Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Simulating Complex Multi-core Computing Systems: Techniques and Tools

Simone SECCHI

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXIII Cycle
March 2011

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Simulating Complex Multi-core Computing Systems: Techniques and Tools

Simone SECCHI

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXIII Cycle
March 2011

To those I often neglected
and forgot to appreciate

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the era of frequency scaling has come to an end, the computing industry changed course and *parallelism* is now the new buzzword. In the last decade, several factors, such as the ever-increasing need for computational power and the renewed attention to energy-related issues, have led to the reconsideration of parallel computation in its multiple forms as a viable way to sustain the pervasive need for computational power of most electronic devices. Complex computing architectures have been explicitly conceived to benefit from (a combination of) the different features of parallelism, ranging from instruction level (ILP), through data level (DLP), to the so-called thread level parallelism(TLP) [7]. This trend has become widespread, with different timing and at different scale, in all areas of computing.

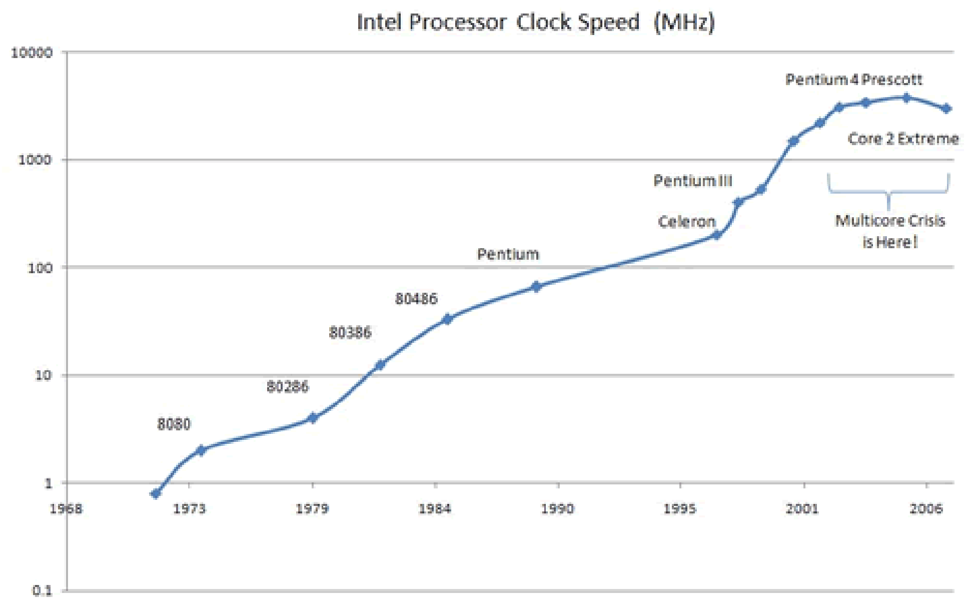


Figure 1.1: Frequency scaling over last generations of Intel CPUs

In the embedded systems and general purpose computing domain, this evolution translated into the integration of complete heterogeneous multi-core systems on a single chip (Multi-Processor Systems-on-Chip, MPSoCs). As a consequence of this trend, even the well known Moore's Law has been adapted to address the increasing number of cores integrated in a single chip. Instead of predicting a steady increase in single-processor operating frequencies (which are expected to stabilize when not decrease because of power issues), the evolution of the most famous statement of digital electronics now anticipates a doubling of the number of per-chip cores every two years, as the market enters the so called "multi-core era". Figure 1.1 show the flat curve for Intel's main CPUs frequency scaling over different processors generations.

Figure 1.2 instead, plots the number of cores integrated on single-chip parallel processors, as stated by the reformulation of Moore´s Law. The heterogeneity and inherent complexity of such systems have in turn increased the complexity of the overall embedded systems hardware/software design flow. Moreover, this complexity has to be handled under strict time-to-market constraints.

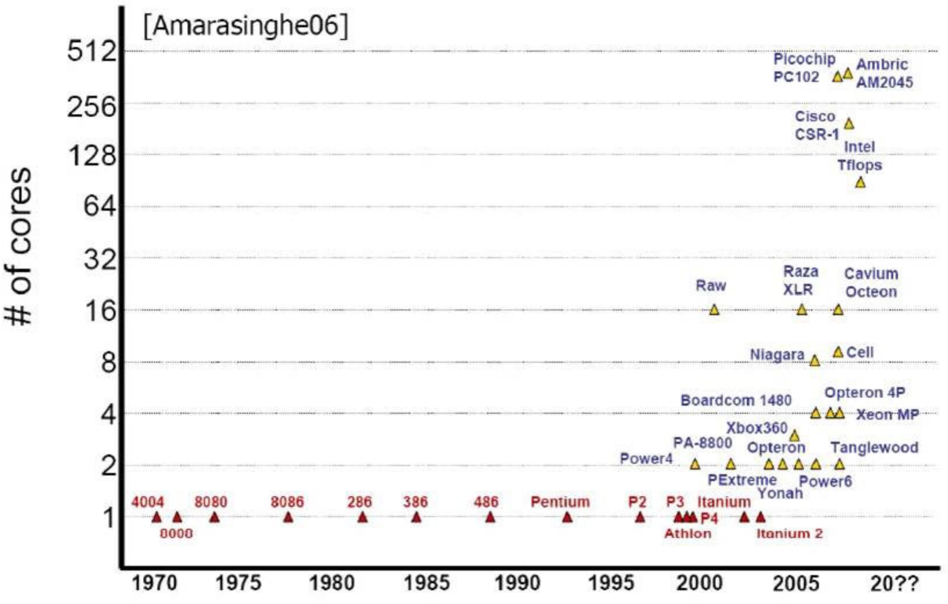


Figure 1.2: Number of cores integrated on single chip processors

On the large-scale and high-performance computing domain, the computational power needed by advanced scientific problems in domains such as earth sciences, material sciences, chemistry, biology, physics and social sciences is the major driving force to the development of faster supercomputing machines. Although the fastest machine in the world is now able to reach a peak performance speed of 4.7 PetaFlops (10^{15} floating point operations per second)[37], USA and China are already the leading players in the race to the development of next-generation Exascale computing architectures. It is anticipated that not only such architectures will aggregate millions of cores with inter- and intra-chip parallelism (several thousands of at least 1000-cores processors), but also they will exploit the edge-cutting performance of complex multi-threaded hybrid-core architectures, as complex GPU and hybrid APU chips are entering the domain [21][1]. Figure 1.3 plots the current and future trends in high performance computing, mentioning the kind of processors employed for each system generation. Figure 1.4 plots the performance growth of supercomputing machines during the last 70 years. As of 2010, the Tianhe-1A supercomputer at the National Supercomputing Center in Tianjin, China, has overcome by 1.4 times the US AMD Opteron-based Cray XT5 Jaguar at the Oak Ridge National Laboratory. According to Nvidia, Tianhe-1A has achieved a processing rate of 2.507 PF on the LINPACK benchmark. The machine consists of 14,336 Intel Xeon CPUs and 7,168 Nvidia Tesla M2050 GPUs with a proprietary interconnect subsystem of Chinese origin, reporting twice the speed of InfiniBand. Tianhe-1A spans 103 cabinets, weighs 155 tons, and consumes 4.04 megawatts of electricity. Interestingly, however, the most relevant aspect of the performance trend is not the current and future scale of such high performance computing systems (it could have easily been predicted as a natural prosecution of the raw computational power aggregation trend), but the inherent high complexity of the design of such machines, resulting from the switch toward hybrid CPU-GPU architectures.

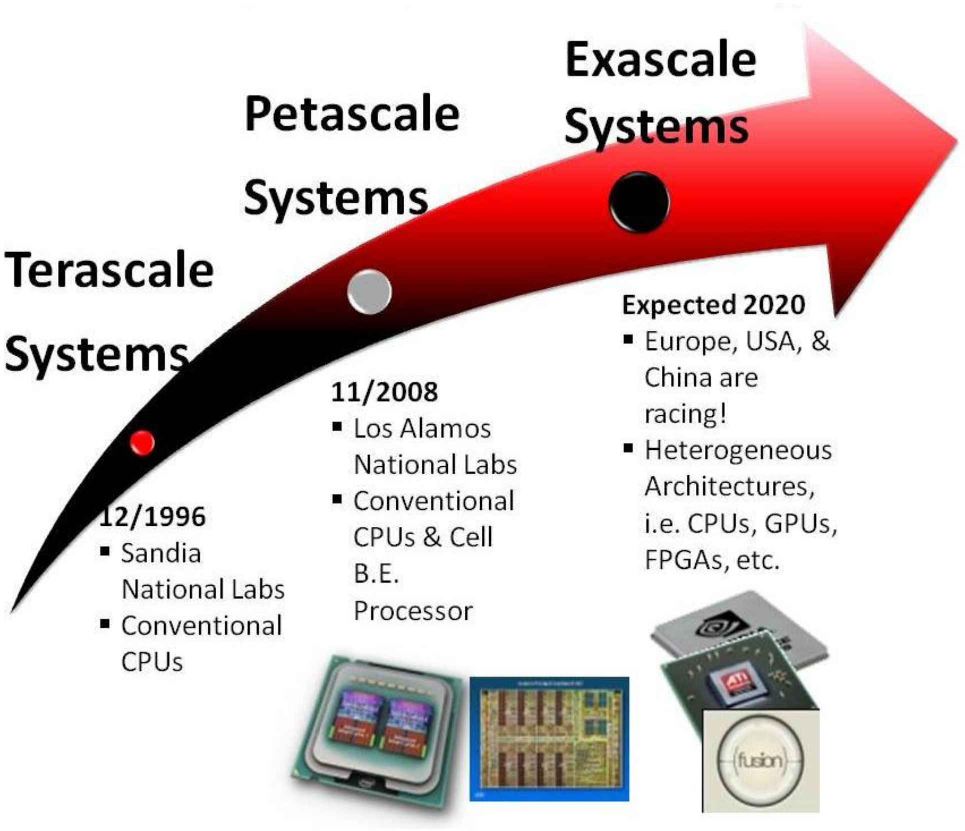

Terascale Systems
Petascale Systems
Exascale Systems
12/1996
▪ Sandia National Labs
▪ Conventional CPUs
11/2008
▪ Los Alamos National Labs
▪ Conventional CPUs & Cell B.E. Processor
Expected 2020
▪ Europe, USA, & China are racing!
▪ Heterogeneous Architectures, i.e. CPUs, GPUs, FPGAs, etc.
fusion

Figure 1.3: Trends and processors in high performance computing

Because of the considerations mentioned above, the complexity of designing such systems has increased considerably. On the embedded side, the design space to be explored is typically vast, mainly because of the high heterogeneity of the system and the resulting number of degrees of freedom that the designer needs to take into account. To mention some, one could consider the number and kind of homogeneous/heterogeneous processing elements, the architectural parameters of the interconnection subsystem, the number of layers and the operating policies of the memory hierarchy, or the different types of hardware support for threading and synchronization. Moreover, the continuous reduction of the technological feature size and the reduced power budget increase the probability for possible modifications to happen late in the design flow, when not leading to costly respins of the entire project. Regarding high performance systems, the design complexity mainly follows the scale of the machine. Developing machines intended to run billions of threads, tailoring

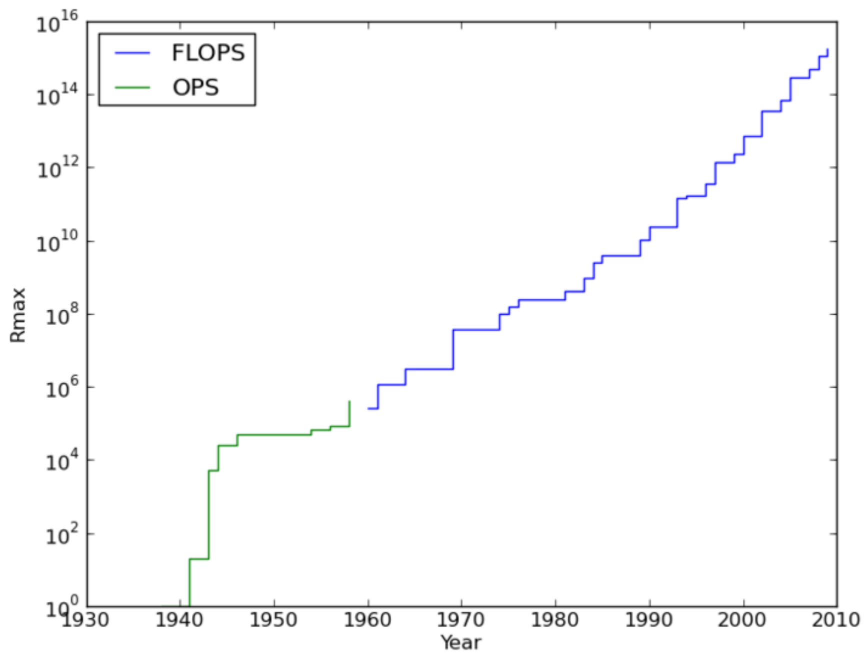


Figure 1.4: Peak performance growth of supercomputing machines over the last 70 years.

a fast communication medium to let them synchronize one with each other and enabling parallel access to hundreds of TeraBytes of distributed memory are rapidly becoming very challenging tasks.

To facilitate the design of such complex platforms, effective techniques for predicting the performances and identifying the most critical design choices at the early stages are needed. Functional verification has been for years the main tool that hardware designers have been resorting to, in order to assess whether the built system was able to perform the logic functionality it was intended to perform. Functional verification was also used to explore the available design space, meaning to anticipate the effects that different design choices could have on the overall performance and behavior of the system under development.

In hardware design, functional verification can be pursued in many different ways, among which the most applied solutions are certainly:

- *Logic simulation* - this term usually identifies all the verification approaches that take as input an exact description of the system to be developed, specified with a hardware description language, and simulate its behavior by running a software, called simulator, on a separate platform. The description of the hardware under development can be made at different levels of abstraction, ranging from high level descriptions (usually called algorithmic), through cycle-accurate levels (Register Transfer Level, RTL), to very detailed gate- or even layout-level descriptions.

- *Logic emulation* - this term identifies all the approaches to functional verification that try to mimic the functionality of a piece of logic with different logic. Using different logic to emulate the system under development might be convenient due to different reasons. For instance, the selected logic used to emulate the system under design might speed-up the entire verification process, or reduce the cost to affordable levels. Moreover, one might even want to validate the behavior of a complex system built out of pieces of logic whose hardware description is not available, and thus resort to alternative logic that guarantees adequate similarity of the overall system-level behavior.

Both the approaches mentioned above have their respective advantages and disadvantages. Simulation is the most natural and general way for the designer to get feedback about its design. Because it basically takes as input the design itself, the designer interacts with it using the vocabulary and abstractions of the design itself. There is no layer of translation to obscure the behavior of the design. The accuracy can be, for synchronous logic, the highest desirable at cycle-level. Moreover, the cost is usually very low and the effort required to debug and then verify the design is proportional to the maturity of the design. That is to say that early in the design flow, bugs and incorrect behavior are usually found quickly. As the design matures, it takes longer to find the errors. While this is beneficial early in the design process, it becomes more problematic later. As for the disadvantages, simulation requires considerable computing resources to be performed and, as the complexity of the design increases, always gets more and more time-consuming. Emulation has several advantages, which depend merely on the kind of logic that it is chosen to emulate the desired piece of hardware. The emulating system can usually be more rapid with respect to simulation infrastructure, although slower enough with respect to the real hardware to still be less expensive to realize. This higher speed can be crucial when complex software validation has to be performed on top of the emulated system. Emulation is often the right choice in the early steps of the design flow, as several details on specific hardware parts are not yet available. It is often performed employing reconfigurable hardware platforms, in which case the emulation speed usually overcomes by orders of magnitude raw simulation speed. This approaches are often referred to as prototyping (although sometimes this term also indicates simulation techniques). The disadvantages of emulation are quite obvious, the system under development is, in some of its composing blocks, different from the design objective, therefore it is inappropriate even defining an accuracy level, since only functional correctness can be proven. For the same reason, probing of single signals is unfeasible, as opposed to simulation-based approaches.

In the recent past, these two areas of functional verification have been undergoing radical changes, in order to cope with the ever-increasing complexity of the computing systems. This trend in functional verification has been going on, with different shapes and paces, both in the embedded system and high performance computing domains. In general, the huge complexity of today's and tomorrow's computing systems has exacerbated the constraints posed by the most important trade-off that holds in functional verification, namely the speed-accuracy trade-off. In fact, as the systems get larger and more complex, simulating or emulating at the maximum detail the entire system becomes too much time-consuming.

In the embedded systems domain, after the advent of heterogeneous MPSoC architectures, software cycle-accurate simulation has turned to be not practical anymore because of the large timing overhead required to handle all the signals, interfaces and states present in the system. Moreover, due to tight integration and time-to-market constraints, hardware/software co-development has been established as the major design paradigm, forcing the simulation/emulation to provide support to verify hardware functionalities and run complex software, such as operating systems or complex applications, at the same time.

Finally, there is an increasing need to narrow the gap between the earlier and the latter stages of the hw/sw design flow, since modifications in advanced stages generates unsustainable costs and impact on the overall project design closure. Therefore, evaluation of technology and physical metrics (prospective on-silicon area occupation, power and energy consumption, operating frequency, thermal distribution) about the system under design are desirable already in the early design steps.

For all these reasons, both in the academic and industrial communities, the interest has recently shifted toward the adoption of hardware-based emulation/prototyping platforms. Such platforms are able to achieve higher speed with respect to pure software simulation while maintaining cycle-accuracy, and are thus able to run complex software stacks and applications in reasonable times. One of the key driving forces behind the success gained by hardware-based emulation is the availability of reconfigurable devices, such as the Field-Programmable Gate Arrays (FPGAs). Their ability to implement the functionalities of every generic piece of logic, combined with efficient synthesis/implementation tools that convert RTL descriptions into real hardware, make them a convenient layer for hardware emulation. In addition to that, ASIC technology scaling has an impact also on FPGA integration capabilities, cost and price, promising to reach, in the next futures, ever higher *equivalent gates* figures.

Also in the high performance computing field, pure cycle-accurate software simulation has been for years an essential tool for the development of new architectures and for the analysis of software performance. Nevertheless, simulating high performance parallel machines, with high core/processor counts and complex architectures, poses several contrasting challenges. Designing sufficiently fast simulators to explore new architectural features with meaningful applications for systems of this size is difficult. Communication backbones should be correctly modeled, because they significantly influence the execution times of the applications. Moreover, interesting applications often have a large memory footprint, so simulators should limit as much as possible their space overhead.

Only fast simulators can adequately support the research of new software approaches, from operating systems to languages, from runtimes to applications, for present and future machines of this class. At the same time, high simulation speed should not trade off simulation accuracy, to guarantee relevant analysis with respect to existing architectures. The majority of architectural and system simulators is single-threaded [47][13][15][48][34]. Even if some of them are very accurate, these solutions are too slow to run more than small application kernels in reasonable times, and cannot adequately scale to multicore processors or large multiprocessor systems. This is a significant limit, considering that future supercomputing machines are expected to count thousands of cores and hundreds of thousands of processors.

The current trend in this field is to achieve further simulation speedup by looking at parallel software simulation, aiming at applying current parallelization techniques at the simulator code itself. To this extent, Shared Memory multiProcessor (SMP) servers and small clusters have become more affordable and consequently more attractive for accelerating sequential simulator codes. Thus, in the last few years the research focus has shifted towards simulators able to export different types of parallelism for accelerating the simulation of multicore or multiprocessor systems [39][40][38].

1.1 Main objectives and thesis organization

This thesis is organized in two parallel tracks, as it aims at reflecting the objectives of the two different research threads that have been carried on, in the field introduced above, during the last years of activity. The two main objectives and activities can be summarized as follows:

- The first section of the document covers the research activity performed in the field of *technology-aware FPGA-based emulation of multi-core architectures for embedded systems*. The basic idea behind this research activity is to target the problems and trends mentioned above for the embedded systems design field. In particular, the objective is to take profit from the advantages provided by on-hardware prototyping and to consider, already at system/architectural level, the variables related to the low-level implementation, introducing the concept of "system-level design with technology awareness". The activity has been mainly focused on coupling FPGA-based hardware prototyping and RTL-based power modeling to address the problem of technology-aware rapid emulation of embedded computing systems. A complete library-based framework for rapid and accurate prototyping of Network-on-Chip-based multi-core embedded systems over FPGA platforms has been developed and will be presented, together with some validation use cases.

- The second section of this thesis will instead focus on simulation of high performance computing systems. The objective of this research activity was to address the current and future trends in simulating high-performance architectures, by exploiting parallel computation to speed-up the simulator performances. In particular, the development of a fast and accurate parallel software simulator for the Cray XMT multithreaded supercomputer will be presented and discussed. The simulator is able to execute the same, native and unmodified, applications of the real machine, by exploiting large SMP commodity servers having an almost constant simulation speed as the number of simulated processor increases, up to the number of available host cores, while maintaining runtime tunable cycle-level accuracy.

Chapter 2

State of the art

In this chapter the major solutions at the state of the art for simulating/emulating both multi-core embedded systems and large-scale high-performance computing machines will be presented and discussed. Particular focus will be put in the solutions that have been relevant to the conception of the base ideas for the research activities described in this document. This chapter will proceed according to the organization of the entire thesis, which features two parallel tracks. Likewise, the first section will address the discussion of the state of the art in hardware-based emulation of embedded computing systems, while the second section will go through the state of the art in parallel software simulators for large-scale high-performance computing machines.

2.1 State of the art in FPGA-based system-level emulation

As already mentioned in chapter 1, the idea of resorting to hardware-based system-level emulators has been proposed in the recent past as a prospective solution to the speed-accuracy trade-off that takes place when simulating highly complex heterogeneous embedded systems, like typical MPSoC platforms usually are. FPGA devices seem to be the most promising platform for the development of hardware-based emulation strategies because of their (re-)configuration capabilities and because the relative toolchains are mature, reliable and rich of debugging features. These tools take in input an HDL description of the logic to be implemented which is actually very similar to the RTL description of the system under design, therefore little effort needs to be done to adapt the code for prototyping. Moreover, their integration capacity, scalability, the relatively low cost and the decreasing power consumption figures suggest FPGAs are going to be the reference platform for hardware-based emulation onto the next years to come [51]. The idea of adopting FPGAs for system-level hardware-based emulation has been around for couple of years now, and putting effort to improve efficiency of the related tools and to foster IP reuse even for emulation purposes has been recognized as crucial to the effective success of on-chip many-cores architectures [7]. In order for this kind of approaches to establish theirself as they promise to do, the availability of open-source modules, from open-access repositories such as Opencores.org [22], Open SPARC [45], and Power.org [4] needs to be strongly encouraged.

2.1.1 RAMP - Research Accelerator for Multicore Processors

Among the different works that addressed FPGA-based system-level emulation, the most famous is the RAMP project. The Research Accelerator for Multiple Processor (RAMP) project is an NSF-funded open-source effort of a group of faculty, institutions and companies (UCBerkeley, UTAustin, MIT, Stanford, University of Washington, Carnegie Mellon University, Intel, IBM, Xilinx, Sun, Microsoft) to create an FPGA-based emulation and computing platform that will enable rapid innovation in parallel software and multicore architecture [46], [53].

The major sources of inspiration that led the researchers in conceiving such project were:

- The difficulty for researchers to build modern chips, due to the high complexity, heterogeneity and strict time-to-market constraints.
- The rapid advance in FPGAs, which are doubling in capacity every 18 months. FPGAs now have the capacity for millions of gates and millions of bits of memory, and they can be reconfigured almost as easily and rapidly as modifying software.
- Flexibility, large scale, and low cost is more important than absolute performance for researchers, as long as performance is fast enough to do their experiments in a timely fashion. This perspective suggested the use of FPGAs for system emulation.
- “Smaller is better” means that many of these hardware modules can fit inside an FPGA today, avoiding the much tougher mapping problems of the past when a single module had to span many FPGAs.
- The increasing success of IP reuse as a design concept, coupled with the increasing availability of open-source RTL modules and standard for inter-module interfacing [43] which can be inserted into FPGAs with little effort.

The project is structured as a cluster of single-target specific efforts. Different platforms have been developed to demonstrate the advantages of FPGA-based emulation. All these platforms shared the same underlying prototyping board, called Berkeley Emulation Engine (BEE), which has also been developed within the RAMP project. A view of the actual prototyping board is provided in Figure 2.1.

The BEE prototyping board contains five Xilinx Virtex-II Pro 70 FPGAs, the latest devices available at design time, each containing two PowerPC 405 hard cores and connected to four independent 72-bit DDR2 banks, capable of a peak throughput of 3.4 GBps. There are four user FPGAs, connected in a 5 GBps ring, and one control FPGA, which is instead connected in a 2.5 GBps star with the user FPGAs. The control and user FPGAs have two and four 10 Gbps high-speed serial I/O links off-board, respectively. These serial links run to 10GBASECX4 connectors, which allow Infiniband, 10 Gb Ethernet or XAUI (Ten-gigabit Attachment Unit Interface) connections over fiber or copper. Finally, a robust set of peripherals, including RS232 and Ethernet transceivers, are connected to the control FPGA, allowing the BEE to run a complete kernel of Linux.

The different RAMP initiatives are the following:

- **RAMP Blue** - led by the Berkeley research unit, this initiative built a large-scale FPGA prototype of a multi-core architecture (up to 1008 cores on 20 BEE boards) out of reusable soft IP cores (Xilinx Microblaze CPUs) that uses MPI-like message passing

Figure 2.1: BEE prototyping board

primitives to run real high performance applications like the NAS parallel benchmarks ([11], [32]) or to simulate the behavior of Internet-based applications using the TCP/IP protocol stack. Further information on the implementation details of this platform can be found in [33].

- **RAMP Red** - also known under the name ATLAS [54], this initiative, led by the Stanford research unit, is the first prototype of a Chip MultiProcessor with hardware support for transactional memory. It embeds 8 PowerPC cores which access coherent shared memory in a transactional manner. The full-system prototype operates at 100MHz, boots Linux and provides significant performance and ease-of-use benefits for a range of parallel applications. Once again, the ATLAS infrastructure provides significant benefits for transactional memory research such as 100x performance improvement over a full software simulation solution.
- **RAMP Gold** - led by the Berkeley research unit, the RAMP Gold prototype [50] is a high-throughput, cycle-accurate full-system simulator that runs on a single Xilinx Virtex-5 FPGA board, and which simulates a 64-core shared-memory target machine capable of booting real operating systems. Because of some of its interesting main features and since it is the only actual FPGA-based framework developed within the RAMP project, we will discuss in detail the RAMP Gold project in the following.

The RAMP Gold simulator

Figure 2.2 shows the structure of RAMP Gold. The most important aspect of this simulation framework is that the timing and functional models of the design under test are implemented in the FPGA but kept separated. The functional model is responsible for executing the target ISA and maintaining architectural state. The timing model determines how much time an instruction takes to execute in the target machine and schedules threads to execute on the functional model accordingly. The interface between the functional and timing models is designed to be simple and extensible to facilitate rapid evaluation of alternative target memory hierarchies and microarchitectures. The distinction between functional and timing models is quite usual in the simulation of multi-core architectures, as can be found in many software simulators [9]. The advantages of such a separation are a simplified FPGA mapping of the functional model and a higher configurability and reuse capability of both the models.

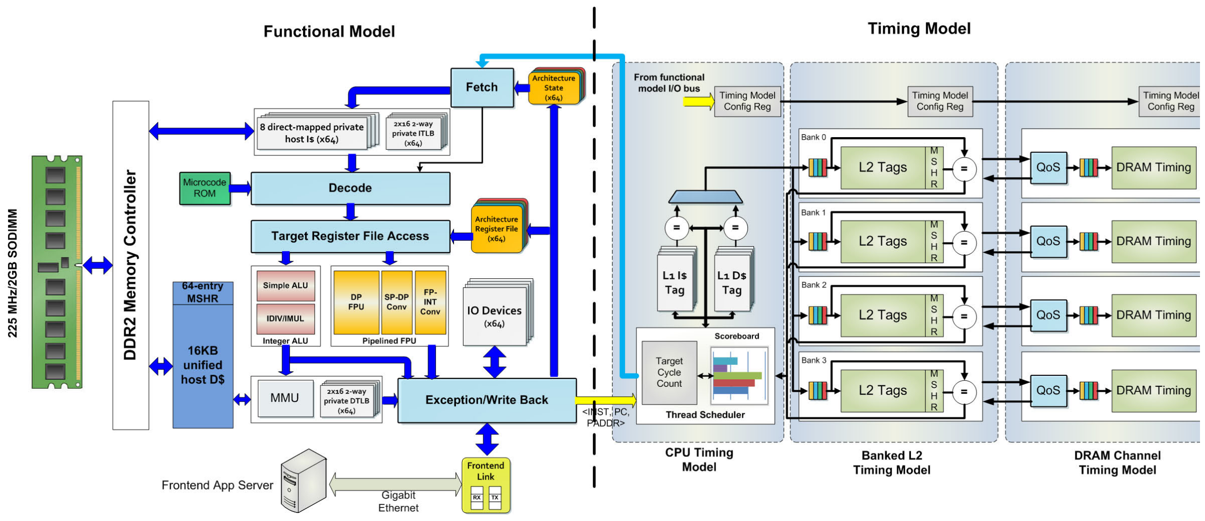


Figure 2.2: RAMP Gold structure

The functional model is essentially a 64-thread feed-through pipeline where each thread simulates an independent target core. The functional model supports the full SPARC V8 ISA in hardware on the FPGA, including floating point and precise exceptions. It also has sufficient hardware to run an operating system, including MMUs, timers, and interrupt controllers. The functional functional model has been validated using the SPARC V8 certification suite from SPARC International, and it can boot the Linux 2.6.21 kernel as well as ROS, a prototype manycore research operating system.

The timing model, on the other hand, tracks the performance of the 64 cores of the simulated target architecture. The base target core is an in-order single issue core that sustains one instruction per cycle, except for instruction and data cache misses. Each target core has private L1 instruction and data caches. The cores share an L2 cache via a nonblocking crossbar interconnect. Each L2 bank connects to a DRAM controller, which models delay through a first-come-first-serve queue with a fixed service rate. Most of the timing model parameters can be configured at runtime by writing control registers that reside on the I/O bus. Among these are the size, block size, and associativity of L1 and L2 caches, the number of L2 cache banks and their latencies, and DRAM bandwidth and latency. To measure target performance, 657 64-bit hardware performance counters have been included in the timing model, and in the functional model to measure the simulator performance itself.

2.1.2 The PROTOFLEX simulator architecture

PROTOFLEX ([20], [27]) is a full-system FPGA-based simulation architecture, proposed by a research team at Carnegie Mellon University led by Professor Eric S. Chung. The two key concepts that characterize the PROTOFLEX approach are hybrid software-FPGA simulation, realized with a mechanism called *transplanting*, and time-multiplexed interleaved simulation. The first instantiation of the PROTOFLEX architecture incorporating the aforementioned concepts is the BlueSPARC simulator, which models a 16-core UltraSPARC III SMP server, hosted on the same BEE2 prototyping platform developed within the RAMP project. Hybrid simulation within BlueSPARC couples FPGA-based emulation with software simulation through the adoption of VirtuTech Simics simulator [34]. Figure 2.3 shows the hybrid simulation functional structure.

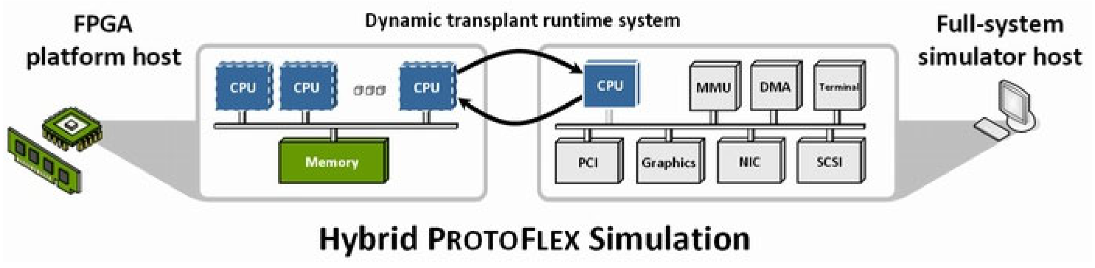

FPGA platform host
Dynamic transplant runtime system
Full-system simulator host
CPU
CPU
CPU
Memory
CPU
MMU
DMA
Terminal
PCI
Graphics
NIC
SCSI
Hybrid PROTOFLEX Simulation

Figure 2.3: Protoflex hybrid simulation architecture

Hybrid simulation is essentially a mechanism originally devised to accelerate complex and unbalanced software simulations. Components in a simulated system are selectively partitioned across both FPGA and software hosts. This technique, called transplanting in PROTOFLEX, is motivated by the observation that the great majority of behaviors encountered dynamically in a simulation are contained in a small subset of total system behaviors. It is this small subset of behaviors that determines the overall simulation performance. Thus, to improve software simulation performance while minimizing the hardware development, one should apply FPGA acceleration only to components that exhibit the most frequently encountered behaviors. Transplanting works by defining a *state* for the simulation, common to the FPGA and software host platforms. This state gets “transplanted” to and from FPGA when an acceleration is needed and possible through specific hardware placed on the FPGA.

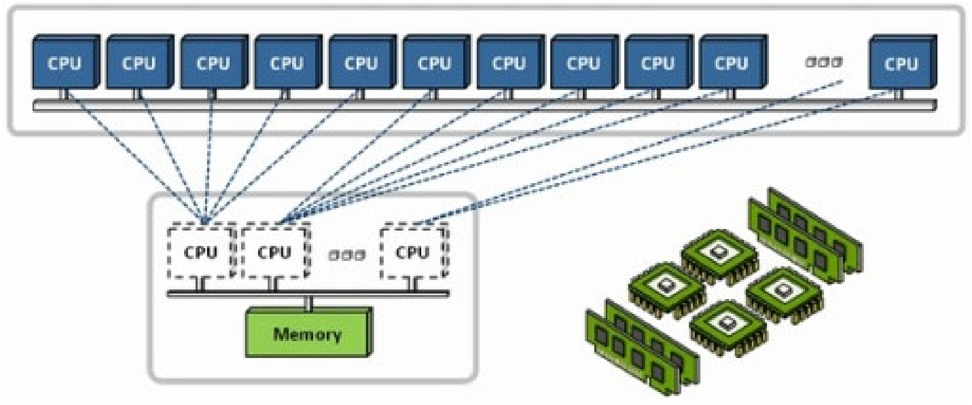
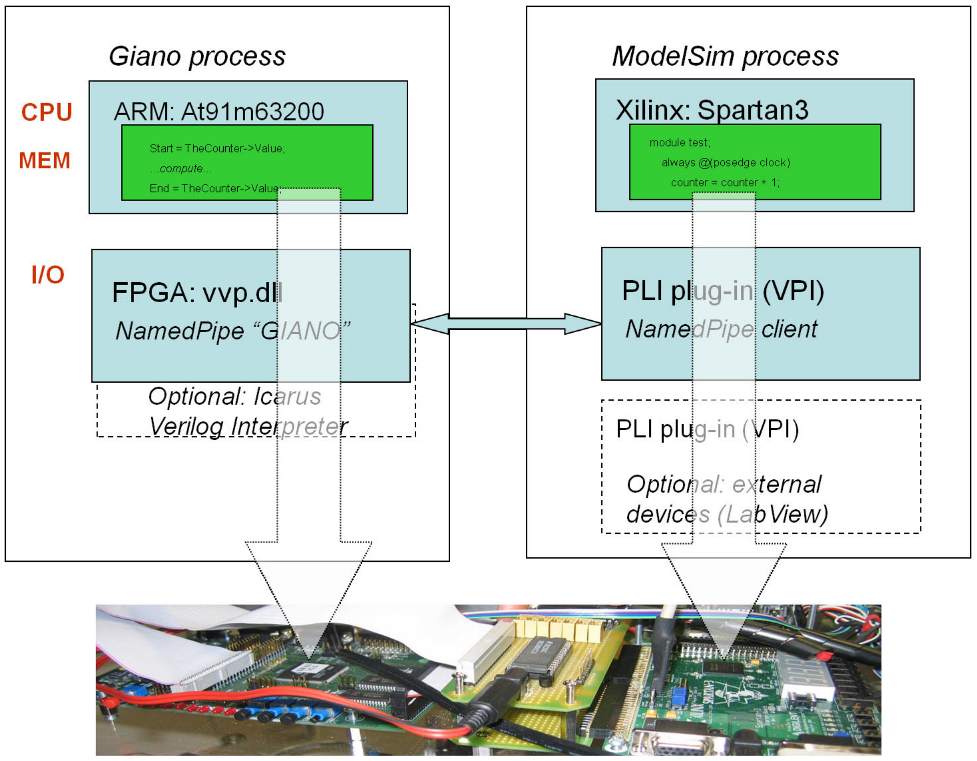


Figure 2.4: PROTOFLEX instruction-interleaved virtualization

The second major feature of the PROTOFLEX simulation architecture is virtualization of the simulation of multiple processor contexts onto a single fast engine through time-multiplexing. Virtualization decouples the scale of the simulated system from the required scale of the FPGA platform and the hardware development effort. The scale of the FPGA platform is in fact only a function of the desired throughput (i.e., achieved by scaling up the number of engines). Figure 2.4 illustrates the high-level block diagram of a large-scale multiprocessor simulator using a small number of interleaved engines. Time-multiplexing is implemented, in PROTOFLEX, by essentially borrowing the concept of Simultaneous Multi-Threading (SMT) from common multi-threaded architectures, and by letting each pipeline simulate instructions from different target cores. Moreover, in the figure, multiple simulated processors in a large-scale target system are shown mapped to share a smaller number of engines.

2.1.3 The Microsoft GIANO simulator

Giano [30] is an extensible and modular simulation framework developed at Microsoft Research for the full-system simulation of arbitrary computer systems, with special emphasis on the hardware-software co-development of system software and Real-Time embedded applications. It allows the simultaneous execution of binary code on a simulated microprocessor and of Verilog code on a simulated FPGA, within a single target system capable of interacting in real-time with the outside world. The graphical user interface uses Microsoft Visio to create the interconnection graph of the user-provided simulation modules in PlatformXML, an XML-based platform description language.



```
Start = TheCounter->Value;
...compute...
End = TheCounter->Value;
```

```
module test;
  always @(posedge clock)
    counter = counter + 1;
```

Figure 2.5: Microsoft GIANO co-simulation structure

Figure 2.5 shows a typical host processor - FPGA device partitioning and the primitives through which the communication between the two worlds is realized. The host code running on the host microprocessor runs a DLL for handling the communication with the FPGA device, which instead instantiates classical I/O devices. The GIANO simulator has been used in different design cases. Its accuracy cannot be defined as cycle-level, since it merely depends on the accuracy level of the modules description and partitioning between FPGA and host microprocessor.

2.1.4 FAST: FPGA-Accelerated Simulation Techologies

FAST is a simulation methodology that aims at defining techniques and mechanisms to produce fast cycle-accurate full-system simulators capable of running unmodified applications and operating systems on top of current instruction sets such as the x86. FAST is another example of partitioned simulators. It defines a speculative functional model component that simulates the instruction set architecture and a timing model component that predicts performance. The speculative functional model normally runs on a host machine on top of a standard software simulator, possibly parallelized, while the timing model is implemented in FPGA hardware for achieving high speed.

The functional model simulates the computer at the functional level including the instruction set architecture (ISA) and peripherals, and executes applications, operating system and BIOS code. The timing model simulates only the microarchitectural structures that affect the desired metrics. For example, to predict performance, structures such as pipeline registers, arbiters and associativity need to be modeled. On the contrary, because data values are often not required to predict performance, data path components such as ALUs, data register values and cache values are generally not included in the timing model. The functional model sequentially executes the program, generating a functional path instruction trace, and pipes that stream to the timing model. It is often the case that the functional path is equivalent to the right path where branches are always correctly predicted. Each instruction entry in the trace includes everything needed by the timing model that the functional model can conveniently provide, such as a fixed-length opcode, instruction size, source, destination and condition code architectural register names, instruction and data virtual addresses and data written to special registers, such as software-filled TLB entries.

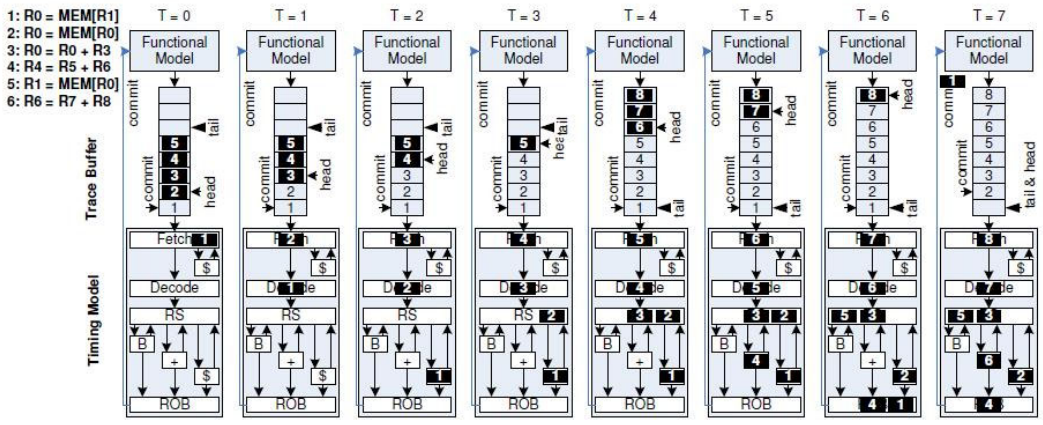


Figure 2.6: FAST operation example with a single-processor architecture

In the example shown in Figure 2.6, the functional model executes and outputs eight instructions to the timing model via the trace buffer (TB). Each logical TB entry contains information used by multiple stages in the timing model and is thus not deallocated until the instruction is fully committed. In that case, the target is a single issue machine with three functional units, ALU (+), Load/Store-DataCache ($) and Branch (B), and the ability to write up to three instructions (one per functional unit) to the ROB per cycle. The timing model first “fetches” from the TB, then cycle-by-cycle processes each instruction by arbitrating for and consuming the required resources in the correct order, thus accurately predicting what would happen in the target microarchitecture.

2.1.5 FPGA-based emulation for thermal modeling

An interesting exploitation of FPGA-based hardware emulation is been proposed by Atienza and others ([24], [8]), at the Laboratory of Integrated Systems at EPFL institute, Losanna. They developed an automatic framework for full-system FPGA-based emulation and then used it to extract in real-time execution traces and pass them to a thermal model, that feed-backs control inputs to a thermal manager unit, implemented in hardware on the FPGA, that applies appropriate thermal management policies to reduce the thermal dissipation into the chip. Moreover, the paper estimates clearly the speed-up obtained by the FPGA-based emulation framework itself over standard software simulation as three orders of magnitude, without accounting for the FPGA synthesis and implementation phase. Figure 2.7 plots the operating flow graph of the FPGA-based framework, when coupled to a thermal modeling and closed-loop control engine.

Although this approach slightly differs from emulation since the thermal modeling and feedback phase is not part of any emulation process, but instead aims at regulating temperature in the actual FPGA die, still it provides some interesting information on how the FPGA-based emulation can be used to obtain (real-time) activity traces that can eventually be used to estimate different metrics, not only purely functional ones (execution time, interconnection congestion, cache hit/miss rate, ...) but also related to physical entities (power and energy consumption, thermal distribution within the die, ...).

2.1.6 Lessons learned from the state of the art

All the different approaches presented and discussed so far have led us to make some considerations on the use of FPGA devices for system-level emulation, on the advantages and disadvantages, on the opportunities offered by these devices, that we want to summarize in the following list of features:

- *speed* - FPGA devices can normally run, at the state of technology, at 100-200 MHz speed. Although 10 times slower than an average processor operating frequency, this speed is still high enough to overcome simulation speed, without accounting for FPGA synthesis and implementation overhead.

- *scalability* - FPGA devices currently integrates in a single chip up to couple of million equivalent gates, and technology promises to scale in the near future. Therefore, these devices will be soon able to integrate many cores and peripherals, facing the growing complexity of embedded systems.

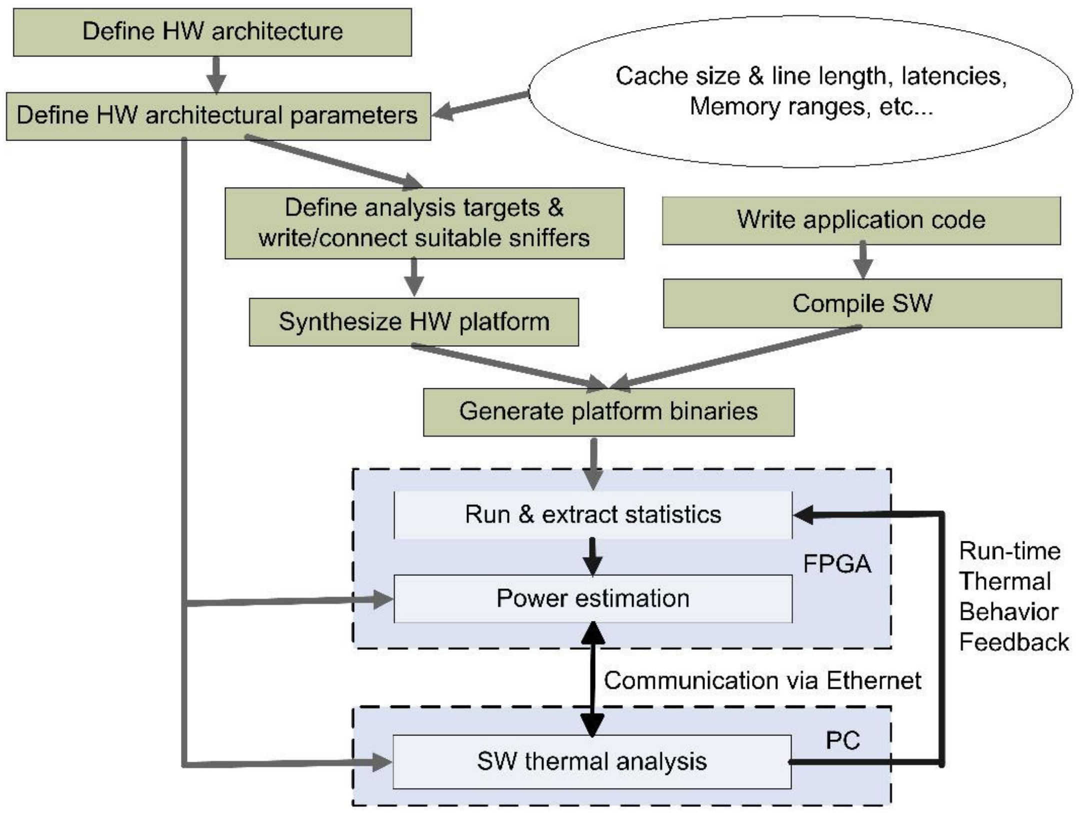

Define HW architecture
Define HW architectural parameters
Cache size & line length, latencies, Memory ranges, etc...
Define analysis targets & write/connect suitable sniffers
Write application code
Synthesize HW platform
Compile SW
Generate platform binaries
Run & extract statistics
FPGA
Power estimation
Run-time Thermal Behavior Feedback
Communication via Ethernet
PC
SW thermal analysis

Figure 2.7: FPGA-based emulation framework with thermal modeling

- *toolchain* - The toolchain for FPGA synthesis and implementation are quite mature and, although they still are the real bottleneck in the implementation flow in terms of time-consumption, designer's life is really simplified. Nevertheless, research to mitigate the effect of synthesis and implementation on the overall FPGA development time needs to be carried on.
- *reusability* - IP core reusability is an inherent feature of hardware-based emulation. Creation of open-access module libraries and repositories needs to be encouraged in order to enable effective system-level framework development.
- *state monitoring* - FPGA devices can really provide real-time monitoring of signals and architectural states. The number of chip I/O pins is still a limiting factor, though.
- *co-simulation* - Several solutions exist at the state of the art for FPGA-based accelerated co-simulation. Often the simulation is partitioned between timing and functional models, respectively running on the FPGA device and on a host processor through software simulation. Research still needs to be done on efficient interfacing of the two worlds and on partitioning.
- *physical-awareness* - FPGA devices can enable the extraction of real-time activity traces that can be passed to dedicated models for estimation of physical metrics such as power/energy consumption or thermal distribution.

2.2 State of the art in parallel software simulators for high-performance computing

2.2.1 The Wisconsin Wind Tunnel II (WWTII) simulator

The Wisconsin Wind Tunnel II simulator [40] is a parallel software simulator explicitly designed for multi-core target processors. It is the successor to the famous WWT [17] software simulator, which was designed to run on a single host processor and to simulate mainly uni-processor architectures. WWT employed instrumented direct execution to calculate the execution times of the target machines. Direct execution is one of the main techniques to emulate target ISA and binary code on top of a host machine with a different ISA, by basically providing a low-level translation of the target elementary instructions on the host machine ISA. The WWT II simulator is able to run parallel simulations of large multi-core systems on Sparc platforms ranging from workstation clusters (COW) to symmetric multiprocessor architectures (SMP).

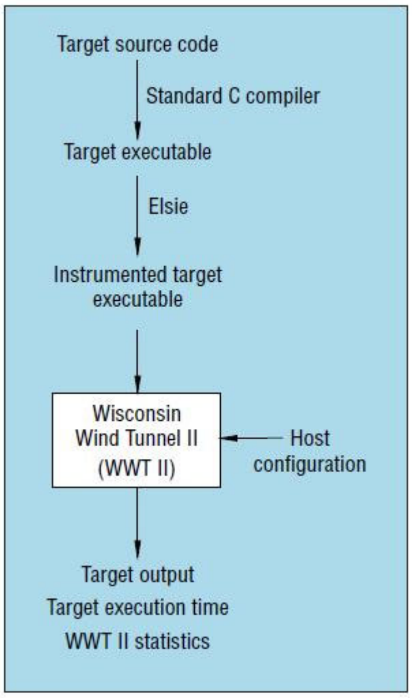


Figure 2.8: Elsie's relation to the WWTII simulation framework.

The main features of the WWTII simulator are summarized in the following list:

- *direct execution* with executable instrumentation to calculate target architecture execution times. For instance, insertion of instruction to handle specific clock variables is performed on the target binary code. WWTII developers used pre-existing binary editing libraries to build an executable editor tool named Elsie (Edits Loads and Stores In Executables), to perform the target clock instrumentation on target executables. Elsie also replaces target memory instructions (loads and stores, for example) with snippets that jump into the simulator, which simulates the target memory subsystem. Figure

2.8 shows how target binary code instrumentation is related to the whole WWTII emulation framework.

- *emulation of host missing features* that are instead required by the target binary code. This feature is essential and must be implemented in each direct execution based simulator, since it is very likely that not all the target system instructions might be traduced into host instructions for direct execution. A classical example of such need is when shared-memory target machines have to be simulated on top of distributed memory host machines. Nevertheless, emulation of missing features increases the simulator generality, thus resulting in improved portability. The WWTII approach replaces target memory reference instructions with code segments that transfer control to the simulator itself for execution. Emulation is implemented through the Elsie tool as well.

- *Communication of target inter-processor messages* is implemented through a specific messaging library, called SAM, which abstracts the target machine architectural details, and therefore enables the simulator to be run on distributed machines (which generally use the MPI library over distributed memory), on COW (which generally use Sockets) and on SMP (which generally use shared-memory interprocess communication primitives).

- *Synchronization of host processors* is handled through lax time quantum based barriers. This means that the host processors do not necessarily synchronize at every clock cycle, but once every N target clock cycles. N must be tuned at simulation configuration time according to the host machine architecture, since it affects the resulting accuracy. Figure 2.9 plots a sample synchronization case with 4 host processors. The synchronization is implemented inside the SAM library as well.

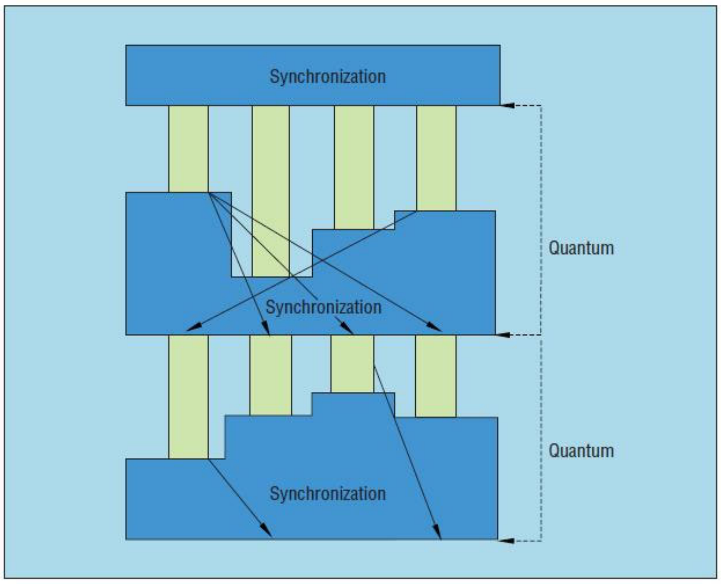


Figure 2.9: Quantum-based host processors synchronization

2.2.2 The COTSon simulator

COTSon [39][28] is a multithreaded simulator framework for many-core architectures jointly developed by HP Labs and AMD. It splits simulation in fast functional emulation and timing models, cooperating to improve at runtime the simulation accuracy at a speed sufficient to simulate the full stack of applications, middleware and OSs. It basically abandoned the idea of "always-on" cycle-level simulation in favor of statistical sampling approaches that can trade accuracy for speed.

Functional simulation "emulates" the behavior of the target system, including the OS and common devices such as disks, video, or network interfaces. An emulator is normally only concerned with functional correctness, so the notion of time is imprecise and often just a representation of the wall-clock time of the host. COTSon uses AMD´s SimNow simulator [25] for the functional simulation of each node in the cluster. The SimNow simulator is a fast and configurable x86 and AMD64 platforms simulator for AMD's family of processors. It uses dynamic compilation and caching techniques to speed up CPU simulation.

Timing simulation is used to assess the exact performance of a system. It models the operation latency of devices simulated by the functional simulator and assures that events generated by these devices are simulated in a correct time ordering. Timing simulations are approximations to their real counterparts, and the concept of accuracy of a timing simulation is needed to measure the fidelity of these simulators with respect to existing systems. COTSon architecture relies on the assumption that absolute accuracy is not always strictly necessary and in many cases it is not even desired, due to its high engineering cost. In many situations, substituting absolute with relative accuracy between different timing simulations is enough for users to discover trends for the proposed techniques. In order to implement timing simulation in COTSon, HP Labs and AMD have jointly augmented the SimNow simulator with a double communication layer which allows any device included in the target architecture to export functional events and receive timing information from them. All events are directed by COTSon to their timing model, which is selected by the user. Each timing model may describe which events it is interested in via a dynamic subscription mechanism. There are two main types of device communication: synchronous and asynchronous.

Figure 2.10 shows an overview of the COTSon architecture.

Considering the aforementioned separation between functional and timing simulation, COTSon architecture is *functional-directed* (also called trace-driven) meaning that it lets the functional simulation produce an open-loop trace of the executed instructions that can later be replayed by a timing simulator, in case a higher accuracy is desired. The functional simulation inserts extra code in the instruction cache that dynamically generates events for the timing modules. The cycle-accurate simulation of these events, namely instructions and memory accesses, involves fetching the instruction, decoding it, renaming it, and so on.

One of the main features of the COTSon simulation framework is statistical sampling, through which timing simulation phases are triggered. Sampling consists of determining what are the interesting or representative phases of the simulation and just simulating those. The results from these samples are then combined to produce global simulation results. Sampling is central to asynchronous devices. The sampler is selected by the user per experiment and is responsible for deciding the representative phases of execution. It does so by selecting the type of the current sample and its length, i.e., COTSon asks the sampler what to do next and for how long. The sampler may reply with a command to enter one of four distinct phases: *functional*, *simple warming*, *detailed warming* and *simulation*. The two

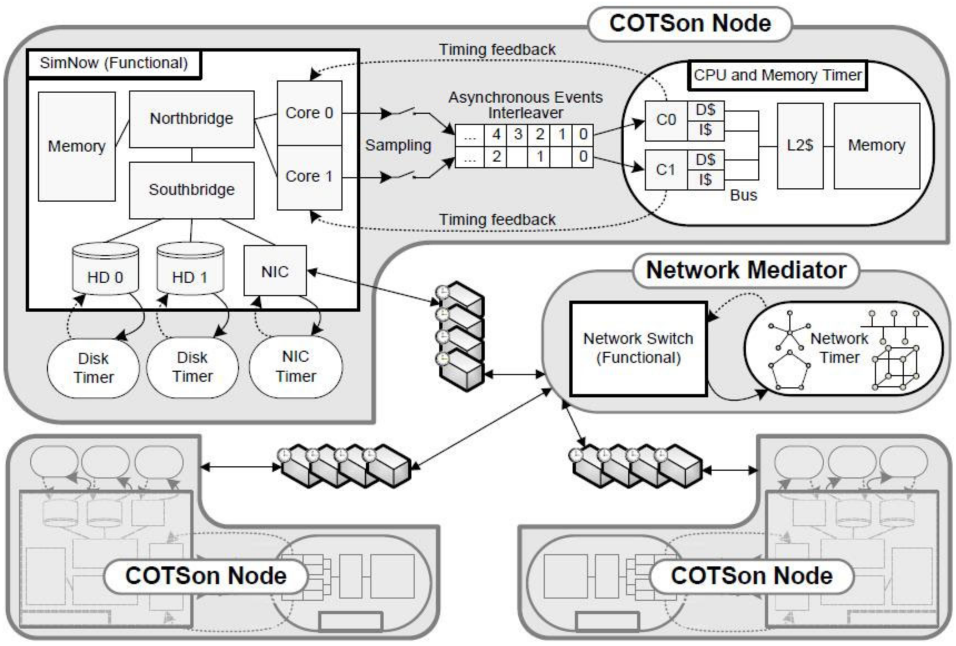


Figure 2.10: The COTSon simulation framework architecture

warming phases are necessary to warm up the timing phases, cleaning the caches, branch target buffers, reordering buffers, renaming tables.

2.2.3 The Graphite simulator

The Graphite simulator [38] is a parallel simulation framework developed at the Massachusetts Institute of Technology. It provides both functional and performance modeling for cores, on-chip networks, and memory subsystems including cache hierarchies with full cache coherence. The design of Graphite is modular, allowing the different models to be replaced to simulate different architectures or tradeoff performance for accuracy. Graphite runs on commodity Linux machines and can execute unmodified Pthread applications. Figure 2.11 plots the high-level organization of the Graphite simulation infrastructure, together with the basic target thread - host thread mapping. Every target core is mapped onto a host thread of execution. Multiple threads then run within a single host process and then share a address space. The different host processes communicate at low-level through TCP/IP Socket technology.

Among the different features of Graphite, we now recall and discuss the following: direct execution, multi-machine distribution, analytical modeling and lax synchronization.

- *direct native execution* is implemented on the host machine to increase the performance during functional modeling of the computing cores. Through dynamic binary translation then, Graphite adds new functionality (e.g., new instructions or a direct core-to-core messaging interface) and intercepts operations that require action from the simulator (e.g., memory operations that feed into the cache model). There is no need to recompile target applications or binary codes for different configurations of the simulation system. Application threads are executed under a dynamic

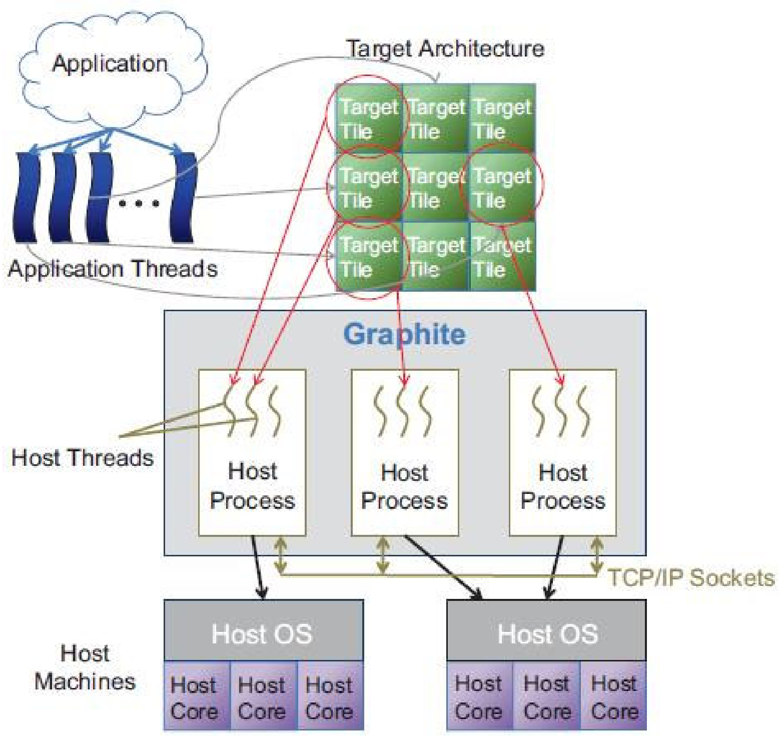


Figure 2.11: Graphite high-level organization

binary translator which rewrites instructions to generate events at key points. These events cause traps into Graphite´s simulation backend which contains the compute core, memory, and network modeling modules. Points of interest intercepted by the dynamic binary translator include memory references, system calls, synchronization routines and user-level messages.

- *multi-machine distribution* is allowed to accelerate simulation and enable the study of large-scale multicore chips. This ability is completely transparent to the application and programmer. Threads in the application are automatically mapped and distributed to cores of the target architecture spread across multiple host machines. The simulator maintains the illusion that all of the threads are running in a single process with a single shared address space.

- *analytical timing models* take in input specific instructions and events from the core, network and memory subsystem and update individual local clocks in each core.

- *target cores synchronization* is handled through message timestamps. However, to reduce the time wasted on synchronization, Graphite does not strictly enforce the ordering of all events in the system. In certain cases, timestamps are ignored and operation latencies are based on the ordering of events during native execution rather than the precise ordering they would have in the simulated system.

- *host threads creation and synchronization* is handled in different manners. Graphite implements a threading interface that intercepts thread creation requests from the application and seamlessly distributes these threads across multiple hosts. Host cores synchronization supports several strategies that represent different timing accuracy and simulator performance tradeoffs: lax synchronization, lax with barrier synchronization, and lax with point-to-point synchronization. Lax synchronization is the most permissive in letting clocks differ and offers the best performance and scalability. Lax with Barrier Synchronization implements quanta-based barrier synchronization (LaxBarrier), where all active threads wait on a barrier after a configurable number of cycles. Lax with Point-to-point Synchronization implements a synchronization scheme called point-to-point synchronization (LaxP2P). LaxP2P aims at achieving the quanta based accuracy of LaxBarrier without sacrificing the scalability and performance of lax synchronization. In this scheme, each tile periodically chooses another tile at random and synchronizes with it.

2.2.4 The BigSim simulator

The BigSim parallel simulator [58][19] is a project developed at the University of Illinois at Urbana-Champaign, based on the CHARM++ parallel programming system [57]. The authors claim to be able to emulate effectively very large scale machines with oversubscription indices of up to 1000 (e.g. simulating a 1M-core target machine on a 1K-core cluster). This inaccurate emulation mode, however, does not provide any performance estimation on the target architecture. The basic idea of the BigSim simulation infrastructure involves letting the emulated execution of the application code proceed as usual, while concurrently running a parallel algorithm that corrects time-stamps of individual messages. In the programmer´s view, each host node consists of a number of hardware-supported threads with common shared memory. A runtime library call allows a host thread to send a short message to a destination target node. The header of each message encodes a handle function to be invoked at the destination. A designated number of threads continuously monitor the incoming buffer for arriving messages, extract them and invoke the designated handler function. The simulation philosophy is called Parallel Discrete Event Simulation (PDES), and provides two different operating modes, online (direct execution) mode and postmortem mode event simulation. Online mode simulation runs the parallel simulation along with the actual execution of the applications. The advantage of this online direct execution approach is that it makes possible to simulate programs that perform runtime analysis.

Figure 2.12 plots the architecture of the BigSim simulator.

The physical target processors are mapped to logical processors (LPs), each of which has a local virtual clock that keeps track of its progress. In the simulation, user messages together with their subsequent computations play the role of events. The different LPs are synchronized through an optimistic synchronization approach, which mainly consists of:

- *Checkpointing overhead* - time spent in storing program state before an event is executed which might change that state.
- *Rollback overhead* - time spent in undoing events and sending cancellation messages.
- *Forward execution overhead* - time spent in re-executing events that were previously rolled back.

Regarding timing simulation of the computational core, three approaches are implemented, with different accuracy levels and complexity. First, user supplied expressions for

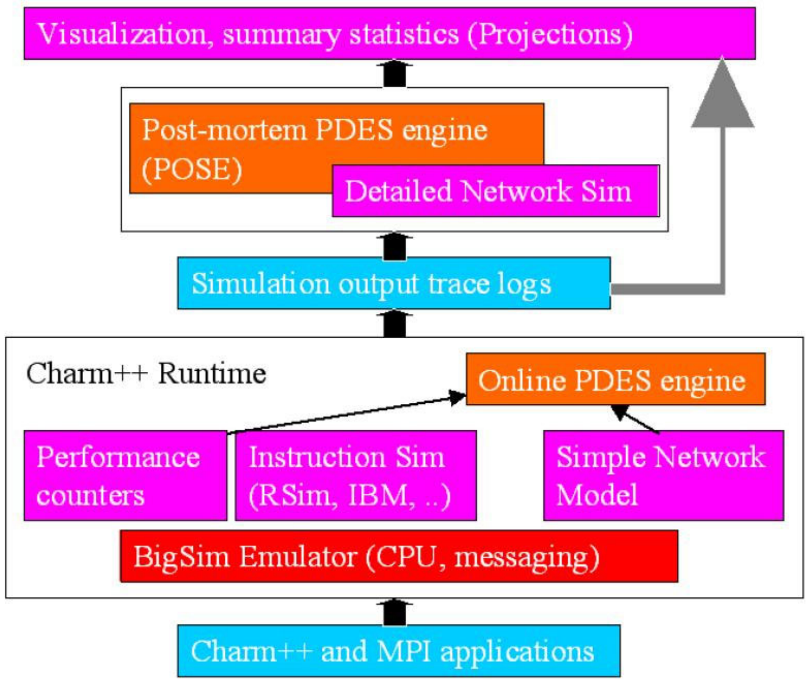

Visualization, summary statistics (Projections)
Post-mortem PDES engine (POSE)
Detailed Network Sim
Simulation output trace logs
Charm++ Runtime
Online PDES engine
Performance counters
Instruction Sim (RSim, IBM, ..)
Simple Network Model
BigSim Emulator (CPU, messaging)
Charm++ and MPI applications

Figure 2.12: The BigSim simulator architecture

every block of code estimating the time that it takes to run on the target machine can be provided. Second, wall clock measurements of the time taken on the host machine can be used via a suitable multiplier (scale factor) to obtain the predicted running time on the target machine. Third, with the highest accuracy, hardware performance counters on the host machine can be used to count floating point, integer and branch instructions (for example), and then to use a simple heuristic to give the predicted total target machine computation time. Cache performance and memory footprint effects can be approximated by percentage of memory accesses and cache hit/miss ratio.

Regarding network performance prediction, static latency estimation is performed as a default solution in the simulator. This estimation merely depends on architecture- and topology-dependent factors, and does not take into account any kind of network contention. A contention-aware model is also usable, and described in [19], within the BigNetSim network simulation environment.

2.2.5 Lessons learned from the state of the art

The analysis of the state-of-the-art solution carried on so far has led us to identify the main features that a modern parallel software simulator aimed at simulation of large scale computing machines should provide. These features also define the most important choices that the simulator developer has to make in designing his/her simulator.

- *direct execution* - the simulator could provide direct execution of target binary code on the host machine. It is usually implemented through assembly-level translation of single instructions from the target ISA to the host one. It usually improves performance but also reduces simulator generality and portability over different host platforms.

- *separation between functional and timing models* - different accuracy levels are usually implemented and included in different sections of the simulator, namely the functional and timing parts. This is considered a best-practice since cycle-accuracy for such large scale machines is simply not feasible *everywhere* and *everytime* during the simulation time. Consider also dynamic accuracy switching policies for further performance scaling.

- *host threads/cores synchronization* - this is an issue that every simulator designer has to face. Parallel simulators run on parallel machines with different architectures and memory organizations such as MPPs, CoWs, SMPs, therefore this problem has to be solved according to the selected platform. Does affect the portability if not handled appropriately.

- *target threads/cores mapping onto host threads/cores* - this is another must-face issue. Affects the choice on host cores synchronization and performance in general. Even trivial solution (1 target thread to 1 host thread) can be satisfactory.

- *customization* - this is probably the most controversial feature. A simulator usually aims at being as general and widely applicable as possible, but some architectures specifically require simulator tuning, especially in a field where performance matters this much, as high-performance machines simulation.

Chapter 3

An FPGA-based framework for technology-aware system-level emulation

As described in Section 1, the increasing complexity in embedded system design poses the need for comprehensive tools that could facilitate accurate and fast explorations of such huge design spaces, by providing rapid functional performance exploration and design characterizations by a physical/technological viewpoint. The high flexibility of modern reconfigurable devices is often considered a viable way to speed-up emulation of such systems, while analytic modeling has been demonstrated to help in physical system characterization, by targeting classic features such as power consumption, area obstruction, operating frequency estimation for the advanced design steps. The basic approach of this research activity is to concurrently explore the adoption of such techniques to achieve rapid system-level emulation and provide, at the same time, sufficiently accurate modeling of the main physical features of the design of interest. If successful, such an approach would considerably shorten the whole design flow traversal time, by condensating in a single multi-objective exploration phase a number of previously separate design phases, such as functional performance estimation and physical characterization. Although not completely accurate, technology-aware system-level simulation could help the designers in reducing the number of design refinement iterations and the occurrence of potential late costly design modifications. This chapter will therefore describe the research activity performed in the field of FPGA-based technology-aware system-level emulation of embedded multicore systems. In particular, we will discuss the conception and the development of a semi-automatic framework for the system-level FPGA prototyping of multi-core platforms, able to provide detailed functional information on the platform under development and physical metrics on power consumption, maximum operating frequency and area occupation of a prospective ASIC implementation of the system.

As already hinted in Section 2, the analysis of the most relevant solutions at the state of the art suggested a number of critical features that tend to appear in modern hardware-based simulation systems. The main ideas that led us in developing our emulation framework are listed and discussed as follows:

- *extensibility and library-based approach* - the framework should base upon re-usable IP modules with standard interfaces, in order to be able to emulate as many different systems as possible. By using standard interface modules the framework would also gain in extensibility. This features requires to develop an instantiation engine that relies on libraries of modules, described at RTL, targeting FPGA devices. A standard interface for communication among the different modules should be defined as well.

- *analytic RTL-trace based models* - the framework should be able to provide information related to the eventual physical implementation of the system under design. In order to do so, different physical models have to be developed for the different selectable ASIC processes which the actual realization will rely on. These models should at least target system-level metrics such as power and area consumption or operating frequency. The models aiming at evaluating dynamic metrics, such as dynamic power consumption, should be able to take as input traces extracted from the FPGA execution of an RTL system description.

- *system-level description input* - The number of necessary interactions between the framework and the designer should be minimized. Further tuning of the design could always be applied by modifying the prototyped system at any phase of the design. The designer should be able to obtain a first set of emulated metrics already at the earliest design steps, hence inputing a properly instrumented system-level platform description. By system-level, we mean that the building blocks of the platform configurations should have the granularity of the processing elements, with basic interconnection architecture and topology description. A basic organization of the address mapping of the different memorization cores and I/O blocks should be provided as well.

- *counter-based performance extraction* - Functional performance extraction should be provided. Classic functional metrics are application execution times, number of memory accesses, average cache efficiency and related hit ratio, congestion of the different interconnection modules, interrupt event tracing. This is the typical kind of information that allows the designer to choose the best system-level configuration and architectural parameters. Such information might be extracted with the development of hardware performance counters and monitors that can be instantiated and pinpointed in a semi-automatic way, by providing explicit declaration at system-level description.

- *minimize synthesis-flow overhead* - The emulation speed is crucial, in order to overcome the limits posed by classic software-based simulation. FPGA-based hardware emulation has proven to be effective to this aim, but still it is affected by the time necessary to traverse the entire synthesis and implementation flow. One could argue that using such approaches inside a design exploration loop or with generic iterative refinement processes, this overhead could be unaffordable. In order to solve this issue, approaches aimed at minimizing this overhead in design exploration contexts have to be developed.

- *capability of running complete sw stacks* - The emulated platform should be capable of running complete software application as well as selected computation kernels. In order to do so, standard compilation toolchains should me made available to produce binary code for the selected processing elements. In addition, a set of communication libraries must be developed to enable compliance with the de-facto standard models of computation. Such libraries must build an Hardware Abstraction Layer (HAL) to abstract the underlying hardware architectural details and implement the effective communication according to the application model of computation.

3.1 Framework Overview

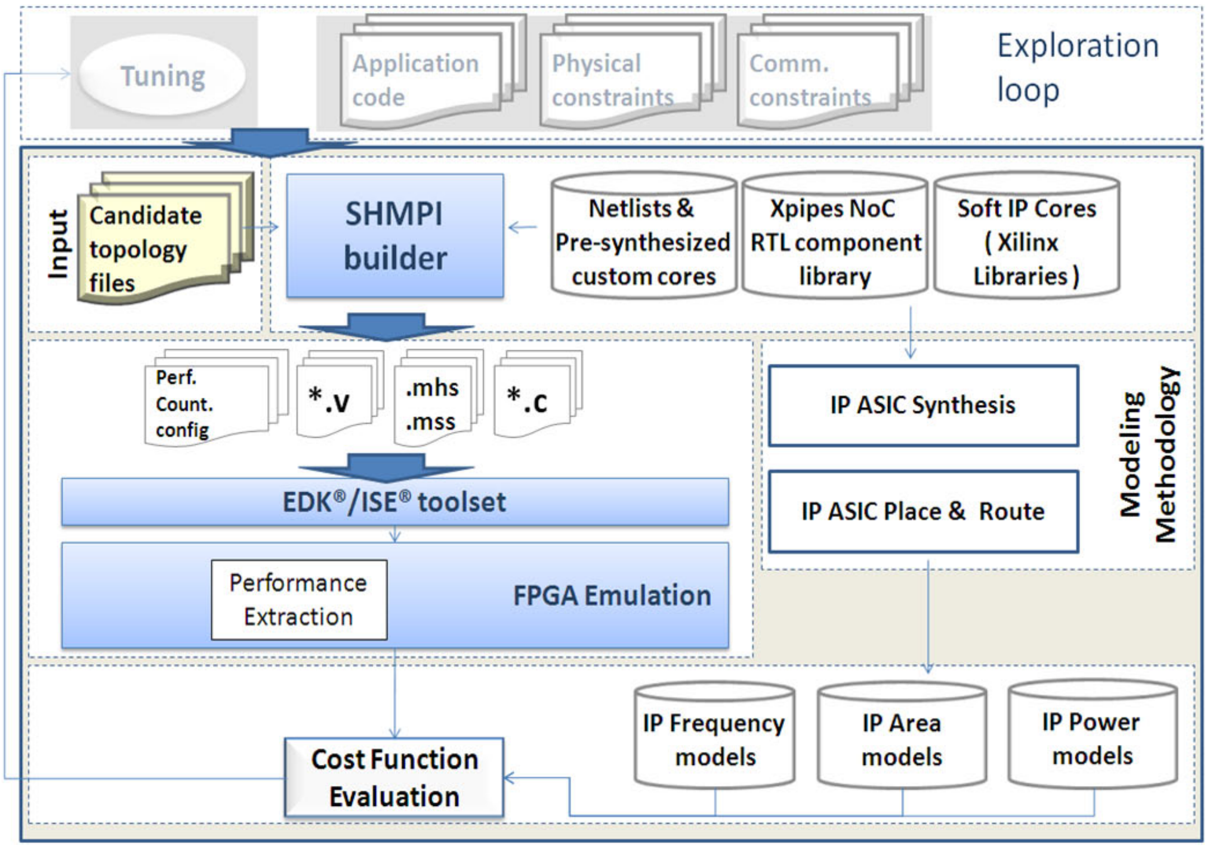


Figure 3.1: Overview of the framework

Figure 3.1 gives a block view of the framework, when employed in a possible design space exploration loop. It should be noted that this is not the primary purpose of this research activity, neither it is the aim of this document. Instead, the application for design space exploration purposes is only one of the possible use scenarios for the proposed framework. The inputs of the whole exploration would be the application code and the constraints to be satisfied by the final system, either physical (area, power, frequency) or functional (latency, bandwidth, execution time). The designer specifies, within a topology text file, a list of all the cores to be instantiated in the system, along with the specific interconnection description, architectural parameters and the different memory address spaces organization. To date, all the applications tested in the benchmarks have been parallelized according to a shared-memory model of computation, employing an in-house implemented library for multi-tasking on multi-processor systems. The parallelization in tasks is fully parametric, enabling the creation of as much tasks as it is needed. All the synchronization is handled through the explicit use of specific hardware semaphores for regulation of the accesses to the shared memory. A detailed description of such modules will be provided in the following.

A high-level topology compiler is in charge of parsing the topology description file and generating the RTL files that describe the hardware top view of the complete platform. This stage is also intended to generate the hw/sw platform description files to be passed to the Xilinx EDK©platform instantiation toolset. This phase of the flow will be specifically addressed and further explained in Section 3.4. The composition and configuration of the selected platform builds upon a repository of soft IP cores. The content of this library will be described in detail in Section 3.3. The use of these repositories does not prevent the inclusion of further modules into the system, since custom cores can always be added as RTL or pre-synthesized netlists with little effort. This is a crucial feature of the RTL libraries, since extensibility is key to the applicability range of the framework. That is also the reason why all the modules included in the library are fully compliant to a common interfacing standard, which is a subset of the well-known OCP-IP communication standard [43], as we will discuss in the following sections.

Regarding the software part of the system, the Xilinx development environment includes the standard compilation and debugging tool-chains for the soft processors that can be instantiated, while the drivers of the peripherals are automatically generated already at the platform compilation stage, according to the parameters provided by the user. The RTL files of the components can potentially be passed, with minor modifications, to an ASIC synthesis flow, in order to obtain power and area figures of the designed platform and to refine the area and power models of the building blocks instantiated within the system. So far, this capability is provided only for the interconnection modules. The framework operating flow proceeds with the FPGA synthesis and implementation through the adoption of the Xilinx proprietary tools (within the Xilinx ISE©environment). The execution of the targeted application on the configured FPGA can be easily profiled with deep accuracy. Moreover, the emulation of the complete platform enables the rapid collection of cycle-accurate information on the switching activity, that can be used, in cooperation with the available power models, to obtain detailed figures related to a prospective ASIC implementation of the system. An example of such kind of models is described in Sect. 3.6.

3.2 System-level platform description input file

The designer is able to input a platform description by passing in input to the framework a text file that describes the main system-level building blocks of the design under emulation. The granularity of the input file is at the single processing, interconnection, I/O and memorization module level. Regarding, the interconnection subsystem architecture, the designer can describe, in case a source routing NoC-based is selected (which is the case of the Xpipes interconnection library), the number of switches, the n-arity of each switch, the number of buffering input/output stages, the routing tables and normally has to tag the different links connecting the switches. Regarding the memorization layer, the type of memory has to be selected. Currently the system is able to instantiate on-FPGA Xilinx proprietary hard BRAM modules, configuring them to emulate single- and double-port memory cores. The address space and the related processor into which the memory module is connected have to be specified. If the memory is shared among the different processing element, the module has to be declared to be such. Regarding the different I/O and synchronization controller and modules, the address spaces have to be declared as well.

The following code contains snippets taken from an actual input system description file:

```
// ----------------------------
// define the topology here
// name, mesh/torus specifier (mesh/torus/other)
// ----------------------------
topology(topology_2x2, other);

// ----------------------------
// define the cores here
// core name and number, switch number, NI clock divider, NI buffers,
// initiator/target type, type of core (if a specific one is requested),
// memory mapping (only if target), fixed specifier
// (only if target and of shared type)
// ----------------------------
core(core_0, switch_0, 1, 6, userdefined, initiator);
core(core_1, switch_1, 1, 6, userdefined, initiator);
core(core_2, switch_2, 1, 6, userdefined, initiator);
core(core_3, switch_3, 1, 6, userdefined, initiator);

core(pm_4,   switch_0, 1, 6, double, target:0x10,high:0x1000ffff);
core(pm_5,   switch_1, 1, 6, double, target:0x12,high:0x1200ffff);
core(pm_6,   switch_2, 1, 6, ocpmemory, target:0x14,high:0x1400ffff);
core(pm_7,   switch_3, 1, 6, ocpmemory, target:0x16,high:0x1600ffff);

core(ts_8,   switch_3, 1, 6, Testandset,target:0xff,high:0xffffffff);
core(ul_9,   switch_0, 1, 6, Uartlite,target:0x46,high:0x4600ffff);
core(shm_10, switch_1, 1, 6, shared, target:0x06, high:0x0600ffff);

// ----------------------------
// define the switches here
// switch number, switch inputs, switch outputs, number of buffers,
// core ID to which the switch performance counter is attached,
// port ID to which the switch performance counter is attached.
// ----------------------------
switch(switch_0, 5, 5, 6, 0, 0);
switch(switch_1, 5, 5, 6, 1, 0);
switch(switch_2, 5, 5, 6, 2, 0);
switch(switch_3, 5, 5, 6, 3, 0);

// ----------------------------
// define the links here
// link number, source, destination
// ----------------------------
```

```
link(link0,  switch_0,  switch_1);
link(link1,  switch_1,  switch_0);

.
.
.

link(link6,  switch_1,  switch_3);
link(link7,  switch_3,  switch_1);

// -----------------------------
// define the routes here
// source core, destination core, the order in which switches need to be
// traversed from the source core to the destination core
// -----------------------------
route(core_0, pm_4,   switches:0);
route(core_0, ts_8,  switches:0,1,3);
route(core_0, ul_9,  switches:0);
route(core_0, shm_10, switches:0,1);

.
.
.

route(core_3, pm_7,   switches:3);
route(core_3, ts_8,  switches:3);
route(core_3, ul_9,  switches:3,2,0);
route(core_3, shm_10, switches:3,1);

route(pm_4,  core_0, switches:0);
route(pm_4,  pm_5, switches:0,1);
route(pm_4,  pm_6, switches:0,2);
route(pm_4,  pm_7, switches:0,1,3);

.
.
.

route(pm_7,  core_3, switches:3);
route(pm_7,  pm_4, switches:3,2,0);
route(pm_7,  pm_5, switches:3,1);
route(pm_7,  pm_6, switches:3,2);

.
.
.
```

```
route(ts_8,  core_0, switches:3,2,0);
route(ts_8,  core_1, switches:3,1);
route(ts_8,  core_2, switches:3,2);
route(ts_8,  core_3, switches:3);
```

The code snippet contains the description of a sample NoC-based topology named `topology_2x2`, which contains 4 processing elements, 2 double-port local memories, 2 single-port local memories, a test&set synchronization module, an UART controller for interfacing with a serial port and a shared memory, all declared with the same `core()` primitive and different parameters. The parameters allow for specifying different clock domains (unused in this case), the number of buffering stages in the network interface (6 in this case), the core identifier and the basic memory mapping of each device. Local memories with multiple ports have different mappings for each port with respect to the local processor and the other processors, if direct messaging is enabled.

The NoC switches are defined using the `switch()` primitive, whose parameters allow the designer to specify a name for the switch, the number of inputs and outputs of that switch, which define the arity of the switch itself, the number of buffering stages (output buffering is used in this interconnection library module) and two other IDs. These two other IDs respectively identify the core and the port which the switch performance counters are actually attached to. The actual performance counter logic will be described in detail in Section 3.5. The `link()` primitive enable link identification. Its parameters simply point to the interconnected switch modules. Links have to be declared as if they were half-duplex, meaning that between two switches connected through a full-duplex link there must be two separate links. In addition to that, if source routing is used, the `route()` primitive enables specifying the routing path between every core pair inside the network, and its parameters specify the list of switch traversed by the packets from source to destination. Direct communication between two computing cores is not allowed. If direct messaging is intended to happen at higher level, the underlying mechanism implies a communication from the source processing element to the remote memory that is attached to the destination processing element.

3.3 The soft IP cores RTL repository

This section will describe the components currently available in the soft IP cores repository. All the modules included in the libraries are highly parametric. This is a key feature with respect to the prospective adoption of the framework for design exploration purposes. Different computing and memorization elements have being recently added to the libraries that will not be described within this document. The building blocks have been made compliant, where not already done, to a subset of the well known OCP open communication standard [43]. We will first provide a brief introduction to the main features of the standard.

3.3.1 Elements of the Open Core communication Protocol (OCP)

The Open Core Protocol (OCP) defines a bus-independent between IP cores that reduces design time, design risk, and manufacturing costs for SOC designs. An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. The OCP defines a single-clock synchronous interface point-to-point interface between two communicating entities such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave. The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP.

A transfer across this protocol occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a wrapper interface module that abstracts the underlying hardware details). The interface module plays the request across the on-chip physical interconnection system, which can be a bus, a crossbar, a NoC or whatever. The OCP does not specify the underlying hardware functionality. Instead, the interface designer converts the OCP request into a transfer for the underlying hardware. The receiving wrapper interface module (as the OCP master) converts the underlying hardware transfer operation into a legal OCP command. The system target (OCP slave) receives the command and takes the requested action.

The OCP standard defines a huge number of signals, which are grouped in different categories, according to the functionality that the different interconnected modules provide. We chose to implement only compliance with a small subset of the OCP signals, whose list is reported in Table 3.1 together with a brief signal functionality description.

Name	Width	Driver	Function
Clk	1	varies	OCP clock
MAddr	configurable	master	Transfer address
MCmd	3	master	Transfer command
MData	configurable	master	Write data
MDataValid	1	master	Write data valid
MRespAccept	1	master	Master accepts response
MByteEnable	4	master	One-hot byte enable
SCmdAccept	1	slave	Slave accepts transfer
SData	configurable	slave	Read data
SDataAccept	1	slave	Slave accepts write data
SResp	2	slave	Transfer response

Table 3.1: Implemented OCP signals

Where needed, OCP-compliant wrappers modules have been developed to adapt the original module interface to the one specified by the protocol.

3.3.2 Processing elements, memories and I/O

The computing element library has been mainly focused around the FPGA-oriented Xilinx proprietary soft cores, that is to say the Microblaze©[55] and PowerPC©[56] cores. Recently, a set of other processing elements have been added to the repository, namely a set of multimedia oriented Application Specific Integrated Processors (ASIP) provided by Silicon Hive, Inc. and a processor specifically designed for the packet processing domain provided by

Lantiq Deutschland GmbH. Nevertheless, their restricted access does not allow to publish here the architectural details. For this reason, we cannot provide here all the architectural details on the processing element library.

As for what regards the memorization and I/O elements, standard Xilinx soft cores are currently being used. The memory modules can be automatically configured as single- or double-port BRAM-based modules. The separation between such cases is operated according to the system-level description provided by the designer. In case direct processor-to-processor message-passing is enabled, double-port memories will be selected, together with the necessary logic to handle DMAs at the network interface level.

Regarding the I/O, standard controller modules are available, such as UART, Ethernet and RocketIO controllers.

3.3.3 Interconnection elements

The Xpipes NoC component library ([14], [6]) is a highly flexible library of component blocks that has been chosen as baseline reference for the development activity. The library is suitable for the creation of arbitrary topologies, thanks to the capability of its modules of being almost completely configured at design time. Xpipes, natively, includes three main components: switches, network interfaces (NIs) and links. Figure 3.2 plots the basic architecture of the Xpipes switch. It is a very simple switch configuration, where output buffer is employed through multi-stage variable-latency FIFOs. A round-robin priority arbiter with selectable inputs is employed to allocate the all-to-all crossbar output ports. The minimum traversal latency per switch is 2 clock ticks per flit.

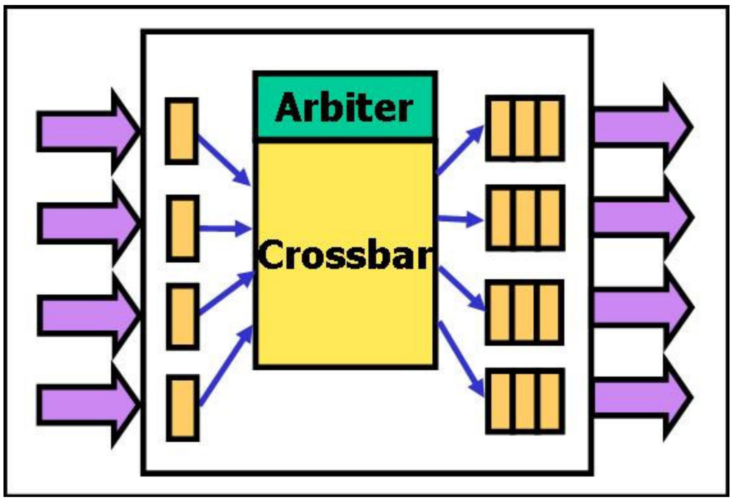


Figure 3.2: Xpipes switch architecture

The backbone of the NoC is composed of switches, whose main function is to route packets from sources to destinations. Arbitrary switch connectivity is possible, allowing for implementation of any topology. Switches provide buffering resources to lower congestion and improve performance. In Xpipes, both output and input buffering can be chosen, i.e. FIFOs may be present at each input and output port. Switches also handle flow control issues, and resolve conflicts among packets when they overlap in requesting access to the same physical links. A NI is needed to connect each IP core to the NoC. NIs convert transaction requests/responses into packets and vice versa. Packets are then split into a sequence of flits before transmission, to decrease the physical wire parallelism requirements. In Xpipes, two separate NIs are defined, an initiator and a target one, respectively associated to OCP system masters and OCP system slaves. A master/slave device will require an NI of each type to be attached to it. The interface among IP cores and NIs is point-to-point as defined by the OCP subset described in Table 3.1, guaranteeing maximum reusability and compliance with the interface standards.

NI Look-Up Tables (LUTs) are used to specify the path that packets will follow in the network to reach their destination (source routing). Two different clock signals can potentially be attached to the NIs: one to drive the NI front-end (OCP interface), the other to drive the NI back-end (Xpipes interface). The Xpipes clock frequency must be an integer multiple of the OCP one. This arrangement allows the NoC to run at a fast clock even though some or all of the attached IP cores are slower, which is crucial to keep transaction latency low. Since each IP core can run at a different frequency of the Xpipes frequency, mixed-clock platforms are possible. Inter-block links are a critical component of NoCs, given the technology trends for global wires. The problem of signal propagation delay is, or will soon become, critical. For this reason, xpipes supports link pipelining, i.e. the interleaving of logical buffers along links. Proper flow control protocols are implemented in link transmitters and receivers (NIs and switches) to make the link latency transparent to the surrounding logic. Therefore, the overall platform can run at a fast clock frequency, without the longest wires being a global speed limiter. Only the links which are too long for single-cycle propagation will need to pay a repeater latency penalty.

Within the development of the framework, the original Xpipes library has been extended explicitly for adaptation and integration in current project, and to provide advanced communication capabilities required for fast prototyping. Here follows a list of the main features that have been added to the library:

- Capability of initializing and handling DMAs (meaning direct memory to memory transfers). The need for this feature has appeared in order to support, at low level, all those models of computations that rely on direct processor-to-processor message passing. In order to implement this added capability, additional logic has been inserted in the processor and memory network interfaces. The way this logic works is basically that the sending processors programs, through memory-mapped registers, a DMA transfer from its memory to a destination memory, by specifying the destination network address and the burst length. The transaction is then translated into an OCP burst transfer, that takes place from the source memory directly to the destination one. Upon receive, the destination network interface is able to store the incoming data on a temporary memory buffer or, if the receiving processor has already reached the receiving primitive call within the application, directly into the destination memory area. Further detail on the software implementation of the message-passing strategy will be provided in Section 3.3.5.

- Runtime reconfiguration of the routing strategy has been implemented in order to enable the interconnection subsystem to reorganize its source routed paths to account for prospective modifications within the interconnection architecture. this feature is particularly useful in case faulty elements are detected within the network. In such cases, reconfiguring the routing strategy can help in bypassing, at traffic flow control level, these nodes. However, the use of such approaches is not object of this research activity.

- Insertion of performance counters inside NoC modules has been enabled through addition of dedicated hardware monitors directly attached to the output buffers of the switch. The value of the counters are then written into dedicated memory-mapped registers through which they are accessible to the processing element.

- Bypassing capabilities inside the switches have been implemented in order to reproduce combinatorial links between two nodes of the network. This capability enables hardware modification of the actual network topology, and is useful for reducing the overhead of the FPGA synthesis and implementation process. An entire section, Sect. 3.8 will be dedicated to describe this approaches and the related hardware/software implementation.

All the mentioned additional features required some modifications of the processor-to-NI interface circuitry. Several dedicated adapters have been developed in this aim, allowing at the same time the seamless integration of the Xpipes library (natively compliant with OCP) with the rest of the environment. Some address decoding logic has been added inside the core in order to detect those load/store operations that are not intended to generate traffic over the network, such as accesses to memory mapped registers, to performance counters, to reconfiguration circuitry. Here is a list of the modifications required to implement the mentioned features:

- a master-and-slave (initiator-and-target) network interface: this module, added to the original Xpipes library is used when message passing and memory-to-memory transfers have to be supported, to make network node capable of performing the operations related to both communication ends.

- hardware counters have been instantiated at the core interface and inside the switches, to detect the activity metrics needed to use the models described in Section 3.6. This counters are needed to evaluate in detail the performances achievable with every candidate architecture.

- routing LUTs programmability: the LUTs that contain the directives related with the routing strategy for every source-destination pair in the system have been made programmable by local connected core or remotely, through the use of a special packet.

3.3.4 Synchronization modules

Every application written to exploit thread-level parallelism on top of a multi-core platform, not only in the embedded field, requires synchronization among the different running threads. In case the memory organization includes some kind of shared memory layer, this can be implemented through well-established lock-unlock mechanisms. This mechanisms can be implemented through full software solutions that rely on specific atomic instructions and ISAs, or through dedicated hardware modules that handle the atomicity control. In order to implement atomic memory access to specific locations, since not all the processors included in the processing element library supported native LL/SC instructions within their

ISAs, we decided to develop hardware semaphore modules for locking/unlocking specific memory locations. These modules implement in hardware the atomic *Test&Set* mechanism.

The basic idea of such mechanism is, before actually accessing the shared memory location, to acquire a hardware lock for that location. The hardware lock is asked for acquisition via a simple load to a specific memory address (the address of the semaphores bank). Upon request, the bank of semaphores has specific logic to check if a request for that location has already arrived, and if not, replies with an ack (encoded in the data field) and atomically lock that location. If that location is already locked, on the opposite, the reply data will contain a particular encoding for a nack. The hardware implementation for such a mechanism requires a bank of registers (the lock registers) to store the requested location addresses and the semaphore bits (one for each location), plus all the logic to compare in a combinatorial fashion the desired address with all the locked addresses. The hardware *Test&Set* has been implemented with a standard OCP interface and can be added as a normal I/O module, by specifying its memory-mapping.

On the software side, two low-level functions have been implemented to lock and unlock a specific location. The functions have been implemented as assembly microcode to optimize the execution time. Their code is structured as follows, considering a network address for the *Test&Set* modules of $0xFF$ in the address MSB:

```
void lock(int * lock_index){

__asm__ (
"addi r1, r1, -12 \n\t "
"  sw r10, r0, r1  \n\t "
"  swi r9, r1, 4  \n\t "
"  swi r11, r1, 8  \n\t "
" or r0, r0, r0  \n\t "
" ori r9, %0, 0xff000000  \n\t "
" LOCK: \n\t "
" lw r10, r0, r9  \n\t "
" or r0, r0, r0  \n\t "
" or r0, r0, r0  \n\t "
" bnei r10,LOCK  \n\t "
" or r0, r0, r0   \n\t "
" or r0, r0, r0   \n\t "
      "  lw r10, r0, r1  \n\t "
      "  lwi r9, r1, 4  \n\t "
      "  lwi r11, r1, 8  \n\t "
      "  addi r1, r1, 12  \n\t "
:
:"r" (lock_index) //input
);
}

void unlock(int * lock_index){
```

```
__asm__(" UNLOCK:      "
" addi r1, r1, -8 \n\t "
"  sw r10, r0, r1  \n\t "
"  swi r9, r1, 4  \n\t "
" or r0, r0, r0  \n\t "
" ori r9, %0, 0xff000000  \n\t "
" sw r0, r0, r9  \n\t "
" or r0, r0, r0  \n\t "
" or r0, r0, r0  \n\t "
"  lw r10, r0, r1  \n\t "
      "  lwi r9, r1, 4  \n\t "
      "  addi r1, r1, 8  \n\t "
:
: "r" (lock_index) //input
);
}
```

3.3.5 Software libraries

As part of the library-based approach, we developed several software routines that constitute the framework Hardware Abstraction Layer (HAL). These routines are required to provide to the application/firmware level the necessary APIs to implement the desired model of parallel computation, and can be included through standard header files. Here follows a list of the main software functions, along with their functionality:

- *Shared memory lock/unlock primitives* - these functions have been already described in Section 3.3.4. They provide lock/unlock primitives for shared-memory multi-core systems and rely on the hardware *Test&Set* synchronization module.

- *Thread spawn and wait primitives* - these functions emulate the creation of threads of execution on remote processors. The thread creation is emulated through pointer passing. Basically, the spawning thread calls a `create()` function, that sets a pointer in a shared memory location and wakes up a remote processor that was polling on that address. This mechanism, however, has the strong implication that all the processors load the same instruction and initialized data regions at startup, in a SPMD fashion. In such a way that thread instructions are already located in each processor local instruction memory. Otherwise, remote thread creation would have implied a thread load from the shared memory or, more likely, a thread load from the off-chip memory. The spawned thread, as already hinted, runs on a processor that was stuck waiting for the function pointer on that specific location, by calling a `wait_task()` function. Upon exit, the `wait_task()` function returns the pointer to the task to be executed.

- *Shared memory barrier synchronization primitives* - these functions implement the barrier synchronization for shared memory systems. They rely on the specific `bar_type` data type, whose struct follows:

  ```
  typedef struct bar_type_ {
  ```

```
volatile int num_p;
volatile int counter;
volatile int lock_b;
volatile int flag;
} bar_type ;
```

The barrier is implemented by atomically (locking the `lock_b` variable) incrementing a shared counter (`counter`) and by busy waiting until it reaches the predefined number of accesses (`num_p`), meaning that all the desired threads have entered the barrier. The barrier has to be first initialized by calling the `barinit()` function, that specifies the locking address and the number of expected threads.

- *Message-passing send and receive primitives* - these functions implement the classic send and receive primitives for message passing models of computations. The library contains an MPI-like implementation of the primitives [49]. The `SHMPI_send()` function implements a non-blocking send and takes as input the address of the memory region to be sent through the network and the related size, plus the identifier of the receiving process and an unique message identifier, which resembles the classic MPI *tag* concept. What the function does is basically programming the DMA logic inside the source processor's network interface logic to perform the OCP burst transfer to the desired destination, and then return in a non blocking fashion. Accordingly, the `SHMPI_recv()` function implements a blocking receive primitive. Its parameters are complementary to those of the `SHMPI_send()` function, meaning that the source processor must be identified explicitly, and the tag field must match the one of the sent message. Inside the `SHMPI_recv()` function, one of two cases might occur. First option, the message could have been already received at the calling time, therefore it must be copied from the local memory buffer (where the DMA had previously written it) to the desired memory address. Second option, the message might be still in flight from the source processor to the destination one, therefore the receiving DMA could be programmed to perform, upon receive, a direct copy to the desired memory location, saving a memcopy from the message buffer to the desired area. In the latter case, the function behaves as a blocking receive. Alternatively, a non blocking `SHMPI_nb_recv()` function might be used. It basically performs the first check of the two previously mentioned. If the message has already been received it performs a memory copy from the buffer to the desired address, otherwise it returns with a negative code. The non-blocking function is useful to the aim of overlapping communication and computation but, however, must be coupled with a polling mechanism to avoid erroneous usage of not yet arrived data.

- *printing functions* - these functions have been designed for debug purposes and print characters and numbers through the UART controller to the serial I/O port. Atomic use of the controller is guaranteed both at the single charachter level and at entire string level. Their names are `shmpi_print()` and `shmpi_putnum()`.

- *functions for accessing the performance counters* - these functions access the memory-mapped performance counters and print them on the UART controller for debugging and elaboration purposes. There are separate functions for accessing the performance counters related to processing/memorization elements, called `print_core_pc()` and for the switching elements, called `print_switch_pc()`. More details on the performance counters will be provided in Section 3.5.

3.4 The SHMPI platform builder

As illustrated in Figure 3.1, the platform instantiation stage is handled by the SHMPI topology compiler, a tool that automatically creates the hardware/software platform description files basing on the specification input file provided by the user, instantiating the desired set of building modules (cores, interconnection building blocks, memories) from the library of configurable soft-cores. The SHMPI topology compiler extends to the composition of the entire multi-core hw/sw platform and to the integration with the Xilinx development tools the ×pipes compiler, a tool developed for the automatic instantiation of application-specific interconnection networks [31]. In further detail, the SHMPI topology compiler able to construct the desired platform, instantiating and interconnecting, through a customized ×pipes NoC layer, an arbitrary number of processors, memories (private or shared), memory controllers, I/O peripherals, buses, bridges, dedicated point-to-point communication channels, etc.

The topology compiler automatically generates the hardware/software description files that are necessary for the FPGA implementation of the whole hw/sw platform. Since the Xilinx proprietary tools are used to handle the FPGA synthesis flow, the generated files must respect the specific syntax in order to be correctly processed. Among the generated output files, the following are included:

- a high-level hardware description (*.mhs* file) of the entire platform to be synthesized. This file includes the specification and configuration of all the hardware cores, along with the interconnection of their I/O ports.
- a description (*.mss,.h,.c* file) of the software part of the platform. These file include the address definition for all the peripherals, in addition to the actual drivers that are going to be included from the application to effectively use the peripherals.
- all the necessary RTL files (*.v*) describing the modules composing the platform, being these either custom modules or library soft-cores.

From an implementation viewpoint, the platform builder has been developed as a standalone sequential C/C++ code, which runs through different consecutive phases, as described in the following list:

- a parsing phase, which scans the entire system-level description input file to build a set of data structures that store the number and identifiers of cores, memories, interconnection elements, links and routing tables.
- a switch configuration phase, which calls a different program to configure the switch according to the designer description and eventually generate their RTL description. This phase also generates the routing tables initialization phase in order to correctly implement source routing within the network.

- a *top module* generation phase, which builds the RTL description of the top level Verilog platform description file. This file contains the highest view of all the modules included in the system to be synthesized for FPGA.

- a phase that generates all the files necessary to the Xilinx proprietary FPGA synthesis and implementation flow. Other than the *.v* files, necessary files include some hardware platform description files (*.mhs* and *.mpd*), a software platform description file (*.mss*), a peripheral list file (*.pao*), an ISE project file (*.xise*).

- a simulation script generation phase. This phase generates a ModelSim script file (*.do*) and the related testbench file (*.v*) in case software waveform simulation of the platform needs to be performed for debugging purposes.

- a phase to generate the memory initialization files (*.bmm*), that direct the Xilinx toolchain in correctly mapping the application binaries into the different BRAM modules instantiated in the platform.

Upon successful execution of the SHMPI builder, all the directories and files are in place for continuing the FPGA implementation flow down to device configuration. These phases rely on Xilinx proprietary toolchain, therefore require its availability for the designer. If no manual tuning is needed at this stage, the toolchain can be traversed with a single script which is available for the designer.

3.5 Performance extraction

The extraction of the performance metrics is handled through the insertion of a dedicated set of event-counters, directly connected to the monitored logic. The insertion of this measurement subsystem does not overload the whole emulation in terms of occupied hardware resources within the FPGA fabric, since the event-counters involve very scarce logic utilization. Three types of performance counters are allowed, according to the architectural element they are connected to. It is possible to insert monitors at the processing core interface, at the switch output channel interface and at the memory port interface, as depicted in a sample platform in Figure 3.3. The specification of which events are intended to be monitored can be easily included in the topology file that is passed as input to the whole framework. The SHMPI topology compiler is able to handle the insertion of the necessary hardware modules and the automatic binding of the event-counters to the proper signal.

The basic usage of the event-counters does not imply the instantiation of dedicated Block-RAM buffers for the storage of event traces. However, the addition of such buffers and of the necessary communication structure (a bus shared amongst the counters) is quite straightforward, inside the topology description file.

The overhead introduced while accessing the performance counters depends on three utilization factors, namely which core is going to access them, when they are accessed, how they are physically connected to the rest of the system. The allowed alternatives are:

- Regarding which core is intended to access the performance extraction subsystem, two main options exist. A dedicated processing core can be added to the emulated system, in order to perform the access to the event-counters without affecting the regular execution on the other emulated processors. Alternatively, in order to save hardware

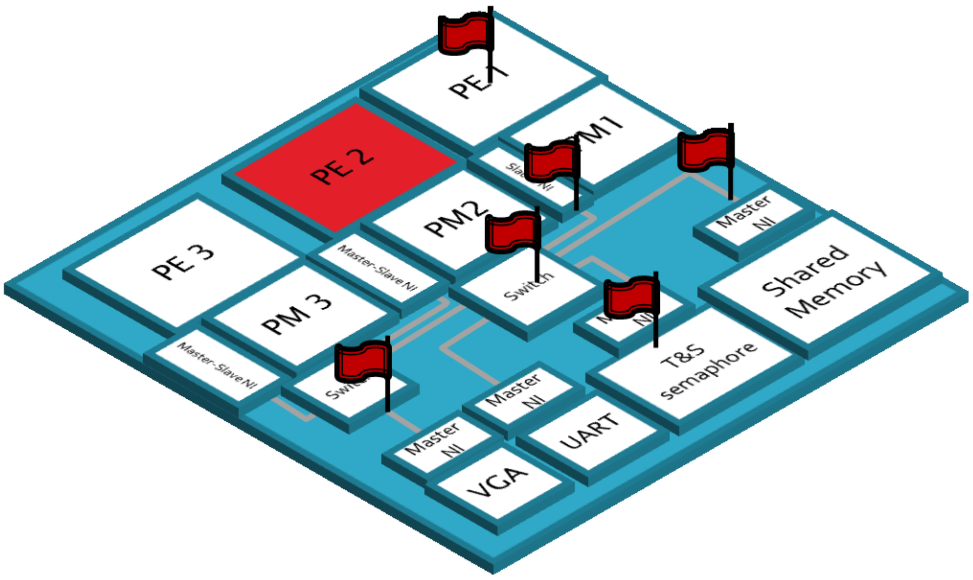


Figure 3.3: Performance counters pinpointing graphical view

resources, the same processing cores of the emulated platform can interleave the execution of their instructions with the accesses to the performance counters.

- Concerning the time of the access to the performance counters, they can be accessed off-line (at the end of the execution) or at runtime, in case the read values should be used to implement runtime resources management mechanisms.
- Finally, the event-counters can be connected to the rest of the system via dedicated point-to-point connections, at the price of additional hardware resources to be utilized, or they can be accessed through the same interconnection layer already present in the emulated system, at the price of an overhead in the actual traffic pattern generated by the application.

3.6 Analytic modeling for prospective ASIC implementation

As already mentioned, within the proposed framework the use of analytic models is coupled to FPGA fast emulation, in order to obtain early power and area figures related to a prospective ASIC implementation, without the need to perform long-lasting cycle-accurate simulations. The metrics extracted with the FPGA-based emulation of the system are passed as input to the analytic models for the estimation of the physical figures of interest. So far, the modeled physical metrics are static and dynamic power, operating frequency and area obstruction. Some of them can be predicted resorting only to static architectural parameters, other need activity traces from the FPGA-based execution. The considered models were already present at the state of the art at the beginning of this research activity, and are described in [36], referring to the power consumption and area occupation of the ×pipes NoC building blocks. The models parameters and relations had been proposed based on prior knowledge on the switch architecture and building blocks, while the tuning of the coefficient parameters had been done by defining two separate training and validation sets of architectures, running ASIC synthesis (0.13 μm process) on them and performing the tuning process sketched in Figure 3.4.

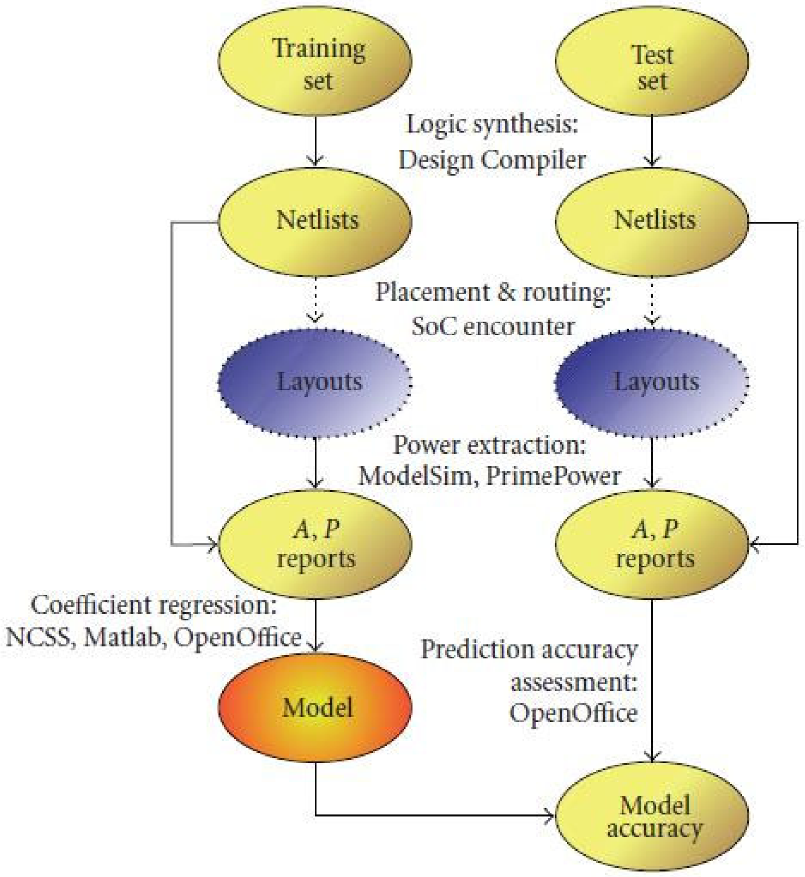


Figure 3.4: Model development methodology

The main model parameters are:

- switch cardinality (number of ports). To account for rectangular switches, we separately consider the amount of input ports (np_i) and output ports (np_o)
- amount of buffering devoted to flow control handling and performance optimization, also called buffer depth (bd) (expressed in terms of single-flit buffering elements)
- number of bits of the incoming and outgoing elementary data blocks, also called flit width (fw).

The accuracy of these models is assessed in [36] to be lower then 10% when complete topologies are considered, with respect to post layout analysis of real ASIC implementations. This accuracy value is quite reasonable for system-level estimation purposes.

3.6.1 Area modeling

The formula for the area occupation model is as follows:

$$A(fw, bd, np) = A1 * np_o * fw * bd + A2 * np_i * fw + A3 * np_o * np_i + A4 * fw * np_o * np_i \quad (3.1)$$

The coefficients $A1$, $A2$, $A3$, $A4$ depend on the architectural configurations and are tuned as mentioned before. The rationale of this formula is that the area of the target switch can be rendered as the sum of four contributions: output buffers, input buffers, arbitration and flow control logic, crossbar. Each contribution strongly depends on a known combination of architectural parameters as follows:

- Output buffers, which are dominated by flip-flop area, can be supposed to depend linearly on flit width fw and buffer depth bd (Xpipes switches are output-buffered), which respectively represent the width and depth of the buffer. There are np_o of such buffers.

- Input buffers are similar to the case above, but since they have a constant depth, they do not depend on bd. Obviously np_i is used in place of np_o

- Since a distributed arbitration technique is used in the target switch, one arbiter is instantiated at each output port. Each arbiter has a complexity proportional to the number of candidate input ports np_i, therefore the overall contribution is the product of the input and output cardinalities. The arbiter logic is clearly independent of datapath parameters such as the flit width and the buffer depth.

- The area overhead due to the crossbar must have a linear dependency on flit width, must be independent of the buffering resources, and must have a linear dependency on the product of input and output cardinalities.

3.6.2 Power modeling

The formula for the power model is as follows:

$$P(bd, fw, np_o, np_i, T) = P_A + \sum_{j=1}^{np_o} [P_B * T_{O_j}] + \sum_{j=1}^{np_o} [P_C * T_{OC_j}] + \sum_{j=1}^{np_i} [P_C * T_{IC_j}] \quad (3.2)$$

where P_A, P_B, P_C and P_D depend on architectural configurations and are tuned with the same approach defined earlier, while the traffic variables are:

- T_{O_j}: percentage of time during which the output port j is successfully transmitting flits. This coefficient models traffic in absence of congestion.

- T_{OC_j} : percentage of time during which the output port j is trying to transmit, but flits are rejected. This coefficient models external congestion due to traffic spikes.

- T_{IC_j} : percentage of time during which the input port j of the switch is trying to transmit flits through one of the output ports, but arbitration is denied by the switch logic. This coefficient models the contention for the same output port inside of the switch.

This document presents only the results obtained by the application of the models to the modules that build the interconnection network, since the utilized processing and memorization cores are specifically targeted for FPGA-based systems instead of an eventual ASIC implementation. Nevertheless, the approach is widely extensible in case more general processing and memorization elements should be taken into consideration.

The insertion of the analytic models in the emulation framework, in addition to the plain estimation of power and area costs of a prospective ASIC implementation, enables the evaluation of the variation of such metrics versus the variation of the selected target frequency of the synthesis process. Obviously, since the target ASIC system and the emulated FPGA-based system operate at different frequencies, emulation consistency issues might arise when interfacing with off-chip devices (e.g. memories, I/O ports). In fact accessing latencies, expressed in terms of clock ticks, might be considerably different in the two cases. To this aim, the clock speed of all the off-chip devices that are intended to interface with the final system must be scaled up/down proportionally to the emulated operating frequency.

3.7 Framework validation

The framework has been employed in two prospective use scenarios, in order to assess its usefulness for rapid architectural emulation and accurate performance extraction, both at the functional and the technology levels. For each of them, the performed experiments and the extracted metrics will be presented and briefly discussed. The adopted hardware FPGA-based platform features a Xilinx Virtex5 XC5VLX330 device, counting over 2M equivalent gates. Figure 3.5 portraits the adopted prototyping board. It also features the most common I/O interfaces (Serial, VGA, Ethernet, FireWire, USB, leds and switches), 4 sockets for FPGA-based daughter boards, a LCD display, a number of quartz oscillators for selectable-frequency clock generation, banks of off-chip DRAM and a connector for a Flash storage device.

The considered application is in both cases a shared memory implementation of the *RadixSort* algorithm, included in the well known SPLASH2 suite [44]. The application kernel is configurable in terms of the number of tasks that operate in parallel, therefore the mapping of different versions of the code on the platform, with different levels of parallelism requires little effort.

3.7.1 A first case study: analysis of the scalability of a parallel kernel

The first case study employs the emulation framework to analyze the scalability of a shared memory application over a regular 2D 4x4 quasi-mesh topology, where every node of the mesh includes a processing core (MicroBlaze) and a local on-chip memory (32 KBytes), both connected to a packet-based switching core, instantiated from the ×pipes interconnection library. The memory architecture includes, in addition to the on-chip local memories connected to each processor, a 64KBytes on-chip shared memory, attached to a border-line switch on a corner of the mesh. All the accesses to shared memory locations are handled with lock/unlock mechanisms, which employ a bank of hardware test and set registers, connected to the opposite corner of the mesh, with respect to the shared memory connection point.

Figure 3.5: Virtex5-based prototyping board

The time required for the entire hw-sw emulation is negligible and, provided that the code is properly instrumented to access the performance counters, several metrics can be evaluated. As already explained in Sect. 3.5, the accessible metrics vary from functional to technological ones. Figure 3.6 shows an example of the kind of information that the proposed framework is able to provide, after a single execution of the application on the FPGA emulated platform.

In Figure 3.6 (top-left), the execution times and the total latencies for the different mapping patterns over the quasi-mesh topologies are presented, changing the number of parallel processes from 2 to 16. Figure 3.6 (top-right) shows in further detail the scalability of the latencies, averaged over the number of accesses and over the different processors, separating the accesses to the shared memory from those directed to the Test&Set locations. It is worth noting that it has been decided to consider the latency of accessing the Test&Set peripheral as the amount of time necessary to acquire the lock on a precise location, thus including within a single latency count different trials to obtain a lock on a specific location. The remaining figures relate to the technology-dependent metrics that it is possible to extract. The power consumption is estimated at every switch using information on the flit congestion at each output channel: Figure 3.6 (bottom-right) shows the contribution of each switch to the dynamic power consumed in the system, for the case of 16 active processes. Figure 3.6 (bottom-left) shows the total power consumed by the topology, scaling up the number of active processes. In the same picture, the energy consumption is depicted: the execution time reduction, for the higher number of active processes, mitigates the difference in average power consumption.

The best configuration depends on the constraints on the application execution time. If all the 4 architectures satisfied these constraints, then the best choice would probably be the 4-cores mesh, since it halves the execution time (clock ticks) while keeping the energy consumption almost constant. The tool also helps in modeling analytically the operating

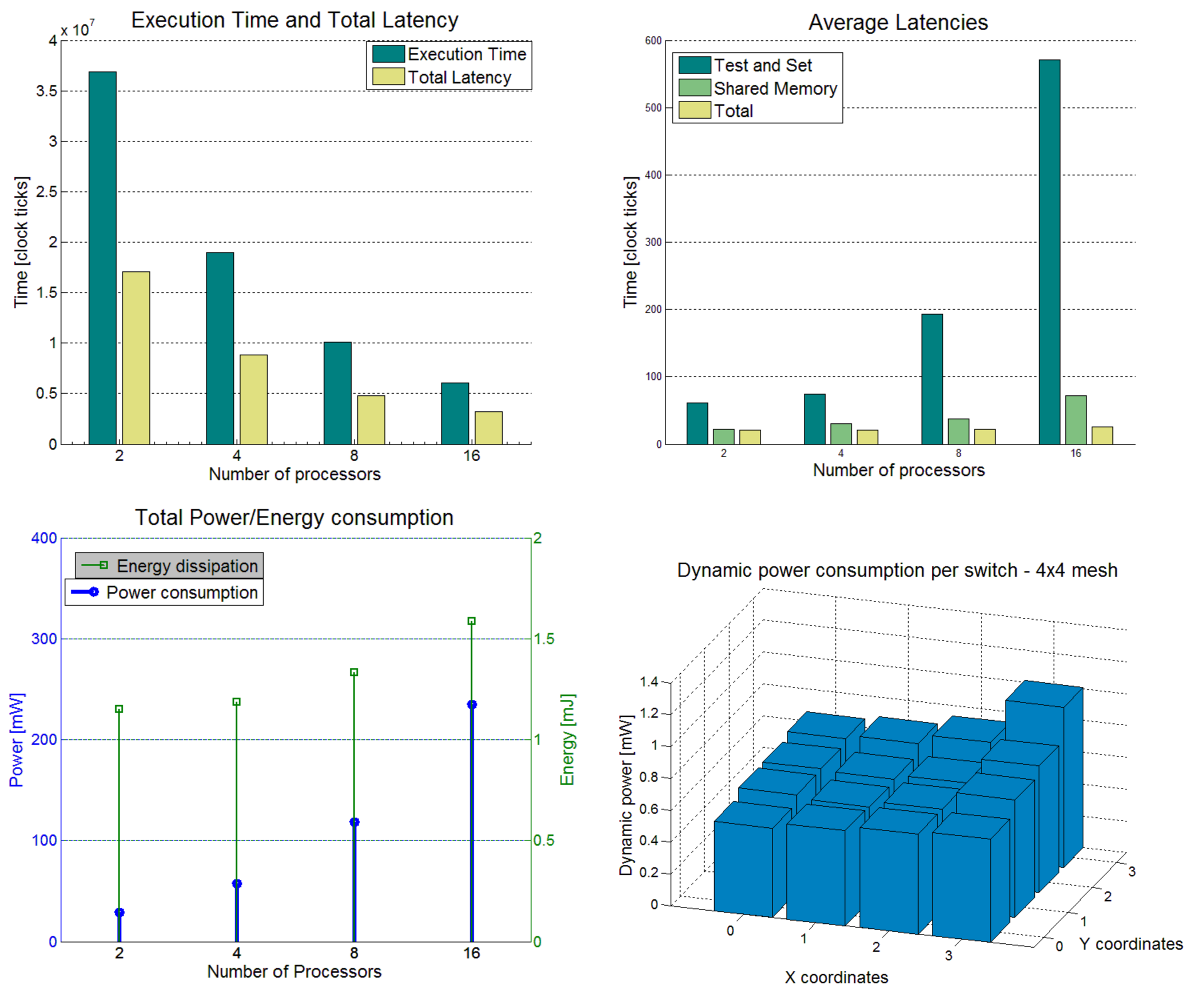


Figure 3.6: Performance metrics for the scalability analysis. The upper figures show the execution times compared with the total latencies (left), and the average latencies for different types of accesses (right). The lower figures show the power/energy consumption, for the total topology (left) and for the single switches(right).

frequency (to convert clock ticks in execution time) and the area occupation of the interconnection network.

3.7.2 A further case study: architectural design space exploration

In this subsection we present a second case study. We show how the framework can be exploited as an evaluation platform to explore performance and hardware costs of different architectural configurations. The user would be able to compare different solutions, to find out the optimal ones and to evaluate bottlenecks and detailed performance metrics of different candidate architectures, for example within a Design Space Exploration environment.

As an example, we show the results of the comparison between three different system configurations featuring different NoC topologies as interconnect infrastructure. The evaluated system configurations are depicted in Figure 3.7. Each configuration includes 8 proces-

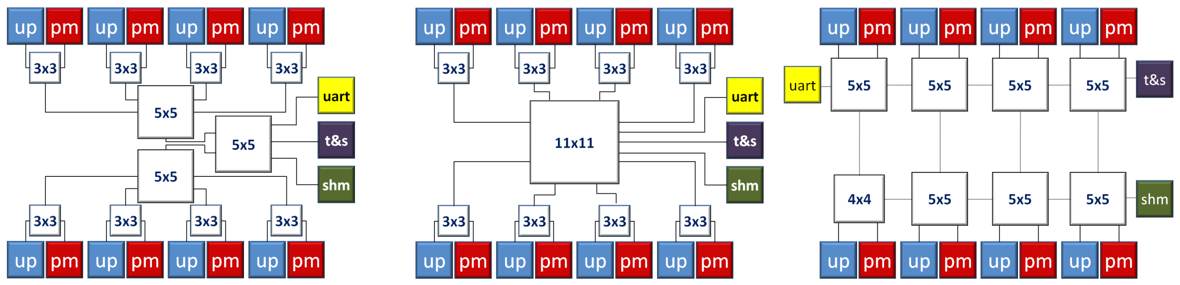


Figure 3.7: Explored topologies layout: the connected cores and the instantiated switches are drawn.

sors, 8 local memories and three shared devices (a memory, a hardware semaphore and an I/O controller). The first interconnection topology (called "star" in the following) features 8 clusters including a 3x3 switch connected to a processor and to its private memory, linked to a big 11x11 central switch that is directly communicating with the shared devices. Since using only one single switch would be unfeasible given the size of the required switch, "star" is the topology with the smaller number of hops that implements the communication patterns required by the application. The second topology (called "tree" in the following) is similar, but the 11x11 switch, whose crossbar complexity appears to be likely critical for the working frequency of the whole network, was replaced with three 5x5 switches, thus trading off an higher bandwidth with an increased number of hops bringing to higher latencies. The two mentioned topologies have been compared with a regular topology, a 4x2 quasi-mesh. The position of the cores inside the mesh layout has been chosen in order to not exceed the 5x5 size for the switches inside the interconnect fabric.

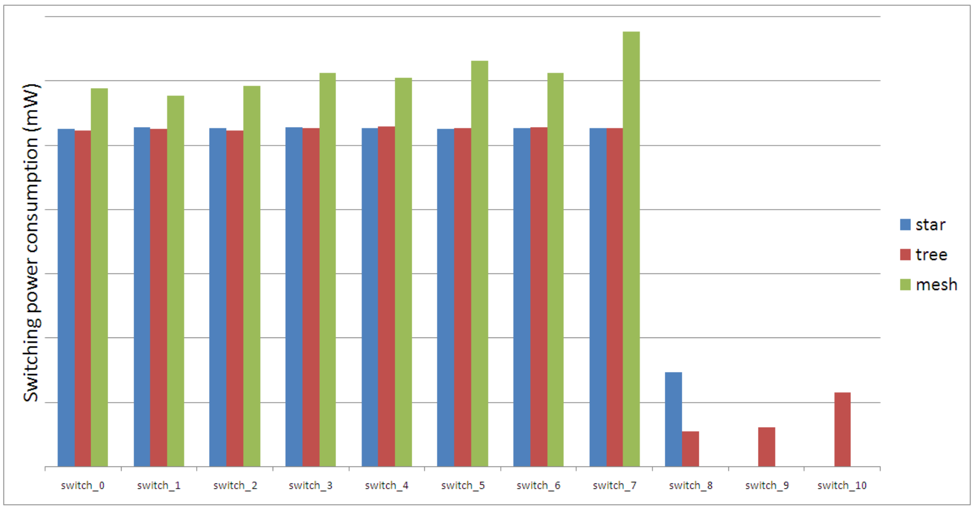


Figure 3.8: Dynamic per-switch power consumption for the three topologies under test.

Some system level comparisons, obtained, after the execution of the *RadixSort* parallel kernel on the three candidate architectures, accessing the performance/event counters inside the system, are plotted in Figure 3.9. The framework was able to detect total execution time expressed in terms of clock ticks, general latencies, average latencies towards each different device, number of occurrences of traffic-related events and many other metrics, according to what specified in the system-level description file. To keep coherence with a prospective ASIC implementation of the considered architecture, the previously mentioned power, frequency and area models were used to back-annotate the evaluation results. In this way we were able to obtain a detailed pre-estimation of the behavior of the considered architecture with respect to die area occupation, power and energy consumption, and execution time, as plotted. This step enhances the prototyping with the technology-awareness needed to perform meaningful design space exploration.

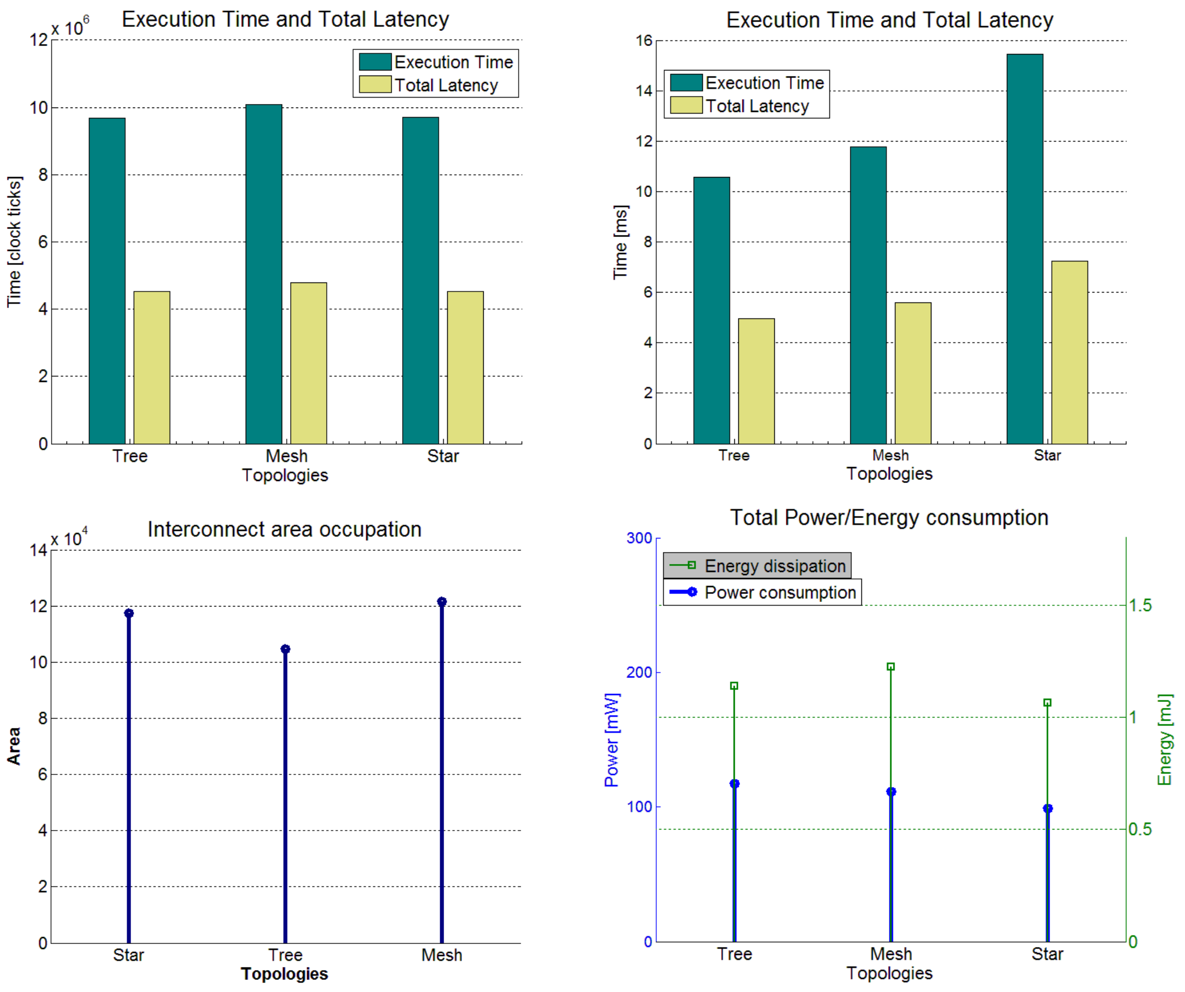


Figure 3.9: The upper figures plot the execution times and the total latencies, both in terms of clock ticks (left) and real seconds (right). The times are calculated considering the maximum operating frequencies of the different topologies, estimated with the analytic models. Area, power and energy figures are shown in the lower figures.

As an example, it can be noticed, observing Figure 3.9, how the back-annotation step allows to detect the benefits in execution time that can be obtained trading off latency with frequency in passing from “star” to “tree”, or how “star” is able to burn less energy than “tree” even if slower in executing the kernel. In a real-life design, such kind of trade-offs would have been evaluated according to the constraints input to the framework. In Figure 3.8 for example we plotted the “dynamic” power consumption related to each switch in each topology (i.e the additional power dissipated by switches due to the traffic flowing through them or congested in their buffers), to find out if bottlenecks or hot spots can be identified. The same kind of graph could be plotted for the “static” power (i.e. the power dissipated independently of the traffic conditions). Thanks to the simultaneous use of FPGA performance counters and models, the effort required to evaluate the three architectures was limited to their implementation on the target device, while without the framework, the same results would have needed much longer executions of the kernel on a software-based simulator (simulation times depend on the simulation accuracy but easily exceed one hour with such kind of kernels, for cycle-true simulations) and preliminary synthesis and place and route experiments on ASIC (usually several hours). A discussion related to the effort needed by the synthesis is reported in section 3.7.3.

3.7.3 Implementation effort evaluation

In the first case study that we presented, the user exploits the framework for prototyping different software configurations and mapping strategies on a pre-determined hardware architecture. In this scenario, to obtain the experimental results related to a given configuration, the framework just needs to iteratively update the bitstream with different application binaries and rerun the execution on the configurable hardware. These steps are commonly quite fast so the number of iterations that are performed during the analysis will hardly become critical. A different situation arises when the goal is to explore different hardware architectural configurations, as we did in the second case study, since every step of the analysis requires the iteration of the whole implementation flow.

The implementation process needed to program modern FPGA devices requires often a considerable CPU time, that obviously grows when the architecture under test becomes more complex. This time has to be duely taken into account if the user plans to use the framework as evaluation step within a design space exploration process, since it can become a critical factor limiting the number of estimations that can be performed during a reasonable time. Very often DSE algorithms use cycle-accurate evaluations only for a limited number of key iterations, using faster, less accurate models during the rest of the optimization cycles [42]. In Figure 3.10 we plot an overview of the processing effort needed to implement different quasi-mesh topologies featuring a different number of processors.

The implementation flows have been performed by ISE 10.1.3 running on a Dual Core AMD Opteron Processor (2210 MHz) with 6 GB RAM. To shrink the synthesis effort we managed to build a library of reusable pre-synthesized components, featuring different parameter configurations. In this way the synthesis time, provided that all the cores instantiated in the considered configuration are present in the library, is reduced to the time needed by the synthesis tool to link the cores and to elaborate the connections in the top-module, which is quite short if compared with the other implementation steps. Obviously, all the steps require a time that increases with the complexity of the architecture. For topologies not bigger than 8 processors, the implementation effort is lower than one hour. Thus, taking into account that the execution of the parallel kernel to be tested on the implemented system is several orders of magnitude faster than the same execution on a cycle accurate software-based simulator [24], the exploitation of the framework as evaluation platform reduces significantly the iteration time. This is especially true when the software application is complex but the number of cores to be integrated hardly increases very much, like, for example, in embedded systems domain. Obvious evolution of the research, from this point of view, will be the tailoring of the implementation flow to manage pre-routed macros, in order to shrink the mapping and place-and-route time as we did for synthesis. As can be noticed from the Figure 3.10, this

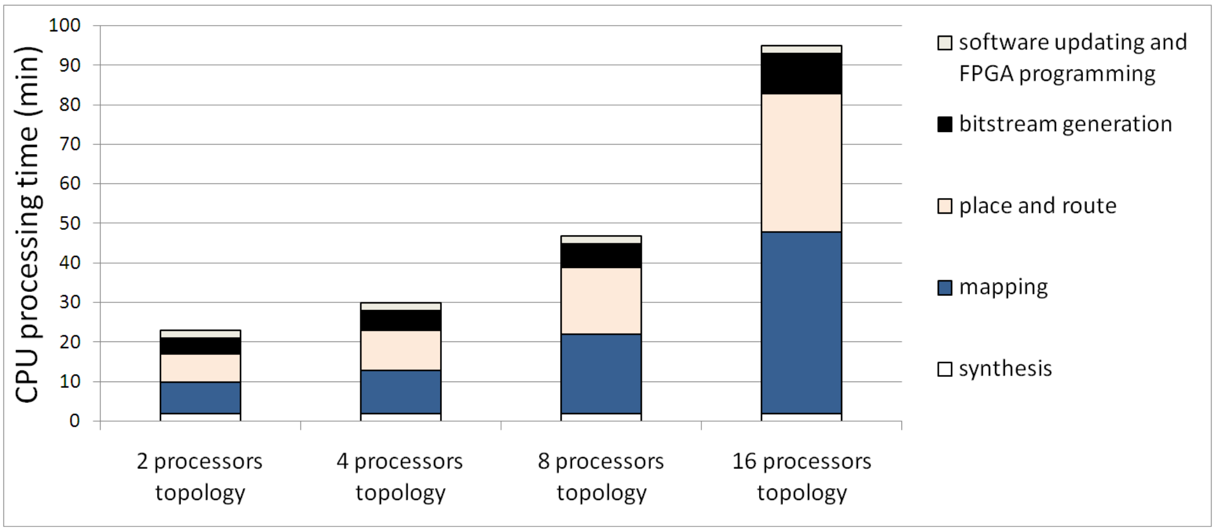


Figure 3.10: Computational effort needed to complete the implementation flow for different architectural configurations

can bring to a big reduction of the implementation time, since it would tackle the longest implementation steps.

3.8 Support for runtime reconfiguration

We already discussed in Section 3.7.3 some results on the effort introduced, in the overall system emulation time, by the FPGA synthesis, implementation and configuration processes. Figure 3.10 plotted the overall decomposed in the main phases of the FPGA implementation flow against an increasing area of the platform under test. Especially when the required FPGA resources approach the physical limit of on-chip available resources, this overhead increases significantly, since the Place&Route and mapping algorithms usually end up performing more iterations before converging to a solution. In order to reduce the impact of this overhead, we now propose an approach based on runtime FPGA reconfiguration. The considered case features a number of candidate architectures to be emulated, normally requiring one complete FPGA synthesis and implementation process for each architecture to be emulated.

The different steps of the design flow of NoC-based architectures often present cases in which one or more features of the system under exploration are fixed. A classical example is the exploration of the interconnection topology where the design space is defined only by the network architectural features, while for example the number of processors or the memory hierarchy could be reasonably considered stable. The interconnection topology selection for a specific application could be decomposed in two distinct problems. First, a set of candidate topologies needs to be defined, either by a human designer leveraging his experience and knowledge of the application domain, or in an iterative way by a dedicated exploration algorithm. Second, a way to evaluate the performances of the candidate topology and to produce effective metrics that can drive the choice is needed.

In this section, we investigate the use of runtime platform reconfiguration for NoC topology selection purposes, by employing a software-based reconfiguration mechanism of an emulation platform equipped with the hardware resources necessary for the emulation of different interconnection topologies. Our aim is to mitigate the afore-mentioned overhead introduced by the FPGA implementation flow. The reconfiguration method is kept as simple and as flexible as possible.

3.8.1 Problem formulation

Given a set of interconnection topologies to be emulated, we investigate the possibility of identifying and reconfiguring what we call a *worst case* topology (WC). The basic idea behind this approach is to implement on the FPGA a topology that is over-provided with the hardware resources necessary to emulate all the topologies included in the predefined set of candidates; then, at runtime, each specific topology will be mapped on top of the implemented hardware of the WC one, exploiting dedicated software-based configuration mechanisms. If such an approach is feasible, then several emulation steps could be performed after a single FPGA synthesis and implementation run, resulting in a speed-up of the whole topology selection process.

In figure 3.11, an example of definition of a worst case topology is provided. We first consider three architectures under test, namely $UT-i$ with $i = 1,2,3$. The only difference between the three is the interconnection topology, while the number and kind of computing, memorization and peripheral devices is the same, as can be seen by the picture. We start by associating a switch to every memory or processing core of the architecture and then iteratively add other switches as we parse the three interconnection topologies, to obtain topology $WC-2$. By considering that many of the switches have become $1x1$ switches, we get rid of them to define $WC-C$, the final *worst case* topology.

Specific hw-sw mechanisms supporting runtime reconfiguration need to be implemented. To mention one of the most challenging problems, considering a processing core, connected to a specific switch in the WC topology, it might be connected to a different one in the topology under emulation. To avoid a new synthesis run, there must be the possibility of mapping the topology under test on top of the WC one, configuring the connections to that processing core to avoid accounting for latencies introduced by the switching elements that are not included in the topology to be emulated. Thus, static direct zero-latency connection of two specific ports of a generic switch must be made configurable at runtime, resulting in the emulation of a combinational path bypassing a certain switch.

3.8.2 Algorithm description

In Sect. 3.8.1 the guidelines of the proposed approach have been defined, and the idea of performing different topology emulations through software-based runtime reconfiguration of the same WC topology hardware has been presented. In this section we present in detail the definition of the aforementioned worst case topology and how the topologies to be emulated are mapped on it.

The WC topology is defined iteratively. As a first step a switch is assigned to every core (processing element, memory or peripheral) to be included in the architecture. Thus, being P the number of cores to interconnect, P switches (referred to as “core switches” hereafter) are instantiated inside the topology. The collection of the switches to be included in the topology is then updated according to the analysis of every topology under evaluation. For each candidate configuration, the number of switches S_{NC}, that equals the number of

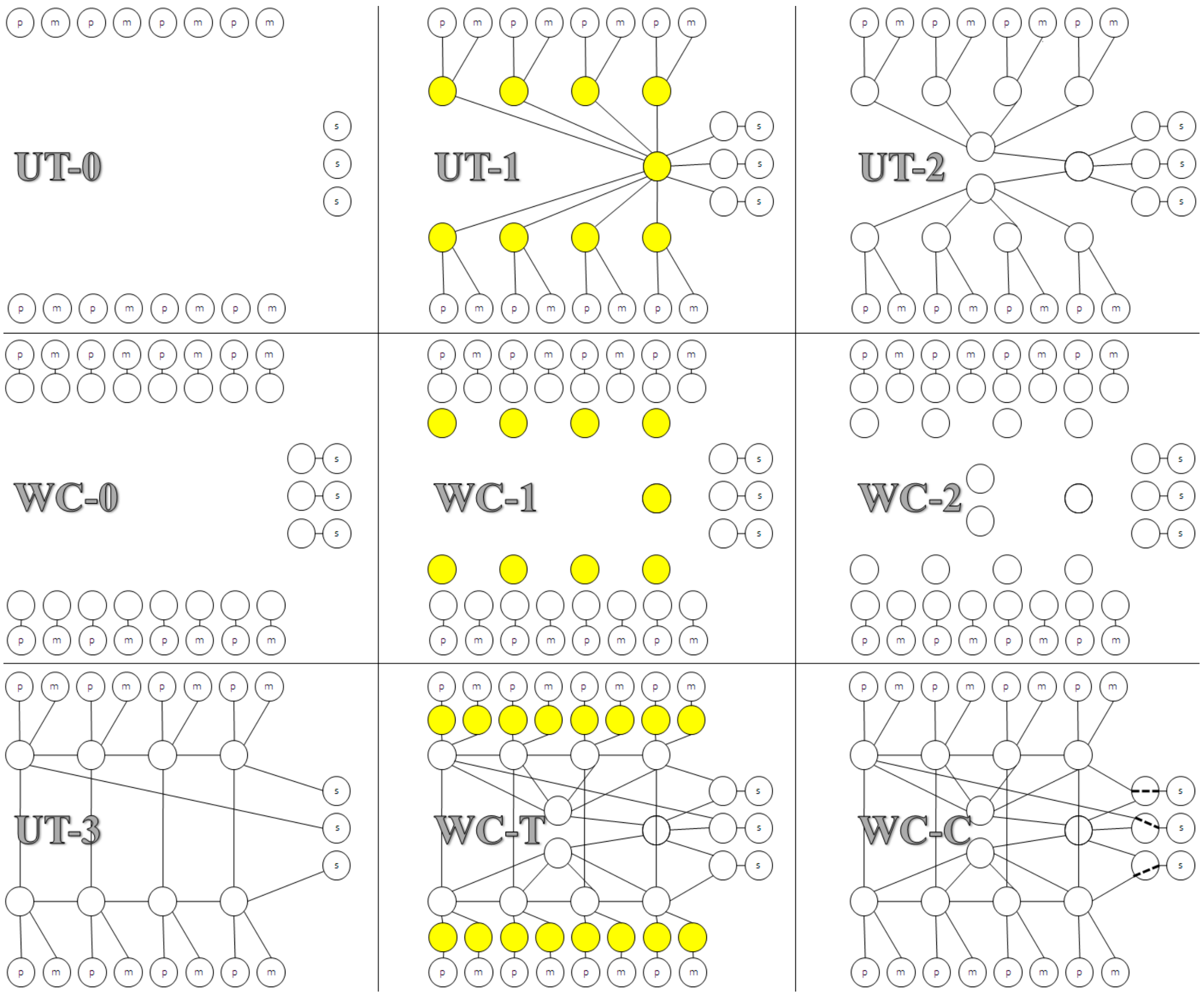


Figure 3.11: Example of a worst-case topology definition

switches that are not connected to any core or that are connected to more than one core ("non-core switches" hereafter), is calculated. After the analysis of the whole set of topologies to emulate, the collection includes a number of switches S_{WC}, i.e. the sum of the number of cores to interconnect and the highest number of switches S_{NC} among all the N topologies to emulate:

$$S_{WC} = P + max\{S_{NC}(i)\} \qquad for \quad i = 1, ..., N; \tag{3.3}$$

The S_{WC} switches are initially connected in an all-to-all fashion, therefore the size, expressed in terms of number of ports, equals to S_{WC} for those switches that are not directly connected to a core and to $S_{WC}+1$ for those that are directly connected to a core (e.g. upper-left sketch in Figure 3.11).

The topologies, whether they are one of those to be emulated or the WC one, can be characterized by a connectivity matrix CM and two auxiliary vectors, SW_C and P_C, defined as follows:

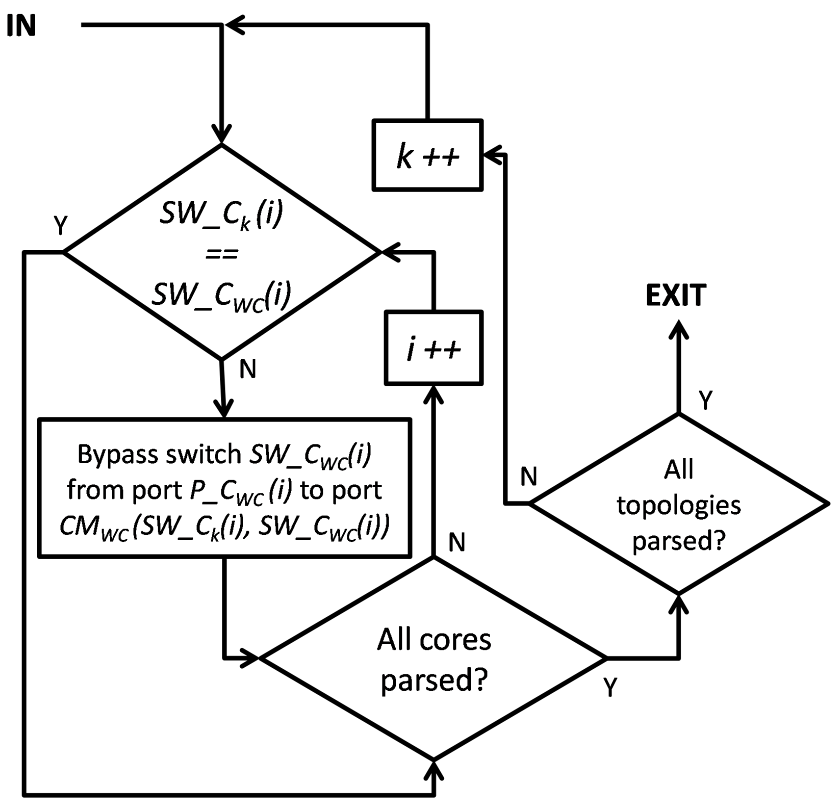


Figure 3.12: Flow chart of the algorithm for mapping the topologies to be emulated on top of the worst case topology

$$CM_{ij} = \begin{cases} OP_{ij} & \text{if link between switch } i \\ & \text{and switch } j \text{ exists;} \\ -1 & \text{if link between switch } i \\ & \text{and switch } j \text{ does not exist.} \end{cases} \tag{3.4}$$

where OP_{ij} is the identifier, within switch i, of the output port of switch i connected to switch j. We assume that the switch hardware is built in such a way that OP_{ij} identifies also the input port of switch i connected to switch j in the backwards link (i.e.: the port is full-duplex and has only one identifier). The size of the CM matrix obviously equals the number of switches present in the related interconnection topology.

The SW_C vector has size P, and its i-th element indicates the number of the switch connected to the i-th core of the topology. The P_C vector has size P, and its i-th element indicates the number of the output port of the switch $SW\_C(i)$ that is connected to the i-th core of the topology.

The matrices defined above must be identified for each topology under emulation, included the WC, and can be used as inputs for the mapping algorithm. This process, whose flow-chart is plotted in Fig. 3.12, is in charge of generating the reconfiguration patterns, bypassing the switches where needed to ensure the correct emulation of the candidate topologies.

The algorithm scans all the cores included in the system, for each of the candidate topologies to be emulated. For the i-th core in the system, it first checks if the switch connected to the core is the same in the worst case topology and in the emulated topology, by looking-up into SW_C_k and SW_C_{WC} vectors. If so, then the mapping of the considered core is straight-forward and no bypass must be established. On the contrary, if the core is connected to different switches in the two topologies, then a bypass is needed from the switch connected to the core in the WC topology to the switch connected to the core in the emulated topology, namely in switch $SW\_C_{WC}(i)$ from input port $P\_C_{WC}(i)$ to output port $CM_{WC}(SW\_C_k(i), SW\_C_{WC}(i))$.

Moreover, for every topology, the algorithm annotates, performing a comparison between CM_{WC} and CM_k, which ports and links inside the WC topology are actually used in at least one candidate configuration. The unneeded port and links can thus be removed in the WC template and the corresponding switch resized. This is a crucial step in order to save FPGA resources, as we will discuss in the following. Remaining switches that just directly connect one core with another switch (i.e. $1x1$ switches with two full-duplex ports) can be removed and replaced with direct connections (e.g. the switch connected to the node $N6$ in lower left sketch in Figure 3.11).

We can now examine again Figure 3.11 to explain in detail the example of worst case topology identification.

- **Example 1:** Definition of the overprovided topology able to emulate three different topologies under test (UT1, UT2, UT3). The three topologies under emulation interconnect 19 cores (8 processors, 8 memories and 3 shared devices).
- **Step 0:** The first step of the algorithm is the analysis of the system population (represented in UT-0). The first version of the overprovided topology is obtained assigning a switch to every core in the system (WC-0).
- **Step 1:** UT-1 is parsed, 9 switches (highlighted) that are connected to more than one core are identified. WC-0 includes only “core switches”, thus the overprovided topology is updated (WC-1) with the insertion of the highlighted 8 switches.
- **Step 2:** UT-2 is parsed, 9 switches connected to more than one core and 2 switches that are not connected to cores are identified. The number of “non-core” switches in the WC-2 is thus increased to 11.
- **Step 3:** UT-3 is parsed, 8 “non-core” switches are identified. WC-2 already includes 11 “non-core switches”, thus the WC-2 topology does not need to be updated.
- **Step 4:** The switches inside the topologies are connected in an all-to-all fashion than the unused connections, never referenced by any topology under test, are removed, as represented in WC-T.
- **Step 5:** The unneeded $1x1$ “core switches” with two input ports and two output ports, that would be permanently set in bypass-mode independently on the candidate topology under consideration, are removed and replaced with direct links. WC-C is the eventual overprovided topology to be synthesized.

The selection of the links to bypass is another crucial step of the algorithm. Obviously, this bypass can be realized through several paths. The algorithm checks if the 1-hop direct path between the two switches is already in use in the emulated topology, by checking the link between switches $SW\_C_k(i)$ and $SW\_C_{WC}(i)$ in CM_k matrix. If not, the bypass is implemented through that path, and the algorithm annotates the related link, that will then be used by the framework configuration engine to generate all the hardware and software parameters and routines necessary to actually implement the bypass mechanism, as explained in Section 3.8.3.

If the 1-hop direct path is already in use in the emulated topology, it cannot be bypassed since the combinational path would obviously invalidate the emulation of the topology, altering the timing of the communications. Therefore, the algorithm searches for a 2-hop path connecting the two switches of interest; all the possible paths are checked. If it finds a 2-hop path composed of links not used in the emulated topology, then the bypass directives are activated for connecting the switches with a combinational path. In case it is not possible to find a 2-hop path, then the process fails, returning an error message.

We decided to stop the search criterion at 2-hop paths in order to avoid creating longer combinational closed loops. As a matter of facts, the hardware support for the bypass mechanism that we implemented succeeds in recognizing up to 2-hop combinational paths and in disabling them, whereas for longer paths the combinational loop would preclude the feasibility of the hardware synthesis and implementation process. This limit is mainly due to the creation of combinational closed loops, that are obviously unacceptable in hardware logic synthesis.

3.8.3 Toolchain and hardware/software implementation

We have developed a whole design flow implementing the proposed technique, essentially enriching the framework we described in this chapter so far. It still receives in input a system-level specification of a target multi-core system but, instead of a single interconnection topology, a set of candidate interconnect topologies is provided. Again, the framework instantiates an FPGA platform for emulation, on which an accurate profiling of a target software application can be performed and cycle-accurate information on the switching activity can be collected. The designer is still allowed to insert specific performance counters at the processing core interface, at the switch output channel interface and at the memory port.

A general overview of the modified design flow is given in figure 3.13.

We modified the tool for platform instantiation described in Section 3.4. When invoked on one single file, runtime reconfigurability for multi-topology prototyping is not enabled. In this case, the tool only generates the input for the Xilinx proprietary tools that are used to handle the FPGA synthesis flow. The hw/sw generated files are still compliant with the specific syntax and formats in order to be correctly processed. To allow fast prototyping of multiple candidate interconnect configurations inside the system, we allowed the tool to accept different topology specification files. Among the topologies, one is specified to be the WC topology, generated offline according to the definition mechanism explained in section 3.8.2, the others are considered as topologies under emulation. After the definition of the WC topology, the tool applies the mapping algorithm, as described in section 3.8.2, to identify how to map the topologies under emulation on the WC template and to produce the information needed to program it accordingly. The identified reconfiguration patterns are produced in output as:

- parameters included by the HDL modules, defining which ports inside the switches must be equipped with the circuitry supporting the bypass

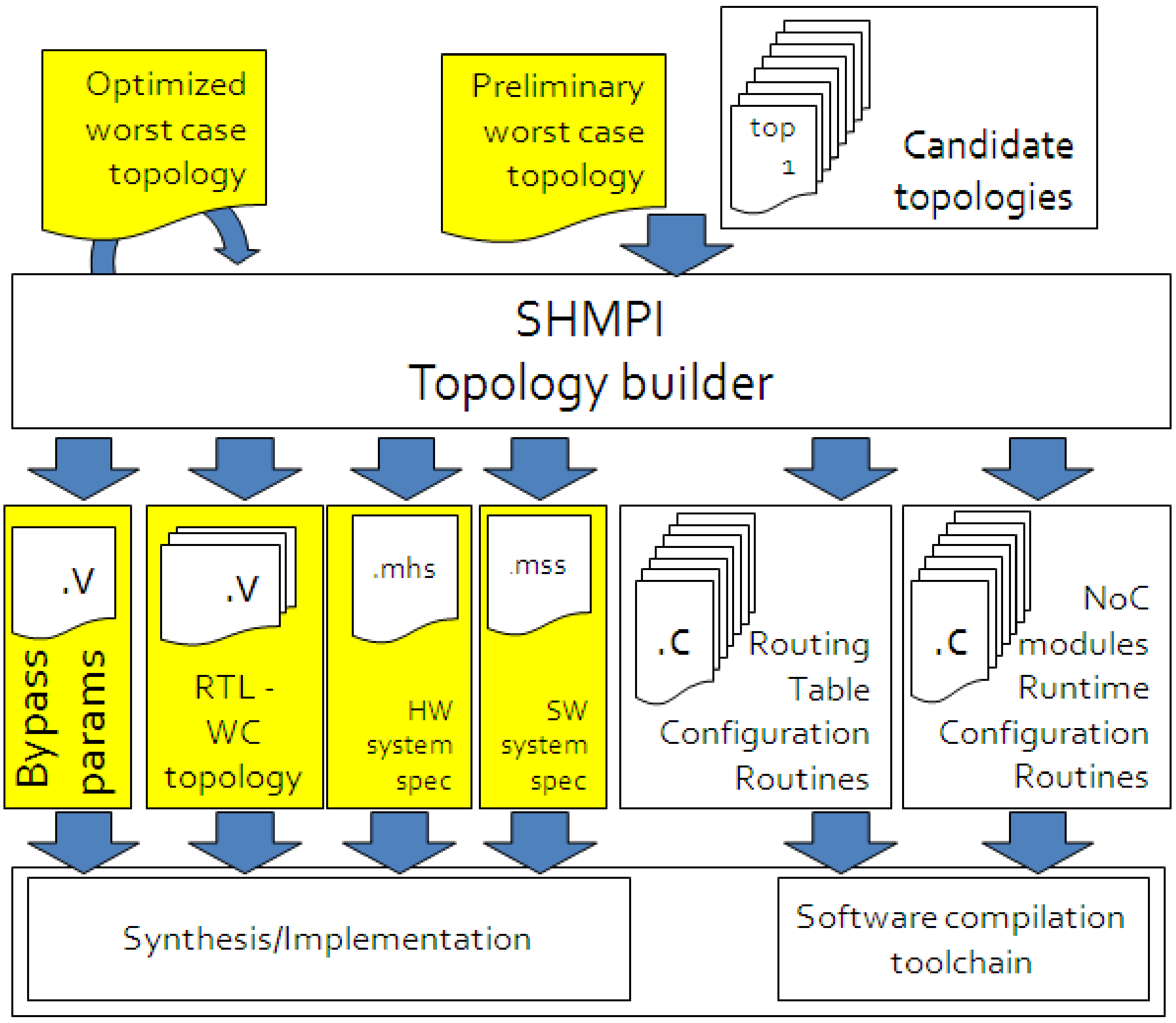

Optimized worst case topology
Preliminary worst case topology
top 1
Candidate topologies
SHMPI Topology builder
.V
Bypass params
.V
RTL - WC topology
.mhs
HW system spec
.mss
SW system spec
.C
Routing Table Configuration Routines
.C
NoC modules Runtime Configuration Routines
Synthesis/Implementation
Software compilation toolchain

Figure 3.13: General overview of the prototyping flow

- C functions to be included for compilation that, when executed by the processors in the system, trigger the runtime reconfiguration.

The hardware and software mechanisms enabling this kind of process will be described in further detail in the following sections.

3.8.4 Hardware support for runtime reconfiguration

The HDL modules describing the switch hardware architecture, which are included in the framework repository as described in Section 3.3.3 have been enriched with reconfiguration capabilities:

- The reconfigurable switches can be set in bypass mode: a static direct connection between an input port and an output port builds a zero-latency combinational path among them, that bypasses all the switching logic for that specific I/O pair.
- The routing tables inside the Network Interfaces have been made programmable, allowing to define different routing paths during emulation of different topologies.

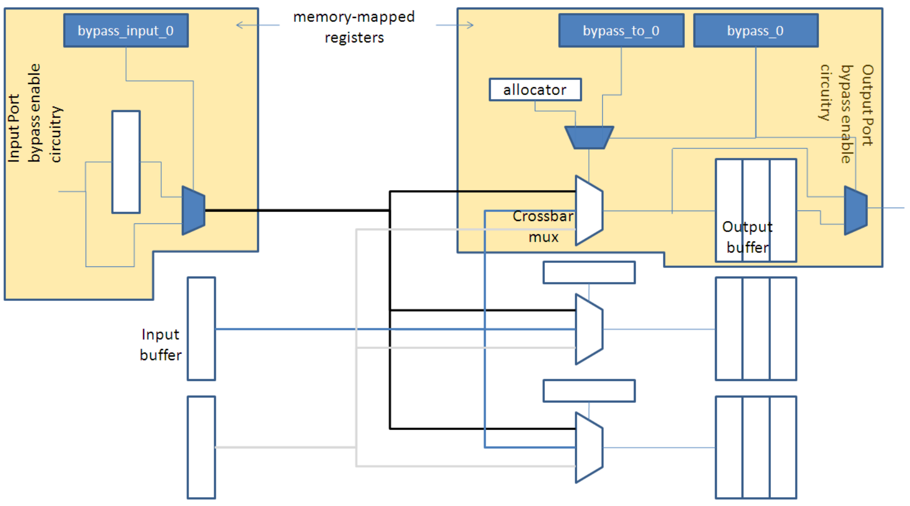


Figure 3.14: Sketch of the hardware resources dedicated to the implementation of the switch reconfiguration mechanism.

Every switch in the WC topology is programmable by a processor, connected through a direct point-to-point link. The processors in the system can thus enable the switch bypassing mechanism via software; this inhibits the use of the involved input and output ports as long as that configuration is preserved. In order to provide this capability, we added, for each switch port, several memory-mapped registers, namely two per output port and one per input port. The memory mapped register at the input (e.g. $bypass\_input\_0$ in Fig. 3.14) has to be written when the user wants to bypass the corresponding input buffer. A combinational path is thus established between the corresponding input port of the switch and the output circuitry. To complete the bypass path, at the considered output port, the processor connected to the considered switch has to write:

- one "input selection" register (e.g $bypass\_to\_0$ in Fig. 3.14), that specifies the ID of the input port that has to be directly bypassed with the considered output.
- one "bypass enable" register (e.g $bypass\_0$ in Fig. 3.14), that enables the bypass of the output buffer and forces the selection input of the corresponding crossbar branch to be statically driven by the content of the "input selection" register.

The routing tables, originally implemented as LUT ROMs, have been made accessible for writing as memory mapped devices. A RAM memory has been developed, whose content is updated at the beginning of a new emulation with the routing information related to the topology under test, according to the software programming routines produced by the SHMPI topology builder (see section 3.8.5). At runtime, the RAMs are accessed in read mode by the NIs, every time a network transaction required by the application is initialized, to obtain the routing information to be placed in the header flit.

Finally, besides topology selection, a further exploration capability has been enabled. The Xpipes links width has been made programmable via software (the size can be chosen among a predefined set), allowing to rapidly solve the bandwidth/area-power trade-off for the target application. A memory mapped register is added inside the NI so that, by writing it, a processor is capable of configuring the link width. The WC topology is parameterized to instantiate the widest links. According to the programmable register content, the NI exploits for communication only part of the physically available wires, to emulate narrower flits, driving the remaining bits in the links with meaningless zeros. It is thus possible to emulate perfectly accurate latency figures corresponding to different serialization factors.

3.8.5 Software support for runtime reconfiguration

As depicted in Fig. 3.13, the runtime reconfiguration mechanism is handled via dedicated circuitry inside the interconnection network building blocks, but also with specific software modules. The SHMPI topology builder, indeed, generates the software routines for:

- the configuration of the routing tables of all the cores included in the system: each topology to be emulated on top of the WC topology needs a set of routing tables. In order to generate these tables, the SHMPI topology builder, when parsing the high-level topology declaration files, annotates the routing tables referred to the emulated topology.
- the configuration of the memory-mapped registers that implement, as described in Sect. 3.8.4, the bypass mechanisms in hardware. These routines are generated right after the execution of the algorithm described in Sect. 3.8.2, which identifies the bypass configurations that need to be implemented.

Since the processing cores of the system are the modules in charge of configuring the bypass registers and the routing tables, the routines generated at this step are then linked by the executables running on each processor, and called right before the start of the actual application code. The execution of the application on the configured platform will start only after all the switches are properly bypassed and all the cores have the routing tables updated.

3.8.6 FPGA implementation overhead reduction techniques

As a consequence of the provision of runtime reconfiguration capabilities, it is easy to expect a degradation of the quality of results with respect to the hardware implementation of a single sample topology on the FPGA device. In particular, aspects related to the implementation quality, that can be affected and potentially preclude the usability of the proposed approach are:

- the area occupation of the WC topology, which determines whether or not the prototyping platform fits on one given target programmable device,
- its working frequency, that, if impacted by switches with a high number of ports or by long combinational paths traversing bypassed switches, can potentially increase the emulation time and reduce the benefits of on-hardware emulation.

The mentioned aspects have been duely taken into account while implementing the toolchain. As already mentioned, in order to minimize the area occupation of the WC netlist, during the parsing of the topologies under emulation, the platform Builder annotates which links are used by at least one among them. At the end of the analysis, the unused switches are removed, the remaining switches are accordingly resized, and thus the amount of programmable logic blocks needed by the netlist is reduced. Moreover, the communication paths flowing through every output port of all the switches are analyzed, and the switch hardware is customized minimizing the size of the multiplexers. Resulting area/frequency figures, corresponding to the use cases presented in Section 3.9, are reported as experimental results assessing how the mentioned overhead can be effectively controlled and how the proposed approach is applicable to systems characterized by considerable complexity.

3.9 Use Cases

In this section we present an use case of the previously described runtime reconfiguration techniques. We plot the results obtained while performing the topology selection process over a set of 16 different system configurations. The considered system included 8 processors, 8 private memories, one shared memory, one hardware based synchronization device (namely a test-and-set semaphore used to support mutual exclusion in case of concurrent accesses to shared memory), one output peripheral (used to output the results related to performance extraction and switching activity estimation). The different topologies used to interconnect the cores inside the system where obtained from the application of a simple exploration algorithm, that iteratively clusters the network switch to trade-off latency (the number of hops in the network increases with clustering) vs. frequency (smaller switches have a smaller critical period). The application executed on the system is again the well known $RadixSort$ included in the Splash2 benchmark suite [44].

The adopted hardware FPGA-based platform is the same reproduced in Figure 3.5. In figure 3.15 we show the results of the evaluation obtained with respect to total execution time, to total latency, and to total energy and power dissipation. The figures showing absolute time, power and energy values are obtained by back-annotating the FPGA emulation results with the models described in Section 3.6.

The RTL libraries synthesized for FPGA (for evaluation) are the same libraries that would be used for ASIC (for real production). Provided that the bypassing mechanism is adequately applied and given that the presence of unused ports inside the WC topology do not affect the switch functionality, we can state that the prototyping does not insert any error in the estimation of "functional related" (execution time, latency, switching activity) performances. Cycle/signal level accuracy is guaranteed by definition without the need of a test comparison (emulated vs. prospective implementation). The accuracy of the models described in Section 3.6 is assessed to be lower then 10%, with respect to post layout analysis of real ASIC implementations. From the performed analysis a designer could estimate $topology\_0$ to be the optimal configuration for the target application from the point of view of energy consumption and actual execution time. To identify communication bottleneck or congestion/power hot-spots inside the topologies, node-level detailed performances can be obtained, referring to each single port of the switches included in the topology under test.

All the presented data are obtained after traversing only one synthesis/implementation flow. As already proved in Section 3.7.3, the time needed to run synthesis and implementation with commercial tools increases with the size of the system and can be estimated in a matter of hours. Exploiting the runtime configuration capability, enables to try different interconnection topologies and configurations by performing only one actual FPGA synthesis and implementation flow plus several software compilations and FPGA programming, con-

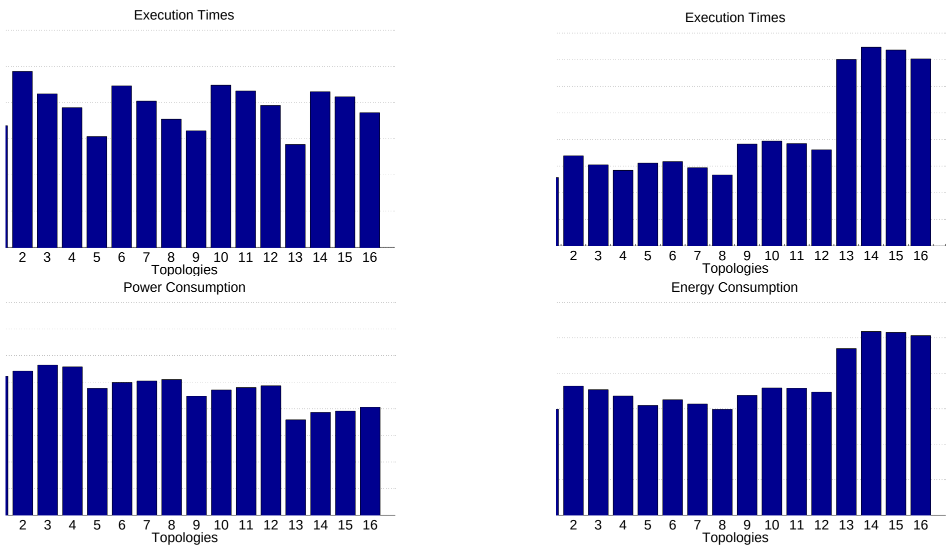


Figure 3.15: Use cases results. Execution times are reported for the different topologies under emulation, both in terms of clock ticks and real seconds, thus accounting for the architectural maximum operating frequency, which is obtained via analytic modeling. Moreover, modeled power and energy consumption figures are reported for the different topologies

suming approximately a matter of few minutes each, allowing a time saving that increases with the number of candidate topologies under prototyping. Figure 3.16 plots the impact of runtime reconfiguration on the overhead introduced by FPGA synthesis/implementation process.

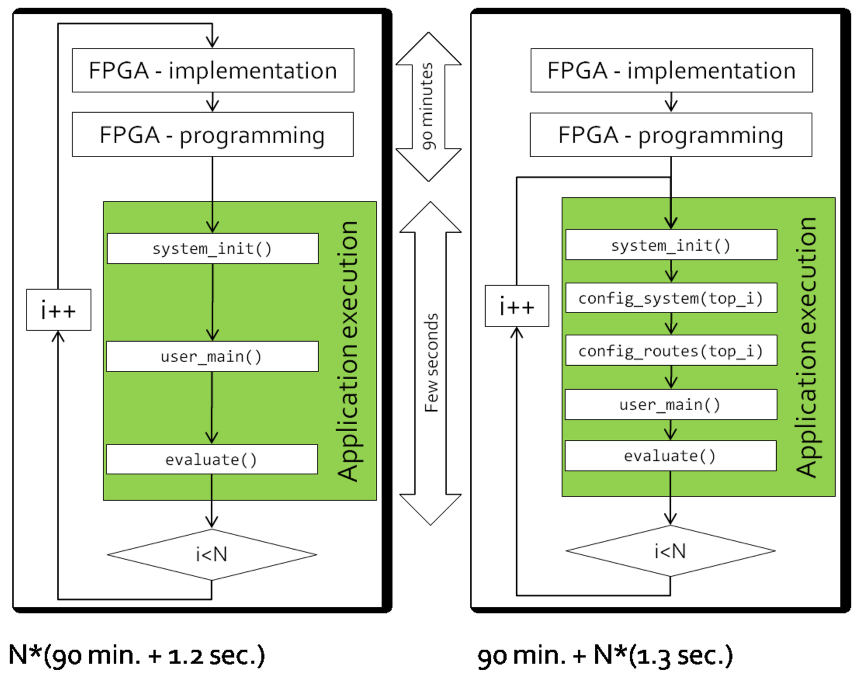


Figure 3.16: Impact of runtime reconfiguration on FPGA synthesis and implementation overhead. The original framework traversing time is compared with the modified framework one.)

3.9.1 Hardware overhead due to runtime configurability

We present an overview of the overhead introduced by the support for runtime programming of the network in terms of hardware resources. We compare, for two different Design Space Explorations (DSE in the following), the programmable logic resources required for the implementation of the most hardware-hungry topology under test (when implemented for a stand-alone evaluation without support for runtime reconfiguration) with those required to implement the corresponding WC template. The DSE runs involved respectively 4 and 16 topologies under prototyping (TUP in the figure). As can be noticed from Tables 3.2 and 3.3, the introduced device utilization overhead is limited in both cases and is controllable when the number of candidate topologies increases. We report also a comparison related with working frequency, potentially limiting the emulation speed. The table shows how the critical path is almost insensitive to the introduction of the support for rapid prototyping. This is mainly due to the fact that the critical path is always bounded inside the switches, thanks to the WC topology definition algorithm that limits the bypassed switches at the boundaries of the topology. The results show how the overhead reduction mechanisms explained in section 3.8.6, effectively allow to support prototyping of quite complex systems with state-of-the art commercial devices.

	Occupied Slices	**Slice Registers**	**Slice LUTS**
Largest TUP (4 topologies DSE)	17327(33%)	33,885(16%)	44673(21%)
WC (4 topologies DSE)	20627(39%)	41313(19%)	58862(28%)
Largest TUP (16 topologies DSE)	17397(33%)	34487(16%)	44926(21%)
WC (16 topologies DSE)	21815(42%)	44943(21%)	64696(31%)

Table 3.2: Experimental results related with the hardware overhead introduced by the support for fast prototyping

	Critical path
Slowest TUP (4 topologies DSE)	10,902
WC (4 topologies DSE)	10,902
Slowest TUP (16 topologies DSE)	10,976 ns
WC (16 topologies DSE)	11,307 ns

Table 3.3: Experimental results related with the critical path overhead introduced by the bypassing logic.

Chapter 4

A parallel software simulator for high performance computing systems

As described in Section 1, the high performance computing field is undergoing a phase of crucial changes and complexity increases, with the introduction of the first supercomputer architectures based on hybrid-core computing nodes, that couple the programmability of classic parallel CPUs with the impressive amount of parallelism of modern GPU units. This boost in both system and single-node complexity requires support for effective full-system simulation, and recent approaches have started to look at simulator scalability by exploiting modern parallel commodity computing platforms, like SMPs, grids and CoWs.

In this chapter, we present the main research activities carried on in the field of parallel software simulators for high performance systems. We will present and discuss a cycle-level simulator of the highly multithreaded Cray XMT supercomputer. The simulator runs unmodified XMT applications. We discuss how we tackled the challenges posed by its development, detailing the techniques introduced to make the simulation as fast as possible while maintaining a high accuracy. By mapping XMT processors (ThreadStorm with 128 hardware threads) to host computing cores, the simulation speed remains constant as the number of simulated processors increases, up to the number of available host cores. The simulator supports zero-overhead switching among different accuracy levels at run-time and includes a network model that takes into account contention and hot-spotting. On a modern 48-core SMP host, our infrastructure simulates a large set of irregular applications 500 to 2000 times slower than real time when compared to a 128-processor XMT, while remaining within 10% of accuracy. Emulation is only from 25 to 200 times slower than real time.

Irregular applications, such as data mining and analysis or graph-based computations, are applications that show unpredictable memory/network access patterns and control structures [41]. Highly multithreaded architectures with large processor counts, like the Cray MTA-1, MTA-2 and XMT, appear to address their requirements better than commodity clusters. However, the research on highly multithreaded systems is currently limited by the lack of adequate architectural simulation infrastructures due to issues such as size of the machines, memory footprint, simulation speed, accuracy and customization.

As already hinted in Section 2, the analysis of the most relevant solutions at the state of the art suggested a number of critical features that appear in modern parallel software simulation systems. The main ideas that led us in developing our simulator are listed and discussed as follows:

- Execution of unmodified code - The simulator should implement the whole Cray XMT ISA and be able to run unmodified binary code, in order to be fully employable in an actual software development cycle with none or minimum overhead. Moreover, this would allow to monitor with the highest accuracy the runtime behavior.
- Customization - For such a particular machine, the simulator has to be customized to match some of the most inherent architectural features. For example, physically distributed memory with an abstract shared address space must be reproduced and/or modeled. Fine-grain synchronization is another critical feature which the machine implement at the level of the single memory cell, and must be reproduced exactly.
- Speed - The simulator has to be as high-speed as possible, in order to enable real simulation scalability to high number of processing cores and memories (thousands). In order to do so, some aspects may have to be simulated with high detail, while others may be modeled at a more abstract level. For example, as we will discuss in the following sections, since the machine implements a fine-grained memory scrambling scheme, a pretty uniform network traffic can be supposed to take place because of the uniformly spread memory accesses. This assumption may lead to the consideration that the whole interconnection layer might be simulated by only modeling memory access latency with reasonable accuracy.
- Dynamically selectable speed/accuracy balancing - Different accuracy-level simulation operating modes might be implemented and provided for dynamic switching at application execution time. This feature is crucial in order to allow the execution of full binaries including OS, runtime and application. Different accuracy levels might be desirable in different sections of the binary execution, trading off accuracy and simulation speed.
- Thread-core mapping - Particular attention should be paid to the mapping of the simulated cores to the simulating threads, and in turn to the physical cores of the host machine. This factor plays an important role in simulator performance and scalability.

4.1 The Cray XMT

Before presenting the simulator conception and development, we are in this section going to discuss the main features of the target supercomputing machine, the Cray XMT. The Cray XMT is a shared-memory multithreaded supercomputer developed by Cray under the codename “Eldorado“ [29] to specifically address the needs of irregular data-intensive applications. The Cray XMT is the successor to the Tera MTA and the Cray MTA-2. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with ThreadStorm processors, communicating through the Cray Seastar-2.2 high speed interconnect. It can allocate up to 96 ThreadStorm processors per cabinet and 8192 per system, with up to 128 TB of shared memory.

Figure 4.1 plots the system organization, while Figure 4.2 gives a view of the single node architecture organization.

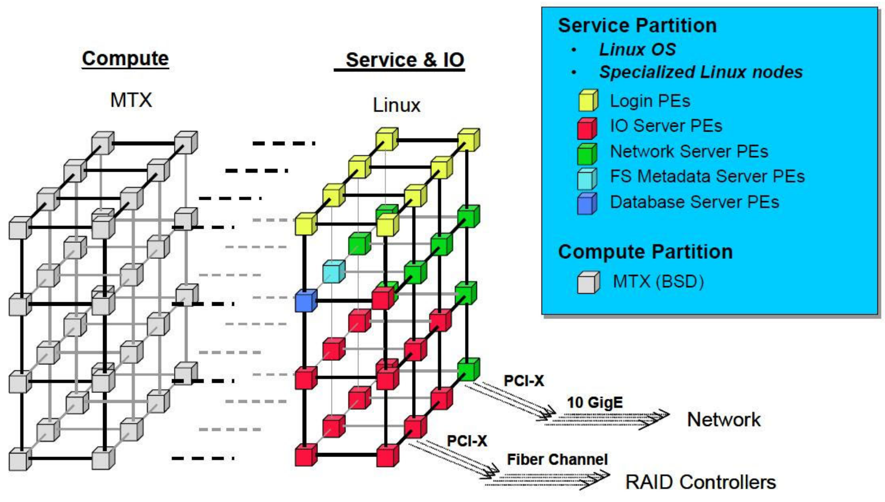


Figure 4.1: System-level XMT architecture

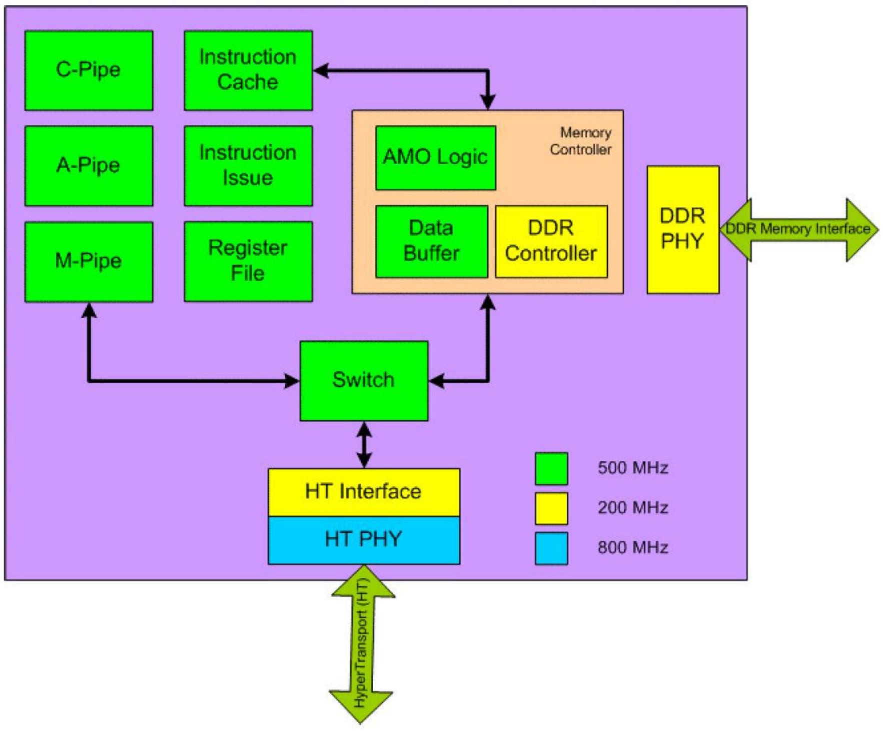


Figure 4.2: Node-level XMT architecture

Each ThreadStorm is a 64-bit Very Long Instruction Word (VLIW) processor containing a Memory unit (M-unit), an Arithmetic unit (A-unit) and a Control unit (C-unit). The Thread-Storm is able to switch, on a cycle-by-cycle basis, among 128 fine-grained hardware streams to avoid the stalls generated by memory accesses and provide a high level of latency tolerance. At runtime, a software thread is mapped to a hardware stream which includes a program counter, a status word, a set of target and exception registers, and 32 general purpose registers. The pipeline has a length of 21 stages for all the instructions and, by design, a new instruction from the same stream cannot be issued if the previous did not exit the pipeline. However, the processor supports look-ahead, so independent instructions of the same stream can be issued every 21 cycles, even if memory operations have not completed. The look-ahead is inserted by the compiler and it is up to 8 instructions, so each stream may have up to 8 pending memory operations at the same time. Thanks to the lookahead feature, the memory operations can complete out-of-order. The processor has a 64 KB, 4-way associative instruction cache for exploiting code locality, and runs at a nominal frequency of 500 MHz. All the processors are connected through a 3D packed-switched toroidal interconnection network based on the Cray Seastar2 System-on-Chip. The Seastar2 is an evolution of the Seastar chip [16], which includes, in a single ASIC, a 3D router, high-speed serial links, a Network Interface and a dedicated PowerPC 440 embedded processor to implement DMA functionalities.

The memory subsystem of the service and the compute nodes are completely separated, and do not interact each other. The memory controller of each ThreadStorm processor manages up to 8 GB of 128-bit wide DDR memory and has a 128 KB, 4-way associative data buffer (small cache) that helps in reducing the access latencies. The memory of the system is accessed with a shared memory abstraction: load/store operations to any physical memory location can be generated from any ThreadStorm processor connected to the Seastar-2.2 network. The network is configured in a 3D toroidal topology. Physical addresses up to 48 bits are supported. The memory is hashed with a granularity of 64 bytes, so that logically sequential data are allocated to physical memories attached to different processors, reducing memory and network hot-spots. The hashing mechanism is always active: even when using a single processor, the accessed data are distributed, almost uniformly, on the memories connected to all the processors of the full system. Figure 4.3 represents the hashing mechanism in a 128-processor machine configuration.

Each memory location is associated with a *full-empty bit*, which works as a lock, a *pointer-forwarding bit*, which signals memory locations containing pointers rather than data, allowing automatic generation of a new memory reference, and two *trap bits*, which allow the generation of a signal when the location is accessed for storing or loading. Trap bits are widely used by the runtime system for setting monitors (for synchronization) or setting function continuations (for workload distribution). The ThreadStorm processor only uses 2 additional bits with respect to memory words of 64 bits. The first represents the full-empty state, while the second signals if the location is forwarded or has any of the trap bits enabled. In such cases, the value contained in the memory cell is considered a pointer, so the last three bits, which are not used for addressing (memory is addressed in 64-bit words), can store the real values of these access state bits. The memory references may also return specific error codes for latency limits or after a predefined number of retries, if the location is not in the expected unlocked or locked state.

The software environment on the Cray XMT is composed by a custom, multithreaded operating system for the ThreadStorm compute nodes (MTX), a parallelizing C/C++ cross-compiler targeting the ThreadStorm processor, a standard Linux 64-bit environment for the service and I/O nodes, and by the libraries that provide communication and interaction between the two parts of the system. The parallelism of an application for the Cray XMT is

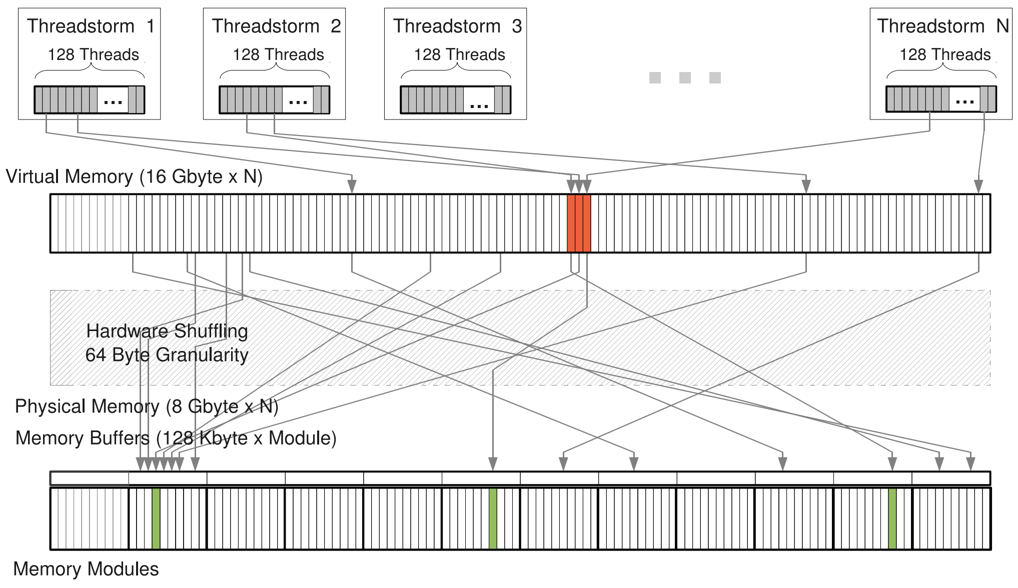


Figure 4.3: Overview of the Cray XMT hashing mechanism

expressed through the use of *pragmas* and is mainly extracted by analyzing loop nests and by mapping the loop iterations to threads.

4.2 Simulator overview

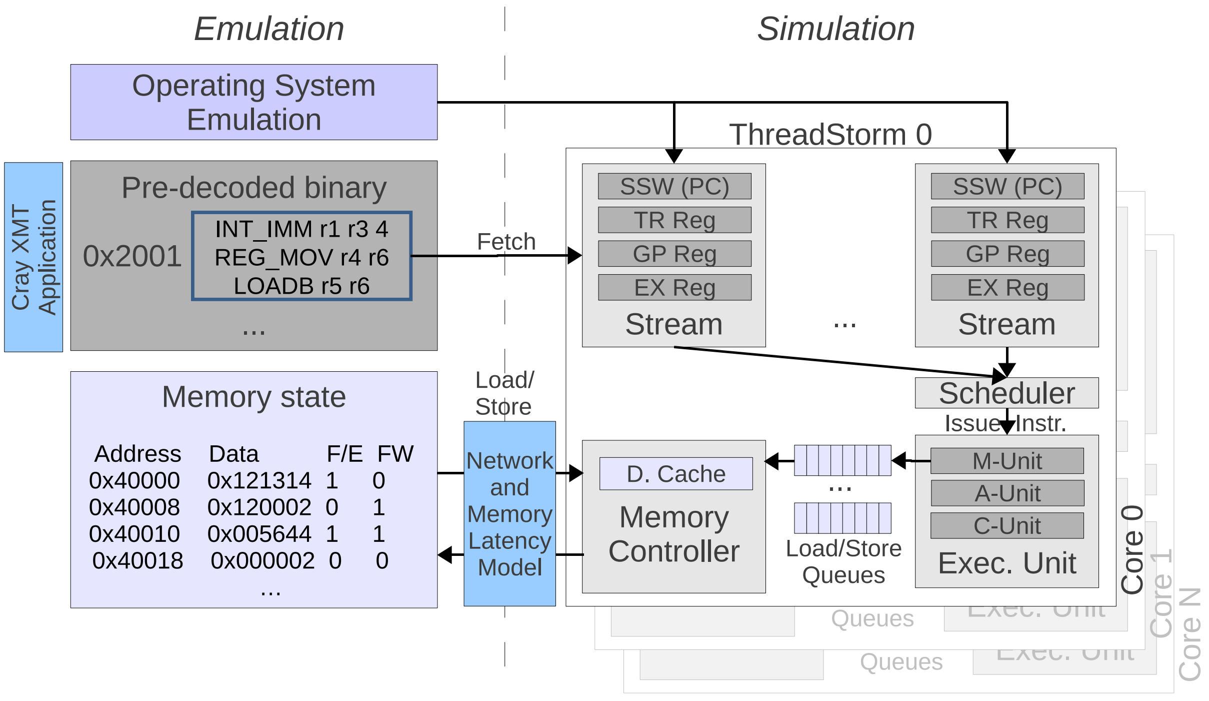


Figure 4.4: Overview of the simulator infrastructure

Figure 4.4 shows an overview of the simulator architecture. The simulator has been developed as mixed C/C++ code, and is portable across different Linux platforms. The simulator only models the ThreadStorm compute nodes, where the computation of the applications happens. The binary of the application is loaded and predecoded at simulator startup. Pre-decoding is a key feature of our simulator, since it allowed us to avoid modeling explicitly the pipeline decoding stage. The binary instructions are therefore already decoded after this pre-decoding stage. The memory state of the simulated applications is saved in a structure optimized for size and speed, which contains both the data and the access state bits of each memory cell. Operating system emulation support is provided by intercepting the system calls in the simulated programs and by mapping them to the host system calls. As we will see in the following, this system call emulation mechanism corrupts the simulated functional timing concept, therefore has to be isolated from the timed simulation phases. Binary decoding, memory state, and system call management constitute the emulation component of our infrastructure, while the ThreadStorm models represent the simulation component. The various architectural building blocks of the processors are modeled in dedicated C++ classes, which offer a clock tick method defining their timed behavior. The whole system is then instantiated inside a top-level *System* class, responsible for activating the clock tick methods of the whole hierarchy of components. Figure 4.5 shows a diagram of the class hierarchy within the project. Only the main classes are shown for the sake of simplicity.

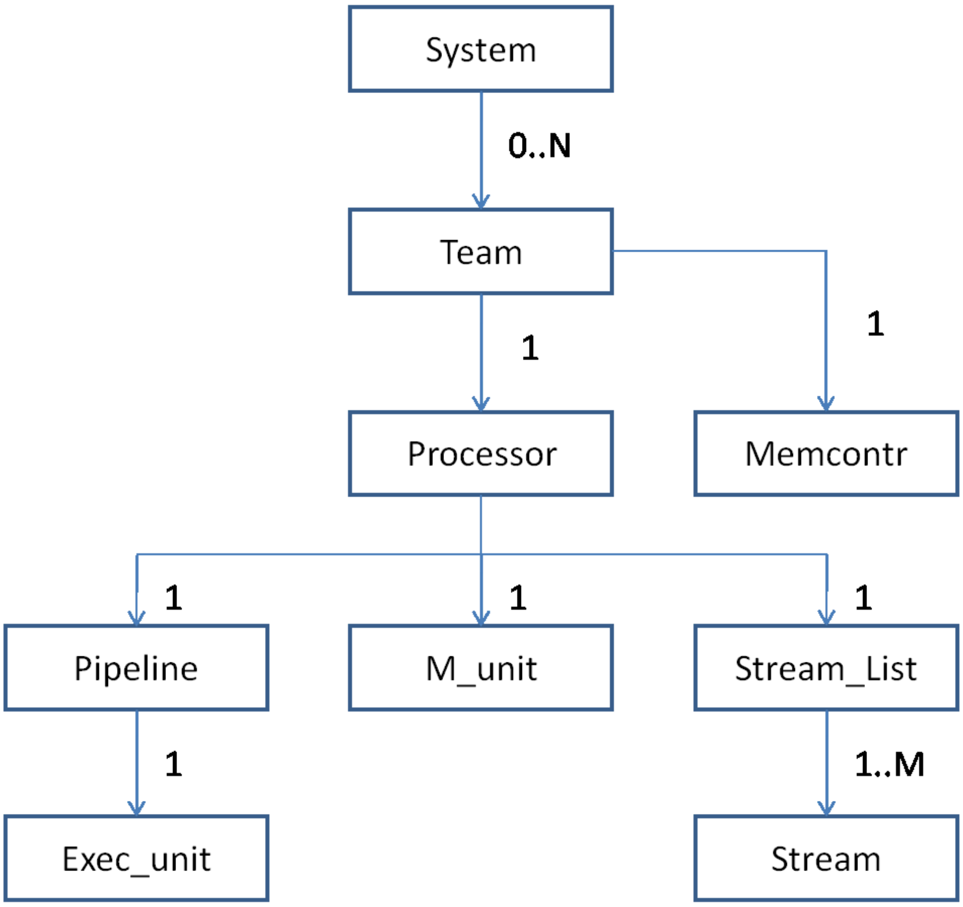


Figure 4.5: C++ class hierarchy

The parallelization strategy of our simulator generates a host thread for each simulated ThreadStorm processor and relies on *pthreads*. In the best case scenario, a host core is assigned to each simulated processor. In case there are more simulated processors than host cores, we try to assign the same number of ThreadStorms to each host core. This mapping strategy was chosen in order to provide a first level of load balancing among the different threads that simulate the target cores. Figure 4.6 shows the adopted mapping strategy.

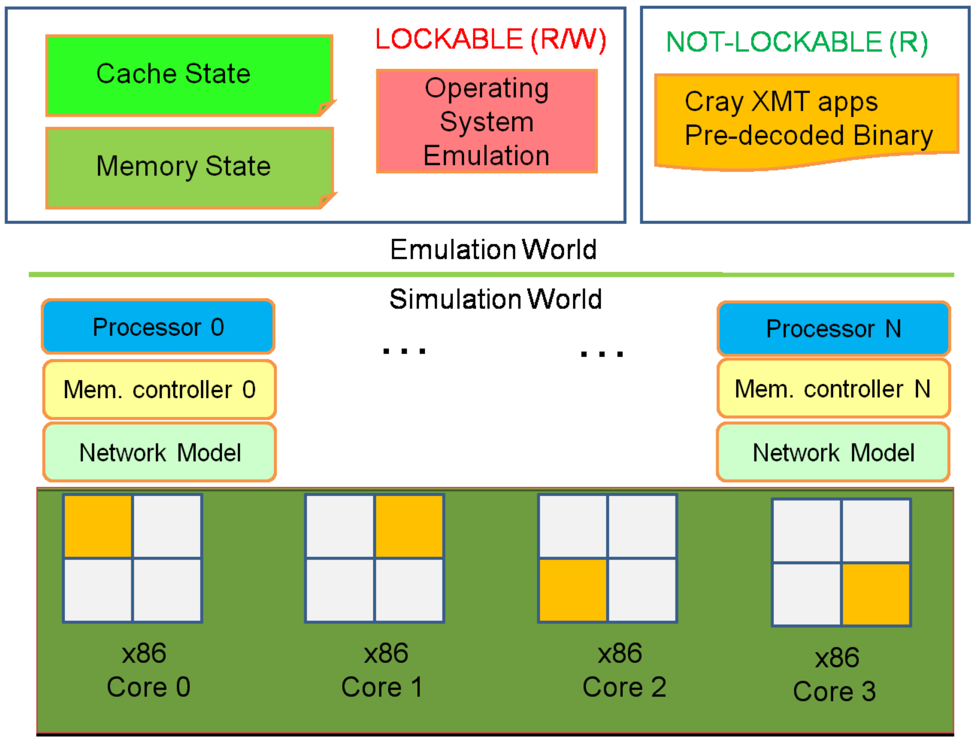


Figure 4.6: Sketch of the target core - host core mapping strategy on a 4-core x86 processor.

Regarding the simulation of the memory access latency, our simulation approach associates to each issued instruction a latency counter. When the latency of the instruction has expired, we update the corresponding stream state. At each clock tick, the simulator updates the clock variables and decrements the latency counters. We will describe in detail the latency assignment phase in the following sections. High performance of simulation is maintained by applying several strategies, both inside the processor models and all over the simulation infrastructure, that take into consideration the specific characteristics of the Cray XMT and of the SMP host systems.

At the processor level, we support dynamic addition and removal of streams, depending on the requests of the XMT runtime, which we run together with the applications through static linking. Moreover, we integrated the support for zero-overhead switching among different accuracy levels at the application level, as will be discussed in the following. Since the Cray XMT can execute multiple jobs at the same time, the runtime is also able to dynamically allocate processors (defined as logical *Teams* of streams) from a minimum to a maximum requested number. On the real machine this is useful for using processors that still have free streams in a multi-job environment. On the simulator, the team allocation calls can be intercepted to dynamically spawn new simulator threads.

At the system level, the main elements of our approach rely on the objective of simulating a machine that widely exploit the shared memory abstraction. As discussed previously in Section 4.1, the memory of the Cray XMT is physically distributed, but a large number of memory references is always traversing the network of the processors due to the address space hashing. Thus, the typical approach in state-of-the-art simulators of modeling each node as a component able to access only its own physical memory subset would incur in a high overhead. This overhead would descend mainly from the messaging among the nodes and from the operations required by the memory controllers for reading/writing data to the memory state. The presence of memory hot-spots in the simulated applications, where many references are concurrently accessing the same locations, would also hinder the overall performance of a parallel simulation by overloading few cores with all the incoming memory operations. Instead, we observe that, when running on a SMP system, the data structures of the simulator are already shared. Consequently, each ThreadStorm model can already access the locations referred by the memory operations, without requiring to describe the interaction among the different models and to perform a detailed simulation of the interconnection network. The timings of local and memory remote operations can, in fact, be obtained through a sufficiently accurate analytic variable latency model [17]. However, the latency model should be able to account for contention effects, and allow easy modifications of the parameters for exploring possible future new configurations. Furthermore, this approach requires efficient synchronization strategies to handle cases when multiple simulator threads (each one simulating a ThreadStorm) try to access the same memory location at the same time. Finally, it is still necessary to find low overhead solutions to synchronize the simulation threads, since some threads can potentially proceed faster than others.

The emulation part of the infrastructure is composed of two elements. The first is a frontend that decodes the XMT binary applications and manages the shared memory state. The second implements system call support. We present them in the following sections.

4.2.1 Front-end

The simulator frontend [35] is responsible for decoding the binary of the XMT applications and for managing the shared memory state of the simulator. At simulation boot-time, the frontend reads the executable, decodes the VLIW instructions, and organizes the data in three memory regions. The first contains the .TEXT segment, the second hosts the .BSS and .DATA segments and the third is used to manage stack and heap segments. The first two regions are organized as radix trees, to allow a fast lookup when fetching the instructions or loading/storing data, while maintaining a reasonable size of the structures. For the .TEXT region, each leaf correspond to a pointer to a structure containing the operations (up to three) forming the VLIW instructions and the lookahead value. For the .BSS and .DATA region the leaves contain the data of the simulated memory cells. Once the simulated virtual address has been identified in the host, the physical host address is used, avoiding the repetition of the search in the radix tree. The elements of the stack and heap regions, instead, are dynamically allocated on the host when requested by the simulated programs. To guarantee high performance when allocating, reading, writing and freeing these dynamic memory regions, their identifiers and sizes are tracked through tables, accessed by masking and shifting a reserved range of addresses of the simulated memory space. The advantage of using masks and shifts over accessing array elements is established with modern processor architectures (in terms of available registers) and compiler technologies (in terms of best exploitation of the available registers).

The simulated memory cells of the .BSS and .DATA regions and of the stack and heap regions are composed by a 64-bit word, which contains the real data, and a 8-bit tag which stores the access state bits. We use 8-bits for the access state to guarantee portability of the simulator and efficiency in accessing the simulated memory cells.

4.2.2 System calls

In the Cray XMT, when a program performs an operating system call, it executes a load on a reserved memory location where the syscall table is contained. In our simulation infrastructure, we trap these loads, and determine which specific system call is invoked by checking the offset of the load on the syscall table. We then subsequently intercept the jump to the syscall and execute it on the host, reading the parameters and writing the return values in the appropriate registers following the function call convention of ThreadStorm Application Binary Interface (ABI).

The Cray XMT environment supports around 240 system calls which perform operations relating to memory allocation (e.g. *brk*, *mmap*, *munmap*), file statics and reading/writing, process control and information maintenance. Process control and information maintenance system calls includes identification of the executing streams, the identification, acquisition and release of teams, the setting up and the reading of processor hardware counters (e.g. load/store counters, trap counters, stream create counters) and are proprietary for the machine. We implemented the full set of common operating system calls and all the required specific XMT system calls to seamlessly execute any application. When intercepted, memory allocation system calls are obviously mapped to our frontend, which allocates and deallocates the requested memory from the memory state. File management system calls, instead, are executed directly on the host file system. Note that the XMT runtime is based on Free BSD, while our simulator has been designed for Linux hosts, so even the file management system calls may return different values. The simulator integrates the conversion logic to map the values returned by the host-executed system call in the format expected by the runtime. Process control and information maintenance proprietary system calls finally interact with the simulator as requested. We underline that in the XMT, while team acquisition and release are controlled with system calls, streams are managed with explicit create and quit assembly instructions.

When a system call is invoked, before executing it we guarantee the consistency of the state of the processor by allowing the completion of all the current pending memory operations. If a syscall is invoked during parallel simulation, we acquire a lock as soon as it is recognized, assuring that there is only a syscall executing and so protecting memory allocation and file operations.

4.3 Processor model

In this section, we describe the implementation of the ThreadStorm processor model, detailing the mechanisms for dynamic addition and removal of streams and teams. As shown in Figure 4.4, for each ThreadStorm processor, we designed the stream states, the stream scheduler and the pipeline with the three execution units. Each stream state comprises:

- the 32 General Purpose Registers (GP)
- the Stream Status Word (SSW)

- the 8 Target Registers (TG)
- the 8 Exception Registers (EX)

The M-unit supports lookahead the way the real machine does, and has a load-store queue for each stream that manages up to 8 pending memory operations with out-of-order retirement. We developed specific benchmarks to assess that also the real machine is implementing out-of-order retirement of the completed memory operations. The A-unit and the C-unit are mostly similar, since the C-unit can execute many of the arithmetic operations of the A-units plus some control operations, and are modeled together. Each processor has its own memory controller, which manages the accesses to the memory state following the access control rules dictated by full/empty, forwarding and trap bits. The memory controller also contains the model of the 128 KB data cache, and interacts with the memory and network latency model when local or remote memory locations are accessed. The main features of the memory controller will be discussed in the following sections.

4.3.1 Dynamic stream and team addition/removal

When the runtime system requires new streams, it executes one of the assembly instructions that verifies how many streams are available in the processor and eventually reserves them for future create operations (STREAM_RESERVE). To effectively create a new stream, it then invokes another assembly instruction (STREAM_CREATE_IMM), which sets the basic context, the initial status, the first program counter and finally sets the stream as available for the hardware scheduler. The stream is released with a STREAM_QUIT assembly instruction. It is important to say that all these assembly operations require the entire VLIW instruction (M-A-C operations).

To speed up the simulation of the ThreadStorm processor, instead of instantiating the stream contexts from the beginning, we implemented a solution that intercepts the calls to the stream-related assembly instructions and dynamically adds or remove the stream contexts to the model. From the implementation point of view, we use a circular double-linked list where each element of the stream list points to a dynamically allocated stream context. Elements are circularly connected to the previous and next elements of the list, so streams can be dynamically added and removed at any position with a constant number of operations, given only the stream identifier. The stream is available for scheduling if it appears in the linked list, so no checking of the stream status is required, and the only procedure performed by the scheduler is to follow the list. Since in many sections of the program the runtime does not use all the 128 streams of the processor, the approach significantly reduces the simulation time without changing the modeled behavior. Furthermore, the approach can also limit the memory footprint of the models when simulating those programs for which the developers asked, through the appropriate MTA pragmas, a low number of streams.

A similar strategy is followed to spawn the threads that simulate the ThreadStorm processors. The XMT runtime, in fact, is able to acquire processors progressively, up to the maximum number that is requested when a job is submitted to the machine. We take advantage of this behavior by trapping these requests, which, as previously discussed, on the real machine are performed through system calls and dubbed as *Team* additions. After trapping the calls, we consequently spawn new threads, each one simulating a different ThreadStorm processor.

4.4 Dynamic switching

Another crucial feature for speeding up simulation is the support for dynamic accuracy switching during the simulation itself. Dynamic accuracy switching mechanisms make possible to enable accurate, but slow, simulation only on the interesting parts of an application (e.g., computational kernels), while other parts (e.g., runtime boot, memory allocation) execute in a functionally accurate, but much faster, emulation mode. This allows to run complete applications with full datasets rather than stripped down versions, augmenting the relevance of the analysis. We implemented this feature with no overhead for the simulator, by offering a primitive that can be included at the application level. The following application code snippet gives an example of how this simulator call can be inserted in the application, to dynamically adapt the accuracy level to the relevance of different code regions.

```
...
#pragma mta fence
SIMULATE(1);
#pragma mta fence
\emph{Non timing critical code}
...
#pragma mta fence
SIMULATE(2);
#pragma mta fence
...
for(unsigned i=0; i<ARRAY_SIZE; i++){
reduction+=array[i];
}
#pragma mta fence
SIMULATE(0);
#pragma mta fence
...
```

The first code region is emulated with intermediate accuracy level, by calling the `SIMULATE(1)` primitive. The MTA *fence* pragmas tell the compiler to not move instructions before or after the place where they are placed. In the code, the reduction for is then simulated with high accuracy by dynamically switching to the `SIMULATE(2)` mode, while the remaining of the code is just emulated, by switching back to the `SIMULATE(0)` mode.

The simulator traps this primitive, similarly to a system call, during the execution of the program and changes the accuracy level without disrupting the execution flow. The `SIMULATE()` primitive is actually mapped in the same syscall table, but with reserved address values. The dynamic switching works by changing the latency counters used in our memory access latency simulation approach and by intercepting the stream-related instructions. The simulator supports three accuracy levels, as follows:

- Emulation 0: single cycle latency for memory operations and the pipeline, high retry limit, no stream creation is allowed
- Emulation 1: single cycle latency for memory operations and the pipeline, high retry limit, stream creation allowed

- Simulation: Network and memory latency models for memory operations, 21 cycles latency for the pipeline, real machine retry limit, stream creation allowed.

Emulation 0 removes all latencies, thus it is to issue an instruction and execute all launched memory operations in each simulated clock tick. It runs a single stream for each team (processor) allocated, therefore achieving higher speed. *Emulation 1* adds stream creation, allowing the application to grow the number of streams to the number requested by the runtime (or up to the physical limit, whichever is reached first). It can be useful for executing warm-up runs, allowing the runtime to perform stack expansion before the timed runs. For these two accuracy levels we also raise the retry limit for the memory references. The retry limit is a parameter of the Cray XMT runtime system that determines the maximum number of times a memory reference can be retried before the memory operation returns with an error code, when the destination memory cell is locked through the full empty bit. When the retry limit is reached, an exception is generated and then handled by a runtime trap handler, which eventually re-executes the memory operation. If it fails again, it moves to a monitor-type synchronization by setting one of the trap bits and queuing the waiting operation. This mechanism is used for synchronization-intensive programs to reduce the traffic due to repetitive polling on locked memory locations. Since the trap handler is quite large, and all the latencies are zeroed, in low accuracy modes we allow the operations to continuously retry until a location is unlocked. *Simulation*, instead, enables the maximum accuracy by using the full network and memory latency models and simulating the real latency of the ThreadStorm pipeline. Since traps are relevant for synchronization operations, the retry limit is also set to the values used on the real XMT.

Table 4.1 recaps the main features of the three accuracy levels selectable at runtime.

	Emulation 0	**Emulation 1**	**Simulation**
Memory Latency	NO	NO	YES
Cache Latency	NO	NO	YES
Retry Limit Traps	NO	NO	YES
Pipeline Latency	NO	NO	YES
Stream Creation	NO	YES	YES

Table 4.1: Main features of the three accuracy levels

4.5 Host thread synchronization

One of the key features of every parallel software simulator is thread or process synchronization. It usually is the limiting factor toward scalability, and is required both with shared-memory based simulators, which usually employ threading libraries to this aim, and with distributed-memory simulators, which usually rely on process APIs such as the MPI library. In our simulator, all the simulated processors are clocked independently. From the functional point of view, synchronization of the simulated programs is obtained by the simulated synchronization features inherent to the ThreadStorm themselves. However, when simulating systems with multiple ThreadStorms at the maximum accuracy level, we may have threads that perform more work than others due to an unbalanced stream andor team assignment. Therefore, to obtain a state coherent with the real machine, we still need a synchronization mechanism to reduce the divergence among the simulator threads.

As explained in Section 2.2, several alternatives were available at the state of the art for the issue of simulation threads synchronization. Some are trivial, other are complex and employ speculative execution of the simulating host threads and consequent roll-back mechanism in case inconsistencies are generated. In general terms, a trade-off can be identified between the synchronization granularity, which defines the slacks that might occur in simulated time among the host threads, and the performance overhead introduced by the synchronization primitives.

In our simulator, this objective is reached by implementing a “relaxed” *pthreads* barrier synchronization. We insert the barrier construct every 65,000 simulated clock ticks, which we empirically verified to allow for fair progression of all the simulation threads. The parameter is, anyway, tunable. Furthermore, when the number of simulated processors is less or equal to the number of host cores, our infrastructure also exploits thread pinning to remove eventual overheads due to context switching. When the number of threads is higher than the number of cores, instead, we trigger every 1,000 clock cycles a scheduler yield that makes it switch to other waiting threads, guaranteeing a more efficient progression.

The following code snippet identifies the main features of our thread synchronization approach.

```
void * run_thread ( void * data )
{
    ...

    System * system = ( System * ) data->system;

    if ( num_teams > num_proc )
pin_thread ( tid );

    pthread_barrier_wait(&barr);
    int64_t i = 0;

    while ( i < cycles || cycles == -1 ) {
(system->teams[tid])->clock_tick ();

if (i % 1000 == 0)
       sched_yield();

        if ( i % 65536 == 0 ) {
     pthread_barrier_wait(&barr);
            if ( ! ( system->active ) ) {
break;
     }
     pthread_barrier_wait(&barr);
        }
        i++;
    }
...

    return NULL;
}
```

4.6 Memory and Network model

In the Cray XMT, the shared memory abstraction and the granularity of the memory hashing make it extremely difficult to efficiently model the network and its interaction with the memory controller. On the other hand, these same features also allow using a faster latency model for describing the behavior of the memory operations and the network. In fact, the occurrence of network congestion or memory hot-spots is mitigated by the address scrambling. Our assumption is that, when simulating the ThreadStorm processors on a SMP, the memory state is already shared among the threads mapped to the different models, so we only need to determine the delay of a memory operation and, after that delay, change accordingly the appropriate slot of the M-unit. This means that there is no need to perform an actual remote host memory operation every time a target memory operation is simulated, but we only need to model the latency that the target memory reference would incur into, in order to maintain timing accuracy.

To determine these delays, we developed a variable latency model, that takes into consideration topology of the network and behavior of the data caches on the memory controllers. In addition to these factors, the model is also able, upon selection from the designer, to capture contention effects. This model has been conceived and tuned by running ad-hoc benchmarks on the real machine. The validation of the model has also been carried out by comparison with application executions on the real machine. By using such a latency model, we let the cores that simulate different processors access the same shared structure, thus we also devised an efficient method for synchronizing the accesses. In general terms, memorynetwork latency can be expressed as the sum of two components:

$$L = L_{static} + C(\lambda) \tag{4.1}$$

where L_{static} equals the (static) latency of the packet traversing an empty network and C is an estimation of the dynamic contention dependent on the actual load of the network λ. To capture this behavior, we introduced in the latency estimation subsystem a mechanism based on *contention counters*. We are now going to describe how we estimate the static and dynamic contribution to memorynetwork latency.

4.6.1 Static latency estimation

To estimate the static latency of the memory subsystem, we have performed a set of experiments on the real machine, programming several read accesses to a large block of virtually contiguous-address data, while executing a single stream on a single processor. The exact latency of each access has been measured using the ThreadStorm 64-bit clock counter, populating the histogram that we show in Figure 4.7. Since the specific Cray XMT system used for the experiments adopts a 4x4x8 3D-Torus network topology, with dimensional routing, the maximum number of hops covered by a message, ignoring faults, is 8.

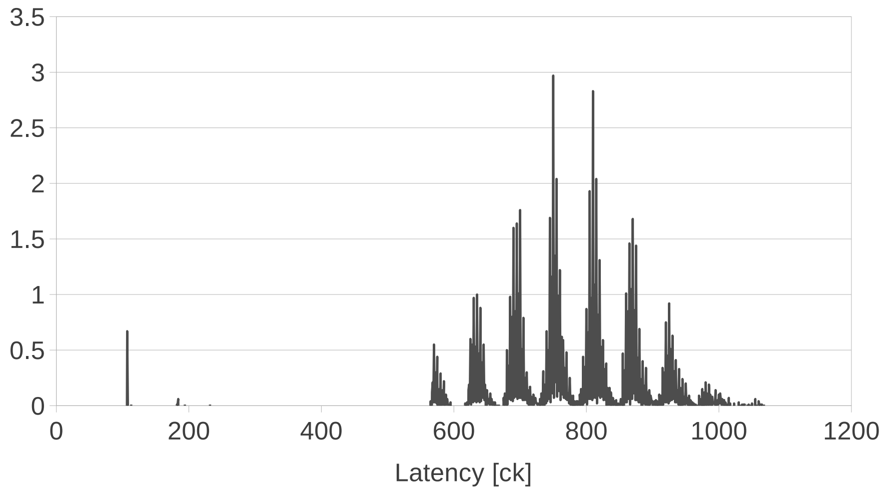


Figure 4.7: Latency histogram for memory-scrambled array accesses

Local memory cache hit and cache miss latencies, as well as the latency incurred by a single-hop memory access within the network, can be spotted from a visual inspection of Figure 4.7. The lowest measured latencies pertain to read accesses on the local memory (i.e., the memory residing in the same network tile of the processor that issues the read operation) with a cache hit. The next larger latency also pertain to a local memory access, but shows an additional delay due to a cache miss. All the higher-latency samples are related to read accesses to remote memories. Such kind of accesses undergo the latency of the source and destination memory controllers and network interfaces, in addition to a variable count of hop delays, depending on the distance between the issuing processor and the target memory. Note that these measurements only depend on the size of the Cray XMT system: even when using a single processor, data are distributed all over the memory partitions due to the shared memory abstraction and the scrambling. Thus, when determining the latency of the memory operations, we are only interested in the size (total number of processors) of the simulated XMT system.

Estimated values of the latency parameters can be obtained as follows:

$$L_{max} = N\_HOP_{max} * L_{HOP} + L_{MC+NIC} + L_{miss} \tag{4.2}$$

$$L_{min} = L_{HOP} + L_{MC+NIC} + L_{hit} \tag{4.3}$$

where L_{MC+NIC} and L_{HOP} respectively stand for the latency of the memory controller - network interface path and for the latency of a single hop within the torus network. L_{max} and L_{min} are easily measured along with the latency acquisition, while L_{hit} and L_{miss} can be determined with high accuracy from the lowest latency values. Therefore, considering a 4x4x8 3D torus network, N_HOP_{max} equals 8, L_{HOP} and L_{MC+NIC} can be obtained with the same accuracy.

4.6.2 Contention evaluation

The latency model described in Section 4.6.1 accounts for the latency variability introduced by the scrambling. However, for high workloads or particular memory access patterns, network latency is also affected by memory/network contention. Contention may appear when applications employ high processor counts or when a limited number of processors access simultaneously the same few memory locations. The latter case might lead to hot-spots in the interconnection network or in the memory controller, even if countermeasures exist in the hardware to mitigate their effects [16], [5].

Memory contention evaluation

We started considering memory contention separately from network contention. In order to evaluate contention on the memory controller only, we isolated the case of different memory references incoming to a single memory controller. The tests run with an increasing number of streams, from 1 to 100, on a single processor. Each stream executes two loads on the local memory controller. The loads are scheduled to be issued at the same time, in a non-blocking fashion. The accessed address is identical for all the streams, ensuring that the same bank inside the local memory is selected. The streams are also explicitly synchronized via an assembly-optimized barrier before the loads can be issued.

The ThreadStorm M-Unit supports a maximum of 180 total pending memory references for all the streams that are running on the processor. Since our test generates up to 100 streams, scheduling 2 loads per stream practically ensures that this limit is never reached, as the streams are still slightly de-synchronized. This residual de-synchronization is due to the cycle-by-cycle stream scheduling process and is thus unavoidable.

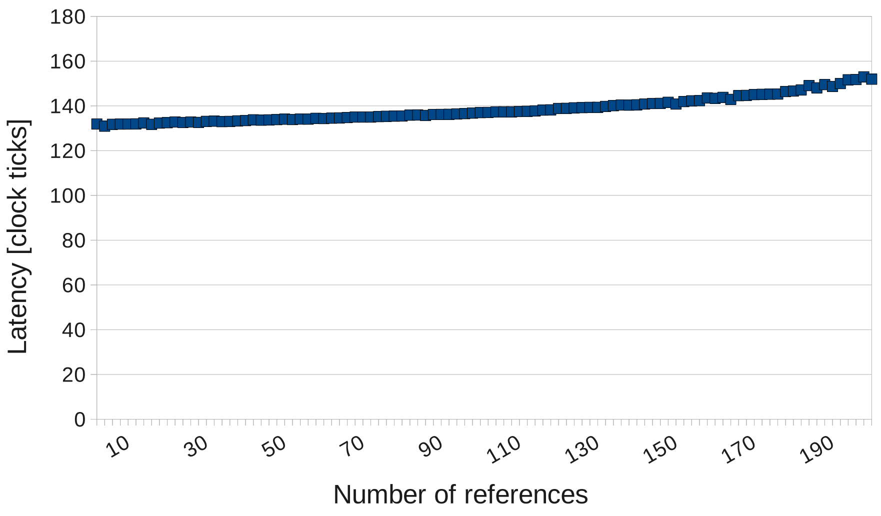


Figure 4.8: Contention on the memory controller

Figure 4.8 plots the results of the described test. The contention generated at the memory controller interface is limited to few tens of clock ticks for up to 200 simultaneous loads.

Network contention evaluation

Regarding contention on the whole network, we designed different tests, using 128 processors to generate an increasing number of memory loads. Every load is performed by a dedicated stream. The number of streams running in each of 127 processors is randomly chosen to perform the desired overall number of loads. The 128th processor, instead, runs a single stream, which performs another load to a specific address and measures the related latency. The address is chosen in order to define a specific distance from the source processor to the destination memory. For a 4x4x8 torus network, distance ranges from 1 to 8 hops. The contention measures were acquired according to two benchmark configurations, which generate traffic patterns with opposite distributions:

1. The interfering streams, running on 127 processors, perform a load on a random memory address, causing the statistical distribution of the generated network traffic to be uniform. The 128th stream, which measures the latency, performs a load on a memory address to an increasingly distant memory, from 1 to 8 hops.

2. All the interfering streams, running on 127 processors, perform a load on the same memory address which is being accessed by the reference stream, on the 128th processor. Again, the distance between the 128th processor, which measures the load latency, and the accessed memory increases from 1 to 8 hops. Since every stream is now accessing the same address from randomly different sources, the network traffic is not uniformly distributed. Instead, the destination router and the closest ones will undergo a heavy traffic, resulting in the formation of network hot-spots.

Figure 4.9 shows the latency for the case of maximum source to destination distance (8 hops), when the interfering streams access random addresses or the same address. In presence of randomly directed traffic, the measured contention is limited to 100 clock ticks, resulting in a barely noticeable 10% delay increase. When all the streams access the same address, instead, a hot-spot occurs and contention generates delays of thousands of clock ticks. It is worth nothing that network contention and memory contention are not distinguishable by only looking at these numbers. However, it is likely that the contention measured when multiple streams are accessing the same location is originating both from the memory controller and from the last routers along the communication path. Thus, we decided to focus on the modeling of the *global* contention behavior, rather than trying to separately identify the two contributions.

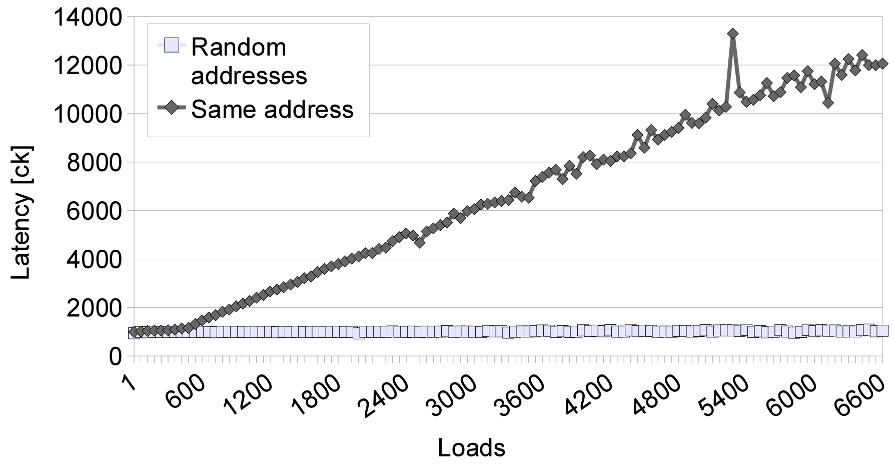


Figure 4.9: Contention measurements for a single load to a memory location at 8 hops, with an increasing number of interfering accesses on *random addresses* or on the *same address*.

Also, Figure 4.9 suggests a linear relationship between the load latency and the total number of packets in the network. This confirms that the arbitration algorithm implemented in the SeaStar router assigns the available bandwidth to the output-queued packets in a fair manner, through the adoption of an age-based arbitration scheme [5].

Single hop network contention

In order to assess the contention occurring on a single router output channel, we also ran a more detailed benchmark. We measured the latency of a load from a processor to a 1-hop distant memory controller, when all the other neighboring processors are accessing, at the same time and with multiple streams, the same memory controller. This benchmark limits the contention generated by the interfering streams to the output port of the router to which the accessed memory controller is attached. In fact, since the interfering streams run on the neighboring processors (1-hop distance), they all converge to the same output port of the destination router from different input ports.

We enabled a maximum of 200 memory load per processor, yielding a total number of 1200 interfering memory loads. Figure 4.10 plots the resulting latency.

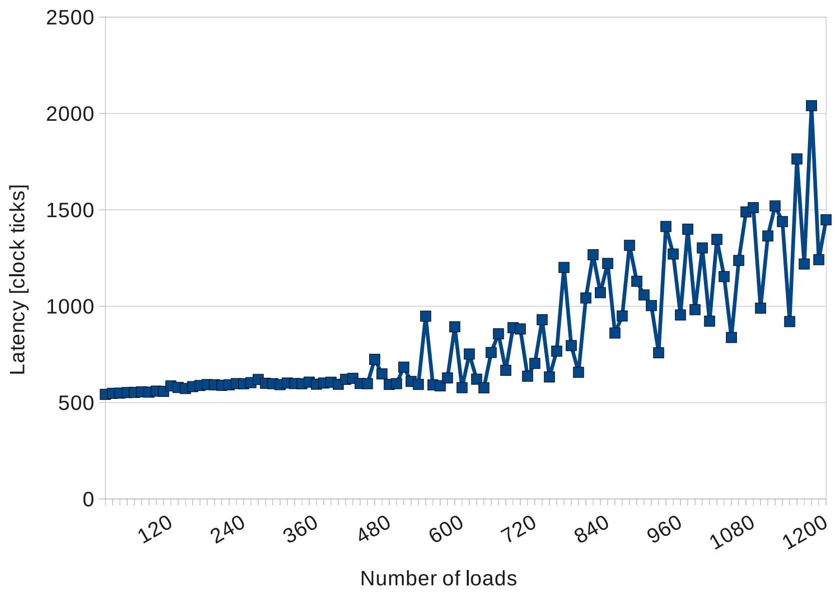


Figure 4.10: 1-hop load latencies with neighboring processors interfering to the same address.

4.6.3 Contention modeling: a speed-accuracy trade-off

In modeling the contention that takes place in the network and memory subsystems, it is well known that a speed-accuracy trade-off holds [17], [23]. In the evaluation of the best solution to this trade-off, we found that the Cray XMT machine, with its particular memory organization and fine-grained hashing of the address space, was worthy a detailed exploration. In this section, we present the characteristics and the implementation details of three different accuracy-level network models that we want to compare. We also highlight how the contention estimation is related to the results obtained through the benchmarks described in Section 4.6.2.

In general, latency assignment to every memory operation within the simulator is handled in a twofold manner. First, a static component of the latency is assigned at the generation of the memory reference, inside the M-Unit. If network contention simulation is enabled, a dynamic contribution is then added according to the level of estimated contention.

To estimate this dynamic contribution to latency, we decided to implement a set of mechanisms based on contention counters. In designing a counter-based contention prediction technique, the main available degrees of freedom are:

- the *granularity* of the contention counters. This ranges from a single contention counter that accounts for all the memory references entering the network to dedicated contention counters for each input/output port of every single router of the network.
- how the latency is calculated as a function of the contention counters. The complexity of this function contributes to the overhead on the simulator speed.

As hinted above, there is a trade-off between the accuracy of contention estimation and the resulting overhead on simulation speed that needs to be solved conveniently. The single counter solution is very simple and lightweight, but does not account for the network topology at all and is not able to distinguish between accesses to different memory locations, therefore missing the formation of network hot-spots. On the other end, dedicated counters for each router input/output port are able to track the actual message path, hence detecting possible network hot-spots, but show high complexity and simulation overhead. We implemented three different models, that simulate the network behavior at an increasing level of detail. Each one of these represents a different evaluation of the simulation speed-accuracy trade-off. All the models are integrated in the simulation framework and the selection on which model to use is done at the beginning of the simulation. The main features and related differences are described below.

Static model

The first model is also the basic operation mode of the simulator, which we already described in Section 4.6.1. It essentially accounts for static network parameters and does not consider any kind of contention when predicting the memory access latency. On the other hand, this solution has two important advantages with respect to simulation speed.

First, very few calculations have to be done to predict the latency value. It is only required to translate the accessed virtual address into a physical address and then calculate the number of hops from source to destination. Dimensional routing is used, according to the SeaStar interconnection network architecture [5]. Second, when integrated into a parallel simulator, this solution does not introduce any data structure to be shared among the simulator threads. For this reason, network modeling does not require any further synchronization mechanism.

Destination-based contention model

The second model is able to capture network contention. It extends the static latency assignment of the previous model with a contention estimation based on counters. Each memory inside the network has an associated event counter. Every memory reference operates on the counter related to the destination memory. The contention contribution depends only on the value of that specific counter. Therefore, the assigned latency depends only on the number of references that are specifically accessing the same destination memory.

The rationale behind such model comes from the experiment results plotted in Figure 4.9. From those experiments, it appeared that the uniformly distributed traffic does not experience significant network contention. Instead, when undergoing highly-patterned traffic flows, the network was not able to mitigate the effects of resource contention, and end-to-end latency increased noticeably.

From an implementation point of view, the simulator operates as follows. When processing the memory reference, the contention counter will be incremented every time a memory reference is entering the network, and decremented at the exit point, separately for the forward and backward paths. The contention contribution is then estimated as a linear function of the counter value over a certain threshold. By looking at Figure 4.9, the linear approximation over a certain threshold appears to be quite a straightforward choice. In Section 4.6.2, we found the slope and threshold values to vary with the distance between measuring processor and accessed memory. Likewise, we now set the threshold value and the slope to depend on the distance between the source and the destination processors. Since we chose to associate a different contention counter to each destination processor, the threshold and slope parameters have been tuned to fit the results for the case of interfering accesses directed to the same address.

This model introduces an additional overhead in simulation speed with respect to the static model. The computation of the contention for each memory reference is quite similar, since it requires only an additional linear curve evaluation. Instead, accessing the counters, which are shared among the different processors, requires specific synchronization for the simulation threads, therefore potentially reducing overall speed.

Detailed distributed contention model

The third model has the highest level of detail and is able to capture contention on the single router output link assignment. It associates an event counter at each different router output port and traces the complete path followed by the packet from source to destination. The entire path through the network is simulated through a Finite State Machine (FSM). For every transaction, the FSM keeps trace of the router that contains the packet, at each specific clock tick. The path is calculated using dimensional XYZ routing. The state changes when the latency necessary to traverse that router has expired. The latency is estimated before entering the router, and accounts for two contributions. The first one models static router delay and is constant. The second one models actual contention on the required output port, considering the number of packets that are currently requiring that port. This number is stored in the contention counter associated to the desired output port. The counter is incremented once the packet enters the router and decremented once the link is assigned to that packet. In Section 4.6.2 we presented a benchmark that isolated contention on a specific router output port. Figure 4.10 reported the latency-vs-traffic curve. According to the curve, we modeled the relation between the contention contribution to router latency and the reference counter to be linear with threshold. The threshold and slope values have been tuned to fit the least square regression line of the curve.

Although the model is able to capture link contention, it does not account for packet size and for worm-hole routing effects, which take place when packets have more than 1 flit. This assumption relies on the fact that SeaStar2 network packets have a relatively low limit for the payload size (64 bytes) [5].

This model generates a significant overhead to simulation speed, both for latency computation and for thread synchronization. First, it inserts all the computation necessary to handle the network FSM. Second, each memory reference requires several accesses to the shared output port contention counters, instead of a single access as in the previous model.

Evaluating the tradeoff

This section presents the quantitative evaluation of the different models described above with the actual XMT machine. The reference application is based on an implementation of the Aho-Corasick string matching algorithm, which we will describe in detail in the following. So far, it is important only to know that we experienced on the real machine a lot of contention with this application. The considered application has a reference English dictionary of 20K patterns. We used three different inputs, an English text file, a TCP traffic dump and random input. The machine on which the experiments have been run is an 8-core workstation with 2 sockets and a quad-core Intel Xeon X5560 processor per socket, running at 2.8 GHz. The machine is equipped with 24 GB of RAM and supports the HyperThreading technology. The reference XMT machine is a 128-processor configuration that employs a 4x4x8 3D toroidal network topology. The results are presented with respect to simulation accuracy and simulation speed, in order to evaluate the trade-off in modeling detail.

Left part of Figures 4.11, 4.12 and 4.13 compare the execution times of the XMT against application execution when employing the different network models. In this case, a static network simulation with 8 processors underestimates the overall execution time of 50%. It is important to recall that, on such kind of DSM machines, even when low processor counts are used to run the application kernel, the memory is still physically distributed across all the network. Therefore, the distribution of the destination memories still spans across the entire 128-nodes network.

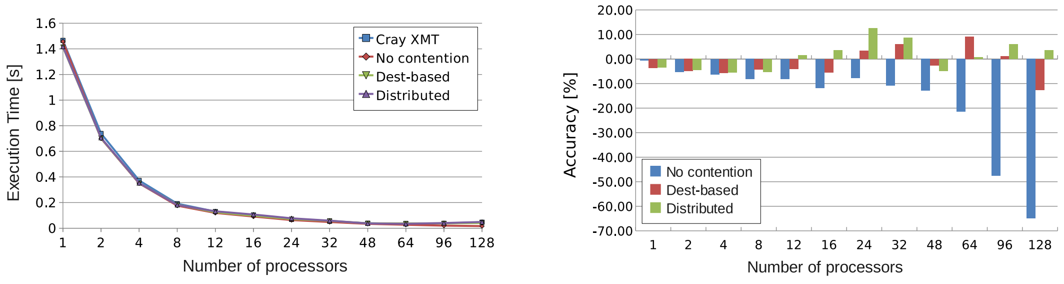


Figure 4.11: English text input

Right part of Figures 4.11, 4.12 and 4.13 plot the accuracy numbers with respect to execution on the reference Cray XMT machine. Accuracy measurements are expressed as signed numbers. A positive number indicates that simulation is overestimating the execution time with respect to the real machine.

Figure 4.14 plots the simulation speed results for the different network models. As expected, the comparison results in decreasing speeds as the level of detail of the simulation

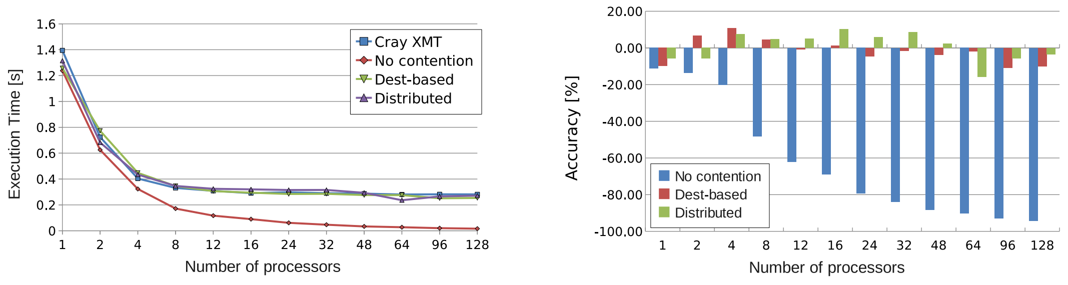


Figure 4.12: TCP dump input

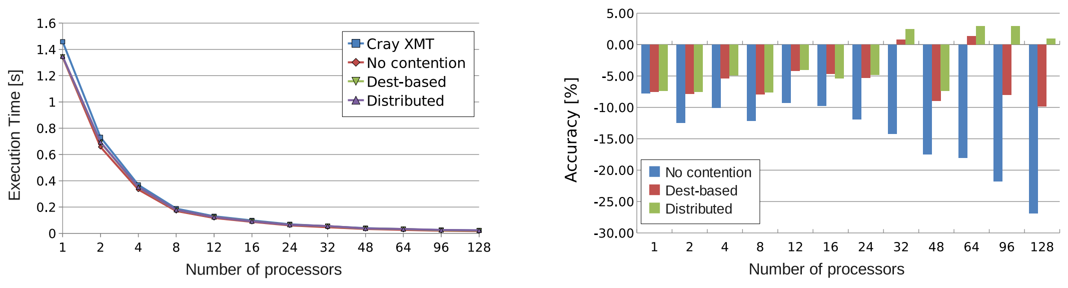


Figure 4.13: Random input

increases. This is more evident as long as the simulator POSIX Pthreads are not context-switched to share host cores, thus up to 8 simulated processors for our experimental configuration. Beyond that point, thread switching overhead becomes dominant and the speed difference is less noticeable. The static latency prediction mechanism obviously achieves the highest speed, about 500 KCyclessec on this specific machine. When inserting the dynamic contention estimation, the destination-based model introduces the shared counter logic and further inter-thread synchronization overhead that result in a 20% reduction of the overall simulation speed. In turn, when contention estimation is improved to a finer granularity, like in the distributed model, another 15% of the simulation speed is lost.

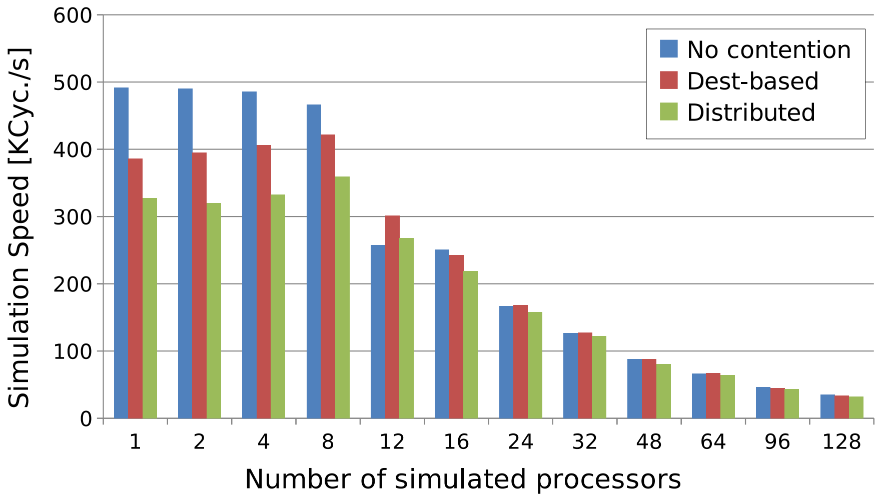


Figure 4.14: Simulation speed results.

In terms of number of simulated processors, the results for all the different models are coherent with the parallelization scheme of the full-system simulator. By mapping each simulated processor to a single *pthread* and synchronizing them with a loose barrier scheme, simulation speed is essentially kept constant up to the number of available physical cores (i.e. 8 in this experimental configuration). Over the number of available physical cores, simulation speed starts to decrease due to the overhead of thread context switching. If specific multi-threading technologies are available on the machine that runs the simulator (e.g., Intel HyperThreading technology) this effect might be mitigated up to the number of available logical cores. Nevertheless, thread synchronization and memory bandwidth of the host machine still have an impact on simulation speed.

Accuracy results have shown that the static latency estimation mechanism is not viable for specific inputs or high processor counts. Of the two different dynamic latency assignment models that have been developed, the most convenient solution to the speed-accuracy trade-off seems to be the destination-based contention model. In fact, in terms of accuracy, the results are similar whereas, in terms of simulation speed, the distributed contention model is 15% slower.

Although it might seem strange that the distributed contention estimation obtains the same accuracy as the destination-based model, the actual reasons for these results are inherent to the whole Cray XMT simulator organization. In fact, the simulator models the actual memory scrambling guaranteeing that the *overall* data distribution across the network has the same distribution (i.e. uniform), but it does not implement the same exact scrambling mechanism implemented in the real hardware. Moreover, the mapping of the logical processors to the actual processors of the network is performed by the operating system, and might change noticeably from run to run. In particular, fine-grained synchronization and the related timing are highly dependent on the mapping of some specific variables. All these factors make it unfeasible to achieve accuracies finer than 5-10%. This is especially true for applications that execute in times of the order of seconds, like the ones we considered for this work. The mentioned remarks are generalizable for all the multi-threaded machines that make use of memory scrambling and fine-grained synchronization mechanisms to operate on a large physically distributed shared address space.

4.6.4 Cache modeling

The memory controller of the ThreadStorm processor hosts a 128 KB 4-way associative data cache. This is the only data cache present in the architecture and it does not need any coherency protocol, since it only caches the data from its own memory partition. The cache acts with a Least Recently Used (LRU) policy, and has a line size of 8 words (64 bytes), corresponding to the granularity of the XMT scrambling. Consequently, when there is a miss in the cache, the whole line is pulled from memory and saved in the cache, even if only the requested value is returned to the memory operation. Subsequent local or remote accesses to one of the words in the line will then hit the cache, reducing the latency of the memory operation.

The rationale behind this cache is not to exploit locality, usually not present in irregular applications, but to enable protocol translation between Threadstorm and HyperTransport. However, it has a noticeable impact on the overall accuracy. To account for the effects of this cache in our variable latency model, we directly modeled it in the memory controller. When the delay of the memory operation is determined (regardless of the fact that contention modeling is enabled or not), we know which is the location of the requested data in the simulated memory space by just checking the address and following the scrambling algorithm. By knowing the simulated memory location, we calculate the number of network hops that the memory operation should traverse to reach the destination memory controller. We then check the cache pertaining to the destination memory controller, to see if the data is already present. If so, only the cache hit latency is added to the network latency and the memory controller - NIC interface latency. If the data is not present, instead, we add the latency of the cache miss, also including the actual memory access, and consequently cache the whole memory line, following the LRU replacement algorithm. Note that checking and eventual replacement of cache lines is performed by the thread executing the simulated processor model, eliminating the need of communication among the threads.

4.6.5 Synchronization and further optimizations

Since the simulator threads running the ThreadStorm models do not communicate, a synchronization mechanism is required in case the same location of the memory state is concurrently accessed by more than one simulated processor. Our design choice was to implement a vector of mutexes, assigning multiple memory state locations to the same lock. Empirically, we set the size of this vector to 1024 elements. Implementing a lock for each memory location would, in fact, be overly expensive in terms of memory occupation, and hardly useful. A memory operation updates the memory state only when the delay calculated through the latency model has passed, so it is very unlikely that more than 1024 different locations are updated exactly at the same time from different simulated processors. On the other hand, we still guarantee that the memory state seen by the various processors when the same memory location is accessed intensively remains coherent.

As previously explained, cache updates are performed by the threads that request a memory operation. Consequently, in case of a hit, it is necessary to lock the cache line to update the data, while in case of a miss it is required to lock the line chosen for replacement. For synchronizing these operations we defined, for each data cache, a vector of locks whose size corresponds to the number of cache sets. Since the cache is 4-way associative, there are 512 sets, with each set containing 4 lines, and thus we lock a full set. As cache sets are frequently accessed, this choice guarantees high performance by only acquiring a lock if really necessary for set updates. When simulating a machine with 128 Threadstorms, the vector of locks for the caches has size 65,536.

We also optimized the M-Unit model to reduce the overheads when managing memory operations. In the ThreadStorm, when lookahead is enabled, each stream launches memory operations in order (as in a circular queue) and retires them out-of-order. Thus, a standard implementation of the load store/queues with a vector would require continuous checking to find the started and the completed memory operations. Each of the 128 streams of a ThreadStorm can launch up to 8 memory operations, so every clock tick the check would be performed 1024 times. Instead, we count and separately save the indexes of those operations effectively started. The variable latency model is then applied only on them. Memory operations complete when the latency calculated through the model has passed. When this happens, a flag is set. Retirement (i.e., register write back, exception generation) is performed only at those clock ticks in which the flag is active.

4.7 Experimental evaluation

We performed several experiments to assess the accuracy and the performance of the simulator with respect to the real Cray XMT supercomputer. In this section we initially describe the large SMP simulation hosts and the applications used as benchmarks, discussing their main features. Then, we show how our accuracy achieved on all the benchmarks. Finally, we discuss the performance scaling of the simulator.

4.7.1 Experimental setup

Many of the considered applications are irregular. Most of them have been specifically optimized for the Cray XMT and benefit from its architectural features. In detail, we chose the following benchmarks:

- a 1000-by-1000 *matrix multiplication*. Although matrix multiplication is not irregular, we included it to validate the simulator performance and accuracy only with the synchronization introduced by the runtime for workload distribution.
- a multithreaded implementation of the *Breadth-First Search* (BFS) algorithm for the Cray MTA-2 [10], using as input a 20,000-vertices-200-neighbors graph.
- an implementation for the Cray XMT of the *Aho-Corasick* string matching algorithm [52], using a dictionary of 20,000 English words and three different input streams: the King James Bible (KJV), a TCP/IP dump and a data set with entries generated at random from the ASCII alphabet with an uniform distribution (RND). We execute a standard and an optimized (to reduce hot-spotting) version of the algorithm.
- 5 kernels from *GraphCT*, a Graph Characterization Toolkit designed for the analysis of graphs representing social network data [26]. The benchmark is also able to generate artificial datasets, configurable in size and shape. In our experiments, we used the *Degree Distribution*, the *Connected Components*, the *Modularity*, the *Conductance* and the *Clustering Coefficient* kernels on artificial graphs with 2^{19} vertices.
- an optimized *triadic analysis* applications for the Cray XMT [18], typically used for finding interesting aspects of a social network. We used two different datasets: a medium sized one, The Edinburgh Associative Thesaurus from the Pajek datasets [12] and a large one, representing the graph of the related clips from Youtube.

All the applications, after being compiled and statically linked with the Cray XMT toolchain 6.4, have been executed unmodified on the simulator. We conducted the simulations on two different SMP host machines. The first is a dual Intel Xeon 5650 (Westmere) machine, where each processor includes 6 cores with HyperThreading (2 threads per core, for a total of 12 cores and 24 threads), 12 MB of L3 cache and runs at 2.66 GHz. The machine has 24 GB of DDR3-1066 memory. The second system, instead, presents 4 AMD Opteron 6176 SE (Magny Cours) processors and 256 GB of DDR3-1066 memory. Each Opteron features 12 cores, organized in two dies of 6 cores with 6 MB of L3 cache and runs at 2.3 GHz. Thus, the system has a total of 48 cores. In the rest of the section, we refer to the Intel machine as the *12-core Xeon* and to the AMD machine as the *48-core Opteron*. Both the machines use Redhat Enterprise Linux 5.5 with kernel version 2.6.18-194.26.1.el5. The simulator has been compiled with GCC 4.4 at optimization level O3, enabling the specific flags for each architecture. The results of the simulator have been compared to execution times of the applications on the real Cray XMT supercomputer with 128 ThreadStorms.

4.7.2 Simulator accuracy

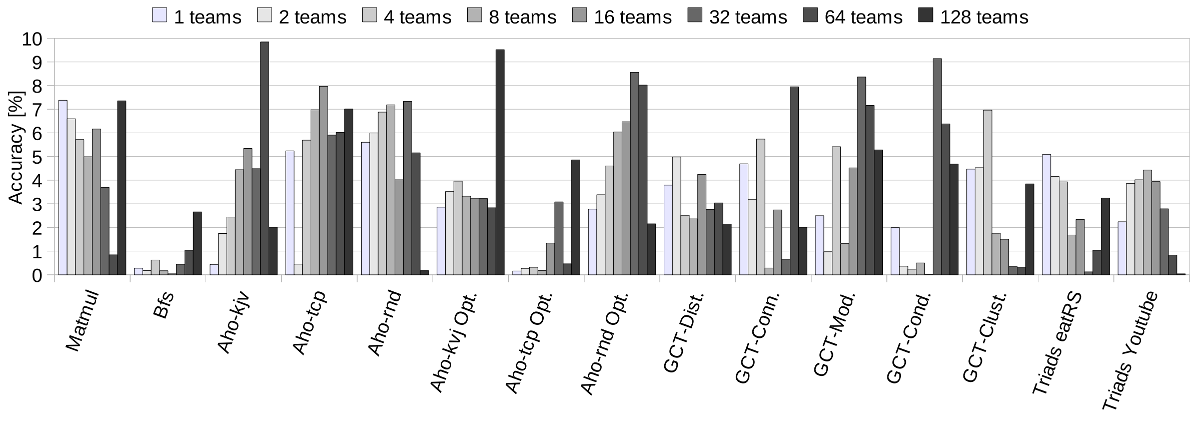


Figure 4.15: Simulator accuracy for the different benchmarks, with increasing number of teams.

Figure 4.15 reports the accuracy data in *Simulation* mode while increasing the number of teams up to 128 for all the benchmarks. For each applications, we performed 10 runs both on the actual hardware and on the simulator, averaging the execution times and determining the percent relative accuracy. The simulated execution times do not diverge from the real XMT ones by more than 10%, for all the applications. The majority of the results is under the 5% range, also including long benchmarks, such as the Triadic analysis on the Youtube data set (around 180 seconds on a single ThreadStorm), and benchmarks with large memory footprints, such as the GCT kernels. BFS, which is a synchronization intensive benchmark, remains under 1% up to 64 teams and under 3% for 128 teams.

Figure 4.15 also highlights a non predictable fluctuation of the simulator accuracy.

This effect is due to the memory scrambler of the XMT and to the team allocation performed by the runtime. In fact, the simulator follows the uniform distribution of the XMT, but the actual hardware scrambling mechanism is proprietary. Furthermore, the specific ThreadStorms chosen by the runtime on the XMT may be different from run to run, possibly not coinciding with those selected on the simulator. This behavior alone can already cause performance variations on the real machine around 10%. Both scrambling and team allocation determine different network hop distances for the variables involved in stream and team management (spawning, synchronization, joining), thus influencing the execution times. Errors under 10% for a supercomputing machine also are excellent with respect to the current state of the art.

4.7.3 Simulator performance

Figures 4.16 and 4.17 show the slowdown incurred by the simulator in its different accuracy levels with respect to our reference XMT machine, while scaling the number of teams of the

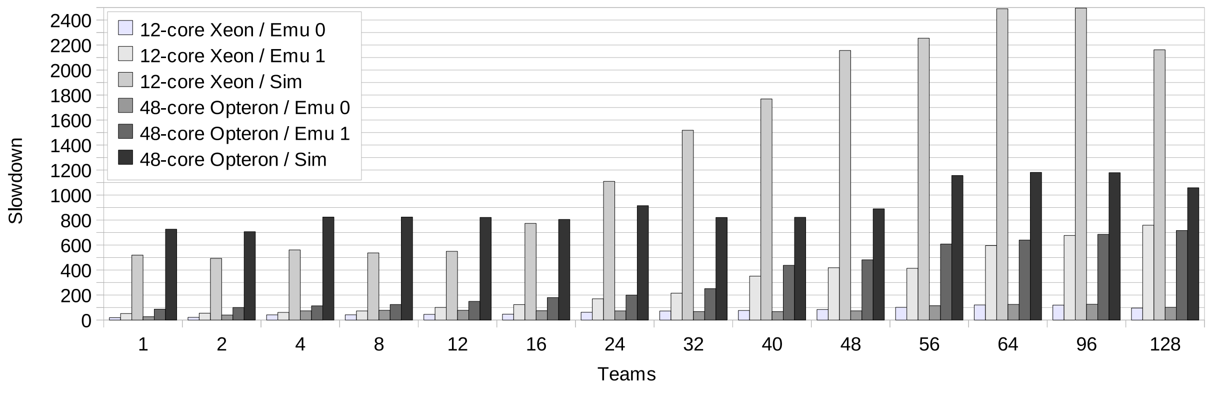


Figure 4.16: Slowdown of the standard Aho-Corasick benchmark on the KJV data set with increasing number of teams

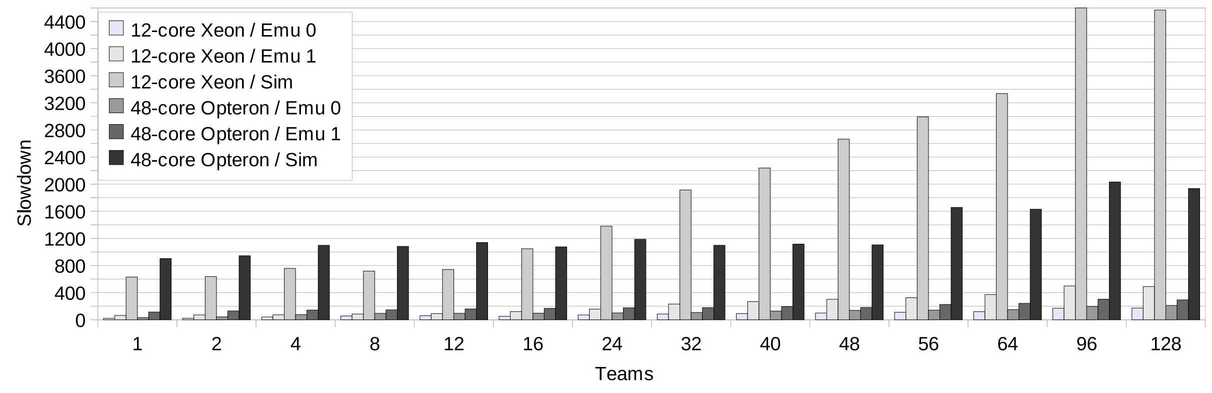


Figure 4.17: Slowdown of the matrix multiplication benchmark with increasing number of teams

target architecture up to 128. By discussing the slowdowns, calculated by dividing the simulation or emulation times by the execution times of the real machines, we can fairly compare the various operating modes. We consider the slowdowns for the Aho-Corasick algorithm and for the matrix multiplication, because these represent two opposites in our benchmark set. In fact, matrix multiplication only involves the static synchronization of the runtime for spawning and joining threads. Aho-Corasick, instead, presents larger amount of synchronization at the application level. This in turn causes higher synchronization overheads in the runtime, for dynamic work scheduling among simulated streams, and in the simulator, for contented locks on the shared memory state. We can see that, in *Simulation*, the infrastructure maintains a constant slowdown with respect to the XMT, up to the number of available host cores.

With Aho-Corasick (Figure 4.16) the 12-core Xeon maintains a slowdown around 500 times up to 12 teams, while the 48-core Opteron presents a slowdown between 700 and 900 times up to 48 teams. Although the 12-core Xeon shows a better single threaded performance, due to the higher clock rate and the improved architecture, the 48-core Opteron shows a significant performance advantage, resulting in many cases more than twice as fast, when simulating higher number of teams. The reason is, obviously, that the host cores are oversubscribed with fewer simulation threads. Since all the simulated teams have a substantial workload, there are not benefits from HyperThreading on the 12-core Xeon: the slowdown goes from 500 to 800 times when raising the number of teams to 16, and doubles with 24. In *Simulation*, these slowdowns correspond to a peak performance around 1,000 KCycles/s for the 12-core Xeon and around 600 KCycles/s for the 48-core Opteron, when mapping a single simulation thread to each host core. The 48-core Opteron reaches 250 KCycles with 128 teams. As a reference, the ThreadStorm runs at 500 MHz. *Emulation 0* slowdowns start from 20 times for the 12-core Xeon and 25 times for the 48-core Opteron with 1 team, and remain under 100 times with 128 teams for both. In this mode, all the latencies are removed, the contention is not considered and only a single stream per team is used. In the Aho-Corasick benchmark, the synchronization overheads become prevalent and the multi-processor/multi-die configuration of the 48-core Opteron has a significant impact. In fact, lock and barrier constructs are up to twice as slower on the 48-core Opteron, and this is reflected in performance similar to the 12-core Xeon for large numbers of teams. The behavior is confirmed when moving to *Emulation 1*, where stream creation is enabled. Again, the performance of the host machines remain similar with many teams, because the synchronization overhead has a major impact. Since each team is now emulating multiple streams, and each stream is performing synchronization operations, the overhead is even higher. So, the performance of *Emulation 1*, which is much better than full *Simulation* for few teams, gets progressively worse. Nevertheless, it still remains useful for warm-up runs, when compared to full *Simulation*.

The matrix multiplication benchmark (Figure 4.17) shows the same, stable behavior in *Simulation*, when there are fewer simulation threads than host cores, for both the machines. Since the workload distribution among the teams is more balanced, the slowdown grows proportionally with the number of teams assigned to a host core. For example, for the 48-core Opteron, we can see that the slowdowns up to 48 teams remain between 1000 and 1200 times, the slowdown for 56 and 64 teams aligns at 1600 times and the slowdowns for 96 and 128 teams are on the 2000 times line. However, the *Simulation* slowdown of matrix multiplication with respect to Aho-corasick is higher, due to the higher computational intensity. In terms of KCycles/s, in *Simulation* our infrastructure reaches 800 KCycles/s on the 12-core Xeon for low number of teams, and 200 KCycles/s on 48-core Opteron system for 128 teams. *Emulation 0* slowdowns range from 21 to 172 times for the 12-core Xeon and from 31 to 211 times for the 48-core Opteron. *Emulation 1*, instead, runs from 64 to 500 times slower for the 12-core Xeon and from 113 to 300 times slower for the 48-core Opteron. With high numbers of teams, the 12-core Xeon results faster than the 48-core Opteron in *Emulation 0*, but it is slower in Emulation 1. At the opposite of Aho-Corasick, it is the single stream used in *Emulation 0* that increases the synchronization requests of the runtime. In fact, the work queues are static and always present the same number of elements (i.e., all the independent operations resulting from the unrolling of the multiplication loop) at all the accuracy levels. However, since the operations are independent, with multiple teams the work is distributed more evenly to the different simulated teams.

Combined together, the high performance and the good accuracy make our infrastructure a valuable tool for the study of multithreaded architectures. By purposely modifying key elementary components and latencies, such as cache, memory, network hops, load/store queues or thread switching, with parameters from other vendors, roadmaps or by introducing potentially interesting new architectural solutions, our simulator will be able to provide interesting insights on the development of next generation massively multithreaded machines.

Chapter 5

Conclusion

In this thesis, the problem of simulating complex multi-core architectures has been addressed the two outermost fields of the vast computing spectrum, namely multi-core heterogeneous embedded systems and many-processors high performance computing machines. We described the conception, design and development of two simulators that faced the most critical problem in modern simulation techniques, namely the trade-off between simulation speed and accuracy.

The first part of the thesis describe the research activity performed in the field of technology-aware system-level FPGA-based emulation of embedded multi-core architectures. We aimed at developing a framework for rapid power- and energy-aware emulation of such architectures, that would be able to reduce the gap between the different design steps and facilitate the design closure. An FPGA-based framework for the exploration and characterization of MP-SoC architectures has been presented, with particular emphasis on NoC-based systems. The two main points of strength of the proposed framework are high-level automatic hw-sw platform instantiation, integrated with Xilinx proprietary tools for FPGA implementation, and the use of analytic models that, basing on functional information provided by the FPGA emulation, are able to estimate different technology-related parameters of a prospective ASIC implementation, such as power and energy consumption, area occupation, and maximum achievable operating frequency. Moreover, we also look at software-based platform reconfiguration in order to reduce the impact of the FPGA synthesis/implementation process. The problem of NoC topology selection has been addressed as a possible design exploration use-case. The main point of strength of the proposed approach is that, by looking at the runtime software-based reconfiguration capabilities of the hardware platform, several emulation steps could be performed after a unique FPGA synthesis and implementation run. In such a way, we show that different NoC-based interconnection topologies could be emulated on hardware by mapping them via software on a larger worst case topology.

The approach has proved to be orders of magnitude faster than pure software cycle-accurate simulation. Moreover, looking at software runtime reconfiguration, the experimental data have shown that the overhead introduced by the over-provision of hardware resources to the worst case topology that is actually implemented on hardware does not preclude the feasibility of the approach. This is especially true with large FPGA devices that are entering the market in the latest years.

In general, we can say that the presented use cases validate the usefulness of the framework in all the contexts where rapid simulation methodologies are required; in general, it is possible to foresee an employment of the proposed framework as an effective support to quantitative design space exploration or simply as an environment for rapid prototyping of complex multi-core platforms.

In the second part of the thesis, we addressed the problem of effectively simulating high performance machine, with thousands of cores, on top of commodity parallel clusters. To this aim, we presented a parallel simulation infrastructure of the Cray XMT supercomputer optimized for SMP hosts. The Cray XMT is a multithreaded machine specifically designed for the execution of data-intensive parallel irregular applications and, due to its architectural features, it does not easily adapt to currently existing simulation tools. We explained how SMP hosts cope with the characteristics of the target machine. We detailed the various components of the simulation infrastructure: the frontend, the system call support, the processor model and the memory and network model. We discussed the design choices behind each one, made by considering the features the host systems, of the target machine and of its runtime. Various optimization and synchronization strategies allow reaching high simulation speed without reducing the accuracy. The network and memory model takes into account contention with limited performance overhead. The simulator runs unmodified XMT binary code, and supports dynamic accuracy switching at the application level.

On current SMP workstations with up to 48 cores, our infrastructure is able to simulate a set of irregular applications, with large memory footprints, from 500 to 2000 times slower than the real execution times of a 128-processor XMT machine within an accuracy of 10%. While emulating, the slowdowns range from 25 to 200 times. We believe that this tool represents an important step towards the efficient simulation of large scale multithreaded machines, paving the way for the evolution of these architectures.

5.1 Future developments

Regarding the FPGA-based emulation framework, already planned future work includes the extension of the library of components, as well as the consideration of hard routed macros and hardware partial reconfigurability mechanisms to further reduce the overhead introduced in the architectural exploration by the FPGA synthesis and implementation flow. The addition of configurability and extensibility inside the processing unit ISA will be taken into account as well, to enable the easy instantiation of highly-specific heterogeneous platforms. Currently, the framework is employed in two EU-funded projects [3] [2] to explore complex ASIP-based multicore architectures. We envision additional dissemination of the framework and employment for the exploration of different industrial-strength system architectures.

On the high performance computing side, the simulator is going to be released soon under the GPL open-source license. We believe its spreading will be huge within the Cray XMT user supercomputing community. Currently, we are working on the simulation of the interconnection subsystem, to make it more machine-independent. We foresee both minimal modifications to support the next generation products of the XMT family, the Cray XMT2 and XMT3. At the same time, we are working on some architectural exploration to investigate on multi-core solutions for the single node. We intend to simulate the performance of multi-core chips that feature several ThreadStorm processors and memory controllers, both to maximize the per-core IPC and to match the per chip injection rate with the network sustainable bandwidth.

Bibliography

[1] The AMD Fusion family of APUs, 2010. http://fusion.amd.com. [cited at p. 2]

[2] Architecture Synthesis and Application Mapping (ASAM), 2010. http://eolab.diee.unica.it/research/projects/asam-automatic-architecture-synthesis-and-applicatio. [cited at p. 94]

[3] MADNESS - Methods for predictAble Design of heterogeneous Embedded Systems with adaptivity and reliability Support, 2010. http://www.madnessproject.org/. [cited at p. 94]

[4] The Power architecture Organization, 2010. http://www.power.org/. [cited at p. 9]

[5] Dennis Abts and Deborah Weisser. Age-Based Packet Arbitration in Large-Radix k-ary n-cubes. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 5:1–5:11, New York, NY, USA, 2007. ACM. [cited at p. 80, 82, 83, 85]

[6] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo. Contrasting a NoC and a Traditional Interconnect Fabric with Layout Awareness. In *Proceedings of the DATE '06 Conference*, Munich, Germany, 2006. [cited at p. 35]

[7] Krste Asanovic and et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Univ. of California, Berkeley. [cited at p. 1, 9]

[8] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, and Jose M. Mendias. A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 618–623, New York, NY, USA, 2006. ACM. [cited at p. 16]

[9] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002. [cited at p. 12]

[10] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 89]

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991. [cited at p. 11]

[12] Vladimir Batagelj and Andrej Mrvar. Pajek datasets, 2006. [cited at p. 89]

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. [cited at p. 6]

[14] D. Bertozzi and L. Benini. X-pipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31, 2004. [cited at p. 35]

[15] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical report, Cambridge, MA, USA, 1991. [cited at p. 6]

[16] R. Brightwell, K.T. Predretti, K.D. Underwood, and T. Hudson. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *Micro, IEEE*, 26(3):41 –57, 2006. [cited at p. 68, 80]

[17] D.C. Burger and D.A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 22 –31, apr. 1995. [cited at p. 18, 72, 82]

[18] George Chin, Andres Marquez, Sutanay Choudhury, and Kristyn Maschhoff. Implementing and evaluating multithreaded triad census algorithms on the cray xmt. In *IPDPS '09: the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–9, 2009. [cited at p. 89]

[19] Nilesh Choudhury, Yogesh Mehta, Terry L. Wilmarth, Eric J. Bohm, and Laxmikant V. Kalé. Scaling an Optimistic Parallel Simulation of Large-Scale Interconnection Networks. In *Proceedings of the 37th conference on Winter simulation*, WSC '05, pages 591–600. Winter Simulation Conference, 2005. [cited at p. 23, 24]

[20] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2:15:1–15:32, June 2009. [cited at p. 13]

[21] The Convey Computer Corporation. The Convey Computer, 2010. http://www.conveycomputer.com. [cited at p. 2]

[22] Open Cores. The OpenCores initiative, 2010. http://www.opencores.org/. [cited at p. 9]

[23] Donglai Dai and D.K. Panda. How Much Does Network Contention Affect Distributed Shared Memory Performance? In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 454 –461, August 1997. [cited at p. 82]

[24] P.G. Del Valle, D. Atienza, I. Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, and G. De Micheli. Architectural Exploration of MPSoC Designs Based on an FPGA Emulation Framework. In *Proceedings of XXI Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 12–18, 2006. [cited at p. 16, 51]

[25] AMD Developer Central. AMD SimNow Simulator, 2010. http://developer.amd.com/simnow.aspx. [cited at p. 20]

[26] David Ediger, Karl Jiang, Jason Riedy, David A. Bader, Courtney Corley, Rob Farber, and William N. Reynolds. Massive social network analysis: Mining Twitter for social good. In *ICPP '10: Proceedings to appear*, 2010. [cited at p. 89]

[27] E.S. Chung et al. PROToFLEX: FPGA-accelerated hybrid functional simulator. pages 1–6, March 2007. [cited at p. 13]

[28] A. Falcon, P. Faraboschi, and D. Ortega. Combining simulation and virtualization through dynamic sampling. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:72–83, 2007. [cited at p. 20]

[29] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM. [cited at p. 66]

[30] Alessandro Forin, Behnam Neekzad, and Nathaniel L. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research. [cited at p. 14]

[31] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. XpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, 2004. IEEE Computer Society. [cited at p. 41]

[32] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. [cited at p. 11]

[33] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz. Ramp blue: a message-passing manycore system in fpgas. In *In 2007 International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 27–29, 2007. [cited at p. 11]

[34] 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. [cited at p. 6, 13]

[35] Joseph B. Manzano, Andres Marquez, and Guang G. Gao. MODA: A memory centric performance analysis tool. In *11th LCI International Conference on High-Performance Clustered Computing*, 2010. [cited at p. 72]

[36] P. Meloni, I. Loi, F. Angiolini, S. Carta, M. Barbaro, L. Raffo, and L. Benini. Area and Power Modeling for Networks-on-Chip with Layout Awareness. *VLSI-Design Journal, Hindawi Publications*, (ID 50285), 2007. [cited at p. 43, 44]

[37] Hans W. Meuer. The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience. *Informatik-Spektrum*, 31(3):203–222, June 2008. [cited at p. 2]

[38] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. pages 1 –12, jan. 2010. [cited at p. 6, 21]

[39] Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, 2009. [cited at p. 6, 20]

[40] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8:12–20, 2000. [cited at p. 6, 18]

[41] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *PPOPP '95: the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–79, 1995. [cited at p. 65]

[42] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM. [cited at p. 51]

[43] (OCP-IP). Open Core Protocol Standard, 2003. http://www.ocpip.org/home. [cited at p. 10, 30, 33]

[44] J. Pal Singh, S. C. Woo, M. Ohara, E. Torrie, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proceedings of the International Symposium on Computer Architecture*, 1995. [cited at p. 46, 61]

[45] Ishwar Parulkar, Alan Wood, Sun Microsystems, James C. Hoe, Babak Falsafi, Sarita V. Adve, and Josep Torrellas. OpenSPARC: An open platform for hardware reliability experimentation, 2007. [cited at p. 9]

[46] D.A. Patterson. Ramp: research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:1, 2006. [cited at p. 10]

[47] J. Renau, B. Fraguela, J. Tuck, M. Prvulovic W. Liu, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. http://sesc.sourceforge.net, January 2005. [cited at p. 6]

[48] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):34 –43, 1995. [cited at p. 6]

[49] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998. [cited at p. 40]

[50] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 463–468, New York, NY, USA, 2010. ACM. [cited at p. 11]

[51] K.D. Underwood and K.S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Field-Programmable Custom Computing Machines - FCCM 2004. 12th Annual IEEE Symposium on*, pages 219–228. [cited at p. 9]

[52] Oreste Villa, Daniel Chavarria-Miranda, and Kristyn Maschhoff. Input-independent, scalable and fast string matching on the Cray XMT. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. [cited at p. 89]

[53] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic;. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27:46–57, 2007. [cited at p. 10]

[54] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A Practical FPGA-based Framework for Novel CMP Research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM. [cited at p. 11]

[55] Xilinx. MicroBlaze Processor Reference Guide UG081.(v 9.0), 2010. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf. [cited at p. 34]

[56] Xilinx. PowerPc Processor Reference Guide UG011.(v 1.3), 2010. http://www.xilinx.com/support/documentation/user_guides/ug011.pdf. [cited at p. 34]

[57] Mohammed J. Zaki and Ching J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. [cited at p. 23]

[58] G. Zheng, Gunavardhan Kakulapati, and L.V. Kale. BigSim: a Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 78, apr. 2004. [cited at p. 23]

List of Publications Related to the Thesis

Journal papers

- P. Meloni, S. Secchi, L. Raffo - *An FPGA-based Framework for Technology-Aware Prototyping of Multi-Core Embedded Architectures.* IEEE Embedded Systems Letters, 2010. (Relation to Chapter 3)
- O. Villa, A. Tumeo, S. Secchi, J.B. Manzano - *Fast and Accurate Simulation of the Cray XMT Multithreaded Supercomputer.* IEEE Transactions on Computers, 2011. Conditionally accepted. (Relation to Chapter 4)

Conference papers

- S. Secchi, P. Meloni, L. Raffo - *Exploiting FPGAs for technology-aware system-level evaluation of multi-core architectures.* ISPASS - IEEE International Symposium on Performance Analysis of Systems and Software, White Plains, NY, 2010. (Relation to Chapter 3)
- P. Meloni, S.Secchi, L. Raffo - *Technology-Aware Prototyping of Multi-Core Architectures: an FPGA-based framework.* Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO´2010), held in conjunction with the HIPEAC 2010 conference, Pisa, Italy. (Relation to Chapter 3)
- P. Meloni, S. Secchi, L. Raffo - *Enabling fast Network-on-Chip topology selection: an FPGA-based runtime reconfigurable prototyper.* 18th IEEE/IFIP International Conference on VLSI and Systems on- Chip, VLSI-SoC 2010, Madrid, Spain. (Relation to Chapter 3)
- S. Secchi, A. Tumeo, O. Villa - *Contention Modeling for Multithreaded Distributed Shared Memory Machines: the Cray XMT.* CCGRID 2011 - The 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. Newport Beach, CA, USA. To appear. (Relation to Chapter 4)

Posters with published proceedings

- S. Secchi, P. Meloni, L. Raffo - *An FPGA Research Environment for Rapid MPSoC Exploration.* Advanced Computer Architecture and Compilation for Embedded Systems (ACACES '09). (Relation to Chapter 3)

List of publications unrelated to the thesis

Conference papers

- F. Palumbo, D. Pani, L. Raffo, S. Secchi - *A surface tension and coalescence model for dynamic distributed resources allocation in Massively Parallel Processor on-Chip.* KRASNOGOR N. et al. (Eds), Proceedings of the Nature Inspired Cooperative Strategies for Optimization conference (NICSO 2007), Springer-Verlag Heidelberg.
- F. Palumbo, S. Secchi, D. Pani, L. Raffo - *A novel non-exclusive dual-mode architecture for MPSoCs-oriented network on chip designs.* Proc. SAMOS 2008, International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece.
- S. Secchi, F. Palumbo, D. Pani, L. Raffo - *A network on chip architecture for heterogeneous traffic support with non-esclusive dual-mode switching.* 11th EUROMICRO Conference on Digital System Design (DSD 2008), Parma, Italy.
- D. Pani, S. Secchi, L. Raffo - *Self organization on a swarm computing fabric: a new way to look at fault tolerance.* ACM International Conference on Computing Frontiers 2010. Bertinoro, Italy.
- A. Tumeo, S. Secchi, O. Villa - *Experiences with string matching on the Fermi architecture.* ARCS 2011 - Architecture of Computing Systems, Como, Italy. To appear.

Posters with published proceedings

- F. Palumbo, S. Secchi, D. Pani, L. Raffo - *Non-Exclusive Dual-mode Approach for NoC Designs.* Advanced Computer Architecture and Compilation for Embedded Systems (ACACES ’08)