Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, DELLE TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE

Ciclo XXVI

Settore Concorsuale di afferenza: 09/E3 Elettronica

Settore Scientifico disciplinare: ING-INF/01

PARALLEL ARCHITECTURES FOR MANY-CORE SYSTEMS-ON-CHIP IN DEEP SUB-MICRON TECHNOLOGY

Presentata da: Daniele Bortolotti

Coordinatore Dottorato	**Relatore**
Prof. Alessandro Vanelli Coralli	Prof. Luca Benini

Esame finale anno 2014

Parallel Architectures for Many-Core Systems-On-Chip in Deep Sub-Micron Technology

Daniele Bortolotti

Dept. of Electrical, Electronic and Information Engineering (DEI)

University of Bologna

A thesis submitted for the degree of

Doctor of Philosophy

2014

Abstract

Despite the several issues faced in the recent past, the evolutionary trend of silicon has kept its constant pace. Exploiting the high integration density offered by *Moore's Law*, today an ever increasing number of cores is integrated onto the same die, thus we are observing the shift from the multi-core to the *many-core* era. The extraordinary computing performance achievable by the many-core paradigm is limited not only by Amdahl's law, but several other factors concur in reducing the degree of effectiveness of such platforms. Memory bandwidth limitation, a problem exacerbated in many-core systems, combined with the lack of efficient synchronization mechanisms can severely overcome the potential computation capabilities. Moreover, the huge HW/SW design space of such architectures requires accurate and flexible tools to perform early architectural explorations and validation of key design choices.

In this thesis we focus on the aforementioned aspects affecting modern many-core architectures. A flexible and accurate *Virtual Platform* has been developed as an infrastructure tool, targeting a typical many-core architecture. With the goal of both flexibility and accuracy in mind, such Virtual Platform (VP) is capable of highly-accurate insights in the micro-architectural domain, full-system simulations of heterogeneous Systems-on-Chips (SoCs) as well as energy efficiency analyses. The tool has been used to perform architectural explorations, focusing on instruction caching architecture and hybrid HW/SW synchronization mechanism for local (intra-cluster) and global (inter-cluster) communication.

Beside architectural implications of modern many-core SoCs, another paramount issue of embedded systems is considered in this work: energy efficiency. *Near Threshold Computing* (NTC) is today a key research area in the Ultra-Low Power (ULP) domain, as it promises a tenfold improvement in energy efficiency compared to super-threshold operation and it mitigates thermal bottlenecks. Un-

fortunately, the physical implications of modern deep sub-micron technological nodes are posing sever limits to the performance and reliability of modern designs. *Reliability* becomes a major obstacle when operating in NTC, especially memory operation becomes unreliable and can compromise system correctness. Read failure, due to the lack of Static Noise Margin (SNM), is one of the principal failure factors, limiting the efficiency of dynamic voltage scaling. In the present work a novel hybrid memory architecture is devised to overcome reliability issues and at the same time improve energy efficiency by means of aggressive voltage scaling when allowed by workload requirements. *Variability* is another great drawback of near-threshold operation. The greatly increased sensitivity to threshold voltage variations in today a major concern for electronic devices: conservative design margins are nowadays not feasible due to the enormous performance waste and new architectural techniques are under investigation to mitigate this problem. In the present work, a variation-tolerant extension of the baseline many-core architecture is presented. By means of a micro-architectural knobs inserted at design and a lightweight runtime control unit, we extend the baseline architecture to be dynamically tolerant to variations.

To my family.

Contents

List of Figures

LIST OF FIGURES

LIST OF FIGURES

List of Tables

List of Abbreviations

6T	6-Transistor SRAM cell
8T	8-Transistor SRAM cell
AFE	Analog Front End
CS	Compressed Sensing
DMA	Data Movement Engine
GPU	Graphics Processing Unit
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
L3	3rd Level Memory (off-chip)
LIC	Logarithmic Interconnection
MoT	Mesh Of Trees
MPSoC	Multi-Processor SoC
NoC	Network On Chip
NTC	Near Threshold Computing
P2012	STM Platform 2012
PE	Processing Elements
RISC	Reduced Instruction Set Computer
SNM	Static Noise Margin
SoC	System on Chip
TCDM	Tightly Coupled Data Memory
ULP	Ultra Low Power
VP	Virtual Platform
WBSN	Wireless Body Sensor Networks

Chapter 1

Introduction

In this chapter an overview of modern many-core architectures is presented, describing the evolutionary path of such systems and highlighting their most fundamental traits. Relevant examples of commercial many-core platforms follows, exposing the characteristics of such architectures at the cluster level, as a background knowledge to understand the properties of the target architecture considered in the present work. Finally, the thesis outline is presented to understand chapters organization and to highlight the main contributions.

1.1 Many-core architectures

Several variants of many-core architectures have been designed and are in use for years now. As a matter of fact, since the mid 2000s we observed the integration of an increasing number of cores onto a single integrated circuit die, known as a Chip Multi-Processor (CMP) or Multi-Processor System-on-Chip (MPSoC), or onto multiple dies in a single chip package. Manufacturers still leverage Moore's Law [66] (doubling of the number of transistors on chip every 18 months), but business as usual is not an option anymore: scaling performance by increasing clock frequency and instruction throughput of single cores, the trend for electronic systems in the last 30 years, has proved to be not viable anymore [6, 13, 33]. As a

consequence, computing systems moved to multi-core[1] designs and subsequently, thanks to the integration density, to the many-core era where energy-efficient performance scaling is achieved by exploiting large-scale parallelism, rather than speeding up the single processing units [6, 13, 33, 51].

Such trend can be found in a wide spectrum of platforms, ranging from general purpose computing, high-performance to the embedded world.

In the general purpose domain we observed the first multi-core processors almost a decade ago. Intel core duo [35] and Sony-Toshiba-IBM (STI) Cell Broadband Engine [41] are notable examples of this paradigm shift. The trend did not stop and nowadays we have in this segment many-core examples such as the TILE-Gx8072 processor, comprising seventy-two cores operating at frequencies up to 1.2 GHz [22]. Instead, when performance is the primary requisite of the application domain, we can cite several notable architectures such as Larrabee [81] for visual computing, the research microprocessors Intel's SCC [40] and Terascale project [88] and, more recently, Intel's Xeon Phi [38]. In the embedded world, we are observing today a proliferation of many-core heterogeneous platforms. The so-called asymmetric of heterogeneous design features many small, energy-efficient cores integrated with a full-blown processor. Its is emerging as the main trend in the embedded domain, since it represents the most flexible and efficient design paradigm. Notable examples of such architectures are the AMD Accelerated Processing Units [15], Nvidia TEGRA family [68], STMicroelectronics P2012/STHORM [11] or Kalray's many-core processors [44].

The work presented in this thesis is focused on the embedded domain where, more than in other areas, modern high-end applications are asking for increasingly stringent and irreconcilable requirements. An outstanding example consist of the mobile market. As highlighted in [87], the digital workload of a smartphone (all control, data and signal processing) amounts to nearly 100 Giga Operations Per Second (GOPS) with a power-budget of 1 Watt. Moreover, workload requirements increase at a steady rate, roughly by an order of magnitude every 5

[1]For clarity, the *multi-core* term is intended for platforms with 2 to few tens cores, while with *many-core* we refer to systems with tens to hundreds of cores. The distinction is not rigid and throughout the dissertation, the terms multi-core and many-core may be used indistinctly.

years.

From the architectural point of view, with the evolution from tens of cores to the current integration capabilities in the order of hundreds, the most promising architectural choice for many-core embedded systems is *clustering*. In a clustered platform, processing cores are grouped into small- medium-sized clusters (i.e. few tens), which are highly optimized for performance and throughput. Clusters are the basic "building blocks" of the architecture, and scaling to many-core is obtained by the replication and global interconnection through a scalable medium such as a Network-on-Chip (NoC) [10, 24]. Figure 1.1 shows a reference clustered

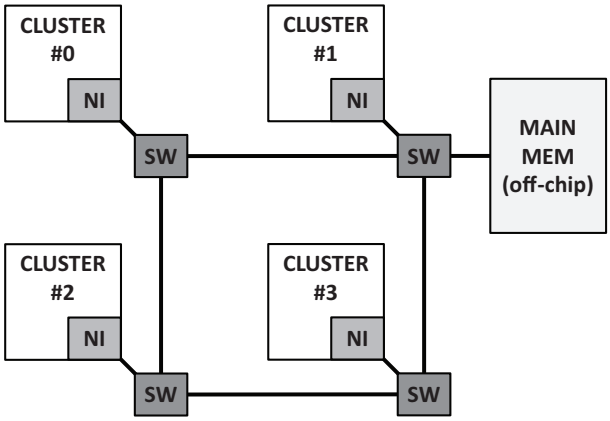

Figure 1.1: Clustered many-core architecture organized in a 4x4 mesh and off-chip main-memory

many-core architecture, organized in 4 clusters with a 4x4 mesh-like NoC for global interconnection. Next section reports some representative examples of recent architectures with a focus at the cluster level.

1.1.1 Cluster Architecture: Relevant Examples

The cluster architecture considered in this work is representative of a consolidated trend of embedded many-core design. Few notable examples are described, highlighting the most relevant characteristics of such architectures.

1.1.1.1 ST Microelectronics P2012/STHORM

Platform 2012 (P2012), also known as STHORM [11], is a low-power programmable many-core accelerator for the embedded domain designed by ST Microelectronics [83]. The P2012 project targets next-generation data-intensive embedded applications such as multi-modal sensor fusion, image understanding, mobile augmented reality [11]. The computing fabric is highly modular being structured in clusters of cores, connected through a Globally Asynchronous Network-on-Chip (GANoC) and featuring a shared memory space among all the cores. Each cluster is internally synchronous (one frequency domain) while at the global level the system follows the GALS (Globally Asynchronous Locally Synchronous) paradigm. In Figure 1.2 is shown a simplified block scheme of the internal structure of a single cluster. Each cluster is composed of a Cluster Controller (CC) and a multi-core computing engine, named ENCore, made of 16 processing elements. Each core is a proprietary 32-bit RISC core (STxP70-V4) featuring a floating point unit, a private instruction cache and no data cache.

Processors are interconnected through a low-latency high-bandwidth logarith-

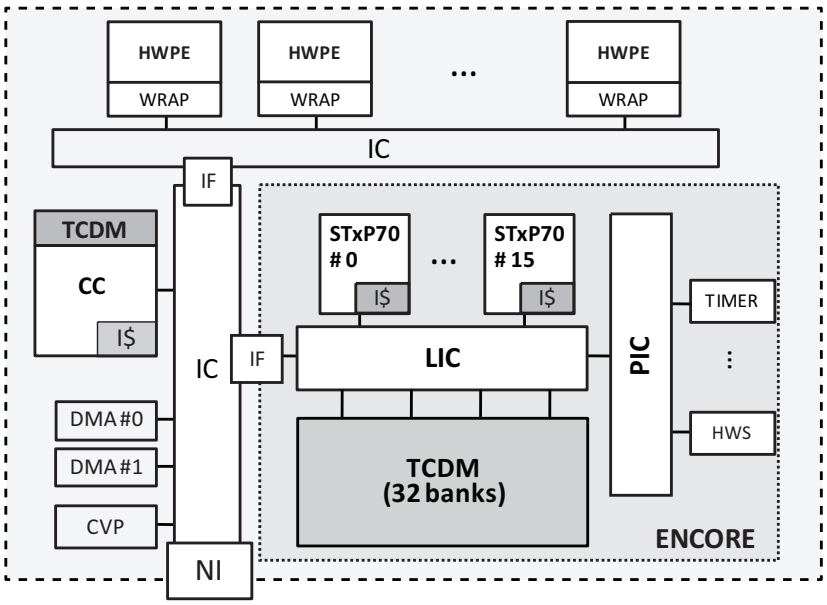

Figure 1.2: Overview (simplified) of P2012/STHORM cluster architecture

mic interconnect and communicate through a fast multi-banked, multi-ported tightly-coupled data memory (TCDM). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM accesses are performed with a two-cycles latency. The logarithmic interconnect consists of fully combinatorial Mesh-of-Trees (MoT) interconnection network. Data routing is based on address decoding: a first-stage checks if the requested address falls within the TCDM address range or has to be directed off-cluster. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. If no bank conflicts arise, data routing is done in parallel for each core. In case of conflicting requests, a round-robin based scheduler coordinates accesses to memory banks in a fair manner. Banking conflicts result in higher latency, depending on the number of concurrent conflicting accesses. Each cluster is equipped with a Hardware Synchronizer (HWS) which provides low-level services such as semaphores, barriers, and event propagation support, two DMA engines, and a Clock Variability and Power (CVP) module. The cluster template can be enhanced with application specific hardware processing elements (HWPEs), to accelerate key functionalities in hardware. They are interconnected to the ENCore with an asynchronous local interconnect (LIC). The first release of P2012 (STHORM) features 4 homogeneous clusters for a total of 69 cores and a software stack based on two programming models, namely a component-based Native Programming Model (NPM) and OpenCL-based [84] (named CLAM - CL Above Many-Cores) while OpenMP [23] support is under development.

1.1.1.2 Plurality HAL - Hypercore Architecture Line

Plurality Hypercore [2] is an energy efficient general-purpose machine made of several RISC processors. The number of processors can range from 16 up to 256 according to the processor model.

Figure 1.3 shows the overall architecture and the single processor structure, which is designed with the goal of simplicity and efficiency in mind (no I/D caches

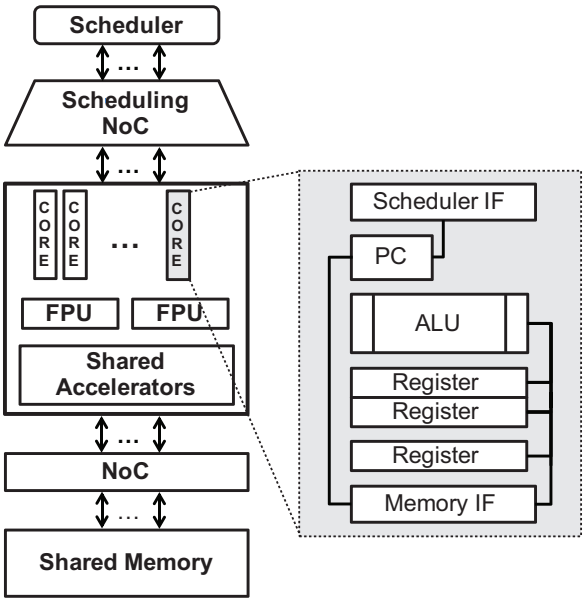

Figure 1.3: Plurality HAL architecture overview

nor private memory, no branch speculation) to save power and area. The memory system (i.e., I/D caches, off-chip main memory) is shared and processors access it through a high-performance logarithmic interconnect, equivalent to the interconnection described in Section 1.1.1.1. Processors share one or more Floating Point Units, and one or more shared hardware accelerators can be embedded in the design. This platform can be programmed with a task-oriented programming model, where the so-called "agents" are specified with a proprietary language. Tasks are efficiently dispatched by a scheduler/synchronizer called Central Synchronizer Unit (CSU), which also ensures workload balancing.

1.1.1.3 Kalray MPPA MANYCORE

Kalray Multi Purpose Processor Array (MPPA) [44] is a family of low-power many-core programmable processors for high-performance embedded systems. The first product of the family, MPPA-256, deploys 256 general-purpose cores grouped into 16 tightly-coupled clusters using a 28nm manufacturing process technology.

The MPPA MANYCORE chip family scales from 256 to 1024 cores with a

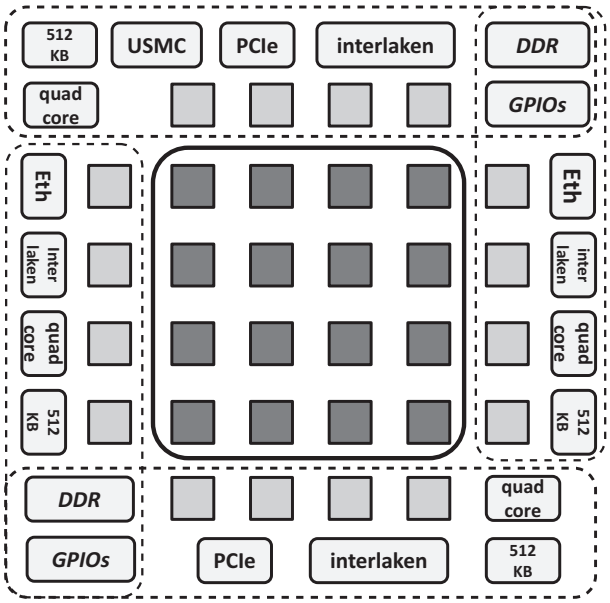

Figure 1.4: Overview (simplified) of Kalray MPPA architecture

performance of 500 Giga operations per second to more than 2 Tera operations per second with typical 5W power consumption. Global communication among the clusters is based on a Network-on-Chip. A simplified version of the architecture is shown in Figure 1.4.

Each core is a proprietary 32-bit ISA processor with private instruction and data caches. Each cluster has a 2MB shared data memory for local processors communication and a full-crossbar. Clusters are arranged in a 4x4 mesh and four I/O clusters provide off-chip connectivity through PCI (North and South) or Ethernet (West and East). Every I/O cluster has a four-cores processing unit, and N/S clusters deploy each a DDR controller to a 4GB external memory. The platform acts as an accelerator for an x86-based host, connected via PCI to the North I/O cluster. Accelerator clusters run a lightweight operative system named NodeOS [67], while I/O clusters run an instance of RTEMS [69].

1.2 Challanges

This section provides a brief overview of the most relevant research topics addressed in this thesis. It is not a comprehensive coverage of the complex challenges faced today by many-core systems, though it provides a background to understand the motivations behind this work.

Many-core architectures provide a new dimension to scale up the number of processing elements (cores) and, therefore, the potential computing capacity. To deploy the disruptive performance offered by such architectures, key challenges have nowadays to be faced. If we look at the mobile market, the goal is to provide 100GOPS within a 1W power budget [87]. As predicted in [13], energy will be the key limiter of performance, forcing processor designs to use large-scale parallelism with heterogeneous cores, or a few large cores and a large number of small cores operating at low frequency and low voltage, near threshold. Heterogeneity in compute and communication hardware will be essential to optimize for performance for energy-proportional computing and coping with variability. Aggressive use of customized accelerators will yield the highest performance and greatest energy efficiency on many applications. Efficient data orchestration will increasingly be critical, evolving to more efficient memory hierarchies and new types of interconnect tailored for locality and that depend on sophisticated software to place computation and data so as to minimize data movement.

This work focuses mainly on two aspects among the aforementioned challenges faced nowadays: *scalable performance* and *energy efficiency*.

Scalable Performance

Despite Pollack's Rule, which states that performance increase is proportional to the square root of the increase in complexity, clearly points to the direction of using a large number of small cores integrated on the same chip, some other bottlenecks in performance exist.

If we consider the cluster architectures presented in the previous section, it is clear that an efficient memory hierarchy utilization plays a key for a scalable

sustainable performance [85]. For any actual application with reasonable size, data may have to be accessed through the memory hierarchy, where long data-access delay occurs, in addition to the contention of the shared data structures in the lower levels of the memory hierarchy. The *memory-wall* problem [92] is due to the disparity of technology advance between CPU speed and memory data access latency. During last three decades memory latency in terms of processor cycles has increased, and the gap is still growing [13], requiring exploration of efficient usage of the memory hierarchy.

Moreover, data orchestration will increasingly be critical in such complex platforms. The many-core paradigm devise scalability by means of clusters combined with an highly scalable interconnection medium such as a Network-on-Chip. When highly parallel computation is allowed by the application running on the architecture, *communication* between the several nodes becomes a major concern for coordination and synchronization between remote clusters. Barrier synchronization becomes increasingly challenging as the level of integration in multi-processor systems-on-chip keeps growing. There is today little doubt on the fact that pure software implementations are not suitable to provide the needed scalability of barrier synchronization in embedded systems and that some form of hardware support is essential.

Energy Efficiency

Power consumption has become one of the main barriers in embedded computing systems and modern applications requirements, combined with a typically tight power budget, have made energy efficiency fundamental. By shrinking feature sizes, in deep-submicron technology (beyond 65nm) the supply voltage of digital systems has remained essentially constant and improvements on dynamic energy efficiency have dramatically stagnated, while leakage currents continue to increase. To face the reduced energy gains of classical super-threshold operation, a promising scenario is represented by the *near-threshold computing* (NTC) domain [13, 29, 59]. By reducing the supply voltage from the nominal value to the level of the threshold voltage the energy per operation has a tenfold improve-

ment with similar impact on performance penalties. [29, 31, 59, 96]. Reducing further the supply voltage in the sub-threshold region is less attractive since the performance will dominate the efficiency improvement. Although NTC provides excellent energy-frequency trade-offs, it faces three key barriers that must be overcome for widespread use: performance loss, performance variation and functional failure. In the context of this work, parallel architecture are the focus and they intrinsically provide a remedy to the reduced performance when not extreme computation is required.

Variability, is on the other hand, a great issue and it is testified by the huge amount of research effort to address this problem. Systematic and random variations are already significant issues in today's advanced technological nodes and operating at low-voltages exacerbates the effects of both. Performance uncertainty in the near-threshold region due to the global process variation alone increases to 5x from 1.3x at nominal supply voltage [20, 29]. Operating at this voltage also heightens sensitivity to temperature and supply ripple, both can have a detrimental effect on system reliability.

Also functional failure must be taken into account: more than the logic cells, embedded SRAM cells will suffer from static and random variations with the high risk of causing several functional failures. For instance, a typical 65nm SRAM cell has a failure probability of 10^{-7} at nominal voltage. However, at NTC this failure rate increases by 5 orders of magnitude to approximately 4%. Keeping the power supply of SRAM cells slightly higher than the core logic will reduce the error rate [20, 29], the leakage power and produce faster memory. On the other hand, the power-hungry memories of modern technology, cannot benefit from additional voltage-scaling if reliability aspects are not taken into account.

1.3 Thesis Outline

In this section we describe the organization of the remainder of the present thesis work.

Chapter 2 presents the Virtual Platform (VirtualSoC) that has been developed, targeting the full-system simulation of massively parallel heterogeneous Multi-Processor Systems-on-Chip (MPSoCs). Driven by flexibility, performance and cost constraints of demanding modern applications, heterogeneous MPSoC is the dominant design paradigm in the embedded system computing domain. The necessity to efficiently cope with the huge HW/SW design space provided by this scenario makes clearly a virtual platform capable of accurate architectural insights and full-system simulation one of the most important tool for both research and design purposes. All the analyses presented in this work and described in the next chapters, leverage ad-hoc versions of the VirtualSoC simulator customized for the specific goals.

Chapter 3 compares different architectures for instruction caching targeting tightly-coupled clusters. The analysis involves (i) private instruction caches per core and (ii) a shared instruction cache per cluster. Indeed, an effective instruction cache architecture is key to support the instruction fetch bandwidth required to have high-throughput and efficient systems. Due to the lack of sophisticated HW support to hide memory latency, the simple processors embedded in many-core systems may experience prolonged stalls on long-latency instructions fetch, with negative effects both on performance and energy consumption. An in-depth study of the two architectural templates is shown, based on the usage of both synthetic micro-benchmarks and real program workloads.

Chapter 4 focuses on a different architectural aspect: barrier synchronization mechanisms. Barrier synchronization is a key programming primitive for shared memory embedded MPSoCs. Communication plays a crucial role when different clusters work on a parallel distributed workload, a common application scenario for many-core systems. As the core count increases, software implementations cannot provide the needed performance and scalability, thus making hardware acceleration critical for such platforms. The proposed interconnect

extension proves its efficacy and the area overhead is marginal with respect to the performance improvements. As a final exploration, a comparison with traditional software implementations is assessed by integrating the HW barriers into the OpenMP programming model and synchronization efficiency is discussed.

Chapter 5 studies energy efficiency and reliability aspects. When the workload requirements of the application vary throughout time, the energy efficiency of the system can greatly benefit from dynamic voltage scaling. Unfortunately, due to physical limitations, aggressive voltage scaling can not be applied because of the lack of reliability. The failure probability of the conventional 6-Transistors SRAM cell increases considerably as the supply voltage is scaled down. To avoid reliability issues and improve energy efficiency we studied a novel hybrid memory architecture that guarantees correct operation in a single voltage domain and adapts the reliable memory portion to workload requirements substantially improving the overall energy efficiency.

Chapter 6 presents a solution to extend the baseline architecture to be resilient to variability. Indeed, Near-threshold Operation is plagued by greatly increased sensitivity to threshold voltage variations, potentially leading to timing failures. In this chapter we devise an architectural scheme to tolerate ambient temperature-induced variations, capable of statically (off- line) and dynamically (on-line) adapting the processor-to-memory latency without compromising execution correctness. Extensive tests in different scenarios validate the approach and different design trade-offs are presented showing the cost, performance and reliability gain compared to state-of-the-art static solutions.

Finally, **Chapter 7** summarizes the main research contributions of the present thesis work.

Chapter 2

VirtualSoC - Virtual Platform

2.1 Overview

Performance modeling plays a critical role in the design, evaluation, and development of computing architectures of any segment, ranging from embedded to high performance processors. Simulation has historically been the primary vehicle to carry out performance modeling, since it allows for easily creating and testing new designs several months before a physical prototype exists. Performance modeling and analysis are now integral to the design flow of modern computing systems, as it provides many significant advantages: i) accelerates time-to-market, by allowing the development of software before the actual hardware exists; ii) reduces development costs and risks, by allowing for testing new technology earlier in the design process; iii) allows for exhaustive design space exploration, by evaluating hundreds of simultaneous simulations in parallel.

High-end embedded processor vendors have definitely embraced the heterogeneous architecture template for their designs as it represents the most flexible and efficient design paradigm in the embedded computing domain. Parallel architecture and heterogeneity clearly provide a wider power/performance scaling, combining high performance and power efficient general-purpose cores along with massively parallel many-core-based accelerators. Examples and results of this evolution are AMD Fusion [15], NVidia Tegra [68] and Qualcomm Snap-

dragon [75]. Besides the complex hardware, generally these kinds of platforms host also an advanced software eco-system, composed by an operating system, several communication protocol stacks, and various computational demanding user applications.

Unfortunately, as processor architectures get more heterogeneous and complex, it becomes more and more difficult to develop simulators that are both fast and accurate. Cycle-accurate simulation tools can reach an accuracy error below 1-2%, but they typically run at a few millions of instructions per hour. The necessity to efficiently cope with the huge HW/SW design space provided by this target architecture makes clearly full-system simulator one of the most important design tools. Clearly, the use of slow simulation techniques is challenging especially in the context of full-system simulation. In order to perform an affordable processor design space exploration or software development for the target platform, trade-off accuracy for speed is thus necessary by implementing new virtual platforms that allow for faster simulation speed at the expense of modeling fewer micro-architecture details of not-critical hardware components (like the host processor domain), while keeping high-level of accuracy for the most critical hardware components (like the manycore accelerator domain).

We present in this chapter VirtualSoC, a new virtual platform prototyping framework targeting the full-system simulation of massively parallel heterogeneous system-on-chip composed by a general purpose processor (i.e. intended as platform coordinator and in charge of running an operating system) and a many-core hardware accelerator (i.e. used to speed-up the execution of computing intensive applications or parts of them). VirtualSoC exploits the speed and flexibility of QEMU, allowing the execution of a full-fledged Linux operating system, and the accuracy of a SystemC model for many-core-based accelerators.

The specific features of VirtualSoC are:

- Since it exploits QEMU for the host processor emulation, unmodified operating systems can be booted on VirtualSoC and the execution of unmodified ARM binaries of applications and existing libraries can be simulated on VirtualSoC.

- VirtualSoC enables accurate manycore-based accelerator simulation. We designed a full software stack allowing the programmer to exploit the hardware accelerator model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.

- The host processor (emulated by QEMU) and the SystemC accelerator model can run in an asynchronous way, where a non-blocking communication interface has been implemented enabling parallel execution between QEMU and SystemC environments.

- Beside the interface between QEMU and the SystemC model, we also implemented a synchronization protocol able to provide a good approximation of the global system time.

- VirtualSoC can be also used in stand-alone mode, where only the hardware accelerator is simulated, thus enabling accurate design space explorations.

This chapter focuses on the implementation details of VirtualSoC and evaluates the performance of various benchmarks and presents some example case studies using VirtualSoC.

The rest of the chapter is structured as follows: in Section 2.2 we provide an overview of related work, in Section 2.3 we present the target architecture, focusing on the many-core accelerator in Section 2.4. The implementation of the proposed platform is discussed in Section 2.5. Software simulation support is described in Section 2.6, finally experimental results and conclusions are presented in Sections 2.7 and 2.8.

2.2 Related work

The importance of full-system emulation is confirmed by the considerable amount of effort committed by both industry and research communities in developing such designing tools as more efficient as possible. We can cite several examples, like

Bochs [52], Simics [56], Mambo [12], Parallel Embra [50], PTLsim [94], AMD SimNow [8], OVPSim [86] and SoCLib [61].

QEMU [9] is one of the most widely used open-source emulation platform. QEMU supports cross-platform emulation and exploits binary translation for emulating the target system. Taking advantage of the benefits of binary translation, QEMU is very efficient and functionally correct, however it does not to provide any accurate information about hardware execution time. In [37] authors have implemented program instrumentation capabilities to QEMU for user application program analysis. This work has only been done for the user mode of QEMU and it cannot be exploited for system performance measurements (e.g. device driver). Moreover, profiling based on program instrumentation can heavily change the execution flow of the program itself, leading to behaviors which will never happen when executing the program in the native fashion. Authors in [62] have instead presented pQEMU, which simulates the timing of instruction executions and memory latencies. Instruction execution timings are simulated using instruction classification and weight coefficients, while memory latency is simulated using a set-associative cache and TLB simulator. This kind of approach can lead to a significant overhead due to the different simulation stages (i.e. cache simulation, TLB simulation), and even in this case the proposed framework can only run user-level applications without the support of an operating system.

QEMU lacks also of any accurate co-processors simulation capabilities. Authors in [76] interfaced QEMU with a many-core co-processor simulator running on an nVidia GPGPU [72]. Despite the co-processor simulator described in [72] is able to simulate thousands of computing units connected through a NoC, it runs at a high level of abstraction and does not provide precise measurements from the simulated architecture. Moreover authors do not address the problem of timing synchronization between QEMU and the co-processor simulation.

Other works have been mainly concentrated on enabling either cycle accurate instruction set simulators for the general purpose processor part or SystemC-based simple peripherals, without considering complex many-core-based accelerators [34].

When interfacing QEMU with the SystemC framework, several implementation aspects and decisions need to be accurately taken into account, since development choices can limit and constraint the performance of the overall emulation environment. The optimal implementation should not possibly affect efficiency, flexibility and scalability.

Establishing the communication between QEMU and SystemC simulator through inter-process communication socket is another approach. Authors in [74] use such facility between a new component of QEMU, named QEMU-SystemC Wrapper, and a modified version of the SystemC simulation kernel. The exchanged messages have the purpose not only to transmit data and interrupt signals but also to keep the simulation time synchronized between the simulation kernels. However using heavy processes does not allow fast and efficient memory sharing, which in this case can be achieved only using shared memory segments. Moreover, Unix Domain Sockets are less efficient, in terms of performance and flexibility, than direct communication between threads.

QEMU-SystemC [65] allows devices to be inserted into specific addresses of QEMU and communicates by means of the PCI/AMBA bus interface. However, QEMU-SystemC does not provide the accurate synchronization information that can be valuable to the hardware designers. [53] integrates QEMU with a SystemC-based simulation development environment, to provide a system-level development framework for high performance system accelerators. However, this approach is based on socket communication, which strongly limits its performance and flexibility. Authors in [93] suggested an approach based on threads since context switches between threads are generally much faster than between processes. However, communication among QEMU and SystemC uses a unidirectional FIFO, limiting the interaction between QEMU and the SystemC model.

We present in this chapter our new emulation framework based on QEMU and SystemC which overcomes these issues. We chose QEMU amongst all simulators cited (e.g. OVPSim [86], SoCLib [61]) because it is fast, open-source and also very flexible enabling its extension with a moderate effort. Our approach is based on thread parallelization and memory sharing to obtain a complete heterogeneous

SoC emulation platform. In our implementation the target processor and the SystemC model can run in an asynchronous way, where non-blocking communication is implemented through the use of shared memory between threads. Beside the interface between QEMU and a SystemC model, we also present a lightweight implementation of a synchronization protocol able to provide a good approximation of a global system time. Moreover, we designed a full SW stack allowing the programmer to exploit the HW model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.

2.3 Target Architecture

Modern embedded SoCs are moving toward systems composed by a general purpose multi-core processor accompanied by a more energy efficient and powerful many-core accelerator (e.g. GPU). In these kinds of systems the general purpose processor is intended as a coordinator and is in charge of running an operating system, while the many-core accelerator is used to speed up the execution of computing intensive applications or parts of them. Despite their great computing power, accelerators are not able to run an operating system due to the lack of all needed surrounding devices and to the simplicity of their micro-architectural design. The architecture targeted by this work (shown in Figure 2.1) is representative of the above mentioned platforms and composed by a many-core accelerator and an ARM-based processor.

The ARM processor is emulated by QEMU which models an ARM926 processor, featuring an ARMv5 ISA, and interfaced with a group of peripherals needed to run a full-fledged operating system (ARM Versatile Express baseboard). The many-core accelerator is a SystemC cycle-accurate MPSoC simulator. The ARM processor and the accelerator share the main memory, used as communication medium between the two. The accelerator target architecture features a configurable number of simple RISC cores, with private or shared I-cache architecture, all sharing a Tightly Coupled Data Memory (TCDM) accessible via a local inter-

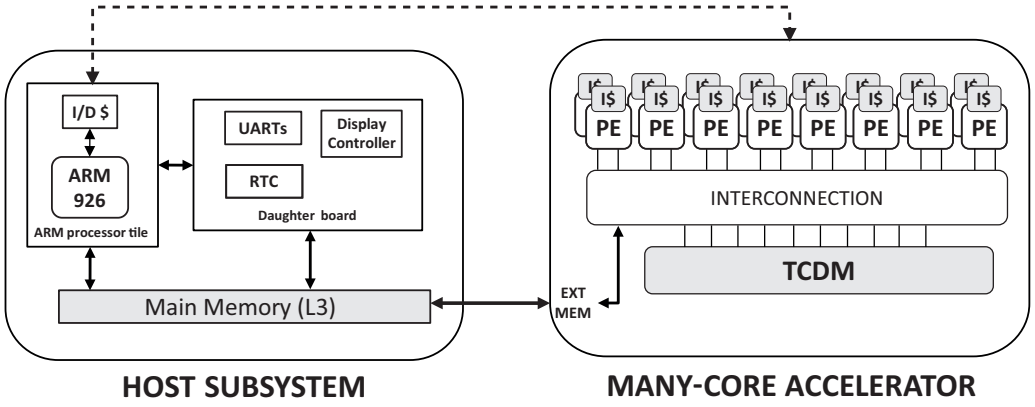

Figure 2.1: Target simulated architecture

connection. The state-of-the-art programming model for this kind of systems is very similar to the one proposed by OpenCL [48]: a master application is running on the host processor which, when encounters a data or task parallel section, offloads the computation to the accelerator. The master processor is in charge also of transferring input and output data.

2.4 Many-core Accelerator

The proposed target many-core accelerator template can be seen as a cluster of cores connected via a local and fast interconnect to the memory subsystem. The following sub-sections describe the building blocks of such cluster, shown in Figure 2.2.

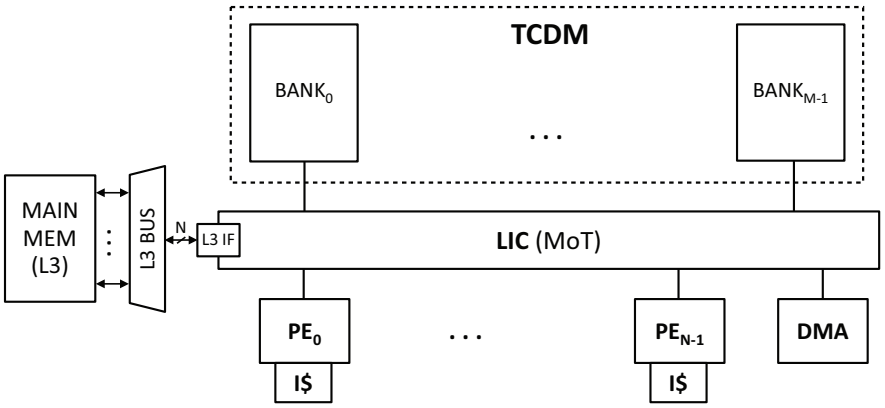

Figure 2.2: Many-core accelerator

2.4.1 Architectural components

In this section we describe the most relevant architectural elements of the many-core accelerator that constitutes the baseline architecture considered throughout the present work.

Processing Elements

Our shared L1 cluster consists of a configurable number of 32-bit ARMv6 processor (ARM11 family [7]). There are several ARMv6 instruction set simulator already available, Skyeye [45], SoClib [36] and SimSoc [39] are just a few representative examples. We chose the one in [39] as our base ISS. To obtain timing accuracy, after modifying its internal behavior to perform concurrent load/store and instruction fetch (Harvard Architecture), we wrapped the ARMv6 Instruction Set Simulator (ISS) in a SystemC module.

L1 Instruction Cache Module

The Instruction Cache Module has a core-side interface for instruction fetches and an external memory interface for refill. The inner structure consists of the actual memory (TAG + DATA) and the cache controller logic managing the requests. The module is configurable in its total size, associativity, line size and replacement policy (FIFO, LRU, random).

Logarithmic interconnect

The logarithmic interconnect module has been modeled, from a behavioral standpoint, as a parametric, Mesh-of-Trees (MoT) interconnection network to support high-performance communication between processors and memories in L1-coupled processor clusters resembling the hardware module presented in [77], shown in Figure 2.3.

The module is intended to connect processing elements to a multi-banked memory on both data and instruction side and is parametric in both master (cores) and slave (banks) ports. Data routing is based on address decoding: a first-stage checks if the requested address falls within the intra-cluster memory

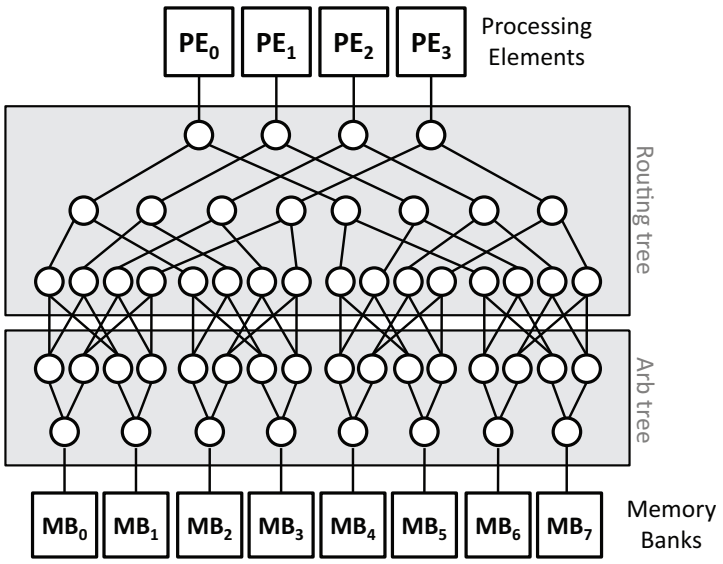

Figure 2.3: Mesh of trees 4x8 (banking factor of 2)

address range or has to be directed off-cluster. To increase module flexibility this stage is optional, enabling explicit main memory (L3[1]) data access on the data side while, on the instruction side, can be bypassed letting the cache controller take care of L3 memory accesses for lines refill. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The last $log_2(interleaving\ size)$ bits of the address determine the destination. The crossing latency consists of one clock cycle. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates access and a higher number of cycles is needed depending on the number of conflicting requests, with no latency in between. In case of no banking conflicts data routing is done in parallel for each core, thus enabling a sustainable full bandwidth for processors-memories communication. To reduce memory access time and increase shared memory throughput, read broadcast has been implemented and no extra cycles are needed when broadcast occurs.

L1 Tightly Coupled Data Memory

On the data side, a multi-ported, multi-banked, Tightly Coupled Data Memory (TCDM) is directly connected to the logarithmic interconnect. The number of memory ports is equal to the number of banks to have concurrent access to different memory locations. Once a read or write requests is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for conflict-free TCDM access. As already mentioned above, if conflicts occur there is no extra latency between pending requests, once a given bank is active, it responds with no wait cycles. Banking factor (i.e. ratio between number of banks and cores) can be configured to explore how this affects banking conflicts.

[1] in the naming convention used in this work, when referring to L3 memory we are considering an off-chip memory. In modern embedded many-core systems such memory consists of DRAM/LPDDR memories.

Synchronization

To coordinate and synchronize cores execution, we modeled two different synchronization mechanisms. The first one consists of *HW semaphores* mapped in a small subset of the TCDM address range. They consist of a series of registers, accessible through the data logarithmic interconnect as a generic slave, associating a single register to a shared data structure in TCDM. By using a mechanism such as a hardware *test&set*, we are able to coordinate access: if reading returns '0', the resource is free and the semaphore automatically locks it, if it returns a different value, typically '1', access is not granted. This module enables both single and two-phases synchronization barriers, easily written at the software level. Theoretically all cores can be resumed at the same time (reading broadcast the value of the semaphore), but there is no guarantee that this happens because of execution misalignment. To get tight execution alignment, we developed two *fast synchronization primitives* based on a *HW Synchronization Handler Module* (SHM). This device acts as an extra slave device of the logarithmic interconnect and has a number of hardware registers equal to the number of cores, where each register is mapped in a specific address range. When a write operation is issued to a given register, a synchronization signal is raised to the corresponding core suspending its execution after one cycle, when the synchronization signal is lowered the execution is resumed. The SHM is programmable in different ways from the software level via APIs. Writing to the OP_MODE register, different synchronization mechanisms can be enabled: if operating in SYNC_MODE, synchronization signals are lowered when all cores have executed the `sync()` API (writing to their respective register, increasing an HW counter inside the SHM), obtaining a cycle-accurate execution alignment. When operating in TWO_PH_MODE, a simple state machine inside the SHM distinguishes cores behavior between master and slaves enabling a two-phases barrier. When the master reaches a `master_wait_barrier()` endsmall primitive, it is suspended until all slaves have reached the `slave_enter_barrier()`. After that, the master is awakened and is the only core executing until the `master_release_barrier()` primitive is reached, reactivating all slaves exactly in the same clock cycle. These

APIs and the underlying HW mechanism offered by the SHM are fundamental for the OpenMP library described in Section 3.4.2.

Instruction Cache Architecture

The L1 Instruction Cache basic block has a core-side interface for instruction fetches and an external memory interface for refill. The inner structure consists of the actual memory and the cache controller logic managing the requests. The module is configurable in its total size, associativity, line size and replacement policy (FIFO, LRU, random). The basic block can be used to build different Instruction Cache architectures:

- *Private Instruction Cache*: every processing element has its private I-cache, each one with a separate cache line refill path to main memory leading to high contention on external L3 memory.
- *Shared Instruction Cache*: there is no difference between the private architecture in the data side except for the reduced contention L3 memory (line refill path is unique in this architecture). Shared cache inner structure is made of a configurable number of banks, a centralized logic to manage requests and a slightly modified version of the logarithmic interconnect described above: it connects processors to the shared memory banks operating line interleaving (1 line consists of 4 words). A round robin scheduling guarantees fair access to the banks. In case of two or more processors requesting the same instruction, they are served in broadcast not affecting hit latency. In case of concurrent instruction miss from two or more banks, a simple bus handles line refills in round robin towards the L3 bus.

2.5 Host-Accelerator Interface

In this section we describe the QEMU-based host side of VirtualSoC (*VSoC-Host*), as well as the many-core accelerator side (*VSoC-Acc*).

Parallel Execution

In a real heterogeneous SoC host processor and accelerator can execute in an asynchronous parallel fashion, and exchange data using non-blocking communication primitives. Usually the host processor, while running an application, offloads asynchronously a parallel job to the accelerator and goes ahead with its execution (Figure 2.4). Only when needed the host processor synchronizes with the execution of the accelerator, to check the results of the computation.

In our virtual platform the host processor system and the accelerator can run in parallel, with VSoC-Host and VSoC-Acc running on different threads: when the thread of VSoC-Acc starts its execution triggers the SystemC simulation. It is important to highlight that the VSoC-Acc SystemC simulation starts immediately during VSoC-Host startup, and the accelerator starts executing the binary of a firmware (until the shutdown) in which all cores are waiting for a job to execute.

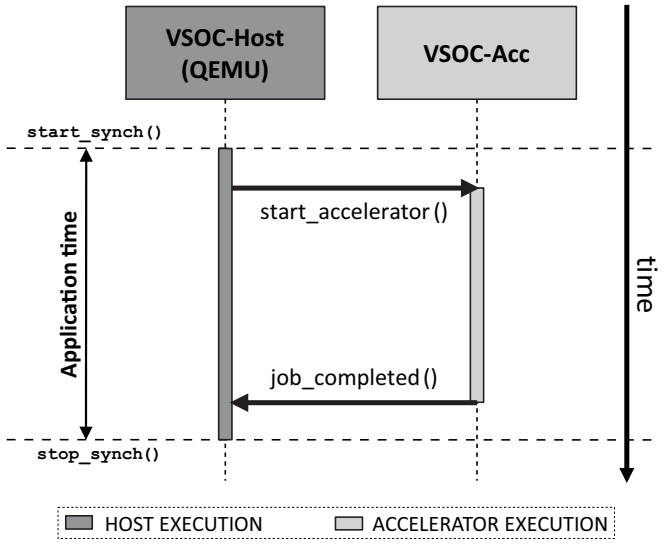

Figure 2.4: Execution model

Time Synchronization Mechanism

VSoC-Host and VSoC-Acc run independently in parallel with a different notion of time. The lack of a common time measure leads to only functional simulation, without the possibility of profiling applications performance even in a qualitative way. Application developers often need to understand how much time, over the total application time, is spent on the host processor or on the accelerator. Also, without a global simulation time it is not possible to appreciate execution time speedups due to the exploitation of the many-core accelerator.To manage the time synchronization between the two environments, it is necessary that both VSoC-Host and VSoC-Acc have a time measurement system. VSoC-Host does not natively provide this kind of mechanisms, so we instrumented it to implement a clock cycle count, based on instructions executed and memory accesses performed. On the contrary for VSoC-Acc there is no need for modifications because it is possible to exploit the SystemC time. The synchronization mechanism used in our platform is based on a threshold protocol acting on simulated time: at fixed synchronization points the simulated time of VSoC-Host and VSoC-Acc is compared. If the difference is greater than the threshold, the entity with the greater simulated time is stopped until the gap is filled.

At fixed synchronization points, cycles count from VSoC-Host (C_H) and VSoC-Acc (C_A) are multiplied by the respective clock period (P_H and P_A) and compared. Given a time threshold h if $|C_A * P_A - C_A * P_A| > h$, one of the two systems is forward in the future in respect to the other and its execution is stopped until $|C_H * p_H - C_A * P_A| > 0$. The Global simulation time is always the greater of the two. It is intuitive to note that the proposed mechanism slows down the simulation speed, due to synchronization points and depending on the difference of simulation speed between the two ecosystems. To avoid unnecessary slowdown, we provide an interface to activate and de-activate the time synchronization when it is not needed (e.g. functional simulation).

2.6 Simulation Software Support

In this section we provide a description of the software stack provided with the simulator to allow the programmer to fully exploit the accelerator from within the host Linux system, and to write parallel code to be accelerated.

Linux Driver

In order to build a full system simulation environment we mapped VSoC-Acc as a device in the device file system of the guest Linux environment running on top of VSoC-Host. A device node `/dev/vsoc` has been created, and as all Linux devices it is interfaced to the operating system using a Linux driver. The driver is in charge of mapping the shared memory region into the kernel I/O space. This region is not managed under virtual memory because the accelerator can deal only with physical addresses, as a consequence all buffers must be allocated contiguously (done by the Linux driver). The driver provides all basic functions to interact with the device.

Host Side User-Space Library

To simplify the job of the programmer we have designed a user level library, which provides a set of APIs that rely on the Linux driver functions. Through this library the programmer is able to fully control the accelerator from the host Linux system. It is possible for example to offload a binary, or to check the status of the current executing job (e.g. checking if it has finished).

Accelerator Side Software Support

The basic manner we provide to write applications for the accelerator is to directly call from the program a set of low-level functions implemented as a user library, called `appsupport`. `appsupport` provides basic services for memory management, core ID resolution, synchronization. To further simplify programming and raise the level of abstraction we also support a fully-compliant OpenMP v3.0 programming model, with associated compiler and runtime library.

2.7 Evaluation

In this section two use cases of the simulation platform are presented. We will show how the proposed virtual platform can be exploited for both software verification or design space exploration.

2.7.1 Experimental Setup

Table 2.1 summarizes the experimental setup of the virtual platform used for all benchmarks discussed. We chose as ARM core clock frequency of 1GHz, even if the ARM modeled by QEMU works at up to 500MHz, to resemble a state of the art ARM processor performance. The frequency would only affect results in terms of global values, all considerations done in this section remain valid even if the ARM core clock frequency is changed.

Table 2.1: Experimental Setup

PARAMETER	VALUE
PLATFORM	
L3 latency	200 ns
L3 size	256 MB
ACCELERATOR	
PE	16
frequency	250 MHz
L1 I\$ size	16 KB
t_{hit}	= 1 cycle
t_{miss}	$\geq$ 50 cycles
TCDM banks	16
TCDM size	256 KB
HOST	
ARM Core clock frequency	1GHz
Guest OS	Debian for ARM (Linux 2.6.32)

2.7.2 VirtualSoC Use Cases

Full System Simulation

As first use case of the simulator we propose the profiling of an application involving both the ARM host and the many-core accelerator. In this example we want to measure the speedup achievable when accelerating a set of algorithms onto the many-core accelerator. The algorithms chosen are: *Matrix Multiplication*, *RGBtoHPG* color conversion, and *Image Rotation* algorithm. All the benchmarks follow a common scheme: the computation starts from the ARM host which in turn will offload a parallel task, one of the algorithms, to the accelerator. Then we compare simulated time obtained varying the number of cores present in the accelerator, with the time taken to run each benchmark on the ARM processor only (i.e. no acceleration).

Figure 2.5 shows the results of this experiment. Using the accelerator with 8 cores we can see a speedup of $\approx 3\times$ times for the matrix multiplication, $\approx 3\times$ for the rotate benchmark and $\approx 5\times$ for the RGBtoHPG benchmark. When running with 16 cores we can appreciate an almost double execution speedup for all the proposed benchmarks.

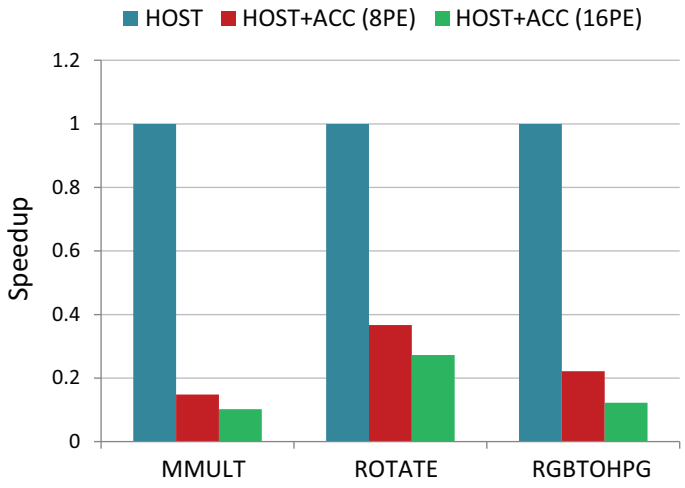

Figure 2.5: Speedup due to accelerator exploitation

Standlone Accelerator Simulation

In this section we show an example of stand-alone accelerator analysis by using two real applications, namely a JPEG decoder and a Scale Invariant Feature Transform (SIFT), a widely adopted algorithm in the domain of image recognition. Our analysis will as first evaluate the effects of L3 latency over the execution time of each benchmark. In a second experiment we evaluate the instruction cache usage made by each application in terms of hit rate and average hit time. Figure 2.6 shows the execution time when varying the L3 latency, and as expected the time increases when increasing the external memory access latency.

The instruction cache utilization is shown in Figure 2.7, depending on the application parallelization scheme the hit rate changes as well as the average hit time. The JPEG benchmark has been implemented in two different schemes: a data parallel implementation and a pipelined implementation. Results show that the data parallel version is more efficient in terms of cache hit rate and globally in terms of execution time. A deeper analysis will be the object of the research work presented in the next chapter.

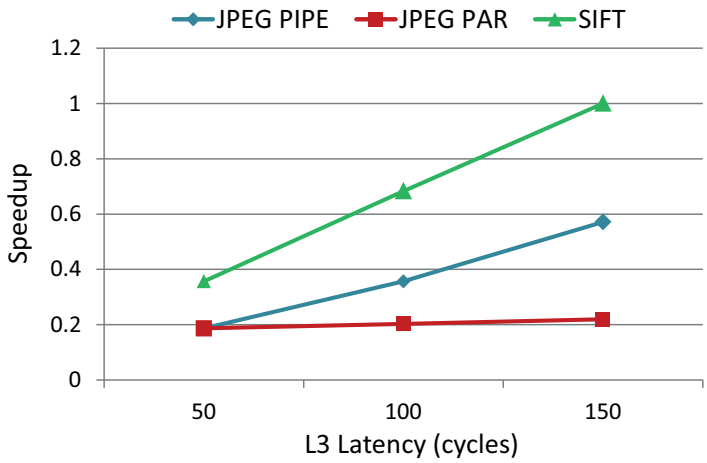

Figure 2.6: Benchmarks execution for varying L3 access latency (shared I-cache architecture)

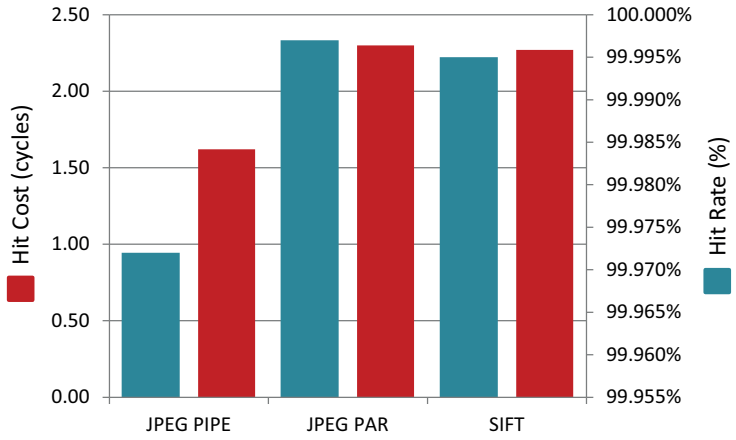

Figure 2.7: Benchmarks hit rate and average hit cost

2.8 Conclusions

VirtualSoC leverages QEMU to model a ARMv6 host processor, capable of running a full-fledged Linux operating system. The many-core accelerator is modeled with higher accuracy using SystemC. We extended this combined simulation technology with a mechanism to allow for gathering timing information that is kept consistent over the two computational sub-blocks. A set of experiments over a number of representative benchmarks demonstrate the functionality, flexibility and efficiency of the proposed approach.

Chapter 3

Instruction Caching Strategies

3.1 Overview

To keep the pace of Moore's law, several Chip-Multiprocessors (CMP) platforms are embracing the many-core paradigm, where a large number of simple cores are integrated onto the same die. Current examples of many-cores include GP-GPUs such as NVIDIA *Fermi* [21], the *HyperCore Architecture Line (HAL)* [2] processors from Plurality Ltd., ST Microelectronics *Platform 2012* [11] or Adapteva [5]. While there is renewed interest in Single Instruction Multiple Data (SIMD) computing, thanks to the success of GP-GPU architectures, strict instruction scheduling policies enforced in current GP-GPUs are being relaxed in the most recent many-core designs to exploit data parallelism in a flexible way. Single Program Multiple Data (SPMD) parallelism can thus be efficiently implemented in these designs, where processors are not bound to execute the same instruction stream in parallel to achieve peak performance.

All of the cited architectures share a few common traits: their fundamental computing tile is a tightly coupled cluster with a shared multibanked L1 memory for fast data access and a fairly large number of simple cores, with ≈ 1 Instruction Per Cycle (IPC) per core. Key to providing I-fetch bandwidth for a cluster is an effective instruction cache architecture design. Due to the lack of sophisticated hardware support to hide L2/L3 memory latency (e.g. prefetch buffers), the

simple processors embedded in many-cores may indeed experience prolonged stalls on long-latency I-fetch.

The main contribution of this work is the analysis and comparison of the two main architectures for instruction caching targeting tightly coupled CMP clusters: (i) private instruction caches per core and (ii) shared instruction cache per cluster. We developed a cycle-accurate model of the target cluster with the two cache organizations, and with several configurable architectural parameters for exploration. The private template achieves higher speed, due to its simpler design, but the smaller L1 memory space seen by each core may induce a lower hit ratio. Moreover, the co-existence of multiple program copies in the system may require more bandwidth to main memory in case of multiple concurrent misses. In contrast, the shared template can offer a lower miss ratio and better memory utilization (less copies) at the cost of increased hardware complexity and thus lower speed.

To efficiently analyze and assess pros and cons of the two architectures we also developed a programming environment targeted at efficient data-parallel computing and based on the popular OpenMP programming model. The compilation toolchain and runtime support have been tailored to the target cluster, thus allowing effective benchmarking. We first characterize the two architectural templates by using synthetic microbenchmarks, useful to stress specific corner cases and to assess the best and worst operating conditions for the two cache architectures. Then we further validate the two approaches with several kernels from representative applications from the image processing and recognition domain, parallelized with OpenMP.

The rest of the chapter is organized as follows: In Sec. 3.2 we discuss related work to ours. The target tightly-coupled cluster and the two cache architectures are described in Sec. 3.3, while the programming framework, compiler and runtime support are discussed in Sec. 3.4. We describe our experimental setup and results in Sec. 3.5, while Sec. 3.6 concludes the chapter.

3.2 Related Work

The organization of memory hierarchy is one of the most important and critical phases in architecture design. This aspect has become more relevant with the advent of modern many-core platforms. Dealing with massively parallel systems, instruction caching plays a fundamental role since it must provide the required bandwidth to all cores and software tasks, complying with tight constraints in terms of size and complexity [80].

The Fermi-based General Purpose Graphic Processing Units (GPGPU) comprises hundreds of Streaming Processors (SP) organized in groups of Streaming Multiprocessors (SM) [21]. The numbers of SMs and SPs per device vary by device. GPGPUs employ massively multi-threading in order to hide the latency of main memory. The GPU achieves indeed efficiency by splitting application workload into multiple groups of threads (called warps) and multiplexing many of them onto the same SM. When a warp that is scheduled attempts to execute an instruction whose operands are not ready (due to an incomplete memory load, for example), the SM switches context to another warp that is ready to execute, thereby hiding the latency of slow operations such as memory loads. All the SPs in an SM execute their threads in lock-step, according to the order of instructions issued by the per-SM instruction unit. SPs within the same SM share indeed one single instruction cache [91].

Plurality's HyperCore Architecture Line (HAL) family includes 16 to 256 32-bit RISC cores, a shared memory architecture, and a hardware-based scheduler that supports a task-oriented programming model [2]. HAL cores are compact 32-bit RISC cores, which execute a subset of the SPARC v8 instruction set with extensions. The memory system is composed by a single shared memory which operates also as instruction cache. The shared memory holds indeed program, data, stack, and dynamically allocated memory. Each core has two memory ports: an instruction port that can only read from memory, and a data port that can either read or write to memory. Both ports can operate simultaneously, thus allowing an instruction fetch and a data access by each individual core at each clock cycle. The processors do not have any private cache or memory, avoiding

coherency problems. However, conflicting accesses cannot be avoided causing latency increasing for not-served requests [2].

STMicroelectronics Platform 2012 (P2012) is a high-performance architecture for computationally demanding image understanding and augmented reality applications [11]. P2012 architecture is composed by several processing clusters interconnected by a Network on Chip (NoC). Each computing cluster features a shared instruction cache memory.

All of the cited platforms have adopted different instruction cache architectures, meaning that there is still not a dominant paradigm for instruction caching in the manycore scenario. Clearly, a detailed design space exploration and analysis are needed to evaluate how micro-architectural differences in L1 instruction cache architectures may affect the overall system behavior and IPC.

3.3 Target cluster architecture

The building blocks of the baseline architecture considered here are the one presented in Section 2.4. Moreover, to help system designers to compare different L1 instruction cache architectures, we have developed a flexible instruction cache architecture system. The proposed templates, written in SystemC [3], can be used either in stand-alone mode or plugged into any virtual platform, we integrated them in an accurate virtual platform environment specifically designed for embedded MPSoC design space explorations [14]. Our enhanced virtual platform is highly modular and capable of simulating at cycle-accurate level an entire shared L1 cluster including cores, instruction caches, shared tightly coupled data memory, external (L3) memories and system interconnections.

3.3.1 Private Instruction Cache Architecture

All the previously described architectural elements are combined together to form the private instruction cache architecture as shown in Figure 3.1.

The cluster is made of 16 ARMv6 cores, each one has its own private instruction cache with separate line refill paths while the L1 data memory is shared

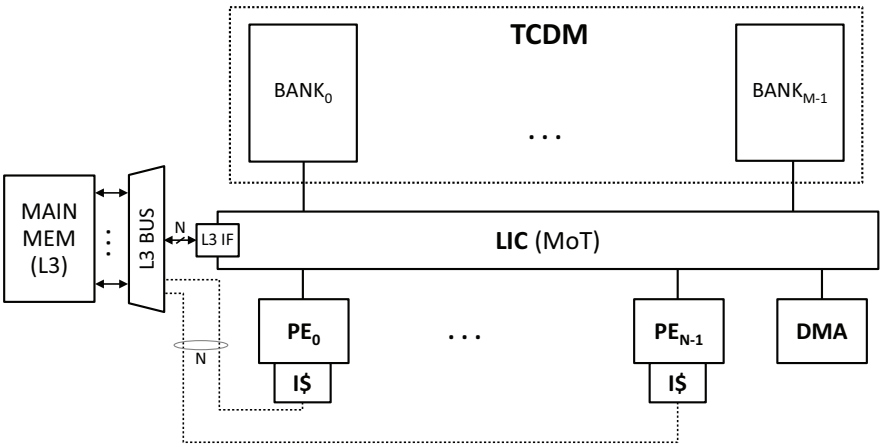

Figure 3.1: Cluster with private L1 instruction caches

among them. An optional DMA engine can be used to carry out L3 to TCDM data transfers. Access to the off-cluster L3 memory is coordinated by the L3 BUS, requests are served in a round-robin fashion. On the data side all cores are able to perform access to TCDM, L3 memory and eventually to HW semaphores or SHM. The logarithmic interconnect is responsible of data routing based on address ranges as already described in the previous section. Default configuration for the private instruction cache architecture and relevant timings are reported in Table 3.1.

Table 3.1: Default private cache architecture parameters and timings

PARAMETER	VALUE
ARM v6 cores	16
$I\$_i$ size	1 KB
$I\$_i$ line	4 words
t_{hit}	= 1 cycle
t_{miss}	$\geq$ 59 cycles
TCDM banks	16
TCDM size	256 KB
L3 latency	50 cycles
L3 size	256 MB

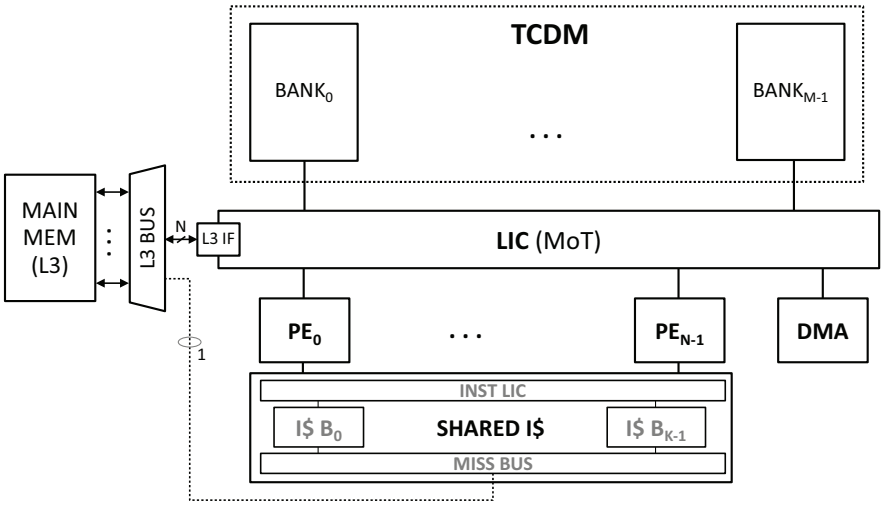

Figure 3.2: Cluster with shared L1 instruction cache

3.3.2 Shared Instruction Cache Architecture

Shared instruction cache architecture is shown in Figure 3.2. From the data side there is no difference between the private architecture except for the reduced contention for data requests to L3 memory (line refill path is unique in this architecture).

Shared cache inner structure is represented in Figure 3.3. A slightly modified version of the logarithmic interconnect described in Section 2.4.1 (the first stage

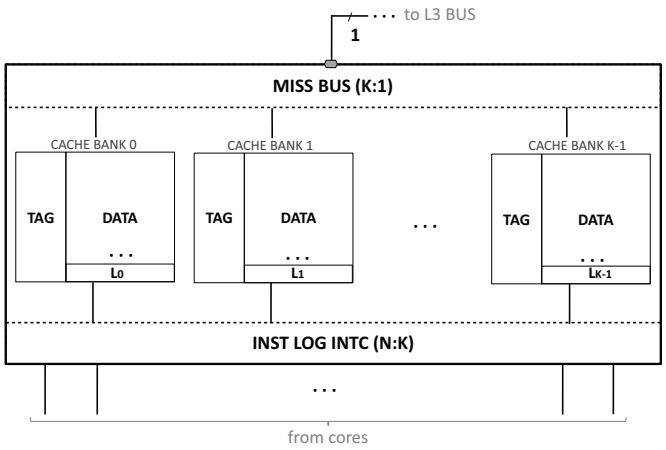

Figure 3.3: Shared instruction cache architecture

of address deconding is disabled) connects processors to the shared memory banks operating line interleaving (1 line consists of 4 words). A round robin scheduling guarantees fair access to the banks. In case of two or more processors requesting the same instruction, they are served in broadcast not affecting hit latency. In case of concurrent instruction miss from two or more banks, a simple MISS BUS handles line refills in round robin towards the L3 BUS. Table 3.2 summarizes the main configuration parameters for the shared cache cluster.

Table 3.2: Shared cache architecture parameters and timings

PARAMETER	VALUE
I\$ size	16 KB
I\$ line	4 words
t_{hit}	$\geq$ 1 cycle
t_{miss}	variable

3.4 Software Infrastructure

In this section we briefly describe the software infrastructure: first compiler and linking strategies used to compile and allocate all the data needed for the execution of all benchmarks. In the second part we will introduce our custom implementation of the OpenMP library, developed to run on the proposed target architectures.

3.4.1 Compiler and Linker

Before describing compiling and linking strategies applied to our benchmarks, it is of primary importance to introduce the memory map seen by all processors in the architecture. Figure 3.4 shows the global memory map of one cluster, in which it is possible to distinguish two memory regions: the L3 MEMORY REGION and the TCDM MEMORY REGION with nominal sizes of 256 MB and 256 KB respectively. The first is the off-chip (L3) memory used to store the executable of the applications, and data too big to be stored in the on-chip data memory. The TCDM region, mapping the shared data scratchpad, is in turn

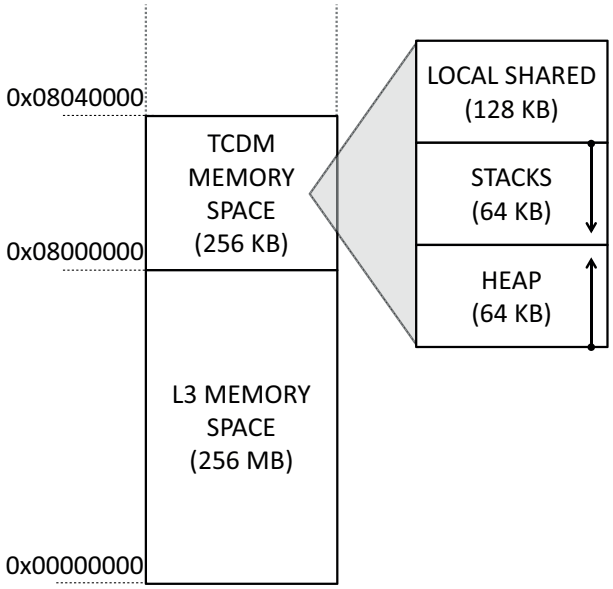

Figure 3.4: Cluster global memory map

divided in three sub-regions: LOCAL_SHARED, STACK and HEAP.

The LOCAL_SHARED region is intended to maintain variables of static size (known at compile time) explicitly defined to be stored in TCDM. To force the allocation of a variable in the on-chip data memory we combined the use of a linker script and gcc attributes. We defined a new section in the ARM binary, namely `.local_shared`, used to contain variables to be stored in this region of the memory map as shown in Listing 3.1.

```
MEMORY {
    GLOBAL_SHARED : org = 0x0, l = 256M
    LOCAL_SHARED  : org = 0x08020000, l = 128K
}

SECTIONS {
    ...
    .global_shared : {
            *(.global_shared)
    }>GLOBAL_SHARED
    .local_shared : {
            *(.local_shared)
    }>LOCAL_SHARED
}
```

Listing 3.1: Linker script memory layout and output sections

The STACK region is defined to maintain the stack of all 16 processors, with a nominal stack size of 4K assigned to each of them. Each processor calculates its own stack top at simulation startup using a combination of linker script and an assembly boot routine. In the linker script side the symbol `__stack_start` is defined, pointing to the top of the STACK region. In the boot routine each core, using the `__stack_start` symbol, computes its stack top according to its processor id (See Listing 3.2).

```
// get processor's id
mov r10, #0x7f000000
ldr r11, [r10, #0x20]
sub r11, r11, #1

// spacing stack pointers
// stride = 0x1000 (4KB)
mov r9, #0x1000
mul r10, r11, r9
sub sp, sp, r10
```

Listing 3.2: Stack pointer assignment in Boot Sequence

Finally, the HEAP region is used for dynamically allocated structures. The allocation is allowed through the `shmalloc()` function, provided by VirtualSoC's applications support (appsupport).

3.4.2 Custom OpenMP Library

To parallelize our benchmarks we used a custom implementation of the OpenMP APIs for parallel programming, adapted to run on our VirtualSoC based architecture. The OpenMP parallel programming paradigm considered is based on two different parallel constructs:

- `#pragma omp parallel`
- `#pragma omp sections`

The first allows the exploitation of SIMD or SPMD parallelism, the iterations of the for cycle are divided in chunks and assigned to the available cores. The second describes task parallel sections of a program, each core can execute a different portion of code therefore a different task. To tailor these two constructs to the target architecture it is necessary to consider that our software infrastructure has no Operating System. In our implementation all cores execute the same binary image as a single process running on each processor, and the work performed is differentiated according to the processor's id. The Master-Slave mechanism

on which OpenMP is based is realized using the two-phase barriers described in Section 2.4.1. We also modified the compiler (`arm-elf-gcc 4.3`) to transform OpenMP annotations in a correct binary form for the VirtualSoC architecture. The compiler is in charge of creating all the structures needed to run a certain application and to differentiate the work to be performed by single processors by means of appsupport functions.

Our OpenMP runtime has a thin software layer based on a set of shared data structures used by the processors to synchronize, share data and control the different parallel regions of the applications. All these structures are stored in TCDM memory using both statically (LOCAL_SHARED) and dynamically allocated structures (`shmalloc()`), some structures are protected by a `lock` which is implemented via the hardware semaphores as described in Section 2.4.1.

3.5 Experimental Results

As already outlined in previous sections, we considered a cluster made of 16 ARMv6 cores connected through a low latency logarithmic interconnect to a multiported, multibanked 256 KB TCDM memory. On the instruction side, private and shared architectures differ in the cache architecture. An off-cluster (L3) 256 MB memory is accessible through the data logarithmic interconnect or through the line refill path. Our investigations focus on varying the total instruction cache size, and hereafter the L3 memory latency.

3.5.1 Microbenchmarks

In this section we present the results of three synthetic benchmarks intended to characterize both architectures and to highlight interesting behaviors. The synthetic benchmarks were written using ARM Assembly Language [32] in order to have complete control of the software running on top of the architectural templates. They consist of a set of iterated ALU or MEMORY instructions performed to highlight a specific behavior. All the synthetic benchmarks share the common structure shown in Listing 3.3 below.

```
        mov r6, N_LOOP
        mov r5, #0
_loop:  cmp r5, r6
        blt _body
        b _end
_body:  ...
        add r5, r5, #1
        b _loop
 _end:  ...
```

Listing 3.3: Synthetic benchmarks structure

The performance metrics considered here are the *Cluster IPC* ($\overline{IPC}$, $0 < \overline{IPC} \leq 16$) and its average value, computed as

$$\overline{IPC} = \sum_{i=1}^{16} \frac{N_i}{T_{cl}} \tag{3.1}$$

where N_i represents the number of instructions executed by PE_i and $T_{cl} = t_{last}^{stop} - t_{first}^{start}$, computed as timestamps difference of the last core exiting the kernel region (t_{last}^{stop}) and first entering (t_{first}^{start}).

Cold misses

The body of this benchmark consists of only ALU operations (i.e. `mov r0, r0`) leading to a theoretical $\overline{IPC} = 16$ (and average IPC = 1) for both architectures. The plot in Figure 3.5 shows on the y-axis the cluster average IPC while x-axis reports how many times the loop is executed.

Increasing `N_LOOP` both architectures tend to the theoretical value, but the private architecture starts from a lower IPC due to the heavy impact of cold misses serialization (16 cores contending for L3 access).

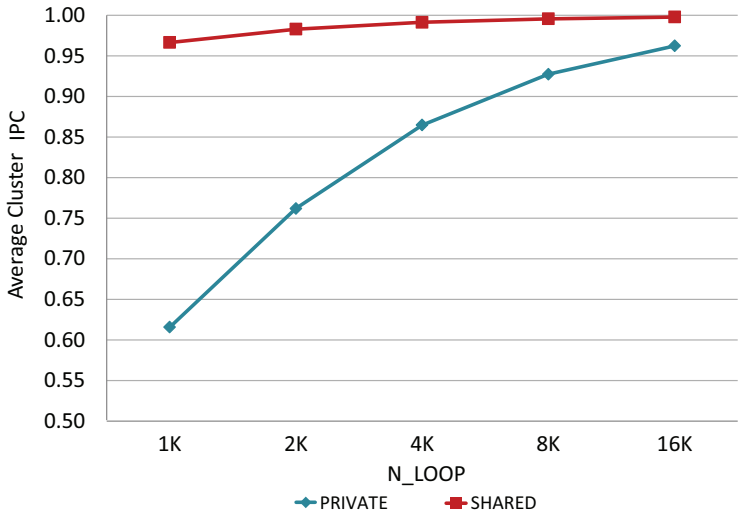

Figure 3.5: Private vs. Shared architectures IPC with only ALU operations

Conlict free TCDM accesses

This benchmark adds the effect of TCDM access. As already mentioned before, in case of conflict free access, TCDM latency is two cycles leading to a single cycle stall between two consecutive instruction fetches. The loop is iterated a fixed number of times (4K in order to lower cold misses effect) and has a variable number of memory operations inside its body. We are considering a banking factor of 1, allowing every core to access a different bank without conflicts. The plot in Figure 3.6 shows on the y-axis the average cluster IPC while on x-axis varies the percentage of memory instructions over the number of instructions of the loop. Both architectures are affected in the same way, with IPC tending to the asymptotic value value of $\frac{1}{2}$ and cluster IPC respectively to 8 because of the absence of any conflict leading to misalignment. In fact, a program consisting of only ALU (1 cycle) or MEMORY (2 cycles for TCDM access) operations gives a per-core IPC equal to:

$$IPC = \frac{N_{alu} + N_{mem}}{1 \cdot N_{alu} + 2 \cdot N_{mem}} \tag{3.2}$$

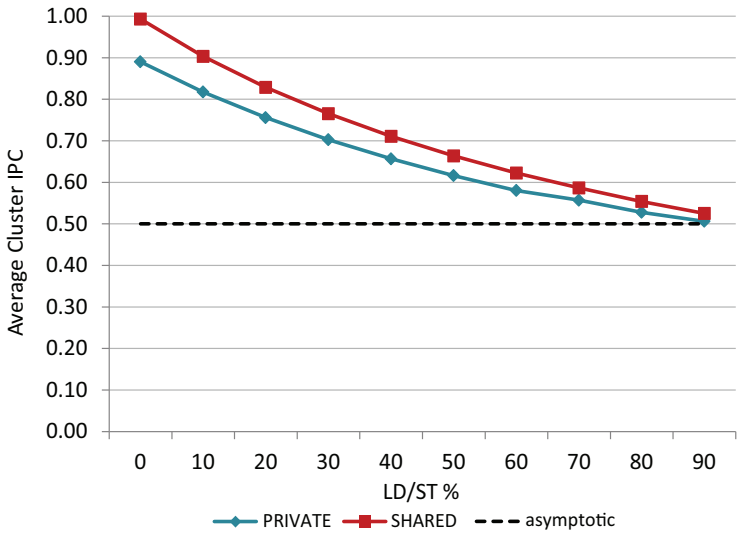

Figure 3.6: Private vs. Shared architectures IPC with conflict free TCDM accesses

Increasing N_{mem}/N_{alu} ratio in equation 3.2, leads to an asymptotic value value of 1/2. Cluster IPC, in this case of perfectly aligned execution, is $\overline{IPC} = 16 \cdot IPC_i$ and its average is equal to the IPC of a single core. Private architecture has an initial lower IPC due to the cold misses effect discussed in the previous paragraph.

Conflicts on TCDM accesses

This benchmark adds another aspect of TCDM accesses: conflicts. Conflicting accesses to the same bank increase TCDM latency thus affecting IPC. In this scenario we considered a realistic ratio between memory and ALU operations of 20%. As before, the loop is iterated 4K times to reduce cold misses effect. The plot in Figure 3.7 shows on the y-axis the cluster IPC while on the x-axis varies the percentage of memory accesses creating conflicts on the same bank. It is interesting to notice that, while there are no conflicts on TCDM, the shared architecture performs better the private one because of its intrinsic lower miss cost, in presence of TCDM conflicts the execution misalignment penalizes the shared cache architecture increasing the average hit time. It is important to underline that just a single conflict creates execution misalignment.

To explain the sharp reduction of the IPC for the shared cache due to TCDM

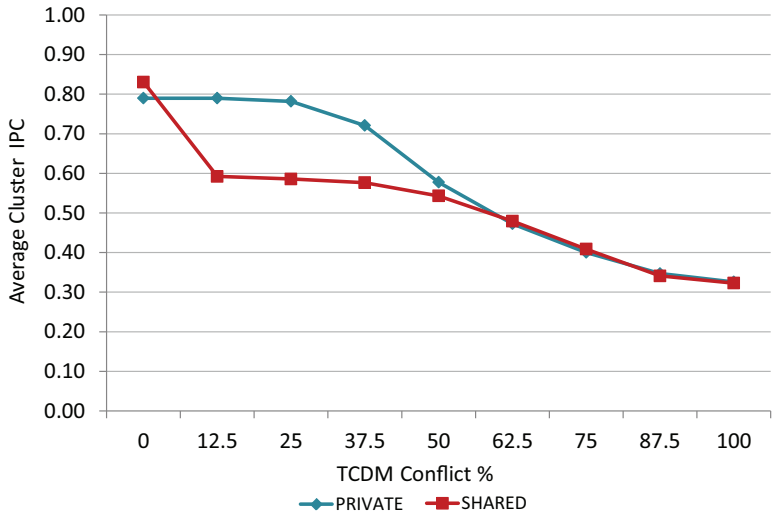

Figure 3.7: Private vs. Shared architectures IPC with conflicting TCDM accesses

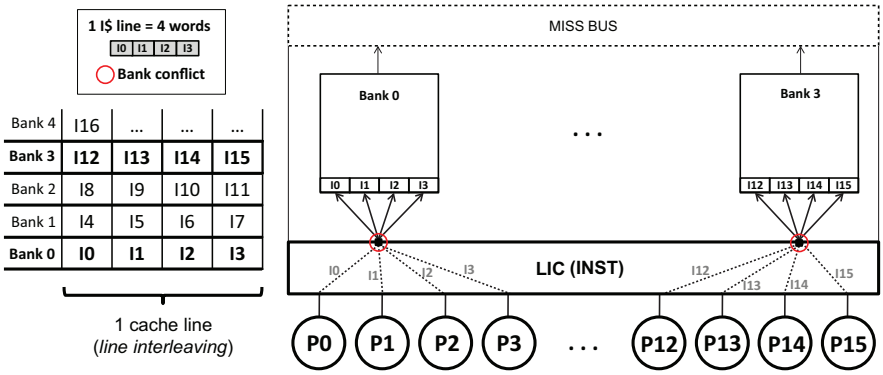

Figure 3.8: Misalignement in instruction fetching for shared cache

conflicts, let us consider a simple program consisting of 16 instructions. Before any TCDM conflict occur, the execution is perfectly aligned leading to synchronous instruction fetching. The conflicting access in TCDM leads to a single-cycle misalignment among all cores in the next instruction fetch. As shown in Figure 3.8, assuming a cache line is made of 4 32-bit words, there will be 4 groups of 4 processors accessing the same line (i.e. bank) but requesting instructions at different addresses. When this situation arises, the average hit time increases from 1 cycle (concurrent access) to 4 cycles (conflicting requests are served in a round-robin fashion). This particular case clearly shows how this architecture is sensitive to execution misalignment.

This phenomenon can stand out in an even worse case when the 4 blocks of

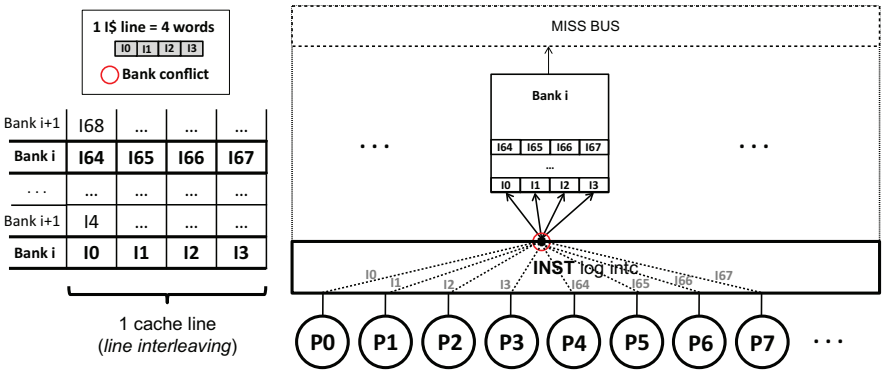

Figure 3.9: Worst case for instruction fetching in shared cache due to misalignment

instructions that are fetched by processors reside in the same bank (situation depicted in Figure 3.9). This leads to the worst-case for instruction fetch, increasing average hit time from 1 to 16 cycles.

3.5.2 Real Benchmarks

In this section we compare the performance of the private and shared I-cache architectures by using two real applications, namely a JPEG decoder and a Scale Invariant Feature Transform (SIFT) [54], a widely adopted algorithm in the domain of image recognition. In particular, our aim is to evaluate the behavior of the two target architectures when considering different types of parallelism at the application level. Therefore we parallelized our benchmarks with OpenMP [60], and considered three different scenarios.

The first case expresses data-parallelism at the application level. Thus we focused on the two data-independent computational kernels in JPEG: *Dequantization* (DQTZ) and *Inverse Discrete Cosine Transform* (IDCT) [1]. With this parallelization scheme all processing elements execute the same instructions, but over different data sets. In the second case we adopted pipeline parallelism in the same JPEG application, where each of the four stages of JPEG [1] - *Huffman DC, Huffman AC, DQTZ, IDCT* - is wrapped in an independent task and assigned to a core. To keep all the 16 cores busy we execute 4 pipelines in parallel. The third example considers three main kernels from SIFT: *Up-sampling*, *Gaussian Convolution*, and *Difference of Gaussians*, all leveraging data-parallelism [54]. In relation with the JPEG data parallel application, SIFT is composed of more complex computational steps that can stress the cache capacity causing more miss.

In what follows we carry out two main experiments, evaluating the performance by (i) varying the cache size and (ii) the L3 latency.
Figure 3.10 shows the results of the first experiment. We considered a fixed latency of 50 clock cycles for the L3 memory, and we varied the cache size. Focusing on the plots on the left side of Figure 3.10, for each of the three benchmarks and for the two architectures we show execution time, normalized to the slowest value

(i.e. the longest execution time for that benchmark). Looking at the data parallel variant of JPEG (JPEG par) it is possible to see that the shared cache architecture performs worse than the private cache. We would expect the SIMD parallelism exploited by this application to be preferred for the shared cache architecture, so this finding is seemingly counterintuitive. The reason for this loss of performance is to be found in an increased average hit cost, due to the banking conflicts in the instruction cache as described in the previous section (Figure 3.7). If we consider an analytical model the overall execution time of an application expressed as

$$T_{EX} = N_H \times C_H + N_M \times C_M \tag{3.3}$$

where N_H and N_M represent the number of hit and miss, and C_H and C_M represent the average cost for a hit and for a miss, C_H may be higher than 1 for the shared cache. To confirm this assumption we report the average cache hit ratio (left y-axis, solid lines) and cost (right y-axis, dashed lines) in the plots on the right side of Figure 3.10. It is possible to see that the average cache hit cost for the shared cache architecture is ≈ 2.4 cycles, while the number of miss is negligible (miss rate $= 0.003\%$). As a consequence, the right-hand part of the formula above does not contribute to the overall execution time. To understand the absence of cold cache miss impact we analyzed the disassembled program code. The DQTZ kernel consists of a loop composed by a few tens of instructions, while the IDCT kernel loop contains roughly 200 instructions. Overall this results in a hundred miss, and no capacity miss are later experienced. Due to the SIMD parallelism all cores fetch the same instructions, thus only the first core executing the program incurs cold cache miss. Instruction fetch from the remaining cores always results in a hit. In the private cache architecture, on the contrary, each core individually experiences 104 miss for cache sizes of 32 and 64 KB, while $\approx$ 400 for 16 KB. This results in a cluster miss rate (total number of miss over total number of instructions) of 0.05% for the private cache and 0.003% for the shared cache.

When considering the SIFT application the difference in the number of miss between the shared and the private architecture is major, as we can see in Fig-

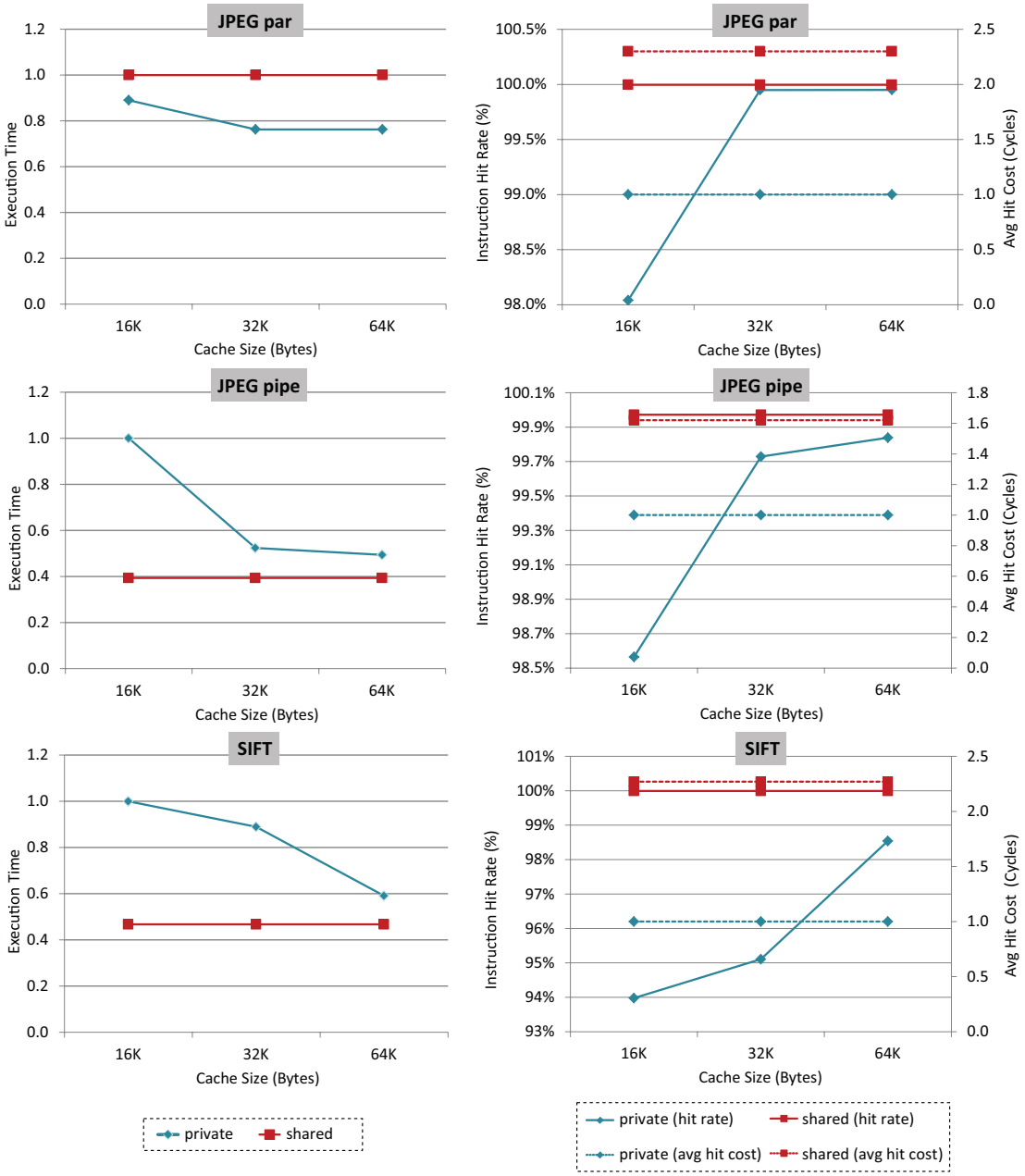

Figure 3.10: Impact of varying the cache size for different benchmarks

ure 3.10. In this case, due to the high average miss cost, the shared architecture provides best results despite the high average cost of an instruction hit (more than 2.25 cycles). Indeed, the average cost of a miss is $\approx$ 800 cycles for the private cache (any size), while for the shared cache it is $\approx$ 300 cycles. Again, this is

due to the fact that for the private cache multiple refills from different cores are serialized on the L3.

For the JPEG pipelined application, the shared cache has a miss rate of 0.03% against 0.3% (64 KB) of the private cache. Moreover in this case the shared cache has lower average costs for a hit (around 1.5 cycles) when compared to the other applications. The shared approach delivers 60% faster execution time for small cache sizes (16 KB), which is reduced to $\approx 10\%$ for bigger caches.

It is important to distinguish when an instruction miss occurs for the first time or not. In the first case we refer it as *cold miss*, while in the second case as a *capacity miss*. Table 3.3 shows the number of capacity miss on the total number of miss in percentage for the private cache architecture across all the applications. The shared cache architecture can better exploit the total cache size, therefore it does not experience capacity miss.

Table 3.3: Percentage of capacity miss over total number of miss

	JPEG PAR	JPEG PIPE	SIFT
SIZE 16KB	73%	88%	86%
SIZE 32KB	5%	41%	84%
SIZE 64KB	5%	4%	52%

Figure 3.11 shows the results for the second experiment, where we considered a fixed cache size (32 KB) and varied the latency of the L3 memory. The plots on the left side show normalized execution time (to the slowest, as before), whereas the plots on the right part show the average cost of a miss. Overall, it is possible to see that for L3 latency values beyond 100 cycles the shared cache architecture always performs better than the private cache architecture.

Considering equation 3.3 again, C_M is the parameter which is mostly affected by the varying L3 latency. In particular, the term $N_M \times C_M$ linearly increases with the L3 latency as we can see on the lower part of Figure 3.11. In the data parallel applications (first JPEG variant and SIFT), the average cost for a miss in the private cache architecture sharply increases with the L3 latency, whereas the same curve for the shared cache has a much smaller slope. This is due to

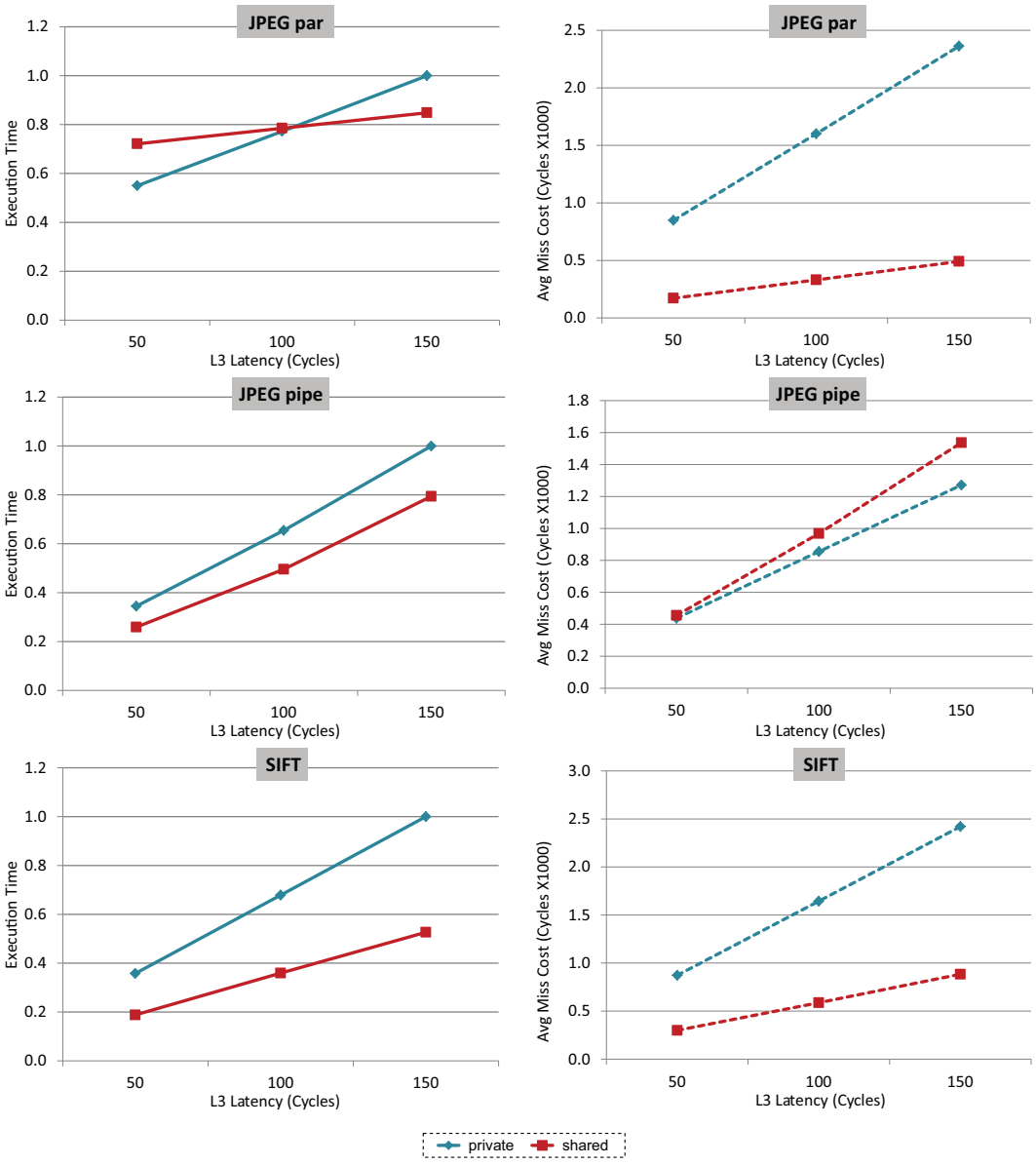

Figure 3.11: Impact of varying the latency of L3 memory for different benchmarks

the fact that private caches generate much more traffic towards the L3 memory (16 line-refill requests against a single refill needed by the shared cache). Then, despite the very low number of miss for the JPEG data parallel application, their contribution accounts for 50% of the overall execution time in equation 3.3.

Regarding the pipelined JPEG application, different from the other examples

the average miss cost is slightly higher for the shared cache. However, the miss rate for the shared cache is $\approx 0.02\%$, while for the private cache it is $\approx 0.2\%$, thus the shared architecture achieves slightly faster execution times.

3.5.3 Frequency Comparison

As a last experiment we investigated how faster the private cache design should be clocked to deliver the same performance achieved with the shared cache architecture. We considered as baseline configuration a L3 latency of 150 cycles and an I-cache of 32 KB. To carry out this comparison increasing the frequency of the clock within the cluster, we kept constant the L3 latency: our default T_{clk} is 10 ns leading to 1500 ns. The plot in Figure 3.12 shows on the y-axis the ratio between shared and private execution time for the benchmarks described in Section 3.5.2, while on the x-axis varies the percentage of frequency speedup.

Increasing cluster clock frequency has significant effect only for JPEG parallel while private architecture is quite insensitive for both SIFT and JPEG pipeline benchmarks. To explain such behavior we have to look at the execution time

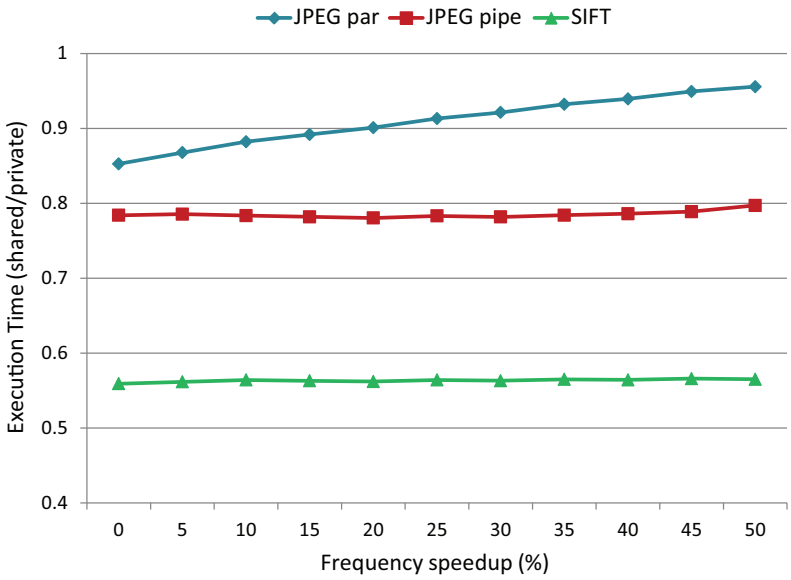

Figure 3.12: Frequency comparison of private and shared cache architectures

Table 3.4: Execution time breakdown for JPEG parallel, JPEG pipelined and SIFT benchmarks

	HIT & TCDM	MISS & L3 DATA
JPEG PAR	51.13%	48.87%
JPEG PIPE	14.72%	85.28%
SIFT	0.96%	99.04%

breakdown. A faster clock inside the cluster affects hit time and TCDM latency but has negligible effect on miss latency (dominated by L3 latency) and L3 data accesses. Table 3.4 shows execution time breakdown for all benchmarks.

Approximately 51% of execution time is affected by cluster clock frequency and determines the performance improvement of the private architecture for JPEG parallel. The same is not true for JPEG pipelined and SIFT benchmarks. This behavior underlines cluster performance is not affected by clock frequency when the program running has the execution time dominated by L3 memory accesses.

3.6 Conclusions

Key to providing I-fetch bandwidth for cluster-based CMP is an effective instruction cache architecture design. We analysed and compared the two most promising architectures for instruction caching targeting tightly coupled CMP clusters, namely private instruction caches per core and shared instruction cache per cluster. Experimental results showed that private cache performance can be significantly affected by the higher miss cost, on the other hand the shared cache has better performance, with speedup up to $\approx 60\%$. However, it is very sensitive to execution misalignment, which can lead to cache access conflicts and high hit cost.

Chapter 4

HW/SW Communication Mechanisms

4.1 Overview

Barrier synchronization becomes increasingly challenging as the level of integration in multi-processor systems-on-chip (MPSoC) keeps growing. There is today little doubt on the fact that software implementations are not suitable to provide the needed scalability of barrier synchronization in embedded systems and that some form of hardware support is essential.

Barrier optimization techniques in the embedded MPSoC domain often consist of optimized memory controller or communication controller interfaces [64], which aim at reducing the overhead of busy wait synchronization algorithms. These approaches focus on accelerating the barrier logic (i.e., loop over the participants for gathering and releasing them) and removing memory and interconnect congestion by providing dedicated local polling registers. However, the exchange of synchronization messages takes place through the main system interconnect, typically a Network-on-Chip (NoC) [90]. This solution is however non optimal, since communication requirements for synchronization and large-grain data movements are very different, and thus it is difficult to devise a topology which efficiently satisfies both. Moreover, the mutual interference between the two traffic flows on

one hand degrades application quality of service and on the other hand fails to provide ultra-low latency synchronizations.

A few solutions for embedded MPSoCs propose the adoption of dedicated communication infrastructures to carry synchronization-related traffic [70] [4]. However, these techniques rely on non-standard implementation technology to materialize congruent savings in synchronization latency. None of these technologies is within reach of a standard cell design methodology and hence of cost-effective implementations in the embedded computing domain. They are rather targeted to chip multiprocessors, where full-custom design techniques are commonly used for performance boosting.

In this chapter we propose a dedicated communication infrastructure implemented with standard cells and with a mainstream industrial toolflow. The superior efficiency, and above all scalability, of our target implementation is non-trivial to materialize because of several challenges. First, the RC propagation delay of on-chip interconnects degrades as feature sizes shrink, hence making global wires increasingly slow. Expected communication performance can be partially restored by routing tools by means of aggressive buffer insertion but at a relevant area and power cost [55]. Even tolerating this overhead, repeater insertion can only delay but not entirely stop the progressive shrinking of the wire feasibility region in the length-operating speed design space as an effect of technology scaling [73]. As an effect, the theoretical latency of different barrier algorithms (e.g., master-slave or tree-based) may not be reflected in the final implementation because of the interconnect bottleneck. Second, propagation delay of logic controllers required by each scheme affects their operating speed, again making relative performance non-trivial.

The most advanced MPSoC platforms achieve scalability through IP core clusterization and cluster replication [2] [11], where each cluster can potentially operate at an independent frequency for the sake of power efficiency. This architectural template is considered in this chapter and adds two new variables to the design space. On one hand, the most efficient hardware barrier implementation at cluster-level may not be the same for the top level, where global

synchronization between large clusters must be achieved. On the other hand, conveying synchronization messages at inter-cluster level is a globally asynchronous locally synchronous (GALS) communication issue that has never been adequately investigated before.

Our first contribution with this chapter consists of a physical design space exploration of collective communication structures at the intra- and inter-cluster level in an advanced 40nm technology library. We search for the most lightweight and performance efficient connectivity pattern for different clustering granularities and layout sizes.

In addition, we compare a global barrier approach independent of GALS partitioning with more sophisticated hierarchical barriers, employing a *master-slave* or a *tree* connectivity pattern at each layer of the hierarchy. While our results show that in absolute terms a global, system-wide barrier always provides the smallest synchronization latency, the hierarchical barrier enables an interesting feature, namely it supports multiple co-existing HW barriers (one per cluster). This is a very important feature for large systems capable of running different applications (or nested parallelism from within a single application) which need to synchronize independently.

As a second contribution, layout-aware performance of the most promising communication structures is annotated in the HDL (SystemC) models of a real-life MPSoC system. This virtual patform is enriched with a software stack composed of a OpenMP-based programming model, compiler and runtime system [23, 60]. The lower level barrier primitives of the OpenMP runtime environment have been customized to sit on top of our hardware support. This allows us to accurately quantify performance improvements with respect to the most efficient software implementations of barrier synchronization.

4.2 Hardware Support for Barrier Synchronizations

We do not rely on full-custom design techniques to achieve low-latency synchronization signaling but rather rely on mainstream industrial toolflows. Therefore, connectivity patterns are fully exposed to the effects of interconnect-dominated nanoscale technologies. Our choice is therefore for simple patterns and for low-bandwidth on-chip *Links* (1-bit width in most cases). These considerations led us to discard the butterfly and the all-to-all barriers presented in [90]. Their high number of links makes them unsuitable for a hardware implementation, especially in light of the high link inference cost that will be highlighted in Section 4.3. We rather selected three connectivity patterns and associated synchronization protocols that hold promise of better silicon implementation efficiency: the Central barrier, the Gline barrier and the Tree barrier, illustrated in the following sections. Since the focus of this work is on clusterized systems, the hardware-barriers under test are explored within a single cluster, which we assume to be covered by a single clock domain, and among clusters, where clock domain crossing becomes an issue.

4.2.1 Intra-Cluster barriers

All the proposed intra-cluster barriers execute the same two-phase protocol of a typical Master-Slave Barrier [90]: the *account phase* and the *release phase*. How-

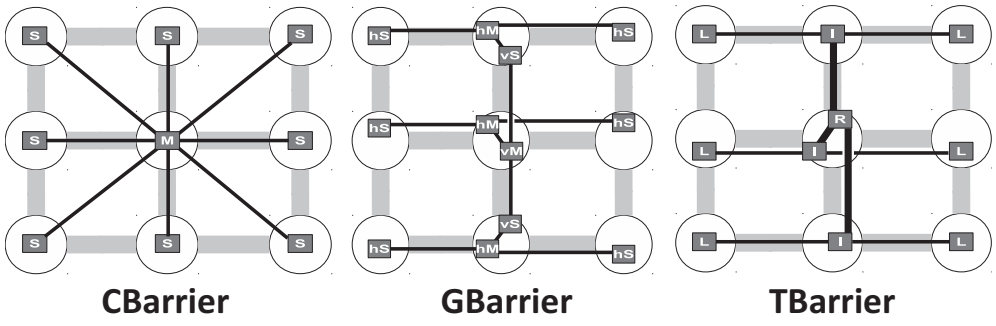

Figure 4.1: Intra-cluster Barriers for a 9-core Cluster.

ever, unlike the software-based schemes, when a thread arrives at the barrier, it has to activate its corresponding intra-cluster barrier's *Controller* in order to initialize the hardware barrier. Hence, once all threads have reached the barrier, the *account phase* is completed and the *release phase* starts. Then, the hardware barrier has to command all threads through the corresponding intra-cluster barrier's *Controllers* to quit the synchronization phase. Moreover, without lack of generality, we consider a cluster architecture composed of 9 single-threaded cores interconnected by a 2D-mesh topology in order to illustrate our hardware barriers.

4.2.1.1 The Central Barrier architecture

CBarrier is schematically shown in Figure 4.1. *Links* are represented with finer black lines[1], while *Controllers* are depicted as dark gray boxes. In particular, there are two kind of controllers, namely Master and Slave (M and S boxes, respectively). We selected *CBarriers* for the minimum number of synchronization stages, a desirable property for hardware acceleration.

The synchronization protocol for the *CBarrier* architecture relies on the exchange of 1-bit messages (signals) between the master and slave controllers. It uses a centralized approach where the master controller is responsible for collecting all signals from slaves (*account phase*), and then, for instructing them to resume execution (*release phase*). As an example, Figure 4.2 represents the

[1]For clarity, we represent links taking into account neither their width nor their physical placement.

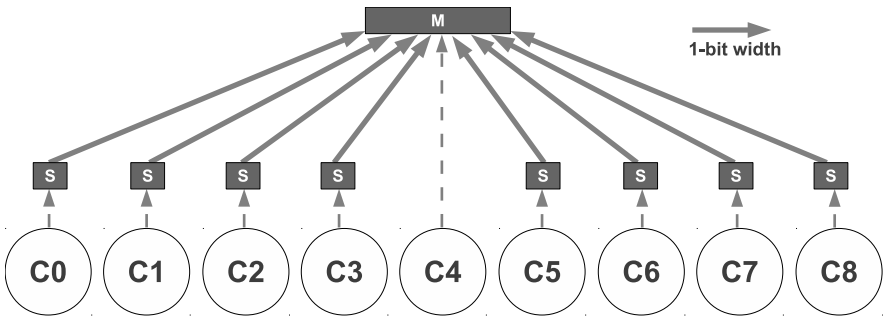

Figure 4.2: Account phase for *CBarrier* architecture.

account phase for the *CBarrier* protocol when all cores participate in the barrier and they execute it at the same time. The release phase is exactly the same except for the notifications flowing in the opposite direction. Only communications between controllers are global and are denoted with solid lines.

4.2.1.2 The Gline-based Barrier architecture

GBarrier is shown in Figure 4.1. As we can observe, there are a number of *Links* that interconnect four sort of *Controllers*. We selected *GBarriers* since they match the 2D mesh structure of the regular array fabric of processing elements. This hardware barrier mechanism is based on the work in [4]. However, rather than leveraging full-custom *G-lines* technology and *S-CSMA* technique, in the present work we implement and assess *GBarriers* with a standard cell design methodology. First, *G-lines* are implemented through conventional on-chip wires, allocating a different line per slave controller (in contrast to [4], where *G-lines* could be shared by different slaves connected to the same master controller). Second, we mimic the *S-CSMA* technique, that allows a master controller to determine the number of simultaneous slaves' signals transmitted over a particular *G-line*, by instructing the master to sample its different slaves' lines in a loop until all expected signals have been received.

The synchronization protocol for *GBarrier* is depicted in Figure 4.3 for the account phase (further details can be found in [4]).

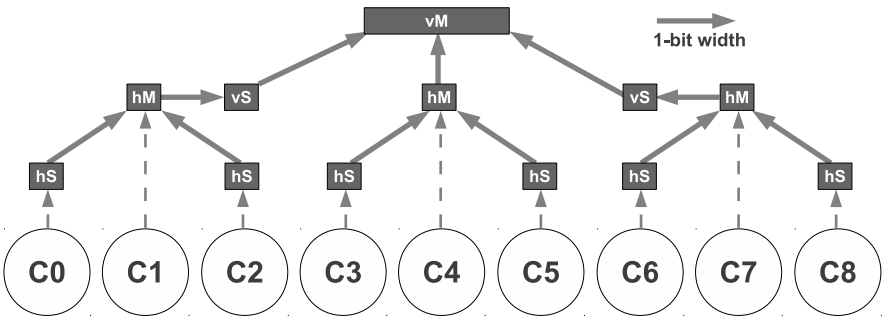

Figure 4.3: Account phase for *GBarrier* architecture.

4.2.1.3 The Tree Barrier architecture

(*TBarrier*) is shown in Figure 4.1. As we can observe, there are three kind of *Controllers* as a consequence of the role that they are playing in this tree-based architecture to implement barrier operations: *L* stems from leaf nodes; *I* is for an internal node; and *R* is for the root of the tree. We selected *TBarriers* for their nice theoretical scalability property with the number of cores, although this may be questioned by performance degradation effects in the physical implementation. The synchronization protocol for the *account phase* is illustrated in Figure 4.4. It implements a tree-based approach where the Root controller is responsible to count the number of participant threads in the barrier. For the first phase, all L controllers send one 1-bit message towards their corresponding I controller. Then, the I calculates how many signals has received and builds a new message that is sent towards the R controller. In this case, the *Link* will be of greater width than the former between L and I controllers. In general, these *Links* will be of $log_2(Leaves)$-bit width (e.g. 2-bit width for the layout in the Figure because it could be necessary to transmit a message containing a maximum value of 3). Hence, we depict these *Links* with wider lines in Figure 4.4. Finally, R receives three messages from I controllers and calculates the sum of all messages' numbers received (i.e. a maximum value of 9 for this setting). In the release phase (not shown for lack of space), the main difference lies in the fact that only 1-bit width *Links* are needed to command the completion of the synchronization.

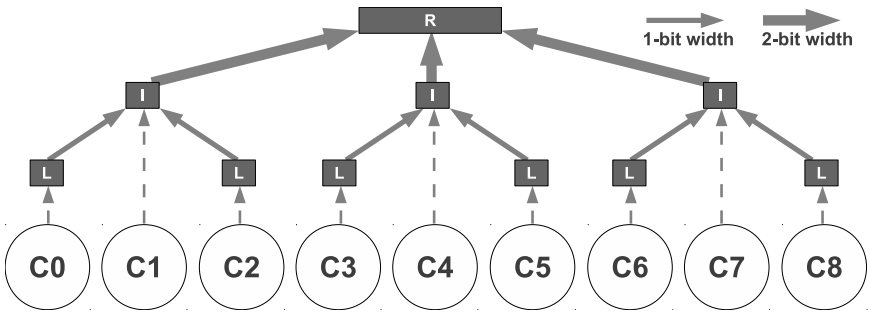

Figure 4.4: Account phase for *TBarrier* architecture.

4.2.2 Inter-cluster Barriers

We have designed inter-cluster barriers spanning multiple clock domains by means of asynchronous global Links (*gLinks*). In order to avoid metastability at the controller in the receiving cluster, we use brute-force synchronizers. We found this kind of synchronization interface more suitable with respect to alternative ones such as dual-clock FIFOs because of the tiny width of our links and of the one-shot nature of communications over them. Also plausible clocking is not suitable due to the multi-port nature of controllers.

As an example, Figure 4.5 shows our *CBarrier* architecture when used at the top layer to carry out inter-cluster synchronizations. For communication between Master and Slave controllers, we use two brute-force synchronizers (*BFsynch0* and *BFsynch1*). Similarly, by simply adding one brute-force synchronizer for every *gLink*, the inter-cluster schemes for *GBarrier* and *TBarrier* would be implemented. Therefore, we omit these explanations for sake of brevity. However, it is worth noting that, every *BFsynch* is 1-bit width except for the *TBarrier* implementation. As pointed out in Section 4.2.1.3, communications between internal and root controllers utilize up to $log_2(Leaves)$ bits, thus requiring $log_2(Leaves)$-bit width *BFsynchs*. We illustrate in Figure 4.6 how the overall synchronization process works for a platform composed of two 2-core clusters covered by two different clock domains. In the example, we assume the *CBarrier* design for both intra- and inter-cluster levels. As we can see, we distinguish between local and

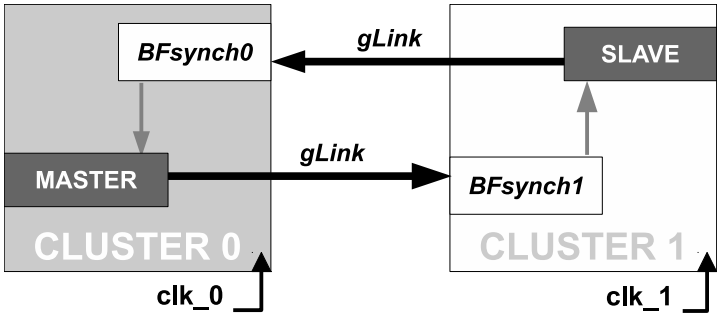

Figure 4.5: Inter-cluster *CBarrier* architecture for 2 clusters.

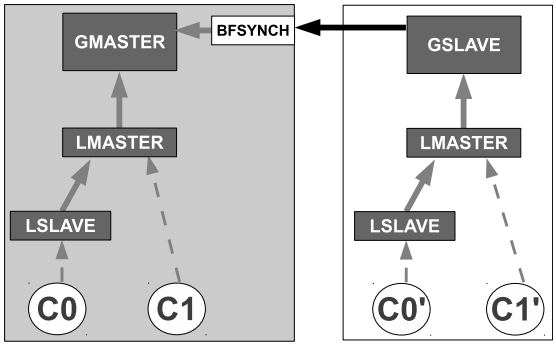

Figure 4.6: Account phase for *CBarrier* at intra- and inter-cluster levels.

global controllers (L and G prefixes, respectively) depending on the level of the synchronization. Notice that, Local Master controllers have also been extended to locally communicate with their corresponding global controller enabling the interplay between the two levels. We highlight in black color the arrows for *gLinks* among the two clusters. Using *GBarriers* and *TBarriers* for inter-cluster synchronization can be done in a similar way. Also, it is possible to use different protocols at each layer of the design hierarchy.

4.3 Evaluation

4.3.1 Experimental Setup

For a realistic characterization of our proposals for hardware barrier acceleration at intra- and inter-cluster levels, this work makes use of a mainstream industrial synthesis toolflow and of an STMicroelectronics 45nm standard cell technology library [83]. Placement-aware logic synthesis is performed through Synopsys Physical Compiler. The final place-and-route step is performed with Cadence SoC Encounter which also involves clock tree synthesis. We assume a single clock domain with a unique clock tree for the intra-cluster barriers, meanwhile for inter-cluster barriers we consider different clock domains/clock trees for every cluster of the configurations. Finally, a sign-off procedure is run by Synopsys PrimeTime to accurately validate the timing properties of our designs. Moreover, our mechanisms have been studied by defining non-routable obstructions. Such obstructions are placed to mimic the area of every core of the simulated systems. In this work, we assume that this area is equal to $0.55 \times 0.55 mm^2$. Additionally, fences are defined to limit the area where the cells of each barrier's controller can be placed. Such obstructions and fences also ensure minimum-length routing for the links in order to reduce their impact on performance and area overhead as the wire length increases.

4.3.2 Intra-cluster Barriers

Assuming that all cores arrive at the barrier at the same time, the theoretical numbers of clock cycles that our intra-cluster barriers take are the following: 6 cycles for CBarrier; 14 for GBarrier; and, 10 for TBarrier. These numbers are derived from the set of operations carried out by the synchronization protocols explained in Section 4.2.1. We point out that each barrier's controller has been implemented by separating the delay that signals take along the wires, from the effective computation that the controllers require to generate their output signals. For small clusters, the critical path is defined by the most complex barrier's controller (e.g. the Master for CBarrier), but as the wire length increases for

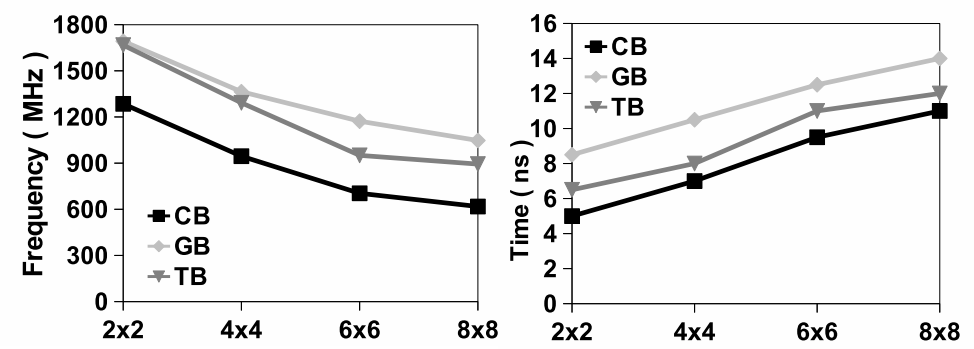

Figure 4.7: Maximum frequency and minimum latency for intra-cluster barriers depending on cluster size.

bigger clusters, the wires could represent such critical path. Consequently, separating wire delays from controllers delays becomes essential in order to achieve maximum clock speeds.

Figure 4.7 depicts the maximum frequencies reached by each intra-cluster barrier depending on the cluster size. As we can observe, each design can run faster depending on the number of steps required to perform a barrier synchronization. That is, the greater number of steps, the higher the frequency that will be achieved. For this reason, the higher frequencies are obtained for the GBarrier design. Moreover, as the size of the clusters becomes larger, the timing critical paths obtained for barrier's controllers and wires are longer translating into lower achievable frequencies. Regarding barrier latencies, CBarrier is the faster architecture for all settings despite of running at lower frequencies for all configurations. It is due to the fact that, CBarrier operates in almost half number of cycles than the other designs, but there is not such difference between the achievable frequencies.

When we consider area figures using the maximum performance setting, faster and greater cells (higher drive strengths) are instantiated in all solutions, therefore area gaps illustrated in Figure 4.8 are reduced, although the most expensive and the cheapest solutions remain the same. For this reason, the relative plot is omitted.

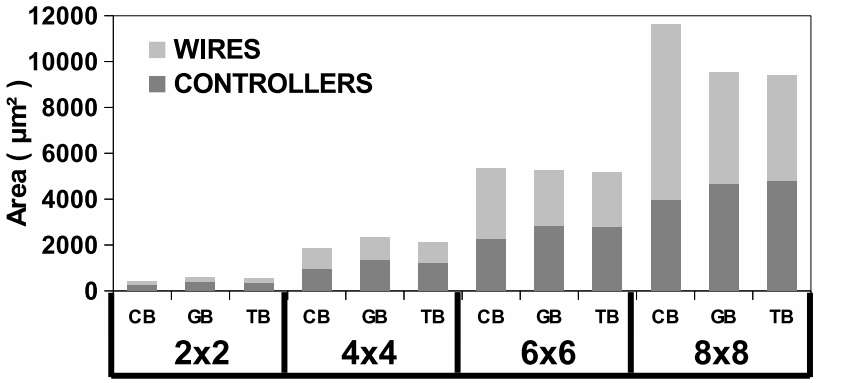

Figure 4.8: Area overhead for intra-cluster barriers running at 600MHz.

4.3.3 Inter-cluster Barriers

In this section, we study our hardware-based barriers at inter-cluster level in terms of area overhead and barrier latency when multiple clock domains/clusters are considered. Particularly, 2x2 and 4x4 clusters, composed of 4x4 cores per cluster. At this layer, *gLinks* could introduce an unpredictable delay since they are considered as asynchronous and potentially unconstrained by the routing tool. Since we are targeting ultra-low latency communications, we explicitly constrain propagation delay across asynchronous links through a *set_max_delay* command equal to a quasi-zero value for every *gLink* in the three designs.

As explained above in Figure 4.6, the synchronization protocol for a system composed of multiple clusters/clock domains is split into two levels of synchronization. On one hand, the first level that is implemented by using a single intra-cluster barrier for every cluster/clock domain. On the other hand, the top level that is implemented by employing inter-cluster barriers to enable the interplay between the different clusters/clock domains. This means that barrier's controllers at inter-cluster level could run using different clock speeds depending on the cluster/clock domain they belong to. Therefore, the maximum frequency at top level for a particular inter-cluster controller is limited by the maximum operating speed achieved at the first level and vice-versa. For the scenarios discussed above, we obtained that the maximum operating speed is imposed by the intra-cluster barrier. Therefore, assuming the most efficient intra-cluster bar-

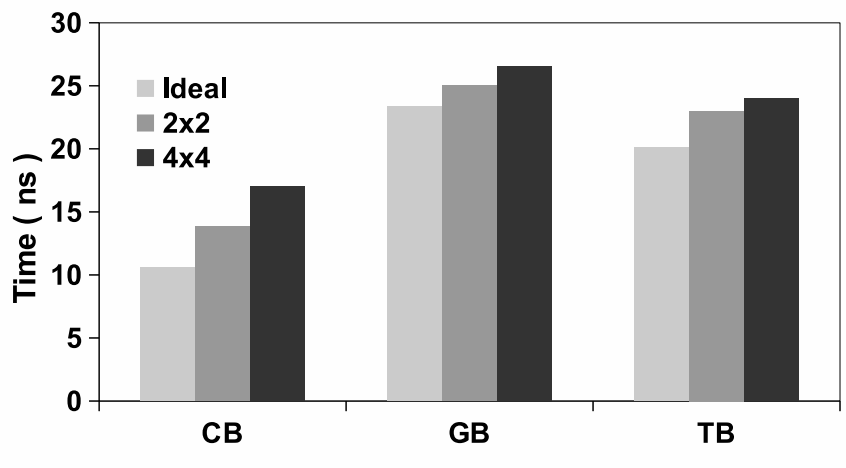

Figure 4.9: Latency for inter-cluster barriers running at maximum frequency.

rier for a 4x4-core cluster (intra-CBarrier), our inter-cluster barriers have been synthesized at 950MHz (see Figure 4.7).

Figure 4.9 shows the barrier latencies for 2x2 and 4x4 clusters (i.e. 64 and 256 cores, respectively) running at the maximum speed discussed above (950 MHz). We would like to point out that these latencies correspond to the inter-cluster barrier operation without adding up intra-cluster time. Besides, we also depict the barrier delay that the three designs take in theory running at the target frequency. As we can observe, the most efficient implementation is still CBarrier for these particular configurations. The only reason why CBarrier could not be the most efficient architecture is when *gLinks* are too long, thus vanishing the benefits of having less number of stages in comparison to the other GBarrier and TBarrier designs. From the Figure, we can observe the higher effect of the length-delay phenomenon of *gLinks* for the CBarrier architecture, as compared to theoretical barrier latencies for the three designs. Nonetheless, this is not enough for outperforming the GBarrier and TBarrier designs. We analyze in depth this issue by using a sign-off tool (i.e. Synopsys PrimeTime). This tool reports that the timing of links as a function of their lengths are as follows: 0.7, 1.2 and 2.2 ns; for 2.2, 4.4 and 8.8 mm respectively. As our architectures do not use longer links than 8.8 mm, negligible penalties in latency are reported thus explaining

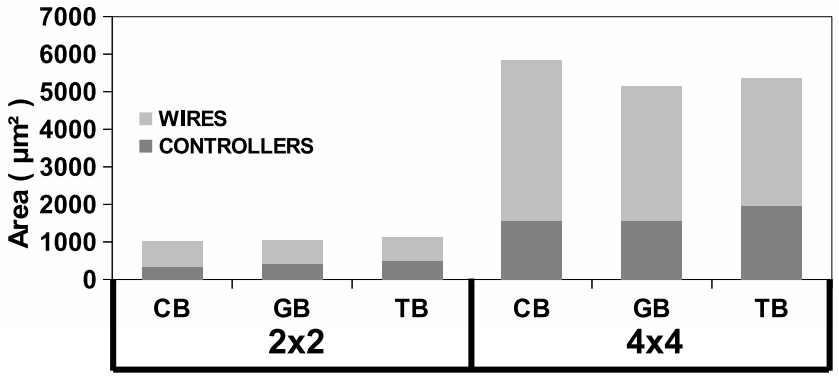

Figure 4.10: Area overhead for inter-cluster barriers.

the higher efficiency of the inter-cluster CBarrier.

Moreover, Figure 4.10 shows the area overhead for the three designs, again ignoring the contribution of the intra-cluster infrastructure. As we can see, *gLinks* constitute the most consuming part of the designs thus introducing the major overhead for CBarrier, since it is the design with the longest wires.

Finally, the above inter-cluster barriers have been also analyzed in terms of barrier latency when different clusters operate at different frequencies for a 4x4-cluster platform. In particular, we use two frequencies (300 and 600 MHz) that have been assigned throughout the 16 clusters in a combinatorial way (see Figure 4.11). We show the barrier latencies for the three inter-clusters in function of all the combinations (X-axis). As we can observe, intra-CBarrier is the most effi-

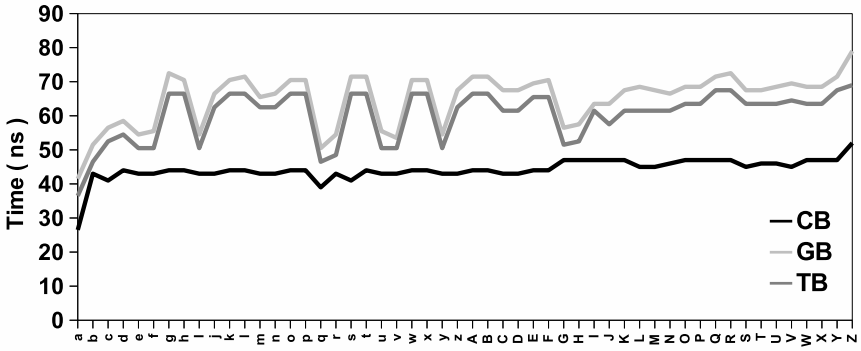

Figure 4.11: Barrier latency for inter-cluster barriers for frequencies from 300 to 600 MHz.

cient implementation and barrier delay increases from left (all clusters at 600MHz) to right (all clusters at 300MHz), that is, from the fastest to the slowest configuration. Moreover, inter-GBarrier and inter-TBarrier report closer times to those obtained by the inter-CBarrier only when either the top-level hMaster/vMaster for the former, or Internal or Root controllers for the latter operate at the maximum frequency (600 MHz). In those cases, the accounting of signals is performed faster what improves the barrier efficiency.

4.3.4 Clusterization Overhead

So far the cluster structure of the MPSoC system has been equally reflected in the cluster structure of the custom interconnect for synchronization signaling. Such an approach has an unique advantages: it is capable of supporting multiple co-existing barriers in the system. Indeed, each of the controllers can be independently programmed by the software, thus enabling disjoint synchronization domains. In this section, we aim at quantifying the overhead with respect to a flat interconnect solution. As a case study, we consider a 64-core platform split into four different clusters/clock domains in which independent applications could run simultaneously on the available 16 cores. This could be likely one of the most appropriate scenarios considering such amount of cores in order to exploit available hardware resources.

From Sections 4.3.2 and 4.3.3, we derive that the inter- and intra-CBarrier architectures are the preferred choice for the target platform when the hierarchical approach is taken (see Figure 4.12).

Notice that, black boxes represent the top level of the hierarchy, that is, the inter-cluster controllers (for the sake of clarity, we do not show the gLinks and brute-force synchronizers explained in Section 4.3.3). The remaining controllers are for the four intra-cluster CBarriers. Besides, we highlight with different colors the four clusters/clock domains. We compare the hierarchical CBarrier with a flat layout composed of a single level in which all controllers are logically placed (see non-hierarchical scheme in Figure 4.12) and connected through the CBarrier pattern to a centralized global master. Of course, brute force synchronizers have

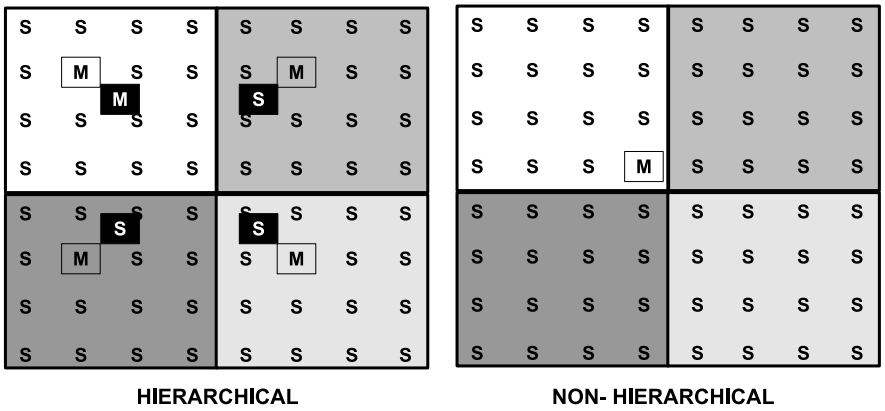

Figure 4.12: Hierarchical/non-hierarchical CBarrier architecture.

been used for all clock domain crossings.

In Table 4.1, we report the performance results in terms of maximum frequency, barrier latency and the area overhead for the hierarchical and non-hierarchical CBarrier layouts. As we can see, the maximum frequency is achieved by the hierarchical design. It is due to the fact that for both scenarios, the Master controllers constitute the critical path to performance (remember that *gLinks* are considered as false paths for their asynchronous nature, thereby not limiting the maximum achievable operating speed). Then, the more complex Master, the lower frequency will be achieved. Since, there is only a single Master in the flat scenario that gathers all signals from all Slaves in comparison to the four smaller Masters presented in the hierarchical design, the latter can support a higher operating speed. Regarding the latency, the flat design is more efficient. First, the hierarchical layout nearly doubles the number of steps (clock cycles) carried out by the flat design, and this is not the case for the gaps between the maximum frequencies. Second, the longest link for both designs has the same length of

Table 4.1: Performance statistics for CBarrier designs.

	Frequency	**Latency**	**Area Wires**	**Area Ctrls**
Hierarchical	950 MHz	22 ns	4,935 μm^2	5,922 μm^2
Non-Hierarchical	620 MHz	17 ns	6,137 μm^2	7,977 μm^2

4.4mm. However, according to our previous study, this wire length introduces a delay of 1.2ns which is enough to cover one clock cycle in the flat design (1.61ns) but not for the other one (1.05ns). This means that the hierarchical design takes in the longest link 2 clock cycles, thus keeping away a little more the distance in terms of barrier latency as compared to the flat design. Regarding area overhead, the flat scheme is the most consuming design. First, it is mainly due to the very huge number of brute-force synchronizers that this layout requires in comparison to the 6 that the hierarchical uses. In more detail, the flat design utilizes as many synchronizers as the number of Slave controllers that do not belong to the Master's clock domain (i.e. 48 Slaves), multiplied by 2 to cover the two senses of the communication. Second, the average link length of the flat design is longer than the hierarchical layout, thus requiring a higher area overhead.

As a conclusion of this study, we could say that the flat architecture is the best design in terms of latency at the expenses of higher area overhead. The main drawback of this design is that it cannot take benefit from mapping a different application to each of the four different clusters, thus vanishing the benefits of using clusterized multiple-domain platforms.

4.3.5 Full-system Simulation

As a final exploration, we want to assess the impact of coupling our HW barriers with a real-life software stack. To this aim, we developed SystemC models of the two main components of our hierarchical barriers, namely the local and global controllers described in Section 4.2. We annotate these models with the latencies extracted from the characterization presented in Table 4.1. The models are finally integrated in a cycle-accurate full-system simulator which allows us to build an instance of the 64-core, 4-cluster MPSoC considered as a use case in Section 4.3.4. Clusters are interconnected through a global NoC. Each of them features 16 cores, communicating through a fast multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM access have two-cycles latency.

To accurately account for the overheads introduced by a realistic software stack, we integrate our HW barrier into a widespread programming model such as OpenMP. In OpenMP, parallelism is specified at a very high level by inserting directives (annotation) to a sequential C program. The compiler is responsible for translating these directives into parallel threads of execution. Many of the parallelization services provided by OpenMP are implemented within a runtime library, queried by the parallel threads. We re-wrote the low level OpenMP runtime primitives for barrier synchronization so that we can select between an optimized software implementation and the invocation of our HW controllers.

As a software implementation, we consider the topology-aware variant of the tree barrier discussed in [60]. We choose this barrier because it is reported in literature as one of the best-performing for distributed systems, and because it has in practice an analogous behavior to its hardware counterpart. Processors in the system are classified into three entities:

- One *Global Master*
- One *Local Master* per cluster
- *Local Slaves* (the remaining processors)

The SW barrier operates in four steps:

1. In the *Local Gather* phase, each of the Local Masters wait for each of its slaves to notify its arrival on the barrier on a private status flag (LOCAL_NOTIFY array). After arrival notification, Local Slaves check for barrier termination on a separate private location (LOCAL_RELEASE array).

2. In the *Global Notify phase*, the Global Master waits for all Local Masters to notify their arrival in a private status flag (GLOBAL_NOTIFY array). After arrival notification, Local Masters wait for global synchronization termination on a local flag (GLOBAL_RELEASE).

3. In the *Global Release* phase, the Global Master notifies the termination of the global synchronization step by writing into each Local Master's GLOBAL_RELEASE flag.

4. In the *Local Release* phase, each Local Master notifies the termination of the whole barrier by writing into each Local slave's private flag from the LOCAL_RELEASE array.

To prevent polling activity from injecting interfering traffic on the interconnect, we distribute notification and release flags so that each processor does busy waiting on a private, local memory cell. In particular, we leverage the multi-banking feature of on-cluster TCDMs to make sure that each slave directs its polling transactions to a different memory bank. Regarding the HW barriers, we considered both the *Hierarchical* and *Non-hierarchical* barriers described in the previous sections. The number of threads involved in a parallel region can be set by the programmer with the clause `num_threads` when invoking the OpenMP `#pragma omp parallel` directive. Our barriers are capable of synchronizing a smaller number of cores than the total, but a *Setup* phase is necessary to appropriately program the controllers. For the *Non-hierarchical* HW barrier this setup only consists of a write to the controller's `max_events` memory-mapped register. For the *Hierarchical* HW barrier and for the SW barrier the setup is slightly more complex.

Based on the number of cores in each cluster (`CPC`) and the number of threads participating in the parallel region, it is necessary to figure out the number of clusters involved in the synchronization operation (`FC`). This number must be annotated into the memory-mapped register of the global controller. All the processors belonging to the first `FC` - 1 clusters will take part to the barrier, thus corresponding local controllers must be programmed to synchronize all of them. On the contrary, not all the processors belonging to the `FC`-*th* cluster may be involved in the barrier, thus the appropriate number must be computed and registered in the pertinent local controller.

The more natural way to integrate barrier setup into the OpenMP execution model is to let the master thread accomplish this programming stage upon parallel

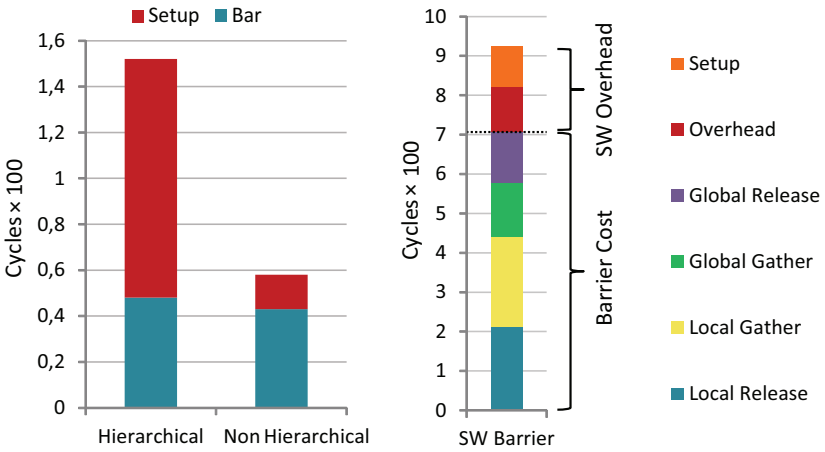

Figure 4.13: HW (left) and SW (right) barriers cost

region creation. We thus inserted this set of write operations inside the runtime library function `parallel_start`. These setup operations are regular writes into memory mapped registers. The corresponding transactions travel through the NoC, directed to each active cluster's controller. Once every controller has been programmed, the underlying hardware mechanism can be triggered via SW by writing/reading into the `bar_reg_in` and `bar_reg_out` registers.

As a first experiment, we compare the cost, in terms of cycles, of both HW and SW barrier implementations. To avoid measuring wait time due to misaligned thread arrival on the SW barrier, we measured SW barrier time from the master thread, ensuring that all slaves have already entered the barrier. The barrier cost breakdown for the SW tree barrier is shown in Figure 4.13 (right). The SW barrier costs approximately 700 cycles, considering the net time for gathering and releasing slaves locally and globally. An additional hundred cycles are induced by call overheads in the OpenMP runtime environment, plus the cost to initialize the barrier itself (setup phase), which amounts to 104 cycles. Overall, synchronizing 64 cores from OpenMP costs slightly more than 900 cycles.

In Figure 4.13 (left) we report the cost for the two HW barrier implementations. It is possible to see that, while the barrier time itself is not very different in the two cases, the setup phase, as expected, takes quite longer for the *Hierarchical*

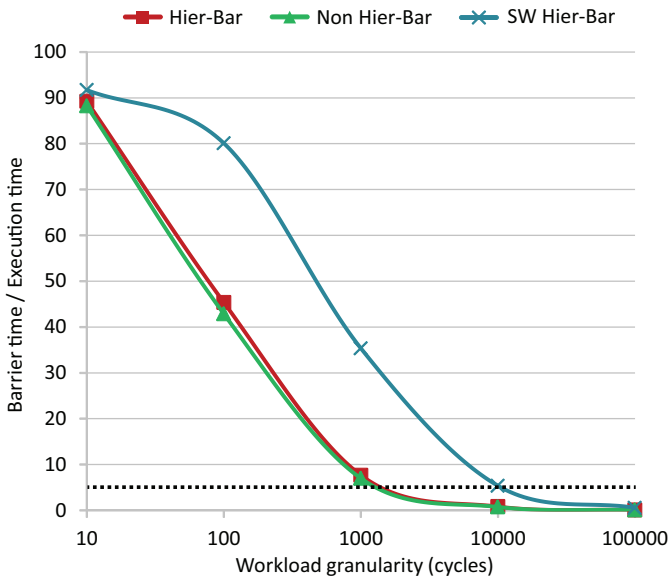

Figure 4.14: Barrier cost / Workload

barrier. It has however to be pointed out that the cost for the setup phase has to be paid only when opening a parallel region with a different number than the previous. If the number of threads does not change among two parallel regions, the cost for the setup phase is drastically reduced (around 15 cycles).

As a second experiment, we want to estimate the granularity of parallelism enabled by the different barrier implementations. To this aim we use a small synthetic loop, which repeatedly invokes a small assembly routine composed uniquely of ALU instructions (to avoid memory contention effects). This routine is annotated with a `#pragma omp parallel` directive, which replicates its execution among all the participating threads. This routine can be parameterized to generate increasing granularities of parallel tasks (10 to 10000 cycles), so as to study how the parallelism is affected by barrier time.

Plots are shown in Figure 4.14 for both HW and SW barriers. Here we show the percentage of time spent in syncronization when the granularity of the parallel task (i.e. the cycles taken for its execution) increases. For extremely small tasks (10 cycles) the barrier time is dominating in all cases ($\geq$ 90%). However, it

is possible to see that clearly hw barriers cut down on latency by one order of magnitude with respect to the sw barrier. If we qualitatively establish that a 5% overhead for synchronization is negligible, it is possible to see that this point is reached by HW barriers at a granularity of thousand cycles, while this same point is reached by the SW barrier at ten thousand cycles. It is also worth underlining that from the software perspective the difference in latency between the two HW barriers is not appreciable, since the overheads introduced by the software stack tend to hide it.

4.4 Conclusions

Designing a dedicated collective communication infrastructure for synchronization signaling with standard design tools and technology libraries is a challenging task since it is directly exposed to the effects of interconnect-dominated deep submicron technologies. In spite of this, hardware barriers with the lower number of stages proved the most performance efficient in our physical implementation framework although using longer links. This is because the interconnect delay is not such to offset the inherent lower cycle count these schemes take to synchronize the system. Where instead the interconnect delay plays a role is in determining area of the hardware barrier, since the place-and-route tool operates aggressive repeater insertion to sustain performance over long links. However, from the software perspective the picture changes slightly. When integrating our HW barriers into complete software stacks (i.e., a programming model and its runtime environment) we saw that the difference in latency between the most performance-efficient implementation and the second best is not that relevant, because the software support for parallelism creation introduces sources of overhead that tend to hide it. Our experiments with OpenMP demonstrate that both the explored HW barrier solutions enable one order of magnitude-finer grained parallelism than pure-software implementations. Moreover, the hierarchical barrier allows to independently synchronize multiple processor groups (one per cluster) concurrently. This is something that is extremely important, for example, when

nested parallelism comes into play, or when multiple disjoint parallel applications are running on the system.

Chapter 5

Memory Energy Efficiency

In this chapter the focus is on Technology scaling enables today the design of sensor-based ultra-low cost chips well suited for emerging applications such as wireless body sensor networks, urban life and environment monitoring. Energy consumption is the key limiting factor of this up-coming revolution and memories are often the energy bottleneck mainly due to leakage power. In this chapter we devise an ultra-low power version of our multi-core architecture targeting e-Health monitoring systems, where applications involve collection of sequences of slow biomedical signals and highly parallel computations at very low voltage. By combining 6T-SRAM and 8T-SRAM memory portions, operating in the same voltage domain, such architecture is capable of dispatching at high voltage a normal operation and at low voltage a fully reliable small memory partition (8T), while the rest of the memory (6T) is state-retentive. Our architecture offers significant energy savings with a low area overhead in typical e-Health Compressed Sensing-based applications.

5.1 Overview

Emerging and future healthcare policies are fueling up an application driven shift toward long term monitoring of bio-signals by means of embedded ultra-low power (ULP) devices. Modern human behavior-related diseases, such as cardiovascular

diseases, require accurate and continuous medical supervision, which is unsustainable for the traditional healthcare system due to increasing costs and medical management needs [71]. Personal health monitoring systems are able to offer large-scale and cost-effective solutions to this problem.

Wearable health monitoring systems, enabled by Wireless Body Sensor Networks (WBSNs), face contrasting requirements such as a continuously tighter power budget and an increasing demand of computation capabilities to preprocess locally the sensors information so as to reduce the amount of data transmitted, as well as response time. To ensure minimal energy several aspects must be considered, combining optimizations of the signal processing aspects and of the technological layers of the ULP architecture.

Several works in literature [58, 78] show that embedded feature extraction algorithms and data compression schemes greatly contribute to minimizing energy. Compressed Sensing (CS) signal acquisition/compression paradigm has recently proved to be effective in reducing energy consumption in embedded ECG monitors. Enabling a sub-Nyquist sampling rate for sparse signals, authors in [58] show a 37.1% improved lifetime compared to state-of-the-art compression techniques.

At the architectural level, voltage scaling has been widely used and proved its effectiveness though it faces several challenges. Supply voltage has remained essentially constant beyond 65nm and dynamic energy efficiency improvements have stagnated, while leakage currents continue to increase.

Motivated by the inherent parallel nature of medical grade ECG monitoring, where multi-channel signal analysis is often embarrassingly parallel, multi-core architectures proved their efficiency compared to single-core solutions [25, 27]. In [25] authors introduced a multi-core architecture where individual leads are processed on different cores in parallel. Parallel processing enables more aggressive voltage-frequency scaling than single-core solutions, though at low workload requirements the single-core solution proved to be more efficient. The efficiency of the multi-core architecture was further extended in [26], by deploying broadcast mechanism in the instruction memory and clock gating on memories, achieving extra 39.5% power savings at high workload requirements. While at low workload

requirements leakage power, mainly due to data and instruction memories, has a big impact and aggressive voltage scaling cannot be applied due to reliability issues for the memories.

Unfortunately, the failure probability of the conventional 6-Transistors (6T) SRAM cell increases considerably as the supply voltage is scaled down [16]. Read failure, due to the lack of Static Noise Margin (SNM), is one of the major failure factors, limiting the efficiency of dynamic voltage scaling. The usage of more reliable SRAM bit-cells, such as 8-Transistors (8T) or 10-Transistors (10T) cells, allows scaling to lower supply voltage, however, such solutions incur in large area penalties (at least 30% overhead for 8T compared to 6T bit-cells [19]).

In the context of CS algorithm, the reliable memory footprint requirement greatly varies according to the different phases of the execution: the *sensing* phase requires the system enough memory to store the sampled data, while the *compressing* phase has a bigger memory footprint to correctly access the data structures used for computation and temporary storage. A typical system performing CS on biomedical signals in real-time, spends most of the time in low workload phases (sensing), while a small portion of its time is spent in high workload phases (compression). In [46], where a single-core CS is implemented in real HW, the ratio between high workload and low workload phases is below 5%.

These considerations motivate the idea of the architecture presented in this chapter: using a hybrid memory architecture, combining classic 6T-SRAM cells with 8T-SRAM cells, we are able to offer reliable operation at lower supply voltage. In the sensing phase of the CS execution, the system works in a low-power state (600mV), where only the memory (8T) needed to store sampled data is active and reliable [89], while the other portion (6T) is idle. In this phase the 6T memory has enough hold SNM to be in data-retentive mode [16] though it cannot be correctly accessed. When compression is performed, the system increases its performance, operating at a higher voltage (1.2V) and the whole 6T/8T memory is active and reliable.

The concept of hybrid memory has already been introduced in literature [19, 28]. The work presented in [19] tolerates an error on the computation related

to the 6T memory when operating at low voltage, while in our architecture such behavior would compromise execution correctness. Moreover, their approach is highly customized for the specific application, avoiding the usage of standard memory compilers. In [28] authors propose a cache architecture with ways capable of operation at near-threshold voltage. The usage of separate voltage domains for cores, 6T and 8T cache ways has a non negligible overhead on the area, making it not feasible for scratchpad memories [57]. Our architecture can therefore benefit from using a single voltage domain, adapting its operating point to different workload scenarios.

The main contributions introduced in this chapter are the following:

- a novel hybrid memory architecture for ULP multi-core biomedical processors is proposed. The combination of 6T and 8T-SRAM banks enables aggressive power management during workload phases with low memory usage and low computational requirements.
- the proposed architecture leads to a significant improvement in energy saving ($\approx 25\%$ in a typical scenario) when compared to a standard architecture that uses solely 6T-SRAM banks.
- we demonstrate that our solution has a negligible area overhead ($\approx 2\%$) with respect to the baseline solution making it preferable to a solution with only 8T-SRAM due to its higher area overhead.

The rest of the chapter is organized as follows. In Section 5.2 the baseline architecture is introduced. Section 5.3 discusses the main features of CS algorithm and execution and describes the proposed hybrid memory architecture for ULP biomedical processors. Next, in Section 5.4 we describe the experimental setup and the results of the comparative study of our architecture with the baseline in terms of energy efficiency and area overhead. Finally, the conclusions are presented in Section 5.5.

5.2 CS Architecture

We consider a baseline architecture similar to several current multi-core architectures targeting biomedical signals processors [26, 27]. The considered architecture, presented in Figure 5.1, features 8 Processing Elements (PEs) each one with a private Instruction Cache. The PEs do not have private data caches, therefore avoiding memory coherency overhead, while they all share a L1 multi-banked tightly coupled data memory (TCDM) acting as a shared data scratchpad memory. The TCDM has a number of ports equal to the number of banks to have concurrent access to different memory locations.

Intra-cluster communication is based on a high bandwidth logarithmic interconnect (LIC). It consists of a Mesh-of-Trees (MoT) interconnection network able to support single-cycle communication between PEs and memory banks (MBs), resembling the hardware module presented in [77]. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates the accesses. To ease the negative impact of banking conflicts we consider a banking factor of 2 (16 banks). Moreover, to reduce memory access time and increase shared memory throughput, PEs can benefit from the broadcast mechanism of the interconnect.

The DMA shown in Figure 5.1 is in charge of periodically moving the data

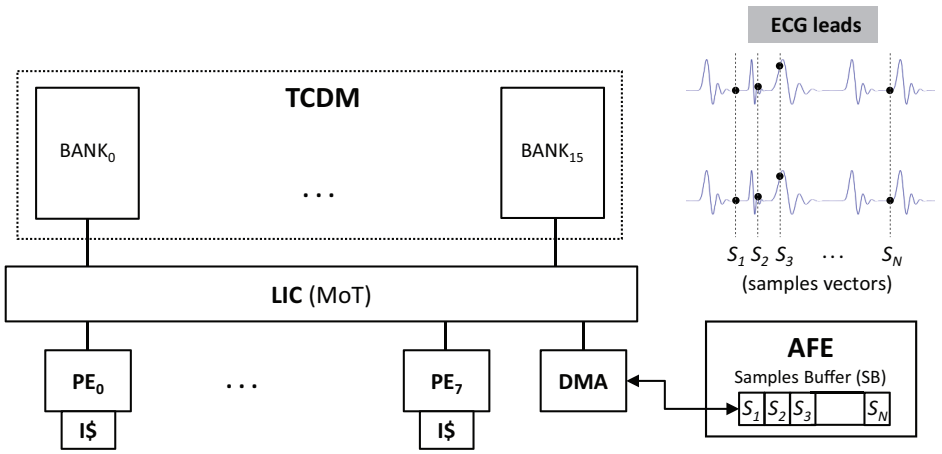

Figure 5.1: Baseline multi-core architecture for CS

sampled by the analog front-end (AFE) buffer to the TCDM making it available to the multi-core processor to perform compression.

5.3 Hybrid Memory Architecture

In this section the baseline multi-core ULP architecture to perform Compressed Sensing (CS) on biomedical signals is presented. We introduce then the CS phases with a qualitative analysis on their characteristics in terms of memory footprint and processing requirements. Finally the proposed memory architecture is presented.

5.3.1 Compressed-Sensing Application

Typical WBSNs-based biomedical applications require to sense biological signals from the patient (i.e. ECG, EMG, EEG) and send them to a more powerful computing node for further analysis. The recently-developed Compressed Sensing (CS) theory states that sparse (and thus compressible) signals can be reconstructed from a smaller number of samples than required by Nyquist sampling frequency [58]. By deploying this sparsity property, which applies to many classes of biomedical signals, the CS paradigm can be suitable for implementing low-resource sensor applications [78], since it reduces the amount of samples required in processing and storage.

In the hereby considered CS architecture, the input multi-channel signal is sampled by the analog front-end (AFE), with a sampling frequency (f_s) according to the dynamics of the signal to analyze and the accuracy needed. The samples (s_i), corresponding to different leads, are stored in a buffer inside the AFE. Once the values are sampled, the DMA is triggered to move the samples from the buffer to the local memory of the CS multi-core processor. Then CS compression algorithm starts, where each core operates on its own subset of the sampled data. We assume that the computation phase must be completed before the first sample of the next window ($N + 1$) is available to avoid double buffering overhead.

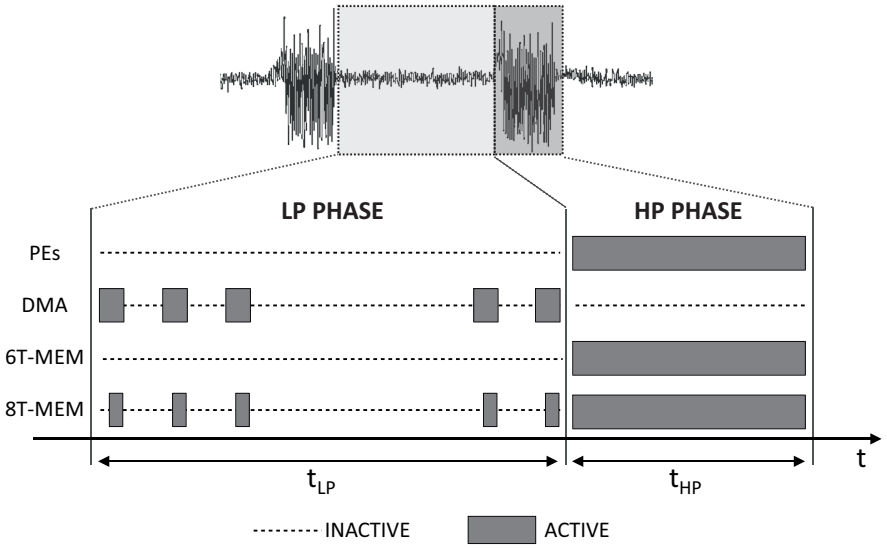

Figure 5.2: Active/inactive architectural elements during CS execution (LP and HP phases)

Such CS application, similarly to other sensor-data based computation, is composed of two phases: *data collection* and *computation*. The first phase is characterized by low-workload/low-memory requirements and a long duration, thus it will be referred as *LP Phase* (Low Performance). The latter instead will be named *HP Phase* (High Performance). This concept is depicted in Figure 5.2 where data collection and computation are shown.

Data Collection (LP Phase)

During the data collection phase the ULP processor waits for the number of samples (N) required to perform CS computation. Considering typical sampling frequencies for biomedical signals, this phase exceeds in time the phase of computation. For instance, with $f_s = 250Hz$ and $N = 512$, the data collection phase lasts 2048 ms. During data collection the only requirement for the architecture is to make available enough memory to store locally the data sampled by the AFE. It is clear that during this phase for most of the time the system is idle thus requiring a ultra-low power state to avoid unnecessary consumption. Figure 5.2 shows a timing diagram of the status of the architectural elements during the LP

phase. The only active elements are the DMA and the portion of the TCDM memory where samples are moved for future elaboration. The required active memory, varies according to system specification (sampling frequency, compression algorithm).

Computation (HP Phase)

Once the data collection phase is over, the DMA has already copied the buffer with N samples to the local (TCDM) memory and the computation phase starts. As introduced before the considered architecture performs a burst of computation on the available data for future transmission. During this phase the system is in an operating point characterized by high workload requirements and high memory footprint. All the processing elements are active and working on the data sampled during the last observation window. The amount of active memory required in HP phase is higher then in LP Phase because of all data structures needed to perform the convolution kernel of the CS algorithm (Section 5.4.1). Moreover, considering that the compression kernel is memory-bound by nature, the bandwidth requirements in core-memory bandwidth implies higher supply voltage for the memory in order to sustain the throughput.

5.3.2 6T/8T Hybrid Architecture

Considering the limitation imposed by classic 6T-SRAM memory when operating aggressive voltage scaling and the characteristics of biomedical applications, as outlined in the previous section, we consider an alternative memory architecture. By combining 6T and 8T-banks the reliable operating range is further extended to lower supply voltage. The proposed 6T/8T hybrid architecture is schematized in Figure 5.3 and compared to the baseline architecture introduced in Section 5.2, it features:

- single voltage domain for the whole architecture. This reduces area overheads and design complexity.
- 8T portion of the TCDM (*LP memory*) able to offer reliable operation down to 600mV.
- 6T portion of the TCDM with reliable access down to 800mV but able to operate in data retentive mode (sufficient hold SNM) at 600mV.
- at voltages higher than 800mV all the TCDM (6T + 8T) operates correctly (*HP memory*).

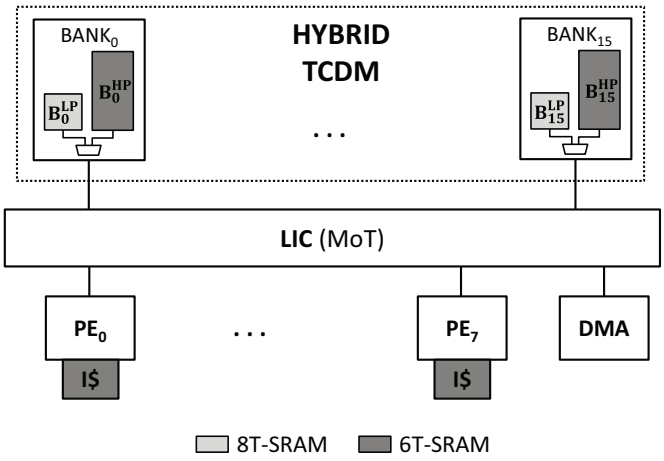

Figure 5.3: Hybrid 6T/8T memory architecture

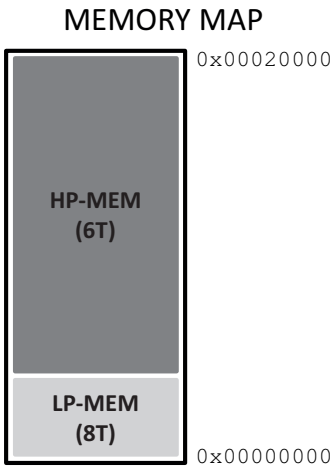

Figure 5.4: Contiguous memory map of hybrid 6T/8T memory

- the interleaving on different banks operated by the logarithmic interconnect (Section 5.2) enables to have a contiguous memory map among the 6T and 8T portions. This concept is depicted in Figure 5.4.

5.4 Experimental Setup and Results

In this section we present the experimental setup and the results of the evaluation of the proposed hybrid memory architecture in terms of energy efficiency and area overhead.

5.4.1 CS Algorithm Analysis

The reference benchmark is a real-time multi-lead ECG processing application composed of two main kernels: *Compressed Sensing* (CS) and *Huffman Coding* (HC). The CS kernel [58] performs compression (50% ratio) on a block of 512 samples of ECG data per lead with a sampling period of 4ms. The HC kernel performs the Huffman encoding on the compressed data, reducing its footprint further for wireless transmission [58]. The CS algorithm operates on 8 leads in parallel where each Processing Element (PE) works on a separate lead data-set.

The CS part has a constant program flow without any dependence on the input data, while the HC part adds a short section of data-dependent program flow. Considering a single lead ECG, the memory footprint of the CS algorithm consists of 648 bytes for instructions and 16 KB for data. The data section consists of two contributions: working data (the samples) and read-only data with a memory footprint of 2048 and 14336 bytes respectively. More in detail the read-only data consists of 3 Look-Up Tables (LUTs), i.e. a vector of random coefficients for the CS kernel (12288 bytes) and two data dependent LUTs (1024 bytes each) for the HC kernel.

Such CS algorithm analysis was used at design time to choose the appropriate memory cuts, for both baseline and hybrid architectures, and to statically allocate the memory structures. The TCDM size is assumed to be 128KB in both architectures, while an instruction cache of 1KB (per-core) is chosen. Considering that during compression every core operates on 512 samples, the 8T-SRAM memory (where sampled data is stored during LP phase) is chosen to be 16KB with 16 banks of 1KB each.

5.4.2 Hybrid Memory Analysis

Table 5.1 shows the power numbers (dynamic and leakage) considered for the evaluation of the proposed architecture. For 6T/8T memories the power values were extracted from the data-sheets of the respective SRAM architectures for a low power 65nm technology library. The memory numbers reported here refer to 1024x32 bits arrays (mux column = 4). The idle power is the standby power of the SRAM, where only the clock and address pins are toggling. Write and read power were measured with 100% activity (back to back cycling), with half of the address and data inputs (only for write) toggling. All inputs are stable (no toggling) for deriving the leakage power. We further assumed the worst case for leakage (i.e. best case for the technology). 8T cells considered here are Low-Leakage (LL) cells, a register-file architecture, which offer better performance and reliability. On the other hand, for the 6T-SRAM, the LL cells incur in reliability problems when reducing the supply voltage to 600mV [16].

Table 5.1: 6T/8T memories and PE energy numbers

	DYNAMIC [µW/MHz]					
	6T-MEM		8T-MEM		PE	
	HP	LP	HP	LP	HP	LP
IDLE	2.20	0.54	2.32	0.56		
READ	11.79	2.87	12.04	2.93	68.76	16.74
WRITE	13.88	3.38	14.11	3.43		
	LEAKAGE [µW]					
	6T-MEM		8T-MEM		PE	
	HP	LP	HP	LP	HP	LP
-40 C	0.61	0.31	0.27	0.13	0.63	0.32
25 C	11.56	5.89	5.35	2.63	11.18	5.69
125 C	326.77	166.23	158.77	80.77	338.44	172.17

For the Processing Element (PE), we considered an average active energy of 68.76 µW/MHz and 16.74 µW/MHz when operating at 1.2V and 0.6V, respectively. These numbers are based on post-synthesis characterization of an openRISC core. For the DMA and the logarithmic interconnect our characterization estimates 63.13 µW/MHz and 54.73 µW/MHz respectively at 1.2V as average active energy (15.37 µW/MHz and 13.13 µW/MHz, respectively, at 0.6V). Comparing the number of NAND equivalent gates of the DMA and the 8x16 interconnect with respect to a single PE, we derived corrective factors for the leakage power equals to 0.92x and 2.19x, respectively. Leakage power is scaled to 0.6V considering the relation expressed in [79].

5.4.3 Area Overhead

To evaluate the area overhead of our solution, in an *iso-size* comparison, we quantified the overhead introduced by the 8T memory portion in the hybrid architecture compared to the baseline (6T-only) solution. Table 5.2 shows the impact of each element on total area.

The overhead of extra-circuitry for the hybrid memory, required by the separation of logical banks in 6T and 8T banks is negligible, leading to a total

Table 5.2: Area comparison (Hybrid vs Baseline)

ELEMENT	HYBRID $[mm^2]$	BASELINE $[mm^2]$
PEs	0.85408	0.85408
6T TCDM	0.70652	0.80746
8T TCDM	0.13323	-
6T I$ (DATA)	-	0.05047
8T I$ (DATA)	0.06662	-
DMA	0.09801	0.09801
LOGINT 8X16	0.23348	0.23348
TOTAL	2.09194	2.04349

overhead below 2%. If instead we consider an architecture with only 8T-SRAM, the overhead on the overall system would be non negligible ($\approx 14\%$) and leakage contribution would affect the energy efficiency.

5.4.4 Hybrid Memory Efficiency

To evaluate the energy efficiency of the proposed architecture, the power numbers of Section 5.4.2 have been integrated in a SystemC-based cycle-accurate virtual platform [14]. The architecture was configured with 8 cores, 1 DMA, an 8x16 logarithmic interconnect and 6T/8T portions as determined in Section 5.4.1. The HP phase is performed in 94.56k clock cycle, while the LP phase takes 24.12k clock cycles (sum of all DMA data movements in an observation window).

HP Phase

The first set of experiments was aimed at comparing the energy efficiency of the proposed 6T/8T hybrid memory architecture to the baseline case of an ULP multi-core architecture where all the TCDM is composed of 6T-SRAM cells. During the HP phase, all cores are active and executing the CS kernels described in Section 5.4.1 operating in parallel on its separate data set. On the memory side, the whole TCDM memory is active, as well as the I-caches. The DMA is idle,

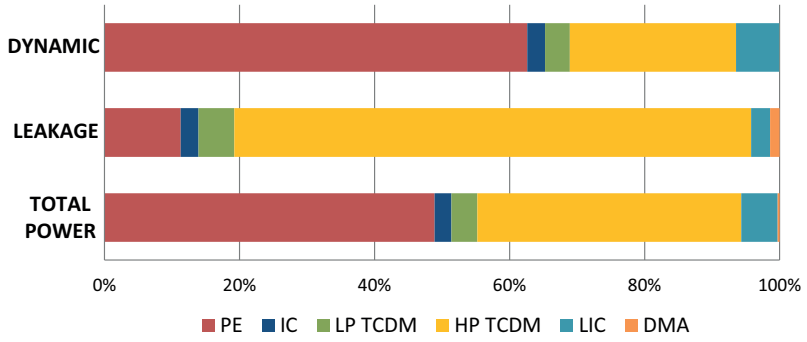

Figure 5.5: Power breakdown for HP phase (hybrid, T=25°C)

contributing only for leakage power. The operating point considered in this experiment is a clock frequency of 100 MHz and a supply voltage of 1.2V.

In Figure 5.5 a power breakdown for the hybrid architecture (T=25°C) is shown. Total power consumption has two main contributions: PEs and HP TCDM (6T-SRAM) as expected. The number of accesses in the HP portion of the TCDM exceeds the number of accesses in the LP portion, mainly due to data structures of the CS kernels and stack. For completeness a separated breakdown for dynamic and leakage is presented, though the dynamic power contributes for

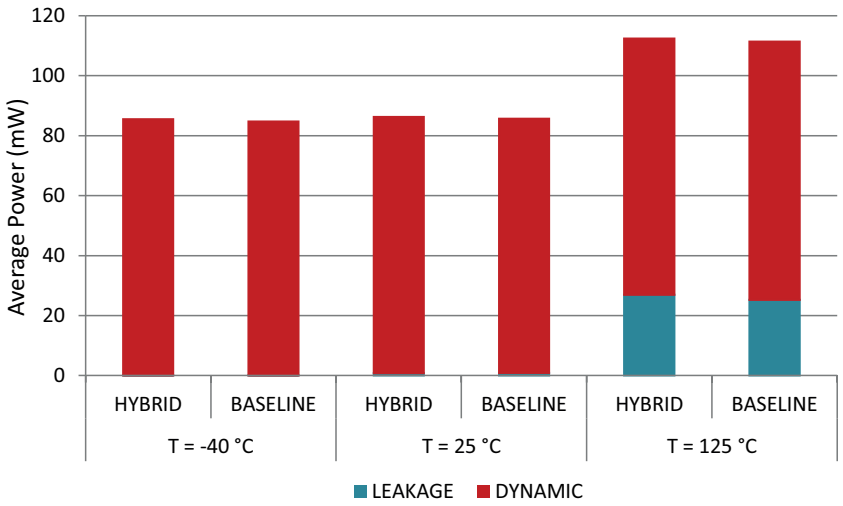

Figure 5.6: Power comparison for HP phase (baseline vs hybrid)

99% to total power.

Figure 5.6 shows the average power during HP phase for the baseline and the proposed architecture. At different temperature leakage contribution (exacerbated at T=125°C) impact both architectures, though the 8T Low Leakage (LL) cells can amortize this effect. As expected in the HP phase our solution has a lower energy efficiency compared to the baseline, mainly due to the higher contribution of dynamic power for the 8T-memory. The impact of the hybrid architecture in the HP phase is very low, being below 1% for all the considered temperatures.

LP Phase

As a second experiment we compared the energy efficiency of our solution and the baseline during the data collection phase. During the LP phase all cores are idle waiting for the sampled data to be ready. Only the amount of memory needed to store the samples is active, while the other portion of memory is clock gated, contributing only for leakage. The DMA is in charge of moving the sampled data from the AFE buffer to the LP-portion of the memory. The operating frequency considered in this phase is 10 MHz. For a fair comparison with the baseline, we consider only 16KB active of TCDM, with the other portion being clock-gated. Considering reliability issue for 6T-SRAM [16], the baseline has a supply voltage of $V_{dd} = 0.8V$, while our solution thanks to the higher reliability of 8T-SRAM

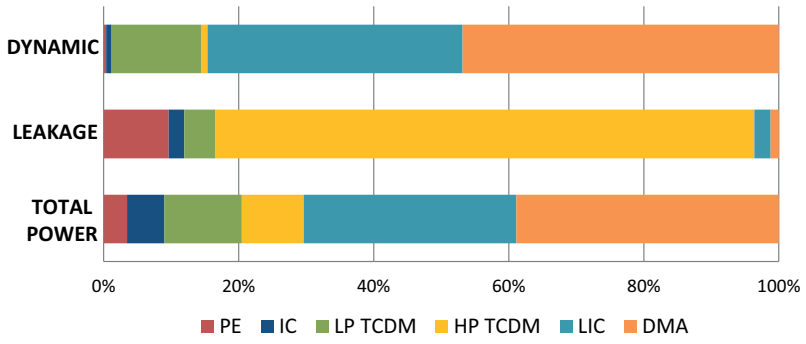

Figure 5.7: Power breakdown for LP phase (hybrid, T=25°C)

memory can operate at 0.6V.

For completeness, in Figure 5.7 is shown a breakdown of total power for the hybrid architecture at the temperature of 25°C.

Figure 5.8 shows the average power during the LP phase for the baseline and the proposed architecture at different temperatures. These results confirm the effectiveness of the proposed solution: thanks to the extended voltage scaling range offered by the reliability of 8T-SRAM the dynamic component can be greatly reduced. At T=25°C the overall reduction of power compared to the baseline is 24.5%.

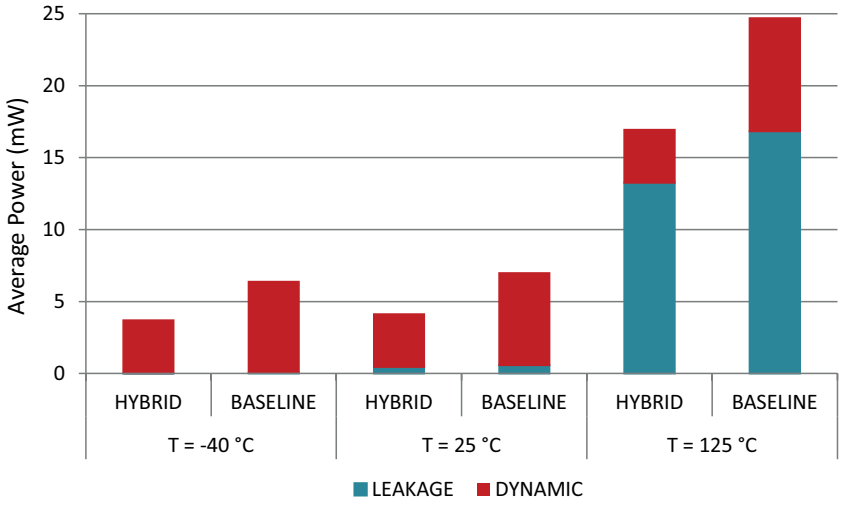

Figure 5.8: Power comparison for LP phase (baseline vs hybrid)

Overall Efficiency

The last set of experiments was intended to evaluate the efficiency of the 6T/8T hybrid memory architecture varying the amount of time spent in LP and HP phases. The average power consumption shown before demonstrates a good improvement for the proposed solution in the HP phase and a small penalty in the HP phase but is not taking into consideration the time spent in the two phases during a period of Compressed Sensing. The results of this analysis are presented in Figure 5.9, where on the x-axis is shown the ratio between HP and LP phases and on the y-axis is shown the energy efficiency of the hybrid architecture with respect to the baseline.

The proposed solution improves energy efficiency of the system for the range 0-90% of HP/LP ratio, with a crossing point at $\approx 90\%$ where the baseline architecture outperforms the hybrid solution. Considering a typical scenario with a 5% ratio between HP and LP phases [46], the proposed solution proves to be $\approx 25\%$ more efficient than the baseline architecture. This result is valid on the whole temperature range considered. The quadratic trend in efficiency validates the motivation behind our solution. Power consumption has a quadratic dependency

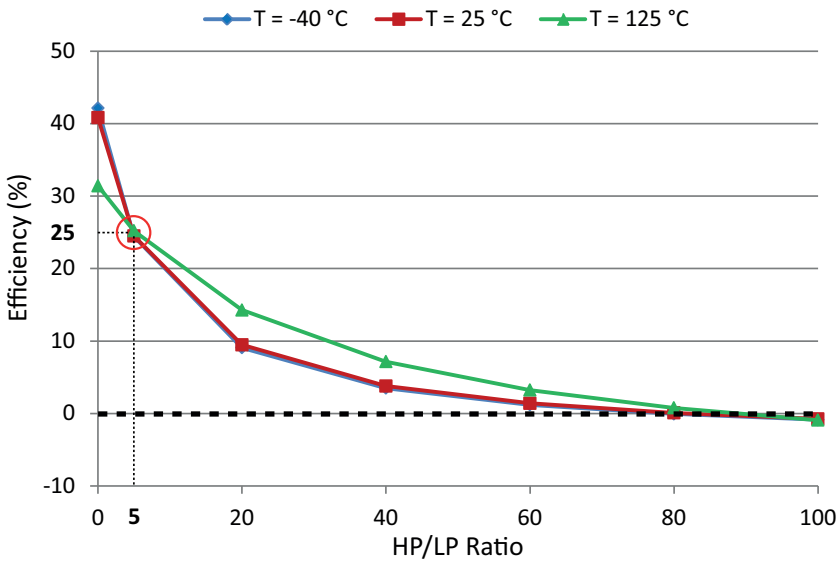

Figure 5.9: Hybrid vs Baseline efficiency varying HP/LP ratio

on supply voltage for the dynamic component and increasing the amount of time spent in LP phase, the more effective becomes the aggressive voltage scaling that can be operated on our hybrid architecture.

5.5 Conclusions

In this chapter we introduced a 6T/8T hybrid memory architecture for multi-core biomedical processors. Classic memory architectures composed of 6T-SRAM memories face reliability issues when reducing supply voltage to threshold. Static noise margin for such memory cells compromise execution correctness making aggressive voltage scaling not feasible. The proposed architecture greatly benefits from the varying workload/memory footprint requirements of biomedical processing, adapting in a reliable way to different operating points. Our solution offers significant improvements in energy saving ($\approx 25\%$ in a realistic scenario) when compared to a 6T-only architecture with a negligible ($\approx 2\%$) area overhead.

Future research directions comprise an extension to perform runtime data allocation and memory management. The OpenMP programming model offers, by means of `#pragmas`, an easy way for programmers to specify, based on workload requirements, in which memory portion allocate the data.

Chapter 6

Variation tolerance

Near-Threshold operation is today a key research area in Ultra-Low Power (ULP) computing, as it promises a major boost in energy efficiency compared to super-threshold computing and it mitigates thermal bottlenecks. Unfortunately near-threshold operation is plagued by greatly increased sensitivity to threshold voltage variations, such as those caused by ambient temperature fluctuation. In this chapter we propose an architectural scheme to tolerate ambient temperature-induced variations capable of statically (off-line) and dynamically (on-line) adapting the processor-to-L1-memory latency without compromising execution correctness. The resilient version of our target architecture has been tested in different scenarios, evaluating the different design trade-offs, showing the cost, performance and reliability gain compared to state-of-the-art static solutions.

6.1 Overview

The pace dictated by Moore's law has slowed down and classical CMOS scaling, which drove the semiconductor growth during the past several decades, is delivering reduced energy gains [13, 29, 47]. Beyond the 65nm technological node the supply voltage has remained essentially constant and improvements on dynamic energy efficiency have dramatically stagnated, while leakage currents continue to increase. In this "Moore's law twilight era", further energy

gain can be achieved by moving to the near-threshold computing (NTC) domain [13, 29, 31, 47, 49, 59, 96]. By reducing the supply voltage (V_{dd}) from the nominal value to the level of the threshold voltage (V_t) the energy per operation decreases by 10x with performance penalties of 10x [29, 59]. Reducing further the V_{dd} in the sub-threshold region is less attractive since the performance will drop by an additional 50-100x with an energy improvement of only 2x [95]. Although NTC provides excellent energy-frequency trade-offs, it faces three key barriers that must be overcome for widespread use: performance variation, performance loss and functional failure.

Systematic and random variations are already significant issues in today's advanced technological nodes and operating at low-voltages exacerbates the effects of both. Performance uncertainty in the near-threshold region due to the global process variation alone increases to 5x from 1.3x at nominal supply voltage [20, 29]. Operating at this voltage also heightens sensitivity to temperature and supply ripple, both can add another factor of 2x to the performance variation resulting in a total performance uncertainty increase of 20x. This issue cannot be tackled with the worst case design common practice, since taking margins with over-design will result in chips running way below of their potential performance, which is wasteful both in performance and in energy due to leakage current.

Another main issue with low-voltage operation is the potential performance loss, which can seriously limit the degree of use of voltage-scaling for a given processing requirement. Parallel computing using multiple cores can alleviate this issue, provided that the algorithms to be executed can be parallelized. The authors in [25] explored the power/performance trade-offs between sequential and parallel near-threshold computations for various biomedical signal processing requirements. They estimate a 34% [25] of energy loss in the single-core design when compared with the multi-core one under high workload requirements. To achieve the same throughput the single-core needs to operate with a V_{dd} twice higher than that of the multi-core solution. In [26] authors show that exploiting NTC in conjunction with multi-core architecture design enables ULP wearable health monitoring systems achieving up to 39.5% power savings with respect to

the state-of-the-art.

Also functional failure must be taken into account: more than the logic cells, embedded SRAM cells will suffer from static and random variations with the high risk of causing several functional failures. For instance, a typical 65nm SRAM cell has a failure probability of 10^{-7} at nominal voltage. However, at NTC this failure rate increases by 5 orders of magnitude to approximately 4%. Keeping the power supply of SRAM cells higher than the core logic will reduce the error rate [20, 29], the leakage power and produce faster memory therefore enabling single cycle latency L1 memory.

Variability constraints when operating in NTC push the architecture toward a topology in which several processing elements communicate with each other through a shared L1 memory system. Recently, several many-core architectures have been proposed that leverage tightly-coupled clusters as a building block [2, 11, 26, 82]. In a shared memory paradigm, these designs try to overcome the scalability limitations encountered when increasing the number of Processing Elements (PEs) that share a unique interconnection and memory system by creating a hierarchical design where PEs are clustered into small-medium sized subsystems. The small number of PEs makes it possible to design a low-latency interconnect between processors and L1 (in-cluster) memories, while scaling to larger system sizes is enabled by replicating clusters and a scalable medium like a Network-on-Chip.

Putting variability in the picture, in such chip multiprocessor architectures the interconnect clearly becomes a single point of failure. Authors in [42, 43] introduced a resilient single-cycle interconnection network, based on pipeline stages [30], that can statically (boot-time) tolerate delay variations due to aging or static variations which is based on a fully combinational Mesh-of-Tree (MoT) interconnection network proposed in [77] suitable for tightly-coupled processor clusters.

Since ULP devices operating at near-threshold voltages due to the low power dissipated are safe from self-heating effects, die temperature is hot-spot free and mainly follows ambient temperature [17, 18, 29] which can greatly vary (daily/seasonal fluctuations, indoor/outdoor transitions). As a consequence of this,

performance variability cannot be effectively addressed only by adopting static solutions, requiring lightweight solutions reactive to dynamic variations that can lead to functional failure when ambient temperature significantly changes.

In this chapter we this issue is tackled by introducing an architectural scheme to achieve resiliency to critical path variations induced by ambient temperature fluctuations. This is done by exploiting the resilient logarithmic interconnect presented in [42, 43] and integrating it with a set of new HW modules capable of sensing the current ambient temperature, recognize possible hazards, checking memory and link consistency and react by reconfiguring, through a SW procedure, the interconnect delays.

The rest of this chapter is organized as follows. In Section 6.2 the baseline target architecture is introduced. Section 6.3 discusses in detail the proposed solutions (both static and dynamic approaches) with details on the building blocks of the schemes as well as their working principle. Next, in Section 6.4 we describe the experimental setup and the simulation framework used to compare the proposed schemes with state-of-the art static solutions. Finally, the conclusions are presented in Section 6.5.

6.2 Baseline Architecture

The recent shift towards many-core architectures brings new architectural paradigms: today several academic and commercial many-cores architectures deploy a hierarchical design where processing elements are organized into small-medium sized tightly-coupled clusters. We chose as a target cluster architecture one similar to [11]. Our shared L1 cluster, shown in Figure 6.1, features 16 Processing Elements (PEs) each one with a private Instruction Cache.

The PEs do not have private data caches or memories, therefore avoiding memory coherency overhead. They all share a L1 multi-banked tightly coupled data memory (TCDM) acting as a shared data scratchpad memory, not as a data cache. Intra-cluster communication is based on a low-latency high bandwidth *Logarithmic Interconnect* (LIC). It consists of a Mesh-of-Trees (MoT) intercon-

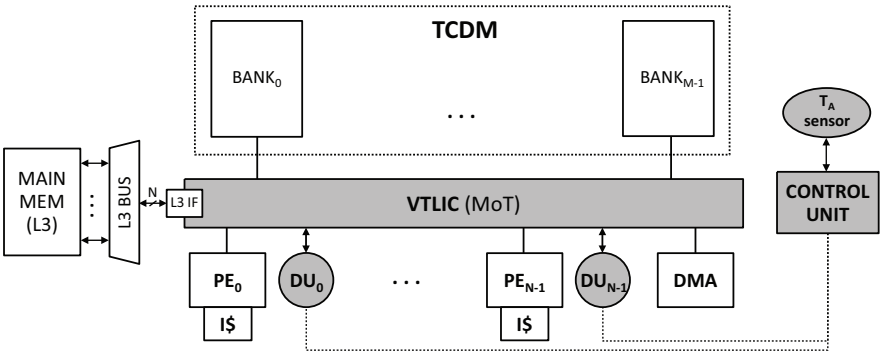

Figure 6.1: Target architecture: baseline and variation tolerant version extension

nection network (Figure 6.2) able to support single-cycle communication between processors and memories, resembling the hardware module proposed in [77]. As shown in Figure 6.2, the MoT network connects $N = 2n$ PEs and $M = 2m$ Memory Banks (MBs). It contains $Log_2(M)$ levels of routing primitives and $Log_2(N)$ levels of arbitration primitives.

The interconnect operates word-level address interleaving on the memory

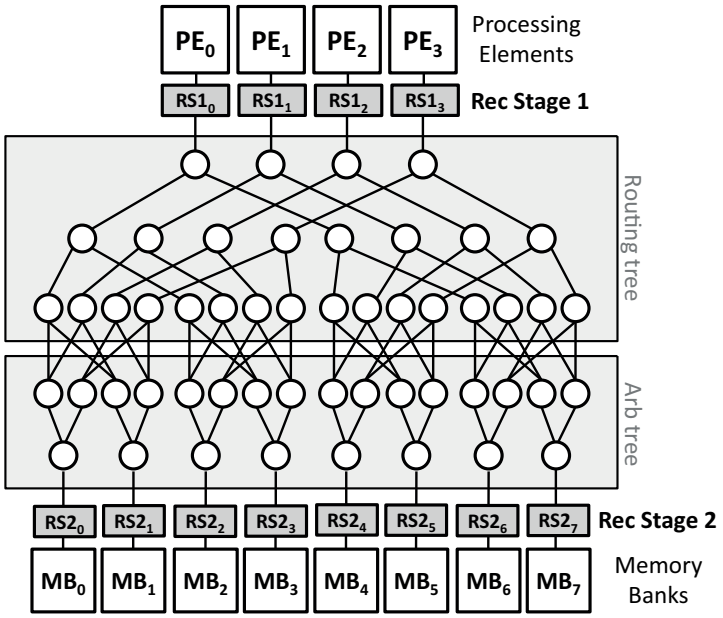

Figure 6.2: Mesh-of-Trees interconnection network (4 cores and 8 memory banks) with 2 reconfigurable stages per link

banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The LSBs of the address field determine the routing path to the destination. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates the access and a higher number of cycles is needed depending on the number of conflicting requests. In case of no banking conflicts data routing is done in parallel for each PE, thus enabling a sustainable full bandwidth for PEs-memories communication.

The TCDM has a number of memory ports equal to the number of banks to have concurrent access to different memory locations. Once a read or write requests is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to a total latency of two clock cycles for conflict-free TCDM accesses. Main memory (where program and global data are stored) is accessible by PEs through explicit data access or by Instruction Caches logic to perform instruction refills.

To make this architecture tolerant to ambient temperature induced variability, we propose a solution whose building blocks (highlighted blocks in Figure 6.1) and their interaction will be described in the next section.

6.3 Temperature Tolerant Scheme

In this section we present all the architectural elements and how they interact to offer both *static* (off-line) and *dynamic* (on-line) tolerance to ambient temperature induced variations. The dynamic variant is an architectural extension of the baseline static scheme. Both solutions consist of different phases: the *detection* of a possible hazard, the *reconfiguration* of the interconnect to tackle variations and a *control policy*. This is obtained by means of the basic building blocks hereby described: *Variation Tolerant Interconnection*, *Detection Units* and *Control Unit*. Both static and dynamic architectures deploy the same HW version of the interconnect, while they differentiate in the detection phase carried out by the Detection Units and in the behavior of the Control Unit.

6.3.1 Variation Tolerant Interconnection

To tackle ambient temperature-induced delay variations, we exploit the logarithmic interconnect introduced in [42] to perform static and dynamic reconfiguration. The approach uses two reconfigurable modules, based on Flip-Flops (FFs), to be used respectively, on the request and response paths as shown in Figure 6.3. As

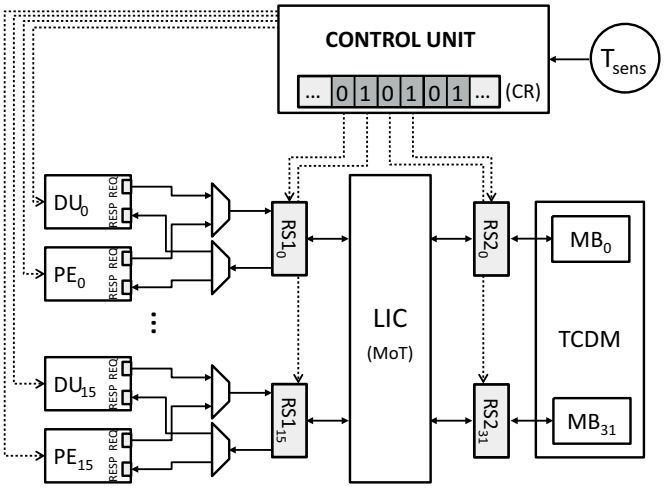

Figure 6.3: Architecture overview: processors-memory paths (16 cores and 32 banks) with reconfigurable stages, detection units and control unit

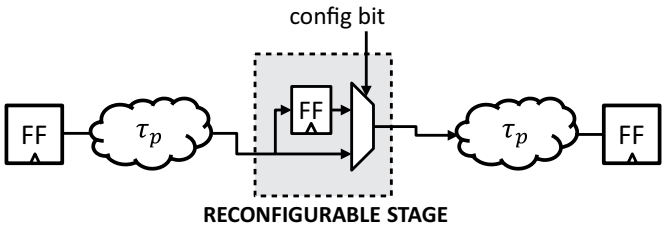

Figure 6.4: Reconfigurable pipeline stage: when the flip-flop (FF) is in the path an extra cycle of latency is inserted

stated by authors, the most critical paths are between processors and the TCDM. They also found that the delay of the paths whose sources and targets are inside the processors is almost 2/3 of that of the critical path (PEs to TCDM). If any variation occurs, those paths that are inside the processors have reasonable margins to tolerate it, thus, we have to take care of the paths between processors and memories.

The FFs can compensate the effect of delay variations by inserting one extra cycle of latency each (up to two cycles per processor-memory path), thus relaxing timing constraints. This operating mode is referred to as *pipeline mode* and is depicted in Figure 6.4. Without delay variations a memory access is completed without increasing the latency. The main property of the variation tolerant interconnect is the capability of introducing up to two cycles for static or dynamic variations for each processor-memory path independently. Since the PEs supports only blocking accesses to the memory, therefore the messages traveling between PEs and TCDM are intrinsically not subject to out-of-order issues.

Every PE has associated one HW Detection Unit (DU) which can detect faults when accessing memory similar to the one in [42]. Detection can be performed both off-line and on-line as described later. If the tester does not find any timing error, the reconfigurable pipeline reconfigures itself in such a way that the processor-memory path becomes fully combinational by selecting the second input of the multiplexer (Figure 6.4) with the FFs out of the path. If variation happens, the FFs can be activated/deactivated to adapt communication latency in processor-memory paths. The interconnect configuration can be stored in a structure where the bit fields drive the multiplexers in the reconfigurable stages.

6.3.2 Detection Units

This module is replicated for each PE and has a key role in the detection phase. In both static and dynamic solutions hereby presented, the Detection Units (DUs) perform a detection of possible failures when accessing the L1 memory, thoroughly scanning all the routing paths of the logarithmic interconnection. A DU implements two similar state machines which generate data and address for the write and read transactions and verify the incoming data from memories. During their activity, all the detected faults are signaled to the CU which in turn manages the information to perform delay insertions to avoid timing errors. There is a slight difference between the static and dynamic version of the DUs as will be explained in Section 6.3.4.

6.3.3 Control Unit

The Control Unit (CU) has a centralized role of control and coordination for detection phase and interconnection reconfiguration. We present hereby both static and dynamic versions of the CU.

Static Control Unit

The static version of the Control Unit coordinates a *one-time* detection and configuration of the interconnection at *boot-time*. Figure 6.3 shows a schematic view of the interaction between the CU and the other architectural elements. When a detection phase starts at boot-time, the CU is in charge of freezing PEs, trigger the self-tuning routine and reconfiguring the Reconfigurable Stages (RSs) updating the content of the Configuration Register (CR) whose bit fields drive multiplexers selectors.

The functionality of the CU is based on:

- *Configuration Register*: such register stores the *configuration information* of the resilient interconnect, i.e. the configuration bits for the RSs distributed in the architecture. Considering N cores, M memory banks and 2 RSs per link, the memory footprint of this register is 2·N·M bits.

- *Ultrasafe Configuration*: this configuration guarantees execution correctness in the worst-case operating point considering PVT variations. It is the initial configuration loaded at boot-time before detection will be performed. This delays configuration is determined at manufacturing time, hard-coded at design time, loaded at boot time and represents the delays configuration that avoids timing faults. It is a sort of rollback mechanism, since no information is available on the current operating point and interconnect delays margins are unknown, this configuration is the only one that ensures zero timing faults when the system starts but at the same time is the slowest possible.

Dynamic Control Unit

In the dynamic version of the proposed architecture, the Control Unit (CU) has a role of on-line control and coordination of the other HW components: sensor readout, dynamic interconnect reconfiguration and control policy. The CU is in charge of periodically sensing the ambient temperature sensor, this is done by hysteresis thresholding to avoid spurious detection activities due to minor temperature oscillations as well as filtering potential sensor readout noise. When the CU detects a potential hazard, PEs execution is frozen and, if necessary, the CU reconfigures the interconnect updating the content of the Configuration Register (CR) whose bit fields drive the reconfigurable stages of the interconnect.

To perform on-line monitoring (details in Section 6.3.6) of the current operating point (T°) and reconfigure the interconnect delays accordingly, the dynamic CU further extends the static version by means of:

- *Threshold Temperatures*: these temperatures (T_i^{th}), stored in a 8-bit memory structure within the Control Unit, consist of all the threshold temperatures that will trigger the Monitoring Algorithm for a potential hazard. The thresholds can be determined at design time based on static timing violations analysis for different operating points (T, V) or as an arbitrarily fine grid of temperatures in the operating range. According to the relative Enabled bit (E_i), a tag for currently enabled thresholds, the set of

temperatures can be refined during execution thus limiting the overhead of the dynamic solution. The enabled temperatures serve as indexes for the Reconfiguration Cache.

- *Reconfiguration Cache*: this memory is intended to store the delays configuration bits (DATA) associated to different threshold temperatures (INDEX). The structure of the reconfiguration cache is shown in Figure 6.5. Every entry consists of the configuration bits for the reconfigurable stages (flip flops) of the possible N·M PE-memory paths. Since every path can have up to 2 reconfigurable stages in pipeline mode, an entry needs 2·N·M bits to store the information. For instance, considering N = 16 PEs and M = 32 banks, entry size is 128B. The Reconfiguration Cache implements a LRU eviction policy. The usage of the Enabled bits for threshold temperatures combined with the caching mechanism allows a learning procedure that reduces the runtime overhead.

It is important to outline here the mechanism used by the Control Unit to search entries in the Reconfiguration Cache. A given configuration is safe (zero timing faults) in a specific temperature range according to the thermal behavior of the system. As will be described in Section 6.4.1, we consider two different thermal behaviors for temperature induced variations, namely *Thermal Inversion* (TI) and *Non Thermal Inversion* (NTI).

When considering TI, at higher temperatures the critical path gets faster while the opposite holds for NTI. This affects which "safe" entry should be looked-up in the Reconfiguration Cache when a hazard is detected. The concept is depicted

	PE_0					PE_{N-1}			
	B_0	B_1		B_{M-1}		B_0	B_1		B_{M-1}
T_0	0	1	···	1	···	1	0	···	1
			⋮					⋮	
T_{H-1}	1	2	···	1	···	1	1	···	1
US	2	1	···	2	···	2	2	···	2

Figure 6.5: Reconfiguration Cache structure

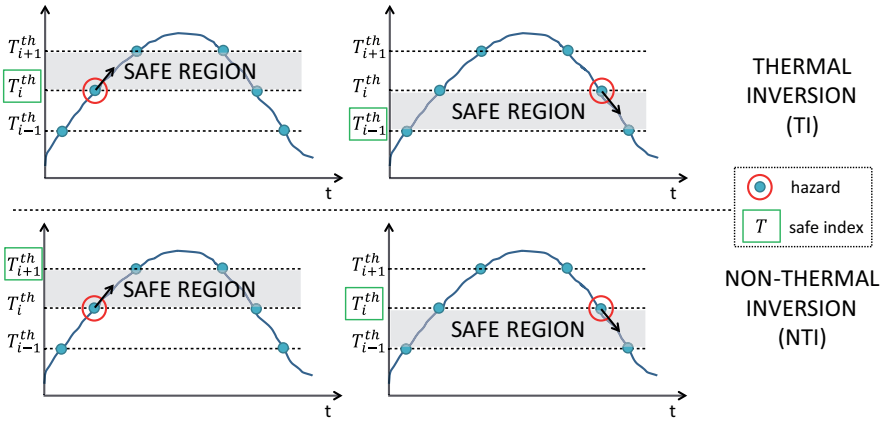

Figure 6.6: Safe index in Reconfiguration Cache in *TI* and *NTI*

in Figure 6.6, where the hazard detected is always related to threshold T_i^{th} but according to the thermal behavior and the temperature trend (arrows) the safe index to be looked-up in cache varies.

6.3.4 Detection

To determine the optimal delays configuration for the current operating point, our dedicated HW testing modules are in charge of detecting faults in L1 memory accesses. The complete test is performed during M phases where M is the number of memory banks. During each phase all DUs write the test pattern a given bank and then read and verify them. The test pattern (data and address) contains one W/R sequence. The data that is written is `01010....01` (TEST). The pseudo-code of the Detection Phase is shown in Listing 6.1.

Each DU writes data to the specified address of each bank and then reads from the same location. Exploiting word interleaving it is possible to have each DU accessing a different bank reducing bank conflicts and detection cost. If the DU reads the wrong data, it means the write or read operations or both do not work correctly due to delay variations and the DU signals the path to the CU for delay insertion. The interaction between DU and CU is repeated until the delays inserted are compensating the variations up to a maximum of two cycles.

```
DU::DETECTION(addr_range):
    foreach addr in addr_range
      curr = read(addr)        /*dynamic only*/
      write(TEST,addr)
      if(read(addr) != TEST)
        mark_inc[addr] = true
        CU::DELAY_RECONFIG()
      else
        write(curr,addr)       /*dynamic only*/

CU::DELAY_RECONFIG:
    foreach addr in mark_inc
      if(delay(path) < 2)
        delay(path)++;
      else
        shutdown(PE(i))

    DU::DETECTION(mark_inc)
```

Listing 6.1: Pseudo-code of Detection (`DETECTION`) and Reconfiguration (`DELAY_RECONFIG`) for DU and CU

If the maximum number of two extra cycles is already reached (both reconfigurable stages are in pipeline mode), our baseline policy is considering the PE as faulty and shut it down.

The address range on which we operate faults detection is the whole TCDM address range since we do not know a priori which memory location can be more affected by temperature-induced variations. When performing an *online detection* we must preserve memory content. As shown in Listing 6.1, to have a non-destructive diagnosis we save and restore current value as first and last step of our detection procedure. On the other hand for the static solution, the detection phase is carried *off-line* out at boot-time and there is no need of preserving memory content since it is not set when the system starts. The test performed at boot time takes into account static delay variations due to random and systematic process variations or aging effects.

6.3.5 Reconfiguration

If during the detection phase a faulty PE-memory path is detected, the DUs notify the CU the marked addresses, updating the relative Configuration Register bit-fields, in order to perform the reconfiguration. Suppose DU(i), associated to PE(i), detects an error accessing a given address physically located in memory bank MB(j) of the TCDM, i and j are signaled to CU. CU checks the relative bit-fields of the Configuration Register CR(i,j) and if the maximum number (2) of reconfigurable stages is not reached, it operates a delay insertion. Otherwise PE(i) is considered a faulty processor and an interrupt signal to freeze PE(i) is raised until the end of program execution thus shutting down the core.

Reconfiguration cost is negligible with respect to detection since it barely consists of driving multiplexer selectors according to Configuration Register content.

6.3.6 Working Principle

All the aforementioned blocks cooperate to provide static or dynamic tolerance to temperature-induced delay variations that may lead to functional failure.

Static Tolerance (off-line)

The baseline static version of the architecture operates a one-time action to detect potential timing failures and to reconfigure interconnect such as to void them. Modules interaction occurs as shown in Figure 6.7. At boot time (t=0) the Static Control Unit (SCU) loads the ULTRASAFE configuration described in Section

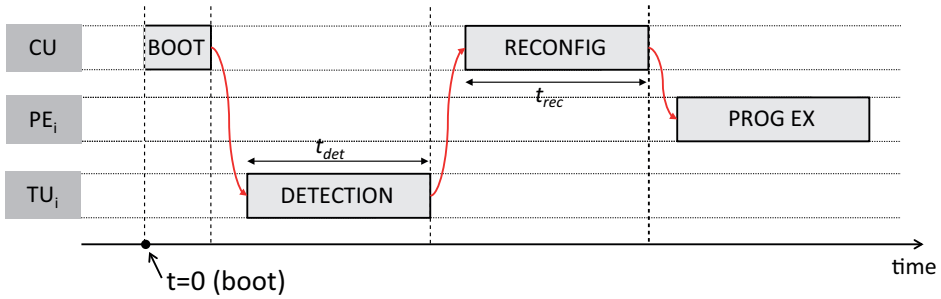

Figure 6.7: Block scheme and timing diagram of static solution

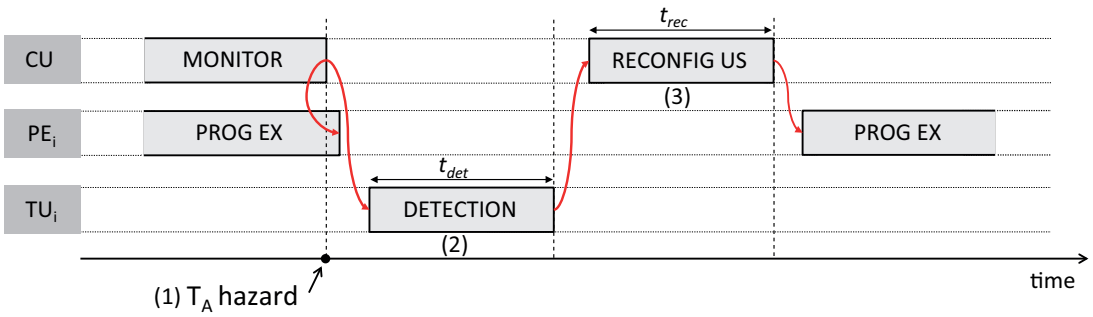

Figure 6.8: Block scheme and timing diagram of dynamic solution (MISS)

6.3.3, this configuration guarantees no errors in cores-memory communication. All Processing Elements are stalled in this early stage with fetch enabled signal forced to low. The SCU triggers the detection procedure as explained above and once the DUs have notified the delays to be inserted, the interconnect reconfiguration takes place. The configuration loaded at this point will not change until the device is shut down.

Dynamic Tolerance (on-line)

With reference to Figures 6.8 and 6.9, modules interaction occurs as follows: when the the Dynamic Control Unit (DCU) is triggered by an enabled threshold temperature (1), all PEs are forced to idle mode and, in case of *MISS* in the Reconfiguration Cache (Figure 6.8), all the DUs are simultaneously triggered to perform the diagnosis (2). After detection is carried out, the delay configuration

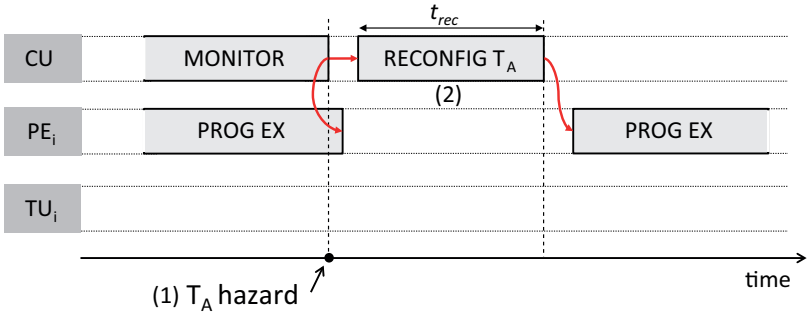

Figure 6.9: Block scheme and timing diagram of dynamic solution (HIT)

associated to current temperature threshold is stored in the cache and the DCU performs the dynamic reconfiguration (3) of the interconnect loading the ULTRASAFE configuration in the *Configuration Register*. When PEs are resumed from idle mode, program execution continues as before with degraded performance due to the safe configuration loaded. In case of a *HIT* in Reconfiguration Cache (Figure 6.9) the DCU skips the unnecessary detection for current threshold temperature and directly reconfigures the interconnect loading the cached configuration in the Configuration Register. According to current operating point and thermal behavior, when PEs are resumed from idle mode, program execution continues as before with the new delays configuration for current temperature.

The DCU is in charge of performing the Monitoring Algorithm and its flow is represented in Figure 6.10. Once the ambient temperature T_a is read from the

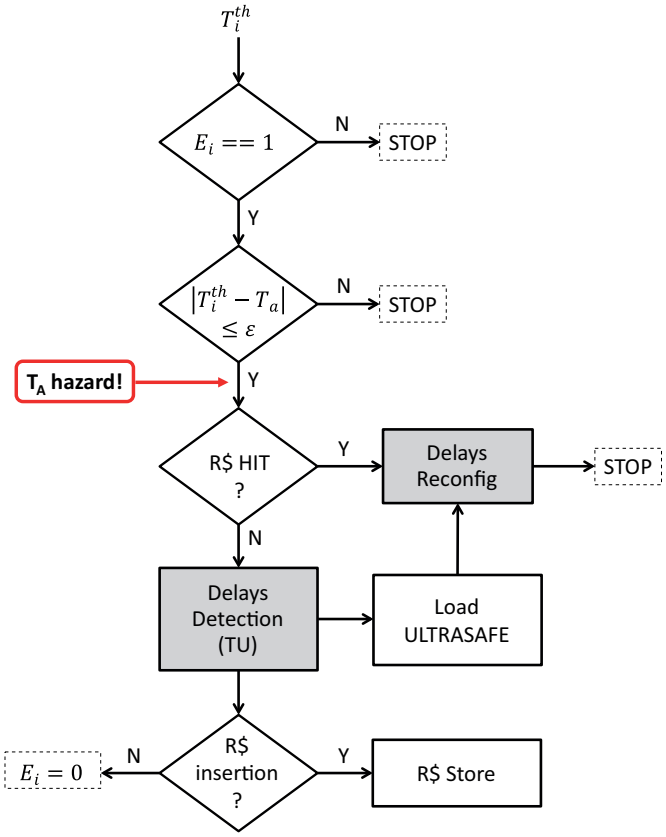

Figure 6.10: Online Monitoring Algorithm (Dynamic Control Unit)

sensor, for each enabled ($E_i == 1$) threshold temperature T_i^{th}, the DCU checks if T_a is close to T_i^{th} ($|T_i^{th} - T_a| \leq \epsilon$). If a hazard is detected, the DCU looks up that particular entry in the Reconfiguration Cache. A certain delays pattern may be already stored in the Reconfiguration Cache because the system has already been at current T_a leading to a previous detection phase. As explained before, the index to be looked-up in the cache depends on the thermal behavior. In case of a *HIT* in Cache, the configuration is directly used by the DCU to reconfigure the interconnection and adapt safely to current operating point (see next paragraph). In case of *MISS* in Reconfiguration Cache, DUs are triggered to perform Delays Detection and ULTRASAFE configuration is loaded right after. If the configuration detected by the DUs passes the insertion test it is stored in the Reconfiguration Cache, otherwise the threshold T_i^{th} is disabled ($E_i = 0$) to not trigger further undesired detections. The R$ Insertion Test (Figure 6.10) test checks if current configuration is different from both the current configuration loaded and the last entry added in Cache. This mechanism enables the DCU to learn which threshold temperatures are significant thus reducing at runtime detection overhead.

6.4 Evaluation

In this section the temperature induced variation modeling is presented as well as the simulation infrastructure. Finally tests and performance evaluation are described.

6.4.1 Modeling

In this section we describe the mathematical modeling of static delay variations and the temperature effect on the critical paths distribution.

Static Delay Variations

Static variations due to aging, random and systematic variations lead to a normal distribution $N(\mu, \sigma)$ of the critical path delay [29], shown in Figure 6.11.

In our experiments we take as nominal critical path delay for the variation tolerant logarithmic interconnect $\tau_d = 4ns$ (corner plus 3σ safety margins) as described in [42] and a critical delay variation, for near-threshold operating circuits, characterized by $\sigma_{\tau_d} = 15.1\%$ as stated in [63]. Without loss of generality, we consider a single realization (Figure 6.11) of the Probability Distribution Function (PDF) with a critical path delay hereafter referred as $\overline{\tau_d}$.

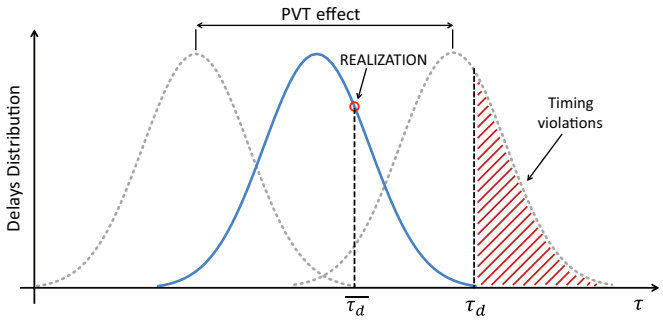

Figure 6.11: Critical path delay variability

Temperature Effect

Sub-micrometer devices are inherently affected by dynamic variations such as ambient temperature. We have modeled this dependency with a linear model. To cover different scenarios we have chosen a set of sensitivity values ($S = \frac{\Delta\tau_d}{\Delta T}$), two in Thermal Inversion (TI) [17] and two Not in Thermal Inversion (NTI). For both regions we chose an High and Low sensitivity value. Table 6.1 shows the average delay $\Delta\tau_d$ variation induced by a temperature variation ΔT of 80^{o}C.

Table 6.1: Sensitivity and average critical path variation

	SENSITIVITY	VARIATION
NTI	LOW	23%
	HIGH	139%
TI	LOW	19%
	HIGH	135%

6.4.2 Simulation Infrastructure

Our evaluation is based on a Matlab/SystemC co-simulation infrastructure. The reason behind is the different time-scales involved in the ambient temperature changes (hours/months) and in the micro-architecture domain in which acts the detection phase as well as the interconnect reconfiguration mechanism.

The modules of our solution, modeled in SystemC, were integrated in a flexible and accurate virtual platform environment based on VirtualSoC [14]. Our enhanced virtual platform is highly modular and capable of simulating at cycle-accurate level an entire shared L1 cluster including cores, instruction caches, shared L1 data scratchpad (TCDM), external memories and system interconnections. Our shared L1 cluster consists of a configurable number of 32-bit instruction-accurate ARMv6 processors. The target architecture features private Instruction Caches per core. The logarithmic interconnect module has been modeled, from a behavioral standpoint, as a parametric, Mesh-of-Trees (MoT) interconnection network capable of dynamically changing PE-memory paths delay

at runtime so as to mimic the real HW described previously. On the data side, a multi-ported, multi-banked (banking factor 2x) TCDM is directly connected to the logarithmic interconnect.

The main simulation parameters regarding the architecture are as follows: PEs = 16, TCDM banks = 32, TCDM size = 256 KB, f_{CLK} = 250 MHz. This setup mimics the configuration of [11].

We have integrated the SystemC cycle accurate simulator and architecture model as a back-end of a Matlab model of the proposed solution and temperature profile (shown schematically in Figure 6.12). Our simulation flow acts as follows: in the high level Matlab simulator we generate a temperature profile, we generate the delay table of the interconnect as realization of the static process variation discussed in Section 6.4.1. For each time step of the temperature profile (minutes) we collect a trace of the thermal hazards and the relative delay table updated with the ambient temperature-delay model as discussed in Section 6.4.1. These information are then passed to the cycle accurate simulation to evaluate detection/reconfiguration costs and the final performance gain considering a matrix multiplication benchmark. This is done by simulating the detection and reconfiguration costs accordingly with the cache size and the current delays configuration and updating it based on our proposed policy.

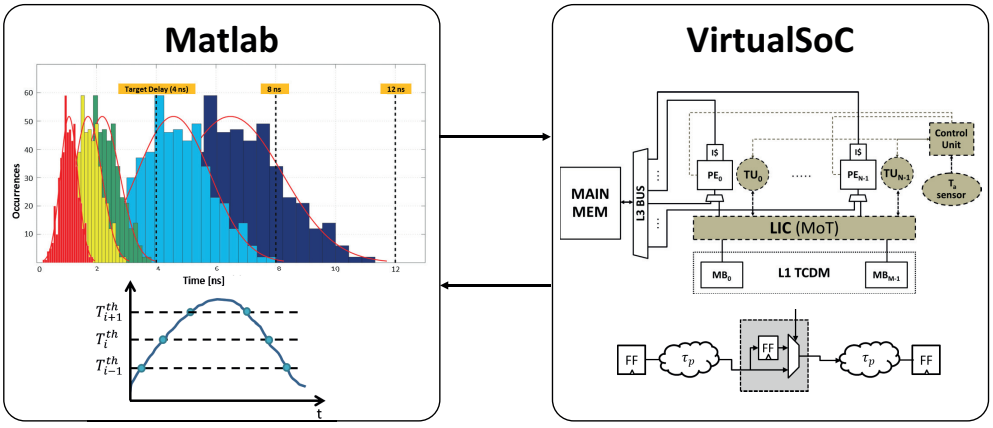

Figure 6.12: Co-simulation infrastructure

6.4.3 Tests and Performance

In this section we discuss the performance of the proposed solutions, both static (off-line) and dynamic (on-line). The first set of experiments was carried out to compare with a classic zero-area-overhead solution (clock scaling). Ambient temperature effect is later introduced to quantify the performance of the dynamic solution and finally the overhead of both solutions is discussed.

Clock Scaling

As a first test we compared the static solution against the classical clock scaling approach (operated at cluster level, thus affecting all processing elements) to compensate variations. We iterated 100 times a matrix multiplication benchmark (a memory-bound application, thus worst case for our architecture), taking the case without variability as the baseline.

To compare our solution with the frequency scaling we considered different realizations of the static process variation and finely tuned the cluster frequency to tolerate the maximum critical path delay. Results of the comparison are shown in Table 6.2 where a reference temperature of $T = 25^oC$ is considered. The table shows that increasing the number of delays insertions required, considering 16

Table 6.2: Clock Scaling Comparison

	FREQUENCY SCALING		VAR. TOLERANT	
MAX(τ_d) (ns)	FREQ (MHz)	OVERHEAD (%)	DELAYS	OVERHEAD (%)
4.00	250.00	0	0	0
4.66	214.51	16.55	19	1.83
5.49	181.99	37.37	79	4.52
6.33	158.04	58.19	179	8.17
7.49	133.45	87.34	298	12.39
8.83	113.30	120.65	404	14.73
9.66	103.53	141.47	464	17.01
10.49	95.32	162.29	519	20.10
11.99	83.40	199.77	618	21.52

PEs, 32 banks and 2 delays per path, the maximum number of delays is 1024. Our solution has a very small overhead compared to the baseline (no variation, ideal case) with a slowdown in average $\approx$ 10x smaller compared to clock scaling.

Speedup

To evaluate the performance of our resilient architecture, we compared it to a static solution which takes into account PVT variations at design time, i.e. adapting the frequency. In this experiment we are also considering the effects of the ambient temperature, which has a great impact on ULP systems.

We analyzed different temperature values at which the device is operating, thus determining the delay configuration at boot-time and measured its performances in terms of speedup compared to a non-resilient frequency-scaling solution. We are considering here 16 PEs and 32 TCDM banks.

The performance metric is the *daily throughput* measured as the total number of benchmarks (matrix multiplication) executed in one day. Clock frequency is set accordingly to the maximum critical path delay taking into account the temperature effect The operating temperature range considered here is between -20^oC and 60^oC and we chose the "high" Sensitivity parameters to increase the overhead of our solution while both TI (Thermal Inversion) and NTI (Not

Table 6.3: Speedup with temperature effect

TEMPERATURE (^{o}C)	SPEEDUP (%)	
	(NTI)	(TI)
-20	9.99	23.10
-10	14.61	22.41
0	15.14	21.86
10	15.87	17.92
20	17.10	17.47
30	17.86	16.83
40	20.08	15.64
50	22.08	14.81
60	24.62	13.52

in Thermal Inversion) models are considered. The speedup values presented in Table 6.3 are computed as average speedup of a consistent number of instances of the random process described before.

Yearly Throughput

We compared our dynamic approach against the static case, consisting of the ULTRASAFE configuration loaded at boot time. If no dynamic information on ambient temperature are exploited, the ULTRASAFE configuration is the only one that ensures correct operation under all possible ambient temperature conditions. These tests have been performed for three representative yearly temperature profiles (also considering daily temperature variations), based on a public meteorologic database[1], namely "*San Francisco*", "*Shanghai*"and "*Moskow*" and by varying the temperature thresholds step (ΔT) and the number of entries in the Reconfiguration Cache. We have tested them for 4 different delay-temperature sensitivity values.

For the "San Francisco" temperature profile we show the impact of the thresh-

[1]http://www.weather-and-climate.com

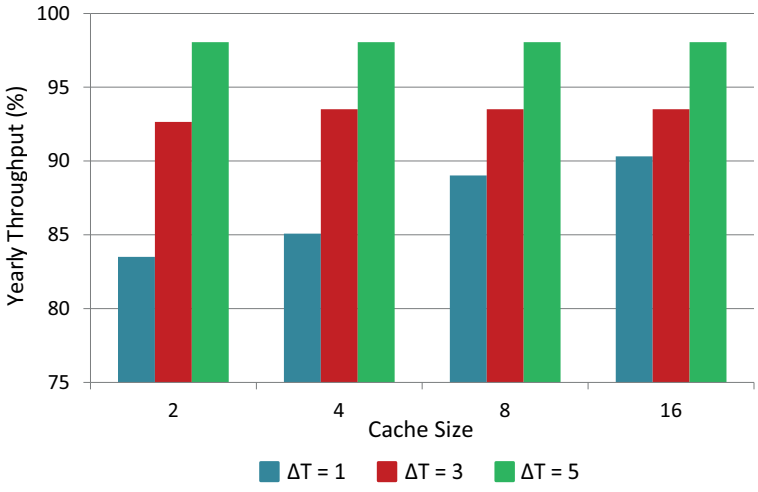

Figure 6.13: Yearly Throughput (San Francisco temperature profile)

olds step (y-axes) and the number of cache lines (x-axes) on the performance gain and detection/reconfiguration overhead compared to the static case. Figure 6.13 shows on the z-axes the average throughput measured as the total number of benchmarks executed in one year with our solution normalized to the no-delay configuration whereas Figure 6.14 shows in logarithmic scale the overhead in milliseconds of our solution when compared to static and no-delay (ideal) configuration. Compared with the latter our solution implies a cost for detection and reconfiguration.

As we can see from the figure, cache size and the chosen threshold have a strong impact on the final overhead. Lower threshold values imply more monitoring and potential detection activities and consequently a higher number of detection and reconfigurations. This has a negative effect on the cache locality increasing the eviction rate. Moreover, it must be considered that when a threshold triggers the Dynamic Control Unit, if a valid entry is not present in the cache the algorithm selects the ULTRASAFE configuration to ensure the absence of errors. This has the negative effect of decreasing the overall throughput as it can be observed in Figure 6.13.

On the other hand this effect almost disappeared when bigger threshold steps

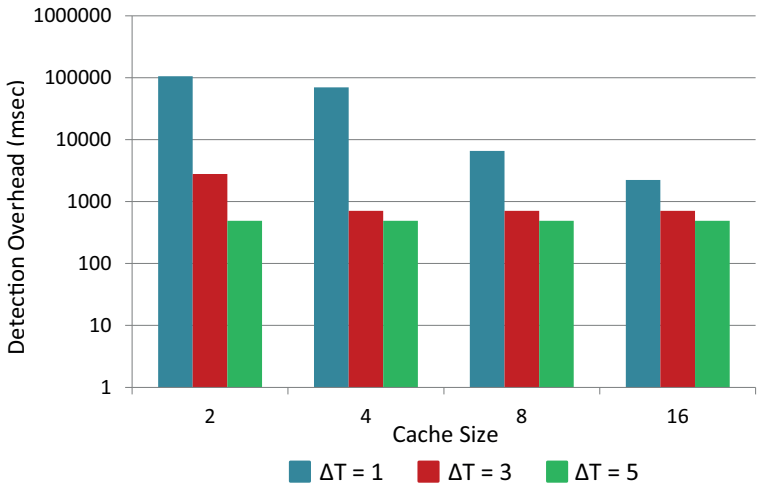

Figure 6.14: Detection Overhead (San Francisco temperature profile)

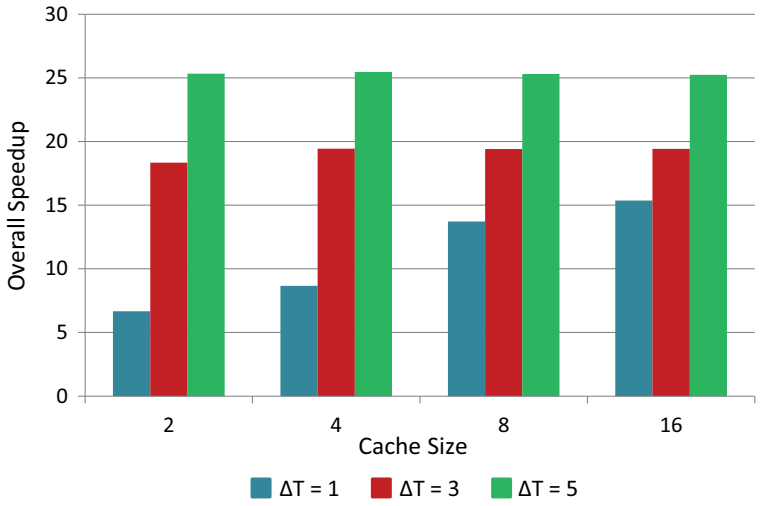

Figure 6.15: Overall Speedup (San Francisco temperature profile)

were used (i.e. $5°C$). This is primarily due to the fact that consecutive days show similar temperature profiles, thus increasing the hit-rate in the Reconfiguration Cache. Figure 6.15 shows the performance gain of our solution compared to the ULTRASAFE one in terms of average throughput in an year of execution. With respect to Figure 6.13 it takes into account the costs of on-line detection and reconfiguration. We can notice that our solution achieve up to 25% of performance gain without compromising the system reliability. The caching mechanism effectively hides the overhead of hazard detection and reconfiguration. On the other side, to limit the area overhead we chose as best trade-off the configuration with 4 cache lines and $\Delta T = 5°C$. In Table 6.4 we show the average performance gain of our solution for the three different temperature profiles and for different

Table 6.4: Performance Speed-Up for all temperature profiles

PROFILE	TI HIGH	TI LOW	NTI HIGH	NTI LOW
SAN FRANCISCO	25.25%	27.74%	19.23%	24.73%
SHANGHAI	16.67%	17.98%	14.66%	17.19%
MOSKOW	23.15%	25.28%	20.08%	22.94%

sensitivity values as discussed in Section 6.4.1.

Area Overhead

To evaluate the impact of our approach, we synthesized our HW modules on a general purpose 65nm commercial technology library [83]. Considering our architecture, we have 16 detection units and 32 pipeline modules for the whole system. As outlined in the previous section, we chose as configuration for the dynamic solution a Reconfiguration Cache with 4 entries and $\Delta T = 5°C$, leading to 16 threshold temperatures for our operating range $(-20°C, 60°C)$.

Table 6.5 shows the area impact of our solution. To compare the impact of HW modules on a per-core basis, we also present the total overhead relative to the Mega-Leon design considered in [42]. These results show that the HW modules to make the baseline architecture resilient to variations have a very low overhead for both static and dynamic solutions.

Table 6.5: Area Overhead of Static and Dynamic solutions

MODULE	AREA (μm^2)	UNITS	OVERHEAD (%)
TU (STAT/DYN)	2180 / 2676	16	0.87 / 1.07 %
RECONF STAGE	1230	48	1.48 %
CU (STAT/DYN)	12097 / 49598	1	0.31/1.24 %
TOTAL (STAT/DYN)	113953 / 151454	-	2.58/3.79 %

6.5 Conclusions

In this chapter we showed an architectural scheme to increase system resiliency to dynamic critical path variations, induced by ambient temperature fluctuations. This problematic is exacerbated in ULP devices operating at the near-threshold voltage when compared to nominal supply voltage. Our solution exploits a resilient logarithmic interconnect and integrates it with a set of new HW modules capable of sensing the current ambient temperature, recognize possible temper-

ature hazards, checking the memory and link consistency and react by reconfiguring, through a SW procedure, the interconnect delays. The solution has been evaluated on our cycle-accurate simulator. The results show that our solution is suitable for bio-sensors and Wireless Body Area Sensor Networks and compared to state-of-the-art static solutions is resilient to the Ambient Temperature variation achieving a performance gain up to $\approx 25\%$ in a typical use case scenario, with a very low area overhead.

Chapter 7

Conclusions

The extraordinary computing performance achievable by the many-core paradigm is limited not only by Amdahl's law, but several other factors concur in reducing the degree of effectiveness of modern MPSoCs. An inefficient memory hierarchy usage, a problem exacerbated in many-core systems, combined with the lack of efficient synchronization mechanisms can severely overcome the potential computation capabilities. Moreover, the quest for energy efficiency by means of near-threshold operation, exposes such platforms in today's technological nodes to other challenges: reliability and variability.

In this dissertation we firstly introduced our Virtual Platform, coded in SystemC, that has been developed to serve as a simulation infrastructure targeting modern many-core architectures. In the first part of the thesis the focus is on scalable performance, addressing two architectural aspects: first, we conducted an in-depth study of two instruction cache templates, based on the use of both synthetic micro-benchmarks and real program workloads, providing useful insights and guidelines for designers. Next, we moved to barrier synchronization mechanisms, is a key programming primitive for shared memory embedded MPSoCs. We integrated our custom barrier implementation into a widespread programming model for shared memory systems such as OpenMP, and discussed synchronization efficiency compared to traditional software implementation.

Beside architectural implications of modern many-core SoCs, another para-

mount issue of embedded systems is considered in this work: energy efficiency. The second part of the dissertation, focuses on reliability and resiliency aspects of Near Threshold Computing, a promising way to achieve ultra-low-power operation. When operating at near-threshold, especially memory operation becomes unreliable and can compromise system correctness. We introduced a novel hybrid memory architecture to overcome reliability issues and at the same time improve energy efficiency by means of aggressive voltage scaling when allowed by workload requirements. Finally, the implications of increased variability effects are taken into considerations. By means of a micro-architectural knobs inserted at design and a lightweight runtime control unit, we extend the baseline architecture to be dynamically tolerant to variations.

Publications

2011

- **Daniele Bortolotti**, Francesco Paterna, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. Exploring instruction caching strategies for tightly-coupled shared-memory clusters. In *System on Chip (SoC), 2011 International Symposium on*, pages 34–41. IEEE, 2011.

2012

- José L Abellán, Juan Fernández, Manuel E Acacio, Davide Bertozzi, **Daniele Bortolotti**, Andrea Marongiu, and Luca Benini. Design of a collective communication infrastructure for barrier synchronization in cluster-based nanoscale mpsocs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 491–496. IEEE, 2012.

2013

- **Daniele Bortolotti**, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 2182–2187. IEEE Computer Society, 2013.

- **Daniele Bortolotti**, Andrea Bartolini, and Luca Benini. An ambient temperature variation tolerance scheme for an ultra low power shared-l1 processor cluster. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 625–632. IEEE, 2013.

- **Daniele Bortolotti**, Davide Rossi, Andrea Bartolini, and Luca Benini. A variation tolerant architecture for ultra low power multi-processor cluster. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, pages 32–38. IEEE, 2013.

2014 (to appear or submitted)

- **Daniele Bortolotti**, Andrea Bartolini, Christian Weis, Davide Rossi and Luca Benini. Hybrid Memory Architecture for Voltage Scaling in Ultra-Low-Power Multi-Core Biomedical Processors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, to appear. IEEE, 2014.

- **Daniele Bortolotti**, Andrea Bartolini and Luca Benini. An Ultra-Low Power Resilient Multi-core Architecture with Static and Dynamic Tolerance to Ambient Temperature-induced Variability. In *Microprocessors and Microsystems*, submitted. Elsevir, 2014.

- **Daniele Bortolotti**, Hossein Mamaghanian, Andrea Bartolini, Maryam Ashouei, Pierre Vandergheynst and Luca Benini. Approximate Compressed Sensing: Ultra-Low Power Biosignal Processing via Aggressive Voltage Scaling on a Hybrid Memory Multi-core Processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, submitted. ACM, 2014.

Bibliography

[1] *The JPEG still picture compression standard*, volume 34, 1991. AcM. 49

[2] Plurality ltd. - the hypercore processor, 2012. URL http://www.plurality.com/hypercore.html. 5, 33, 35, 36, 58, 101

[3] SystemC 2.3.0 Users Guide. 2012. 36

[4] José L Abellán, Juan Fernández, and Manuel E Acacio. A g-line-based network for fast and efficient barrier synchronization in many-core cmps. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 267–276. IEEE, 2010. 58, 62

[5] Adapteva, Inc. Epiphany-IV 64-core 28nm Microprocessor, 2013. URL http://www.adapteva.com/products/silicon-devices/e64g401. 33

[6] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. volume 28, pages 248–259. ACM, 2000. 1, 2

[7] Arm Ltd. ARM 11 product page, 2011. URL http://www.arm.com/products/processors/classic/arm11. 20

[8] Robert Bedichek. Simnow: Fast platform simulation purely in software. In *Hot Chips*, volume 16, 2004. 16

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. 16

[10] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002. 3

[11] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012. 2, 4, 33, 36, 58, 101, 102, 118

[12] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004. 16

[13] Shekhar Borkar and Andrew A Chien. The future of microprocessors. In *Communications of the ACM*, volume 54, pages 67–77. ACM, 2011. 1, 2, 8, 9, 99, 100

[14] Daniele Bortolotti, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 2182–2187. IEEE Computer Society, 2013. 36, 93, 117

[15] Nathan Brookwood. Amd fusion family of apus: enabling a superior, immersive pc experience. *Insight*, 64(1):1–8, 2010. 2, 13

[16] Benton H Calhoun and Anantha P Chandrakasan. Static noise margin variation for sub-threshold sram in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 41(7):1673–1679, 2006. 83, 91, 95

[17] Andrea Calimera, R Bahar, Enrico Macii, and Massimo Poncino. Reducing leakage power by accounting for temperature inversion dependence in dual-

vt synthesized circuits. In *Proceedings of the 13th international symposium on Low power electronics and design*, pages 217–220. ACM, 2008. 101, 117

[18] Anantha P Chandrakasan, Naveen Verma, and Denis C Daly. Ultralow-power electronics for biomedical applications. In *Annu. Rev. Biomed. Eng.*, volume 10, pages 247–274. Annual Reviews, 2008. 101

[19] Ik Joon Chang, Debabrata Mohapatra, and Kaushik Roy. A priority-based 6t/8t hybrid sram architecture for aggressive voltage scaling in video applications. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(2):101–112, 2011. 83

[20] Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. Yield-driven near-threshold sram design. In *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, volume 18, pages 1590–1598. IEEE, 2010. 10, 100, 101

[21] NVidia Corp. Fermi architecture white paper, 2012. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 33, 35

[22] Tilera Corporation. Tilera processors, 2013. URL http://www.tilera.com/products/processors. 2

[23] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 5, 59

[24] William J Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001. 3

[25] Ahmed Yasir Dogan, David Atienza, Andreas Burg, Igor Loi, and Luca Benini. Power/performance exploration of single-core and multi-core processor approaches for biomedical signal processing. In *Integrated Circuit and*

System Design. Power and Timing Modeling, Optimization, and Simulation, pages 102–111. Springer, 2011. 82, 100

[26] Ahmed Yasir Dogan, Jeremy Constantin, Martino Ruggiero, Andreas Burg, and David Atienza. Multi-core architecture design for ultra-low-power wearable health monitoring systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 988–993. IEEE, 2012. 82, 85, 100, 101

[27] Ronald G Dreslinkski, Bo Zhai, Trevor Mudge, David Blaauw, and Dennis Sylvester. An energy efficient parallel architecture using near threshold operation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 175–188. IEEE Computer Society, 2007. 82, 85

[28] Ronald G Dreslinski, Gregory K Chen, Trevor Mudge, David Blaauw, Dennis Sylvester, and Krisztian Flautner. Reconfigurable energy efficient near threshold cache architectures. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 459–470. IEEE, 2008. 83, 84

[29] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. In *Proceedings of the IEEE*, volume 98, pages 253–266. IEEE, 2010. 9, 10, 99, 100, 101, 116

[30] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18. IEEE, 2003. 101

[31] David Fick, Ronald G Dreslinski, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, et al. Centip3de: A 3930dmips/w configurable near-threshold

3d stacked system with 64 arm cortex-m3 cores. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 190–192. IEEE, 2012. 10, 100

[32] Stephen Bo Furber. *ARM system-on-chip architecture*. pearson Education, 2000. 44

[33] David Geer. Chip makers turn to multicore processors. volume 38, pages 11–13. IEEE, 2005. 1, 2

[34] Marius Gligor and Frederic Petrot. Combined use of dynamic binary translation and systemc for fast and accurate mpsoc simulation. In *1st International QEMU Users' Forum*, volume 1, pages 19–22, March 2011. 16

[35] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to intel core duo processor architecture. *Intel Technology Journal*, 10(2), 2006. 2

[36] Alain Greiner, Etienne Faure, Nicolas Pouillon, and Daniela Genius. A generic hardware/software communication middleware for streaming applications on shared memory multi processor systems-on-chip. In *Specification & Design Languages, 2009. FDL 2009. Forum on*, pages 1–4. IEEE, 2009. 20

[37] Christophe Guillon. Program instrumentation with qemu. In *1st International QEMU Users' Forum*, volume 1, pages 15–18, March 2011. 16

[38] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137. IEEE, 2013. 2

[39] C. Helmstetter and V. Joloboff. Simsoc: A systemc tlm integrated iss for full system simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 1759 –1762, 2008. 20

[40] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010. 2

[41] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005. 2

[42] Mohammad Reza Kakoee, Igor Loi, and Luca Benini. A resilient architecture for low latency communication in shared-l1 processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 887–892. EDA Consortium, 2012. 101, 102, 105, 106, 116, 124

[43] Mohammad Reza Kakoee, Igor Loi, and Luca Benini. Variation-tolerant architecture for ultra low power shared-l1 processor clusters. In *Circuits and Systems II: Express Briefs, IEEE Transactions on*, volume 59, pages 927–931, 2012. 101, 102

[44] Kalray, Inc. Kalray MPPA MANYCORE, 2013. URL `http://www.kalray.eu/products/mppa-manycore`. 2, 6

[45] Shuo Kang, Huayong Wang, Yu Chen, Xiaoge Wang, and Yiqi Dai. Skyeye: An instruction simulator with energy awareness. In *Embedded Software and Systems*, pages 456–461. Springer, 2005. 20

[46] Karim Kanoun, Hossein Mamaghanian, Nadia Khaled, and David Atienza. A real-time compressed sensing-based personal electrocardiogram monitoring

system. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011. 83, 97

[47] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1153–1158. ACM, 2012. 99, 100

[48] Khronos OpenCL Working Group and others. The opencl specification. *A. Munshi, Ed*, 2008. 19

[49] Evgeni Krimer, Robert Pawlowski, Mattan Erez, and Patrick Chiang. Synctium: a near-threshold stream processor for energy-constrained parallel applications. In *Computer Architecture Letters*, volume 9, pages 21–24. IEEE, 2010. 100

[50] R Lantz. Fast functional simulation with parallel embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*. Citeseer, 2008. 16

[51] James Larus. Spending moore's dividend. *Communications of the ACM*, 52 (5):62–69, 2009. 2

[52] Kevin Lawton. Bochs: The open source ia-32 emulation project. *URL http://bochs. sourceforge. net*, 2003. 16

[53] Jing-Wun Lin, Chen-Chieh Wang, Chin-Yao Chang, Chung-Ho Chen, Kuen-Jong Lee, Yuan-Hua Chu, Jen-Chieh Yeh, and Ying-Chuan Hsiao. Full system simulation and verification framework. In *Information Assurance and Security, 2009. IAS'09. Fifth International Conference on*, volume 1, pages 165–168. IEEE, 2009. 17

[54] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999. 49

[55] Daniele Ludovici, Georgi Nedeltchev Gaydadjiev, Davide Bertozzi, and Luca Benini. Capturing topology-level implications of link synthesis techniques for nanoscale networks-on-chip. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 125–128. ACM, 2009. 58

[56] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, feb 2002. ISSN 0018-9162. 16

[57] Wai-Kei Mak and Jr-Wei Chen. Voltage island generation under performance requirement for soc designs. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 798–803. IEEE Computer Society, 2007. 84

[58] Hossein Mamaghanian, Nadia Khaled, David Atienza, and Pierre Vandergheynst. Compressed sensing for real-time energy-efficient ecg compression on wireless body sensor nodes. *Biomedical Engineering, IEEE Transactions on*, 58(9):2456–2466, 2011. 82, 86, 90

[59] Dejan Markovic, Cheng C Wang, Louis P Alarcon, Tsung-Te Liu, and Jan M Rabaey. Ultralow-power design in near-threshold region. In *Proceedings of the IEEE*, volume 98, pages 237–252. IEEE, 2010. 9, 10, 100

[60] Andrea Marongiu, Paolo Burgio, and Luca Benini. Supporting openmp on a multi-cluster embedded mpsoc. *Microprocessors and Microsystems*, 35(8): 668–682, 2011. 49, 59, 74

[61] Aline Mello, Isaac Maia, Alain Greiner, and Francois Pecheux. Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 606–609. IEEE, 2010. 16, 17

[62] A.P. Miettinen, V. Hirvisalo, and J. Knuttila. Using qemu in timing estimation for mobile software development. In *1st International QEMU Users' Forum*, volume 1, pages 19–22, March 2011. 16

[63] T.N. Miller, R. Thomas, and R. Teodorescu. Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units. In *Computer Architecture Letters*, volume 11, pages 45–48, 2012. 116

[64] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Efficient synchronization for embedded on-chip multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(10):1049–1062, 2006. 57

[65] Marius Monton, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed sw/systemc soc emulation framework. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 2338 –2341, june 2007. 17

[66] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, 2006. 1

[67] Node Operating System. NodeOS, 2013. URL http://www.node-os.com. 7

[68] NVidia Corp. NVIDIA Tegra 4 Family GPU Architecture Whitepaper, 2013. URL http://www.nvidia.com/docs/IO//116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf. 2, 13

[69] OAR Corporation. Real-Time Executive for Multiprocessor Systems, 2013. URL http://www.rtems.org. 7

[70] Jungju Oh, Milos Prvulovic, and Alenka Zajic. Tlsync: support for multiple fast barriers using on-chip transmission lines. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 105–115. IEEE, 2011. 58

[71] World Health Organization, 2013. URL http://www.who.int/mediacentre/factsheets/fs317/en/index.html. 82

[72] Christian Pinto, Shivani Raghav, Andrea Marongiu, Martino Ruggiero, David Atienza, and Luca Benini. Gpgpu-accelerated parallel and fast simulation of thousand-core platforms. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 53–62. IEEE Computer Society, 2011. ISBN 978-0-7695-4395-6. 16

[73] Antonio Pullini, Federico Angiolini, Srinivasan Murali, David Atienza, Giovanni De Micheli, and Luca Benini. Bringing nocs to 65 nm. *Micro, IEEE*, 27(5):75–85, 2007. 58

[74] Davide Quaglia, Franco Fummi, Maurizio Macrina, and Saul Saggin. Timing aspects in qemu/systemc synchronization. In *1st International QEMU Users' Forum*, volume 1, pages 11–14, March 2011. 17

[75] Qualcomm Inc. Snapdragon s4 processors: System on chip solutions for a new mobile age, 2011. 14

[76] Shivani Raghav, Andrea Marongiu, Christian Pinto, David Atienza, Martino Ruggiero, and Luca Benini. Full system simulation of many-core heterogeneous socs using gpu and qemu semihosting. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 101–109, New York, NY, USA, 2012. ACM. 16

[77] Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–6. EDA Consortium, 2011. 21, 85, 101, 103

[78] Francisco Rincón, Joaquin Recas, Nadia Khaled, and David Atienza. Development and evaluation of multilead wavelet-based ecg delineation algorithms for embedded wireless sensor nodes. *Information Technology in Biomedicine, IEEE Transactions on*, 15(6):854–863, 2011. 82, 86

[79] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003. 92

[80] Mohamed M Sabry, Martino Ruggiero, and Pablo G Del Valle. Performance and energy trade-offs analysis of l2 on-chip cache architectures for embedded mpsocs. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 305–310. ACM, 2010. 35

[81] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008. 2

[82] Korey Sewell, Ronald G Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F Wenisch, Dennis Sylvester, et al. Swizzle-switch networks for many-core systems. In *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, volume 2, pages 278–294. IEEE, 2012. 101

[83] STMicroelectronics, 2013. URL http://www.st.com/. 4, 66, 124

[84] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010. 5

[85] Xian-He Sun and Yong Chen. Reevaluating amdahls law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010. 9

[86] The Open Virtual Platforms. OVPSim, 2013. URL http://www.ovpworld.org/. 16, 17

[87] CH Van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265. European Design and Automation Association, 2009. 2, 8

[88] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008. 2

[89] Naveen Verma and Anantha P Chandrakasan. A 256 kb 65 nm 8t subthreshold sram employing sense-amplifier redundancy. *Solid-State Circuits, IEEE Journal of*, 43(1):141–149, 2008. 83

[90] Oreste Villa, Gianluca Palermo, and Cristina Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 81–90. ACM, 2008. 57, 60

[91] Henry Wong, M-M Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010. 35

[92] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. 9

[93] Tse-Chen Yeh and Ming-Chao Chiang. On the interfacing between qemu and systemc for virtual platform construction: Using dma as a case. *J. Syst. Archit.*, 58(3-4):99–111, mar. 2012. ISSN 1383-7621. 17

[94] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, april 2007. 16

[95] Bo Zhai, Leyla Nazhandali, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Sanjay Pant, David Blaauw, and Todd Austin. A 2.60 pj/inst subthreshold sensor processor for optimal energy efficiency. In *VLSI Circuits, 2006. Digest of Technical Papers. 2006 Symposium on*, pages 154–155. IEEE, 2006. 100

[96] Yanqing Zhang, Yousef Shakhsheer, Adam T Barth, Harry C Powell Jr, Samuel A Ridenour, Mark A Hanson, John Lach, and Benton H Calhoun. Energy efficient design for body sensor nodes. In *Journal of Low Power Electronics and Applications*, volume 1, pages 109–130. Molecular Diversity Preservation International, 2011. 10, 100