This results in a multi-synchronous SoC, where dual-clock FIFOs interface every tile with the NoC. When it comes to large systems, a fully synchronous NoC is no longer a feasible solution. Timing-closure issues, mainly due to increasing clock skew, impose unacceptable constraints that force designers to decrease clock frequency and, ultimately, degrade the maximum sustainable bandwidth of the NoC.

[5] It is possible to layout accelerators and memory tiles so that feedback Acc2Acc messages can share Plane 4 with Mem2Dev messages. This solution saves area but its applicability is constrained by the data-dependency graph.

Synchronous vs. multi-synchronous NoC. One solution I have explored consists in including the routers within the clock domain of the corresponding tile, to obtain a multi-synchronous NoC. In this way, the queues of the routers' local port can be simple FIFOs, whereas the queues for all other directions must be replaced with dual-clock FIFOs. In addition, the clock of the tile must be shipped in all directions alongside data. This requirement has significant drawbacks on the efficiency of the system. First, the count of long wires in the SoC increases by one per each link of the NoC, without however increasing the bandwidth of the interconnect. Even worse, these wires have the highest possible switching activity because they transport clock signals. Unfortunately, a multi-synchronous NoC leads also to a performance degradation due to the dual-clock FIFOs that each packet must traverse.

Typical dual-clock FIFOs are designed such that the synchronization only occurs on the status flags "empty" and "full". Hence, on a multi-synchronous NoC, as long as the average injection rate of every tile is such that queues never get full, nor empty, the available bandwidth will only be limited by the link belonging to the domain with the slowest clock frequency. In general, however, components such the processors tend to communicate using relatively small packets (e.g. a cache line). Therefore, queues are likely to get empty and latency cannot be ignored: every time a packet traverses a router, whose input queue was either empty or full, two to four clock cycles are lost for synchronization, depending on the ratio between the clock frequencies of the two adjacent domains. In this context, the worst case scenario is represented by single-flit packets, for which latency is the only relevant performance metric. Fig. 7.5 compares the expected average latency of single-flit packets for a synchronous NoC and a multi-synchronous NoC. This is done by modeling the traffic as a stochastic process, where the probability of intersecting a conflicting packet follows Poisson's distribution, according to the average injection-rate per tile L.

When $L = 0$, the only packet in-flight is the one observed, therefore there is no contention. Hence, the corresponding lines in Fig. 7.5 show smaller latency than any other curve. The knee visible in all lines for synchronous NoCs is due to the synchronization that always occurs at the local ports of the transmitting router and the receiving router, independently from the number of hops. Beside this initial condition, the curves for the multi-synchronous NoC have a steeper slope, because of the synchronization penalty that can potentially occur at every hop. When the average injection rate per tile L grows, contention exacerbates the penalty of synchronization, as shown

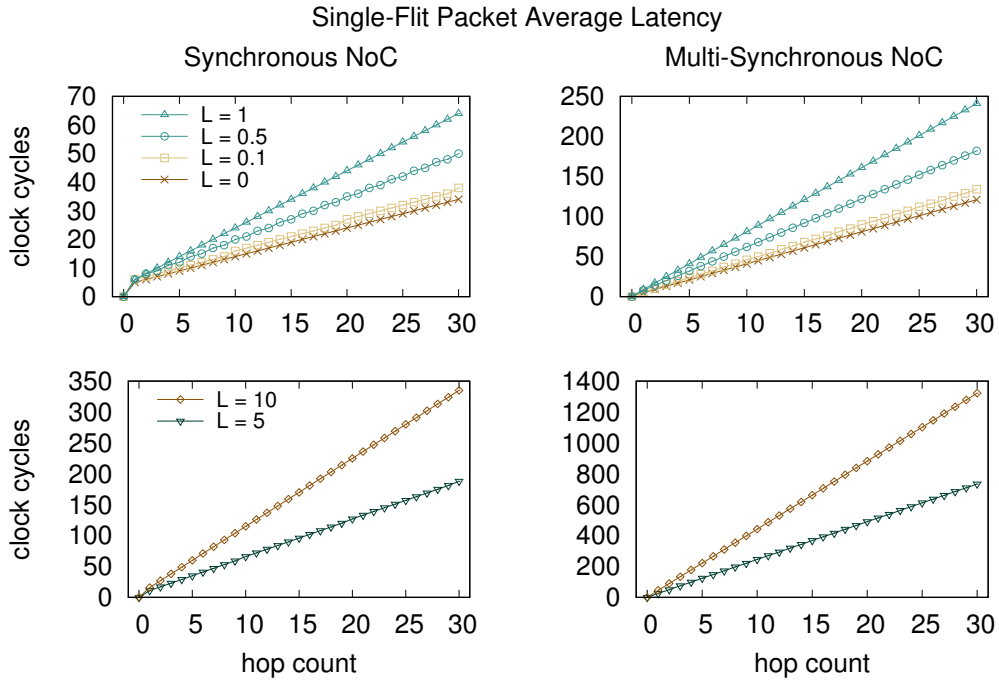


Figure 7.5: Comparison of the average latency for single-flit packets traveling on a fully-synchronous NoC and a multi-synchronous NoC. Traffic is modeled as a Poisson's stochastic process with average injection rate L.

in the bottom-right chart. The conclusion, drawn by analyzing Fig. 7.5, is that for small values of L and a small hop-count the latency of the multi-synchronous NoC is still acceptable. When the system scales, however, not only there are longer paths, but also the probability of congestion increases as more tiles use the network. As the traffic and the hop count grow, the latency curve for multi-synchronous NoC diverges quickly from the corresponding synchronous one. This suggests that even if a multi-synchronous network is easier to layout and route than a fully-synchronous one, it doesn't really scale with respect to its synchronous counterpart, when considering performance metrics at the system level.

Asynchronous NoCs. A valuable alternative would be to implement a fully-asynchronous NoC. This option has been extensively analyzed by many researchers in the literature. For instance, fully asynchronous communication has been proposed to mitigate the issue of clock skew for large NoCs [Bainbridge and Furber, 2002; Bjerregaard and Sparso, 2005; Rostislav *et al.*, 2005;

Kasapaki and Sparso, 2014]. Thanks to the asynchronous interconnect, these designs are optimized to deliver low latency communication for globally-asynchronous-locally-synchronous (GALS) systems and reduce area occupation and power consumption with respect to a synchronous alternative. Latency, area and energy improvements have been shown specifically for a 2D mesh, which is the topology adopted by the ESP network, by combining asynchronous routers with early arbitration to accelerate transactions that span multiple hops [Jiang *et al.*, 2015]. An interconnect based on asynchronous routers has also been proposed to prolong the life time of battery-powered devices by implementing simple three-port routers with 2-phase handshake mechanisms over the router-to-router links [Kunapareddy *et al.*, 2015]. A fully asynchronous NoC is also the backbone for the TrueNorth chip from IBM, which enables large-scale neuromorphic computing with a very low power budget [Akopyan *et al.*, 2015]. Another interesting example is the many-core architecture from STMicroelectronics, named Platform 2012, that implements a GALS system by leveraging an asynchronous NoC that connects clusters, each with sixteen cores.

7.3.1 Hierarchical-NoC Architecture

The main concept beyond the ESP architecture and methodology is to ease design and integration. Therefore, since state-of-the-art CAD tools are sill not mature enough for asynchronous design, the NoC is kept fully synchronous. Nevertheless, in order to achieve the desired goal of scalability, some level of asynchrony is inevitable in the integration of heterogeneous components [Nowick and Singh, 2015]. In fact, the ESP architecture leverages asynchronous design for a NoC-to-NoC bridge that allows designers to create ESP instances with synchronous NoCs of a manageable size, while scaling the number of tiles, components and clock domains by instantiating multiple networks in a hierarchical fashion. The ESP NoC-to-NoC bridge has been designed to be fully asynchronous so that the clock does not travel across the link. In addition, with the support of high-speed on-board traces, or interconnection wires, this bridge is suitable for inter-chip communication. Therefore, each synchronous NoC of the ESP system can be laid out on a separate die, or mapped to a separate FPGA. This divide-and-conquer approach improves yield and simplifies the place and route tasks, by reducing the number of components to place and connect together at the same time. Fig. 7.6 shows a high-level view of three bridges connecting two 3×3-mesh NoCs, which are implemented on a dual-FPGA system. The FPGAs are two `Xilinx Virtex7` [Xilinx, 2016] with high-speed

transceivers and serializer/deserializer (SERDES) IP blocks capable of driving off-chip wires with a data-rate in the Gbps range. These capabilities are generally used to comply with standard communication interfaces, such as Ethernet, or PCI Express. The physical link between the FPGAs consists of high-speed wires attached to FPGA Mezzanine Card (FMC) connectors on a proFPGA system [ProDesign, 2014]. Meshes on every FPGA have 6 planes, each clocked with a 100MHz clock. The clock frequency is the same on both FPGAs to guarantee similar bandwidth of the networks. The clock distribution, however, is independent and no clock is shipped across the bridge. The data-width is set to 32 bits. Therefore, considering head and tail bits, and the fact that each link is bi-directional, there is a total of 432 data-signals to bridge across FPGAs. Given that I/O is a resource as precious as memory, the design of the bridge implements time-division multiplexing to reduce the number of physical wires.

The first step to implement such bridge is to measure latency and throughput over the link. This is done by implementing a proxy design that generates a fixed sequence of data and sends it from one FPGA to the other. Thanks to proFPGA's programmable source clocks [ProDesign, 2014], the data-rate is swept until the receiver is unable to correctly detect the sequence. With two wires, each connecting 140 FPGA pins, the proxy design can transmit data at a maximum rate of 800 Mbps per wire. The measured latency is about 40 ns, corresponding to four clock cycles when the NoC is clocked at the frequency of 100 MHz. Additional latency must be considered to forward data received from the bridge to the synchronous NoC. Note that this latency does have an impact on performance, similar to what shown in Fig. 7.5. However, packets are affected by the extra delay only when they cross the bridge, while any other hop takes just one clock cycle. For this reason, different clustering options of components that interact more frequently should be analyzed during the design-space exploration of the ESP system.

Having assessed that the external wires can sustain a throughput of 800 Mbps, the multiplexing factor of the bridge is set to eight, thus reducing the number of wires per bridge from 432 to 54. From the NoC viewpoint, this means that in one clock cycle eight bits are transmitted for each physical wire. The drawing in the top-right portion of Fig. 7.6 shows the resulting wave-pipelining on the wires: the FPGA SERDES take care of maximizing the difference between the minimum and the maximum delay for each bit to be asserted, which is equivalent to maximize the sampling window. The latter is the period of time in which a clean value can be captured from the incoming

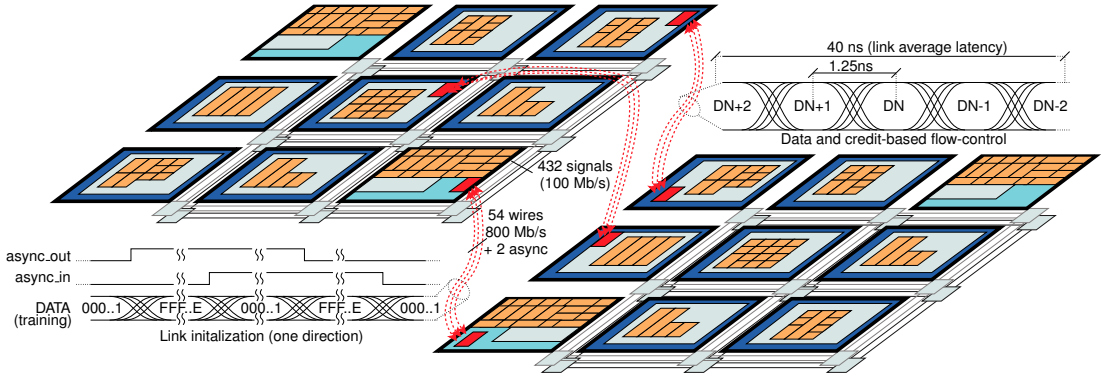


Figure 7.6: Example of hierarchical NoC with inter-FPGA bridges for six 32-bits planes. Each bridge implements asynchronous handshake on control wires and wave pipelining on data wires for up to 5.4GB/s per link.

stream. At 800 Mbps this window is a fraction of 1.25 ns. Considering the measured latency of about 40 ns, at any given time almost 32 bits are in flight on a single wire. In order to make this data transfer possible, a known training sequence is used to initialize the receiving end of the bridge. Additionally, once the sequence has been received correctly, a 4-phase asynchronous handshake triggers the transmitters on both sides of the bridge to switch their source from the training sequence to the system NoC. The 4-phase handshake imposes that both request and acknowledge wires return to zero when the transaction is complete, making the latter more robust, with respect to adopting a transition-based 2-phase handshake. The implementation details are shown in Fig. 7.7, which illustrates an instance of the NoC-to-NoC bridge. For the sake of simplicity, the image represents a bridge instantiated in an empty tile. Hence, the queues of the bridge service coincide with the dual-clock FIFOs of the router's local ports. In general, when other components are located in a tile with the bridge, the local ports are connected to the MUX/DEMUX layer shown in Fig. 7.4, and the queues of the bridge are regular FIFOs in the tile's clock domain.

Dual-Clock FIFO. The implementation of the dual-clock FIFOs is shown on the left of Fig. 7.7. For the NoC-output queue at the top, the write pointer is generated by a "Gray counter", which is incremented on the NoC's clock edge. The output of the counter drives a de-multiplexer that activates the appropriate write-enable signal of the registers. Additionally, the output of the counter is sampled by a simple synchronizer to be compared with the read pointer and determine whether

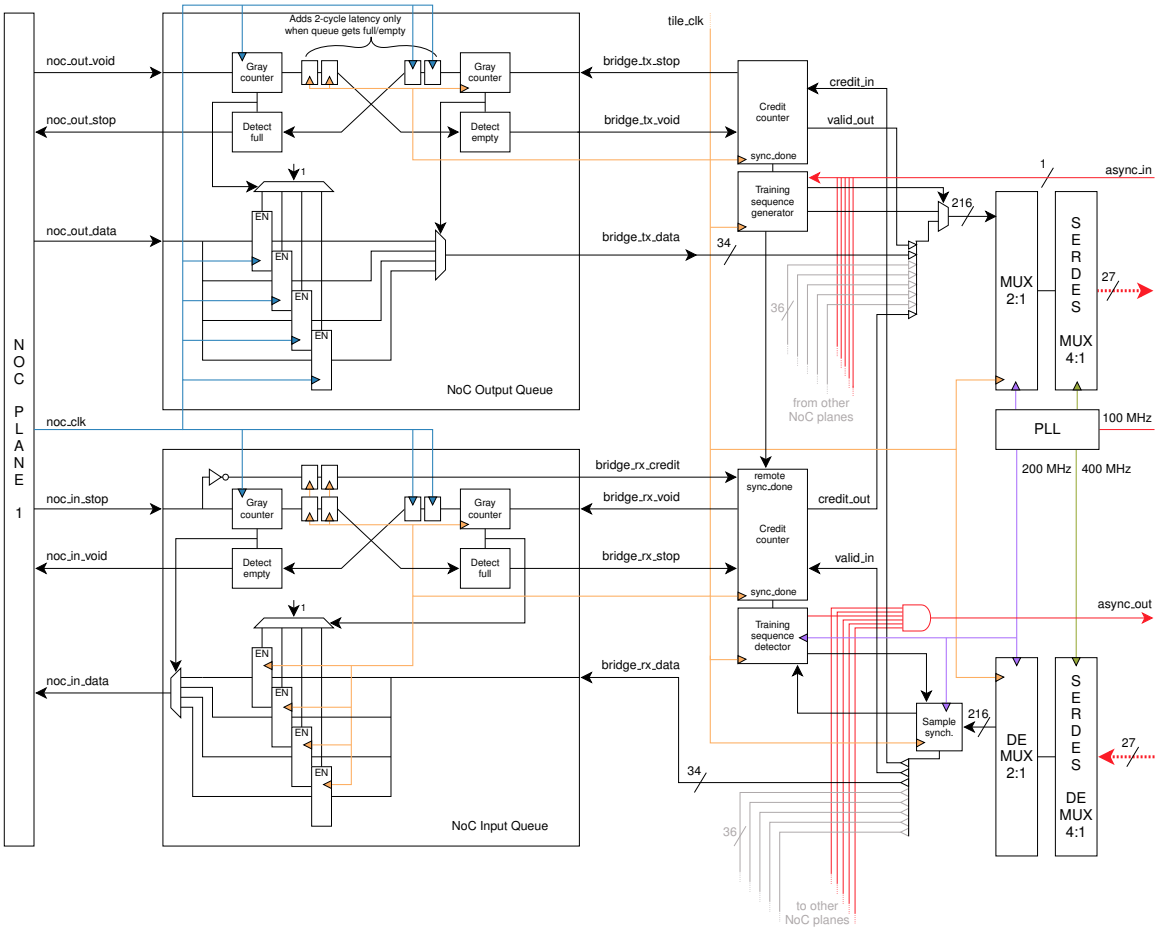


Figure 7.7: Implementation details of the NoC-to-NoC platform service. In this example the bridge is instantiated in an empty tile.

the queue is empty. The synchronizer consists of two registers placed close one another. Placing registers in this way reduces the probability of meta-stability. In addition, "Gray code" guarantees that at every increment, or decrement, the value of the counter changes by one bit only, thus further reducing the likelihood of meta-stability. Similarly, the read pointer is driven by a "Gray counter", which controls the output multiplexer to select the head of the queue. The read pointer is sampled by a synchronizer as well. The synchronized version of the read pointer is compared with the write pointer to detect when the queue is full. The latency to cross the synchronizers is what causes latency degradation on each clock-domain crossing. Throughput loss, instead, is only determined by

the difference in the clock frequencies. There is no loss when both source and destination domains share the same clock frequency, independently on the clock-phase misalignment.

Bridge Transmitter. Past the dual clock FIFOs, Fig. 7.7 shows the implementation of the transmitter on top and the receiver at the bottom. The transmitter logic includes the training-sequence generator and a credit counter. The former is activated at reset, as soon as the clock of the tile is stable. On each 34-bit NoC link the generator produces the sequence of two alternating words as follows: 0X3FFFE, 0X00001, 0X3FFFE, 0X00001, ... Once the receiving end detects the sequence, a transition on ASYNC_IN informs the transmitter that the link is established. At this point the training-sequence generator switches the output multiplexer to receive data from the dual-clock FIFO. In addition, it unlocks the credit counter. The latter is initialized to the size of the receiver queue and generates the control signals to pop a new data item from the queue whenever it becomes available. At every pop the counter is decremented, until it reaches zero and the "stop" signal is asserted. Additional credits are received from the receiving-end of the bridge whenever data are removed from the receiver queue.

The 34-bits output signals of the first plane, shown in Fig. 7.7, are concatenated with a valid bit, a credit-out bit from the local receiver and the output signals of the other planes of the NoC. A total of 216 bits enters a first 2:1 multiplexing stage and then is shipped through the FPGA SERDES with an additional 4:1 multiplexing factor. The clock for the SERDES and the multiplexing stage is generated by a PLL with a 100 MHz reference clock. The multiplexing stage acts as a synchronizer between the tile and the SERDES clock domains, thus allowing the tiles to use DVFS and run at an independent frequency. Note that the multiplexer stage and the SERDES don't need synchronization, instead, because their clocks are derived from the same reference clock and the ratio between their frequencies is an integer number.

Bridge Receiver. The receiver has a training-sequence detector that leverages the faster clock from the SERDES domain to determine when data should be sampled. Once the training-sequence is recognized, the detector completes initialization in three steps: first it freezes the control of the input synchronizer; then it sets the ASYNC_OUT flag; and finally it enables the credit generator. The latter waits for the ASYNC_IN flag to be received by the transmitter and then sends credits to the other end of the bridge whenever the NoC consumes one entry of the input FIFO. After initialization, the

input synchronizer brings data from the SERDES domain to the tile domain before they are stored to the input queue. Similarly to the transmitter, the external interface consists of the SERDES and a de-multiplexing layer that cumulatively apply a de-serialization factor of 1:8.

The implementation described in this section can bridge up to six 32-bits-wide NoC planes with only 56 wires: 54 wires transport data, valid flag and credits, whereas the remaining 2 wires are used by the initialization protocol. With only two proFPGA interconnection cables [ProDesign, 2014] it is possible to instantiate up to five bridges.

With more FPGA modules and interconnect cables, this design makes it possible to scale the size of the ESP system from a few dozen of tiles to as many as the address space can reference. Thanks to asynchronous communication this scaling does not exacerbate the complexity of the physical design nor introduces issues for timing closure. Furthermore, the implementation of the bridge across synchronous NoCs allows designers to deploy instances of ESP on multiple FPGAs, thus removing the problem of having limited resources with respect to an ASIC counterpart.

Part III

Conclusions

Chapter 8

The Impact of ESP and Future Work

My dissertation addresses the need for a simpler scalable design methodology for heterogeneous SoCs. I believe that Embedded-Scalable Platforms is a practical answer to such need. Still, there are several features and ideas that have not been explored or implemented yet, but I foresee them to potentially grow even further the value of ESP. In addition, the public release of Open-ESP, documented in Appendix A, will give other research groups the possibility to bring up their own applications by leveraging the SoC generation flow of ESP, and to contribute with new ideas and architectural enhancements. Indeed, the ESP methodology has already influenced other research projects of students at Columbia University that I mentored during my PhD. Moreover, the ESP methodology has been adopted in an advanced class for Columbia graduate students that aims at growing a new class of system-level engineers.

8.1 ESP Architecture Extensions

Some of the ideas for future work have already spawned side projects in my adviser's group. Visiting students, summer interns, and even new PhD students have been working with ESP and are building their research on top of the ESP infrastructure. Hence, it is worth presenting here a selection of interesting ideas that I hope will become part of Open-ESP in the near future.

8.1.1 Automated Place and Route ASIC Flow

Throughout the ESP project I have been using FPGAs both as emulation tools and as actual system-implementation substrate. The design methodology is however not tied to a specific target technology. Many of the results, in fact, have been collected from an ASIC design flow for an industrial 32 nm CMOS technology. Synthesizing accelerators for an ASIC technology is straightforward: the same SystemC source code can be synthesized for any available technology by simply re-targeting the HLS-tool scheduling script. The PLM generator can also deal with different technologies by providing a description of the available primitive SRAM blocks, their behavioral models for simulation and the corresponding libraries for synthesis, timing analysis and physical design. The SoC-generation flow, instead, requires few additional steps, which consist in creating appropriate wrappers for the technology-dependent components. These are I/O pads, clock buffers, SRAMs used for the processor's caches, and PLLs. Further details on how users can add support for a new technology in Open-ESP are explained in Appendix A.

A PhD student from the EE department tackled the challenge of completing the physical design of one ESP instance with 16 tiles, including 4 LEON3 processors, 9 accelerators, and an I/O bus to interface with a proxy FPGA for accessing memory and other peripherals. In addition, a fifth LEON3 processor, adapted for near-threshold voltage (NTV) operation was integrated on the system. Tiles layout and I/O have been planned to allow each tile to be paired with a stacked integrated voltage regulator to enable the fine-grain DVFS discussed in Chapter 6. Following the flow described in Section 6.2.3, each tile has been synthesized independently from the others and has been characterized for four different VF pairs, ranging from 1 GHz and 1 V to 350 MHz and 0.6 V. Furthermore, the NTV LEON3 was characterized to work at a voltage as low as 0.3 V. Given the complexity of the system, commercial tools for automated place and route were employed for the physical design. Nevertheless, both floorplanning and routing required manual intervention to achieve timing closure and meet all of the design constraints. In particular, each tile could not exceed the area of 1 mm^2. Further, in order to support DVFS, each CPU and ACC tile had to include a PLL, which imposed a significant clearance in the surrounding area to ensure signal integrity. In addition, SRAM blocks enforced special requirements on the power grid. Finally, level-shifters were inserted during synthesis at every voltage-domain crossing. After completing the layout of the tiles, the top-level physical design posed even harder challenges. In particular, the clock distribution

network for the NoC could not easily satisfy the low-skew constraint across the entire interconnect. Moreover, non-standard constraints had to be applied to every tile interface, which correspond to the clock-domains and voltage-domain crossing points. Learning from this experience, the hierarchical interconnect with asynchronous links, presented in Section 7.3, was adopted to avoid creating large synchronous NoCs. As future work, I also envision restructuring the synchronous network itself to ease the task of juxtaposing tiles. Even though expert chip designers may not find any particular obstacle in the physical design of an ESP instance, I wish for many engineers who may have much less experience to still be able to complete the physical design with limited effort.

The physical design of an ESP instance should start from the MEMDBG, MEM and MISC tiles that consist mainly of IP blocks, most of which are black boxes. These tiles are therefore the most irregular ones but also the easiest to layout, because these components, which include I/Os, won't need DVFS in general. In addition, they integrate less memory than accelerators, thus chances are that automated place and route tools will be able to process them with almost no manual intervention. Memories, in fact, impose blockages and imply additional constraints on the clock distribution and the power grid. Hence, when they take most of the available area, the routing algorithm may not be able to succeed in a reasonable amount of time.

CPU and ACC tiles, on the other hand, benefit from a fairly regular design. Each tile consists of a large area dedicated to SRAM banks, a PLL with the DVFS control logic, a datapath, a fix set of components implementing the platform services, and the network interface. The goal is to enhance the SoC generation flow for ESP, such that timing constraints, floorplan of the black-boxes, power plan and routing scripts can be automatically generated for each tile. Provided that all analog components (i.e. pads, PLLs, etc.) are available for the target technology, these scripts should allow inexpert designers to complete the physical design of all the tiles with little or no manual intervention. Furthermore, by grouping each tile with its corresponding router, clock-domain and voltage-domain crossing can be moved from the edge of the tile to internal components, which are the local queues of the routers. The latter are identical for every router, thus it should be possible to automatically instantiate level-shifters and declare power intent and timing constraints for both the tile logic and the router. Finally, by setting appropriate input and output delays on the NoC ports incoming and departing from routers, timing closure after juxtaposing tiles should be guaranteed by construction.

I hope that the automated place and route flow will be soon integrated in ESP, as part of the commitment in hiding as much as possible the low-level implementation details and let designers focus on their IP blocks and on design-space exploration at the system level.

8.1.2 Accelerators for Domain-Specific Applications

Medical image processing. The first concrete application of the ESP methodology has been the design of two accelerators for a microwave-imaging application that aims to detect breast cancer [Pagliari *et al.*, 2015]. These accelerators were designed and optimized with HLS and their structure is based on an early version of the ESP accelerator model. The latter enables the pipelining of the input and output phases with computation, which is critical for this specific application that is both communication and computation intensive. The design methodology allowed one master student to complete the implementation of both accelerators in the span of four months and have them fully integrated on a ZYNQ FPGA-board [Pagliari *et al.*, 2015].

Machine learning. Machine learning (ML) techniques are pervasive across multiple applications which range from computer vision, security, data analytics and advanced embedded applications, including self-driving vehicles and robotics in general. As a result, several groups have put their efforts into designing specialized hardware accelerators for ML applications. The background Chapter 3 lists a few relevant works proposing accelerators for ML, some of which led to ASIC implementations. With ESP, rather than focusing on a single application, it would be desirable to integrate multiple accelerators to speedup relevant ML kernels that appear in different applications. One idea is to leverage ESP's P2P communication to build chains of accelerators that can improve graph traversal or neural network-based computational tasks. Since these applications mostly operate on huge and sparse data structures, the main bottleneck is typically the memory bandwidth. Therefore, it is necessary to explore solutions to improve I/O and accelerate the accesses to sparse data. While it may be difficult to compete against GPUs in terms of performance, there is room for more energy-efficient computation.

As an initial step the challenges of accelerating some ML-related applications were tackled, with the help of summer and visiting students. Some computational kernels taken from the *CortexSuite* [Thomas *et al.*, 2014] and from linear algebra have already been implemented as ESP

accelerators [Pilato *et al.*, 2016]. A few examples of applications from the *CortexSuite* are text classification, image-features extraction and user-rate prediction for movies. Each implemented accelerator was placed in a single ACC tile of ESP. As expected, preliminary results show a performance degradation with respect to high-end processors and GPUs. Furthermore, DMA with sparse data proves to be inefficient, because many times irrelevant data and zeros are streamed into the PLM. On the other hand, results are promising in terms of total energy dissipation and energy per operation that are orders of magnitude smaller, if compared with those of processors and GPUs.

Striking the right balance between specialization for ML and the flexibility of a generic ESP instance is still an open issue. Nevertheless, I believe that a clever control of P2P communication across kernels, combined with specialized caches, capable to exploit temporal locality, is the key to unlock efficient computation for ML applications on ESP.

8.1.3 Embedded RISC-V Processor

The RISC-V open ISA [Asanović and Patterson, 2014] and the open-source ROCKET CHIP GENERATOR [Asanović *et al.*, 2016] have recently proven extremely successful in growing their user base. Interest for open-source hardware has risen from industry and even more from academia [Gupta *et al.*, 2016] and the RISC-V project promises to deliver a competitive processor, implementing an efficient ISA, but with the benefits of open-source licensing. Many research groups investigate problems in the context of SoCs that do not not strictly involve the processor architecture; however, in order to provide a prove of concept for their findings, they must leverage existing processor cores to run software and bring up the system. In the case of ESP, I chose the LEON3 processor [Gaisler, 2004] because of its small footprint. LEON3 is a single-issue processor, consisting of an in-order seven-stages pipeline. Despite being a low-performance core, it is capable of booting the Linux operating system and has a co-processor interface, which I leveraged to implement and connect a high-performance floating-point unit (FPU). The latter is required to run embedded-software versions of most of the computational kernels in the presented case studies and to compare that against dedicated hardware accelerators.

Differently from LEON3, the ROCKET CHIP supports a 64-bits address space, which would unlock the realization of much larger case studies for ESP. In addition, the LEON3 runs the outdated Sparc V8 ISA, whereas RISC-V has been recently defined and is likely to be apt to modern

applications' demands. The ROCKET CHIP allows for the integration of custom IPs in three ways: within the processor pipeline; through a co-processor interface, called ROCC; and as a bus peripheral [Asanović *et al.*, 2016]. All three integration modes result in having accelerators logically tight to the processor. Unlike ESP, in fact, the ROCKET CHIP design is processor-centric, rather than focusing on the SoC. In addition, its bus-based interconnect doesn't provide the natural scalability properties of a network. These premises justify the decision of integrating the ROCKET CHIP into the ESP CPU tile. In order to support the integration of a different core, the adapter layer of the proxies must be re-implemented to comply with the bus protocol of the new processor. Specifically, while the LEON3 is based on the standard AMBA V2.0 from ARM, ROCKET CHIP implements a custom (but open-source) bus protocol, called TILELINK.

Two visiting master students, working under my supervision, were able to disassemble the processor core, with its private caches, from the ROCKET CHIP implementation, and to design the remote-memory and register access proxies' adapters, based on the TILELINK specifications. In this way, they were able to integrate one 64-bits RISC-V processor in the ESP CPU tile, run bare-metal applications, boot Linux, and invoke the execution of accelerators. The integration process, however, is not complete yet: the current implementation requires shared components, such as the interrupt controller, to be placed next to the core and attached to the same bus. As a result, the system cannot scale to multiple processor tiles yet. Also, since the memory hierarchy has not been upgraded to support 64-bits addressing, running applications with larger workloads is still not possible. Preliminary results show a slight reduction in floating-point performance, mainly due to the fact the ROCKET CHIP with its own FPU on a Virtex7 FPGA cannot run faster than 62.5 MHz. Conversely, the LEON3 and the specialized FPU that I designed can run at up to 100 MHz on the same FPGA.

The final target of this derived project is to allow users to generate a multi-core RISC-V version of ESP with coherent ROCKET-CHIP cores on a 64-bit NoC with a 64-bit address space. This will enable scaling of the applications' footprint, which is now limited by the addressable space of LEON3. In addition, RISC-V adopters would be able to choose between a traditional processor-centric system with the ROCKET-CHIP GENERATOR (for small SoCs with more tightly-coupled accelerators) and the ESP scalable architecture (for larger SoCs with loosely-coupled accelerators).

8.2 Teaching

Since the beginning of the project, Embedded Scalable Platforms has fueled the need for a better understanding of high-level synthesis and for a solid design methodology for accelerators. This led the ESP methodology to influence the contents and the practical exercises for the "System-on-Chip Platform" class, offered at Columbia every Fall semester [Carloni, 2016]. The class develops across two interleaved main tracks:

1. The first track illustrates how to think about a digital system in a compositional way. Students learn several models of computation and how to compose IP blocks, given a set of Pareto-optimal implementations, to compute system-level metrics.

2. The second track dives into HLS and high-level hardware description languages. Students focus on SystemC, its simulation environment and how to use it to describe concurrent hardware processes. Once they master the SystemC, they start understanding how HLS works and perform hands-on exercises with commercial tools. As they gain experience with these tools, they leverage advanced knob-settings and face limitations and issues derived from both real dependencies in the control-data-flow graph of the target application and false dependencies simply due to the code style, or to the compiler internal representation. As they progress in the class, they understand how to tune their code and HLS to obtain a larger Pareto set, with better performance and cost spans.

At the end of the semester, the students get a sense of what it means to design a full system, integrate components and write software to control them. The class teaches them how to use virtual platforms, which are simulators that model an entire SoC and allow users to boot an operating system, to develop software, and to estimate power and performance for a target SoC that does not exist yet.

8.2.1 Competitive and Collaborative Design

Once students are able to design accelerators and to write a device driver and an application to invoke them using a virtual platform, they engage in a contest, which requires both competition and collaboration among them. Students are grouped into teams of two people. Each team is assigned

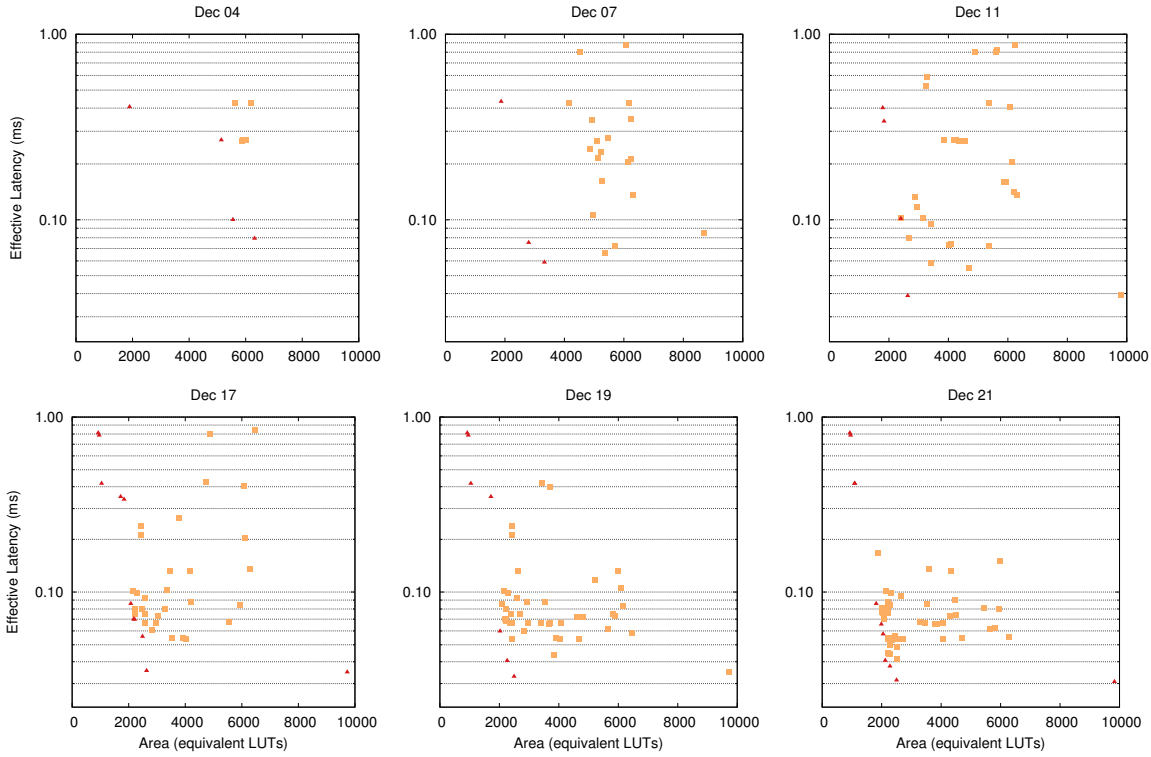


Figure 8.1: Evolution of single-component Pareto set during a team context assigned in class.

an accelerator to design in a fixed amount of time. The single-component design represents the competitive portion of the assignment among a subset of teams. Teams are allowed to submit more than one design during the contest; in fact, they get credits for multiple submissions that improve their previous implementations and improve the Pareto set for their assigned component. Every day all submissions are tested, evaluated and plotted on an area vs. latency chart. Each team is assigned credits based on the distance of their design from the current component Pareto set. Fig. 8.1 shows the evolution of the design-space exploration for one component during the period of time when the project was assigned for the Fall-2015 edition of the course. On December 2^{nd} an initial seed, which was a working un-optimized design, was given to students. In the following twenty days, each team submitted several implementations. The refinement process they went through over time pushed the Pareto set closer to the axis of the chart. In addition, most of the non-optimal designs progressively moved closer and closer to the Pareto set.

The collaborative side of the assignment consists in the following: for every submission, the

teams must choose the implementations of the other components to be combined in the SoC, such that the combination of their design with those selected from other teams leads to a Pareto-optimal point in the application's design-space. By leveraging a compositional methodology, similar to what presented in Section 4.1, all submitted combinations are evaluated and plotted. In this way, the students get continuous feedback on both the quality of their single component and the quality of the combinations they proposed. Again, the teams are assigned additional credits based on the distance from the application-level Pareto curve.

Some students from the "System-on-Chip Platforms" course follow up by taking a project-based course during the Spring semester. In this course, they get access to FPGA boards and attack interesting research projects related to SoC and accelerator design. Some of these projects leverage the ESP infrastructure, which students can exploit to test on a real system what they had previously experimented on the virtual platform. For instance, those students who choose to design a particular hardware accelerator are encouraged to show an FPGA demo at the end of the class, which brings together all they have learned in terms of HLS-driven optimization and system integration, including the programming of software applications and hardware-software integration with device drivers. Indeed, thanks to the ESP methodology, architecture and software stack, in the span of two semesters some students, with no previous expertise in system integration, have been able to combine their IP blocks and build heterogeneous systems on FPGA for domain-specific applications. Each SoC instance features dozens of accelerators, multiple memory controllers and processor cores.

8.3 Conclusions

Throughout the dissertation I presented *Embedded-Scalable Platforms*, which is a system-level-design methodology for heterogeneous systems-on-chip supported by a flexible and scalable architecture and an automated FPGA implementation flow. This combination hides the complexity of heterogeneous integration and lets designers focus on the development of application-specific intellectual property, or accelerators.

The methodology guides the designers through multiple steps of design-space exploration which enables the optimization of the single components, of the target application, and of the entire system.

The architecture exposes to each IP block the platform services that are used to access and

manage all shared resources, including external memory, fine-grained dynamic-voltage-frequency-scaling, and communication over a hierarchical multi-plane network-on-chip.

A companion set of software libraries defines a convenient API that allows designers to port legacy applications across different instances of ESP and seamlessly configure accelerators, manage the external memory allocation policy, and tune power management at a fine spatial and temporal granularity. A multi-threaded library for accelerators enables the quick creation of multi-accelerator workloads, which are useful to tune performance and energy-efficiency of the entire SoC.

In summary, my work addresses the challenges of designing and optimizing scalable heterogeneous systems, while coping with the ever growing complexity of the integration process.

By continuing to influence new students in class and releasing the Open-ESP infrastructure to the research community, I believe that my work will represent a concrete step towards creating a new generation of system-level designers, who will be able to build heterogeneous systems, scaling to a complexity that is unfeasible with current state-of-art design methodologies. The adoption of ESP at Columbia University has already enabled non expert designers to build embedded systems with dozens of accelerators, each operating on a different clock domain, that execute complex computational kernels and process hundreds of megabytes of data.

Part IV

Appendices

Appendix A

Open-ESP

The ESP infrastructure enables the design and design-space exploration of complex heterogeneous systems. Furthermore, during my PhD, the FPGA flow from ESP has served as the starting point and experimental setup for several other research projects and application case studies. Open-ESP aims to extend these benefits to the research community. Through the open-source release of ESP I hope that other research groups will be able to exploit it to quickly bring up SoCs and test their own innovations without the need to know all of the details about system integration. This appendix is a quick tutorial that walks the users of Open-ESP through the main steps required to build heterogeneous systems with custom accelerator IPs.

A.0.1 Dependencies

Open-ESP is released in the form of a GIT repository. The source files include the original ESP code together with other third-party open-source code. Specifically, some of the RTL components, including the LEON3 embedded processor and several I/O peripherals, are part of the GRLIB open-source RTL library [Gaisler, 2004]. The release version integrated in Open-ESP is `grlib-gpl-1.5.0-b4164`. In addition, to support the Linux operating system, Open-ESP includes a branch of the Linux mainstream in the form of a GIT submodule. This branch includes the configuration files for the LEON3 processor and implements an ESP-specific system call to handle DVFS. The entire code of Open-ESP and its dependencies can be fetched by cloning a GIT repository with a recursive clone operation. The recursive option ensures that GIT submodules are retrieved together with the parent repository.

Cloning the Open-ESP repository

```
$>   git clone --recursive git@dev.sld.cs.columbia.edu:esp.git
```

A.0.2 Tools Version Requirements

The Open-ESP tool chain relies on several commercial CAD tools and open-source software to complete each step of the the design flow. The following list specifies which version of each tool has been tested for Open-ESP. Note that these are not strict requirements, but changing the version of some tool or library might require to update the scripts, or source files, in Open-ESP.

CentOS v. 7 operating system.

GCC v. 4.8.5 compiler.

sparc-elf BCC v. 4.4.2 cross-compiler.

sparc-leon3-linux GCC v. 4.7.2 cross-compiler.

sparc-leon3-linux GCC v. 4.7.2 cross-compiler.

Python 3.4.5 interpreter.

Qt v. 4.8.5 graphic libraries.

Qmake v. 2.01a compiler.

Cadence Stratus HLS v. 16.21-s100 for high-level synthesis.

Cadence Incisive v. 15.10-s010 for SystemC simulation and SystemC and RTL co-simulation.

Mentor Graphics Modelsim SE-64 v. 10.5c for RTL simulation.

Xilinx Vivado v. 2016.3 for FPGA synthesis and implementation.

Synopsys Design Compiler v. L-2016.03 for logic synthesis.

Pro Design proFPGA v. 2016B for board-specific IP and configuration tools.

Cobham GRMON v. v2.0.68 for debug with the LEON3 processor.

A.0.3 Repository Structure

The directory tree in Fig. A.1 shows the organization of the Open-ESP repository, where a generic working folder is highlighted in pink. Open-ESP provides two example instances that target two different FPGA boards, but the user can add as many design instances as needed by creating and

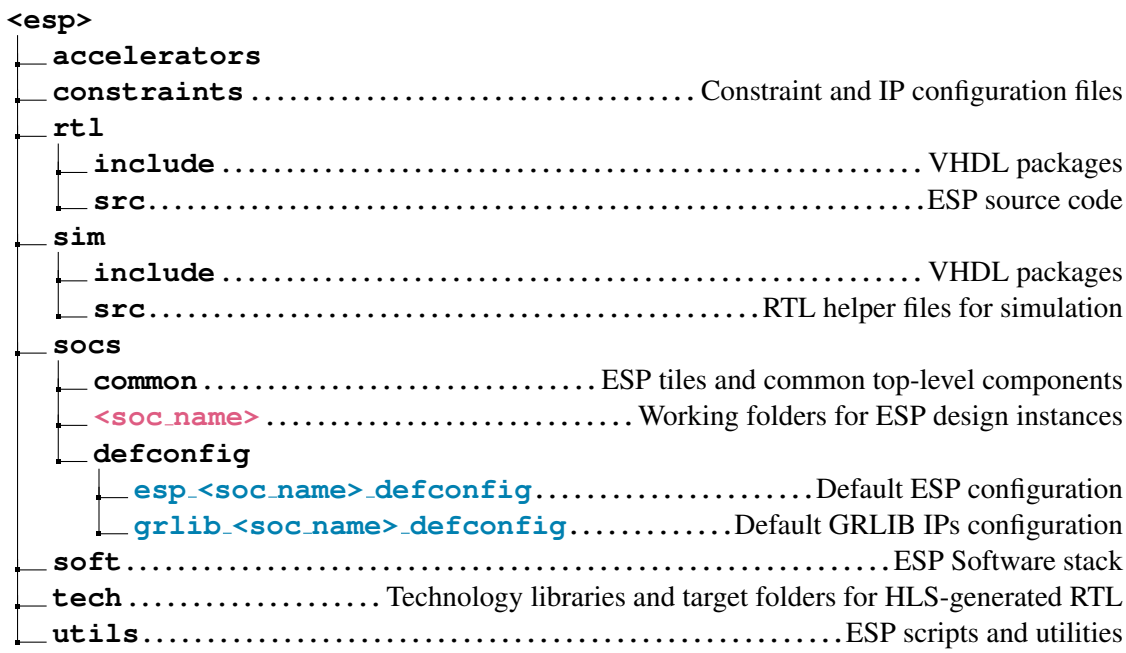


Figure A.1: Directories organization for the Open-ESP.

configuring new working folders. For each SoC folder the user should save the default configuration file for ESP and a default configuration file for the IPs instantiated from the GRLIB open-source RTL library [Gaisler, 2004]. Both files are highlighted in light blue in the tree.

This appendix is structured as follows:

- Section A.1 illustrates how to add a new accelerator to ESP.
- Section A.2 shows how to generate an instance of ESP by using the configuration GUI and running the appropriate Makefile targets. This tutorial leverages two example designs targeting the "Xilinx VC707" development kit and the ProDesign proFPGA system.
- Section A.3 describes how to support a different FPGA board or a given ASIC technology. This requires access to a physical design kit from a third party vendor.
- Section A.4 explains how to customize, build and run the ESP software when using the LEON3 embedded processor.

A.1 Adding New Accelerators

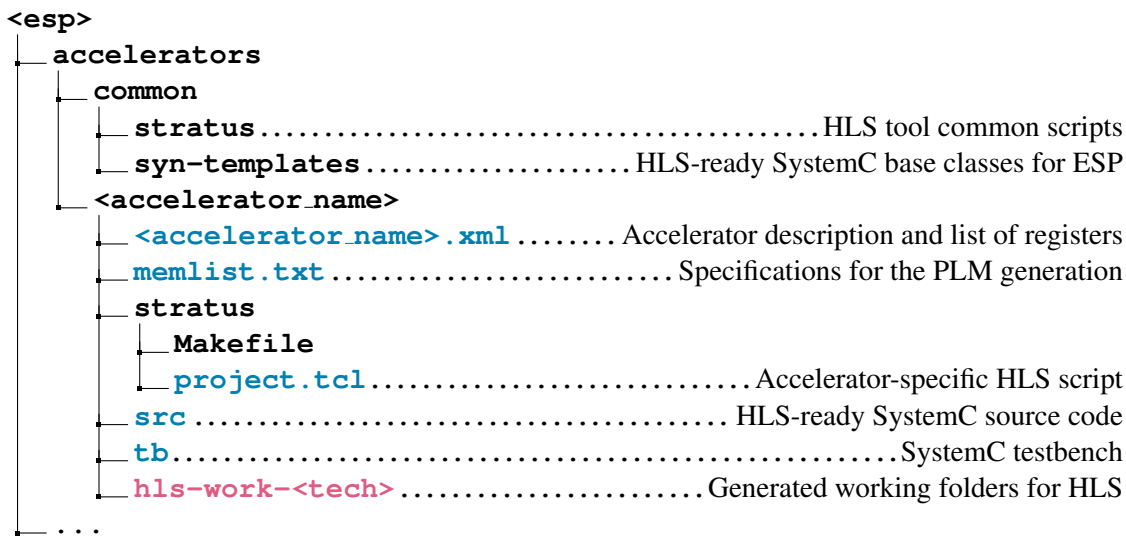


Figure A.2: Directories organization for an accelerator design in Open-ESP.

The directory tree in Fig. A.2 shows the organization of the ESP accelerators folder. This contains a common directory, which includes scripts for the HLS tool and the ESP SystemC classes used to create DMA-capable loosely-coupled accelerators in ESP. In addition, there is a folder for each accelerator that stores the accelerator-specific SystemC code, the configuration parameters and the PLM description. Folders and files that should be edited by the user are highlighted in light blue. In order to begin the design of a new ESP accelerator it is sufficient to run a script which takes as argument the name of the target accelerator.

Path: <esp>

```
$>  export ESP_ROOT=${PWD}
$>  ./utils/scripts/init_accelerator.sh <accelerator_name>
```

The script generates a folder at the path `<esp>/accelerators/<accelerator_name>` and creates template files for the new design, including file stubs for the SystemC implementation of the accelerator and its testbench. The user should complete these files by following the comments in the code. For instance, by passing the argument `fft` to the script, the set of files in Fig. A.3 will be generated.

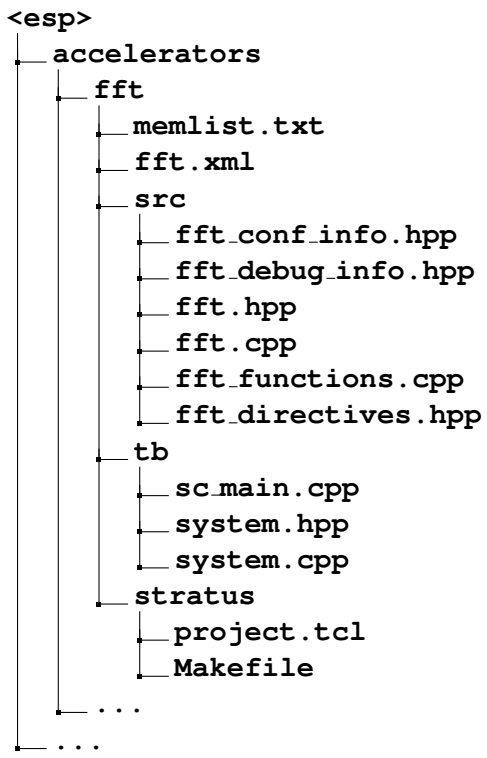


Figure A.3: Directories organization for a new accelerator named “fft”.

fft.xml defines the accelerator by specifying its name, a quick description, the number of entries required in TLB for scatter-gather DMA, and a “unique device ID” that is used to identify the type of accelerator from software. In addition, this file includes a list of parameters corresponding to both configuration and read-only registers. Each parameter has a name, a quick description, a size in bits, and an optional value. When the optional value is specified, the corresponding parameter is implemented as a read-only register that always return such value on any read access.

src/fft_conf_info.hpp defines the data structure used for configuration, which corresponds to the input registers of the accelerator.

src/fft_debug_info.hpp defines an optional data structure used for debug.

src/fft_directives.hpp defines the tool-dependent HLS directives for one or more set of knob settings. For example, if the user creates two HLS configurations, named `BASIC` and `FAST`, this file should include two sets of definitions for all HLS directives as follows.

File: <esp>/accelerators/fft/src/fft_directives.hpp

```
#if defined(HLS_DIRECTIVES_BASIC)
#define HLS_COMPUTE1_LOOP \
  HLS_UNROLL_LOOP(OFF)
#define HLS_COMPUTE1_CONST \
  HLS_CONSTRAIN_LATENCY(2, 3, ''constraint-compute 1'')
// ...
#elif defined(HLS_DIRECTIVES_FAST)
#define HLS_COMPUTE1_LOOP \
  HLS_UNROLL_LOOP(ON)
#define HLS_COMPUTE1_CONST \
  HLS_CONSTRAIN_LATENCY(0, 1, ''constraint-compute 1'')
// ...
#else
#define HLS_COMPUTE1_LOOP
#define HLS_COMPUTE1_CONST
// ...
#endif
```

In this example, the HLS configuration "BASIC" is implemented by not unrolling the loop "COMPUTE1" and by constraining the latency of its body to take at most three cycles. The HLS configuration "FAST", instead, enables loop unrolling for the loop "COMPUTE1" and requires its body to have a latency of one cycle only.

`src/fft.hpp` specifies the accelerator module that inherits from the ESP base classes. The user should add the declaration of any custom method, as well as the instances of memory elements chosen from the memory list specification of the PLM, given in the file "memlist.txt".

`src/fft.cpp` defines the implementation of the accelerator. The user is responsible for completing the configuration of the DMA transactions for both input and output phases. In addition, the user must implement the computation phase. Directives for the HLS tool can be specified within the code in the form of pre-processor macros, while it is recommended to keep their definition in a separate file named `fft_directives.hpp`.

`src/fft_functions.hpp` specifies optional functions for the accelerator. This file is only used for coding style purposes. In particular, it is recommended to keep the definition of the SystemC processes separate from the implementation of the accelerated computational kernels.

`tb/sc_main.cpp` implements the SystemC `main` function; this file requires no edit.

`tb/system.hpp` defines the testbench module, which inherits from the ESP base classes, and binds the accelerator ports to the model of the DMA controller for simulation. The user is responsible to declare application-specific variables and optional methods.

`tb/system.cpp` models the ESP hardware socket for simulation purposes only. The user is responsible for completing the configuration phase and implementing the following functions: `load_memory` that initializes data for processing; `dump_memory` that retrieves the accelerator's results from memory; and `validate` that checks the results for correctness.

`memlist.txt` specifies the list of memory elements in the PLM and their characteristics. Each line that the user adds in this file causes the generation of a banked memory subsystem that can be used to implement a data structure from the accelerator source code. The box below shows an example of memory specification.

File: <esp>/accelerators/fft/memlist.txt

```
# <accelerator_name>_<memory_name> <#_words> <#_bits> <access>
fft_plm_block_multi_op 1024 32 1w:1r 4w:2r 0w:2ru
```

The PLM element `fft_plm_block_multi_op` implements a 1024×32-bits memory with four write interfaces (p_0 to p_3) and two read interfaces (p_4 and p_5). The generated addressing logic enables to access this memory in three different ways:

1. **1w:1r.** one parallel write operation and one parallel read operation to any address in the range [0:1023].
2. **4w:2r.** four parallel write operations and two parallel read operations distributed such that each write interface p_j and each read interface p_i can only access memory locations at the addresses that satisfy the following equations:

$$A_{wr} \mod 4 = j, \forall j \in [0, 3] \tag{A.1}$$

$$A_{rd} \mod 2 = i - 4, \forall i \in [4, 5] \tag{A.2}$$

In this case, interface p_0 writes to addresses 0, 4, 8, etc.; interface p_1 writes to addresses 1, 5, 9, etc.; interface p_2 writes to addresses 2, 6, 10, etc.; interface p_3 writes to addresses 3, 7, 11, etc.; interface p_4 reads from addresses 0, 2, 4, etc.; and interface p_5 reads from addresses 1, 3, 5, etc..

3. **0w:2ru.** two parallel read operations to any address in the range [0:1023] from both interfaces. The option **u** next to **r** or **w** specifies that the access pattern is unknown and no assumptions can be made on the relation between the interface number and the address being accessed. In most cases this option incurs data and storage duplication.

Note that combining these three access patterns implies that write interfaces p_0 and p_1 and read interfaces p_4 and p_5 are connected to all of the PLM banks. Write interfaces p_2 and p_3, instead, are only connected to some of them, according to equation A.1. In some cases, the combination of multiple parallel operations may generate a full crossbar across PLM interfaces and SRAM ports; hence the user should only specify required operations to avoid generating unused addressing logic.

`stratus/project.tcl` is a project file for Stratus HLS. This file generates one HLS configuration named BASIC and a default configuration for simulation. The user should customize the project description by adding optional source and include files, together with accelerator-specific simulation arguments. Furthermore, this script should define all relevant HLS configurations for the design-space exploration.

A.1.1 Installing New Accelerators

TARGET	DESCRIPTION
print-available-accelerators	print a list of accelerators available for HLS.
<**accelerator**>**-hls**	schedule all HLS configurations available for <accelerator> and generate corresponding RTL files.
<**accelerator**>**-sim**	run all simulations defined for <accelerator> using the SystemC testbench; HLS may start if the RTL files are out-of-date with respect to the SystemC source code.
<**accelerator**>**-plot**	run all simulations defined for <accelerator> and plot results on a latency vs area chart; HLS may start if the RTL is out-of-date.
<**accelerator**>**-clean**	clean HLS working directory, but keep the generated RTL for <accelerator>; HLS cache is not deleted.
<**accelerator**>**-distclean**	clean HLS working directory and remove the generated RTL for <accelerator>; HLS cache is not deleted.
accelerators	make <accelerator>-hls for all available accelerators.
accelerators-clean	make <accelerator>-clean for all available accelerators.
accelerators-distclean	make <accelerator>-distclean for all available accelerators.
sldgen	generate RTL wrappers for the scheduled accelerators; this target is always called as a dependency before ESP simulation and synthesis.

Table A.1: Makefile targets to generate and install accelerators in Open-ESP.

Completing the file stubs with the accelerator-specific code is not sufficient to have the new accelerator available in ESP. This requires to run the PLM generator tool, the HLS tool and the ESP scripts for IP encapsulation.

To do so, the user should select an example SoC instance folder, or create a new one, and follow the accelerator flow as described in the quick reference guide embedded in the Makefile. The latter can be printed to screen by building the `help` target.

Path: <esp>/socs/<design_name>

```
$>   make help
```

Table A.1 lists the Makefile targets that can be used to run HLS for one or all available accelerators, simulate them and plot the resulting design points on an area vs. performance chart.

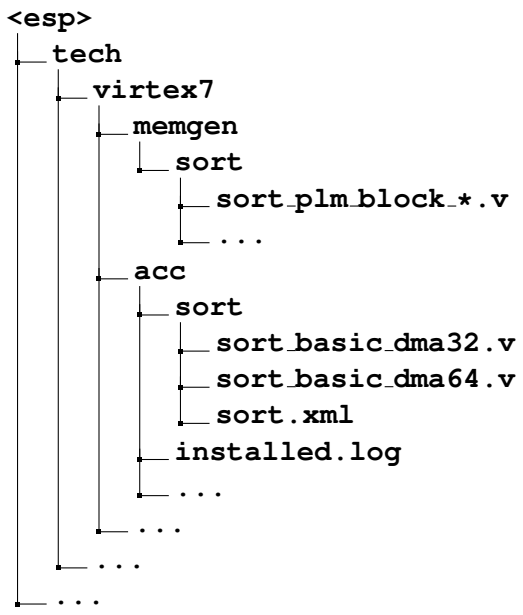


Figure A.4: Directories and files generated by building the HLS target in Open-ESP for SORT.

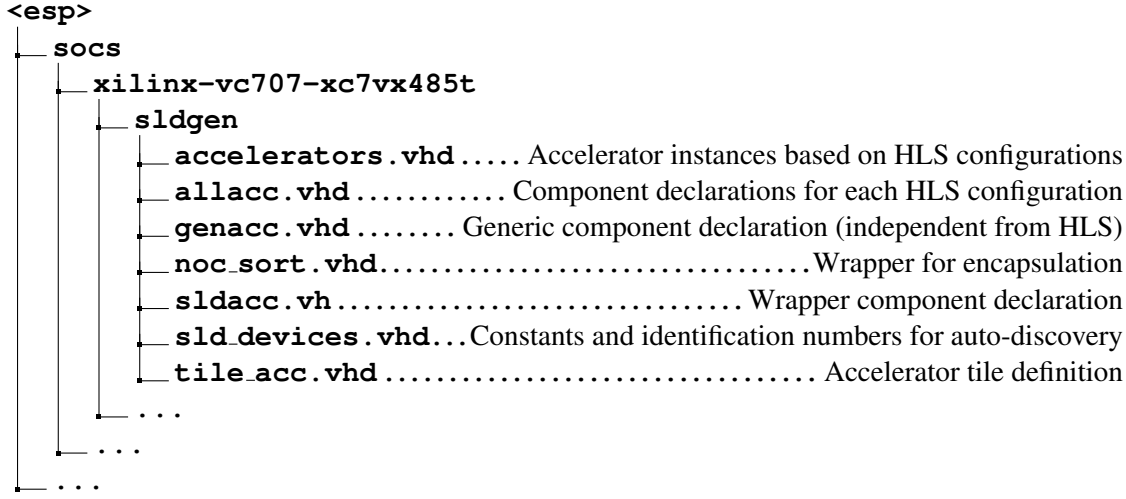


Figure A.5: Directories and files generated when running the Open-ESP accelerator flow for SORT.

It is sufficient to run the `accelerators` target to synthesize and install all available accelerators. This synthesize the RTL for all design points of all available accelerators. The memory generator is invoked before HLS and prints on screen the relevant information about the generated PLM. At this stage, users can run the `plot` targets to visualize the Pareto set of each accelerator and choose which components should be integrated in the instance of Open-ESP.

Running any of the targets for the design of an accelerator causes the generation of a corresponding HLS working directory, as highlighted in the directory tree in Fig. A.2. For an extensive design-space exploration the user may decide to enter such directory and leverage the GUI of the HLS tool. In addition to the GUI, the Makefile for HLS allows the user to run each HLS and simulation configuration standalone. As an example, users can run the accelerator flow for the given implementation of SORT within one of the sample SoC folders.

Path: <esp>/socs/xilinx-vc707-xc7vx485t

```
$>   make sort-hls
$>   make sldgen
```

The first command runs HLS and generates the directories and files listed in Fig. A.4 within the Open-ESP technology folder. The list of installed accelerators "installed.log" is updated by adding the entry "sort". The second command, instead, generates the files listed in Fig. A.5 and prints some relevant information about the accelerators that is found within the technology folder. These files are used for integrating the accelerator into the Open-ESP system.

A.2 Creating an Instance of ESP

TARGET	DESCRIPTION
esp-defconfig	generate default ESP configuration files.
esp-config	generate ESP configuration files based on the content of the design file .esp_config.
esp-xconfig	open the ESP configuration GUI.
sim	compile all RTL source files and simulate the current ESP instance (requires Modelsim 10.5c or higher).
sim-gui	compile all RTL source files and simulate the current ESP instance with graphic user interface (requires Modelsim 10.5c or higher).
vivado-syn	launch Xilinx Vivado and generate a bitstream for the current instance of ESP.
vivado-prog-fpga	download the current bitstream to the target FPGA through the Vivado hardware server. Users should set the environment variables FPGA_HOST and XIL_HW_SERVER_PORT to point to the workstation where the FPGA is connected through the Xilinx or Digilent JTAG cables. This workstation should also run an instance of the Vivado hardware server.
profpga-prog-fpga	download the current bitstream to the proFPGA system. Users should set the appropriate backend in profpga.cfg according to the
accelerators-clean	make <accelerator>-clean for all available accelerators.
accelerators-distclean	make <accelerator>-distclean for all available accelerators.
sldgen	generate RTL wrappers for the scheduled accelerators; this target is always called as a dependency before ESP simulation and synthesis.
grlib-defconfig	generate default configuration files for the GRLIB IPs.
grlib-config	generate configuration files for the GRLIB IPs based on the content of the design file .grlib_config.
grlib-xconfig	open the GRLIB configuration GUI.
leon3-soft	compile the bare-metal program for LEON3, assuming that the `main` function is available in the design file "systest.c". The resulting executable is used for both RTL simulation and FPGA execution. Compilation requires the LEON3 tool-chain and connecting to the core on FPGA requires GRMON from http://www.gaisler.com
linux.dsu	compile Linux and packages the root files system to boot LEON3 on FPGA. Compilation requires the LEON3 tool-chain and connecting to the core on FPGA requires GRMON from http://www.gaisler.com.

Table A.2: Makefile targets to generate and configure an instance of Open-ESP.

When all desired accelerators are installed into the technology folder, a customized instance of Open-ESP can be generated by following the SoC flow shown in the quick reference guide embedded in the Makefile, for which relevant targets are listed in Table A.2.

A.2.1 Preliminary Configurations

Before configuring ESP, the user should initialize a few environment variables and network addresses in order to be able to properly generate and run the system on FPGA. Note that most variables defined in the local Makefile of each sample SoC can be reused across custom designs. However, depending on the particular network configuration of the user, some variables must be updated.

- the environment variable FPGA_HOST defines the network address (name or IP) of the workstation where the "hw_server" daemon from Xilinx Vivado is running. The target FPGA should be connected to this workstation via JTAG or FTDI cables.
- the environment variable XIL_HW_SERVER_PORT defines the port used by the "hw_server" for the incoming connections.
- when using the LEON3 processor [Gaisler, 2004], the default configuration for GRLIB enables the Ethernet debugging interface. This allows users to access the processor debug unit and to quickly load programs into DRAM. This interface, however, does not support DHCP; hence, the IP must be assigned at design time, according to the available address range on the local network. In order to change this address, users must run `make grlib-xconfig`, click the tab labeled "Debug Link" and set both the MSB and LSB bits of the IP address. Optionally, the MAC address can be changed to avoid conflicts on the network. For instance, if the designated address for the debug link is `192.168.1.10`, the MSB bits should be set to C0A8, while the LSB bits should be set to 010A.
- when using the proFPGA system the applicable backend interface must be set in the proFPGA configuration files "profpga.cfg" and "mmi64.cfg". For instance, if connecting through Ethernet at the address `192.168.1.11`, the configuration files should include the following lines.

File: <esp>/socs/<design_name>/[

```
backends :
{
  tcp :
  {
    ipaddr = "192.168.1.11";
    port = 0xD11D;
  };
};
```

A.2.2 Simulating and Synthesizing ESP Sample SoCs

After completing the preliminary configuration steps, the user can test the Open-ESP generation flow for one of the sample SoCs. The default configuration for the Xilinx VC707 Development Kit consists of one memory and debug tile, one miscellaneous tile, one processor tile and one empty tile. The proFPGA example, instead, replaces the empty tile with a second memory tile assuming the default proFPGA setup illustrated in Fig. A.6: the top of the proFPGA motherboard hosts one virtex7 FPGA module and two DDR3 daughter cards, while the bottom of the motherboard hosts one multi-interface daughter card with a stacked DVI daughter card and one gigabit-Ethernet daughter card. Daughter cards and FPGA modules not available in the user's proFPGA system should be removed from both "profpga.cfg" and "mmi64.cfg" configuration files. Note that one

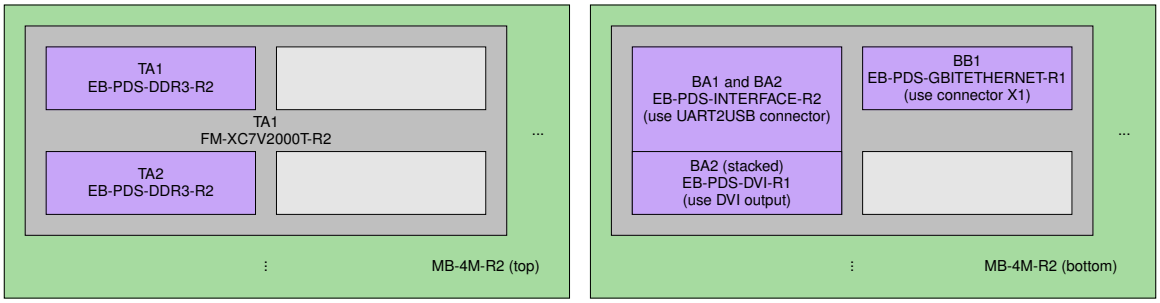


Figure A.6: proFPGA default setup for the sample SoC instance in Open-ESP.

DDR3 daughter card is required for the minimal configuration. If only one DDR3 daughter card is installed in the proFPGA system, users should run `make esp-xconfig` and remove the tile "mem_lite" from the SoC configuration. If the Ethernet daughter card is not installed, instead, users should run `make grlib-xconfig`, enter the "Debug Link" tab, disable the Ethernet link and enable the JTAG link [1]. Finally, if the multi-interface or the DVI daughter cards are not installed, users won't be able to connect a console through UART or leverage the integrated frame buffer, respectively. In this minimal configuration scenario, access to the system is still possible through the GRLIB debug interface as explained in Section A.4.

The RTL simulation can be prepared and launched with `make sim` or `make sim-gui`. Simulation targets build both the RTL source files and the program source files. The default program is coded in "systest.c" and is a simple "Hello world" that will run on the LEON3 processor tile. The RTL default testbench is instrumented to trigger an assertion when the LEON3 processor completes the execution of the program. If the RTL simulation terminates correctly, FPGA synthesis can be run with `make vivado-syn`. This step may require a few hours depending on the complexity of the ESP configuration and the performance of the host workstation. Upon completion of the synthesis and place-and-route steps the "top.bit" file can be downloaded to FPGA with `make vivado-prog-fpga` for any Xilinx FPGA board or `make profpga-prog-fpga` for the proFPGA system. The instructions for connecting to the debug link and running both bare-metal and Linux applications apply to both Xilinx and proFPGA boards and are included in Section A.4. If using the proFPGA system, it is recommended to shut down the FPGA modules when not needed with `make profpga-close-fpga`.

A.2.3 ESP SoC Generator

The default sample SoCs do not place any accelerator tile in the system. These designs, in fact, should only serve as a template to create customized instances of Open-ESP that integrate users' accelerators. An Open-ESP instance can be configured by writing the ".esp_config" file based on the examples available in the "defconfig" folder, or by using the SoC generator graphic user interface. The latter is invoked with `make esp-xconfig`, which opens the application shown in Fig. A.7. The panel labeled "General SoC configuration" lists some information about the current design.

[1] The Xilinx Integrated Logic Analyzer cannot be used with this configuration.

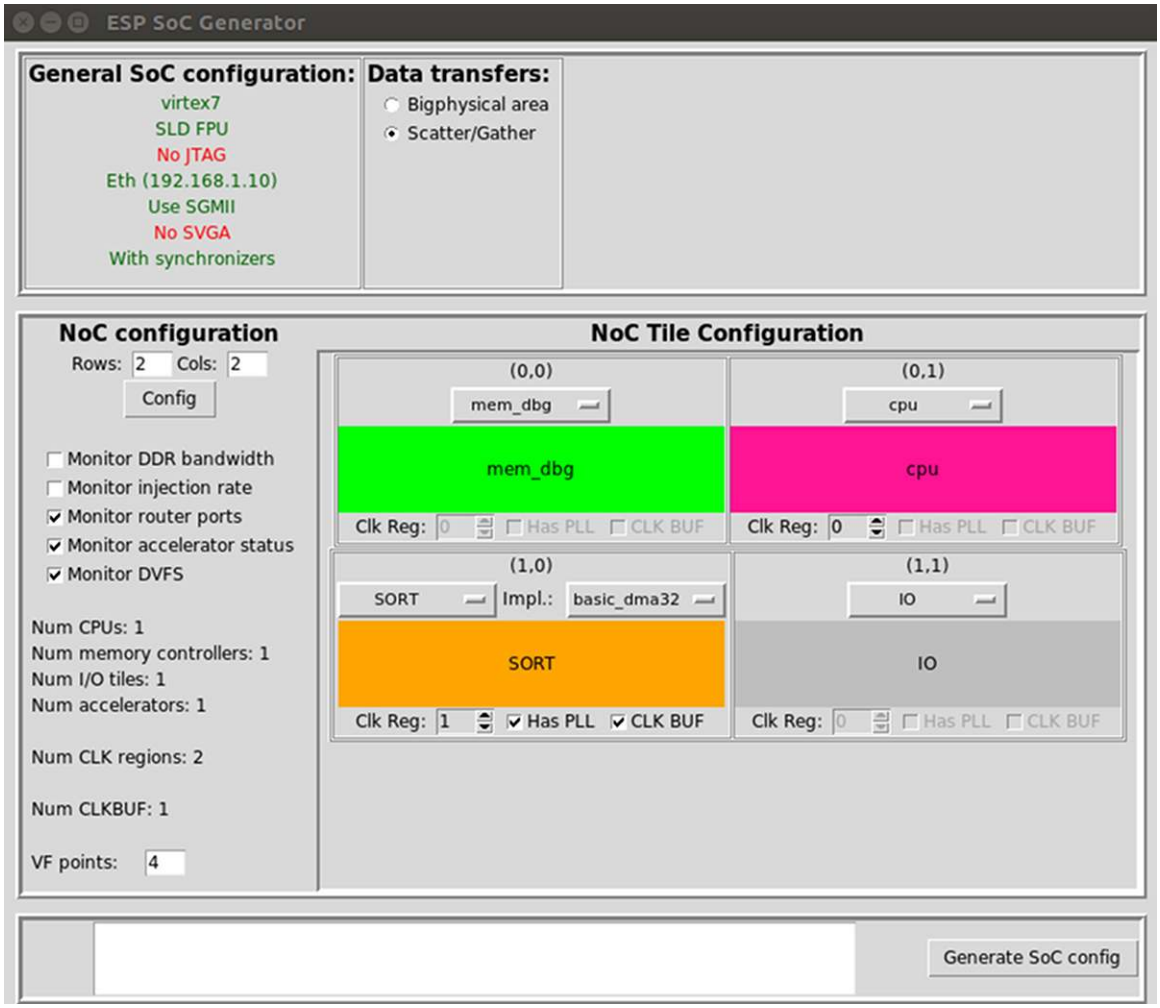


Figure A.7: ESP SoC Generator graphic user interface.

- In this example, the selected target technology is "virtex7" and corresponds to the variable TECHLIB assigned in the local Makefile. Note that a corresponding technology folder must exist.

- The type of FPU is chosen from the GRLIB configuration. By default the SLD FPU designed as part of Open-EPS is selected. This configuration can be overwritten by running `make grlib-xconfig` and selecting the tab "Processor" → "Floating-Point Unit".

- The JTAG debug link is disabled by default to allow users to insert the integrated logic an-

alyzer for debug purposes. In order to peek at internal signals, in fact, the integrated logic analyzer from Xilinx requires exclusive access to the JTAG test access point of the FPGA. This configuration can be overwritten by running `make grlib-xconfig` and enabling the radio button "Debug Link" → "JTAG Debug Link".

- The Ethernet debug link and its IP address can be configured as explained in Section A.2.1.

- Depending on the target FPGA board, the Ethernet may require the serial gigabit media independent interface (SGMII) IP from Xilinx. This configuration is chosen automatically based on the content of the constraint folder for the selected FPGA board.

- The GRLIB SVGA video output is enabled by default only for the proFPGA system, assuming the appropriate daughter card is installed as shown in Fig. A.6. This configuration can be overwritten by running `make grlib-xconfig` and changing the status of the radio button "Peripherals" → "Keyboard and VGA interface" → "SVGA graphical frame buffer".

- The instance of dual-clock FIFO synchronizers is inferred automatically based on the configuration of the clock regions for each tile. If the "Clk Reg" setting for all tiles is left to the default value (i.e. 0) and there is only one memory tile, then no synchronization is required between the NoC and the tiles. Note that two memory tiles cannot belong to the same clock domain because the two DRAM channels have independent controllers.

The panel labeled "Data transfers" allows users to choose two different memory allocation methods for the accelerators. Choosing "Bigphysical area" implies that the DMA buffers will be allocated as contiguous physical memory relying on the Linux `bigphysarea` patch. This option implements simpler DMA controllers without TLB, which will slightly improve performance and reduce resource usage. Note, however, that benefits are only noticeable for accelerators operating on small data sets in the order of a few MB. By selecting "Scatter/Gather", instead, the SoC generator instantiates a TLB in every accelerator tile sized depending on the XML specification of the corresponding accelerator. This option assumes that users leverage the `contig_alloc` memory allocation method as explained in Chapter 5.

The panel labeled "NoC configuration" is used to select the size of the NoC mesh, to enable the ESP probes for run-time monitoring, and the number of voltage-frequency pairs. Note that the current

version of Open-ESP can only support four operating points. Probes can be instantiated on any ESP design, however, the embedded real-time monitor application requires a dedicated Ethernet port and relies on the proFPGA MMI64 interface, which is not available on the Xilinx VC707 development board. Users should choose the proFPGA system if they intend to record performance and power metrics during the FPGA execution with the provided ESP monitor.

The "Config" push button in the "NoC configuration" panel activates the panel labeled "NoC Tile Configuration". This is where the mix of tiles is selected to create a customized instance of an ESP system. Each tile can have one of the following types:

- **"mem_dbg"** is the memory and debug tile that provides access to one memory channel and integrates a debug interface to access and boot the system;
- **"mem_lite"** is a memory tile that provides access to one additional memory channel;
- **"cpu"** is a processor tile that integrates an off-the-shelf processor (LEON3 by default);
- **"<accelerator name>"** is a tile that encapsulates an accelerator of type <accelerator name>;
- **"IO"** is a miscellaneous tile that includes I/O peripherals, such as UART and DVI, and shared resources, such as the interrupt controller and the main system timer.

A correct configuration of Open-ESP must satisfy several requirements; any violation will be listed in the message box at the bottom of the SoC generator and will prevent users from generating the system memory map and routing tables. Specifically, there must be one and only one "mem_dbg" tile that includes one memory channel and the debug links. Similarly there must be one and only one "IO" tile that includes UART and DVI interfaces, the interrupt controller and the system timer. Additional memory tiles must be of the type "mem_lite". Adding one or more "mem_lite" tiles implies that the address space is split equally across all memory channels, including the one integrated in the "mem_dbg" tile, which is assigned to the lowest portion of the address space. With respect to computing elements, there must be at least one "cpu" tile, while there can be any number of accelerator tiles, including none. Users can select the RTL implementation of accelerator tiles by using the drop-down menu next to the label "Impl.". The names of the available implementations correspond to the HLS configurations specified in the "project.tcl" file before running the high-level synthesis tool. Note that different accelerator tiles can integrate the same accelerator implementation. At the

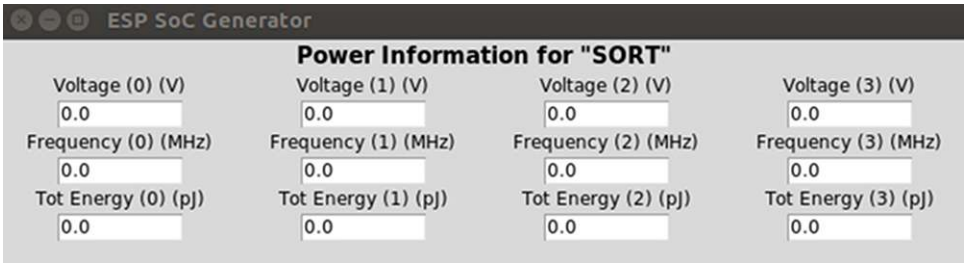


Figure A.8: ESP SoC Generator window tab for energy consumption.

RTL level accelerators of the same type share the device ID, which is used to discover the accelerator during the initialization of the device driver. All accelerators, however, are distinguished by a numeric identifier and by the address assigned to their set of registers.

The clock region selector ("Clk Reg") allows users to determine the clock domain of every processor and accelerator tile. All tiles with a clock region different from zero support DVFS and there must be one and only one tile integrating the PLL that drives the clock for each DVFS-enabled region. Furthermore, tiles that host a PLL need at least one clock buffer to drive the clock distribution tree, but they may include an optional clock buffer for improved quality of result. Note that the number of available clock buffers is limited and is specified in the data-sheet of the FPGA device. Since some clock buffers are silently instantiated as part of the components connected to I/O (e.g. Ethernet, DDR controller, etc.), the users should only enable the optional clock buffer for complex accelerator tiles, which may otherwise fail timing constraints.

When DVFS is enabled for an accelerator tile, the users should annotate the average energy consumption per clock cycle of the selected accelerator for all of the operating points. Double clicking on the colored rectangle of an accelerator tile opens a new window, similar to the one in Fig. A.8. For each operating point the users should specify the supply voltage, the target frequency, and the average energy per clock cycle. This information is used by the ESP monitor application to compute energy and power dissipation estimates during the execution on FPGA.

Once the configuration is completed, the push button "Generate SoC config" can be used to create the RTL files that specify the system memory mapping and the routing tables. In addition, header files with the address of the system probes are generated to enable the real-time monitor application to access the system information from FPGA. At this stage users can simulate and implement the system by using the same set of commands illustrated in Section A.2.2.

A.3 Supporting a Different Technology

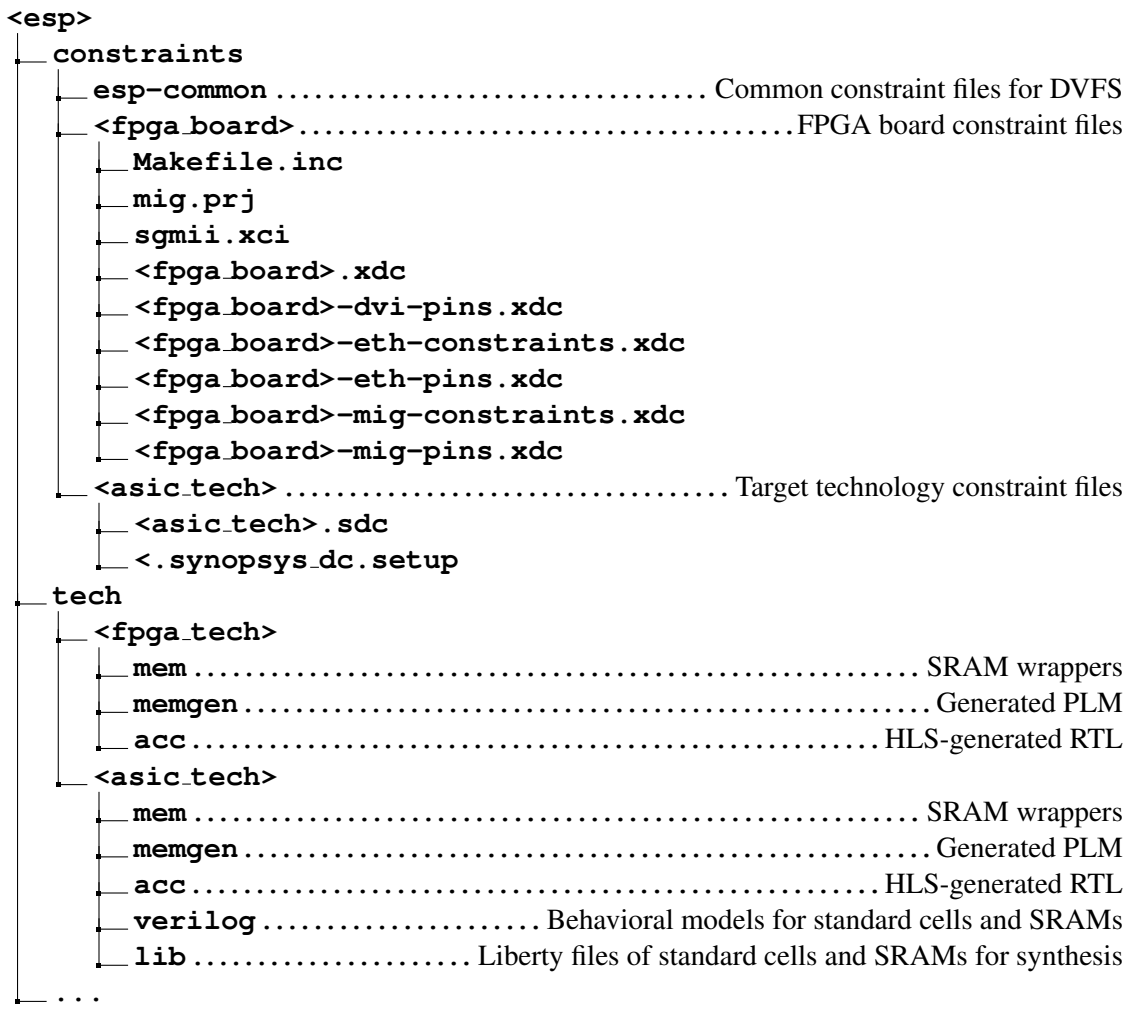


Figure A.9: Technology-dependent files in Open-ESP.

The current release of Open-ESP embeds the implementation flow for the Xilinx FPGA devices. In particular, the available constraint files target the FPGA devices integrated in the Xilinx VC707 development board and in the proFPGA module FM-XC7V2000T-R2. For these two boards Open-ESP has a corresponding folder that includes all necessary constraint files for Vivado, as shown in Fig. A.9. Users may add support for any other Xilinx-based FPGA board by creating the same set of files in a new folder. The name of such folder must match the content of the environment variable "BOARD", which is set in the local Makefile of an SoC design. Note that all the `xdc` files shown in

Fig. A.9 must exist, but some of them can be empty if not required. For example, if the new target board has no support for Ethernet, the related constraint files can be empty. The IP configuration files for the memory controller and the Ethernet interface are optional, but if no memory controller IP is provided, the user must replace it with a custom implementation. Recall, in fact, that any instance of Open-ESP must include one memory and debug tile, which provides access to external memory.

If the chosen board integrates a Xilinx FPGA device not part of the Virtex7 family, users need to create a new technology folder that includes the low-level SRAM wrappers, an empty folder that serves as a target for the generated PLM and an empty folder used to store the HLS-generated RTL files. The structure of an FPGA technology folder is also shown in Fig. A.9. Note that unless specified in the data-sheet of the FPGA device, all Xilinx devices share the same library of primitives. Hence, creating a new technology folder can be done by copying the given "virtex7" folder into a new folder with the name of the target Xilinx technology (e.g. "artix7", "zynq", etc.). Note that each technology must have a separate folder because the HLS tool and the memory generator will implement different RTL based on the given target FPGA family.

Adding constraint files for an ASIC technology library, instead, only requires to implement one `sdc` file with timing constraints and the `.synopsys_dc.setup` file that specifies the name of the target technology libraries, including SRAMs. In this case, the technology folder must include additional subdirectories, as illustrated in Fig. A.9. These directories contain the liberty files for synthesis: one for the standard cells and one for each SRAM primitive block. If using DVFS there must be multiple versions of each liberty file characterized for all the desired operating points. Finally, the verilog folder must contain behavioral models for the SRAM primitive blocks to enable RTL simulation. For gate-level simulation the behavioral model of the standard cells must be provided as well [2].

[2] An agreement with the vendor providing the physical design kit forbids the release of the technology-dependent files for the CMOS library used for the ESP research.

A.4 Building the ESP Software Stack

TARGET	DESCRIPTION
soft	compile the bare-metal program systest.c. The target architecture is selected based on the variable "CPU_ARCH".
barec-all	compile all available bare-metal drivers for the target "CPU_ARCH". Executable are placed in "barec".
make linux	compile Linux for "CPU_ARCH", create root file-system and compile all ESP core drivers, available drivers for accelerators, and test applications. The Linux configuration must be specified in the "LINUX_CONFIG" variable.
<**accelerator**>**-barec**	compile the "CPU_ARCH" bare-metal device driver for the specified accelerator.
<**accelerator**>**-driver**	compile the Linux device driver for the specified "CPU_ARCH" and accelerator. Drivers are placed to "sysroot/opt/drivers" and load automatically during OS boot.
<**accelerator**>**-app**	compile the Linux test application for the specified "CPU_ARCH" and accelerator. Executables are placed to "sysroot/applications/test".

Table A.3: Makefile targets to generate and configure an instance of Open-ESP.

Open-ESP provides simple Makefile targets to build the entire software stack. Relevant targets are listed in Table A.3 and they should be run from the SoC design folder.

For simulation purposes, the test program "systest.c" is dumped to the model of the memory, therefore the target `make soft` is invoked as a dependency of the simulation targets.

When moving to FPGA, instead, any program can be loaded to the memory present on the board. If available, users can compile bare-metal device drivers for their accelerators by using the command `make barec-all`, or `make <accelerator>-barec`. Similarly, Linux drivers and test applications can be compiled with the command `make linux`, or `make <accelerator>-driver` and `make <accelerator>-app`. When running Linux-related targets, the Makefile first generates the root file system based on a provided template, which includes an embedded version of the C library and busy-box. Then, it compiles the Linux kernel, according to the specified configuration which can be selected by setting the environment variable "LINUX_CONFIG" in the local Makefile. Lastly, the Makefile looks for the available device drivers and test applications, compiles them and copies the executable files to the root file system, so that they will be available on the target system after booting Linux.

A.4.1 Adding a New Device Driver

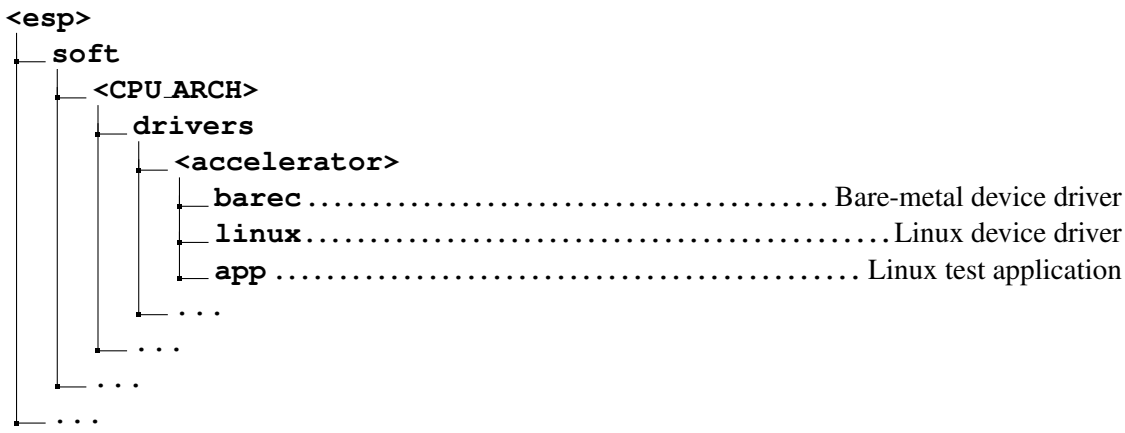


Figure A.10: Device drivers location in Open-ESP.

In order to add support for a new accelerators, users are responsible for implementing the accelerator-specific portion of the device driver and the test application. These files should be placed in the Open-ESP repository at the paths shown in Fig. A.10. The given driver example for SORT shows how to implement the necessary functions and data structures. Specifically:

- `<accelerator>_prep_xfer` writes to the configuration registers that must match what specified in the accelerator xml file described in Section A.1;
- `<accelerator>_xfer_input_ok` checks that the user-provided configuration parameters are within acceptable ranges;
- `<accelerator>_probe` performs the initialization steps for a given accelerator and reads the read-only registers to gather information about the capabilities of every matching accelerator discovered in the system;
- `<accelerator>_remove` frees the memory used to allocate accelerator-specific data structures;
- `<accelerator>_init` calls the ESP initialization function;
- `<accelerator>_exit` calls the ESP exit funtion;

- `<accelerator>_device` is a data structure that defines accelerator-specific parameters and one generic ESP accelerator;
- `<accelerator>_device_ids` is a data structure that lists all devices that the driver is able to handle (the listed device IDs must match what specified in the accelerator xml file described in Section A.1).
- `<accelerator>_access` is a data structure that defines the parameters for the `io_ctl` system call that invokes the accelerator.

The test application and the bare-metal drivers do not need to comply with a strict template. However, it is recommended to leverage the bare-metal library for probing devices and to follow the structure of the given test application provided with the the example for SORT. Technical details about the test application, the ESP memory allocator (`contig_alloc`) and the ESP multi-threaded library are discussed in Section 4.3.
Thanks to the Open-ESP software stack the user is only required to write few lines of the device driver. The goal for a future release of Open-ESP is to generate these lines as well and have users focus on the test application only.

A.4.2 Connecting to an ESP instance with LEON3

When integrating the LEON3 [Gaisler, 2004] embedded processor, users can build the necessary files to run Linux on FPGA with a few commands. Specifically, assuming that all necessary software and the configuration files are complete, the following commands allow to run the ESP instance on a Xilinx VC707 board.

Path: <esp>/socs/xilinx-vc707-xc7vx485t

```
$>  make vivado-syn
$>  make linux.dsu
$>  make vivado-prog-fpga
$>  grmon -eth <Debug Link IP address> -u -nb -stack <Stack address>
$>  load linux.dsu
$>  run
```

In order to connect to the proFPGA system, instead, the list of commands changes as follows:

Path: <esp>/socs/profpga-xc7v2000t

```
$>   make vivado-syn
$>   make linux.dsu
$>   make profpga-prog-fpga
$>   grmon -eth <Debug Link IP address> -u -nb -stack <Stack address>
$>   load linux.dsu
$>   run
```

The `grmon` application is part of the debug tools provided by the GRLIB library [Gaisler, 2004]. This application allows the user to connect to the LEON3 processor, load the desired program in memory (Linux in this case) and run the application. The `-eth` flag specifies that the debug link is Ethernet, but can be replaced with `-jtag` and no arguments if the JTAG link is enabled and the cable is connected to the workstation that runs `grmon`. The flag -u redirects the UART input and output to the `grmon` console. This makes it possible to interact with the LEON3 processor even without connecting the UART cable. Alternatively, when the UART is connected to the workstation running `grmon` and the option `-u` is not used, any serial console can visualize the system output and forward inputs from keyboard. The flag `-nb` disables the debug unit exceptions and prevents a failure during Linux boot. This option may be removed for bare-metal execution if the user wants `grmon` to catch all exceptions. Finally, the `-stack` option is necessary to tell the operating system which portion of the address space is exclusively reserved to the processor and which is, instead, managed by `contig_alloc`. The value used as a stack pointer for `grmon` must be consistent with the parameter passed to the `contig_alloc` module. Parameters for `contig_alloc` can be changed by updating the initialization script at the path `sysroot/etc/rc.d/init.d/drivers` and rebuilding the root file system. For instance, if the starting address passed to `contig_alloc` is 0x60000000, then the stack pointer should be set to 0x5ffffff0: this address is aligned to the size of a cache line and points to the last portion of memory exclusively dedicated to the processor.

Appendix B

Beyond Embedded

The original concept of ESP pertains to the context of high-performance embedded systems. Nevertheless, the flexible ESP system-level design methodology can be applied to different scenarios, while preserving the principles of scalability and the ease of integrating heterogeneous components. In practice, it is possible to obtain a non-embedded version of ESP by removing the embedded processors, relaxing the constraints on the size of each tile, and creating high-speed links to interface with workstation class and server class processors.

This appendix presents two preliminary non-embedded instances of ESP that will be integrated in the Open-ESP repository to extend its user-base beyond the embedded-system research community.

B.1 AXI-Based Scalable Platform

ESP can be interfaced with an ARM-based system through standard AXI interfaces. Fig. B.1 shows the block diagram of the AXI tile that replaces the functionality of the memory tile in ESP. Three platform services must be implemented on this tile: interrupt handling, DMA, and register access. Assuming the host system is entirely relying on the AXI interconnect, these three services can be implemented with three proxies that interface the NoC with the interconnect. Note that the bus itself, highlighted in red, is part of the host system; hence, it is not integrated in the ESP tile.

Interrupts are generally delivered as a bundle of wires, each corresponding to one interrupt line. The proxy for the interrupt requests receives a one-flit packet from the NoC with the activated interrupt number and sets the corresponding line of the host system.

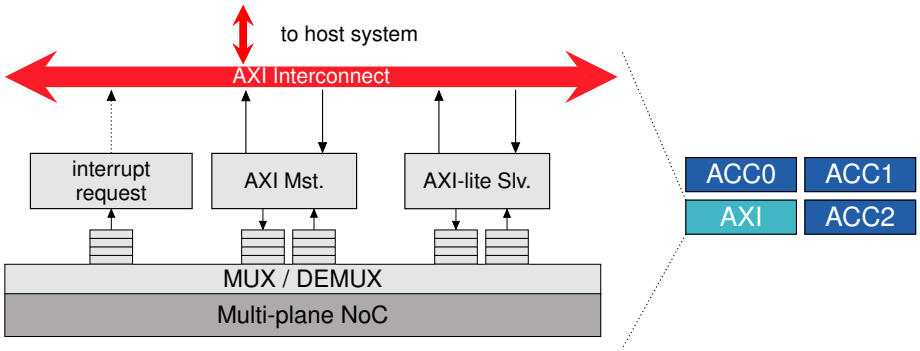


Figure B.1: Non-embedded instance of ESP with AXI interface.

The DMA requests from accelerators are forwarded from the NoC to the AXI interconnect with a proxy that is able to initiate master transactions in bursts. Depending on the specific version of the AXI interconnect and the features that the host system supports, the length of the burst transactions may be limited. In this case, the proxy will split the DMA request into multiple burst transactions with memory.

Finally, the host processor can access the registers of the accelerators to configure and run them through a slave interface. A proxy implementing the AXI-lite protocol serves this purpose by translating the bus transactions into NoC packets directed to the addressed accelerator.

In order to improve performance, there can be multiple AXI tiles, each providing access to a portion of the address space and handling a subset of the available interrupt lines. Similarly to the additional memory tiles in the embedded version of ESP, having multiple AXI tiles improves the traffic patterns on the network and enables a better utilization of the bandwidth of the host interconnect and memory. The slave proxy could be split as well by having each proxy addressing some of the ESP accelerators. Given the low bandwidth requirements for register access, however, it is recommendable to have only one AXI tile integrating the slave proxy to save area and reduce the requirements on the number of AXI interfaces for the host system.

Note that the entire ESP software stack can still be used in an AXI-based instance of the SoC, including the device drivers. Depending on how the host system discovers attached peripherals, however, a few lines of the ESP core driver may need to change in order to support correct bus enumeration and be able to communicate with the accelerators.

B.2 PCIe-Based Scalable Platform

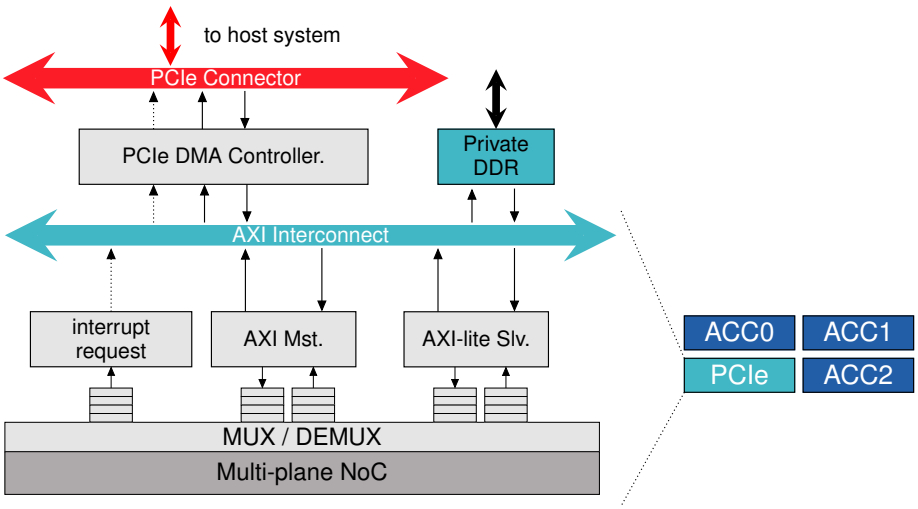


Figure B.2: Non-embedded instance of ESP with PCI Express interface.

Starting from the AXI-based implementation illustrated in the previous section, an ESP instance on FPGA can also be interfaced with a generic workstation or server through the high-speed serial link offered by the PCI Express (PCIe) standard. In this case, the AXI interconnect is integrated in the ESP tile and serves as an intermediate layer for the DMA controller over PCIe provided by the FPGA vendor. The PCIe connector, instead, is part of the host system.

The accelerators on ESP have no direct access to the host system bus and memory. Therefore, a dedicated channel to a private memory is placed on the AXI interconnect. Differently from the ESP instance for AXI-based systems, the private memory is the target for all the transactions initiated by the master proxy. Data are transferred from system memory to the ESP dedicated memory by the the PCIe DMA controller, which is also a master on the AXI interconnect. This is the same behavior adopted by discrete GPUs that rely on the high bandwidth of their private DDR memory. Beside handling DMA transactions, the PCIe DMA controller can also address registers in ESP and deliver interrupt requests to the host system. Depending on the target address, the controller will either initiate a DMA transaction, or forward the request to the AXI-lite slave proxy.

Even in this scenario the ESP software stack can be reused. However, the core driver must integrate the necessary functionality to control the third-party IP for PCIe transactions by embedding third-party code provided by the IP vendor.

Part V

Bibliography

Bibliography

[Aarno and Engblom, 2014] D. Aarno and J. Engblom. *Software and System Development Using Virtual Platforms: Full-System Simulation with Wind River Simics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[Abadi *et al.*, 2015] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Ahrons *et al.*, 1965] R. Ahrons, M. Mitchell, and J. Burns. MOS micropower complementary transistor logic. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, volume VIII, pages 80–81, Feb 1965.

[Akopyan *et al.*, 2015] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015.

[Amazon, 2017] Amazon. Amazon EC2 F1 Instances (Preview). Available at: https://aws.amazon.com/ec2/instance-types/f1/, 2017.

[Andersen *et al.*, 2013] T. M. Andersen, F. Krismer, J. W. Kolar, T. Toifl, C. Menolfi, L. Kull, T. Morf, Marcel, Kossel, M. Brandli, P. Buchmann, and P. A. Francese. A 4.6W/mm2 power density 86% efficiency on-chip switched capacitor DC-DC converter in 32 nm SOI CMOS. In *Proceedings of the IEEE Applied Power Electronics Conference*, pages 692–699, March 2013.

[Andersen *et al.*, 2014] T. M. Andersen, F. Krismer, J. W. Kolar, T. Toifl, C. Menolfi, L. Kull, T. Morf, M. Kossel, M. Brandli, P. Buchmann, and P. A. Francese. A sub-ns response on-chip switched-capacitor DC-DC voltage regulator delivering 3.7W/mm2 at 90% efficiency using deep-trench capacitors in 32nm SOI CMOS. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 9–13, February 2014.

[Andersen *et al.*, 2015] T. M. Andersen, F. Krismer, J. W. Kolar, T. Toifl, C. Menolfi, L. Kull, T. Morf, M. Kossel, M. Brandli, and P. A. Francese. A feedforward controlled on-chip switched-capacitor voltage regulator delivering 10W in 32nm SOI CMOS. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 22–26, February 2015.

[Angiolini *et al.*, 2006] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo. Contrasting a NoC and a traditional interconnect fabric with layout awareness. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*, pages 124–129, March 2006.

[Arvind, 2003] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, page 249, 2003.

[Arvind, 2014] Arvind. Simulation is passé; all future systems require FPGA prototyping. *Keynote Address at Embedded System Week (ESWEEK)*, October 2014.

[Asanović and Patterson, 2014] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[Asanović *et al.*, 2016] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, K. Asanović, R. Avizienis, B. Keller, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto,

A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. In *SemanticScholar*, 2016.

[Awasthi *et al.*, 2010] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330. ACM, 2010.

[Bachrach *et al.*, 2012] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1212–1221, June 2012.

[Bailey and Martin, 2006] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer-Verlag, 2006.

[Bainbridge and Furber, 2002] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, Sep 2002.

[Barker *et al.*, 2013] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[Benson *et al.*, 2012] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, February 2012.

[Bjerregaard and Sparso, 2005] T. Bjerregaard and J. Sparso. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Design, Automation and Test in Europe*, pages 1226–1231 Vol. 2, March 2005.

[Black *et al.*, 2009] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[Black *et al.*, 2010] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up, Second Edition*. Springer-Verlag, 2010.

[Borkar and Chien, 2011] S. Borkar and A. A. Chien. The future of microprocessors. *Communication of the ACM*, 54:67–77, May 2011.

[Borkar, 2009] S. Borkar. Design perspectives on 22nm CMOS and beyond. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 93–94, 2009.

[Burton *et al.*, 2014] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill. FIVR – fully integrated voltage regulators on 4th generation Intel Core SoCs. In *Proceedings of IEEE Applied Power Electronics Conference and Exposition*, pages 16–20, March 2014.

[Carloni *et al.*, 2001] L. P. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[Carloni, 2004] L. P. Carloni. *Latency-Insensitive Design*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, Berkeley, CA 94720, August 2004. Memorandum No. UCB/ERL M04/29.

[Carloni, 2015] L. P. Carloni. From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, November 2015.

[Carloni, 2016] L. P. Carloni. Invited - the case for embedded scalable platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 17:1–17:6, New York, NY, USA, 2016. ACM.

[Chang *et al.*, 2010] L. Chang, R. K. Montoye, B. L. Ji, A. J. Weger, K. G. Stawiasz, and R. H. Dennard. A fully-integrated switched-capacitor 2:1 voltage converter with regulation capability and 90% efficiency at 2.3A/mm2. In *Proceedings of the Symposium on VLSI Circuits*, pages 55–56, June 2010.

[Chen *et al.*, 2014] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, New York, NY, USA, 2014. ACM.

[Chen *et al.*, 2016] Y. Chen, J. Cong, Z. Fang, B. Xiao, and P. Zhou. ARAPrototyper: Enabling rapid prototyping and evaluation for accelerator-rich architectures. *Cornell University Library*, abs [1610.09761], 2016.

[Collobert *et al.*, 2011] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[Concer *et al.*, 2010] N. Concer, L. Bononi, M. Soulie, R. Locatelli, and L. P. Carloni. The connection-then-credit flow control protocol for heterogeneous multicore systems-on-chip. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 29(6):869–882, June 2010.

[Cong *et al.*, 2011] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[Cong *et al.*, 2014] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[Cota *et al.*, 2014] E. Cota, P. Mantovani, M. Petracca, M. Casu, and L. P. Carloni. Accelerator memory reuse in the dark silicon era. *Computer Architecture Letters*, 13(1):9–12, Jan-Jun 2014.

[Cota *et al.*, 2015] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 202:1–202:6, June 2015.

[Cota *et al.*, 2016] E. G. Cota, P. Mantovani, and L. P. Carloni. Exploiting private local memories to reduce the opportunity cost of accelerator integration. In *Proceedings of the International Conference on Supercomputing*, pages 27:1–27:12, New York, NY, USA, 2016. ACM.

[Coussy and Morawiec, 2008a] P. Coussy and A. Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.

[Coussy and Morawiec, 2008b] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[Coussy *et al.*, 2009] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[Crockett *et al.*, 2014] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, UK, 2014.

[Daga *et al.*, 2011] M. Daga, A. M. Aji, and W. c. Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*, pages 141–149, July 2011.

[Dally and Towles, 2001] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 684–689, June 2001.

[Dally *et al.*, 2013] W. Dally, C. Malachowsky, and S. Keckler. 21st century digital design tools. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, May 2013.

[Danowitz *et al.*, 2012] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. Cpu db: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, April 2012.

[Dasika *et al.*, 2008] G. Dasika, S. Das, K. Fan, S. Mahlke, and D. Bull. DVFS in loop accelerators using BLADES. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 894–897, June 2008.

[DDWG, 1999] D. D. W. G. DDWG. *Digital Visual Interface*. DDWG Promoters, 1999.

[De *et al.*, 2017] V. De, S. Vangal, and R. Krishnamurthy. Near threshold voltage (ntv) computing: Computing in the dark silicon era. *IEEE Design Test*, 34(2):24–30, April 2017.

[De Micheli, 1994] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[Dennard *et al.*, 1974] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[Densmore *et al.*, 2006] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design & Test of Computers*, 23(5):359–374, May 2006.

[Di Guglielmo *et al.*, 2014] G. Di Guglielmo, C. Pilato, and L. P. Carloni. A design methodology for compositional high-level synthesis of communication-centric SoCs. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 128:1–128:6, June 2014.

[DiBene *et al.*, 2010] J. T. DiBene, P. R. Morrow, C.-M. Park, H. W. Koertzen, P. Zou, F. Thenus, X. Li, S. W. Montgomery, E. Stanford, R. Fite, and P. Fischer. A 400A fully integrated silicon voltage regulator with in-die magnetically coupled embedded inductors. In *Proceedings of Advanced Power Electronics Conference*, February 2010.

[Ernst *et al.*, 2003] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A low-power pipeline based on circuit-level timing speculation. In *IEEE Micro*, December 2003.

[Esmaeilzadeh *et al.*, 2011] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 365–376, New York, NY, USA, 2011. ACM.

[Esmaeilzadeh *et al.*, 2012] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[Eyerman and Eeckhout, 2011] S. Eyerman and L. Eeckhout. Fine-grained DVFS using on-chip regulators. *ACM Transactions on Architecture and Code Optimization*, 8(1):1–24, February 2011.

[Fajardo *et al.*, 2011] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao. Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 966–971, New York, NY, USA, 2011. ACM.

[Fingeroff, 2010] M. Fingeroff. *High-level Synthesis Blue Book*. Mentor Graphics Corp., 2010.

[Gaisler, 2004] J. Gaisler. An open-source VHDL IP library with plug&play configuration. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August, Toulouse, France*, pages 711–717, 2004.

[Gerstlauer *et al.*, 2009] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(10):1517–1530, October 2009.

[Goering, 2009] R. Goering. Are SoC development costs significantly underestimated? http://www.cadence.com/Community/blogs, 2009.

[Group, 2009] K. Group. Opencl specification. https://www.khronos.org/opencl/, 2009. Accessed: 2017-04-06.

[Gupta and Brewer, 2008] R. Gupta and F. Brewer. High-level synthesis: A retrospective. In P. Coussy and A. Morawiec, editors, *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.

[Gupta *et al.*, 2016] G. Gupta, T. Nowatzki, V. Gangadhar, and K. Sankaralingam. Open-source hardware: Opportunities and challenges. *Cornell University Library*, abs/1606.01980, 2016.

[Hameed *et al.*, 2010] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 37–47, June 2010.

[Hamey, 2015] L. G. C. Hamey. A functional approach to border handling in image processing. In *Proceedings of the International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–8, Nov 2015.

[Hennessy and Patterson, 2011] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[Herbert and Marculescu, 2007] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, August 2007.

[Herbert *et al.*, 2012] S. Herbert, S. Garg, and D. Marculescu. Exploiting process variability in voltage/frequency control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(8):1392–1404, August 2012.

[Hill and Marty, 2008] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[Horowitz, 2014] M. Horowitz. Computing's energy problem (and what we can do about it). In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 10–14, February 2014.

[Isci *et al.*, 2006] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–358, December 2006.

[ITRS-2.0, 2015] ITRS-2.0. International Technology Roadmap for Semiconductors. *www.itrs2.net*, 2015.

[ITRS, 2011] ITRS. 2011 International Technology Roadmap for Semiconductors. *www.itrs2.net*, 2011.

[Jevtic *et al.*, 2015] R. Jevtic, H.-P. Le, M. Blagojevic, S. Bailey, K. Asanovic, E. Alon, and B. Nikolic. Per-core DVFS with switched-capacitor converters for energy efficiency in manycore processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(4):723–730, April 2015.

[Jia *et al.*, 2014] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the International Conference on Multimedia (MM)*, pages 675–678, New York, NY, USA, 2014. ACM.

[Jiang *et al.*, 2015] W. Jiang, K. Bhardwaj, G. Lacourba, and S. M. Nowick. A lightweight early arbitration method for low-latency asynchronous 2d-mesh noc's. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[Juan *et al.*, 2013] D.-C. Juan, S. Garg, J. Park, and D. Marculescu. Learning the optimal operating point for many-core systems with extended range voltage/frequency scaling. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, September 2013.

[Kasapaki and Sparso, 2014] E. Kasapaki and J. Sparso. Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 45–52, May 2014.

[Kathail *et al.*, 2016] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo. Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 4–4, New York, NY, USA, 2016. ACM.

[Kaul *et al.*, 2012] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (ntv) design - opportunities and challenges. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1149–1154, June 2012.

[Kaxiras and Martonosi, 2008] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[Kim and Seok, 2015] S. Kim and M. Seok. Variation-tolerant, ultra-low-voltage microprocessor with a low-overhead, within-a-cycle in-situ timing-error detection and correction technique. *IEEE Journal of Solid-State Circuits*, 50(6):1478–1490, June 2015.

[Kim *et al.*, 2008] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 123–134, February 2008.

[Komuravelli *et al.*, 2015] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have your scratchpad and cache it too. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 707–719, June 2015.

[Kornaros and Pnevmatikatos, 2014] G. Kornaros and D. Pnevmatikatos. Dynamic power and thermal management of noc-based heterogeneous mpsocs. *ACM Transactions on Reconfigurable Technology and Systems*, 7(1):1–26, February 2014.

[Kumar *et al.*, 2009] A. Kumar, J. Rabaey, and K. Ramchandran. SRAM supply voltage scaling: A reliability perspective. In *Proceedings of the International Symposium on Quality of Electronic Design (ISQED)*, pages 782–787, March 2009.

[Kunapareddy *et al.*, 2015] S. Kunapareddy, S. D. Turaga, and S. S. T. M. Sajjan. Design of asynchronous noc using 3-port asynchronous t-routers. In *International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*, pages 1–5, March 2015.

[Lee *et al.*, 2015] S. K. Lee, T. Tong, X. Zhang, D. Brooks, and G.-Y. Wei. A 16-core voltage-stacked system with an integrated switched-capacitor DC-DC converter. In *Proceedings of the Symposium on VLSI Circuits*, pages 318–319, June 2015.

[Lee *et al.*, 2016] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P. F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, Mar 2016.

[Leibson, 2006] S. Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Li and Martinez, 2006] J. Li and J. F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 77–87, February 2006.

[Li *et al.*, 2011] B. Li, Z. Fang, and R. Iyer. Template-based memory access engine for accelerators in socs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC*, pages 147–153, Piscataway, NJ, USA, 2011. IEEE Press.

[Lim *et al.*, 2013] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. *SIGARCH Comput. Archit. News*, 41(3):36–47, June 2013.

[Liu *et al.*, 2012a] H.-Y. Liu, M. Petracca, and L. P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*, pages 641–646, March 2012.

[Liu *et al.*, 2012b] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 367–376, New York, NY, USA, 2012. ACM.

[Luk *et al.*, 2005] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005. ACM.

[Lyons *et al.*, 2012] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–22, January 2012.

[Macken *et al.*, 1990] P. Macken, M. Degrauwe, M. V. Paemel, and H. Oguey. A voltage reduction technique for digital systems. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 238–239, February 1990.

[Mair *et al.*, 2015] H. Mair, G. Gammie, A. Wang, S. Gururajarao, I. Lin, H. Chen, W. Kuo, A. Rajagopalan, W. Z. Ge, R. Lagerquist, S. Rahman, C. J. Chung, S. Wang, L. K. Wong, Y. C. Zhuang, K. Li, J. Wang, M. Chau, Y. Liu, D. Dia, M. Peng, and U. Ko. 23.3 a highly integrated smartphone soc featuring a 2.5ghz octa-core cpu with advanced high-performance and low-power techniques. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 1–3, Feb 2015.

[Mantovani *et al.*, 2016a] P. Mantovani, E. G. Cota, K. Tien, C. Pilato, G. D. Guglielmo, K. Shepard, and L. P. Carloni. An fpga-based infrastructure for fine-grained dvfs analysis in high-performance embedded systems. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, June 2016.

[Mantovani *et al.*, 2016b] P. Mantovani, G. D. Guglielmo, and L. P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC*, January 2016.

[Mantovani *et al.*, 2016c] P. Mantovani, E. G. Cota, C. Pilato, G. D. Guglielmo, and L. P. Carloni. Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, pages 1–10, Oct 2016.

[Martin and Smith, 2009] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[Metz, 2016] C. Metz. Microsoft bets its future on a reprogrammable computer chip. https://www.wired.com/2016/09/microsoft-bets-future-chip-reprogram-fly/, September 2016. Accessed: 2017-04-12.

[Moore, 1965] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114 ff., April 1965.

[Muralidhara *et al.*, 2011] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, New York, NY, USA, 2011. ACM.

[Nickolls and Dally, 2010] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[Nikolic, 2015] B. Nikolic. Simpler, more efficient design. In *Proceedings of the European Solid-State Circuits Conference (ESSCIRC)*, pages 20–25, Sept 2015.

[Nowick and Singh, 2015] S. M. Nowick and M. Singh. Asynchronous design - part 1: Overview and recent advances. *IEEE Design Test*, 32(3):5–18, June 2015.

[NVIDIA, 2006] NVIDIA. Cuda technology. http://www.nvidia.com/object/cuda_home_new.html, 2006. Accessed: 2017-04-06.

[Ortego and Sack, 2004] P. M. Ortego and P. Sack. Sesc: Superescalar simulator. http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/, December 2004. Accessed: 2017-04-17.

[Pagliari *et al.*, 2015] D. J. Pagliari, M. R. Casu, and L. P. Cartoni. Acceleration of microwave imaging algorithms for breast cancer detection via high-level synthesis. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 475–478, Oct 2015.

[Panda, 2001] P. R. Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)*, pages 75–80, New York, NY, USA, 2001. ACM.

[Park *et al.*, 2010] J. Park, D. S. Shin, N. Chang, and M. Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Proceedings of ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 419–424, August 2010.

[Park *et al.*, 2013] J. Park, I. Hong, G. Kim, Y. Kim, K. Lee, S. Park, K. Bong, and H. J. Yoo. A 646gops/w multi-classifier many-core processor with cortex-like architecture for super-resolution recognition. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 168–169, Feb 2013.

[Pilato *et al.*, 2014a] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proceed-*

ings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS), pages 18:1–18:10, October 2014.

[Pilato *et al.*, 2014b] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. System-level memory optimization for high-level synthesis of component-based socs. pages 18:1–18:10, New York, NY, USA, 2014. ACM.

[Pilato *et al.*, 2016] C. Pilato, Q. Xu, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. On the design of scalable and reusable accelerators for big data applications. In *Proceedings of the International Conference on Computing Frontiers (CF)*, pages 406–411, New York, NY, USA, 2016. ACM.

[Porter *et al.*, 2010] R. Porter, A. M. Fraser, and D. Hush. Wide-area motion imagery. *IEEE Signal Processing Magazine*, 27(5):56–65, 2010.

[ProDesign, 2014] E. G. ProDesign. proFPGA Prototyping Systems. http://www.prodesign-europe.com/profpga, 2014.

[Putnam *et al.*, 2014] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[Pyo *et al.*, 2015] J. Pyo, Y. Shin, H. J. Lee, S. i. Bae, M. s. Kim, K. Kim, K. Shin, Y. Kwon, H. Oh, J. Lim, D. w. Lee, J. Lee, I. Hong, K. Chae, H. H. Lee, S. W. Lee, S. Song, C. H. Kim, J. S. Park, H. Kim, S. Yun, U. R. Cho, J. C. Son, and S. Park. 23.1 20nm high-k metal-gate heterogeneous 64b quad-core cpus and hexa-core gpu for high-performance and energy-efficient mobile application processor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 1–3, Feb 2015.

[Qadeer *et al.*, 2013] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing.

In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–35, New York, NY, USA, 2013. ACM.

[Rangan *et al.*, 2009] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 302–313, June 2009.

[Rostislav *et al.*, 2005] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar. An asynchronous router for multiple service levels networks on chip. In *11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 44–53, March 2005.

[Salihundam *et al.*, 2011] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Yatin, Hoskote, S. Vangal, G. Ruhl, P. Kundu, , and N. Borkar. A 2 Tb/s 6x4 Mesh Network for a Single-Chip Cloud Computer With DVFS in 45 nm CMOS. *IEEE Journal of Solid-State Circuits*, 46(4):757–766, April 2011.

[Sanders *et al.*, 2013] S. R. Sanders, E. Alon, H.-P. Le, M. D. Seeman, M. John, and V. W. Ng. The road to fully integrated DC-DC conversion via the switched-capacitor approach. *IEEE Transactions on Power Electronics*, 28(9):4146–4155, September 2013.

[Sangiovanni-Vincentelli, 2007] A. Sangiovanni-Vincentelli. Quo vadis SLD: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[Schmit and Huang, 2016] H. Schmit and R. Huang. Dissecting xeon + fpga: Why the integration of cpus and fpgas makes a power difference for the datacenter: Invited paper. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 152–153, New York, NY, USA, 2016. ACM.

[Semeraro *et al.*, 2002] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwaradasas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2002.

[Shacham *et al.*, 2010] O. Shacham, O. Azizi, M. Wachs, S. Richardson, and M. Horowitz. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, Nov 2010.

[Shao *et al.*, 2015] Y. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. The Aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3):58–70, May 2015.

[Shukla *et al.*, 2013] S. K. Shukla, Y. Yang, L. N. Bhuyan, and P. Brisk. Shared memory heterogeneous computation on pcie-supported platforms. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sept 2013.

[Simunic *et al.*, 2001] T. Simunic, L. Benini, A. Acquaviva, P. Glynnt, and G. D. Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 524–529, 2001.

[Song *et al.*, 2003] J. Song, T. Shepherd, M. Chau, A. Huq, L. Syed, S. Roy, A. Thippana, K. Shi, and U. Ko. A low power open multimedia application platform for 3g wireless. In *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, pages 377–380, Sept 2003.

[Sorin *et al.*, 2011] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[SPEC, 2017] S. P. E. C. SPEC. Cpu2006/2000/95, omp2012/2001, mpi2007 benchmarks. https://www.spec.org, Apr 2017. Accessed: 2017-04-03.

[Strano *et al.*, 2010] A. Strano, D. Ludovici, and D. Bertozzi. A library of dual-clock FIFOs for cost-effective and flexible MPSoC design. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, pages 20–27, July 2010.

[Sturcken *et al.*, 2012] N. Sturcken, M. Petracca, S. Warren, P. Mantovani, L. P. Carloni, A. V. Peterchev, and K. L. Shepard. A switched-inductor integrated voltage regulator with nonlinear feedback and network-on-chip load in 45nm SOI. *IEEE Journal of Solid-State Circuits*, 47(8):1935–1945, August 2012.

[Sturcken *et al.*, 2013] N. Sturcken, E. J. O'Sullivan, N. Wang, P. Herget, B. C. Webb, L. T. Romankiw, M. Petracca, R. Davies, R. E. F. Jr., G. M. Decad, I. Kymissis, A. V. Peterchev, L. P. Carloni, W. J. Gallagher, and K. L. Shepard. A 2.5D integrated voltage regulator using coupled-magnetic-core inductors on silicon interposer. *IEEE Journal of Solid-State Circuits*, 48(1):244–254, January 2013.

[Synopsys, Inc., a] Synopsys, Inc. Power Compiler User Manual. Available at: http://www.synopsys.com/.

[Synopsys, Inc., b] Synopsys, Inc. SiliconSmart ACE User Manual. Available at: http://www.synopsys.com/.

[Taylor, 2012] M. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1131–1136, June 2012.

[Theano, Development Team, 2016] Theano, Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[Thomas *et al.*, 2014] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor. Cortexsuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 76–79, Oct 2014.

[Tien *et al.*, 2015] K. Tien, N. Sturcken, N. Wang, J. woong Nah, B. Dang, E. O'Sullivan, P. Andry, M. Petracca, L. P. Carloni, W. Gallagher, and K. L. Shepard. An 82%-efficient multiphase voltage-regulator 3D interposer with on-chip magnetic inductors. In *Proceedings of the Symposium on VLSI Circuits*, pages C192–C193, June 2015.

[Toprak-Deniz *et al.*, 2014] Z. Toprak-Deniz, M. Sperling, J. Bulzacchelli, G. Still, R. Kruse, S. Kim, D. Boerstler, T. Gloekler, R. Robertazzi, K. Stawiasz, T. Diemoz, G. English, D. Hui, P. Muench, and J. Friedrich. Distributed system of digitally controlled microregulators enabling per-core DVFS for the POWER8 microprocessor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 9–13, February 2014.

[Venkatesh *et al.*, 2010] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, March 2010.

[Vo *et al.*, 2013] H. Vo, Y. Lee, A. Waterman, and K. Asanović. A case for OS-friendly hardware accelerators. In *Proceedings of the Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*, pages 1–8, June 2013.

[Vogel *et al.*, 2015] P. Vogel, A. Marongiu, and L. Benini. Lightweight virtual memory support for many-core accelerators in heterogeneous embedded socs. In *Proceedings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*, pages 45–54, Oct 2015.

[Wang *et al.*, 2014a] N. Wang, D. Goren, E. O'Sullivan, X. Zhang, W. J. Gallagher, P. Herget, and L. Chang. Ultra-high-Q air-core slab inductors for on-chip power conversion. In *Proceedings of IEEE International Electron Devices Meeting (IEDM)*, pages 15–17, December 2014.

[Wang *et al.*, 2014b] X. Wang, J. Xu, Z. Wang, K. J. Chen, X. Wu, and Z. Wang. Characterizing power delivery systems with on/off-chip voltage regulators for many-core processors. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*, pages 1–4, March 2014.

[White, 2012] M. A. White. Low power is everywhere. Synopsys Insight Newsletter (online), 2012.

[Winterstein *et al.*, 2015] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides. Matchup: Memory abstractions for heap manipulating programs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 136–145, New York, NY, USA, 2015. ACM.

[Xilinx, a] Xilinx. 7 Series FPGAs Clocking Resources User Guide (UG472). Available at: http://www.xilinx.com/.

[Xilinx, b] Xilinx. 7 Series FPGAs MMCM and PLL Dynamic Reconfiguration Application Note (XAPP888). Available at: http://www.xilinx.com/.

[Xilinx, 2016] Xilinx. 7 Series FPGAs Overview (DS180 v2.2). Available at: http://www.xilinx.com/, 2016.

[Yang *et al.*, 2016] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer. LMC: Automatic resource-aware program-optimized memory partitioning. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 128–137, 2016.

[Yoon *et al.*, 2013] Y. Yoon, N. Concer, and L. P. Carloni. Virtual channels and multiple physical networks: Two alternatives to improve NoC performance. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 32(12):1906–1919, December 2013.

[Yuffe *et al.*, 2011] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 264–266, Feb 2011.

[Zyuban and Strenski, 2002] V. Zyuban and P. Strenski. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 166–171, 2002.