Raising the Level of Abstraction: Simulation of Large Chip Multiprocessors Running Multithreaded Applications

Alejandro Rico Carro

Advisors:
Alejandro Ramírez Bellido
Mateo Valero Cortés

Submitted to the Departament d'Arquitectura de Computadors
for the degree of

Doctor of Philosophy in Computer Architecture

at the
Universitat Politècnica de Catalunya · BarcelonaTech

September 2013

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de qualificació de tesi doctoral

Curs acadèmic: 2012/2013

Nom i cognoms
Alejandro Rico Carro

Programa de doctorat
Arquitectura de Computadors

Unitat estructural responsable del programa
Departament d'Arquitectura de Computadors

Resolució del Tribunal

Reunit el Tribunal designat a l'efecte, el doctorand / la doctoranda exposa el tema de la seva tesi doctoral titulada

Raising the Level of Abstraction: Simulation of Large Chip Multiprocessors Running Multithreaded Applications

Acabada la lectura i després de donar resposta a les qüestions formulades pels membres titulars del tribunal, aquest atorga la qualificació:

☐ NO APTE ☐ APROVAT ☐ NOTABLE ☐ EXCEL·LENT

(Nom, cognoms i signatura) President/a	(Nom, cognoms i signatura) Secretari/ària

(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal

_____, _____ d'/de _____ de _____

El resultat de l'escrutini dels vots emesos pels membres titulars del tribunal, efectuat per l'Escola de Doctorat, a instància de la Comissió de Doctorat de la UPC, atorga la MENCIÓ CUM LAUDE:

☐ SÍ ☐ NO

(Nom, cognoms i signatura) Presidenta de la Comissió de Doctorat	(Nom, cognoms i signatura) Secretària de la Comissió de Doctorat

Barcelona, _____ d'/de _____ de _____

Abstract

The number of transistors on an integrated circuit keeps doubling every two years. This increasing number of transistors is used to integrate more processing cores on the same chip. However, due to power density and ILP diminishing returns, the single-thread performance of such processing cores does not double every two years, but doubles every three years and a half.

Computer architecture research is mainly driven by simulation. In computer architecture simulators, the complexity of the simulated machine increases with the number of available transistors. The more transistors, the more cores, the more complex is the model. However, the performance of computer architecture simulators depends on the single-thread performance of the host machine and, as we mentioned before, this is not doubling every two years but every three years and a half. This increasing difference between the complexity of the simulated machine and simulation speed is what we call the simulation speed gap.

Because of the simulation speed gap, computer architecture simulators are increasingly slow. The simulation of a reference benchmark may take several weeks or even months. Researchers are concious of this problem and have been proposing techniques to reduce simulation time. These techniques include the use of reduced application input sets, sampled simulation and parallelization.

Another technique to reduce simulation time is raising the level of abstraction of the simulated model. In this thesis we advocate for this approach. First, we decide to use trace-driven simulation because it does not require to provide functional simulation, and thus, allows to raise the level of abstraction beyond the instruction-stream representation.

However, trace-driven simulation has several limitations, the most important being the inability to reproduce the dynamic behavior of multithreaded applications. In this thesis we propose a simulation methodology that employs a trace-driven simulator together with a runtime sytem that allows the proper simulation of multithreaded simulations by reproducing the timing-dependent dynamic behavior at simulation time.

Having this methodology, we evaluate the use of multiple levels of abstraction to reduce simulation time, from a high-speed application-level simulation mode to a detailed instruction-level mode. We provide a comprehensive evaluation of the impact in accuracy and simulation speed of these abstraction levels and also show their applicability and usefulness depending on the target evaluations. We also compare these levels of abstraction with the existing ones in popular computer architecture simulators. Also, we validate the highest abstraction level against a real machine.

One of the interesting levels of abstraction for the simulation of multi-cores is the memory mode. This simulation mode is able to model the performance

of a superscalar out-of-order core using memory-access traces. At this level of abstraction, previous works have used filtered traces that do not include L1 hits, and allow to simulate only L2 misses for single-core simulations. However, simulating multithreaded applications using filtered traces as in previous works has inherent inaccuracies. We propose a technique to reduce such inaccuracies and evaluate the speed-up, applicability, and usefulness of memory-level simulation.

All in all, this thesis contributes to knowledge with techniques for the simulation of chip multiprocessors with hundreds of cores using traces. It states and evaluates the trade-offs of using varying degress of abstraction in terms of accuracy and simulation speed.

Acknowledgements

First of all, I want to thank my advisor, Alex Ramirez, and co-advisor, Mateo Valero. They gave me support, guidance and freedom to make my own decisions through all these years. I have learnt a lot from them not only technically, but also in many other aspects in life. Alex has an impressive ability to turn the smallest idea into great contributions. I also want to thank him for putting up with my pessimism and criticism when I have not been able to see the promise and value of an idea. Mateo's clarity and experience clearly improved the quality of the work and helped put it in a broader context.

I also want to thank my mentor, Pradip Bose, and collaborators, Jeff Derby and Bob Montoye, during my internship at IBM TJ Watson Research Center. I felt their trust from the first day. Pradip was always helpful and willing to teach me even the most obvious things. I also want to thank Roberto and Jose. I also learnt a lot from Roberto, and without their friendship my internship would have not been the same.

I thank my mentor, Chris Adeniyi-Jones, during my internship at ARM. He provided me with all the resources I needed and gave me freedom to carry out the work on my way. I also want to mention Gabor, Andreas, Roxana and James for their help.

I also want to thank the funding bodies that supported this work. This thesis has been supported by the European Social Fund and the Departament d'Universitats, Recerca i Societat de la Informació of the Generalitat de Catalunya under the FI scholarship no. FI-2006-01133; the Spanish Ministry of Education under the FPU scholarship no. AP2005-4245; the FP6 SARC project (contract no. FP6-27648); the FP7 ENCORE project (contract no. FP7-248647); the FP7 Mont-Blanc project (contract no. ICT-FP7-288777); the European Network of Excellence HiPEAC; and the Comisión Interministerial de Ciencia y Tecnología (contract nos. TIN2004-07739-C0201, TIN2007-60625 and TIN2012-34557).

I also want to mention that CellSim and TaskSim are the effort of many people. Main contributors to CellSim were Felipe, David, Toni and Augusto. Main contributors to TaskSim were Felipe, Augusto, Milan, Carlos and Victor.

Vull agrair a Felipe, Miquel, Oriol i Xevi pels bons moments tots aquests anys i les tertúlies de sobretaula. Quiero agradecer especialmente a Felipe. Estuvo ahí desde el primer día y me ha dado su amistad y apoyo durante todo este tiempo. També vull agrair especialment a Miquel. És un gran amic, sempre està a punt per ajudar i va ser un gran recolzament durant els primers mesos a IBM.

I also want to mention the members of the group: Augusto, Milan, Carlos, Karthikeyan, Victor, Puzo, Rajo, Paul, Isaac, Thomas and Ugi; and other col-

leagues at BSC: Enric, Marc,... Being surrounded by friendly and helpful people makes work a happy activity everyday.

También quiero dar las gracias a mis padres Alejandro y Emilia, y a mi hermano Abel. Desde siempre me han apoyado en todo lo que he querido hacer, me han dado los recursos y libertad para hacerlo, y me han aguantado cuando no he estado en mis mejores momentos. Les agradezco también que me regalaran mi primer ordenador pocos meses antes de empezar la universidad y liberarme así de una vida aburrida y darme la oportunidad de trabajar con ordenadores, algo que estoy disfrutando cada día gracias a ellos.

Quiero dar las gracias a Luna por su apoyo y amor estos últimos años. Es un placer estar con una persona que lo da todo, está siempre ahí para levantar el ánimo y te recuerda cada día que es lo más importante en la vida.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

1.1.1 Chip Multiprocessors

A microprocessor is an integrated circuit (chip) that incorporates the central processing unit (CPU) of a von Neumann-style computing system. The first microprocessors appeared in the early 1970s [176, 74]. Since then, the number of transistors on a chip has increased exponentially. This fact was observed by Gordon Moore in 1965, a claim that is popularly known as Moore's law [124]. The rate at which the number of transistors on a chip increased was set to double every year in 1965. In 1975, Moore adjusted his observation to double every two years [123]. This observation has served as an industry driver. Companies' roadmaps target Moore's law transistor count growth rate, which sets targets for research and development divisions and results in the development and introduction of new technology nodes to achieve the required transistor density.

Figure 1.1 shows the transistor count per chip, die size, technology node and transistor density for a set of microprocessor from 1971 to 2012. Data actually confirms Moore's law, and shows a 2x increase in transistor count per chip every two years. This increase was a combination of higher transistor density (+22%/year) and larger die size (+15%/year) until 1992. Since 1992, die size has not increased significantly. Server microprocessors have larger die sizes between 300 and 600 mm^2, while desktop microprocessor die sizes are between 100 and 300 mm^2. However, transistor density has increased 2x every two years (+40%/year) since 1992, thus keeping up with Moore's law transistor count growth rate. This increase in transistor density growth rate was partially thanks to the inclusion of on-chip caches. Caches are more regular than other pipeline logic structures and thus have a higher transistor density.

The availability of more transistors allowed architects to integrate devices, that were so far out of the chip, on the chip (landmarks shown in transistor density chart in Figure 1.1). Examples are the inclusion of floating-point units, caches and memory controllers. Also, it allowed architects to build more complex architectures. First, the bit width of data and address paths was increased to improve performance and to be able to reference a larger address space. The first microprocessors had 4- and 8-bit data paths, with 16- and 32-bit archi-

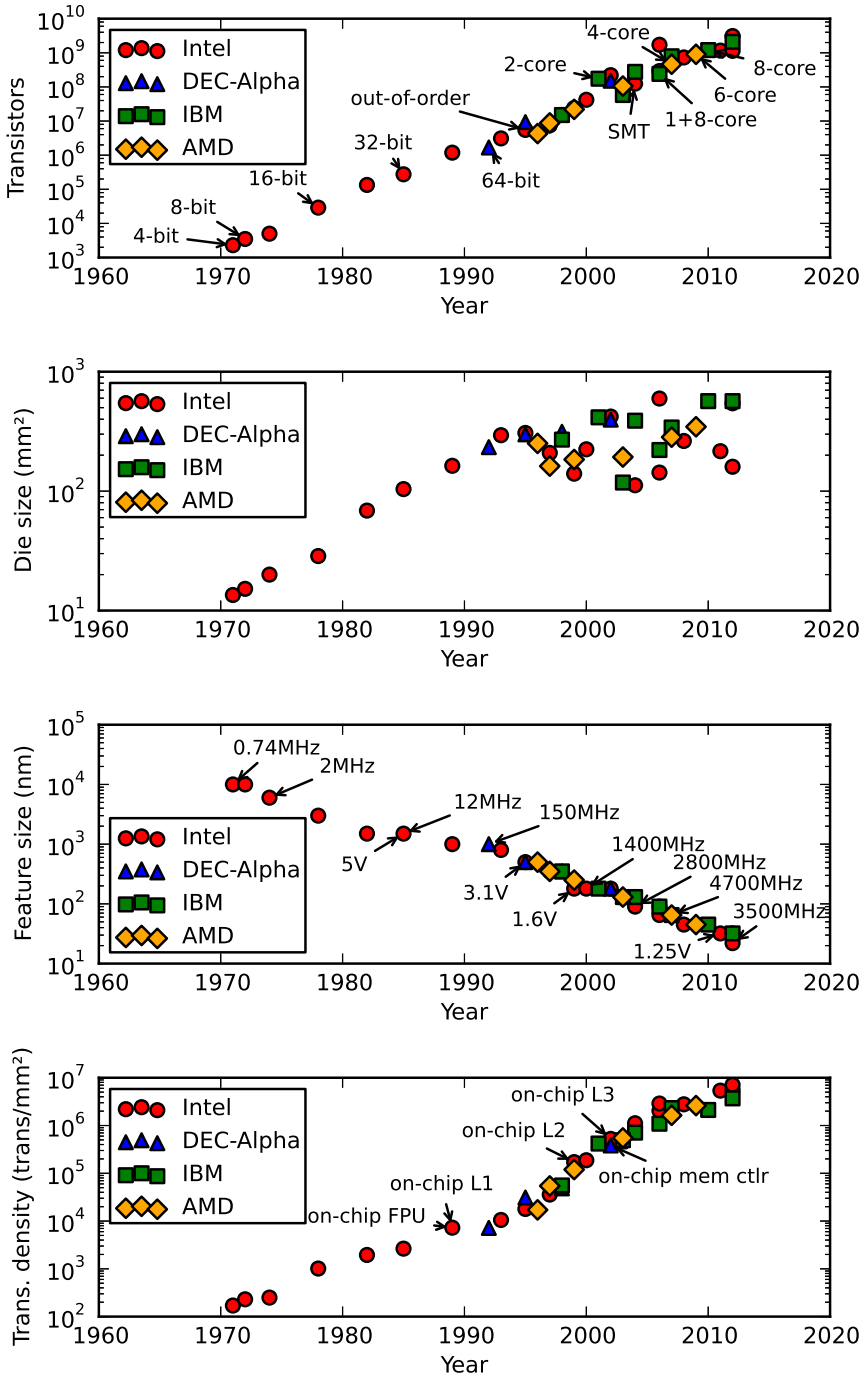


Figure 1.1: Transistor count, die size, technology node and transistor density for a set of microprocessors from 1971 to 2012.

tectures appearing less than fifteen years later. The move from 32 to 64 bits took 18 years for personal computers. However, in the server market, where the system memory footprint is larger, there were 64-bit machines way earlier. An example is the DEC Alpha 21064 introduced in 1992.

Having smaller transistors also allowed to increase frequency, and thus increase performance. This increase in frequency also came together with a reduction in capacitance and voltage (landmarks shown in technology node chart in Figure 1.1). This scale down of voltage for smaller feature sizes was stated in a scaling theory by Robert Dennard et al. in 1974. This scaling theory is referred to as Dennard scaling. The result was that, having more and faster transistors together with a reduction in voltage and capacitance, new design chips provided more performance at the same power. This enabled the introduction of complex architectural techniques to improve performance, such as speculation, out-of-order execution and simultaneous multithreading (SMT). Also, the pipeline structures, such as branch prediction tables, reorder buffer and issue queues, were enlarged to exploit more instruction-level parallelism.

However, Dennard scaling stopped in the early 2000s. Since then, new technology nodes provide more transistors, but voltage does not scale down any more. Voltage is over 1V in desktops, laptops and servers and just below 1V in embedded systems. The result is that power density increases and, as a side effect, frequency cannot be scaled up because affordable heat dissipation solutions cannot dissipate so much heat and the cooling solutions that would are too expensive. This limitation in power density is popularly known as the *power wall*. Moreover, at the same time, computer architects were experiencing diminishing returns in the latest improvements to exploit instruction-level parallelism [172, 143].

Under this scenario, microprocessor design shifted towards chip multiprocessors or *multi-cores*. A multi-core is an integrated circuit including multiple processing cores, in contrast to the single processing core chips had so far. This design is more power efficient [38] and, as a result, more transistors can be used to increase performance by integrating more cores on the chip while preventing power density from skyrocketing.

The first academic paper proposing a multi-core design was presented in 1996 [132] and the first commercially-available multi-core was introduced in 2001. Since then, the trend is to include more cores per chip in every generation, which complicates their interconnection, the management of resources shared among cores and the programming of applications that shifted from focusing on instruction-level parallelism to targeting to exploit thread-level parallelism.

1.1.2 Chip Multiprocessor Simulation

Computer architecture research is mainly based on simulation. This is because the execution of a workload on a complex microprocessor can hardly be modeled analytically, and prototyping every design point is economically unviable.

In the 1980s and early 1990s, microprocessor simulators focused on the modeling of cache behaviour and the pipeline structures in the context of a single processing core per chip. Initially, pipeline models considered in-order execution on a wider or narrower superscalar design. In time, the design complexity kept increasing with the introduction of increasingly complex techniques for out-of-order execution, speculation, branch prediction and simultaneous multi-

threading. On the other hand, the design space of the cache hierarchy included cache size, associativity, latency and writing policies for one or two cache levels.

However, with the introduction of multi-cores, microprocessor simulation turned into the simulation of a parallel machine. The simulation of a parallel architecture is even more challenging than simulating a single core. Several execution streams stress not only core-private components but also shared resources in the architecture. These shared components include shared caches, off-chip memory and the on-chip interconnection among cores. Modeling such sharing and resource contention is already challenging for multiprogrammed workloads. However, for multithreaded applications, it is also sensitive to the level of parallelism, inter-thread data sharing and thread synchronization of the particular application. To faithfully account for these effects, the simulation model requires the inclusion, among others, of cache coherence protocols, cache data placement policies, network arbitration and cache partitioning techniques.

Many of the modeling challenges of multi-core simulation are not new, as they were already present when simulating shared-memory multiprocessor systems with cache coherency across multiple chips. However, there are fundamental differences of having such a parallel system on a single chip compared to multi-chip multiprocessors. Communication latencies are lower thanks to fast on-chip interconnection networks. And memory bandwidth is also lower because an increasing amount of processing cores have to share a limited amount of pins to access off-chip memory.

Due to these fundamental differences, researchers need to assess the applicability of techniques developed for shared-memory multiprocessors in the context of multi-cores. But, at the same time, multi-core designs also open new research opportunities that are unique for multi-cores. As a result, multi-core simulation becomes a fundamental tool for either revisiting existing ideas and explore new ones.

1.1.3 The Simulation Speed Gap

Computer architecture simulation is mainly a single-threaded problem because the fine-grain interaction of the components in the chip limits its parallelization. Most computer architecture simulators are thus sequential and, as a result, simulation speed highly depends on the single-thread performance of the machine were the simulator is running.

While the complexity of microprocessors keeps increasing, single-thread performance is not increasing at the same pace. The efforts to develop or improve techniques to exploit instruction level parallelism have decreased due to diminishing returns and power density issues. This shifted the focus of computer architects to higher aggregated multi-core performance rather than higher single-thread performance.

A popular benchmark suite to measure single-thread performance is SPEC (Standard Performance Evaluation Corporation) CPU. The SPEC CPU benchmark suite is divided in two subsets: integer (CINT) and floating-point (CFP) benchmarks. We focus on the SPEC CINT benchmarks because computer simulation code is mainly integer code. We have gathered the SPECint results for a set of microprocessors from 1991 to 2013 and adjusted them[1] to the 2006

[1] We use the methodology in J. Preshing's blog [137], abiding the SPEC fair use rules [10].

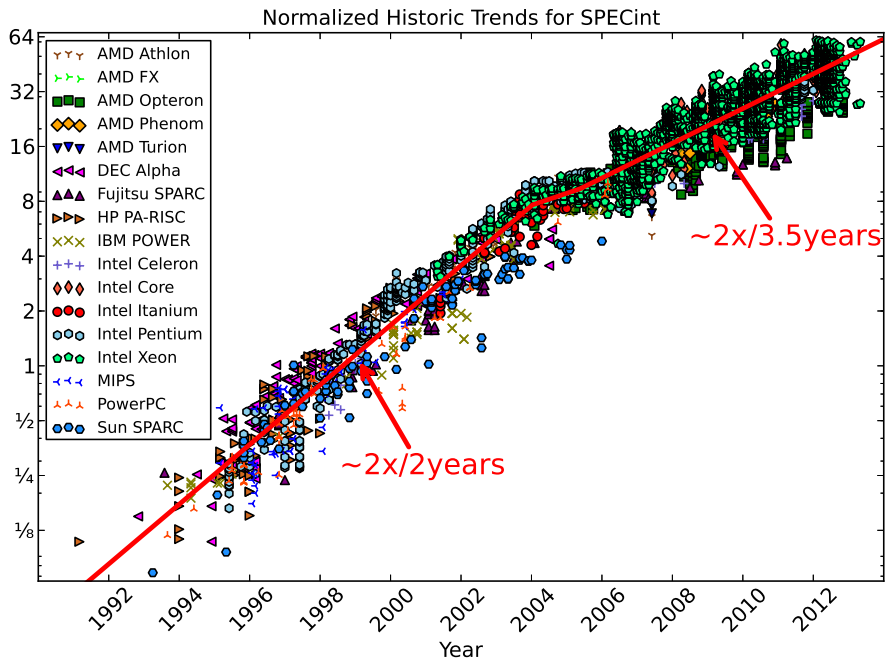


Figure 1.2: Single Thread Performance

standard to show the historical trends for single-thread integer performance. Figure 1.2 shows these results. Until 2004, integer performance doubles approximately every two years. In 2004, multi-cores start becoming mainstream and, in some cases, the first approaches to multi-core design were to integrate simpler cores than the ones in the latest single-core microprocessors. For this reason, from 2004 to 2006, the chart shows some stagnation in performance improvement. Since 2006, single-thread performance improves again, but now it doubles approximately every 3.5 years.

While single-thread performance improvement slows down, the complexity of multi-cores keeps increasing at the same pace thanks to the still-increasing amount of transistors per chip (see Figure 1.1). This translates into two facts. The complexity of the simulation model, which depends on the complexity of the simulated machine, keeps increasing at 2x every 2 years. But, the improvement on simulation speed, that depends on single-thread performance, is slowing down and improves at 2x every 3.5 years. This is what we call the *simulation speed gap*, and it is sketched in Figure 1.3. If the transistor count and single-thread performance trends hold for the next few years, there will be a 10x gap in around year 2020.

This increasing gap leads to increasingly longer simulation runs or to restrict the complexity of the model, for example, restrict the number of simulated cores. This fact is reflected in research works in top conferences where they use multi-core models with the same or less cores than the ones in commercially-available products. While there are commercially-available multi-cores for conventional servers with 16 cores [46], and not so conventional with 64 [29], 79% of the

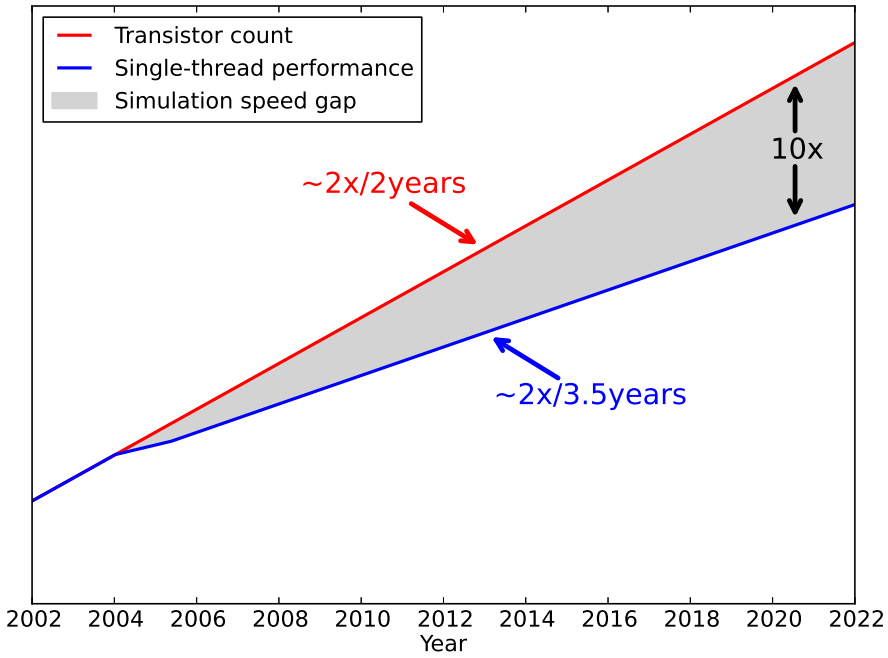


Figure 1.3: Simulation speed gap.

research papers in major computer architecture conferences between 2009 and 2012 simulate at most 16 cores, and only 5% simulate more than 64 cores [2].

1.2 Thesis contributions

The simulation speed gap poses a challenge on computer architects to simulate large multi-core designs. To complete the simulation of large multi-cores in a reasonable time, we require higher simulation speeds. Researchers are concious of this problem, and they have proposed several techniques to reduce simulation time such as statistical simulation, sampling and parallel simulation.

The approach in this thesis is to raise the level of abstraction of the simulation model. This allows to increase simulation speed at the expense of modeling accuracy in order to reduce simulation time. Some reputed researchers advocate for this approach [39, 180] and previous works have applied it in other scenarios such as cluster and network simulation [24, 120].

To implement high levels of abstraction, we advocate for the use of trace-driven simulation. However, one of the most important limitations of trace-driven simulation is precisely its inability to reproduce the dynamic behavior of multithreaded applications, which are absolutely necessary for the evaluation of multi-core systems. This limitation is because the application behavior in multiple threads is statically captured in a trace and does not change for different simulated configurations as it would happen in a real machine.

To overcome this limitation, the first contribution in this thesis is:

[2] Check Section 2.6 for more simulation statistics in computer architecture conferences.

- **A simulation methodology** for simulating multithreaded applications running on multi-cores using trace-driven simulation. In this simulation methodology, we combine the trace-driven simulation of the timing-independent parts of the application, with the execution of timing-dependent operations at simulation time. This way, run-time decisions are made based on the simulated machine and are thus different for different configurations as it would happen in the real machine. This work is supported by the following papers:

  [150] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, M. Valero. Trace-driven Simulation of Multithreaded Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 87–96, Apr. 2011.

  [151] A. Rico, A. Duran, A. Ramirez, M. Valero. Simulating Dynamically-Scheduled Multithreaded Applications Using Traces. *IEEE Micro*. Submitted for publication.

Once we can reliably use trace-driven simulation for multithreaded applications running on multi-cores, we use it to raise the level of abstraction of simulation. However, several questions arise when using higher levels of abstraction regarding what are the right levels of abstraction, their insight, accuracy and simulation speed.

The second contribution in this thesis is:

- **Two fast high-level simulation modes** along with two lower-level ones. These simulation modes at different levels of abstraction are based on our definition of application abstraction levels and target application scalability, accelerator architectures, memory system and pipeline modeling, respectively. We evaluate the insight of these simulation modes, their accuracy and their simulation speed compared to the levels of abstraction used in popular computer architecture simulators. This contribution is supported by the following publication:

  [148] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, M. Valero. On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, 2012.

One of the abstraction levels in our definition targets multi-core memory simulation. This level of abstraction uses an abstract model for processing cores that focus mainly on memory accesses. To speed up memory simulation, previous works use filtered traces that include only L1 cache misses to avoid the cost of simulating L1 hits assuming that these do not imply additional delays in the simulated application. This technique, called *trace stripping* [138], has been successfully used in the past for single-thread scenarios. However, little work has been done to use it for multithreaded applications, in which inherent inaccuracies appear due to cache invalidations. A filtered hit may miss in a multithreaded scenario due to the invalidation of the accessed data from a write in another cache.

The third contribution in this thesis is:

- **A trace generation technique** based on the structure of multithreaded applications that captures in the trace L1 hits that may potentially miss in a multithreaded application execution due to invalidations. With this technique, we cover potential invalidations due to different thread interleavings and different dynamic schedulings. Our evaluation shows that our technique consistently reduces the error of the state-of-the-art technique at the expense of small losses in trace size reduction and simulation speed-up. This work is supported by the following publication:

  [154] A. Rico, A. Ramirez, M. Valero. Trace Filtering of Multithreaded Applications for CMP Memory Simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, pages 134–135, Apr. 2013.

One of the potential inaccuracies of our simulation methodology in our first contribution is that the execution of timing-dependent operations is not exposed to the simulator, and thus cannot be accurately accounted in the application timing modeling.

The fourth contribution in this thesis is:

- **A fast high-level timing model for timing-dependent operations** that are executed at simulation time using our simulation methodology. This timing model is based on the execution of these operations on the host machine to account for different algorithm complexities and runtime application states. Additionally, we also model their contention on accessing data structures shared by multiple threads on the simulated application.

1.3 Timeline

Figure 1.4 shows a timeline of relevant events related to the work in this thesis to put it in context. At the top of the figure, we show the releases of multicore commercial products. The trend followed by the main manufacturers is to increase the number of cores per chip in every new generation of multi-cores. As explained in previous sections, this motivates our work to bridge the simulation-speed gap.

The work in this thesis has contributed to several projects. From January 2006 to December 2009, we contributed to the SARC project with the development of CellSim (see Section 2.1). CellSim served to carry out the work of several partners in the project, leading to multiple publications and PhD dissertations (see Section 7.2). CellSim was also the base for SARCSim: an extension of CellSim including timing models from other SARC partners, such as models of extended vector accelerators and scalable inter-core communication mechanisms.

In the context of the MareIncognito project (from January 2007 to December 2009), CellSim was relevant in the research for the design of next-generation Cell/B.E. microprocessors (see Section 2.1.1). From March 2010 to February 2013 we contributed to the ENCORE project. The simulation methodologies proposed in this thesis were used in ENCORE for the evaluation of explicit management of coherent and non-coherent cache hierarchies. Also, the analysis

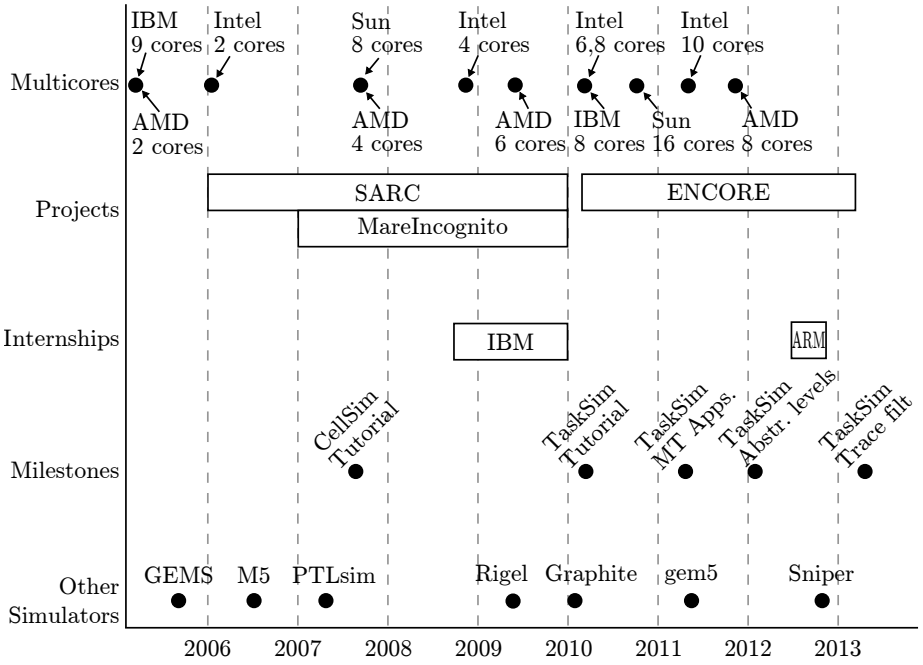


Figure 1.4: Thesis timeline

of the runtime system we carried out for the simulation of the runtime system timing (see Chapter 6) was used in ENCORE to understand the overheads of the several components in the runtime system.

The course of this thesis was interrupted by two industrial internships. The first one at the IBM TJ Watson Research Center (Yorktown Heights, NY, USA) for the development of a vector accelerator took place from October 2008 to December 2009. The second internship was at ARM Ltd. (Cambridge, UK) from August 2012 to November 2012 and focused on the evaluation of the ARM Cortex-A family of microprocessors for high performance computing.

Some milestones worth mentioning as outcomes of the work in this thesis are the following:

- **CellSim Tutorial.** We performed a full-day tutorial on our CellSim simulator (see Section 2.1) in the Parallel Architectures and Compilation Techniques (PACT) conference at Brasov, Romania, on September 2007.
- **TaskSim Tutorial.** We performed a tutorial for the ENCORE project partners on our TaskSim simulator (see Section 2.2) on March 2010.
- **Publication of "Trace-Driven Simulation of Multithreaded Applications".** Publication in Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS) 2011 at Austin, TX, USA, on April 2011.
- **Publication of "On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels".** Publication in the ACM Transactions on Architecture and Compiler Optimization (TACO), Vol. 8, No. 4, Article 36, January 2012.

- **Publication of "Trace Filtering of Multithreaded Applications for CMP Memory Simulation".** Publication in International Symposium on Performance Analysis of Systems and Software (ISPASS) 2013 at Austin, TX, USA, on April 2013.

In the course of this thesis, other groups in the computer architecture community working on simulation of multi-cores published their tools in related conferences and journals. Some of them shown at the bottom of Figure 1.4 are GEMS, M5, PTLsim, Rigel, Graphite, gem5 and Sniper. We cover these works in Section 2.5.

1.4 Thesis Organization

Figure 1.5 illustrates the organization of this document in the several chapters it is composed of.

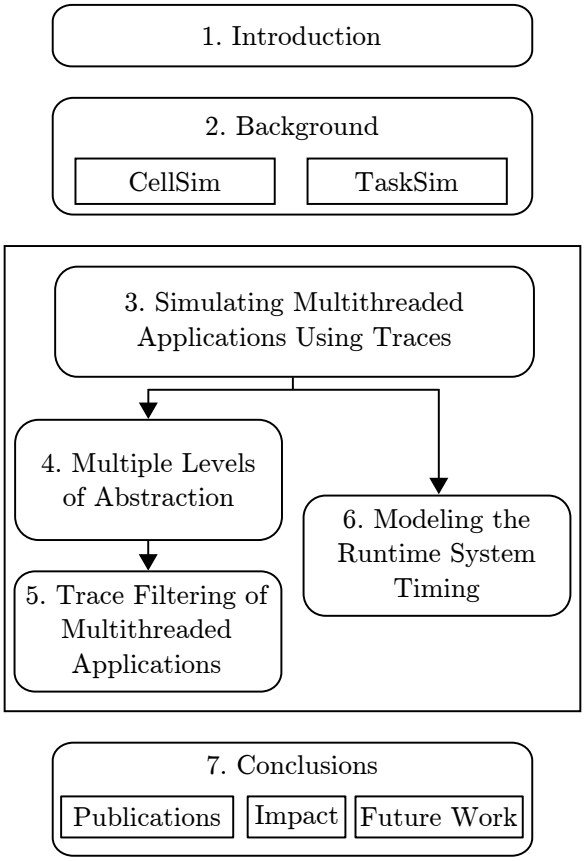


Figure 1.5: Thesis organization.

After this introductory chapter, we devote Chapter 2 to cover related work for this thesis. In this chapter, we include an explanation of CellSim and TaskSim, two simulators to which we contributed to develop in the course of

this thesis. The lessons learned in the development of CellSim motivated the development of TaskSim and the research that led to the contributions in this thesis. The rest of the background in Chapter 2 is general and relevant for the rest of the document.

We cover our contributions in Chapters 3, 4, 5 and 6. We include general related work and state of the art in Chapter 2 and the specific related work and state of the art relevant to each contribution in each one of the corresponding chapters.

Chapter 3 explains and evaluates the first contribution of this thesis: a simulation methodology for simulating multithreaded applications using a trace-driven simulation approach. This simulation methodology enables the works in Chapters 4 and 5 and motives the work in Chapter 6.

Chapter 4 covers our definition of multiple application abstraction levels, the corresponding simulation modes at multiple levels of abstraction, and their evaluation in terms of insight, accuracy and simulation speed. In Chapter 5 we cover the description and evaluation of our trace filtering technique for multi-threaded applications and in Chapter 6 we cover our high-level timing model for timing-dependent operations to be used with the simulation methodology in Chapter 3.

Finally, we conclude this thesis in Chapter 7 with our contributions and associated publications, the impact of our work in other works and some recommendations for future work.

Chapter 2

Background

In this chapter we cover related and relevant work for this thesis. First we introduce CellSim, a simulator we developed for modeling the Cell/B.E. microprocessor. The difficulties and experiences during the development of CellSim motivated us to initiate research with the objective of exploring new simulation techniques to reduce simulation time by raising the level of abstraction.

We also cover related work on other techniques for reducing simulation time. This includes techniques used in state-of-the-art simulators such as statistical simulation, sampling and parallel simulation. We also cover works using field-programmable gate arrays (FPGA) prototyping with the aim of speeding up computer simulation.

We explain a set of existing multi-core simulators that are relevant for the work in this thesis either because the are widely used or because they implement some of the simulation time reduction techniques explained before. We also include a survey on the use of these simulation tools and techniques in major computer architecture conferences.

Finally, we give an overview of task-based parallel programming models. This background is important because the multithreaded applications driving the work in this thesis are programmed in OmpSs [68], a task-based programming model.

2.1 CellSim

CellSim [49, 147, 50, 51, 142] is a simulator modeling the Cell Broadband Engine (Cell/B.E.) microprocessor [100]. The introduction of the Cell/B.E. was a breakthrough due to its unique characteristics. It was the first high-performance heterogeneous multi-core. Heterogeneous designs have been widely used in the embedded market. Microprocessors such as the NXP Viper [71], TI OMAP [56] include a general-purpose core, a very-long-instruction-word (VLIW) core and a set of multimedia accelerators such as video and audio encoders and decoders. However, the Cell/B.E. was the first to have an impact on high-performance computing (HPC) although it was initially designed for the video-game market, precisely as the microprocessor of the Sony PlayStation 3 games console. A proof of its impact in HPC is that the number one supercomputer in the Top500 list [11] of June 2008 was mainly composed of Cell/B.E. microprocessors.

Its high-performance heterogeneous design opened new research opportunities including both software and hardware. There was no Cell/B.E. simulator publicly available, and we decided to develop one ourselves called CellSim. However, we found several difficulties during the development of CellSim that resulted in a reduced simulation speed that end up limiting its usability for exploring larger Cell/B.E.-like designs. Also, the Cell/B.E. line of microprocessors was discontinued because it was not economically successful. Altogether, we decided to discontinue CellSim, but the lessons learned from its development were the foundation for the research in this thesis.

2.1.1 The Cell/B.E. Microprocessor

The Cell/B.E. microprocessor is an initiative by Sony, Toshiba and IBM. Its development started in 2001, and it was introduced in 2005. It had a first implementation in 2005 using 90nm technology node, known as Cell Broadband Engine (Cell/B.E.). This implementation was mainly used in the PlayStation 3 games console. In 2008, a second implementation that was mainly a shrink to 65nm was announced. This implementation was known as PowerXCell 8i and became the microprocessor fueling the RoadRunner supercomputer [27], the first to achieve an HP Linpack (HPL) [67] performance over one PetaFLOP (PFLOP), and the fastest in the Top500 list from June 2008 to June 2009.

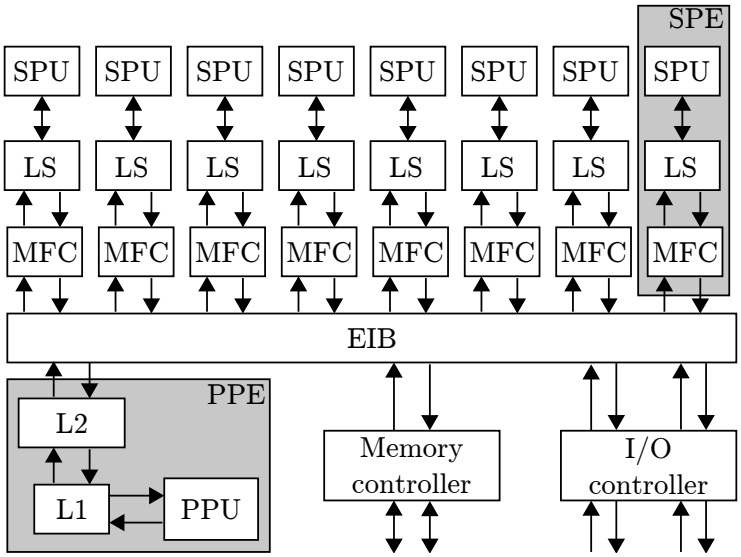


Figure 2.1: The Cell/B.E. microprocessor architecture [100].

Figure 2.1 shows a scheme of the Cell/B.E. microprocessor. It is composed of one general purpose core, called the Power Processor Element (PPE), and eight vector accelerators, each of them called the Synergistic Processor Element (SPE). The PPE is composed of a two-way SMT in-order 64-bit Power-Architecture [8] core, called the Power Processing Unit (PPU), with 64 KB of L1 cache (32 KB for instructions plus 32 KB for data) and 512 KB of L2 cache.

Both levels of cache are private to the PPE. The PPE executes the operating system and acts as a controller of the eight SPEs.

An SPE includes a SIMD processing core, called the Synergistic Processing Unit (SPU), a 256 KB local memory called Local Store (LS), and a direct-memory-access (DMA) engine called the Memory Flow Controller (MFC). The SPU is a two-wide-issue in-order core with a new SIMD-based instruction set architecture (ISA) [12]. It incorporates just SIMD execution units and has simple hint-based branch prediction because it targets energy efficiency for regular data-intensive codes. The SPU can only access data in the LS. The LS works as an scratchpad memory. To move data in and out of the scratchpad memory, it must be done using DMA commands in the MFC.

The eight SPEs and the PPE are connected using a three-ring interconnection network called the Element Interconnect Bus (EIB). The EIB also gave access to off-chip memory through the on-chip memory controller and to off-chip devices through the input/output (I/O) controller.

Both implementations of the Cell/B.E. were clocked at 3.2 GHz and provided a peak single-precision floating-point performance of 204.8 GFLOPS using all eight SPEs. The PowerXCell 8i included, unlike its predecessor, fully-pipelined double-precision floating-point units in the SPEs that provided 102.4 GFLOPS up from 12.8 GFLOPS in the first implementation. These facts confirmed how, although the first implementation of the Cell/B.E. targeted multimedia computing, for which double-precision floating-point is irrelevant, the interest of the scientific community led to a second implementation providing full-fledged double-precision floating-point capabilities.

However, the Cell/B.E. architecture also has some weaknesses. Programming to achieve high performance is tedious. The non-coherent nature of the LSs requires explicit data movement and the SIMD nature of the SPEs requires programming with vector data structures and deal with alignment, gathering and scattering of data while using close-to-assembler semantics. These programmability limitations added to the high design and development costs that led to its discontinuation.

2.1.2 CellSim Design

Our approach to simulating the Cell/B.E. was to use a modular infrastructure. Using a monolithic approach, apart from being generally bad software engineering practice, would lead to many dependencies among architecture components. Having a modular infrastructure allows having encapsulated components with a clear interface among them. For this purpose, we employed the UNISIM infrastructure [20] that allows to describe an architecture by specifying a set of modules and the connections among them.

In CellSim, the PPU, caches, SPU, LS, MFC, EIB and memory controller are independent modules. Figure 2.2 shows a scheme of CellSim including its modules and interconnections. The design allows a configurable number of PPEs and SPEs. This allowed to set up a Cell/B.E. configuration with one PPE and eight SPEs, and also exploring future Cell/B.E. implementations with more PPEs and SPEs.

CellSim is an execution-driven simulator. This implies that it has to support two different ISAs, the 64-bit Power ISA and the SPU ISA. Since the SPU ISA was new, we had to implement it from scratch and almost completely as we need

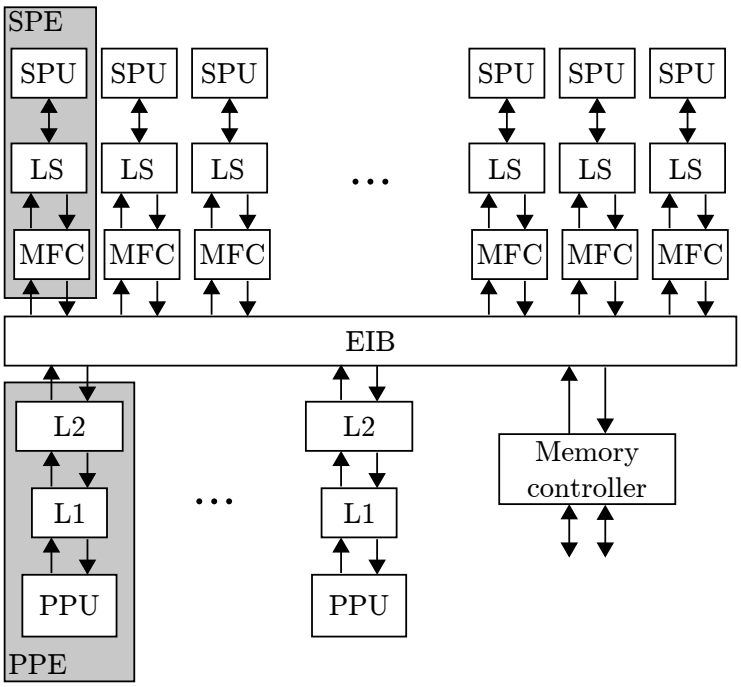


Figure 2.2: The CellSim simulator.

to cover around 75% of the instructions for the applications used for evaluation. The PPE works in system-call emulation mode. System calls are emulated by forwarding them to the native operating system running in the host. This implied there is no simulation of the operating system and thus there is no simulation of I/O devices, as shown in Figure 2.2.

The PPU pipeline model was simple, as our focus was on modeling the performance of the SPEs, which is the relevant processing component for HPC applications. The SPU pipeline model included a configurable number of execution units and a configurable issue width. The LS had configurable latency and allowed simultaneous access to different banks from the SPU and MFC. The MFC DMA engine was also modeled in detail to account for configurable processing delays, data transfer rates, packet sizes and transfer synchronizations.

We performed a functional validation of the PPU and SPU modules and a performance validation of the interconnection network [147, 51].

2.1.3 Lessons Learned

The main problem of CellSim is that it is slow for simulating large configurations. The largest configurations we simulated included 16 SPEs. This is due mainly to three facts:

- **Module communication overhead.** Modularity brings a set of benefits such as encapsulation and reusability. For example, our module cache served both as L1 and L2 cache. However, one must be careful with the complexity of module communications. UNISIM states a complex

communication protocol between modules. The sender module writes the data in the output port at the beginning of the cycle. Then, the receiving module accepts it or rejects it. If the receiver accepts it, then the sender can enable it or not. This three-way protocol unnecessarily complicates communication and introduces a large overhead for every interconnection port every cycle.

- **Software emulation.** Due to the execution-driven nature of CellSim, instructions must be functionally simulated (emulated). This has some benefits as we explain in Section 2.3. From a developer's point of view, implementing the instruction set functionality provides a deep understanding of the core features and capabilities. However, from a performance-prediction perspective, emulation adds a simulation overhead for every instruction, something that seems unnecessary if the objective is just to determine the instruction timing.

  An additional problem of execution-driven simulation appears when the model is tied to the ISA. In the case of CellSim, the SPU ISA was specific to the Cell/B.E. This restricted the flexibility of CellSim. When the Cell/B.E. was discontinued, software development for the Cell/B.E. stopped and we found ourselves with a restricted amount of applications to feed our simulator with.

  In modern execution-driven simulators, functional simulation is decoupled from the timing model. The functional simulation component translates the instructions to an intermediate "ISA-independent" representation that is fed to the timing model. This way multiple ISAs can be supported and the timing model becomes ISA independent [36].

- **Detailed modeling.** The use of execution-driven simulation allows a detailed modeling of the processing cores because all the information about the running instruction is available for the timing model. This is useful for detailed modeling of a specific core. However, when that detailed model is replicated for a large number of cores, simulation speed (cycles per second) drops dramatically (typically super linearly). Some of our experiments focused on the interconnect and off-chip memory bandwidth with an increasing number of SPEs. In these cases, most of the time was spent in the detailed modeling of the SPU and LS, while our exploration was not focused in those components. Also, for interconnect and memory bandwidth studies, it is interesting to find the sweet spot design that best manages contention with an increasing number of cores. However, simulating more than sixteen SPEs using such detailed models required long simulation times. As an example, simulations of sixteen SPEs and one PPE using CellSim run at approximately 9 kilo cycles per second (140 kilo instructions per second (KIPS)) on a Pentium 4 at 3 GHz. This is a slowdown of 33 333x compared to native execution.

The lessons learned from these experiences are the following:

- **Keep modules communication simple.** Modularity provides benefits in terms of clean and structured code, encapsulation and reusability. However, the communication protocol among modules must be simple: enough to do the job with the minimum performance overheads.

- **Keep the timing model ISA independent.** Functional simulation in execution-driven simulation must be decoupled from the timing model and an intermediate representation must be used for this to be ISA independent.
- **Use the appropriate modeling detail for each component.** The components modeled in detail must be those to which the analysis is targeted to. Depending on the type of studies, parts of the timing model may be abstracted to gain simulation speed at the expense of some accuracy loss in the not-so-relevant components. For example, for interconnect, cache hierarchy and memory studies, the model of the processing core, which is the most time-consuming part of the model, can be abstracted. This allows researchers to scale their simulations to larger numbers of cores.

2.2 TaskSim

After the discontinuation of CellSim, we decided to develop a new simulation framework from scratch with the objective of keeping the strengths and get rid of the weaknesses of CellSim.

First, we developed a simulation engine, referred to as *CycleSim*, to replace UNISIM. From our experience with CellSim, we learned that modularity gives flexibility, encapsulation and reusability, so CycleSim is also based on modules. However, one of the issues in UNISIM was the complexity of the communication protocol between modules, so we simplify it and went from a three-step to an up-to-two-step protocol.

The simulation procedure is similar as with UNISIM, but we redesigned it for speed. UNISIM executes all modules every cycle and all connections must be set every cycle. In CycleSim, we only execute those modules that need to be executed in a given cycle, and even skip *empty* cycles in which no module has scheduled activity.

Using CycleSim, we developed a set of modules to model the components in multi-core architectures. These modules are then glued together and configured to compose the target architectures of interest.

The CycleSim simulation engine, the modules and the configurations using those modules compose the simulation framework we refer to as TaskSim. TaskSim has been used to carry out a variety of research studies (see Section 7.2) and is the platform on which we have incorporated and evaluated the simulation methodologies and techniques proposed in this thesis.

In the following sections we provide more details on the CycleSim engine, and the modules and configurations we have simulated with TaskSim.

2.2.1 CycleSim

CycleSim allows to describe an architecture using a set of modules and their interconnections. Figure 2.3 shows a producer-consumer communication example between modules using CycleSim. A Producer module sends some data to a consumer module using the connection at the top of the figure that connects the *out* port in Producer to the *in* port in Consumer. Consumer has a queue to store that data until it can be processed due to its processing latency. Then,

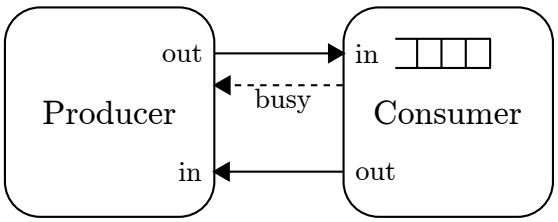


Figure 2.3: Producer-consumer module example using CycleSim.

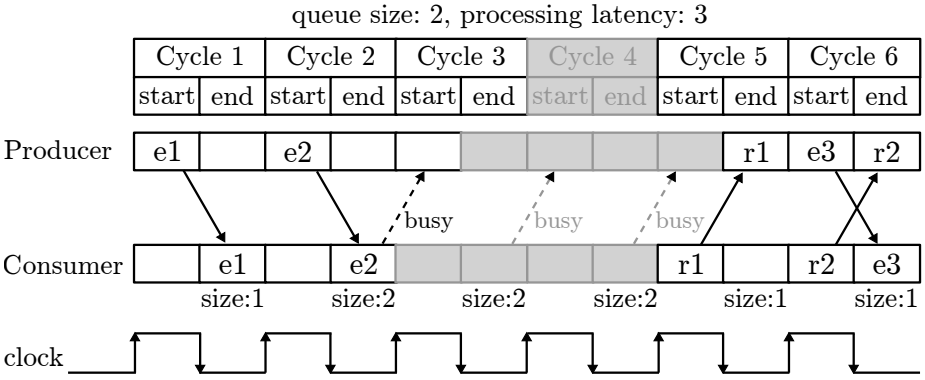


Figure 2.4: Example of communication between Producer and Consumer modules in CycleSim.

when the Consumer queue gets full, it has to notify the Producer to stop it from sending more data that it could not fit in the queue. For this kind of situations, CycleSim provides the *busy* signal. The busy signal is generally associated to a interconnection for data, and serves to notify whether the receiving module is ready for processing the data, or is busy and cannot process it. In the example, the Consumer module can set the busy signal so the Producer module does not send more data until the busy signal is unset.

After processing the data, the Consumer module sends the result to the Producer through the interconnection shown in the bottom of the figure. This interconnection does not need a busy signal because the Producer module is at any time ready to process that communication.

As previously mentioned, UNISIM requires all signals to be set every cycle, including the data in the sender module, the acceptance in the receiver module and the enabling of the data in the sender module. Contrarily to UNISIM, in CycleSim, the signals between modules does not necessarily have to be set every cycle, and the busy signal is optional. This results in a much lower overhead per cycle.

Also, in UNISIM, all cycles must be simulated. CycleSim, however, only simulates a module if it has some activity in that cycle or if a signal in its input ports changed. Cycles where no module has activity are skipped, similar to the operation of event-driven simulators.

Figure 2.4 shows the cycle-by-cycle operation of the producer-consumer ex-

ample shown before. In this example, the Consumer module has a queue that can hold two elements, and it takes three cycles to process the element and send back a response. By convention, modules send data at the start of the cycle, and set/unset their busy signal at the end of the cycle. In the figure, we show the number of elements in the Consumer queue at the end of each cycle.

In the first two cycles, the Producer module sends two elements to the Consumer and its queue gets full. The Consumer then sets the busy signal, and notifies that it does not have to do anything until Cycle 5. In Cycle 3, the Producer reads the busy signal, and notifies that it will not have anything to do until that signal changes or it receives any other data. Therefore, the Consumer module is not executed again until Cycle 5, when it sends back the response to element e1, and the Producer module awakes to handle it. In Cycle 6, the busy signal is not set, the Producer module then sends a third element, and the Consumer sends back the response to element e2.

With this operation, in Cycle 3 the Consumer module was not executed and the Producer only for half cycle. Cycle 4 was never simulated, and in Cycle 5 the Producer only simulates the end of the cycle. The cycles not simulated are shown in grey in the figure.

This implementation allows simulations scalable to large numbers of cores. This is because the number of cycles to be simulated does not determine simulation time, but simulation time is determined by the activity to be simulated in the modeled components.

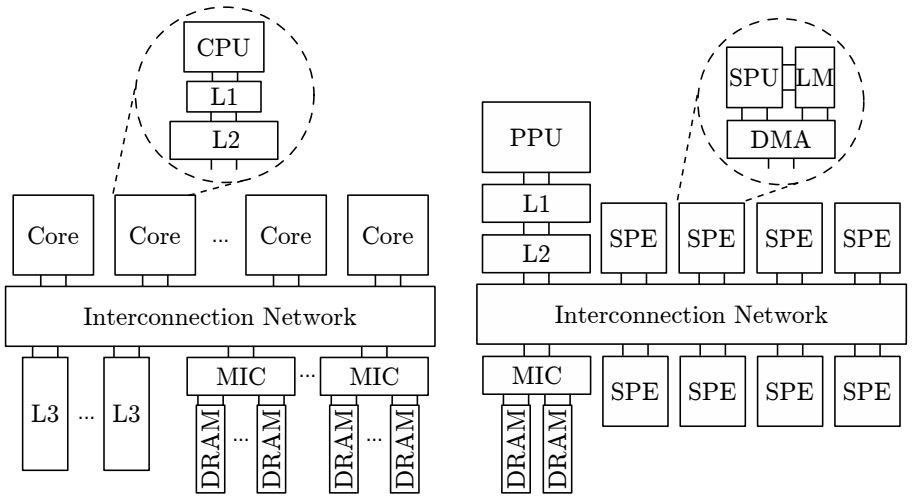
2.2.2 Modules and Configurations

Using the CycleSim semantics, we implemented a set of modules to model cores, caches, local memories, DMA engines, memory controllers, off-chip memory modules and interconnects. This set of modules is used to compose different architectures, thus demonstrating one of the benefits of modularity: reusability.

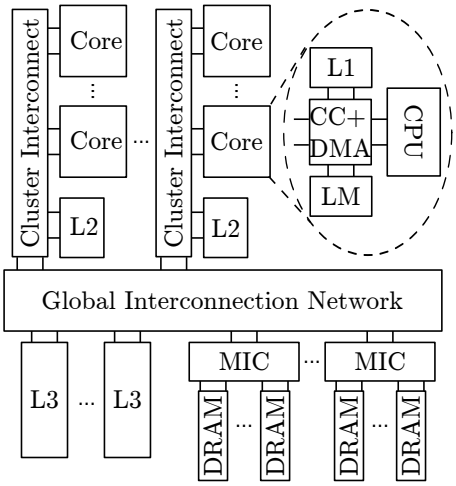
Figure 2.5 shows three examples of target architectures depicted in terms of modules and their interconnections. The first case, in Figure 2.5a, is an SMP configuration with three levels of cache, with a configurable number of cores, L3 cache banks, memory controllers and memory modules.

The configuration in Figure 2.5b models the Cell/B.E. microprocessor (see Section 2.1.1). In this configuration, the L1 and L2 caches are configured to mimic the ones available for the Cell/B.E. PPU, the interconnection network module is adapted to provide the same bandwidth and latency as the Element Interconnect Bus, the scratchpad memory (LM) modules are configured as Local Stores, and the DMA engine module is modified to work as the Memory Flow Controller.

The third example, in Figure 2.5c, is the SARC project architecture [141]. A two-level hierarchical network connects the computation cores in the system to a three-level cache hierarchy and a set of memory controllers giving access to off-chip memory. The several last-level cache (L3) banks are shared among all cores, and data is interleaved among them to provide maximum bandwidth. The L2 banks are distributed among different core groups (clusters) and their data placement policy can be configured to optimize either for latency (data replication), or bandwidth (data interleaving) [170]. Each core has access both to a first-level cache (L1), and to a scratchpad memory (LM). To have deterministic quick access to data, applications may map a given address range to the



(a) SMP (b) STI Cell/B.E.



(c) SARC architecture

Figure 2.5: Examples of architectures modeled using TaskSim.

LM, thus avoiding cache coherence issues. A mixed cache controller and DMA engine manages both memory structures (L1 and LM), and address translation.

In TaskSim, the modeling effort is devoted to the memory system modules (cache hierarchy, scratchpad memories and off-chip memory), and interconnection network. We model these components in detail and, in contrast, abstract the core model. This is due to two reasons:

- The focus of our research is in *macroarchitectural* studies regarding the memory system, multi-core density and multi-core organization.
- A detailed modeling of the core pipeline operation is one of the main performance limiting factors in terms of simulation speed due to its costly operation [108].

Therefore, we allow multiple abstraction levels for the core model depending on the speed and accuracy required. These abstraction levels of the core model are one of the contributions in this thesis and are presented in Chapter 4.

2.3 Trace- vs Execution-driven Simulation

A plausible classification of computer architecture simulators states two groups: functional and performance simulators. Functional simulators just provide functional correctness and programmers use them for application development and debugging. Performance simulators on the other hand aim to provide a timing prediction for the application running on a target machine. Functional simulators are inherently execution-driven, while performance simulators can be either execution-driven or trace-driven. In this section, we compare execution-driven and trace-driven simulation in the context of performance simulators.

The fundamental difference between execution-driven and trace-driven simulators is their input. The input of an execution-driven simulator is the application binary, associated libraries and input files. Using these inputs, execution-driven simulators emulate the execution of the application and forward the required information to the timing model. Trace-driven simulators, on the other hand, receive as input a *trace* file including enough information about the application execution to predict its timing. In this case, trace-driven simulators read the required information directly from the trace and forward it to the timing model.

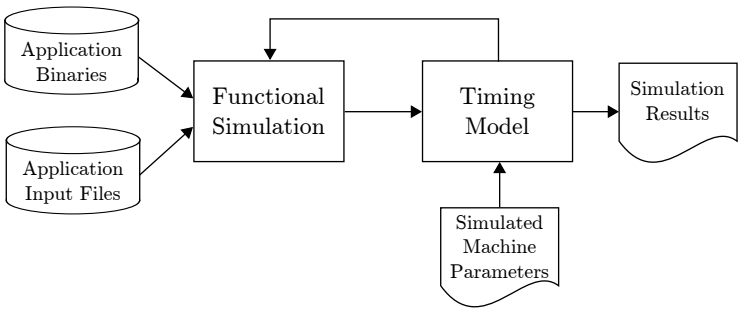
Figure 2.6 shows the flow chart of the simulation process for an execution-driven simulator (Figure 2.6a) and for a trace-driven simulator (Figure 2.6b). The difference is in the front-end. The execution-driven simulator feeds the timing model with the output of the functional simulation, while the trace-driven simulator does it with the output of the trace reader. The timing model, simulated machines and kind of results from simulation can be common to both approaches.

In both cases, the information required by the timing model depends on what is being modeled and how accurate is the model. For example, to simulate a cache memory in terms of hit ratios, the sequence of memory accesses is enough. However, to model the performance of a processing core pipeline in detail, the timing model requires the executed instructions, memory accesses and branch directions and targets.

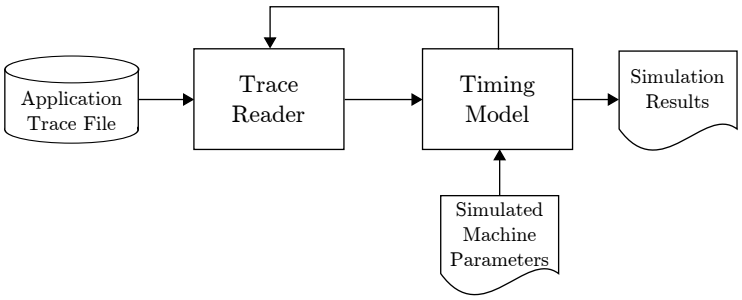
Although the difference between the two simulation types refers to the front-end of the simulation process, it has implications in terms of capabilities and limitations that we explain in the following sections.

2.3.1 Host System Requirements

An execution-driven simulator has to emulate the execution of the target application running on the simulated machine, and therefore it inherits the application system requirements, such as memory space for code and data, and disk space for input files. For simulating large multi-core configurations, these system requirements may exceed the resources of the host machine. As an example, some high-performance applications require several hundred megabytes of memory per thread [134]. Then, for weak scalability experiments on a future



(a) Execution-driven simulation



(b) Trace-driven simulation

Figure 2.6: Flow chart of execution-driven and trace-driven simulation.

multi-core configuration with hundreds of cores, the simulation will require tens of gigabytes of memory, which limits the execution-driven simulation of such scenarios to large-scale host machines.

A trace-driven simulator, however, does not inherit the system requirements of the application, as it does not need to emulate the application execution, and thus, does not need to allocate its data or store its input files. However, trace files for long applications are usually large. A trace file including instruction-level information for a few seconds of execution may require tens of gigabytes of storage. To reduce this disk space requirements, trace files may be compressed using standard compression algorithms that are agnostic to the semantics of the trace, such as those used in standard tools like *gzip* [65] and *bzip2* [44]. Moreover, there are also trace compression techniques that take advantage of the semantics of the trace contents to reduce the amount of bytes to represent them. Several works in the late 1980s and early 1990s proposed trace reduction techniques motivated by the low disk capacities in those times [156, 107, 138]. These techniques targeted to reduce trace file size, but in some cases they also served to reduce simulation time [167].

2.3.2 Dynamic Behavior Support

Applications may have dynamic behavior that depends on the machine state, for example computations based on random numbers, such as Monte-Carlo methods [5]. In these applications, different code may be executed based on the value of some piece of data, such as the case of random numbers. Dynamic behavior

is also present in dynamically-scheduled multithreaded applications and operating systems. Multithreaded applications with dynamic scheduling may assign work to threads based on the availability of resources or considering heuristics to improve performance or energy efficiency based on the characteristics of the underlying machine. The same applies at a higher-level in the operating system scheduler, where multiple processes are assigned to multiple execution threads. Moreover, dynamic behavior is also present in synchronization operations of parallel applications, including operating systems, where the waiting decisions depend on the state of the threads involved in the synchronization.

Thanks to the emulation of execution-driven simulators, these can reproduce dynamic behavior at simulation time based on the state of the simulated machine. Even some full-system simulators are able to boot unmodified versions of operating systems that execute on real machines [114, 33].

However, trace-driven simulators are restricted to the behavior statically captured in the trace file during the trace generation run. Therefore, the target application behavior can not typically change for different machine configurations as it would happen in a real machine. Some works in the early 1990s addressed this problem for statically-scheduled applications by reproducing synchronization operations such as locks and barriers at simulation time [105, 88, 82]. One of the contributions in this thesis is to enable the use of trace-driven simulation for dynamically-scheduled applications, by reproducing the dynamic behavior using an unmodified version of the parallel application's runtime system in cooperation with a trace-driven simulator (more details in Chapter 3). The problem still exists for simulating operating systems using traces, so more research is needed in this direction.

As a result, execution-driven simulators can simulate any kind of application provided that they support the instructions to be executed. However, trace-driven simulators have been limited to single-threaded, multiprogrammed and statically-scheduled applications. In Chapter 3, we present our approach to simulate dynamically-scheduled applications using traces. The simulation of operating systems using trace-driven simulation remains an open problem.

2.3.3 Modeling Abstraction

Execution-driven simulators have to emulate the execution of the target application. This implies that the simulator needs to process the executed instruction-stream. This is the highest level of abstraction an execution-driven simulator, in its overall, can get. The timing model may then use instruction-level information or may use a higher level of abstraction, such as processing memory accesses. However, having to provide functional simulation already incurs in a significant slowdown in the order of hundreds or thousands of times slower than native execution [36, 148].

Trace files can include any kind of information, which is only restricted by the kind of information required by the timing model. Therefore, the level of abstraction of the simulated model can be raised and this will directly translate in a simulation speed increase. One of the contributions in this thesis is to explore the implications of raising the level of abstraction in simulation. We also compare levels of abstraction in popular execution-driven simulators with the ones we developed in our trace-driven simulator. More details can be found in Chapter 4.

2.3.4 Restricted-Access Applications

Execution-driven simulators require the target application binaries. Therefore, in some situations where the access to target application binaries is restricted, execution-driven simulation is just not possible.

In these cases, the owner of the application can distribute traces including information about the application that allows performance analysis but does not disclose sensible information that would break the confidentiality agreement over the application. This situation happens in microprocessor-design companies working with clients to develop a new chip targeting the client's applications, but these applications cannot be shared due to the legal policy of the client. In this scenario, it is very useful to provide the microprocessor-design company with trace files of the target applications and use trace-driven simulation to drive the design of the chip.

2.3.5 Speculation Modeling

The modeling of processing cores with speculation support requires the simulation of the instructions in the wrong-path on branch mispredictions. Instruction-level trace files for trace-driven simulators typically include the stream of executed instructions, but not those in the wrong path. This limits the accuracy of trace-driven simulators as they can not account for the impact of speculative instructions, such as speculative memory access reads. The impact of these inaccuracies was studied in a previous work by Mutlu et al. [129] which stated a 5% error by not simulating the wrong-path instructions.

This is generally understood as a fundamental limitation of trace-driven simulation but, in fact, there have been already works addressing this issue. An approach by Bhargava et al. [31] searches for executions of the wrong path on a branch misprediction in other locations in the trace where that branch took the other direction and thus was the right path in the real execution. This allows to have both paths of the execution for every branch. They report to cover 99% of the branches with this technique.

2.3.6 Development Effort

Execution-driven simulators need to support the target ISA, or at least the ISA subset that covers all instructions in the target applications. This implies the development of object file readers, instruction decoders, and the implementation of the functionality of every instruction. This usually requires a higher development effort than coding trace readers and writers, and it is also sensible to the host machine characteristics, such as when the endianness of the target ISA and the host differ.

Trace generation for trace-driven simulators has benefited from existing binary instrumentation tools that already provide the capabilities to execute the application binary [161, 113, 130] and execute a set of user-defined callbacks with certain parameters to provide the required information for the simulation to be stored in the trace. This eases the development of trace generation tools that can make use of a unified trace writer/reader that can also be used in the simulator front-end.

In the case of execution-driven simulators coding the front-end is a time-consuming development effort. For this reason, there are approaches to use binary instrumentation tools, virtual machines and standalone emulators to serve as the simulator front-end:

- **Binary instrumention.** There is a trend on using binary instrumentation tools to trace the application online and feed the timing model without having to store a trace file on disk [92, 97, 120]. This removes the disk space requirements of trace-driven simulation but requires to *generate* the trace in every simulation. Whether binary-instrumentation-driven simulators are closer to execution-driven or trace-driven ones in terms of capabilities and limitations depend on the features of the binary instrumentation tool itself. For example, to simulate speculative instructions, the instrumentation tool needs to checkpoint the application, execute the wrong path on a mispredicted branch, and restore the checkpoint later.

- **Virtual Machines.** Binary instrumentation tools are useful as a front-end to simulate single applications. However, they can not be used to simulate full operating systems. The development of full-system simulation and support for operating system in an execution-driven simulator is costly. For this reason, there are approaches to use virtual machines as a simulation front-end [155, 103, 133]. In this cases, the front-end has information of the guest operating system and the processes running on it and can forward it to the timing model. In this case, the simulator does not need to model devices because the virtual machine already does this job.

- **Standalone Emulators.** To avoid developing a full new emulator for new performance models, there are approaches to use standalone emulators [114, 18]. Emulators, or functional simulators as described at the beginning of this section, aim to provide functional correctness, but some of them also support to provide information to drive a performance simulator [114]. Emulators usually model devices and provide support for operating systems so it is possible to perform full-system simulation.

These alternatives for the simulator front-end are attractive because they allow researchers to concentrate on the development and refinement of the performance model required to perform their experiments. In general, binary instrumentation tools, virtual machines and emulators do not target performance simulation and thus do not have a notion of timing. In some cases, these have been extended to have this notion [103] or to accept feedback from the performance simulator to adjust the speed at which they execute the target application [18]. In some other cases, these tools already have a notion of timing or even accept to plug timing models by providing a specific API for this purpose [114].

To avoid the limitations of these tools inherent to the fact that they do not target performance simulation, there are works that provide an open execution-driven front-end specifically designed to attach external performance models to it [166].

2.3.7 Summary

Trace-driven simulation has a number of advantages and disadvantages over execution-driven simulation that we have covered in our discussion in previous sections and are listed in Table 2.1.

In general, one of the clear advantages of using traces is that simulation avoids the system requirements of the target application, such as memory and disk space. Also, the sharing of traces instead of binaries for confidential applications is a clear advantage, but is only exploited in specific cases where the driving applications are secret. This happens more often in industry rather than in academia, where the driving applications are usually standard benchmarks with a free or purchasable license.

The main limitations of trace-driven simulation with respect to execution-driven is the reproduction of dynamic behavior. We have explained works to cover statically-scheduled applications, and we refer to Chapter 3 for more details on our approach to cover dynamically-scheduled applications. Full-system support for operating system is still not possible with trace-driven simulation.

In terms of development effort, trace-driven simulation has an advantage due to the lack of a complex emulation front-end. However, the existing front-end options we covered in the previous section reduce the burden of developing new execution-driven simulators and close the gap in terms of development effort with trace-driven simulation.

Nonetheless, for all of these options, simulation speed is determined by the front-end overhead to execute the application and extract the information required for the timing model. In trace-driven simulation, there is no need for application emulation and this opens the door to raising the level of abstraction. This is the most important advantage of trace-driven simulation to achieve higher simulation speed, thus enabling the simulation of larger and more complex multi-core systems.

2.4 Simulation Time Reduction

Simulation is a slow process. As an example, a compliant run of the SPEC CPU2006 benchmark suite requires the simulation of 26 benchmarks with several trillion instructions each. Such single-thread simulations may take weeks or even months using detailed cycle-level simulation.

Researchers are conscious of this problem and have been proposing techniques to reduce simulation time. In this section we cover such techniques in six different categories: reduced setup and reduced input set, truncated execution, statistical simulation, sampling, parallelization, and FPGA acceleration.

We do not cover techniques to reduce simulation time by raising the level of abstraction in this section because we devote an entire chapter for this purpose. A comprehensive discussion about using multiple abstraction levels for speeding up simulation can be found in Chapter 4.

2.4.1 Reduced Setup or Input Set

Simulation time typically increases super linearly with an increasing number of simulated cores. This is because, even simulating the same number of total instructions, accesses to shared resources from different cores require the

Table 2.1: Main advantages and disadvantages of trace-driven simulation over execution-driven simulation

Advantages	Disadvantages
Not dealing with application data. The simulation thus avoids the application memory size requirements.	Inability to simulate the wrong-path instructions in speculative execution. Addressed by Bhargava et al. covering 99% of wrong-path instructions [31].
Not dealing with application input files. The simulation thus avoids the application disk size requirements.	Inability to simulate the dynamic behaviour of multithreaded applications. Addressed in our work in Chapter 3.
Allowing fast high-level-abstraction modeling. Execution-driven must provide, at least, functional simulation (emulation).	Inability to simulate the dynamic behaviour of operating systems. That prevents using traces for full-system simulation.
Sharing of confidential applications through the sharing of traces instead of code or binaries.	Dealing with potentially large trace files.

simulation of additional cache coherence actions, more complex interconnect arbitration and resolution of contention scenarios.

To avoid unreasonably long simulation times, researchers end up reducing the simulated machine setup. This results in evaluations using machine configurations similar to the ones of already-existing commercial products. However, it is necessary to assess the validity of new techniques for future (larger) microprocessors if researchers want to undoubtedly influence their design. In Section 2.6.2, we show statistics about the number of cores simulated in research papers published in main computer architecture conferences. These statistics show that using a reduced setup is a widely used technique for reducing simulation time. Between 40% and 60% of papers simulate no more than four cores, and, in general, around 80% of papers simulate up to sixteen cores.

Another technique to reduce simulation time is using reduced input sets. The length of an application execution is determined by the complexity of the computation and the amount of data to be processed. The *reference* input sets of benchmark suites such as SPEC CPU, require the execution of several trillion instructions, which makes the simulation of a single core to take several weeks or even months. For this reason, benchmark suites provide reduced input sets for simulation.

The objective of the reduced input set is to match the characteristics of the reference input set. Such characteristics can be in the form of instruction mixes, cache miss ratios or instruction-level parallelism, and some works have used them to assess the validity of reduced input sets [104, 73]. Examples of reduced input sets are the MinneSPEC [104] input sets for SPEC CPU2000, the *test* and *train* input sets of SPEC CPU2000 and SPEC CPU2006, and the

simulation input sets of the PARSEC benchmark suite [32].

2.4.2 Truncated Execution

Truncated execution consists on simulating just a certain amount of consecutive instructions from the total execution. This method, however, does not guarantee that the simulated execution chunk has the same characteristics of the entire execution.

Applications usually have an initialization phase, a computation phase (which is the largest phase and may have multiple sub-phases with different behaviors), and a finalization and cleanup phase. Studies simulating a given amount of instructions starting from the beginning of the execution are prone to consider a non-representative part of the execution, as they simulated the whole initialization and part of the computation phase. For this reason, many works fast-forward simulation for a certain amount of instructions, and start detailed simulation (measurement) for another given amount of instructions with the hope of taking a large enough chunk of the computation phase that is representative of the entire execution.

Another problem of this technique is that the architectural state of the simulated machine is empty after fast-forwarding and right before detailed simulation, a situation called *cold start*. The state is then different to the state of the execution at that point if detailed simulation is performed from the beginning to the end. This leads to wrong behavior (increased cache misses and branch mispredictions) in the initial part of the measurement. For this reason, this method usually includes a warm-up during a certain amount of instructions after fast-forwarding and before measurement. During warm-up, detailed simulation is performed to fill the architectural structures. The behavior during warm-up is discarded and only what happens during measurement is considered for evaluation purposes.

Truncated execution usually employs a fixed amount of instructions for fast-forward, warm-up and measurement. This is problematic as different benchmarks have completely different initialization, computation and cleanup durations. However, and in spite of the lack of representativeness of truncated execution, a previous work in 2005 [179] reports that it was the most used technique for reducing simulation time in main computer architecture conferences from 1995 to 2005.

2.4.3 Statistical Simulation

Statistical simulation consists on simulating a trace that is statistically similar to the average overall application behavior but is orders of magnitude shorter than a trace of the entire application execution [180].

The first step of statistical simulation is to profile the application and extract statistical characteristics that define its behavior. Using this profile, a synthetic trace is generated including synthetic instructions in the same magnitudes and characteristics as in the original execution. Synthetic instructions do not intend to cover the whole ISA of the machine where the profiling was carried out, but are a representative smaller set. A synthetic instruction may include its type, dependencies and whether it produces a branch misprediction or an instruction or data cache miss.

The synthetic trace is simulated with a trace-driven simulator that implements a simplified architecture model. The execution of different instruction types on the core pipeline considers resource and data hazards as in a detailed simulator. However, branch mispredictions and cache misses are modeled in a simpler way: the simulator just flushes the pipeline and computes the miss delay, respectively.

Due to the shorter simulated trace and the simpler simulated model, statistical simulations are orders of magnitude faster than detailed simulation. However, these simplifications are at the expense of simulation accuracy. Existing works have performed experiments showing absolute errors within 8% [131] and 18% [72], and even smaller relative errors.

2.4.4 Sampling

Sampling consists on selecting a subset of a statistical population to estimate the characteristics of the whole population by extrapolating the characteristics of the subset.

In computer architecture simulation, sampling is used to select a subset of the application execution (sample) to estimate the characteristics of the entire execution. As done by Yi et al. [179], we explain sampled simulation techniques in three categories: representative, periodic and random sampling.

Representative sampling consists on selecting a subset of the executed instructions that is representative of the entire execution. An example of representative sampling is SimPoint [159, 135]. SimPoint selects a few chunks of execution based on the list of basic blocks that they execute. The first step is to do a profiling of the application by splitting it in chunks of a certain number of instructions. For each chunk, it determines the basic blocks that are executed and how many times are they executed. Then, it applies a clustering algorithm do detect chunks that are similar in terms of the basic blocks they execute and how many times each basic block is executed. Then, they select one or few samples for each cluster, and the resulting set of chunks is considered representative of the whole execution. The authors of SimPoint compare the error of simulating the representative set over simulating the entire execution and report an absolute error within 8% for IPC, and within 20% for metrics such as cache misses and branch prediction.

Periodic sampling consists on selecting a subset of the executed instructions at fixed intervals. SMARTS [177] is an example of periodic sampling that uses statistical sampling theory to give an estimate of the IPC error of the sampled simulation compared to the entire simulation. This estimation allows the user to tune the sampling frequency and length of each sample to increase accuracy. The authors report a CPI error within 3%. They also compare their approach with SimPoint, and show that SMARTS is consistently more accurate for their experiments but at the expense of requiring a longer simulation time: 5.0 hours (SMARTS) instead of 2.8 hours (SimPoint).

Random sampling consists on selecting a subset of the executed instructions randomly. The parameters for this process are the number of samples and their length.

All sampling methods suffer the *cold start* problem previously explained for truncated execution. The state of the simulated machine at the beginning of the simulation of a sample is different to the state at that point when simulat-

ing the entire execution. The solution for sampled simulation is the same as for truncated execution: simulating for a period of time in detail to warm up the architectural structures before starting measurement. The usual method is then to fast-forward simulation using functional simulation, warm up before the sample, and simulate in detail for the duration of the sample.

Although functional simulation is faster than detailed simulation, it is still orders of magnitude slower than native execution. To further accelerate the process, the authors of SMARTS proposed TurboSMARTS [175]. In this approach, the state of the machine before each sample (checkpoint) is saved using a single functional simulation. Then, for every experiment, the state of the simulated machine is restored before warm up and detail simulation of every sample. This completely avoids functional simulation. With this technique they report a reduction in average simulation time for the SPEC CPU2000 benchmarks from 7.0 hours (SMARTS) to 91 seconds (TurboSMARTS).

2.4.5 Parallelization

Parallel simulators execute the simulation of different parts of the simulated machine on multiple threads of the host machine. The common approach is to simulate each core on a separate thread and synchronize them on accesses to shared resources, such as a common shared cache. However, this fine-grained synchronization decreases the advantage of parallelization, and gets worse for increasing numbers of cores sharing a common set of resources. This is the approach by Parallel Turandot CMP (PTCMP) [66], a parallelization of the Turandot simulator [125, 127], explained in more detail in Section 2.5.6. The parallelization of PTCMP is evaluated for up to four cores, and the maximum speed-up achieved is 1.5x running on 3 separate threads.

Synchronization is required to avoid timing violations. A timing violation happens when a simulated thread that is behind in terms of simulated cycles carries out an action that affects another thread that is ahead in simulated cycles. In this situation, it is already too late for the thread that is in a future cycle to account for the action that happened in the past.

In order to get higher speed-ups, other works employ more complex ways of synchronization. TimeWarp [99] and SlackSim [58] let threads to run freely (TimeWarp) or having up to a certain difference in simulated cycles (SlackSim), while they periodically save the state of the simulated machine. Then, on a time violation, they restore the latest checkpoint before the timing violation and resume simulation from there. Graphite [121] goes a step further and let timing violations to happen, thus assuming the error they incur. To decrease the amount of timing violations, Graphite has the option of synchronizing on application synchronization operations such as barriers and point-to-point synchronizations.

Although parallel simulation reduces simulation time, it is only useful for the evaluation of one or few configurations. The reason is that parallel simulation does not allow the exploration a larger design space. This is because, for reducing simulation time, it employs more host machine resources. Then, for design space exploration, multiple host threads can be used for running multiple simulations in parallel rather than executing one simulation in parallel. As a result, when many configurations must be evaluated, running many serial simulations in parallel is more efficient and provides better simulation throughput.

2.4.6 FPGA Acceleration

Another approach to accelerating multi-core simulation is to implement the simulated model on an FPGA. Research Accelerator for Multiple Processors (RAMP) [174, 106, 162] is a project to build the hardware and software infrastructure to simulate multi-cores and multiprocessors using FPGAs. They implement a communication infrastructure among multiple FPGAs, modeling several cores each, and advocate for a community-maintained set of modules that can be used to quickly setup new configurations. The model for each core is split into the functional and timing models [162]. They show that running the functional simulation only is in the same order of magnitude in terms of speed compared to software functional simulators. However, when a timing model is added, the FPGA is several orders of magnitude faster than the software simulator. In a comparison of the RAMP simulator to the Simics/GEMS software simulator (see Section 2.5.2), simulations of 64 cores with two levels of cache and cache coherence, RAMP is 263x faster.

2.5 Chip Multiprocessor Simulators

There are many computer architecture simulators that are used for doing research in multi-core systems. In this section, we cover the ones with some special interest either because they are widely used or because they use some of the simulation time reduction techniques explained in Section 2.4.

2.5.1 Simplescalar and Derivatives

Simplescalar [43, 21] is an execution-driven simulator and one of the most used computer architecture simulators in the 2000s with more than 4000 citations in research papers. It models a single out-of-order superscalar core with two levels of cache and a fixed latency to main memory. The core model assumes a unified structure for the issue queue, reservation stations and reorder buffer that, together with the functional units, it is called the *register update unit*.

This simple model made Simplescalar attractive because introducing modifications and getting experimental results was a relatively fast procedure. Also, the fact that some works compared its accuracy to real hardware [64] also increased its popularity although it was shown to have an average error on a set of SPEC CPU 2000 benchmarks of up to 36% on IPC.

There have been works to improve the modeling accuracy of Simplescalar or extending it for simulating multi-cores [64, 182, 53, 17, 178, 58, 111].

One of them is the Zesto simulator [111]. The main objective of Zesto is to increase the modeling detail of Simplescalar. It provides separate issue queues, reservation stations and reorder buffer, models caches in more detail with limited miss handling status registers (MSHR) and prefetching support, among others, and adds a DRAM detailed model together with a model of the memory controller. Because of this low-level detail, Zesto is claimed to be slower than other research pipeline simulators of the time. Zesto is aimed to single core simulations, but it can also simulate multi-cores but it is limited to multiprogrammed workloads.

Another multi-core simulator based in SimpleScalar is SlackSim [58]. SlackSim, apart from extending Simplescalar to simulate multiple cores, it also par-

allelizes simulation. It allows multiple cores to run in parallel but preventing them to be ahead of other threads by more than a given number of cycles (slack). By adjusting this slack appropriately they minimize timing violations (see Section 2.4). They also incorporate a mechanism to periodically save the machine state (checkpoint), and backtrack simulation to the lastest checkpoint in case of a timing violation [59]. This mechanism is not new though, as it was already proposed in 1982 with the name of Time Warp [99].

2.5.2 Simics and Derivatives

Simics [114] is a full-system functional simulator, thus execution-driven, that supports a large variety of ISAs. Its main target is to work as a virtual platform for software development. However, it has the option to run having a notion of timing and, in this mode, it allows to plug timing models.

Simics was introduced in the early 2000s and became a popular alternative for carrying out full-system simulation. Simics models all the necessary devices to run unmodified versions of a large variety of operating systems including Linux, Windows, MS-DOS and Solaris. This eased the work of researchers that just had to use Simics together with their custom timing models to perform their experiments [1, 22, 40, 60].

A timing model for Simics that has been widely used is GEMS [116]. GEMS includes a SPARC core pipeline model called Opal, and a cache hierarchy, interconnection network and memory model called Ruby. Simulations can be performed with Ruby only or with Ruby and Opal together. Ruby supports several cache coherence protocols, including broadcast-, token- [115], and directory-based versions of MSI, MESI or MOESI. The Opal model is very detailed and this results in accurate but slow simulations. As shown later in Chapter 4, adding the Opal core modeling over Ruby makes simulation an order of magnitude slower.

Another timing model for Simics is Flexus [91]. Flexus is interesting because it implements the TurboSMARTS [175] sampling technique explained in Section 2.4.4. As explained before, this sampling technique consists in simulating only some statistically representative parts of a benchmark to reduce simulation time. To do this, they save the machine state (checkpoint) before each representative chunk of the benchmark. To save all the required checkpoints fast, they do it using functional simulation. Then, for detailed simulation, they restore the machine state at the beginning of a representative sample and then perform detailed simulation from there and until the end of the sample. They repeat the process for all samples.

Both GEMS and Flexus are mainly used for simulating multi-cores, but their detailed slow operation generally limits simulation to 32 and 16 cores respectively.

2.5.3 M5/gem5

M5 [34] is a full-system execution-driven simulator initially targeted to networking workloads. For this purpose, M5 allows the simulation of multiple machines and run client/server applications to analyze the performance of network protocols and interconnects with a focus on hardware/software co-design.

M5 is open-source and that made it attractive as an alternative to Simics, which is a commercial product. That is the case that, in 2009, it started the merge of GEMS and M5, giving birth in 2011 to the gem5 simulator [33]. Since then, GEMS (for Simics) is discontinued and the efforts of the GEMS team are focused on gem5. With this merge, researchers have not only the M5 cores, caches and interconnects models, but also Opal and Ruby from GEMS available for simulation with gem5.

gem5 supports Linux for the Alpha, ARM and x86 ISAs, and Solaris for the SPARC ISA. This has attracted several companies, such as AMD and ARM, to benefit from full-system simulation which is necessary for the analysis of OS-intensive applications. It has also attracted researchers to use it for their experiments, to integrate it with other existing simulation platforms [93], or to assess its accuracy [45].

2.5.4 Graphite

Graphite [121, 120] is a parallel simulator that uses PIN [113], a dynamic-binary instrumentator, for functional simulation. It models a tiled multi-core architecture and is able to simulate each tile on a separate host thread. It is also able to spread the simulation over multiple host machines.

To avoid the overhead of synchronization on every access to shared resources out of a tile, namely the interconnection network, it uses *lax synchronization*. It does not synchronize on every access out of the tile, but only on those where the receiver is behind the sender, that is the receiver does not process the message until it is at the same time stamp as when the message was sent. However, if the receiver is ahead of the sender, that is it receives a message in the past, it just processes the message and assumes the error. It also allows to synchronize on application synchronization operations such as barriers and point-to-point synchronizations. With lax synchronization, they get better speedup at the expense of accuracy. They report a 4x speedup by using 80 cores (10 host machines).

Sniper [52] is an extension to Graphite that replaces the core model by an analytical model called *interval simulation* [83]. It also employs sampling, is integrated with the McPAT power model [110] and provides visualization support.

2.5.5 TPTS - Filtered Traces

TPTS (Two-Phase Trade-driven Simulation) [108] is a trace-driven simulator that includes techniques to model the performance of an out-of-order superscalar core using memory access traces. For this purpose, it generates memory access traces using Simplescalar and embeds for each memory access the number of cycles and instructions to the previous memory access and the dependencies with previous memory accesses. It uses the number of cycles to issue the memory access and the number of instructions to manage the size of the reorder buffer. Then, only memory accesses that are in the reorder buffer, considering the number of non-memory instructions in between accesses, are issued to memory if they do not have dependencies with pending memory accesses. This model is called *reorder buffer occupancy analysis* (ROA) [109] and assumes that the reorder buffer is the performance limiting factor of the processor core. This

assumption is based on the analysis of the performance of superscalar processors in a previous work [102].

The main purpose of TPTS is to explore cache hierarchy and main memory configurations while assuming the same core configuration as in the Simplescalar trace generation run.

They also extend their experiments to multithreaded applications. In this case, they assume statically-scheduled applications and synchronize threads on lock and barrier operations as explained in Section 2.3.

Another interesting feature of this work is that they employ *stripped* memory access traces [138]. They propose several ways to deal with the inaccuracies of using stripped memory access traces for multithreaded applications, but do not evaluate them in their work. We cover the concept of using stripped memory access traces for multithreaded applications and propose a technique to improve its accuracy in Chapter 5.

2.5.6 Others

MPSim [13] is an extension to the SMTSim single-core simulator [165] that adds the simulation of multiple cores and uses a trace-driven front-end for multi-programmed workloads. It has been used to simulate the Alpha and PowerPC ISAs. MPsim has been used as the multi-core simulator in a multi-scale simulation methodology for the simulation of large HPC applications running in supercomputers [89]. This methodology is validated against the MareNostrum supercomputer [3] showing an error within 33% for MPsim dual-core simulations of complex HPC applications.

PTLsim [181] is a full-system execution-driven simulator for the x86 ISA. It became popular because it was the only open-source x86 full-system simulator at the time of its release in 2006. It gets full-system support by integrating the Xen virtual machine monitor [26] and provides in-order and out-of-order core models. The accuracy of PTLsim was assessed against a real AMD K8 core showing an error withing 5% for the *rsync* application [181].

MARSS [133] is an extension of PTLsim to use QEMU [76] as the front-end for full-system simulation instead of Xen.

COTSon [18] is a full-system execution-driven simulator that uses the AMD SimNow emulator [28] as a front-end. It has shown a case to simulate 1000 cores by parallelizing simulation [122].

Turandot [125, 127] is a trace-driven multi-core simulator modeling a multi-core resembling the IBM POWER4. Turandot was validated [126] and shown to have a deviation within 5% for SPECint95. It provides a detailed power model [42] and it has also been parallelized [66]. In the parallel implementation, multiple simulated cores run on separate threads and synchronize on accesses to the shared L2 cache. It is reported to have a 1.5x speedup running on three threads. In this same work, they extend Simplescalar for simulating multiple cores and parallelize it using the same strategy.

CMP$im [97, 98] is a cache hierarchy and memory system simulator using PIN for functional simulation. Its focus is on memory behavior analysis of multi-cores running single-threaded, multithreaded or multiprogrammed workloads.

SESC [145] is an execution-driven simulator that uses MINT [168], a MIPS emulator, for functional simulation. It models an out-of-order core, caches and

interconnection network. It is claimed to be simple and fast (1.5 MIPS), and this has made many researchers to adopt it for their experiments.

2.6 Simulation in Major Conferences

We maintain a data base with statistics about the simulation methodology employed in research papers in the main computer architecture conferences, namely International Symposium on Computer Architecture (ISCA), the International Symposium on Microarchitecture (MICRO), the International Symposium on High Performance Computer Architecture (HPCA), and the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

In this section, we show some data concerning several aspects of the simulation methodology employed in papers published in these four conferences that are relevant to the work in this thesis. These aspects are: the simulation type, the size of the simulated machine in number of cores and the simulation tools employed.

2.6.1 Simulation Types

Figure 2.7 shows the percentage of papers published in main computer architecture conferences by the simulation type used in their experiments. We have classified papers using simulation in three categories: trace-driven, execution-driven and binary-instrumentation-driven. We also include the percentage of papers not using simulation. These papers are mainly software-only proposals or performance analysis works that carry out their experiments through measurements in real machines.

The majority of papers using simulation employ execution-driven simulators. This is mainly due to the limitations of trace-driven simulators for the simulation of multi-cores in terms of reproducing the dynamic behavior of multithreaded applications and operating systems (see Section 2.3). Most of the works using trace-driven simulation are for proposals focusing on the off-chip memory or interconnection network. In these works, they feed the timing model of the off-chip memory or the interconnection network with traces of memory accesses or network messages, respectively. Usually, these works require to simulate large numbers of threads or nodes to expose the system to contention. This prevents the use of execution-driven simulators due to their slow operation with increasing numbers of cores, and that is why trace-driven simulators are used for these works instead.

On the other hand, execution-driven simulators are used for multi-core studies involving processing core or cache hierarchy modifications. They are also used for reliability studies, for which it is necessary to have the application data in order to inject errors and see how those affect the functionality of the application.

It is also interesting to note that most works published in ASPLOS are software-only. In 2011 and 2012, more than 75% of the papers presented in the conference did not use any simulation at all.

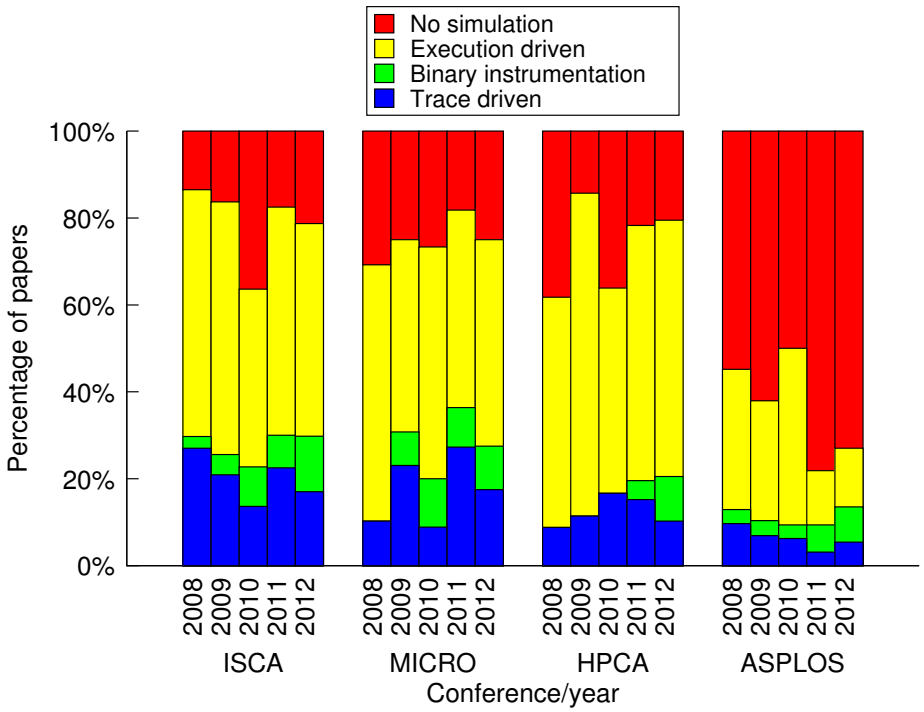


Figure 2.7: Breakdown of papers per simulation type in main computer architecture conferences from 2008 to 2012

2.6.2 Simulated Machine Size

Figure 2.8 shows the percentage of papers published in main computer architecture conferences by the number of simulated cores. We only include papers using simulation in their experiments. Also, in papers simulating several configurations with multiple numbers of cores, we just consider the largest configuration (maximum number of cores) for that paper.

The results show that around 40% of the papers simulate four cores or less (in ASPLOS around 60%). Nowadays, most existing desktop and server microprocessors include at least four cores. Even mobile microprocessors with four-cores are the norm for high-end cell phones and tablets. Also, around 80% of the papers simulate 16 or less cores, which is again in the range of cores of existing server processors. Only around 5% of the papers simulate large configurations with more than 64 cores.

While gathering these statistics, we found a clear differentiation between works using execution-driven and trace-driven simulation. Trace-driven simulation works are the ones using the more numbers of cores, generally more than 32, while execution-driven simulators are most of them limited to a maximum of 16 cores. In fact, there is a correlation between the percentages of papers by number of simulated cores with the percentage of papers by simulation type. In the years with more trace-driven simulation, there are more works simulating large configurations.

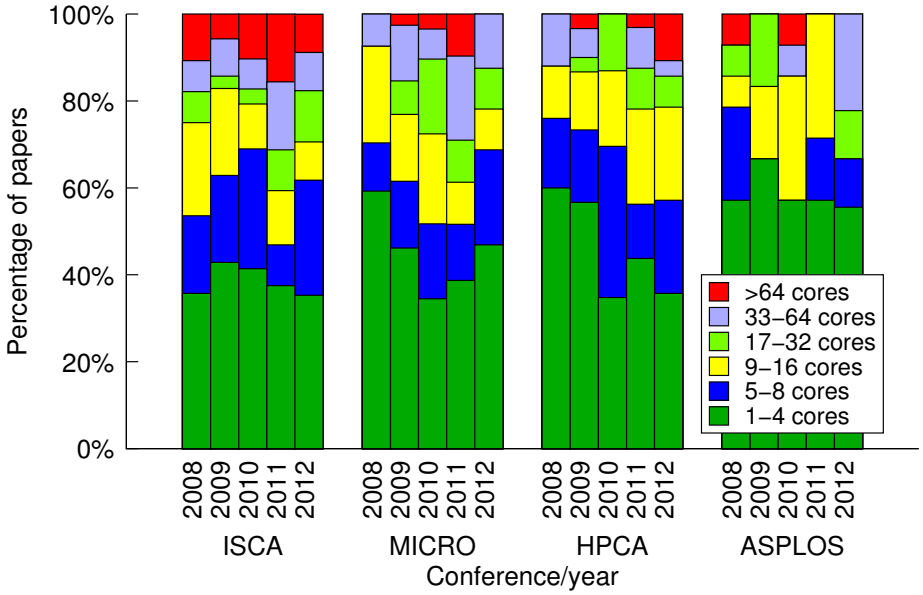


Figure 2.8: Breakdown of papers per maximum simulated number of cores in main architecture conferences from 2008 to 2012

This makes sense considering our previous discussion on the type of works for which the different types of simulators are employed. Off-chip memory and interconnection network studies require larger configurations if they want to account for contention effects when having an increasing number of cores sharing those shared resources. In the case of execution-driven simulation, techniques to improve processing core pipeline designs or cache management techniques (among many others) may be better analyzed in small configurations. However, those techniques must be considered to be an improvement of current existing processors as long as they are not assessed for larger configurations where higher contention or the lack of scaling of some components may offset or even reverse the benefits of those techniques.

2.6.3 Simulators

Figure 2.9 shows the percentage of papers in main computer architecture conferences by the simulation tool employed. We have included only papers using simulation, and have classified them in several categories considering the simulators presented in Section 2.5. For example, the *Simplescalar* label corresponds to works using Simplescalar and also works using simulators based on simplescalar, such as the ones mentioned in 2.5.1. We have included a category for the most used simulators, including the GPGPU simulator GPGPU-Sim [25], and then two other categores: *Other* and *In-house*. Under *Other*, we have aggregated the papers using simulators that have been published. *In-house* corresponds to simulators developed in the institution of the researchers publishing the work and that are neither published, nor disclosed for general use.

The figure shows that the most popular simulators are those based on Sim-

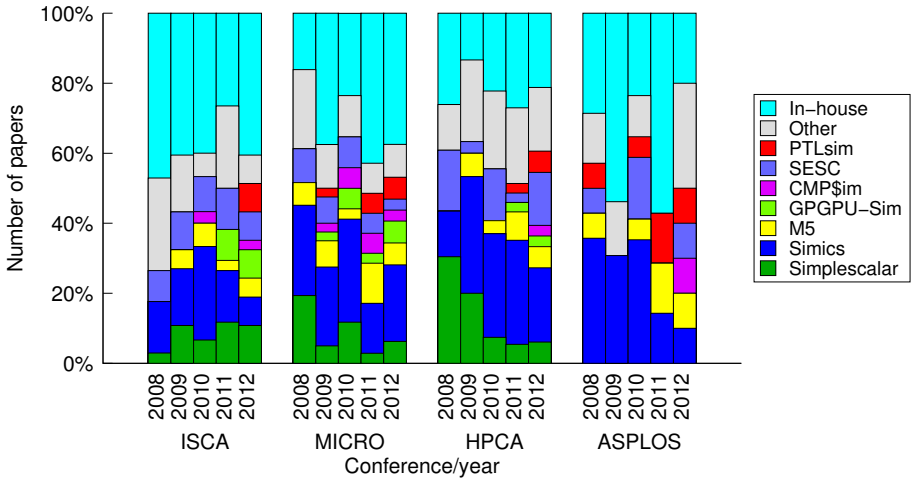


Figure 2.9: Breakdown of papers per simulation tool in main computer architecture conferences from 2008 to 2012

plescalar and Simics, accounting for between 20% and 40% of papers using simulation in these conferences. Moreover, there is a large portion of papers that use in-house simulators that also accounts for between 20% and 40%.

Most of the papers using Simplescalar are single-threaded experiments regarding pipeline enhancements and fault tolerance. Most of the papers using Simics, they do it together with the GEMS timing models. For the rest of works, the simulation tools employed are quite diverse, showing that computer architects have not yet found a one-size-fits-all simulator that can be used for different kinds of studies, simulated architectures and simulation setups.

2.7 Task-Based Programming Models

Shared-memory parallel programming models allow the programming of multithreaded applications by specifying which parts of the computation can be executed concurrently. Low-level programming models provide primitives to specify parallel work using semantics close to the ones used by operating systems to run multiple execution threads on multiple processing elements. For complex parallel applications, using those low-level semantics can be a productivity limiting factor. High-level programming models introduce abstractions that are more friendly for programmers to specify which computation can be executed concurrently.

A family of high-level programming models are those based on *tasks*. Task-based programming models introduce the concept of *task* as a means to define a piece of potentially parallel work that processes a specific set of data. These programming models are becoming popular because the concept of task is intuitive for programmers, which may increase productivity [163], and their dynamic scheduling of individual tasks yields better load balancing than statically-scheduled approaches for parallel loops.

Table 2.2: Task-based programming models

Programming model	Task Nesting	Task Dependencies	Automatic Cut-off
OmpSs [68]	Yes	Yes	Yes
StarPU [19]	Yes	Yes	No
Threading Building Blocks [144]	Yes	Yes	No
OpenMP 3.1 [7]	Yes	No	No
Cilk [80]	Yes	No	No
Sequoia [77]	Yes	No	No
X10 [57]	Yes	No	No
Chapel [55]	Yes	No	No
Charm++ [101]	No	No	No

Task-based parallel applications are partitioned into tasks that may execute in parallel. A task is typically a function or a block of code and it is not conceptually related to a specific execution thread. The runtime system software associated to the programming model assigns tasks to threads at run time. This scheduling is dynamic and makes decisions based on the number of threads and their availability. If a task needs to consume data produced by another task, it waits for the producer task to complete using synchronization primitives that usually allow it to wait for a specific task or for all previously created tasks. We refer to these scheduling and synchronization operations as parallelism-management operations or *parops* for short.

Table 2.2 shows a non-exhaustive list of programming models that support tasks. All these programming models provide constructs in the form of special keywords, data structures or compiler directives to define tasks that will be created at run time. Examples of these constructs for creating and synchronizing tasks are shown in Figure 2.10. It shows parallel implementations of the Fibonacci number recursive algorithm using three different programming models from Table 2.2: OpenMP, Cilk and Threading Building Blocks. OpenMP makes use of extended *pragma* annotations for tasks (in bold) and Cilk adds special keywords to the language (in bold). In both OpenMP and Cilk, *fib(n-1)* and *fib(n-2)* execute concurrently, and the calling task—*fib(n)*—waits for their completion on *taskwait* and *sync* respectively. Threading Building Blocks does not use language extensions and that results in a more complex piece of code. Threading Building Block's *spawn(b)* creates task *fib(n-2)* asynchronously, and *spawn_and_wait_for_all(a)* creates task *fib(n-1)* and waits for both *fib(n-1)* and *fib(n-2)* to complete.

All the programming models in Table 2.2, but Charm++, support that tasks create other tasks, which is referred to as task nesting. In Charm++, only the main thread creates tasks. Another feature of task-based programming models is whether tasks are defined independent or have dependencies. OmpSs, StarPU and Threading Building Blocks support task dependencies by managing a task-dependency graph at runtime similar to the way a superscalar processor manages instruction dependencies. That is, if a task needs to consume data to be produced by other pending tasks, it waits and is not scheduled until its dependencies are satisfied.

```
int fib(int n)
{
  int i, j;
  if (n<2) return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);
    #pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);
    #pragma omp taskwait
    return i+j;
  }
}
```

(a) OpenMP

```
cilk int fib(int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x+y);
    }
}
```

(b) Cilk

```
long fib(int n) {
    int sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;
}
class FibTask: public task {
public:
    const int n;
    int* const sum;
    FibTask(int n_, int* sum_) :
        n(n_), sum(sum_) {}
    task* execute() {      // Overrides virtual function task::execute
        int x, y;
        FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
        FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
        // Set ref_count to 'two children plus one for the wait'
        set_ref_count(3);
        // Start b running
        spawn(b);
        // Start a running and wait for all children (a and b)
        spawn_and_wait_for_all(a);
        // Do the sum
        *sum = x+y;

        return NULL;
    }
};
```

(c) Threading Building Blocks

Figure 2.10: Task-based implementation of the Fibonacci number recursive algorithm in (a) OpenMP 3.1, (b) Cilk and (c) Threading Building Blocks.

OmpSs also features a runtime-system mechanism to choose, at a task creation point, whether it actually creates the task or if it executes the task sequentially. This mechanism named *cut-off* [69] is useful when there are many pending tasks, all threads are busy and there is no need to open more parallelism, which would incur an unnecessary overhead. Cut-off can also be used in other programming models, but it is the programmers responsibility to implement separate serial and parallel versions of the task and call either of them depending on a user-defined heuristic. OmpSs provide this feature built-in, it

is automatic and the heuristics are configurable.

Regardless of the programming model features, task-based programs are composed of tasks and the parops to manage them. This is a feature that we exploit to enable the trace-driven simulation of multithreaded applications. See Chapter 3 for more details about this technique.

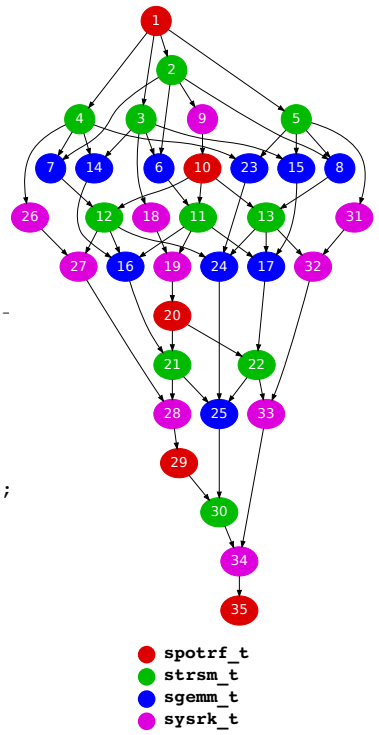
2.7.1 OmpSs

The OmpSs parallel programming model is an extension of OpenMP that incorporates the concepts of the StarSs programming model (CellSs [30] and SMPSs [136] are implementations of StarSs for the Cell/B.E. and SMP platforms respectively. As in OpenMP, OmpSs allows the definition of potentially parallel tasks on a sequential code using pragma annotations on functions or code blocks. Pragma annotations are extended with the *in*, *out* and *inout* clauses to allow the programmer to indicate the input (read-only), output (write-only) and input/output (read-write) data of the task. This allows the automatic resolution of task dependencies and the extraction of task-level parallelism at runtime.

The OmpSs toolchain is completed with the Mercurium source-to-source compiler [4] and the NANOS++ runtime system [6]. Mercurium implements the necessary transformations to compile OpenMP and OmpSs applications and generate code that calls NANOS++ parops.

```
#pragma omp task in(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M],
             float c[M][M]);

#pragma omp task inout(a)
void spotrf_t(float a[M][M]);

#pragma omp task in(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma omp task in(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);

-------------------------------------------

float A[N][N][M][M]; // NxN blocked matrix,
                     // with MxM blocks
for (int j = 0; j<N; j++) {
   for (int k = 0; k<j; k++)
      for (int i = j+1; i<N; i++)
         sgemm_t(A[i][k], A[j][k], A[i][j]);

   for (int i = 0; i<j; i++)
      ssyrk_t(A[j][i], A[j][j]);

   spotrf_t(A[j][j]);

   for (int i = j+1; i<N; i++)
      strsm_t(A[j][j], A[i][j]);
}
```

(a) OmpSs source code



(b) Task dependency graph

Figure 2.11: Cholesky decomposition of a blocked matrix using OmpSs.

Figure 2.11 shows an example of an application written in OmpSs. The application is a Cholesky decomposition of a blocked matrix. Figure 2.11a shows the source code including the task definitions (top) using *pragmas* including their *in* (read-only), *out* (write-only) and *inout* (read-write) task data dependencies. The task-generating code (bottom) executes sequentially creating new tasks at every function call. Figure 2.11b shows the task dependence graph of an execution of the Cholesky decomposition on a 5x5 blocked matrix. The graph shows tasks as nodes, and inter-task dependencies as the edges interconnecting nodes. Different colors correspond to different task types and the node numbers show the task creation order.

Chapter 3

Simulating Multithreaded Applications Using Traces

In this chapter we introduce our approach to enable the simulation of dynamically-scheduled multithreaded application using traces.

First, we explain the problem of capturing the behavior of a mulithreaded application in a trace and then trying to reproduce that behavior in a trace-driven simulation. Then, we cover the state of the art in this matter., which is limited to statically-scheduled applications.

We introduce a simulation methodology to address the problem for dynamically-scheduled applications. We also explain our implementation of this methodology and show some experiments that prove its feasibility and usefulness. Finally, we discuss the coverage and limitations of our approach and the research opportunities it opens for trace-driven simulation.

3.1 Problem

Multithreaded applications are parallel applications where all parallel execution streams access data in a common address space. These applications are written using shared-memory parallel programming models. As mentioned in Section 2.7, regardless of the programming model, multithreaded applications consist of *user code* and *parallelism-management operations*. The user code is the part of the application that defines its intrinsic behavior regardless of whether the application is executed in parallel or not. The parallelism-management operations, or *parops* for short, is the part of the code that manages the synchronization of multiple threads and schedules work to be executed concurrently.

One of the properties of trace-driven simulation is that the application behavior is statically captured in a trace file and does not change during the simulation of multiple machine configurations. This is a problem for simulating multithreaded applications due to the dynamic behavior caused by timing-dependent operations. The behavior of a timing-dependent operation changes for different machine configurations, something that a trace-driven simulator can not reproduce.

Figure 3.1 shows the execution of a multithreaded application with a mutual exclusion. Dashed lines mark the user code execution that is split in several

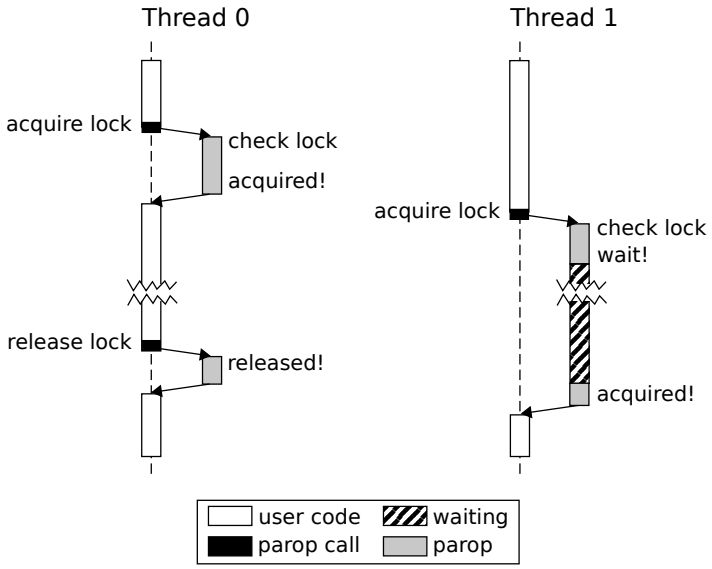


Figure 3.1: Execution of an application with a mutual exclusion using two threads.

fragments shown as white boxes. The user code execution is interleaved with the execution of parops, shown outside the dashed lines. In this example, the executed parops are for acquiring and releasing a lock before and after the execution of the critical section. The small black boxes show the calls to these operations. Thread 0 starts executing until it reaches the point where it needs to acquire a lock to enter the critical section. At that point, the operation to acquire the lock executes, it actually gets the lock, and Thread 0 enters the critical section. Afterwards, Thread 1 tries to acquire the lock but it has to wait for Thread 0 to release it. When Thread 0 finishes the critical section, it releases the lock. Then, Thread 1 acquires it, and enters the critical section.

Let us say the execution in Figure 3.1 happens during a trace generation run and it is captured in a trace file. Then, during simulation, Thread 0 will always acquire the lock and Thread 1 will wait regardless of which of them arrives first to the critical section. Therefore, the effects of different machine configurations on which thread acquires the lock first, which one waits and for how long it waits can not be reproduced using trace-driven simulation.

For statically-scheduled applications, timing-dependent operations include locks and barriers. The variability of these operations relates to whether the thread must wait and for how long it has to wait. However, for dynamically-scheduled applications, timing-dependent operations are more complex. They include operations to create parallel work and heuristics to scheduled and assign that parallel work to available threads depending on the application and machine states.

We cover existing works that manage to reproduce the dynamic behavior of statically-scheduled multithreaded applications in the next section. In the rest of the chapter, we explain our approach to address the problem for dynamically-

scheduled applications which, to the best of our knowledge, no other work has done before.

3.2 State of the art

In the 1980s and early 1990s, trace-driven simulation was widely-used in computer architecture research. The problem explained in the previous section already applied at that time, but it was in the context of shared-memory multiprocessors.

Goldschmidt and Hennessy [88] compared trace-driven simulation and binary-instrumentation-driven simulation in terms of accuracy for multithreaded applications running on shared-memory multiprocessors. They conclude that trace-driven simulation is accurate as long as there are not timing dependencies, precisely the problem we explain in the previous section. In statically-scheduled applications, they identify timing dependencies in locks and barriers. Then, to model the different behavior on different configurations, they include a lock or barrier event in the trace. When the simulator finds one of those events, it emulates its behavior depending on the state of the simulated machine. An ad-hoc implementation of lock and barrier operations carries out the necessary busy-waiting loop iterations until the synchronization condition is satisfied. This assumes the implementation of all locks and barriers in the application is the same. However, for dynamically-scheduled applications, they do not manage to isolate the synchronization and scheduling timing-dependent code and conclude that trace-driven simulation is inaccurate in such cases.

Other works [105, 82, 108] employ the same technique and, in all cases, they only consider statically-scheduled applications and use an ad-hoc implementation for locks and barriers as done in Goldschmidt and Hennessy's work. A recent work [183] proposes the same technique for reproducing locks and barriers without citing any of the previous papers. Their novelty is the annotation of locks and barriers to force the same execution order as in the original trace-generation run to provide deterministic replay.

3.3 Methodology

We developed a simulation methodology [150] to properly reproduce the dynamic behavior of dynamically-scheduled multithreaded applications during simulation, and overcome the problem explained in Section 3.1. This methodology targets runtime-managed multithreaded applications and is based on their structure.

As mentioned in previous sections, multithreaded applications consist of user code and parops. User code is generally timing independent. To illustrate this concept, let us define user code as a set of *sequential sections*. A sequential section is a piece of code that executes the same instructions and in the same order regardless of the machine configuration. This means that instruction i (in sequential order) always executes after instruction *i-1* and before instruction *i+1*. Examples of sequential sections are an iteration in an OpenMP parallel loop, a Cilk parallel routine and an OmpSs task.

Parops determine the order and timing in which sequential sections are executed. These include, but are not limited to: thread management operations (creation/spawn/join), synchronizations (locks, barriers, point-to-point), parallel work scheduling, and transaction begin and commit in transactional memory systems. These operations are timing dependent because the instructions they execute depend on the machine configuration and state.

As an example, the sequential code section of each thread in Figure 3.1 maintains its timing-independent order regardless of being interleaved by the execution of parops. However, the execution of the lock acquire parops actually varies on the machine state, more precisely on the lock variable state: whether it is free or busy.

Given these properties, our methodology establishes that user code is simulated using a trace-driven approach, and parops are executed at simulation time based on the state of the simulated machine.

3.3.1 Tracing

The application instrumentation for trace generation is aware of the application structure instead of blindly recording the instruction stream. First, the instrumentation detects the entry and exit points of sequential sections in order to store each of them separately. At the same time, instead of capturing the execution of parops, the instrumentation discards it and includes a *parop call* event for each of them. The event is stored at the point where the parop was executed, and includes all necessary information to reproduce its execution at simulation time. For example, for a lock acquire operation, the event would include the address or identifier of the lock variable to synchronize with.

The information included in a trace file is:

- Information to feed the timing model for each sequential section separately. For example, list of instructions or memory accesses.

- Parop call events including the information to carry out their execution at simulation time.

Application traces are indexed to enable the individual access to the information of each sequential section. This allows a simulated thread to switch to run a new sequential section or resume the execution of a suspended one when the execution of a parop carries out some scheduling operation. In the example in Figure 3.1, the corresponding trace would include the two sequential sections (denoted with the dashed line), and one event for each lock acquire and release calls (black boxes) including the lock identifier for each of them.

Using this tracing method, an application is fully represented in a single trace, and it is not necessary to generate a trace for every number of threads to be simulated. In other words, one trace is generated per application and is used to simulate the execution of that application over multiple machine configurations with different numbers of cores. This results in a significant reduction in required disk space and tracing time compared to the state-of-the-art approaches requiring a separate thread for every number of cores to be simulated.

3.3.2 Simulation Infrastructure

Having these traces, the simulation tool requires to access to an implementation of the required parops. In parallel programming models, parop implementations are typically provided as part of a runtime system library. In our methodology, the programming model runtime system is integrated in the simulation infrastructure and, in order to work as decoupled as possible, a clearly-defined interface is required to provide the necessary services to both the simulator timing model and the runtime system.

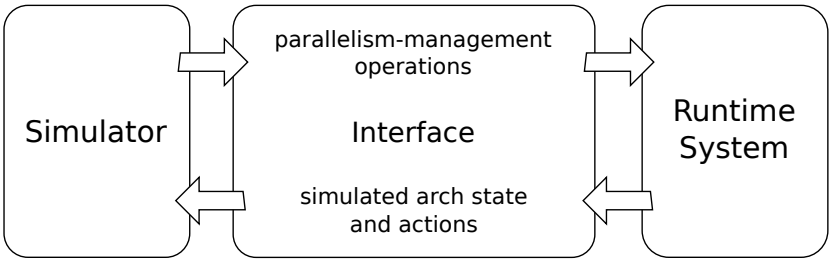


Figure 3.2: Simulation infrastructure scheme.

Figure 3.2 shows a scheme of the simulation infrastructure. It has three components: the trace-driven simulation engine, the runtime system and the *engine-runtime interface*. The trace-driven simulation engine includes the simulated architecture timing model and the front-end to read the application trace. The runtime system is the one associated to the programming model of the simulated application. The engine-runtime interface has two parts. The first (shown at the top of the interface box in the figure) exposes and interface to the runtime system parops for the simulation engine to invoke them at simulation time. The second (shown at the bottom) are the architecture dependent operations that the runtime system requires for the execution of parops. An example of these operations is the creation and join of threads, such as fork/join in OpenMP parallel loops.

3.3.3 Simulation Process

A simulation using multithreaded application traces in this infrastructure runs as follows. The simulator begins the simulation of the starting thread in the application, commonly known as the *main* or *master* thread. All remaining treads start idle. The simulator performs the timing simulation of the user code in the main thread until it reaches the first parop event in the trace. At that point, the simulator invokes the parop execution in the runtime system through the runtime system interface. The corresponding parop is then executed based on the state and characteristics of the simulated machine. This is different to approaches using binary instrumentation, where functional simulation is based on the host machine state and characteristics, and to traditional trace-driven techniques where parop execution depends on the state and characteristics of the machine where the trace was collected. As an example of our simulation adaptability, scheduling decisions carried out by our simulation infrastructure

are based on the number of cores of the simulated architecture, the core status (running/wait/idle), or the overall work load at a given point in simulation time.

Then, after dynamically simulating a given parop, simulated threads may have been assigned new sequential sections according to the runtime system decisions. Each simulated thread then proceeds with the timing simulation of the sequential section assigned to it. For instance, when a parallel Cilk function is invoked, the corresponding parop is called. According to the Cilk *work-first* scheduling policy, the calling thread switches to execute the parallel function, while the calling context is switched to an available thread, or suspended until there is one available. Using our methodology, the simulation proceeds in the same way as in the real machine: it assigns the parallel-function sequential section to the calling thread, and the calling context is either assigned to an available simulated thread, or is suspended until some simulated thread becomes available. This operation is possible because different sequential sections are captured and stored separately in the trace file and can be accessed individually to be simulated by different threads in the simulated machine.

3.4 Implementation

We implemented the proposed methodology using the TaskSim trace-driven simulator (see Section 2.2), and the OmpSs runtime system called NANOS++ [6] (see Section 2.7.1). This implementation supports multithreaded applications written in the programming models supported by NANOS++, which currently are OpenMP and OmpSs.

In this environment, the target applications are compiled using the Mercurium source-to-source compiler. The resulting binary calls an instrumented version of NANOS++ that generates traces for TaskSim. The resulting traces are simulated on TaskSim, and NANOS++ is used as the runtime system for the execution of parops at simulation time.

3.4.1 Instrumentation

NANOS++ incorporates an instrumentation engine that can be used to generate traces with a custom trace format via the implementation of plugins that are dynamically loaded at runtime. For our implementation, we developed a NANOS++ plugin to capture the runtime events of interest and generate traces for TaskSim. Additionally to the captured events, instruction or memory access traces are generated using PIN [113]. Since all parops are encapsulated in the runtime system, the instrumentation captures all the necessary information for simulation.

The user-code sequential sections in this environment correspond directly to NANOS++ tasks. The main execution thread, which is a sequential section in itself, is treated as a NANOS++ task as well. In the rest of this section, we refer to sequential sections as *tasks*, and to the main-thread associated task as *main task*.

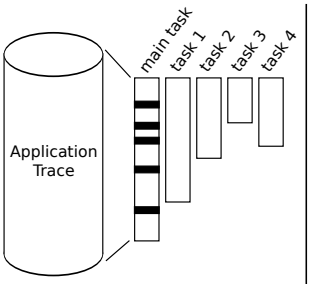
Figure 3.3 shows an example of an OmpSs application and a scheme of the corresponding trace. The application creates four tasks and passes a separate matrix block to each of them as an argument. Afterwards, the main task synchronizes with the four created tasks using the *taskwait* directive before printing

```
#pragma omp task inout(mat)
void t1( float mat[N][N] );

#pragma omp task inout(mat)
void t2( float mat[N][N] );

#pragma omp task inout(mat)
void t3( float mat[N][N] );

#pragma omp task inout(mat)
void t4( float mat[N][N] );
```

(a) Task definition code

```
float matrix[4][N][N];

int main( int argc, char* argv[] )
{
    //...
    t1( matrix[0] );
    //...
    t2( matrix[1] );
    //...
    t3( matrix[2] );
    //...
    t4( matrix[3] );
    //...
        #pragma omp taskwait

    //print matrix contents
}
```

(b) Main task code



main task trace

```
CPU           user_code, main0ins
create_task   id_t1, inout(0x72480,16384)
CPU           user_code, main1ins
create_task   id_t2, inout(0x76480,16384)
CPU           user_code, main2ins
create_task   id_t3, inout(0x7a480,16384)
CPU           user_code, main3ins
create_task   id_t4, inout(0x7e480,16384)
CPU           user_code, main4ins
task_wait
CPU           user_code, main5ins
```

task4 trace

```
CPU           user_code, t4ins
```

t4ins trace

```
...
20: sub    r15, r12, r13
24: store  r35, r15 (0x7e6a0)
28: sub    r3, r31, r4
2C: load   r21, r7 (0x80a88)
30: addi   r3, r3
34: beq    r3 (next_i: 7C)
7C: mul    r32, r8, r9
80: mul    r33, r10, r11
84: mul    r34, r12, r13
8C: store  r32, r17 (0x7f280)
90: store  r33, r18 (0x7f284)
...
```

(c) Trace contents

Figure 3.3: OmpSs application example and its corresponding traces for the TaskSim – NANOS++ simulation platform.

the contents of the matrix.

A trace of each task instance is captured separately and includes all parop-call events as they originally occurred. In the main task, there is one parop event per task creation (on each function call) and one parop event for the *taskwait* synchronization. The information related to each parop call (function arguments) is also included in the trace, so they can be properly re-executed at simulation time. Each `create_task` event includes the *id* of the created task and the address, size and directionality (inout, see Section 2.7.1). Computation periods between parops are specified as `CPU` events and include a pointer to the associated instruction or memory-access trace. As an example, the trace of task `t4` has one `CPU` event that points to an instruction trace including all instructions executed by the task. As previously explained, all task event traces and all instructions or memory-access traces are indexed to be accessed individually.

3.4.2 Runtime Integration

Figure 3.4 shows a scheme of the components of our implementation. The interface between TaskSim and NANOS++ is split in two parts: the *runtime bridge* and the *architecture-depenedent operations*. The runtime bridge exposes all necessary parops implemented in NANOS++. This allows TaskSim to invoke them every time it finds a parop-call event in the trace. The architecture-dependent operations allow NANOS++ to manage the simulated threads and to assign tasks for them to execute.

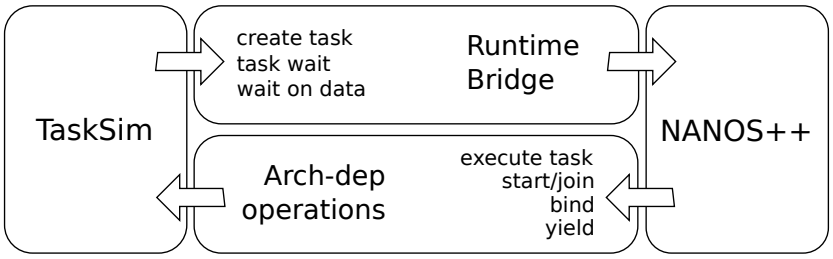


Figure 3.4: Implementation scheme.

Figure 3.4 includes a list of the main parops and architecture-dependent operations used in our implementation. The main parops exposed by the runtime bridge (shown alongside the corresponding NANOS++ function between parenthesis) are the following:

- create task (createWD): the calling task requests the creation of a new task. It specifies the new task id and the input and output data identifiers (address and size) for dependence tracking. The new task may be submitted for execution in any thread, or the cut-off mechanism (see Section 2.7) may decide to execute it *inline*.
- task wait (waitCompletion): the calling task requests to wait for all its previously-created tasks to complete execution. No additional information is required.
- wait on data (waitOn): the calling task requests to wait for the last producer of a specific piece of data to complete execution. It specifies the corresponding data identifiers.

These operations cover the basic task functionality of OmpSs and OpenMP applications.

Similarly, TaskSim exposes the simulated architecture to NANOS++ through the architecture-dependent operations. These architecture-dependent operations allow NANOS++ to assign tasks to simulated threads and set their state (idle/running). This architecture-dependent part of the interface is implemented as a NANOS++ plugin (as it is done for the architecture-dependent operations of different real machines) and includes the following operations (alongside the corresponding NANOS++ function between parenthesis):

- execute task (switchTo/inlineWork): executes a task on the current thread. It can be done on a separate context (switchTo) or on the current context (inlineWork).

- start/join thread (start/join): starts/finishes a new thread.
- bind thread (bind): binds the current thread to a hardware simulated thread.
- yield thread (yield): requests a thread to yield.

Additionally, other architecture-dependent operations provide support for explicit data transfers on distributed shared memory architectures, such as systems including local/scratchpad non-coherent memories (e.g., GPU, multimedia, Cell/B.E.).

For the aforementioned functions to work, a set of data structures representing the several machine threads at the runtime-system layer needs to be created at startup. The simulator engine calls the runtime system to initialize these data structures through the *init* runtime-bridge function with the suitable target architecture features, such as the number of cores and core types (in heterogeneous scenarios). At this point, the runtime system starts and initializes the generic data structures that are common for any underlying architecture. Then, in order to initialize the architecture-dependent data structures, it invokes the implementation of the *createThread* architecture-dependent operation.

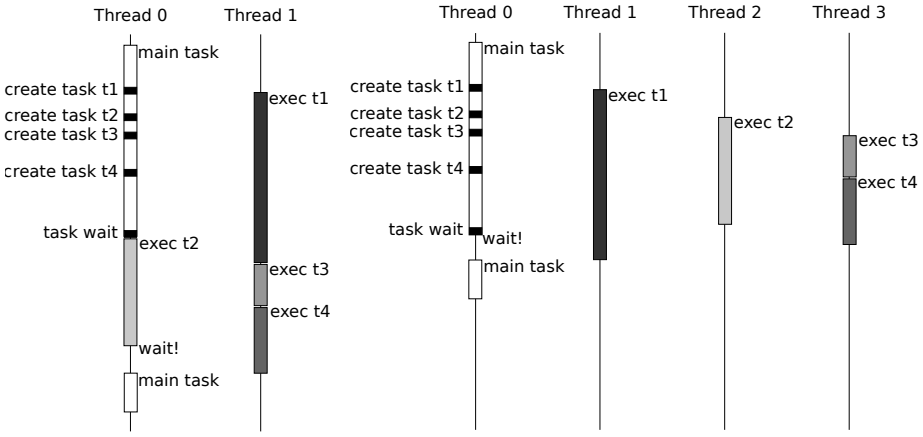
This pre-simulation procedure provides the runtime system with the necessary abstraction layer to represent the threads in the simulated architecture. Thanks to this fact, and the decoupling between the parops and the user code, no data needs to be stored in the trace; and, as a result, the data structures resulting from the runtime system initialization are sufficient for the correct functioning of the previously-listed operations at simulation time.

During simulation, NANOS++ has the illusion of running on a real machine, so tasks are scheduled, executed and synchronized in the same way as in a real execution on whatever architecture configuration is simulated.

To apply this methodology to other programming models, we need all parops to be encapsulated in the runtime system. Then, how easy is to integrate the runtime system with the trace-driven simulator depends on how easy it is to setup the interface between them. The runtime bridge, which follows the *bridge* software pattern, is easy to program as long as there is a clear API for the available parops in the runtime. For the architecture-dependent operations, the ease of implementation depends on how spread out are these in the runtime system code. In NANOS++, all architecture-dependent code is encapsulated in a separate library that can be loaded at runtime. Then, for our implementation we just had to replace the architecture-dependent library of the real machine by one implementing the operations for TaskSim.

3.4.3 Simulation Example

Simulation starts by assigning the main task to one of the simulated threads. All other simulated threads start idle. A separate context (user-level thread) is created for each simulated thread, to provide NANOS++ with the illusion of a multithreaded machine, even though TaskSim is running on a single thread. Then, whenever a simulated thread wants to call a parop, the simulator switches to the context associated to that simulated thread before entering the runtime system execution.



(a) Two threads (b) Four threads

Figure 3.5: Scheme of a multithreaded application simulation that generates 4 tasks and waits for their completion on two and four threads.

Figure 3.5 shows the simulation of the application in Figure 3.3, using two (Fig. 3.5a) and four simulated threads (Fig. 3.5b). Having all simulated threads setup, Thread 0 starts the simulation of the main task until it encounters the first `create task` event. Then, it calls the corresponding parop through the runtime bridge, which results in the creation of task `t1` in NANOS++. Task `t1` is then ready and is scheduled to Thread 1 that is idle. Later, the main task also creates `t2`, `t3` and `t4`. These are executed by different threads depending on the available cores in both examples, following a breadth-first scheduling policy. After creating the four tasks, Thread 0 finds a `task wait` event. At that point in the example with four cores, all created tasks are already *running* in other threads, so Thread 0 waits for their completion. In the example with two cores, only `t1` is *running*, so Thread 0 starts `t2`. Thread 1 finishes `t1`, and then switches to `t3` and later to `t4`. Thread 0 completes `t2`, and switches back to the main thread where it waits for `t4` to complete.

This example shows how our simulation methodology allows the simulation of different hardware configurations using a single trace per application.

3.5 Experiments

In this section we show a set of experiments that demonstrate the feasibility and usefulness of the presented methodology, using our implementation shown in the previous section.

The first experiment is the scalability evaluation of a blocked-matrix multiplication where each matrix block is processed by an OmpSs task. The computation is done using the *sgemm* function in the BLAS library [37]. Figure 3.6 shows its scalability (speed-up with respect to the execution with 8 cores), labeled as *matmul*.

As it can be seen, matmul performance does not scale beyond 32 cores. The analysis of the results showed that the bottleneck is the task generation rate in

the main task, a limitation that has been studied in previous works [153]. This issue leads to core underutilization, as it is shown for the 64-core experiment in Figure 3.7a. Each row in the figure corresponds to one core: gray-colored horizontal bars show task execution and white regions are idle time. The first row at the top of the figure is the main task, which is the only solid line in the figure. Although the main task is continuously creating tasks (shown in black), it is not fast enough to generate tasks at a sufficient rate to feed all 63 remaining cores. This results in long idle periods (in white) where cores are waiting for tasks.

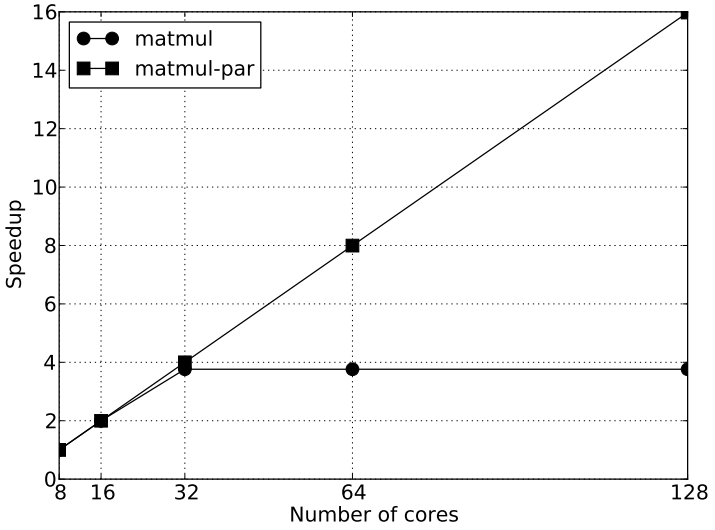


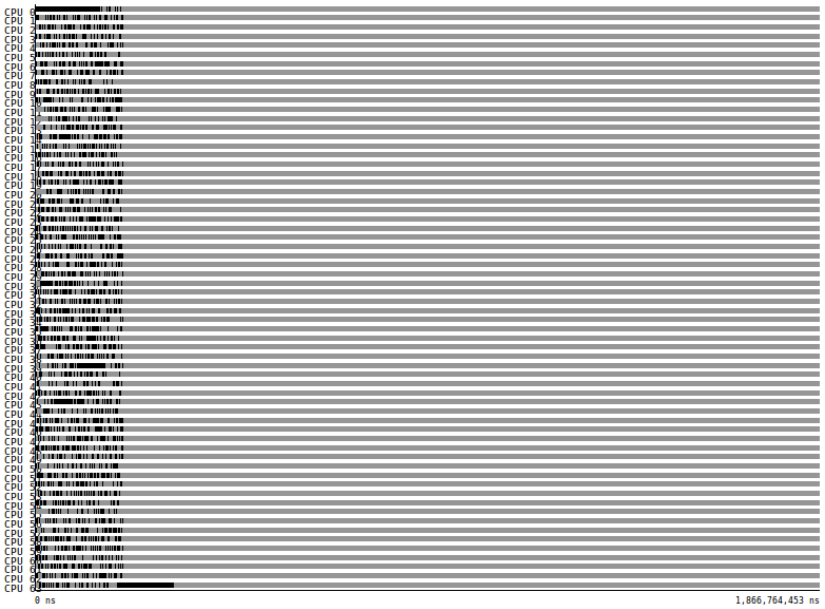
Figure 3.6: Speed-up for different numbers of cores from 8 to 64 with respect to the execution with 8 cores.

We modified the application to parallelize task generation using a hierarchical task spawn scheme: the main task creates a set of task-spawn intermediate tasks, each of them creating a subset of the total matrix-multiplication tasks. Figure 3.6 shows that the matrix-multiplication version with hierarchical task spawn (labeled as *matmul-par*) perfectly scales up to 128 cores. Figure 3.7b shows the core activity for matmul-par. All core rows show that all cores are busy all the time(no idle periods), thus fully utilizing the available resources in the architecture. The black regions showing the task generation phases are now spread across all cores.

This first experiment shows that the methodology allows trace-driven simulations of multithreaded applications for different number of cores using a single trace per application. It demonstrates that it is useful for finding bottlenecks in the application and runtime system that appear on large numbers of cores not yet available in real machines.

The second experiment is the comparison of different area-equivalent multicore designs. The different machine configurations follow Pollack's Rule [38], in which the performance increase of a core design is roughly proportional to the square root of the increase in complexity (core area). The baseline configuration

(a) matmul



(b) matmul-par

Figure 3.7: Snapshot of the core activity visualization of a blocked-matrix multiplication a with 1 task-generation thread, and b with a hierarchical task-generation scheme

in this experiment includes 16 cores. Then we explore two alternatives: one with four cores twice as fast, and another containing 64 cores with half the performance each.

Figure 3.8 shows the speed-up with respect to the baseline configuration using 16 cores. The results in the figure are shown for a set of scientific applications programmed in OmpSs. A single trace per application is used for the simulation of all hardware configurations. The configuration with four cores gets better performance for applications with less parallelism, as individual tasks will execute faster and they do not get any benefit from having more cores available. This is the case of *pbpi* and *stap*, which are limited by the performance of the main task, and they do not scale to 16 cores. In this case, doubling the core performance, doubles the overall application performance, as the sequential bottleneck that dominates execution is completed twice as fast.

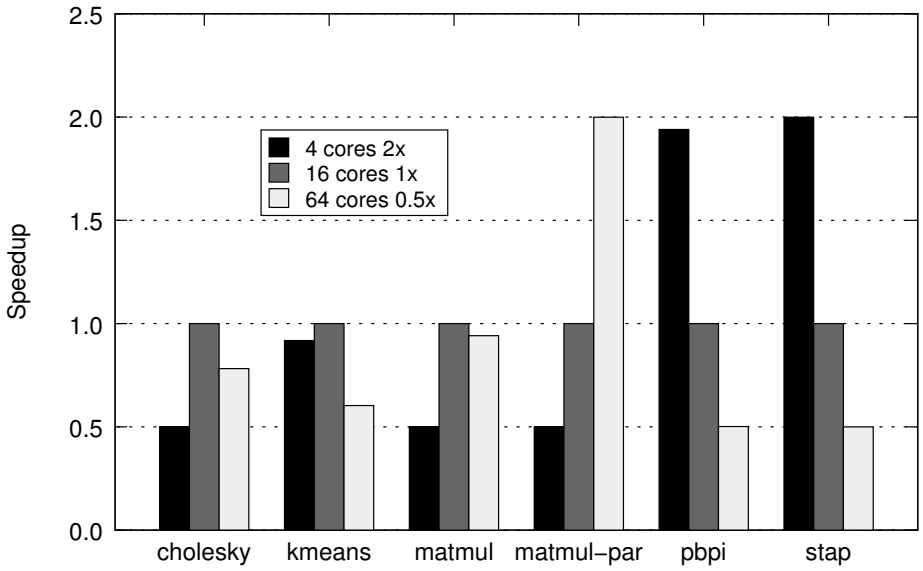


Figure 3.8: Application performance comparison for area-equivalent multi-core configurations. Speed-up with respect to the 16-core baseline.

The configuration using 64 cores performs better for highly-parallel applications, as the benefit of executing more tasks in parallel compensates the individual task performance loss. In our experiments, this applies to applications that scale up to 64 cores, which is the case of *matmul-par*. For the rest of applications, the baseline configuration is the best option. *Cholesky*, *Kmeans* and *matmul* scale to 16 cores, so this configuration outperforms the one using just four cores twice as fast. However, they do not scale to 64 cores, thus having an overall performance loss due to the lower throughput of the individual cores.

For these experiments, our simulation framework required one trace per application and one simulation per configuration. The purpose of the experiment is twofold. First, to show the ability of our implementation to reproduce the applications dynamic behavior for several configurations varying the number of cores and the core type/performance. And, second, to show that the infrastructure allows to get an insight of the reasons for application scalability on different configurations.

3.6 Coverage

The proposed methodology covers multithreaded applications where timing-dependent code is decoupled from the application actual computation. Low-level programming models (such as *pthreads*) allow the user to mix parops with user code. Such applications would have to be modified so that parops are decoupled from user code, and parops can be exposed to the simulator. This needs to be done for each parop and every application which is actually impractical. This results in a limitation of the proposed methodology, that is directly inherited from the mentioned weakness of low-level programming models.

Also, non-deterministic applications that rely on random values (such as Monte Carlo algorithms), include race conditions or depend on machine characteristics (e.g., data structures depend on number of cores) may not be reproducible during simulation. All required computation paths may have not executed on the original run, and thus are not available in the trace, making its trace-driven simulation not doable.

Therefore, we have identified high-level programming models as the best cases for our methodology. Since parops are fully provided in the associated runtime system, user code and parops are decoupled. Therefore, the instrumentation of the runtime system covers the generation of traces for applications in that programming model. Also, the runtime system of these programming models can be directly used as the runtime system in the simulation platform, just by adding the corresponding architecture-dependent operations as explained in Section 3.4.

This simulation methodology opens several research opportunities that are not practical using state-of-the-art trace-driven simulation.

First of all, new ideas on runtime system design, implementation and optimization, such as new scheduling policies or more efficient management of thread/task data structures, can be evaluated on future parallel architecture designs before the actual hardware is available. These evaluations are possible thanks to the interface described in Section 3.3, which provides runtime systems with the illusion of being executed on a real machine. Machines with large numbers of homogeneous cores (64-128) accessing a shared memory are nowadays available, although their cost is a limiting factor for its accessibility. The presented framework allows runtime system development for such large-scale machines.

An interesting area of research is scheduling for heterogeneous architectures. The presented infrastructure allows the evaluation of runtime scheduling techniques for such heterogeneous on-chip architectures, which is not possible using state-of-the-art trace-driven tools, and more difficult for execution-driven approaches as they may require a full compilation toolchain for the heterogeneous simulated architecture sometimes requiring support for multiple ISAs. The trace-driven nature of our tools allows to have ISA-independent traces and just needs to simulate the timing of the different core types.

Power-aware scheduling for large-scale architectures is another interesting research area, which is explorable using our methodology. Multithreaded applications may trade off performance with power savings for non-critical code regions, resulting in non-significant overall performance degradations. Analyzing those trade-offs is, again, not doable for models on existing trace-driven simulators, and may be hard using measurements in real machines as the access

to power sensors is in many cases limited (only a set of sensors are readable at a given point in time) or restricted (only available at the hypervisor level).

The presented methodology also provides dramatic reductions in the tracing time and effort. A single trace represents the intrinsic behavior of the application, and it is not necessary to generate traces for different architecture configurations of the target parallel system. Applications are traced once, and used to evaluate many configurations. Also, there is no need to regenerate traces when there are changes in the runtime system. Whenever a new feature or optimization is applied to the runtime system implementation, it is immediately available to the simulation framework, as the runtime system library is used unchanged.

The time and effort required for tracing applications has been a major issue in research based on trace-driven simulation so far. Previous works had to generate a separate traces per application and for each number of cores to be simulated. Thus, by minimizing the required tracing time and effort, new multithreaded applications can be quickly traced and incorporated as evaluation benchmarks to research projects. Also, our methodology reduces the required disk space (a limitation explained in Section 2.3). This way, more applications or longer executions can be stored in the same disk space.

3.7 Summary

The simulation methodology presented in this chapter is, to the best of our knowledge, the first hybrid methodology combining traces and execution for parallel application simulation. Traces are generated for the timing-independent user code, and parops are re-executed at simulation time.

One of the contributions of this methodology is a significant reduction in the effort for generating traces and disk space requirements. Multithreaded applications are represented by a single trace, that can be used to evaluate multiple hardware configurations. Also, by using multithreaded applications written in high-level programming models, the instrumentation of the corresponding runtime system is enough to generate traces for any application.

The use of traces in our methodology provides all the benefits of trace-driven simulation explained in Section 2.3. We showed simulations with up to 128 cores that completed in a few minutes thanks to the lack of functional simulation and to raising the level of abstraction. More details about this can be found in the next chapter.

Chapter 4

Multiple Levels of Abstraction

The techniques to reduce simulation time explained in Section 2.4 are useful to reduce the simulation time of cycle-accurate simulators. These simulators are fundamental tools for computer architecture research in order to understand the workload stress on microarchitectural structures, but their time-consuming operation limits their usability to exploring multi-cores with a few tens of cores. In general, cycle-accurate simulators are not suitable for simulating large-scale architectures, nor are they actually meant to be used for this.

Early high-level design decisions, or evaluations of the cache hierarchy, off-chip memory, or interconnect may not need such cycle-level understanding of the architecture, as microarchitecture decisions in those cases may be irrelevant or even misleading. As it is widely suggested by the research community [180, 39], this fact opens the door to raising the level of abstraction of the core model. Furthermore, it allows to raise the level of abstraction of the application representation, as it does not necessarily have to be represented as an instruction stream. Most existing simulation tools are tied to this level of application representation: execution-driven simulators need to work with the application instruction stream for functional simulation (see Section 2.3), as well as trace-driven simulators that model the core pipeline.

In this chapter, we evaluate the implications of raising the level of abstraction of the simulation model by raising the level of abstraction of the simulated application. We categorize simulation levels of detail and introduce a definition of application abstraction levels. In this effort, we focus on the more *abstract* end of the spectrum, which is necessary for simulating large-scale architectures and where, to date, comparatively little work has been done. Based on the proposed application abstraction categories, we introduce two very high-level simulation modes that are faster than functional simulaton, and that actually provide more insight. These are complemented with two more detailed simulation modes: one simulating only memory accesses, and another working at the instruction level. We discuss and evaluate the utility and accuracy of these simulation models and analyze their trade-offs between simulation speed and model detail compared to the abstraction levels of popular simulators in computer architecture research.

4.1 State of the art

The idea of variable-grain simulation detail is already present in the literature and, in fact, many existing simulators offer several levels of abstraction with different simulation speeds and levels of detail. Having different abstraction levels is beneficial, not only due to having a faster (usually higher level) or more accurate simulation (usually lower level), but because each of them allows the researcher to reason about certain aspects of the design.

The abstraction levels in many existing simulators range from microarchitectural pipeline modeling of, for example, in-order and out-of-order cores, to functional simulation. Examples are popular execution-driven simulators such as Simplescalar [43], Simics [114] and gem5 [33]. These simulators provide a simulation mode for functional simulation, an intermediate mode that abstracts the core pipeline and simulates the memory hierarchy in detail, and the most detailed mode that simulates both the memory hierarchy and the core pipeline in detail. We cover these simulation modes and their simulation speed and modeling detail trade-offs in Section 4.4.

These aforementioned simulators target cycle-accurate simulation and their detail operation comes at the expense of simulation speed. However, other simulators [128, 24, 164] use event-driven simulation and high levels of abstraction to achieve early design decisions and high-level insights for architectures which cycle-accurate simulation is unfeasible.

An example of this kind of high-level computer architecture simulators is Dimemas [24]. It models distributed-memory clusters with up to thousands of cores, and its main target is parallel application analysis. Dimemas simulations mainly focus on reflecting the impact of different cluster node configurations and interconnection qualities on MPI communications. For this purpose it abstracts the simulation of the cores assuming it has the same performance as in the trace-generation machine (or a performance relative to it) and models MPI communication for a proper network contention simulation. A similar approach is followed by SMPI [63]. SMPI also models a cluster architecture, but the delay of MPI messages is calculated using an analytical model that considers not only transfer rates but also communication protocol delays.

Analytical models have also been used to abstract the model of the core pipeline. Interval simulation [83] is an approach in this direction. It uses a simple model that computes the IPC using an analytical model at intervals of a given size. The analytical model assumes a fixed IPC based on the issue width of the core and considers the IPC drop for an interval when there is a cache miss or branch misprediction.

Another way of raising the level of abstraction is to abstract the simulated machine model while raising the level of abstraction of the application representation. We advocate for this approach, provide a definition of a hierarchy of levels of application abstraction and evaluate the corresponding simulated machine abstractions in the following sections. Previous simulators for trace memory simulation [35, 167, 138, 173, 105, 88, 109] use an application representation at the memory level that considers memory accesses only.

In this chapter, we evaluate from the MPI-level abstraction used by Dimemas, to the memory level of trace memory simulators, and to the instruction level used by execution-driven simulators. Also, we introduce a fourth abstraction level leveraged from our simulation methodology presented in Chapter 3.

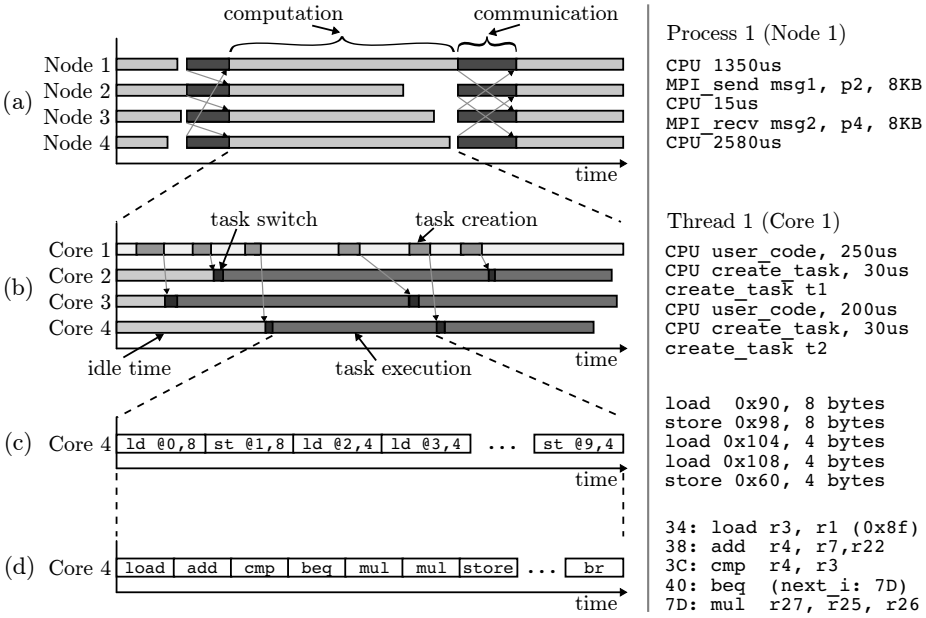


Figure 4.1: Different application abstraction levels: (a) computation plus MPI calls, (b) computation plus parops, (c) memory accesses, and (d) instructions.

4.2 Application Representation Abstraction

In this section we introduce a categorization of application abstraction levels. These levels refer to different application representations as provided to a simulator at different granularities and levels of detail.

The highest application abstraction level in our definition is for multi-process applications, such as those programmed with MPI. As explained in the previous section for Dimemas, the application is composed of computation periods, referred to as *bursts* and communications. Figure 4.1a shows the timeline of an MPI application where the application is represented following this structure. On the right of the timeline, there is a sample of the information included in a Dimemas trace. The contents of a Dimemas trace include the same semantics: computation bursts are expressed as `CPU` events, and there is a specific event for each MPI call (e.g., `MPI_send`, `MPI_recv`).

For the simulation of smaller systems like a multi-core chip or a shared-memory node, the Dimemas approach is not appropriate. Different threads in a shared-memory application make use of common resources to access the shared address space, although they execute on separate cores. In that case, the level of abstraction must be lowered to be capable of observing the internals of the computation in shared-memory applications, or between inter-process communications in MPI applications.

In order to represent computation at that shared-memory level, applications can be expressed as in high-level programming models such as the ones explained in Section 2.7. As in the methodology presented in Chapter 3, the application is represented as sequential sections (user code) and parops. Figure 4.1b shows the

timeline of a task-based parallel application, which could also be part of an MPI application, as it occurs in the Roadrunner supercomputer [27] that features the Cell/B.E. accelerator (see Section 2.1.1) and other systems featuring GPGPUs. As for Dimemas, computation bursts are expressed as `CPU` events, and are interleaved with parops and, additionally, explicit memory transfer operations for distributed-memory multi-cores. In the example in Figure 4.1b, Core 1 creates and schedules tasks (gray phases), and cores from 2 to 4 execute those tasks (phases between dark gray) as they are being created by Core 1. This example works as the one shown in Figure 3.5.

This level of abstraction is appropriate for quickly evaluating application scalability: whether an application can make use of an increasing number of cores; and memory bandwidth requirements: whether data is transferred fast enough to cores to avoid slowdowns. For accelerator-based systems where computation cores work on non-coherent scratchpad memories (e.g., Cell/B.E., GPUs, DSPs), this abstraction level includes all information required for an accurate evaluation of the memory system. Since such accelerators only access data in their scratchpad memories, the computation is not affected by the activity in other components in the system. This means that, as long as the accelerator core and scratchpad memory features remain unchanged, modifications to the rest of the memory system and the interconnection network will not affect the accelerator computation delay. Thus, modeling data transfers between off-chip memory or last-level cache, and scratchpad memories is sufficient for an accurate simulation. We demonstrate this point in Section 4.5.2.

However, for cache-based architectures this approach has, as is intended, some limitations in favor of simulation speed. The level of abstraction must be lowered to simulate cache-based architectures accurately, so the memory system is stressed with the application memory accesses. Figure 4.1c shows a list of memory accesses corresponding to a specific chunk of computation. This kind of representation is widely used by trace memory simulation methods [35, 167, 138, 173, 105, 88, 109], some of them covered in Section 3.2.

Additionally, when it is necessary to explore microarchitectural issues concerning the core pipeline, or a more detailed understanding of the application execution is required, the simulated application needs to be represented at the instruction level. Figure 4.1d shows the list of instructions in a computation burst. This is the lowest level of representation of an application, and is the one used in execution-driven simulators and trace-driven simulators focusing in the pipeline microarchitecture. As previously mentioned, this representation level allows the detailed understanding of the microarchitecture activity, but prevents the exploration of large-scale multi-cores due to its time-consuming operation.

4.3 Model Abstraction

The application abstraction levels shown in Figure 4.1 are the base for the architecture model abstractions introduced in this section and that we implemented in TaskSim. To work with such application abstraction levels, we found appropriate the use of event-driven simulation and the use of traces for all levels. This way, functional simulation is not needed at simulation time, and the application representation can be actually abstracted beyond the instruction stream level.

The aforementioned application abstraction levels allow the definition of four

architecture model abstractions implemented in the following simulation modes:

- **Burst**: based on the shared-memory abstraction level (Fig. 4.1b). Accounts for the execution time of sequential sections and parop events.
- **Inout**: based on the shared-memory abstraction level (Fig. 4.1b). Extends burst with explicit memory transfers.
- **Mem**: based on the memory abstraction level (Fig. 4.1c). Used for memory simulation.
- **Instr**: based on the instruction abstraction level (Fig. 4.1d). Used for instruction-level simulation.

The burst mode operates on a set of cores that simulate the computation bursts assigned to the associated threads, and carry out their synchronization through the parops in the trace. Therefore, simulation of the memory system and core pipeline is not necessary, so they are not even present on simulations in burst mode. However, for all other modes (inout, mem and instr), the cache hierarchy, network-on-chip and off-chip memory are simulated for a proper timing of memory accesses. More details on the implementation of these four architecture abstractions are given in the following sections.

4.3.1 Burst Mode

The burst mode implements a fairly simple event-driven core model. There is one trace for the main task, corresponding to the starting thread in the program, and then one trace for every other task executed in the application. A task trace includes all computation bursts as `CPU` events, and the required parops for a correct parallel execution. Since different tasks have separate traces, they can be individually scheduled to available cores. The parallel work scheduling can be either *fixed* or *dynamic*.

For fixed scheduling, as soon as a task is created and scheduled, any available core can start its execution. This mechanism is implemented using a *semaphore*-like strategy, that is *signal* and *wait* operations. After a task creation, a signal event unblocks the execution of the corresponding task, which is assigned to a core as soon as it becomes free.

The same signal-wait mechanism allows the simulation of thread synchronizations such as *task wait* (waiting for all previously-created tasks), *wait on data* (waiting for a producer task to generate a piece of data) and *barriers*.

The trace format for the burst mode using fixed scheduling includes the following event types:

- `CPU`: indicates the beginning of a computation burst. It includes a computation type id and its duration in nanoseconds.
- `Signal`/`Wait`: used for thread synchronization. Both include a *semaphore* id.

The core model using this trace format operates as follows. One of the simulated cores starts simulation by reading the first event in the main task trace. `CPU` events are then processed by adding their duration to the current

cycle value in the target architecture (a ratio can be applied to burst durations to simulate different core performance levels). The rest of the cores in the system start idle, and wait for tasks to be created and scheduled. Whenever a `Signal` event for task creation is found in the main task, any idle thread can start simulating the corresponding task. The core taking the new task becomes active, and starts processing its events.

For simulations using dynamic scheduling, TaskSim adopts the approach presented in Chapter 3. The simulator employs an interface to a runtime system that executes the parops at simulation time. In this case, the `Signal` and `Wait` event types are not necessary, and the trace includes, instead, the calls to the appropriate runtime system operations. When runtime call events are found in the trace, the runtime system is invoked to execute them, and perform the corresponding scheduling and synchronization operations. These operations are executed based on the state of the simulated machine, and task execution is simulated according to the decisions made by the runtime system. For more information see Chapter 3.

4.3.2 Inout Mode

The inout mode builds on top of the burst mode to provide precise simulation of multi-core architectures using scratchpad memories (non-coherent distributed shared-memory systems). As previously mentioned, the execution of a core accessing a scratchpad memory is not affected by the features of other elements in the system. The access to a scratchpad memory is deterministic, and the core will only be delayed on eventual synchronizations with explicit memory transfers between the scratchpad memory and the cache hierarchy levels, or off-chip memory.

The trace format of the burst mode is enriched with specific events that indicate the initiation of explicit memory transfers (e.g., DMA), and the synchronization with their completion before bursts reading that data:

- `DMA`: used to initiate an explicit memory transfer. It includes the transfer type (read/write), data memory address, size and a *tag id*.
- `DMA_wait`: used to synchronize with previously initiated memory transfers. It includes a *tag id*.

The application simulation in inout mode works as in the burst mode but, when a `DMA` event is found, that core sends the proper commands to program the memory transfer in the corresponding DMA engine. DMA transfers are asynchronous so the simulation of that memory transfer takes place in parallel with the processing of computation bursts. Then, on a `DMA_wait` event, the core stalls until it receives acknowledgement of the completion of the corresponding memory transfer.

The core model in both burst and inout modes is intentionally simplistic. To avoid the slow detailed simulation of pipeline structures, the target core is assumed to feature the same characteristics as the underlying machine on the trace-generation run (or a relative performance using ratios for computation bursts). Despite this simplicity, the isolation of computation on cores with scratchpad memories results in accurate memory system and interconnection

network evaluations that, for large target accelerator-based architectures, complete in a few minutes, as is shown in Section 4.5.2.

4.3.3 Mem Mode

For appropriately stressing the cache hierarchy elements and off-chip memory on cache-based systems, the mem mode considers the list of executed memory accesses of the application. The burst mode is extended to add trace memory simulation to the computation bursts in the target application. For each computation burst, a trace of all corresponding memory accesses is captured and a pointer to the resulting memory access trace is included in the associated `CPU` event. Then, during simulation, when the simulator processes a `CPU` event, it ignores the computation burst duration in the trace, and it simulates the corresponding memory access trace instead.

The trace formats of trace memory simulation works in the 80s and early 90s, included just the memory accesses and, in some cases, the timestamp of the cycle when they were issued. However, in order to more precisely model the core performance, the memory access trace for the mem mode also includes the number of non-memory instructions between memory accesses, thus leading to a trace format analogous to the one of the TPTS simulator explained in Section 2.5.5. The records in the trace include the following information for each memory access:

- Access type: Load or Store
- Virtual memory address.
- Access size in bytes.
- Number of instructions since the previous memory instruction.

The core model uses this information to simulate the performance of an out-of-order core by modeling the re-order buffer (ROB) structure; a technique by Lee et al. [109] also used in TPTS, and referred to as ROB Occupancy Analysis. Using this method, simulation runs as follows. When the simulated core reaches a `CPU` event, it starts processing the associated memory access trace pointed by the event record. It reads the first memory access in the trace, an starts issuing the previous instructions at a given *issue rate*. When all previous instructions are issued, the memory access is issued and sent to the first-level cache. Simulation keeps processing the following memory accesses in the trace in the same manner, allowing several concurrent memory accesses in flight. However, to avoid performance overestimations and account for resource hazards, simulation considers the limitation imposed by the ROB size. This assumes that the ROB is the main limiting factor in the core based on a study by Karkhanis et al. [102]. For every memory access, the associated number of previous instructions and the memory instruction itself, are stored in the ROB. Then, when the ROB is full, no more trace records are processed, and only the memory accesses in the ROB are allowed to be concurrently accessing the memory system. Non-memory instructions in the ROB are committed at a given *commit rate*. Commit blocks when a pending memory access arrives to the head of the ROB, and it does not unblock until the memory access is resolved.

This technique assumes that all memory references are independent. However, for many applications, data dependencies may be the primary factor limiting memory-level parallelism. Memory access traces can be enriched with data dependencies and only allow independent accesses to be sent in parallel. This extension provides a closer approximation in terms of accuracy to instruction-level simulation.

This method is, in any case, faster than working at the instruction level, but we further speed up simulation by applying trace stripping [138, 173] to TaskSim memory access traces. This technique consists of simulating the memory access trace on a small direct-mapped filter cache, and only storing misses and the first store to every cache line (only required for the simulation of write-back caches [173]). The resulting traces experience reductions in size of up to 98% depending on the application, and will perform the same number of misses as the original trace on simulations of caches with a number of sets equal or larger to the number of sets of the filter cache. A limitation of this technique is that the cache line size of the simulated architecture must be the same that was used in the filtering process for the filter cache.

The results in later sections show that this architecture model abstraction provides a high simulation speed that is 18x faster than simulating at the instruction level. We also evaluate this simulation mode in more detail in Chapter 5.

4.3.4 Instr Mode

In order to simulate core activity more accurately on systems with caches, the instr mode extends the burst operation mode to allow the employment of instruction traces for computation bursts, similarly to the extension applied for the mem mode. The instruction stream is captured at tracing time using PIN [113], a dynamic binary instrumentation tool, and an instruction trace is generated for every computation burst. Then, as for the mem mode, the `CPU` event format is extended to include an extra field for storing a pointer to the corresponding instruction trace.

The core model in instruction mode operates exactly as in burst mode, but the duration of `CPU` events is ignored, and the corresponding instructions are simulated instead. There are many ways to encode an instruction trace but, in general, they require to include the following information for every instruction:

- Operation code: it can be more or less generic depending on the target ISA and the detail of the core model (e.g., *arithmetic* rather than the specific instruction opcode).
- Input and output registers.
- Memory address and data size for memory instructions.
- Next instruction for branch instructions.

This mode allows a detailed core simulation. Obviously, the more accurate the core model, the longer the simulation takes to complete. In TaskSim, different instruction-level core models can be used, and even switched from one to another at simulation time, as described in the next section.

4.3.5 Summary

Table 4.1 shows a summary of the presented simulation modes. This includes their applicability to different architecture types or software domains and their usability for computer architecture studies. The burst mode allows to evaluate application scalability and, when it works integrated with a runtime system (as described in Section 4.3.1), it also provides a framework to perform runtime system evaluations as we explained in Chapter 3. On top of these features, the inout mode allows to evaluate the memory system and interconnection network designs, and it is accurate for scratchpad-based architectures. The mem mode provides support for the evaluation of the memory system and interconnection, as well as the inout mode, and also allows to analyze the design of the cache hierarchy components. On top of this, the instr mode provides support for evaluating the microarchitecture of the core pipeline structures. Table 4.1 also shows an estimate of the simulation modes speed to give an idea of the trade-off between simulation speed and modeling detail. These trade-offs are more thoroughly evaluated in Section 4.4.

Table 4.1: Summary of TaskSim simulation modes. This includes their applicability on computer architecture evaluations, and their features also comparing to state-of-the-art (SOA) simulators

Mode	Applicability	Features
Burst	Application scalability RT system evaluation	Potentially faster than native execution 100x faster than SOA functional sims
Inout	Memory system Interconnect	100-1000x faster than SOA instr.-level sims Accurate for scratchpad-based CMPs
Mem	Cache hierarchy	10-100x faster than SOA instr.-level sims
Instr	Core microarchitecture	–

It is worth mentioning that TaskSim allows to switch between different levels of abstraction for different computation bursts along a single simulation. This is very useful to quickly traverse initial phases of an application in burst/inout mode similarly to *fast forwarding* in some sampling-based simulation methodologies (see Section 2.4.4), and then execute the computation bursts of interest in mem or instr mode.

4.4 Speed-Detail Trade-Off

We have chosen three of the most popular computer architecture research simulators to analyze their trade-offs between simulation speed and modeling detail for their different levels of abstraction. The simulators are Simplescalar (see Section 2.5.1), Simics and its timing module GEMS (see Section 2.5.2) and gem5 (see Section 2.5.3).

All three simulators are execution-driven, and thus rely on being provided with the executable binary files of the application. This leads them to work

at the instruction level of abstraction (Figure 4.1d), but they still provide several levels of model abstraction, ranging from the highest level being functional simulation and, the lowest, the simulation of the core pipeline structures. All abstraction levels in Simplescalar, Simics+GEMS and gem5 are compared to the ones explained in the previous section in terms of simulation speed and modeling detail. As a first observation, the instr level in TaskSim considers the application instruction stream as well, but mem, inout and burst benefit from a higher-level abstraction of the application representation to provide higher simulation speeds while still being insightful for different purposes.

Table 4.2 includes the different abstraction levels and performance of the simulators considered in this study. The results were extracted from executions on an Intel Xeon running at 2.8 GHz, with 512 KB of L2 cache, 2 GB of DRAM at 1333 MT/s and running a 32-bit Linux distribution. For each application, we repeat the simulations four times, and took the fastest for all abstraction levels to avoid operating-system noise. We simulated a single-core configuration in all simulators because it is the fastest configuration in all cases. The addition of simulated cores usually leads to a super linear slowdown. We employed a set of scientific applications, listed in Table 4.4, for these experiments, all compiled with `gcc`, `-O3` optimization level and cross-compiled for PISA (Simplescalar), SPARC (Simics), Alpha (gem5) and x86 (TaskSim). The x86 binaries were executed on the host system to generate traces for the different abstraction levels of TaskSim.

The different abstraction levels of Simplescalar in this study are *sim-fast*, which just performs functional simulation supported by system-call emulation; *sim-cache*, which just simulates the first- and second-level caches with immediate update and a fixed latency for L2 cache misses (off-chip memory accesses); and *sim-outorder*, which adds to sim-cache the simulation of an out-of-order processor pipeline. The three levels of abstraction correspond to the different binaries generated when compiling the Simplescalar distribution sources.

The highest abstraction level of Simics is its standalone execution (no timing modules, `-fast` option) for functional simulation. The second level is the incorporation of the GEMS Ruby module, set up for a `MOESI_CMP_token` configuration; and the lowest level is the addition of the Opal module. We could have also included an intermediate simulation setup including Opal but not Ruby. However, detailed modeling of core pipelines without a model for the memory hierarchy does not provide any additional insight because other simulators do not have a configuration as such.

We configured gem5 to run in system-call emulation mode, and to support the Alpha ISA. The different abstraction levels provide functional simulation (*func*: no flags are specified); the simulation of the cache system (*cache*: `--caches --l2cache --timing`); the addition to cache of an in-order core pipeline model (*inorder*: `--caches --l2cache --inorder`); and the addition to cache of an out-of-order core pipeline model (*ooo*: `--caches --l2cache --detailed`). The TaskSim architecture model abstractions were discussed in depth in Section 4.3.

Table 4.2: Comparison of abstraction levels in existing simulators in terms of simulation speed and modeling detail

Abstraction level	Speed (KIPS)	Ratio to fastest	Ratio to native	Model detail
		SIMPLESCALAR		
sim-fast	20,425	1	288.07	functional only
sim-cache	5,946	3.43	999.29	sim-fast + cache hierarchy model
sim-outorder	1,062	19.23	5,669.55	sim-cache + OOO pipeline model
		SIMICS + GEMS		
standalone	32,517	1	128.23	functional only
Ruby	129	251.54	31,827.10	adds cache hierarchy and memory model
Ruby + Opal	14	2,237.40	260,017.16	Ruby + OOO pipeline model
		GEM5		
func	984	1	4,381.74	functional only
cache	401	2,45	10,734.61	adds cache hierarchy and memory model
inorder	47	20,92	98,335.57	cache + in-order pipeline model
ooo	85	11,80	46,984.21	cache + OOO pipeline model
		TASKSIM		
burst	4,175,762	1	0.80	computation bursts and parops
inout	132,573	31.50	25.70	burst + explicit memory transfers
mem	2,067	2,020.32	1,561.27	burst + trace memory simulation
instr	113	36,919.01	31,173.21	burst + core pipeline model

The second column in Table 4.2 shows the simulation speed of the different abstraction levels in kilo-instructions per second. The third column shows the ratio of the simulation speed versus the fastest mode in the same simulator. The values shown are the average for all applications. As expected, as the level of abstraction is lowered, simulation gets slower. However, simulation speeds significantly vary between simulators due to their different simulation approaches: execution- vs. trace-driven, full-system vs. system-call emulation; and their different levels of modeling detail. For example, Ruby, gem5 and TaskSim model the off-chip DRAM memory operation in detail, while Simplescalar applies just a fixed latency to L2 cache misses. The differences between the various functional modes of the execution-driven simulators are quite significant. Functional simulation in Simics is significantly faster than its other levels of abstraction due to its software development target. Even though Simics simulates full-system, it is actually faster than sim-fast, whose model is much simpler. Contrarily, gem5 functional mode is much slower even in system-call emulation mode as, in this case, the target is architecture simulation.

The abstraction levels modeling the core pipeline details, cache hierarchy and memory system are the slowest in all simulators and their speed ranges from 14 KIPS for Ruby+Opal, which implements a very detailed model, to 1062 KIPS for Simplescalar which, apart from being not so complex, also benefits from sitting on a simple memory system model. It is remarkable that the inorder model of gem5 is actually slower than the out-of-order model which, intuitively, should not be the case, since an out-of-order core model is more complex. This fact has been consistently found across different applications and different environment setups of gem5.

The forth column shows the ratio to the application execution on the host machine. It must be noticed that the comparison of simulation speed between different simulators does not match the comparison of their ratio to native execution. This is because different simulators employed binaries compiled for different architectures that required different numbers of executed instructions to complete.

It is worth highlighting that the TaskSim burst mode can be faster than native execution, as was found for the applications in this study, and is 128x faster than the fastest functional mode. This makes sense, as the speed of the burst mode depends on the number of events captured in the trace. In this case, the applications were compiled for task-parallel execution and simulated on a single core, which is useful for the sake of comparison, but the burst mode is more interesting when simulating parallel systems, as is shown in the experiments in Section 4.5.1. The rest of the modes in TaskSim simulate the memory system, which slows down simulation in favor of simulation detail. The inout mode benefits from the aforementioned isolation of execution on accelerators to get a high simulation speed, only 25x slower than native execution, because only explicit memory transfers are simulated. Finally, the mem mode provides an 18x speedup compared to the instr mode. Among the rest of abstraction levels that only simulate the memory system, only sim-cache is faster, which, as previously mentioned, uses a simple model for caches and does not simulate the off-chip memory.

Table 4.3: TaskSim main configuration parameters. The experiments in Section 4.5 use the default values unless it is explicitly stated otherwise

Parameter	Description (units)	Default Value
	Cache (L1/L2/L3)	
Associativity	Number of ways per set	2/4/4
Size	Total cache size (bytes)	32K/256K/4M
Data placement	Replication or interleaving	replication
MSHR entries	MSHR table size	8/64/64
	Scratchpad memory	
Size	Total memory size (bytes)	64K
Latency	Access latency (cycles)	3
	DMA engine	
Queue size	DMA queue size	16
Maximum transfers	Max. concurrent transfers	16
Packet size	DMA packet size (bytes)	128
	Interconnection Network	
Maximum transfers	Max. concurrent transfers	no. of links
Latency	Link latency (cycles)	1
Bandwidth	Link bandwidth (bytes/cycle)	8
	Memory Controller	
Queue size	Access queue size	128
Data interleaving mask	Interleaving granularity (bytes)	4096
Number of DIMMs	Number of DRAM DIMMs	4
	DRAM DIMM	
autoprecharge	Enable/disable autoprecharge	disabled
data rate	Transfers per second (MT/s)	1600
burst	Number of bursts per access	8
$^{t}_{\mathrm{RCD}}$,$^{t}_{\mathrm{RP}}$,CL,$^{t}_{\mathrm{RC}}$,$^{t}_{\mathrm{WR}}$,$^{t}_{\mathrm{WTR}}$	Timing parameters	[3]

4.5 Evaluation

For the experiments in this section we configure TaskSim using the configuration parameters shown in Table 4.3. A cache-based homogeneous multi-core configuration (as the one shown in Figure 2.5a) is used for the evaluations of the burst, mem and instr simulation modes in Sections 4.5.1 and 4.5.3. The architecture configurations for the Cell/B.E. and the SARC architecture shown in Figures 2.5b and 2.5c are used for the evaluation of the inout mode in Section 4.5.2.

4.5.1 Application Scalability using Burst

The experiments in this section show the validation of the burst mode for evaluating application scalability. Simulations are compared to real executions on an eight-core AMD Opteron 6128 processor machine. We use a set of scientific applications used in high-performance computing (HPC) plus *blackscholes* from the PARSEC benchmark suite, which has been ported to OmpSs (shown in Table 4.4).

[3] Default values for DRAM timing parameters match the Micron DDR3-1600 specification.

The applications are first run natively on the real system for different numbers of threads ranging from one to eight. Execution time is measured for the parallel computation section of the application, avoiding the initialization part, and execution time is averaged over ten repetitions. On the same machine, we collect ten traces for each benchmark. We perform simulations with these ten traces per benchmark using the dynamic scheduling approach described in Section 4.3.1 and for multiple numbers of threads from one to eight as done in the real executions. The resulting simulated time per benchmark and number of threads is the average over the ten simulations using the ten different traces.

Table 4.4: List of benchmarks, including their label used in the charts and a description of their configuration parameters.

Benchmark	Label	Parameters
2D convolution	2Dconv	4096x4096 pixels image, 8x8 filter
Cholesky factorization	Cholesky	32x32 blocks of 128x128 elements
LU factorization	LU	32x32 blocks of 128x128 elements
Matrix multiplication	MatMul	32x32 blocks of 128x128 elements
Kmeans clustering	Kmeans	1M points, 64 dimensions, 128 centers
k-NN classification	kNN	200K train points, 5K test points, 64 dimensions, k=30, 20 classes
Blackscholes	BS	PARSEC native input: in_10M.txt

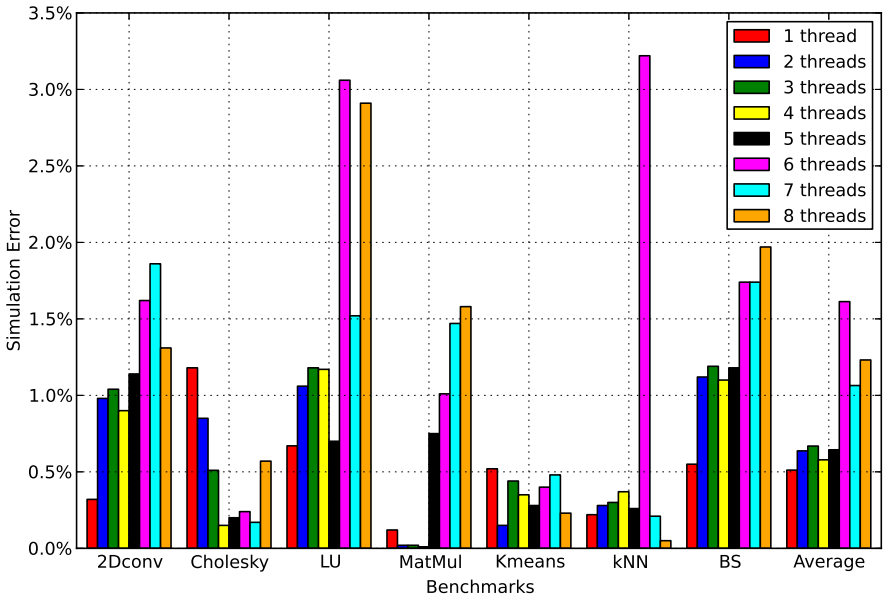


Figure 4.2: Simulation error of the burst mode compared to the real execution on an eight-core AMD Opteron 6128 processor for multiple numbers of threads.

The burst mode assumes that task execution time barely varies from an ex-

ecution on one thread compared to an execution on a higher number of threads. Therefore, codes optimized to fit task data in private caches and not being limited by memory bandwidth are candidates to be accurately simulated, as it is the case of highly-optimized HPC applications and scientific libraries. Figure 4.2 shows the percent error of the predicted execution time using the burst mode and the execution on the real machine. The average error across all benchmarks and numbers of threads is below 1.6%, and the maximum error is 3.2%. Part of this error comes from the difference between the native and the trace-collection runs and from instrumentation overheads.

Table 4.5: Cholesky factorization of a 4096×4096 blocked-matrix using different block sizes. The table shows the number of tasks, the average task execution time and the comparison of real execution and simulation for four and eight threads.

Block size	64× 64	128× 128	256× 256	512× 512	1024× 1024	2048× 2048
Number of tasks	45760	5984	816	120	20	4
Avg. task time	212μs	1.4ms	9.3ms	62ms	371ms	1.8s
Real vs. Simulation - 4 threads						
Real exec. time (s)	2.89	2.12	2.01	2.19	3.26	7.43
Predicted time (s)	2.65	2.12	1.98	2.14	3.13	7.38
Error (%)	8.26	0.15	1.28	2.05	4.03	0.72
Simulation time (s)	3.15	0.41	0.08	0.09	0.36	1.39
Slowdown	1.09	0.20	0.04	0.04	0.11	0.19
Real vs. Simulation - 8 threads						
Real exec. time (s)	1.56	1.08	1.05	1.35	3.06	7.41
Predicted time (s)	1.34	1.07	1.02	1.28	3.02	7.38
Error (%)	14.08	0.57	2.63	5.13	1.31	0.41
Simulation time (s)	3.45	0.46	0.12	0.23	1.28	2.96
Slowdown	2.21	0.48	0.11	0.17	0.42	0.40

We use Cholesky factorization (see Figure 2.11) as a case study of the validity and usefulness of the burst mode for scalability analysis using multiple block sizes (M×M in Figure 2.11a). Table 4.5 shows the number of tasks and average task execution time for multiple block sizes between 64×64 and 2048×2048. We executed these multiple configurations in the real machine for different numbers of threads from one to eight and averaged the execution time over ten repetitions. We also simulated the same configurations using ten different traces and averaged the predicted execution time.

Figure 4.3 shows the results. The configuration with 64×64 blocks shows an error up to 14% because instrumentation overheads start becoming important for such small block sizes. However, for all other block sizes, simulation accurately reproduces real execution for multiple threads, even having collected the trace on one thread. Table 4.5 shows the real and simulated execution times for four and eight threads. The error for all block sizes except 64×64 tops at 5.2%. The results are not only accurate but also insightful. We can see that block sizes larger than 512×512 do not exhibit enough parallelism to fully utilize all eight threads. Block size 1024×1024 stops scaling at five threads, and 2048×2048

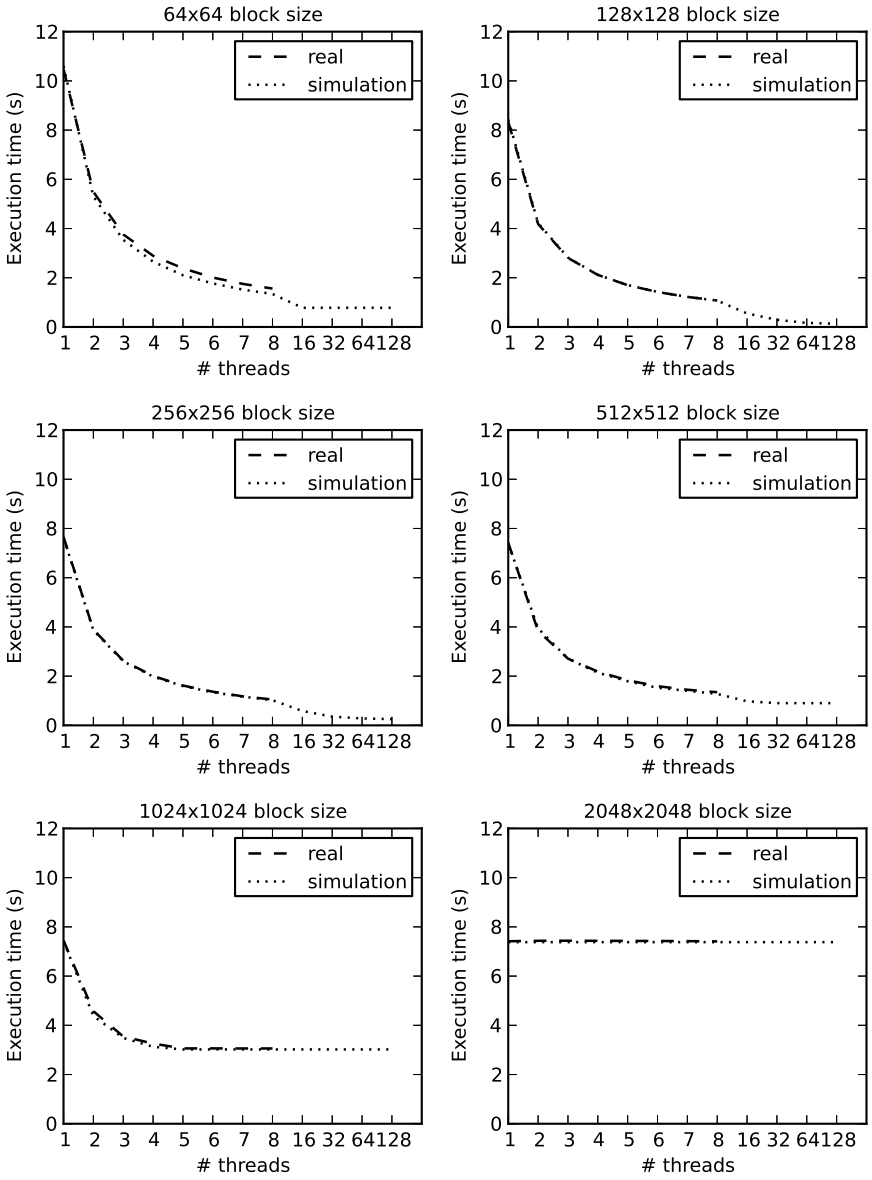


Figure 4.3: Comparison of real and simulated execution time for a 4096×4096 blocked-matrix Cholesky factorization using multiple block sizes.

does not have any parallelism at all due to task dependencies.

We extended the simulations for 16, 32, 64 and 128 threads. The results show that block size 64×64 does not scale beyond 16 threads due to task creation overheads, a problem we already explained for the matrix multiplication experiment in Section 3.5. That is that the rate at which the main thread produces tasks is not fast enough compared to the rate at which tasks are executed by other threads, thus not being able to keep all threads busy. Block size 512×512 stops scaling at 32 threads, and 256×256 has diminishing returns beyond 16 threads, which makes the execution with blocks of 128×128 two times faster than with 256×256 when using 128 threads.

This kind of analysis is done fast because the burst mode is in most cases faster than native execution. Table 4.5 shows the time required to simulate the four and eight configurations on one thread (our implementation is not parallel), and the slowdown with respect to the real execution time. The average slowdown for four threads is 0.28x (3.6x faster than native execution) and for eight threads is 0.63x (1.6x faster).

4.5.2 Accelerator Architectures using Inout

In this section we introduce a set of experiments showing the utility and accuracy of the inout mode for architectures using scratchpad memories. Figure 4.4 shows several timelines of a Fast Fourier Transform (FFT) 3D application written in CellSs [30]. This FFT3D application is also compiled with the Mercurium compiler (see Section 2.7.1), but to use the CellSs runtime system.

Figure 4.4a is the real execution on a real Cell/B.E. Periods in gray represent computation phases, and periods in black represent time spent waiting for data transfers to complete. The application first performs an FFT on the first dimension of a 3D matrix. FFT execution is computation-bound, so it is dominated by computation time (gray). After the first FFT, it performs a transposition. The matrix transposition execution is memory-bound, so it is dominated by waiting time (black). After the transposition, it performs another FFT but on the second dimension, followed by a second transposition and, finally, another FFT on the third dimension. The five application stages are clearly identifiable in the timeline. The second transposition takes much longer than the first one, because data is accessed following a different pattern, which does not efficiently use the available off-chip memory bandwidth.

Figure 4.4b shows the timeline of the simulation in TaskSim using the inout mode with a Cell/B.E. configuration (Figure 2.5b) and the fixed scheduling approach as described in Section 4.3.1. The inout mode is able to closely reproduce the behavior of the real system. As it can be seen in Figure 4.4c, the same simulation takes place but the off-chip memory data-interleaving granularity is set to 128 B instead of the default 4 KB in the real Cell/B.E. The time to complete the second transposition is then greatly shortened, resulting in a delay similar to that of the first transposition. This is because, using a 128 B interleaving scheme, several DIMMs are always accessed in parallel in both transpositions, thus achieving close to peak bandwidth efficiency, and getting a 30% reduction in total execution time.

In Figure 4.4d and Figure 4.4e, the same simulations (4 KB and 128 B interleaving granularities respectively) are carried out using a 256-core configuration of the SARC architecture (Figure 2.5c) excluding the L1 caches. Computa-

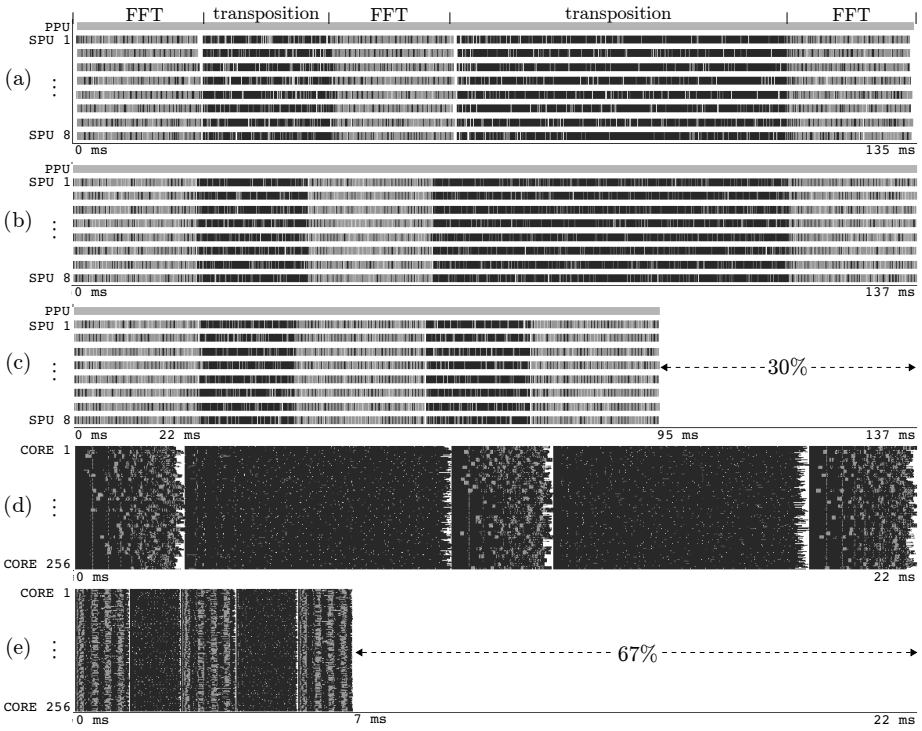


Figure 4.4: Inout mode experiments on scratchpad-based architectures using an FFT 3D application: (a) execution on a real Cell/B.E., (b) simulation of a Cell/B.E. configuration, (c) simulation of a Cell/B.E. configuration using a 128 B memory interleaving granularity, (d) simulation of a 256-core SARC architecture configuration using a 4 KB interleaving granularity, and (e) simulation of a 256-core SARC architecture configuration using a 128 B interleaving granularity. Light gray show computation periods and black shows periods waiting for data transfer completion.

tion is now spread among many more units, thus reducing the total execution time from 137 ms to 22 ms. Also, there is much more pressure on the memory system due to more cores concurrently accessing data. Because of this high memory congestion, there is no significant difference between the time taken by both transpositions despite the different access patterns. However, the 4 KB scheme still cannot make an efficient use of memory bandwidth. The 128 B scheme achieves much higher efficiency, and thus leads to a 67% reduction in total execution time.

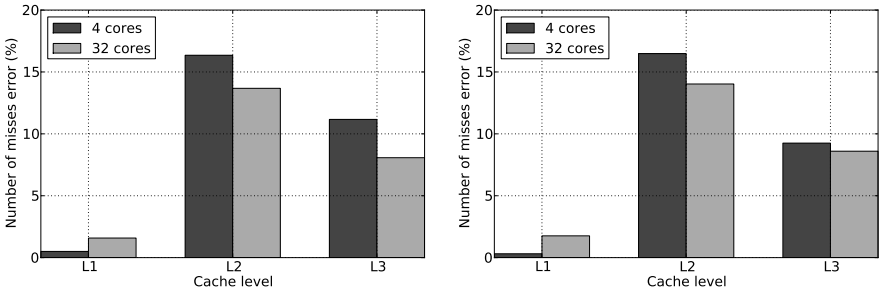
This experiment shows that the problems suffered in small-scale architectures are not necessarily the same as those in large architectures. Also, the potential solutions do not have the same effect at different scales. The magnitude of the improvement by changing a parameter can be completely different in a small configuration and a large configuration. Therefore, the detailed simulation of small architectures is not sufficient for the exploration of future large architecture designs.

The experiment also shows the potential of the inout mode to quickly obtain

an understanding of the application behavior on different memory system and interconnection network configurations for scratchpad-memory-based architectures. It must be noticed that each 256-core simulation shown in this section required approximately three minutes to complete on an Intel Core2 T9400 running at 2.53GHz.

4.5.3 Memory System using Mem

The following experiments show the error of using trace memory simulation with stripped traces in the mem mode. We simulate the SMP architecture in Figure 2.5a in both mem and instr modes. The architecture is configured with 4 and 32 cores, and using two different interleaving granularities for main memory: 4 KB and 128 B. Figure 4.5 shows the percentage difference of the number of misses between the two modes using Cholesky. The number of misses is summed for all caches in the same level across the architecture. As expected, the error in the L1 is small despite the use of stripped traces. However, for the L2 and L3 levels, the error is much higher, as the trace stripping algorithm does not account for the effects on shared caches for parallel execution on multi-cores, a problem that we address in Chapter 5. In any case, the error is consistently below 17% for the different numbers of cores and hardware configurations.



(a) 4 KB DRAM interleaving granularity (b) 128 B DRAM interleaving granularity

Figure 4.5: Difference in number of misses between the mem and instr modes for different simulated configurations using 4 and 32 cores, and different interleaving granularities.

For the sake of comparison, we also repeated the simulations in this section using non-stripped memory access traces. The maximum error in this case is 3%, thus showing the inaccuracy incurred when using trace stripping. That sets once again a trade-off between simulation speed and accuracy. On an Intel Xeon E7310 at 1.6GHz, simulations with full traces were 2x faster than instruction-level simulation, while the ones using stripped traces (with an 8KB filter cache) were around 20x faster, which is consistent with the results in Section 4.4.

The large error of filtered traces compared to full traces led us to analyze its sources. Trace filtering techniques were used accurately in previous works for single-threaded applications. In multithreaded applications, however, a memory access that hits in a machine configuration may miss in another because the data was invalidated by another thread in the system. Therefore, if an access is not present in the trace because it was filtered out during trace generation, it would

not be present during simulation to model a potential miss due to invalidations. This fact motived us to initiate research to enable more accurate simulations of multithreaded applications using filtered traces. We present this work in the next chapter.

4.6 Summary

In this chapter we have analyzed the use of multiple levels of abstraction in computer architecture simulation tools as a base for simulating large-scale architectures. As part of this analysis, we have characterized the levels of abstraction in existing simulation environments. In simulators requiring target applications to be represented as an instruction stream, the highest level of abstraction is functional emulation, which has been shown to be more than 100x slower than native execution for the highly optimized Simics platform.

Also, we presented a definition of multiple application representations that are more abstract than an instruction stream, and several simulation modes implemented in TaskSim based on these representations. The two highest-level modes in TaskSim, burst and inout, have been shown faster than native execution and 25x slower, respectively, while being more insightful than functional simulation. The burst mode has been proven useful and accurate for scalability studies and application analysis, with an error below 8% compared to native execution. Also, we tested the utility and accuracy of the inout mode for architectures using scratchpad memories. The inout mode has been validated against a real Cell/B.E., showing close performance behavior, and it is capable of simulating up to 256-core configurations in less than three minutes. Finally, we revisit trace memory simulation techniques that are incorporated in TaskSim to provide an 18x speedup over instruction-level simulation with a maximum error of 17% on the simulation of the cache hierarchy and memory system. This error showed the inaccuracy inherent in using filtered traces for multithreaded applications, which motives our work in the next chapter.

Chapter 5

Trace Filtering of Multithreaded Applications

Recent works [108, 109] use trace memory simulation and trace filtering techniques such as trace stripping [138, 173] to reduce the simulation time of single-threaded configurations. These works generate a trace including all L1 misses of a benchmark running on a given microarchitecture. Then, they feed this trace to a trace-driven simulator to explore the design space of the *non-core* components, such as cache hierarchy—L2 and above—, off-chip memory and on-chip interconnect. This methodology assumes a fixed core microarchitecture and L1 cache for all trace-driven simulations. This is a significant limitation for single-thread/single-core analysis, as the non-core design space is limited. This kind of methodology seems more valuable for multi-core simulations where the non-core design space is larger and a higher simulation speed is needed.

However, filtering a trace for multithreaded application simulations is not straightforward. Trace stripping consists of filtering out all L1 hits and just store L1 misses in the trace. This methodology assumes that for a given core microarchitecture and L1 cache configuration, the non-core accesses—to the L2 cache—are independent of the non-core configuration. This is true for single-threaded applications where L1 accesses either hit or miss only depending on previous accesses, which are always in the same order for a given core microarchitecture. However, cache coherence operations in multi-core architectures, such as invalidations, may make an access to hit or miss depending on the other threads activities, which actually depend on the non-core configuration. In this scenario, an access that was not recorded in the trace due to being assumed an L1 hit, may miss due to its data being invalidated by another cache in the system.

In this chapter, we introduce our approach to enable trace filtering for multithreaded applications. First, we define the problem of filtering memory trace of multithreaded applications and explain what is the state of the art in this matter. Then, we explain our methodology and our implementation using the mem mode of TaskSim. Finally, we evaluate the method compared to the state of the art in terms of trace reduction, trace generation time, accuracy and simulation speed-up.

5.1 Problem

Trace stripping [138] reduces trace size and simulation time by not including L1 hits in the trace and only simulate L1 misses. Figure 5.1a shows a snippet of a memory access trace. The trace is passed through a direct-mapped cache that informs for every access if it was a hit or a miss (Figure 5.1b). Every hit access is discarded and every miss is recorded in the trace giving as a result the filtered trace in Figure 5.1c.

TYPE	ADDRESS	SIZE
LOAD	0x7ffff488	4
LOAD	0x7ffff484	4
LOAD	0x48fbc	8
STORE	0x48e04	8
LOAD	0x48f90	4
LOAD	0x7ffff480	4
LOAD	0x7ffff47c	4
LOAD	0x7ffff478	4

(a) Original trace

TYPE	ADDRESS	SIZE
LOAD	0x7ffff488	4 (miss)
LOAD	0x7ffff484	4 (hit)
LOAD	0x48fbc	8 (miss)
STORE	0x48e04	8 (miss)
LOAD	0x48f90	4 (hit)
LOAD	0x7ffff480	4 (hit)
LOAD	0x7ffff47c	4 (miss)
LOAD	0x7ffff478	4 (hit)

(b) Filter cache simulation

TYPE	ADDRESS	SIZE
LOAD	0x7ffff488	4
LOAD	0x48fbc	8
STORE	0x48e04	8
LOAD	0x7ffff47c	4

(c) Filtered trace

Figure 5.1: Trace filtering: (a) example of a memory access trace, (b) how trace filtering proceeds, and (c) the resulting filtered trace.

Let us say the filter cache is a 16 KB direct-mapped cache with 64-byte cache lines. In the filtering process, there are a series of cold, conflict and capacity misses. The methodology states that both the full and filtered traces will generate the same number of misses in caches with the same or larger number of sets. Having more sets and/or a higher associativity (e.g., 64KB 2-way set associative cache) will cause some of the capacity and conflict misses during the filtering process to hit in simulation, but that will happen equally for the full and the filtered traces, and in both cases the same accesses will miss.

The described methodology works in single thread scenarios because there is a single order of accesses and there are no external events that may alter the cache state. However, in the case of a cache-coherent multi-core architecture running a multithreaded application, that is no longer the case. Some of the accesses that hit during the filter process, may miss in a multi-core architecture because of cache coherence actions. Figure 5.2 shows two scenarios of the execution of two threads in different simulation configurations that lead to different thread interleavings. In both cases, a cache line invalidation occurs due to a write in a remote cache to a *shared* cache line. In the case on the left, Thread 1 executes Load D before Thread 0 executes Store B. Thus, D hits and the data in its cache gets invalidated afterwards. In the right case, Thread 0 executes Store B before, so Thread 1 misses on Load D as its cache line holding the data is invalid. The problem arises when the filtered trace is generated on the left case and D is filtered out. Then, if a given simulation configuration leads to the right case, D is not present in the trace because it is assumed a hit, and the simulator cannot account for the timing effects of the cache miss.

Another characteristic of parallel programming models that may cause a naively filtered trace to fail is dynamic scheduling. The previous examples in Figure 5.2 assumed static scheduling. However, dynamically-scheduled parallel applications may schedule pieces of work to different threads depending on the

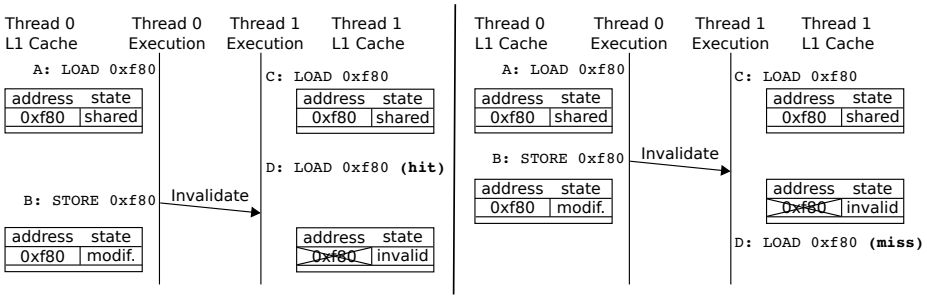


Figure 5.2: Different multithread execution interleavings. On the left, the invalidation occurs after `LOAD D` executes, so `D` hits. On the right, the invalidation occurs before `D` so `D` misses in this case. If the trace is generated with the left execution, `D` will be filtered out, and if the scenario in the right is produced during simulation, `D` will not be there to miss.

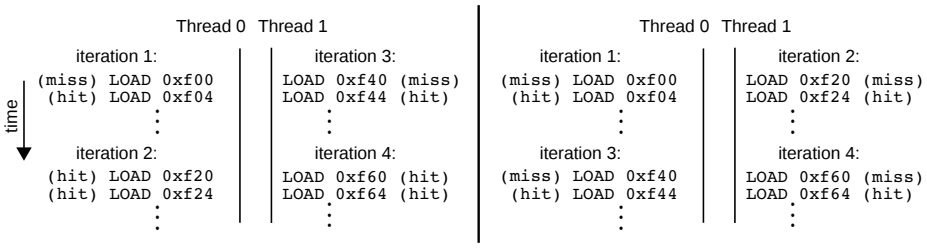


Figure 5.3: Different dynamic scheduling decisions. Iterations of a parallel loop are scheduled in different ways. On the left, iterations 1 and 2 are assigned to Thread 0, and 3 and 4 to Thread 1. That makes iterations 2 and 4 to hit on their first memory access. On the right, 1 and 3 are assigned to Thread 0, and 2 and 4 to Thread 1, making all of them to miss on their first memory access. If the trace is generated using the left case and simulation leads to the case on the right, the first accesses in iterations 2 and 4 would not be present in the trace, and both misses could not be simulated.

application state, for instance, to balance the work load. That may lead to the scenario in Figure 5.3. Given a parallel loop, multiple iterations are scheduled to multiple threads. The example shows four iterations scheduled to two threads. On the left, iterations 1 and 2 are scheduled to Thread 0, and iterations 3 and 4 are scheduled to Thread 1. Given a 64-byte cache line, iterations 1 and 2 access the same cache line, thus leading iteration 2 to hit in the cache from the very first access. The same occurs with iterations 3 and 4, but they access a different cache line than iterations 1 and 2. On the right, iterations 1 and 3 are scheduled to Thread 0, and 2 and 4 to Thread 1. In this case, all iterations miss on their first access, since they do not reuse the data loaded in the L1 cache by the previous executed iteration. If a memory access trace is generated and filtered with the case on the left, the first accesses of iterations 2 and 4 would not be in the trace. Then, if simulation leads to the scenario on the right, the misses of those accesses cannot be simulated.

5.2 State of the Art

In the 1980s and early 1990s, there were several works to reduce the size of memory access traces for trace-driven memory simulation. This had two purposes: reducing the trace file size to make the most of the limited disk space, and reducing simulation time. Some works, such as *Mache* [156] and *AE* [107], just target to reduce trace file size and do not affect simulation time, although they introduce an overhead to *decompress* (reconstruct) the trace. As an example, Mache recognizes different memory access streams during trace generation. Then, it records a tag and a base address for each stream, and just specifies the tag and the address difference for every subsequent access in that stream. The rationale behind this idea is that, in a 32-bit address space, specifying a tag and an address difference usually takes less bits than a full address.

Other works such as *block filtering* [15] both reduce trace size and improve simulation performance, but they do it at the expense of simulation error.

The most interesting work for this thesis is *trace stripping* [138, 173]. It is interesting because it reduces the trace file size, improves simulation speed, and it does not introduce simulation error. The only restriction is to use the same cache line size for the filter cache and the simulated caches. As explained in the previous section, and shown in Figure 5.1, this method uses a direct-mapped cache as a filter cache. While generating the memory access trace, the accesses are passed to the filter cache, and only misses are recorded in the trace. In the work by Wang et al. [173], trace stripping is extended to simulate write-back caches. In this case, not only are the filter cache misses recorded, but every first write to a clean line is also included. This way, the access that may change the state of a cache line to *dirty* is present in the trace and the corresponding write backs take place during simulation.

A more recent work by Lee et al. [109] introduced a simulation methodology to model not only the memory subsystem, but also a superscalar out-of-order core using traces and trace stripping. In this work, the trace is generated with a cycle-accurate simulator from which they obtain a *stripped* (filtered) trace. They present several models that use extra information in the trace such as the number of instructions and/or cycles between memory accesses, and dependencies among them. Then, the trace is used to drive the simulation but replacing the core (including the L1 caches) by a *trace player* that sends memory accesses directly to L2. The time and order in which accesses take place is decided considering the number of cycles between them, the size of the reorder buffer, and when the access is ready (its dependencies are satisfied).

All these previous works use trace filtering in the context of single-threaded applications and multiprogrammed workloads. The only previous work that addressed the use of trace filtering for simulating multithreaded applications is TPTS [108] (see Section 2.5.5). To address the problem explained in the previous section, the authors present four alternatives:

- **Naive filtering.** Filter the multiple threads in the application as done for single-threaded applications and ignore the effects of cache coherence actions.

- **Code region analysis.** Disable filtering on code regions that access shared data.

- **Individual access analysis.** Disable filtering for memory accesses whose ordering is non-deterministic and may potentially miss due to an invalidation (as shown in Figure 5.2).
- **No filtering.** Generate full traces.

The second and third approaches require an analysis that depends on the parallel programming model, something that the authors do not evaluate. The results shown for this work are only for naively filtering the trace: they assume the potential error from not including filter cache hits that may miss in multithreaded application simulations.

For the rest of this chapter, we use naive filtering as the state-of-the-art approach to which we compare our methodology against.

5.3 Methodology

We propose a methodology to overcome the problem explained in Section 5.1. Our proposal takes advantage of the synchronization happening between accesses from different threads to the same data. Whenever a thread wants to access shared data, it has to synchronize with other threads to avoid *race conditions*. Before a synchronization point (e.g., a lock), other threads may have accessed the data, thus potentially modifying the state of the copies in the private cache (e.g., invalid) as shown in Figure 5.2. In the figure, accesses A, B, C and D may be in separate critical sections. Our approach is then to *reset* (clear) the filter cache at trace generation time on every synchronization point. With this technique, every first access to a cache line after the synchronization point is recorded, thus avoiding the filtering of accesses that may potentially miss during simulation. Effectively, this methodology makes every piece of computation between synchronization points (sequential sections) to be filtered separately. This way, each sequential section is independent of any operations occurring before and after its execution.

With respect to the problem related to dynamic scheduling shown in Figure 5.3, our methodology handles this case as well. The *fork* and *join* of iterations in a parallel loop are considered synchronization operations. Then, the filter cache is reset at the beginning of every iteration so, effectively, every iteration is filtered separately and their first accesses are stored in the trace.

We show the impact of these effects in simulation accuracy using a pathological case programmed with a task-based programming model (see Section 2.7). The program creates a task (`A`) that sets all the elements of an array that fits in the filter cache to random values. Then, it creates four tasks (`B`), each of them reading all the elements in the array to do some operation (e.g., reduction). This process is repeated multiple times. The pseudo-code of this program and its task dependence graph are shown in Figures 5.4a and 5.4b, respectively.

The pseudo code uses the keyword *spawn* to denote the creation of a task that may be executed in parallel. For this example, we assume the programming model provides means to automatically identify the dependencies among tasks and schedules them for execution only when they are ready. Figure 5.4c shows the trace generation process for this application. The trace generation takes place on a single thread for simplicity, but the case applies the same to generating the trace in multiple threads. In the trace generation execution, the

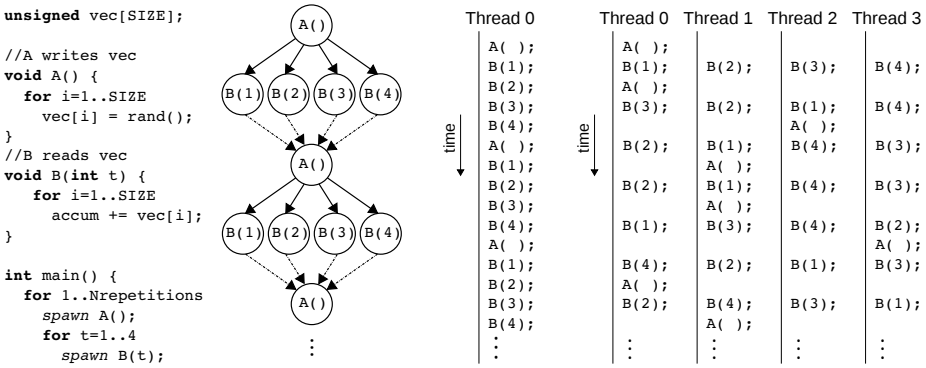


Figure 5.4: Pathological case. The figure shows: (a) the pseudo-code of the application; (b) the task dependence graph showing tasks in circles and the arrows between them are read-after-write (solid) and write-after-read (dashed) dependencies; (c) the execution in a single thread used for trace generation with the order in which tasks are executed; and (d) the simulation of the application on four threads showing to which threads tasks are dynamically scheduled and their execution order.

first execution of `A` loads `vec` in the filter cache, say a 16 KB direct-mapped cache. Then, using the naive approach, the following executions of `B` and `A` hit in the filter cache, so their accesses to `vec` are not recorded in the trace. Figure 5.4d shows a simulation using the application trace and modeling a four-thread multi-core architecture with private caches. Due to task dependencies, the simulation alternates the execution of `A` with the execution of the four instances of `B`. When the first `A` executes, it loads `vec` in Thread 0's L1 cache. Then, the four instances of `B` execute in the four threads. In Thread 0 there is no error, as it hits as it did in trace generation; but the execution on Threads 1, 2 and 3 find a cold cache where they should miss. However, the accesses in the first execution of `B(2)`, `B(3)` and `B(4)` are not present in the trace and those misses are not simulated. The same happens in the second repetition. `A` executes again in Thread 0, but its writes to `vec` are not present because they were filtered out during trace generation, and the copies in the L1 caches of Threads 1, 2 and 3 are not invalidated. Again, when Threads 1, 2 and 3 execute the second batch of `B` instances, their accesses to `vec` should miss, but again they are not present in the trace because they also hit in the trace generation run and were not recorded.

Using our methodology, the filter cache is reset at every *spawn* operation call and task finalization during the trace generation run. Thus, before every execution of `A` and `B`, the filter cache is reset, and the first accesses to `vec` in every task miss in the filter cache and are recorded in the trace. The simulation in Figure 5.4d using our methodology proceeds as follows. First, `A` loads `vec` in Thread 0's L1 cache. Then, the several instances of `B` execute, and in all cases their first accesses to `vec` are simulated. `B(1)` reuses the data loaded by `A` and hits, but `B(2)`, `B(3)` and `B(4)` find cold caches and they miss and

load copies of `vec` to their private caches. After the first batch of `B` instances execute, a second `A` executes again in Thread 0. This invalidates `vec`'s copies in the L1 caches of Threads 1, 2 and 3. Again, the first accesses to `vec` in the following `B`s are present in the trace, and they miss again, this time because those cached versions are invalid. This shows how our simple idea helps to build a filtered trace of a multithreaded application that is suitable for simulation of a multi-core architecture.

We actually carried out these experiments with the implementation of our methodology explained later in Section 5.4. We coded this pathological case in OmpSs (see Section 2.7.1) and set it to run 500 repetitions of the *main* loop. The trace filtering process is as shown in Figure 5.4c: on one thread and using a 16 KB direct-mapped filter cache with 64-byte lines. Using the naive approach, 99.8% of the accesses get discarded, while using our methodology the filtering rate is 93.2%. The cache hit ratio of a sequential access pattern over four-byte consecutive elements that fit in cache is 93.75%: every cache line holds 16 elements, so 1 out of 16 accesses to a cache line miss and the remaining 15 hit. The filtering ratio of our methodology is on that range because the data is not reused in the filter cache because it is cleared for every separate task. The naive approach gets a much higher filtering ratio, but may provide an erroneous simulation as explained before.

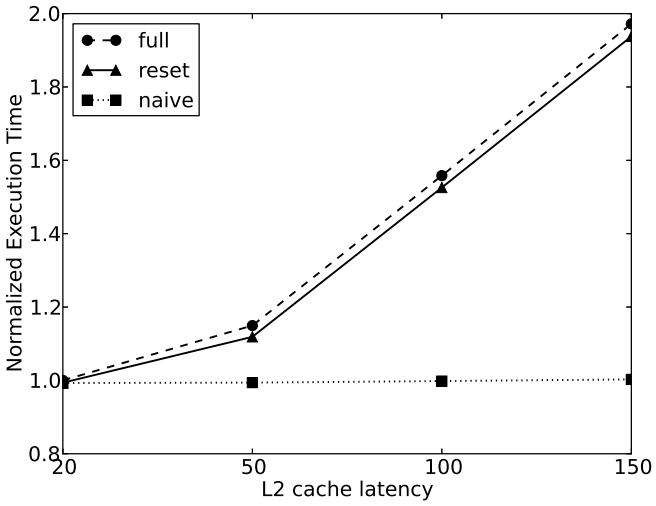


Figure 5.5: Pathological case execution time normalized to full trace with L2 cache latency of 20 cycles. A small L2 cache latency can be hidden by the superscalar core microarchitecture. Longer latencies delay L1 misses that are correctly simulated with our methodology (reset) and using the full trace. The naive method does not simulate those L1 misses, as they were filtered out during trace generation.

Figure 5.5 shows the normalized execution time of the naive method, labeled *naive*; our proposal, labeled *reset*; and the simulation using the full (non-filtered) trace, labeled *full*. We expect to have an underestimation of execution

time using the naive method because it may not simulate some of the misses due to invalidations and dynamic scheduling. The three methods are simulated for multiple L2 cache latencies on a multi-core configuration with five out-of-order cores: one core executes the main loop and creates tasks, and four cores execute those tasks. This way we do not constraint the parallelism shown in the dependence graph. For a latency of 20 cycles, the three methods predict the same performance. Since `vec` fits in L2, the additional latency of an L1 miss that hits in L2 is hidden by the superscalar core model and does not stall the pipeline. However, when going to higher latencies, the core microarchitecture cannot hide that latency anymore and the full and reset configurations predict a higher execution time due to those L1 misses. However, the naive method does not show this effect, as it cannot simulate those L1 misses: the accesses were considered hits during trace generation, so they are not in the trace.

Alternatively to our methodology, a more complex application analysis could be done to identify only accesses to shared data and only avoid filtering those. However, this would require longer developments, should be done for every programming model to be supported, and trace generation may take significantly longer. Also, such analysis would account only for potential invalidated data, but not for dynamic scheduling effects. Our methodology is simple, accounts for both effects and is the same for multiple parallel programming models.

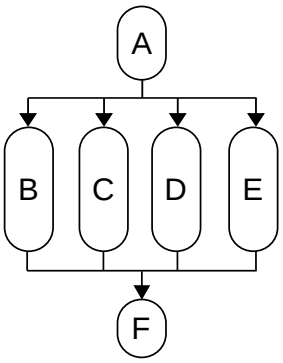
5.4 Implementation

As explained in previous chapters, multithreaded applications are composed of sequential sections and parops. In Figure 5.6a we illustrate this concept for an OpenMP parallel loop. First, the application runs on one thread, which is sequential section A. Then, the application executes a *fork* parop to spawn a parallel loop and schedule its multiple iterations to the multiple available threads (four in this example). Then, every thread executes its share of the parallel loop, and each of the iterations is in itself a sequential section (B, C, D and E). After that, a *join* parop finalizes the parallel loop and continues execution on one thread.

Our methodology leverages this knowledge to effectively filter each sequential section separately. This makes sense because the memory accesses in a sequential section are always in the same order and their effects on the private L1 cache state are always the same with respect to the previous and following accesses. To accomplish this, the trace generation engine must be notified whenever a parop executes so it knows when to reset the filter cache.

The effort required to instrument parallel applications to catch parops during trace generation depends on the programming model. However, in general, all programming models provide a runtime system API (runtime library) and/or compiler support for intermediate code generation. Figure 5.6b shows an example of the code generated by the GCC compiler for an OpenMP parallel *for* loop [2]. The programmer annotates the parallel loop with the *pragma omp for* keywords, and the compiler transforms that construct to an intermediate code that calls the *libgomp* OpenMP runtime library. Opposite to this, the *pthreads* programming model exposes the API to the programmer, who then has to deal with managing parallelism.

Some functions in a parallel programming model's runtime system just pro-



(a) OpenMP *for* scheme

```
#pragma omp for
for (int i=0; i<n; ++i)
{
    body;
}
```

Original OpenMP C code

GCC generated code

```
long i, _s0, _e0;
if (GOMP_loop_runtime_start(0, n, 1, &_s0, &_e0))
do {
  long _e1 = _e0;
  for (i = _s0, i < _e0; i++)
    body;
} while (GOMP_loop_runtime_next(&_s0, _&e0));
GOMP_loop_end ();
```

(b) OpenMP *for* code

Figure 5.6: OpenMP *for* loop construct. The figure shows: (a) a scheme of the execution flow of an OpenMP *for*; (b) the original OpenMP C code and the intermediate C code generated by GCC, including the calls to the *libgomp* OpenMP runtime library.

vide some information about the properties or the state of the application (e.g., *omp_get_thread_num*). But most of them provide the execution of a parop (e.g., *omp_set_lock*). Then, to make use of our methodology, the runtime system API has to be instrumented manually or using a dynamic binary instrumentation tool. At the same time, this instrumentation must work together with the tool used to generate the memory access trace for the notification of parop calls and the reset of the filter cache. Here we comment on two alternatives for trace generation: execution-driven simulators and dynamic binary instrumentation tools; and how they can be notified of parop calls to reset the filter cache.

Execution-driven simulators can be notified of certain application events by means of special instruction codes. Some simulators provide this mechanism built-in and available to users. An example of this is *magic instructions* in Simics (see Section 2.5.2) for C programs. The user can incorporate instrumentation points in the code by using the `MAGIC(X)` construct. Then, the simulator invokes a callback function when it reads the *magic* assembly instruction, and the parameter `X` can be used to pass information to the simulator. Other simulators, such as Simplescalar (see Section 2.5.1), do not provide a built-in functionality but it can be easily added as it is open source and this is not a complex extension.

Dynamic binary instrumentation tools, such as PIN [113] and Valgrind [130], also provide instrumentation points for calling a callback function specified by the user. In this case, the user has to register API function symbols and the corresponding callback functions are invoked when the target program executes them, without having to manually add instrumentation in the code.

Considering this, the operation for using our methodology is the same for both alternatives. Whenever the execution of a parop triggers the execution of the associated callback function, this one resets the contents of the filter cache.

In Section 5.2, we covered some works that assume a fixed core and L1 configuration for all experiments and replace the core and L1 cache models by a

trace reader that sends memory accesses directly to the L2. In our methodology, however, the trace-driven simulator must include the L1 caches. Since we want to account for cache coherent actions, such as invalidations, we need to maintain the L1 cache state for those actions to take place. In the following section, we describe our implementation of this methodology to show its feasibility and as an example of how it can be put in practice.

5.4.1 Sample implementation

We implemented this methodology in the mem mode of TaskSim (see Section 4.3.3) and using the simulation methodology explained in Chapter 3. We use our NANOS++ instrumentation plug-in and PIN to generate traces at the memory level of abstraction (see Section 4.2). Figure 5.7 shows how the different components are combined to generate traces and carry out simulation. Two versions of the OmpSs application binary are generated with Mercurium, one linked to an instrumented version of NANOS++ (instrumented binary) and another one linked to a non-instrumented version (regular binary). The instrumented binary is run natively on one thread and it generates an *application-level* trace at the shared-memory abstraction level (see Section 4.2). The regular binary is executed with PIN, also in one thread, to catch all memory accesses. For this purpose, we developed a PIN tool that generates the memory-access trace for the instrumented application. If filtering is set in the PIN tool, the memory accesses are passed to a configurable filter cache that decides whether or not to record them in the trace. Also, the NANOS++ API functions are registered in our PIN tool to trigger the execution of callback functions to reset the filter cache, if filtering is enabled.

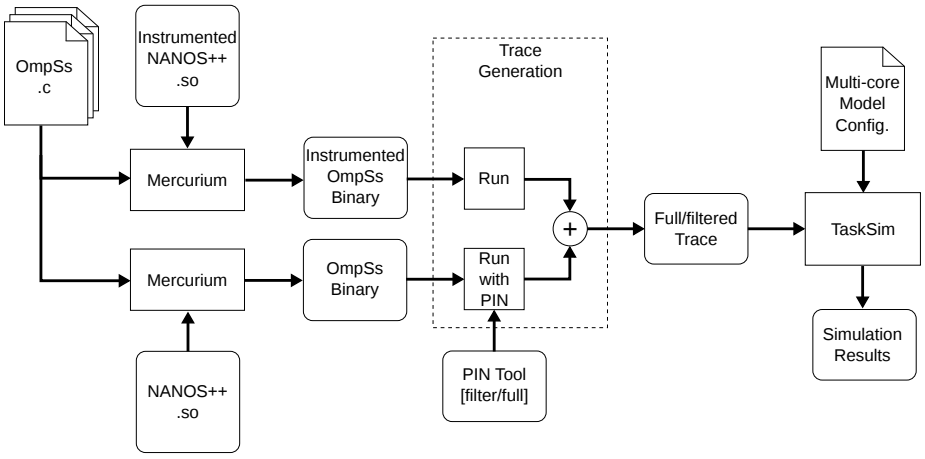


Figure 5.7: Trace generation and simulation process.

The result of the two runs is an application-level trace and one memory-access trace for every computation section. The two traces are combined together to generate a single (full or filtered) trace for TaskSim.

The trace generation process (dashed line in the figure) is carried out by our trace reader/writer library, which is also used in TaskSim to read the input trace.

This helps to develop multiple core models based on a given trace semantics regardless of the trace format as long as the API between the simulator and the trace reader/writer is maintained.

5.5 Evaluation

In this section we evaluate our proposal using our implementation in terms of increased trace size, accuracy, trace generation cost and simulation time. We compare these metrics for three filtering approaches. The first is the state-of-the-art naive approach, labeled *naive*. As previously mentioned, it ignores cache coherence actions and filters the trace without clearing the filter cache at any point. The second is our proposal, labeled *reset*, which clears the filter cache contents on every synchronization point. And the third is the full/non-filtered memory access trace, labeled *full*, which includes all memory accesses and is the baseline in our experiments.

Parameter	Description (units)	Default Value
	Core	
Number of cores	Number of cores in the chip	4/8/16
Frequency	Core/chip frequency (GHz)	1/2
ROB size	Number of ROB entries	64
	L1 Cache	
Associativity	Number of ways per set	2
Size	Total cache size (bytes)	32K
Latency	Access latency (cycles)	2
	L2 Cache	
Associativity	Number of ways per set	8
Size	Total cache size (bytes)	4M
Latency	Access latency (cycles)	20/50/100/150
	Interconnection Network	
Maximum transfers	Max. concurrent transfers	no. of links
Latency	Link latency (cycles)	1
Bandwidth	Link bandwidth (bytes/cycle)	8
	Memory Controller	
Queue size	Access queue size	128
Data interleaving mask	Interleaving granularity (bytes)	4096
Number of DIMMs	Number of DRAM DIMMs	4
	Off-chip Memory	
number of banks	Number of banks per DIMM	8
autoprecharge	Enable/disable autoprecharge	disabled
data rate	Transfers per second (MT/s)	1600
burst	Number of bursts per access	8
$^{t}_{\mathrm{RCD}}$,$^{t}_{\mathrm{RP}}$,CL,$^{t}_{\mathrm{RC}}$,$^{t}_{\mathrm{WR}}$,$^{t}_{\mathrm{WTR}}$	Timing parameters	[4]

Table 5.1: TaskSim simulation parameters.

[4] Default values for DRAM timing parameters match the Micron DDR3-1600 specification.

For trace filtering we use a 16 KB direct-mapped cache with 64-byte cache lines. This configuration is the largest direct-mapped cache with the same number of sets than our target 32 KB 2-way set-associative L1 cache. This and the rest of simulation parameters are shown in Table 5.1. These parameters are the same as in the experiments in the previous chapter for the interconnect, memory controller and off-chip memory. The cache parameters are different because in this chapter we evaluate a two-level cache hierarchy instead of one with three levels. The reason is that this way we can tune the L2 cache latency to show the impact of different penalties of not simulating L1 misses due to invalidations.

Name	Description	Reuse	Parallelism	Bound type
LU	LU decomposition	High	Irregular	Compute
Cholesky	Cholesky factorization	High	Irregular	Compute
Reduction	Array reduction	Low	High	Memory
Vecop	Vector addition	No	Massive	Memory
Matmul	Matrix multiplication	High	High	Compute

Table 5.2: OmpSs benchmarks.

For these experiments, we use a set of scientific kernels written in OmpSs, whose characteristics are listed in Table 5.2. These scientific kernels are widely used in high-performance scientific applications and cover different levels of data reuse, parallelism and computation-versus-memory ratio. The benchmarks target fine-grain parallelism, so we are able to evaluate the effects of an intensive scheduling and synchronization operation.

We simulate the benchmarks for multiple numbers of cores (to see the effects of parallelism), multiple L2 cache latencies (to see the effects of increased L1 invalidation penalties), and for multiple chip frequencies (to see the effects of an increased penalty for off-chip memory accesses).

5.5.1 Trace Size

The trace size reduction of trace stripping techniques depend on the locality exhibited by the target application. High hit ratios result in more accesses being discarded in the trace filtering process and smaller trace files. Therefore, we expect applications with a higher data reuse to have higher hit ratios and, as a result, higher trace size reductions.

Figure 5.8 shows the trace size reduction of naive and reset with respect to the full trace size. As expected, the benchmarks with lower data reuse, Reduction and Vecop, exhibit lower reductions than its high-data-reuse counterparts. The use of filter cache reset shows a 4% drop reduction compared to naive on average, with a maximum loss of 9% for Vecop. This is also expected, as Vecop is the benchmark that has more parop calls because it exploits the finest-grain parallelism among all benchmarks due to its massive parallelism scheme (all tasks are independent).

5.5.2 Trace Generation Time

Trace generation time is heavily affected by disk write performance. For the implementation of our methodology, we employ PIN, which is faster than us-

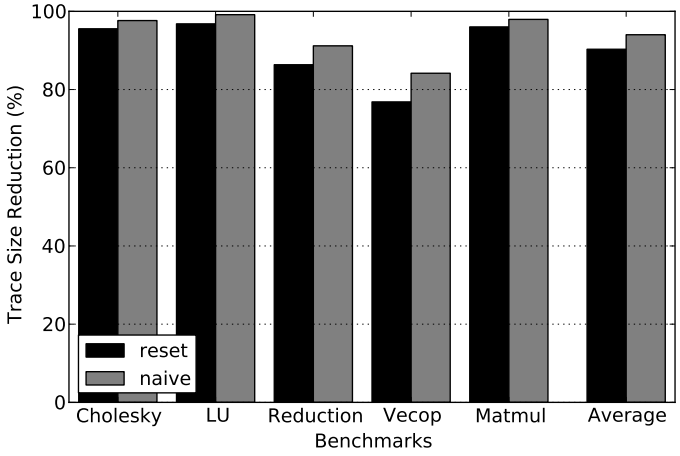


Figure 5.8: Trace size reduction.

ing an execution-driven simulator. Since dynamic binary instrumentation is so fast, adding the simulation of the filter cache for every memory access in the application incurs a significant performance penalty.

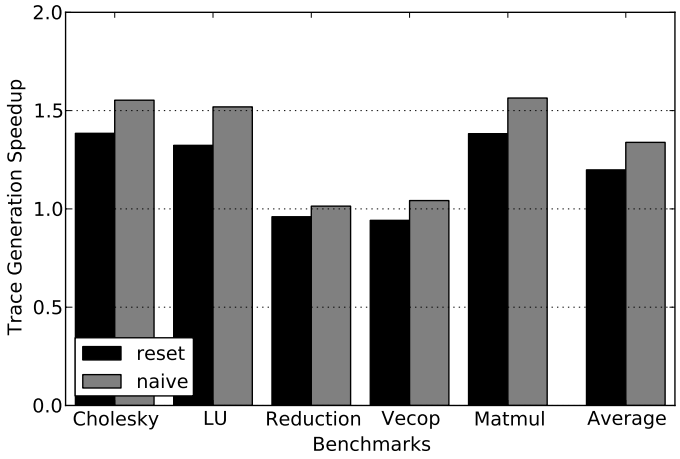


Figure 5.9: Trace generation speed-up.

Figure 5.9 shows the trace generation time reduction of naive and reset compared to the time required to generate the full trace. The host machine for these experiments is a Linux box with a quad-core Intel i7-930 at 2.8GHz, 12GB of RAM and making use of a remote NFS disk to store the traces. Although the use of remote storage increases the latency of disk write operations, the simulation of the filter cache cancels out the benefit of having to record less accesses for naive and reset. In the previous section we have seen trace size reductions over 90%. However, trace generation speed-up is just 1.19x for reset and 1.33x for naive on average. This is due to the overhead of the filter cache

simulation. In our proposal, the process of resetting the filter cache implies an extra cost that makes trace generation longer than for the full trace in the most memory intensive benchmarks Reduction and Vecop.

In any case, trace generation is a one-time operation so, in most cases, its cost is not critical.

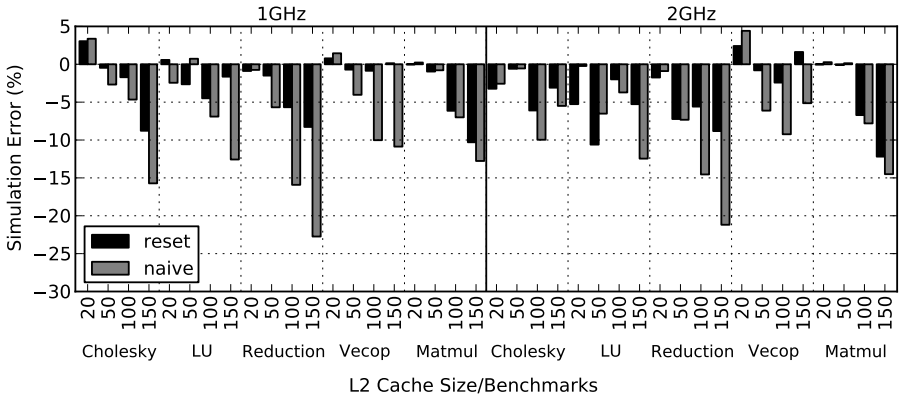
5.5.3 Simulation Accuracy

In this section we give a measure of how our proposal reduces the simulation error of the state-of-the-art naive approach. Figure 5.10 shows the simulation error introduced by trace filtering in both reset and naive having the full trace as the baseline. The figure shows three charts, each of them for a different number of simulated cores: four, eight and sixteen. At the same time, each chart is divided in two halves: the left half is the configuration with 1 GHz cores, and the right half for 2 GHz cores. The x-axis shows groups of L2 cache latencies, each group for a different benchmark. The L2 cache latencies are 20, 50, 100 and 150 cycles, to be consistent with the experiment shown in Section 5.3. Although the largest latencies are unrealistic, we want to show these extreme cases as an upper-bound of simulation error for trace filtering, where L1 misses have a large penalty, even if the data is reused on chip (i.e., they hit in L2 cache). Also, these large penalties are closer to the effect of in-order cores. Our out-of-order configuration is able to hide a 20-cycle latency, something an in-order core could hardly hide.

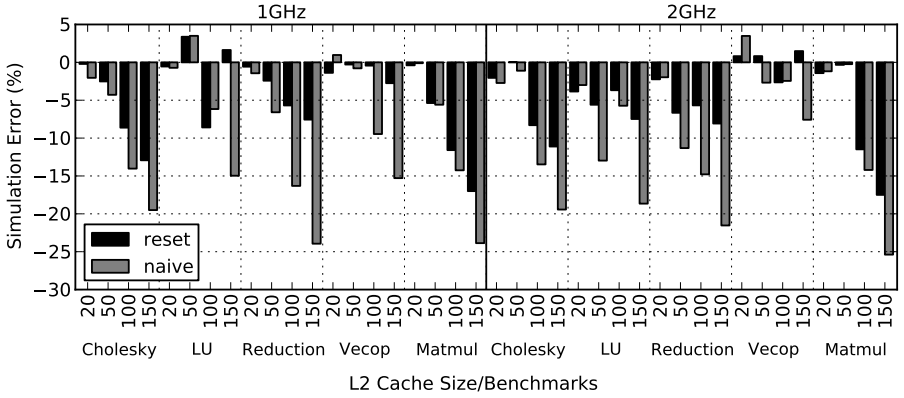
We expected the results of reset to be closer to the simulation of the full trace. However, we have found quite a lot of variability during the simulations due to dynamic scheduling decisions (butterfly effect). Slight differences in timing due to filtering make tasks to be scheduled to different threads in the different configurations which leads to completely different data reuse patterns in the private caches.

There are a few cases where our methodology has a larger error than naive. In most of these cases, apart from the variability inherent to dynamic scheduling, we have found the limitation of filtered traces to handle ping-pong effects due to *false sharing*. This limitation is the same for both reset and naive, but due to dynamic scheduling, reset missed more false sharing conflicts than naive in the cases where reset has a larger error (the conflicting tasks were scheduled in the same thread for naive, and to different threads in the same time frame for reset). The problem with false sharing is that, even if the first accesses to every cache line after a parop execution are recorded, consequent hit write accesses may be accessing a cache line containing data written by another thread. The threads do not share the data, but both data reside in the same cache line that is invalidated on every write. The full trace recreates correctly these scenarios, but filtered traces cannot because the accesses that may ping-pong are not in the trace. A possible technique to handle this would be to avoid filtering writes, and only filter reads. However, this may be too aggressive so we think that further investigation is needed for this purpose.

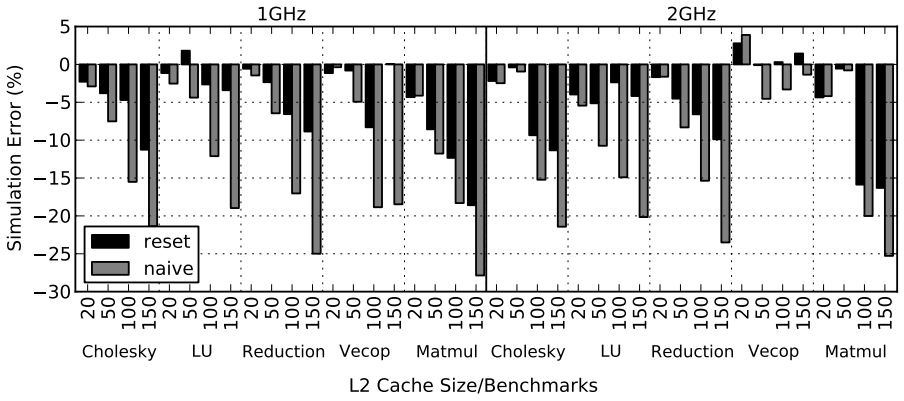
However, even considering these effects, our methodology consistently reduces the simulation error incurred by the incorrect filtering of the naive methodology. Figure 5.11 shows the average across all benchmarks and frequencies for the different multi-core configurations. On average, reset halves the error of naive from 9% to 4.5%.



(a) Simulation error for 4-core CMP



(b) Simulation error for 8-core CMP



(c) Simulation error for 16-core CMP

Figure 5.10: Simulation accuracy.

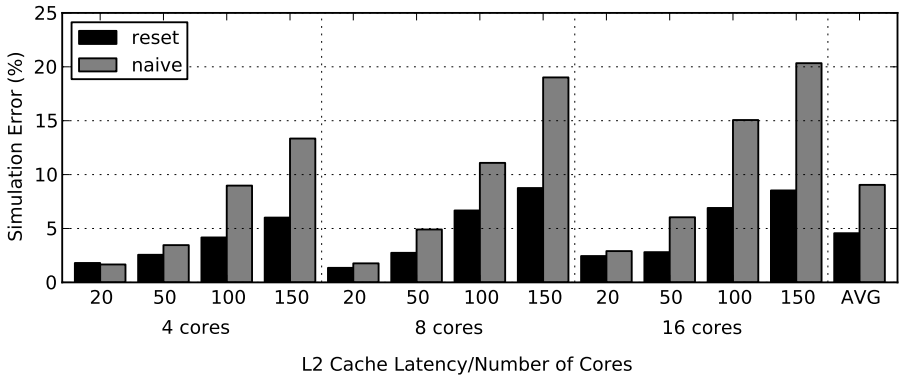


Figure 5.11: Simulation accuracy average across benchmarks and frequencies.

5.5.4 Simulation Speedup

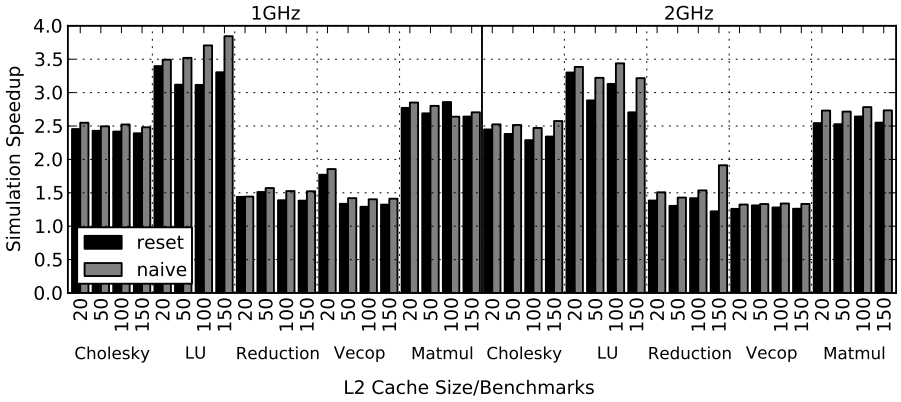
The trace size reductions shown in Section 5.5.1 promise a large reduction in simulation time. However, most of the accesses in the full trace require the simulation of an L1 hit, while most of the accesses in the filtered traces miss. An access to L1 has a lower cost in terms of simulation time compared to an L1 miss. Misses require the simulation of an access to the shared interconnection to the L2 cache and the access to the L2 cache. In the case of also missing in L2, then it further requires the simulation of an access to the memory controller and off-chip memory. Then, the simulation time cannot be proportional to the number of memory accesses, as not all of them have the same simulation requirements. Therefore, simulation time reductions close to the trace size reductions shown previously seem optimistic.

Figure 5.12 confirms this reasoning, but still shows large reductions accounting up to 3.8x simulation speed-ups in the benchmarks with larger data reuse. In the benchmarks with low data reuse, however, the simulation speed-up is below 1.5x in most cases. From the chart we can also confirm that our methodology is close to naive in terms of simulation time. Except for some cases, such as the Vecop benchmark on a 1 GHz CPU and an L2 cache with a 150-cycle latency, our methodology is generally within 10% of naive.

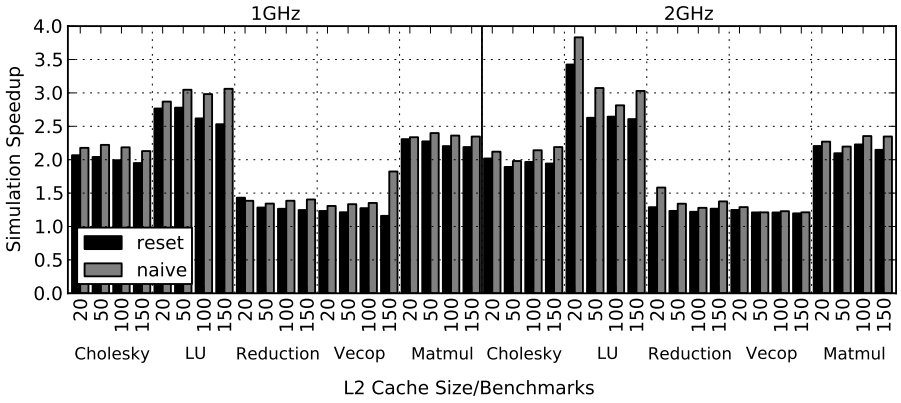
Figure 5.13 shows the simulation speed-up average across benchmarks and frequencies. Simulation speed-up decreases for simulations with higher numbers of cores. In this case, the cost of simulating more cache coherence actions reduces the benefit of saving the simulation of L1 hits in filtered traces. Nevertheless, simulation speed-ups range from 1.5x to 2.3x for reset compared to 1.6x to 2.4 for naive. Overall, reset is just 9.5% slower than naive and 1.9x faster than simulating the full trace.

5.6 Limitations

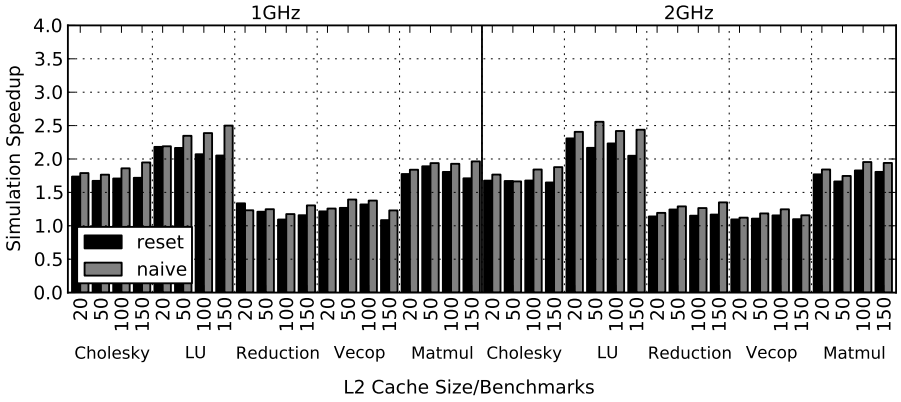
There are a few limitations that must be noted by anybody who wants to employ this methodology. The first is that, since our methodology is for trace-driven simulation, it inherits the limitations inherent to the use of traces, such as the inability to simulate the wrong-path memory accesses on speculative execution.



(a) Simulation time improvements for 4-core CMP



(b) Simulation time improvements for 8-core CMP



(c) Simulation time improvements for 16-core CMP

Figure 5.12: Simulation speed-up.

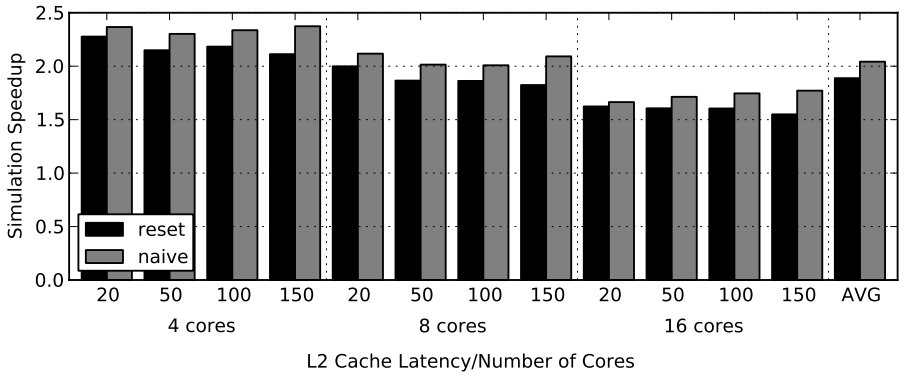


Figure 5.13: Simulation speed-up average across benchmarks and frequencies.

Also, the use of trace-driven simulation prevents the simulation of the full system. The behaviour of the operating system depends on the timing of certain components, such as varied I/O timing when modeling I/O components, which is not possible to capture in traces.

The limitations inherent to our methodology can be summarized in two: race conditions and false sharing. Our methodology assumes the multithreaded application is properly synchronized, so it is race-condition free. If race conditions are possible, the application could just break in a real execution, so the interest in such kind of simulations is expected to be minimal. The case of false sharing is a more important limitation. As explained in Section 5.5.3, memory access hits during trace generation may miss in simulation due to an invalidation from another thread writing to data residing in the same cache line, even though the data is not shared between the threads. Programmers and compilers add padding to data structures to avoid false sharing due to the potential impact it may have in performance. However, false sharing can be found in real multithreaded programs, so more investigation is needed to tackle this problem.

5.7 Summary

In this chapter we presented the first attempt to enable trace filtering for simulating multithreaded applications running on multi-cores. We have provided a deep explanation of the problem addressed and we have shown its potential impact with a synthetic application including a pathological case exhibiting the problematic inherent to the use of trace filtering as it is done in previous works.

We implemented our methodology based on the structure of OmpSs applications, and using dynamic binary instrumentation to generate the full and filtered traces to feed in the mem mode of TaskSim. We show a thorough evaluation of the proposal in terms of trace size reduction, trace generation speed-up, simulation error and simulation speed-up compared to the state-of-the-art technique and to simulating the full/non-filtered trace. Our methodology consistently reduces the simulation error of the state of the art in 50% and just implies a mere 9.5% loss in simulation performance on average.

This poses a trade-off between simulation error and simulation speed-up.

The average error of our technique is 4.5% in our experiments and simulation time is reduced by half. Research works evaluating small differences in performance may prefer to reduce the error margin and simulate the full trace even if it takes twice the time. On the other hand, works requiring very long simulations (several days or weeks) may want to benefit from a 2x cutdown in simulation time at the expense of a relatively small simulation error.

Chapter 6

Modeling the Runtime System Timing

The methodology presented in Chapter 3 simulates sequential sections using traces and executes parops directly on the host machine at simulation time. Using native execution is fast but difficults the timing modeling of parop execution on the simulated machine because parop execution is not exposed the simulation engine.

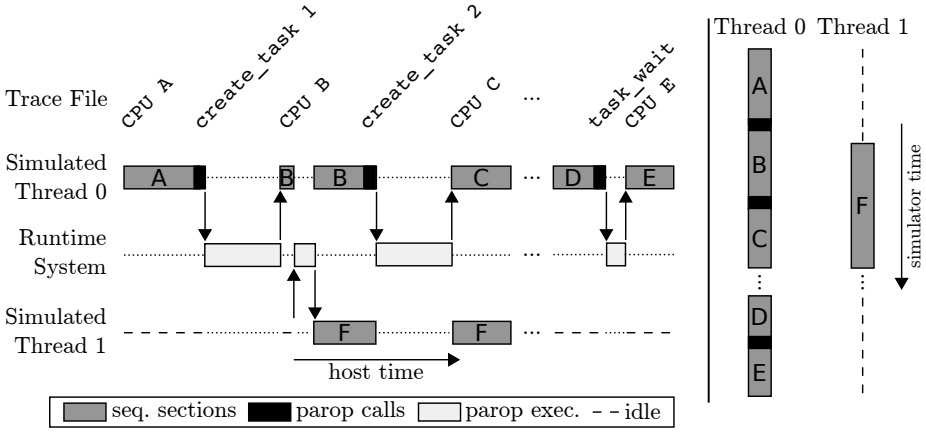
Runtime-managed applications are usually partitioned keeping in mind the overhead of parop execution so the time spent on parops is negligible compared to the time spent in sequential sections between them. However, in applications using fine-grained parallelism to scale to large numbers of cores, parop execution may have a significant performance impact.

In this chapter, we evaluate the importance of modeling the timing of parops and present our attempt to use a fast high-level model for parop timing based on their execution on the host machine.

6.1 Problem

Figure 6.1 shows the timeline of a simulation using our methodology in Chapter 3. The simulated application is task-based, runs on a configuration with two simulated threads and uses a master-worker scheme where one thread creates tasks and another one executes them. Figure 6.1(a) shows the application trace file content and the corresponding simulation stages on the multiple simulation components, namely the two simulated threads and the integrated runtime system. The trace file includes sequential sections, shown as `CPU` events, and parop call events. Simulation starts in Thread 0 and, after a first sequential section (`A`), simulation finds a `create_task` event. Then, the simulation engine calls the runtime system through the Runtime Bridge (see Figure 3.4) to create Task 1. This operation is executed by the runtime system, which takes some host time and, having finished, simulation resumes. Then, Thread 1, which was idle waiting for tasks (dashed line), asks the runtime system for a ready task, and it is assigned the recently-created Task 1. Thread 1 starts simulating Task 1 (which has its separate trace not shown in the figure).

Simulation proceeds analogously in Thread 0 with the creation of Task 2,



a) Simulation Operation b) Simulated Timing

Figure 6.1: Simulation example of a task-based application running on two threads. (a) The simulation alternates between the simulated threads and the runtime system operation, to simulate the actions specified in the events included in the trace. The runtime system performs the creation of tasks, and assign tasks to idle threads (dashed line), such as Task 1 to Thread 1. (b) The simulation engine does not account the timing of the runtime system operations, and only simulates the timing of the sequential sections in the trace.

and so on. Later, close to the end of the application, a task wait event is found, which forces Thread 0 to wait for all previously-created tasks to complete. The simulation engine calls the runtime system for this synchronization primitive and, when executed, it founds that all tasks have finished. Simulation resumes and completes after simulating the last sequential section.

At simulation time, the simulator alternates between timing simulation in the trace-driven simulation engine and parop execution in the runtime system. The problem is that the trace-driven simulator is not aware of the executed runtime system operations, and it can only model the timing of sequential sections in the trace, as it is shown in Figure 6.1(b). This way, the simulator is able to reflect the impact of the runtime system decisions on the application execution, such as tasks being scheduled to different threads. However, it does not account for the overhead of the runtime system operations themselves.

A straightforward approach to account for parop timing is to use the time taken by the operation during trace generation as an estimate for its execution time. The problem with this technique is that the trace generation run uses a specific runtime configuration, including options such as scheduling and cutoff policies (see Section 2.7). Therefore, the problem with this technique is that the timing of parops would be the same (the one at trace generation time) regardless of the runtime system configuration used during simulation.

The work presented in this chapter has therefore two objectives. First, evaluate the variability of parop timing for different runtime system and simulated machine configurations. And second, to model the timing of parops so simulation reflects the performance impact of different runtime system configurations in the predicted execution time on the simulated machine.

6.2 Runtime Analysis

Generally, runtime system operations take a small portion of the total application time: applications are appropriately partitioned, and runtime systems are highly optimized, both to minimize runtime system overheads. However, not all applications can be easily repartitioned, as the problem size for a determined computation slice may be fixed, for example, the size of a macroblock in video decoding applications. Also, advanced runtime system policies, such as locality-aware schedulers, may provide a better application performance, but their costly operation may cancel out their benefits, even leading to performance degradation in some scenarios. In other words, the overhead of a better scheduler may cancel out the benefits of its better scheduling.

In order to understand the factors affecting the performance of runtime system operations, we analyze their computation on different scenarios and the possible situations of contention on shared resources. The object of this study is the NANOS++ runtime system, which is the one used along this thesis. Also, we focus on the parops used in task-based parallel applications written in the OmpSs programming model (see Section 2.7.1).

One of the factors introducing performance variability in NANOS++ parops is the use of cut-off mechanisms (see Section 2.7), referred to as *throttling*. Whenever the application wants to create a new task, the throttling mechanism decides whether to actually spawn the new task and add it to the task dependence graph, so any thread can execute it, or tell the thread creating the task to execute it *inline*. This mechanism aims to alleviate the cost of adding more tasks to the runtime system structures when there is no need for more parallelism, such as when all treads are busy and there are many pending tasks.

Examples of throttling metrics are the total number of in-flight tasks and the total number of ready tasks at a given point in time. When the metric is above a certain threshold, referred to as *throttle-limit*, additional tasks are executed *inline* rather than created and *submitted* for execution by other threads.

Another important factor introducing parop variability is the scheduler. It makes decisions regarding task queuing and execution at certain points of the application, such as on task creation, task completion and when a thread becomes idle. For this, it manages ready task queues. A ready task queue includes a set of tasks ready for execution (dependencies satisfied), and may be global or thread-private. Different schedulers may have different policies, such as breadth-first and work-first [70], and therefore have different algorithm complexities.

Figure 6.2 shows the pseudo-code of four of the main runtime system operations in NANOS++. For each of these operations, we indicate whether it accesses or updates the task dependence graph (DG) or the ready task queues (RQ). Figure 6.2a shows the code for task creation. As previously mentioned, due to the throttling mechanism, the execution may follow different paths with potentially different costs. Also, if a task is actually *submitted* (no throttle), and does not have any potential dependencies, it is considered independent, and is directly pushed to a ready task queue. However, if it has potential dependencies, it must be checked against previous tasks in the task dependence graph and, if there are dependencies, the graph is updated accordingly with the new task.

Another example of variability is the acknowledge of task completion, shown in Figure 6.2b. The operation after completing a task must iterate over its

```
if throttle task creation:
  //execute task inline
  if task has predecessors:  →check DG
    execute predecessors
  create task object & execute
else:
  //create and submit task
  create task object
  if task has possible deps:
    if task does have deps:  →check DG
      add task to dep graph  →update DG
    else:
      add task to ready queue→update RQ
  else:
    add task to ready queue  →update RQ
```

(a) Task Creation

DG: dependence graph
RQ: ready task queue

```
decrease pending tasks          →update DG
foreach task successor:
  release successor dependency  →update DG
  if successor ready
    add successor to ready queue→update RQ
if select ready task:           →update RQ
  execute task
else:
  execute idle loop
```

(b) Task Completion

```
while pending tasks > 0:     →check DG
  //select & exec ready tasks
  if select ready task:      →update RQ
    execute task
```

(c) Task Wait

```
while thread is running:
  //select & exec ready tasks
  if select ready task:      →update RQ
    execute task
```

(d) Idle Loop

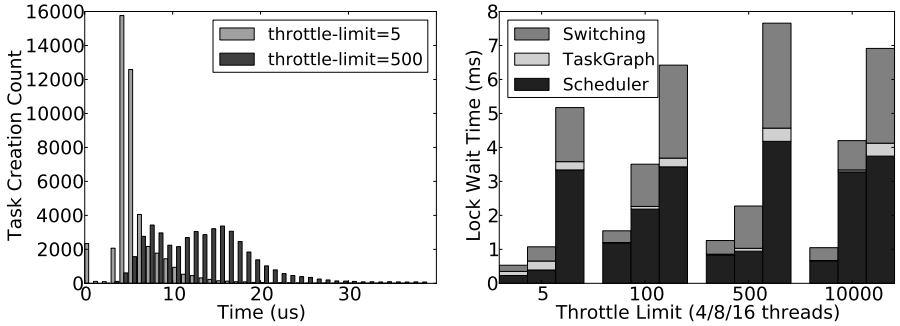
Figure 6.2: Simplified pseudo-code of the main NANOS++ operations, and the actions on shared resources: task dependence graph and ready queue. Checks and updates of shared data are protected by locks or atomic operations depending on their granularity.

successor tasks, meaining other tasks depending on its output. Its cost depends on the total number of successors and if they become ready. Also, for the operations Task Wait (Figure 6.2c), Idle loop (Figure 6.2d) and Task Completion (Figure 6.2b), the scheduler must select a ready task for execution. At this point, the parop cost greatly depends on the complexity of the scheduler algorithm. A simple policy, such as oldest-first, has a lower cost than a policy checking all tasks in the queue. In this case, the cost depends on the number of tasks in the queue and, if empty, on the check of other queues for *work stealing*.

From Figure 6.2, we can also identify the chances of contention among threads. For instance, the task dependence graph may be a center of contention when a thread is completing a task and updates the successors state in the graph, while another thread is creating a task with dependencies, and wants to add it to the graph. In this case, both threads must compete for a lock before entering the critical section that updates the task dependence graph. Another source of contention is the scheduler's ready task queues. An example is when a thread is idle waiting for tasks and checking the ready task queue and, at the same time, another thread creates a task without dependencies and wants to add it to the same ready task queue.

From this analysis, we conclude that the cost of runtime system operations depends on the taken control flow path, the algorithm complexity and the data size on which they operate. Then, apart from the computation cost, an extra overhead has to be considered when there is contention on parops accessing shared resources from separate threads.

Figure 6.3a shows a histogram of the time taken by task creation operations during the execution on a real machine of a Cholesky factorization (see Section 2.11) on eight threads. The figure shows a histogram for two different



(a) Task Creation Time Distribution (b) Average Lock Wait Time per Thread

Figure 6.3: Runtime system operation variability. (a) Task creation cost distribution of Cholesky factorization run on eight threads for two different throttling limits. (b) Breakdown of the lock contention time of Cholesky decomposition run on four, eight and sixteen threads for four different throttling limits.

throttling limits, 5 and 500. Whenever the total number of in-flight tasks is above the limit, new tasks are executed *inline* instead of following the regular submission to the task graph or ready queues. Hence, a lower throttle limit generates more *inlining* than a higher one. The results in the figure show that using a throttle limit of 5 leads to lower task creation costs than using a limit of 500, as many more tasks are *inlined*. Most of the *inlined* creations take less than 10 us, while the regular creation and submission takes between 5 and 20 us.

Figure 6.3b shows the average time per thread spent waiting for acquiring a lock, and its breakdown depending on the operation being synchronized. This is shown for Cholesky factorization using multiple numbers of throttle limits and run on four, eight and sixteen threads. The results show two effects. The first is that lock contention increases with the number of threads for all throttling limits. This is because there are more chances that separate threads want to access a shared resource when there are more threads executing in parallel. The second is that the contention on locks also increases after a certain throttle limit. The contention for locks with limit 5 is significantly lower than for larger limits. This is because in *inlined* task creations the runtime system does not need to access shared resources.

6.3 Runtime Modeling

The results in the previous section show that the cost of parops is sensible to the runtime system and simulated machine configurations, and how these affect the contention on the runtime system shared resources. Therefore, in situations where parops account for a large portion of the application, or when multiple runtime system configurations are compared in terms of overhead, it is necessary to account for time spent in parop execution.

In this section, we introduce a fast high-level timing model for parops, to be used in the simulation methodology in Chapter 3. The objective is to provide an estimate of parop execution time, while maintaining the cost of the model

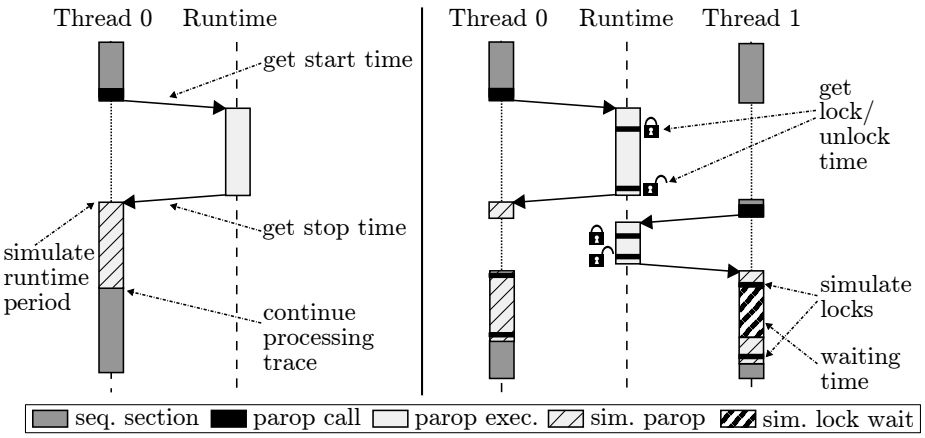


Figure 6.4: Simulation example with host time only (left) and host time plus simulation of locks (right).

low, so it is fast enough to be used in multiple levels of abstraction from detailed to high-level ones.

The timing model lies on top of a light-weight instrumentation library that captures the execution time of parops in the host machine. This gives a measure of the cost of that operation considering the runtime system state during simulation. As the runtime system data state depends on the simulated architecture, the control flow path followed by parop execution, the algorithm complexity and the data size on which it operates are reflected by the parop execution time on the host. Then, the host execution time can be factored to match the simulated machine performance. The use of performance factors has already been explored in existing simulation tools [24, 155], and several databases of performance factors are publicly available [9, 158].

From the previous section we know that, apart from parop computation, contention on shared resources also depends on the runtime system configuration. On a sequential simulator, the simulation of different threads is serialized and, thus, the execution of parops on different simulated threads is also serialized. Therefore, all lock acquire operations are successful and the potential contention is not reproduced. To account for contention, the lock acquire and release operations are also instrumented. As a result, when the control comes back to the simulation engine after a parop execution, the simulator has a *trace* of sequential sections and lock operations of the executed parop that is used to model its timing.

Figure 6.4 illustrates how the parop execution information is captured and used for simulation. The figure shows the methodology's operation for a parop execution without (left) and with (right) locks. Without locks, on a parop execution, its starting and finishing times are recorded, and its total execution time is computed. The resulting trace is a single sequential section with a given execution time (on which a performance factor can be applied to match the simulated machine performance). Finally, the resulting sequential section time is accounted in the parop-calling simulated thread, that resumes processing the trace afterwards.

The example with locks shows how lock acquire and release operations, that do not suffer from contention during the execution on the host machine, are also included in the resulting trace. In Figure 6.4 (right), both threads execute parops that are serialized. However, when they simulate both parop traces, Thread 0 acquires the lock first, and then Thread 1 fails to acquire the same lock and has to wait before Thread 0 releases it, thus reproducing the contention that would exist on that application running on the simulated machine.

6.4 Evaluation

In this section we present a set of experiments to show the impact of throttling limits, scheduling policies and numbers of threads on parop execution and how our methodology models their timing.

We use the integrated TaskSim-NANOS++ simulation infrastructure presented in Chapter 3 and simulate with the burst mode presented in Chapter 4. The experiments compare the following configurations:

- Real: execution on the real machine.
- Naive: simulation using as parop execution time the one from the trace generation run (explained in Section 6.1).
- Host: simulation using as parop execution time the one from the host machine during simulation (Figure 6.4 (left)).
- Host+Locks: simulation using as parop execution time the one from the host maching during simulation and also simulating lock contention from parops in different simulated threads (Figure 6.4 (right)).

For the first experiment we use a blocked-matrix multiplication with 64×64 blocks of single-precision floating-point elements written in OmpSs. We execute the application in the real machine and using the three aforementioned simulation configurations. All experiments are done with two different schedulers: *default* and *cilk*. Default uses a global ready queue and a LIFO policy. Cilk uses one queue per thread and, on task creation, it starts executing the created task and the parop-calling task gets enqueued for other thread to resume its execution. Executions on the real machine are repeated five times and averaged to counteract OS noise.

Figure 6.5 shows the normalized execution time on the real machine and on simulation using the naive, host and host+locks approaches. The x-axis shows throttling limit values from 1 to 9999999 (infinite) and each data series is for a given number of threads. The results are normalized to the execution with throttling limit 1.

The naive configuration is able to reproduce the lack of parallelism of throttling limits below 50, but is not able to reproduce the loss of performance for larger throttling limits due to parop overhead. The introduction of the host time in the *host* configuration is able to account for part of that overhead but does not make as much difference as in the real machine. The simulation of locks gets the predicted execution time closer to the one of the real machine but it does not follow the same trends and does not match its magnitude either.

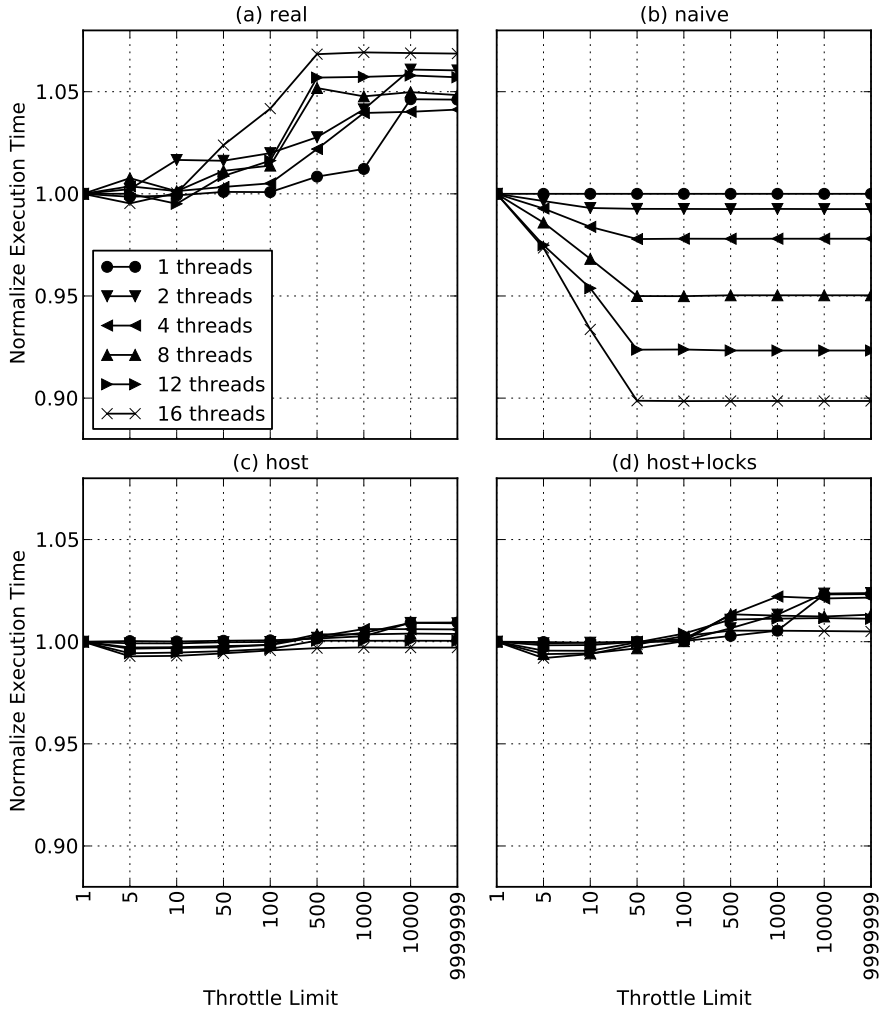


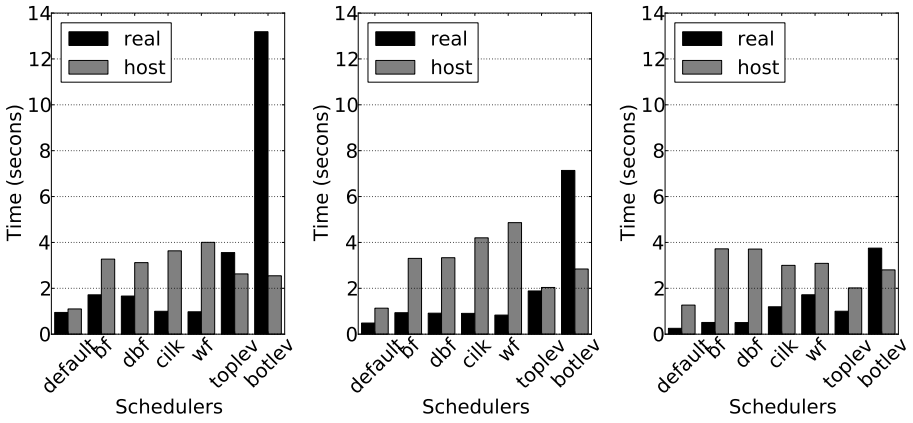
Figure 6.5: Normalized execution time real (a) and simulated (b,c,d) for a blocked-matrix multiplication using 64×64 blocks.

In any case, parop execution time in these experiments is not significant, and the results are in the noise margin which can be affected by the inherent inaccuracies of the burst mode and OS noise during the real, trace generation and simulation executions.

To get a more precise view of the effects of our methodology, we carry out a second case study using an H.264 decoder *skeleton* application written in OmpSs. This skeleton has the same structure as a real implementation of an H.264 decoder using the 3D wave optimization [117]. In this case, we focus on the parop timing across multiple schedulers and numbers of threads and use a throttling limit of 100 for all experiments.

The experiments are done for a set of schedulers including the ones in the

NANOS++ distribution (default, bf, dbf, cilk, wf) and two other schedulers using priorities depending on the chain of dependencies a given task has to the top of the task dependence graph (toplev) and to the bottom of the graph (botlevel). These schedulers are explained in more detailed in the PhD Thesis of Paul Carpenter [54]. The complexity of toplev and botlev is higher than for the other five scheduler, because they have to traverse the task dependence graph to assign priorities to tasks.



(a) Four threads (b) Eight threads (c) Sixteen threads

Figure 6.6: Task creation execution time of the H.264 decoder skeleton both in the real machine and using the *host* simulation approach using multiple numbers of threads.

Figure 6.6 shows the task creation execution time for the real machine (a) and using the *host* simulation approach (b) for the aforementioned schedulers and multiple numbers of threads. In the real machine, the differences for the five schedulers in the NANOS++ distribution are not significant and can be considered noisy. However, the cost of toplev and botlev is very significant. However, the *host* simulation approach is not able to reproduce that cost. We have checked the source of the overhead for these scheduler in the real machine, and it comes from the increased complexity of the task graph insertion rather than from the contention in locks, which is minimal. Therefore, the simulation of locks is not expected to improve this prediction.

6.5 Discussion

The experiments in the previous section show that our high-level model is not able to accurately reproduce the variability of different runtime system configurations at simulation time. The fundamental deficiencies of the method are related to caching effects due to the following facts:

- Serial vs. parallel execution. The simulator runs on a single host thread while the real application executes on multiple separate threads. This has two effects. The first is that parops executing in different threads will

have different cache states in their privates caches while the simulator has a single private cache in which the state of the runtime system can be kept. The second is that the cache coherence actions that happen in the real machine from accesses to shared runtime system resources from separate threads are not present during simulation either.

- Application data vs. simulator data. In the real machine, between two parop calls, the application computation may have completely evicted all runtime system data from the caches. In the simulator, this depends on the level of abstraction of the timing model of sequential sections between parop calls. In a high-level mode such as the burst mode, the computation between parop calls is small and the runtime system data may be kept in the private caches and the accesses in the next parop call will hit in the cache.

These two deficiencies lead to the underestimations of parop execution for some schedulers seen in the previous section.

Another deficiency of the method is instrumentation overheads. The execution of some parops is short and the insertion of instrumentation points can lead to overestimations when simulating the timing of parops using the host and host+locks approaches, as we have seen for some schedulers. A possible solution to this problem is to measure the typical delay of an instrumentation call and subtract it from the measured parop execution time.

As a result, the parop timing models introduced in this chapter are not enough for an accurate reproduction of the time taken by runtime system operations. Further investigation is needed to quantify the deficiencies analyzed in this discussion and propose more detailed models that actually account for the runtime system effects when the cost of the runtime system operations is of interest of the research study being carried out.

6.6 Summary

In this chapter we explained the problem of modeling the timing of the runtime system operations using our simulation methodology in Chapter 3. We showed the sources of variability for runtime system operations and proposed two high-level models for the timing of these operations based on their execution on the host machine.

Our experiments showed that our models are able to partially mimic the trends of parop timing and are definitely better than the naive approach, but generally result in underestimations. We discussed the reasons for these inaccuracies and concluded that further investigation is needed to quantify the deficiencies of these methods and improve the models to bridge the gap between the simulated and real parop execution times.

Chapter 7

Conclusions

In the history of microprocessor from 1972, the number of transistors per chip has doubled every two years. Due to power density issues, among others, this increasing number of chips has been used since the early 2000s to include more cores in the same chip.

The inclusion of multiple cores per chip, has changed the microprocessor design paradigm, which has an impact on single-thread performance. The design of multi-cores focuses on exploiting thread-level parallelism on multiple cores, rather than instruction-level parallelism on a single core. This leads to a slowdown on the increase of single-thread performance that went from doubling every two years until 2004, to doubling every three years and a half since then.

These two facts, the increasing trends of number of transistors per chip and single-thread performance, has an impact on computer architecture simulation, which is the topic in this thesis. Computer architecture simulation is important because it drives most computer architecture research. In this context, the number of transistors determines the complexity of the model to be simulated, and the single-thread performance of the host machine determines simulation speed. Since the number of transistors keeps doubling every two years, the complexity of the simulated model also increases at this rate. At the same time, single-thread performance, and thus, simulation speed, doubles every three years and a half. The difference is what we call the simulation speed gap, and makes simulation of multi-cores increasingly slow over time. As an example, if the simulation of a contemporary architecture takes one week, a simulation of a contemporary architecture in five years will take more than two weeks, and over a month in ten years.

The objective of this thesis is to close the simulation speed gap to reduce simulation time. Other researchers in the computer architecture community are aware of the problem and have proposed techniques to reduce simulation time, such as sampling and parallelization, that we covered in Section 2.4.

The approach in this thesis is to raise the level of abstraction to close the simulation speed gap and reduce simulation time. In this direction, we have proposed a set of simulation techniques and methodologies. In the next sections, we summarize our contributions and list their associated publications, explain works that have used our research, and provide some recommendations on how this work could be continued in the future.

7.1 Contributions and Publications

Our contributions in this thesis are the following:

- **A trace-driven simulation methodology** for dynamically-scheduled multithreaded applications. It combines a trace-driven simulation engine with a runtime system to simulate timing-independent user code using traces and execute timing-dependent parops at simulation time to make scheduling and synchronization decisions based on the simulated machine.

  [150] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, M. Valero. Trace-driven Simulation of Multithreaded Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 87–96, Apr. 2011.

  [151] A. Rico, A. Duran, A. Ramirez, M. Valero. Simulating Dynamically-Scheduled Multithreaded Applications Using Traces. *IEEE Micro*. Submitted for publication.

- **Two fast high-level simulation modes** for evaluating application scalability and accelerators with scratchpad memories. These simulation modes are based on our definition of application abstraction. At the highest level of abstraction, the burst mode, is faster than native execution and is accurate and useful to predict application scalability on larger numbers of cores than the ones available in the real machine. The inout mode has only an average 25x slowdown compared to native execution, and has been shown accurate and useful for the simulation of accelerator-based architectures. The most important fact is that these modes have been shown to be faster and more insightful than functional simulation in execution-driven simulators.

  [148] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, M. Valero. On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, 2012.

- **A trace generation technique** to include memory accesses sensible to cache invalidations in filtered memory access traces for the simulation of multithreaded applications running on cache-based multi-cores. The trace generation technique covers invalidations due to different thread interleavings and different dynamic scheduling decisions. It reduces the average error of the state of the art from 9% to 4.5%, while providing similar trace size reduction (10x smaller traces) and simulation speed-up (2x faster simulations) compared to the simulation of the full trace.

  [154] A. Rico, A. Ramirez, M. Valero. Trace Filtering of Multithreaded Applications for CMP Memory Simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, pages 134–135, Apr. 2013.

- **A high-level simulation model** for modeling the timing of executed parops in our simulation methodology. We found that our technique is not enough to faithfully model the timing of parops due to the different

application and cache states between simulation and real execution. Further investigation is needed to improve the model to better account for parop timing in the applications or studies where it is relevant.

7.2 Impact

In this section we explain other works we carried out related with the work in this thesis; works that use the simulation techniques proposed in this thesis; and other non-related works we have carried out during the course of this thesis.

7.2.1 Our Related Work

Related to the development of CellSim, we published a set of papers, poster abstracts, and a workshop presentation. In chronological order:

[49] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, E. Ayguadé. A module-based cell processor simulator. In *Advanced Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems 2006 Poster Abstracts*, ACACES '06, pages 237–240, July 2006.

[50] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, E. Ayguadé. CellSim: A Cell Processor Simulation Infrastructure. In *Advanced Computer Architecture and Compilation for Embedded Systems 2007 Poster Abstracts*, ACACES '07, pages 279–282, July 2007.

[147] A. Rico, F. Cabarcas, D. Rodenas, X. Martorell, A. Ramirez, E. Ayguadé. Implementation and validation of a Cell simulator (Cellsim) using UNISIM. In *3rd HiPEAC Industrial Workshop on Compilers and Architectures*, Apr. 2007.

[51] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, E. Ayguadé. CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator. In *XVIII Jornadas de Paralelismo (CEDI 2007: II Congreso Espanol de Informática)*, JP '07, pages 187–188, Sept. 2007.

Related to CellSim, we also carried out a full-day Tutorial at the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT) 2007:

[142] A. Ramirez, A. Rico, F. Cabarcas. CellSim: a Modular Simulator for Heterogeneous Chip Multiprocessors. `http://pcsostres.ac.upc.edu/cellsim/doku.php#cellsim_tutorial_documentation`, 15 Sept. 2007. Full-day tutorial at Parallel Architectures and Compilation Techniques (PACT) 2007.

Related to the development of TaskSim, we published a technical report at the Universitat Politècnica de Catalunya – BarcelonaTech:

[146] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable Simulation of Decoupled Accelerator Architectures. Technical Report UPC-DAC-RR-2010-14, Universitat Politècnica de Catalunya, June 2010.

Related to the analysis of the runtime system overheads, we published a journal and a conference paper related to the task creation bottleneck mentioned at several points along the thesis and explained in Section 3.5. In chronological order:

[152] A. Rico, A. Ramirez, M. Valero. Task Management Analysis on the Cell BE. In *XIX Jornadas de Paralelismo*, JP '08, pages 271–276, Sept. 2008.

[153] A. Rico, A. Ramirez, M. Valero. Available Task-level Parallelism on the Cell BE. *Scientific Programming*, 17(1-2):59–76, 2009.

7.2.2 Other Works using Our Work

During the course of this thesis we developed a set of simulation tools and methodologies. In this section we give a measure of the impact of our work listing the publications using our tools and methodologies for the evaluation of their proposals.

- List of publications using CellSim in chronological order:

  [112] D. Ludovici G. Gaydadjiev. SARC Power Estimation Methodology. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing*, Nov. 2007.

  [87] R. Giorgi, N. Puzovic, and Z. Popovic. Implementing DTA support in CellSim. In *Advanced Computer Architecture and Compilation for Embedded Systems 2008 Poster Abstracts*, ACACES '08, July 2008.

  [118] C. Meenderinck B. Juurlink. A Chip MultiProcessor Accelerator for Video Decoding. In *Proceedings 19th Annual Workshop on Circuits, Systems and Signal Processing*, Nov. 2008.

  [84] R. Giorgi, Z. Popovic, N. Puzovic. Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture. In *IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2009, pages 1–8, 2009.

  [85] R. Giorgi, Z. Popovic, N. Puzovic. Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 78–87, 2009.

  [86] R. Giorgi, Z. Popovic, N. Puzovic. Introducing Hardware TLP Support in the Cell Processor. In *International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '09, pages 657–662, 2009.

  [94] X. Huang, X. Fan, S. Zhang, L. Shi. Investigation on Multi-Grain Parallelism in Chip Multiprocessor for Multimedia Application. In *International Conference on Information Engineering and Computer Science*, ICIECS 2009, pages 1–4, 2009.

  [78] E. Fernández Abeledo. Implementation of Nexus: Dynamic Hardware Management Support for Multicore Platforms. Master's thesis, Delft University of Technology, 2010.

[90] C. Gou, G. Kuzmanov, G. N. Gaydadjiev. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 179–188, 2010.

[41] M. Briejer, C. Meenderinck, B. Juurlink. Extending the Cell SPE with Energy Efficient Branch Prediction. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 304–315, 2010.

[61] C. Ciobanu, G. Kuzmanov, G. Gaydadjiev, A. Ramirez. A Polymorphic Register File for Matrix Operations. In *International Conference on Embedded Computer Systems*, SAMOS '10, pages 241–249, 2010.

[119] C. Meenderinck, B. Juurlink. Nexus: Hardware Support for Task-Based Programming. In *14th Euromicro Conference on Digital System Design*, DSD '11, pages 442–445, 2011.

[23] A. Azevedo, B. Juurlink. An Instruction to Accelerate Software Caches. In *Proceedings of the 24th international conference on Architecture of computing systems*, ARCS'11, pages 158–170, 2011.

- List of publications using TaskSim in burst mode in chronological order:

[75] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, M. Valero. Task Superscalar: An Out-of-Order Task Pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '10, pages 89–100, 2010.

- List of publications using TaskSim in inout mode in chronological order:

[48] F. Cabarcas, A. Rico, A. Ramirez, Y. Etsion, C. Villavieja, A. J. Vega, M. Pavlovic, A. Quesada, P. Bellens, R. M. Badia, M. Valero. Castell: A CMP architecture scalable to hundreds of processors. Technical Report UPC-DAC-RR-CAP-2009-33, Universitat Politecnica de Catalunya, Nov. 2009.

[170] A. Vega, A. Rico, F. Cabarcas, A. Ramirez, M. Valero. Comparing Last-level Cache Designs for CMP Architectures. In *Proceedings of the Second International Forum on Next-Generation Multicore/-Manycore Technologies*, IFMT '10, pages 2:1–2:11, 2010.

[47] F. Cabarcas, A. Rico, Y. Etsion, A. Ramirez. Interleaving Granularity on High Bandwidth Memory Architecture for CMPs. In *International Conference on Embedded Computer Systems*, SAMOS '10, pages 250–257, 2010.

[141] A. Ramirez, F. Cabarcas, B. Juurlink, A. Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, G. Gaydadjiev. The SARC Architecture. *Micro, IEEE*, 30(5):16–29, 2010.

[62] C. B. Ciobanu, X. Martorell, G. K. Kuzmanov, A. Ramirez, G. N. Gaydadjiev. Scalability Evaluation of A Polymorphic Register File: A CG Case Study. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, ARCS'11, pages 13–25, 2011.

[96] S. Isaza, F. Sanchez, F. Cabarcas, A. Ramirez, G. Gaydadjiev. Parametrizing Multicore Architectures for Multiple Sequence Alignment. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 31:1–31:10, 2011.

[169] A. Vega, F. Cabarcas, A. Ramirez, M. Valero. Breaking the Bandwidth Wall in Chip Multiprocessors. In *International Conference on Embedded Computer Systems*, SAMOS '11, pages 255–262, 2011.

[16] M. Alvarez, F. Cabarcas, A. Ramírez, C. Meenderinck, B. Juurlink, M. Valero. Scalability of Parallel Video Decoding on Heterogeneous Manycore Architectures. Technical Report UPC-DAC-RR-CAP-2011-12, Universitat Politècnica de Catalunya, June 2011.

[157] F. Sanchez, F. Cabarcas, A. Ramirez, M. Valero. Scalable Multicore Architectures for Long DNA Sequence Comparison. *Concurr. Comput. : Pract. Exper.*, 23(17):2205–2219, Dec. 2011.

[171] N. Vujic, F. Cabarcas, M. Gonzalez Tallada, A. Ramirez, X. Martorell, E. Ayguade. DMA++: On the Fly Data Realignment for On-Chip Memories. *IEEE Transactions on Computers*, 61(2):237–250, 2012.

- List of publications using TaskSim in mem mode in chronological order:

[81] V. Garcia, A. Rico, C. Villavieja, N. Navarro, A. Ramirez. The Data Transfer Engine: Towards a Software Controlled Memory Hierarchy. In *Advanced Computer Architecture and Compilation for Embedded Systems 2012 Poster Abstracts*, ACACES '12, pages 215–218, July 2012.

7.2.3 Our Non-Related Work

From the work during the internship at the IBM TJ Watson Research Center, we published the following work:

[149] A. Rico, J. H. Derby, R. K. Montoye, T. H. Heil, C.-Y. Cher, P. Bose. Performance and Power Evaluation of an In-line Accelerator. In *Proceedings of the 2010 ACM International Conference on Computing Frontiers*, CF'10, pages 81–82, May 2010.

The work during the internship at ARM Ltd. led to the following collaboration:

[140] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, A. Ramirez. Experiences with Mobile Processors for Energy Efficient HPC. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 464–468, 2013.

[139] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the Case for an ARM-Based HPC System. *Future Generation Computing Systems*, 2013. Accepted for publication.

7.3 Future Work

From the work presented in this thesis, we propose two research lines as future work:

- **Exploration of programming models other than OmpSs.** The techniques in this thesis are evaluated in the context of the OmpSs programming model. This serves as a proof of concept of the use of our techniques for a task-based programming model. Based on this experience, we explained in Section 3.4.2 how our trace-driven simulation methodology for multithreaded applications could be applied to other programming models.

  We do not know about fundamental differences of other task-based programming models that would prevent their runtime system to be integrated with a trace-driven simulator. However, difficulties may arise while carrying out this integration, which makes it a very interesting future work. Also, this would allow the comparison of different programming models for future large chip multiprocessors, which is an interesting topic in itself.

- **Abstraction of power models.** Power models are generally a set of calculations based on the technological and architectural characteristics of the target microprocessor, and on the utilization statistics of the different components by the target application. This applies both to simulation-based power models [160, 110], and even to power management units in real microprocessors (Intel's Running Average Power Limiting [95], AMD's Application Power Management [14] and IBM's POWER7 Power Proxy [79]).

  In Chapter 4 we introduce a set of high-level timing models, and explain their usability and accuracy. An interesting future work is to explore abstract power models based on these high-level timing models. The granularity of the utilization statistics generated by these timing models is very coarse. Thus, developing abstract power models based on these coarse-grained collection of statistics while achieving a reasonable degree of accuracy is a challenging task.

- **Performance ratios for cross-machine burst simulation.** The burst mode uses the execution time of sequential sections to predict the execution time of the application running on the simulated machine. As we showed, this is useful and accurate to predict the scalability of an application on larger numbers of cores than those available in the real machine.

  An interesting idea is to use the trace generated on a machine to simulate a different machine using the burst mode. As explained in Chapter 4, performance ratios can be applied to the duration stored in the trace to mimic the execution time for a given sequential section, or a set of sequential sections, on a different machine. Some existing simulators [24, 155] use performance ratios in the context of cluster simulation. However, there are no works assessing the validity of such performance ratios for a multithreaded shared-memory environment.

  We envision an interesting future work on the use of performance ratios for the simulation of different machine configurations using the burst mode,

and the cross-validation of the resulting predicted execution times against the real machine.

- **Sampling of sequential sections.** The size of traces depends on the type of the contents that, at the same time, depends on the timing model. The traces for the burst and inout modes are rather small. However, the mem and instr mode traces can be very large, up to hundreds of gigabytes. Also, the simulation time of such traces is usually very long. In this context, we could apply sampling techniques to only trace one or a few representative sequential sections, the simulation of which could provide an accurate estimation of the average behavior of the application.

  There are already existing sampling techniques, as we explained in Section 2.4. Some of them have even been explored in the context of multithreaded applications. However, none of them have been assessed for dynamically-scheduled multithreaded applications, such as the ones driving the work in this thesis.

  Using such sampling techniques for runtime-managed applications would provide shorter simulation times, and could also provide large reductions in trace size for mem and instr simulations. In this case, one or a few sequential sections would be simulated using detailed simulation, such as mem or instr, and the rest of the application could be fast forwarded in mem and instr mode using the result of the detailed simulation.

- **Deterministic replay of dynamic scheduling**. In the experiments in Chapter 5 we compare the simulation accuracy of three different traces: reset, naive and full. The differences in these traces lead to small timing differences that result in butterfly effects. Differences of a few cycles, may make a task to be scheduled to a different thread, thus changing the whole scheduling. Then, the access patterns to the cache hierarchy change and it is difficult to compare the different configurations when they end up exercising completely different actions in the underlying hardware.

  A potential solution to this problem is the development of two new schedulers: one that records the scheduling of tasks to threads in a file, and another one that replays that scheduling on subsequent simulations. This way, the *reference* simulation can be simulated using the *record* scheduler and the rest using the *replay* scheduler. This would end up having the same scheduling in all configurations, thus resulting in comparable experiments. However, this mechanism may suffer from the same effects as previous works in deterministic multithreaded application simulation. That is that a thread may have to wait to execute the next task in the *recorded* list of tasks because its predecessors have not completed yet. This introduces some idle time that would not be present in the real machine, where the scheduler would have assigned a ready task instead.

  Therefore, for this work, both the reduction of variability and the introduction of unrealistic idle time must be assessed.

- **Improved timing models for parops during simulation**. In Chapter 6 we presented our attempt to model the execution time of parops in the simulated machine using our simulation methodology. However, the

results are not as expected and our models do not reflect the variability seen in the real machine.

In most cases, the cost of parops is negligible in the overall application execution time and, hence, variations in parop execution time are negligible. However, for research studies focusing on parop cost or targeting fine-grained parallelism, a proper timing-model for parops is needed. For this reason we recommend that more research is targeted in this direction.

Glossary

Application abstraction Level of detail in the representation of an application as fed to a simulator.

CMP Chip Multiprocessor.

CPU Central Processing Unit.

DMA Direct Memory Access.

DSP Digital Signal Processor.

FLOPS Floating-point Operations Per Second.

FPGA Field Programmable Gate Arrays.

GPU Graphics Processing Unit.

Green500 Ranking of the 500 most energy efficient supercomputers.

Host machine Machine running the simulator.

HPC High Performance Computing.

HPL High-Performance Linpack.

I/O Input/Output.

ISA Instruction Set Architecture.

Linpack Benchmark used to rank supercomputers in the Top500 and Green500 lists.

Microprocessor Integrated circuit that incorporates the CPU of a von Neumann-style computing system.

Model abstraction Level of detail in the model of a simulated microprocessor component.

MSHR Miss Status Handling Register.

Parops Parallelism Management Operations.

Sequential section Code section in a parallel application the instructions of which execute in sequential order regardless of whether the application runs in parallel or sequentially.

SIMD Single Instruction Multiple Data.

SMT Simultaneaous Multithreading.

Target machine Machine modeled in the simulator.

Timing model Model that predicts the timing of an application running on the target machine.

Top500 Ranking of the 500 most performing supercomputers.

VLIW Very Long Instruction Word.

Bibliography

[1] FeS2: A Full-system Execution-driven Simulator for x86. http://fes2.cs.uiuc.edu. Accessed: June, 12th 2013.

[2] GCC OpenMP Manual. http://gcc.gnu.org/onlinedocs/libgomp. Accessed: July 1st, 2013.

[3] Marenostrum supercomputer ii (2006) system architecture. http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture. Accessed: June, 13th 2013.

[4] Mercurium project website. https://pm.bsc.es/projects/mcxx. Accessed: June 20th, 2013.

[5] Monte Carlo method. http://en.wikipedia.org/wiki/Monte_Carlo_method. Accessed: June, 10th 2013.

[6] NANOS++ Official Website. https://pm.bsc.es/projects/nanox. Accessed: June 20th, 2013.

[7] OpenMP 3.1 Specification. http://www.openmp.org/mp-documents/OpenMP3.1.pdf. Accessed: June, 13th 2013.

[8] Power.org — Collaborative Innovation for Power Architecture. http://www.power.org.

[9] Spec cpu2006 results. http://www.spec.org/cpu2006/results. Accessed: July 7th, 2013.

[10] SPEC Fair Use Rules. http://www.spec.org/fairuse.html. Accessed: 31 May 2013.

[11] Top500 Supercomputer Sites. http://www.top500.org.

[12] Synergistic Processor Unit Instruction Set Architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44, 27 Jan. 2007.

[13] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. The MPsim Simulation Tool. Technical Report UPC-DAC-RR-2009-7, Universitat Politècnica de Catalunya, Jan. 2009.

[14] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Rev 3.08. Technical report, 2012.

[15] A. Agarwal and M. Huffman. Blocking: exploiting spatial locality for trace compaction. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 48–57, 1990.

[16] M. Alvarez, F. Cabarcas, A. Ramírez, C. Meenderinck, B. Juurlink, and M. Valero. Scalability of Parallel Video Decoding on Heterogeneous Many-core Architectures. Technical Report UPC-DAC-RR-CAP-2011-12, Universitat Politècnica de Catalunya, June 2011.

[17] A. Ansari, S. Feng, S. Gupta, and S. Mahlke. Necromancer: Enhancing System Throughput by Animating Dead Cores. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 473–484, 2010.

[18] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for Full System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[19] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, 2009.

[20] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, July 2007.

[21] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[22] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, R. Balasubramonian, and V. Srinivasan. Efficient Scrub Mechanisms for Error-Prone Emerging Memories. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, 2012.

[23] A. Azevedo and B. Juurlink. An Instruction to Accelerate Software Caches. In *Proceedings of the 24th international conference on Architecture of computing systems*, ARCS'11, pages 158–170, 2011.

[24] R. M. Badia, J. Labarta, J. Gimenez, and F. Escal'e. DIMEMAS: Predicting MPI applications behavior in Grid environments. In *GGF '03: Workshop on Grid Applications and Programming Tools*, June 2003.

[25] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pages 163–174, 2009.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, 2003.

[27] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1:1–1:11, 2008.

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

[29] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, 2008.

[30] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 86, 2006.

[31] R. Bhargava, L. John, and F. Matus. Accurately Modeling Speculative Instruction Fetching in Trace-Driven Simulation. In *1999 IEEE International Performance, Computing and Communications Conference*, pages 65 –71, Feb. 1999.

[32] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[34] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[35] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. Can Trace-Driven Simulators Accurately Predict Superscalar Performance? In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, ICCD '96, pages 478–485, 1996.

[36] G. Black, N. Binkert, S. K. Reinhardt, and A. Saidi. *Modular ISA-Independent Full-System Simulation*, chapter 5. Springer, 2010.

[37] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software (TOMS)*, 28(2):135–151, 2002.

[38] S. Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, 2007.

[39] P. Bose. Integrated Modeling Challenges in Extreme-Scale Computing. Keynote at ISPASS'11: International Symposium on Performance Analysis of Systems and Software 2011, Apr. 2011.

[40] A. Bosque, P. Ibañez, V. Viñals, P. Stenström, and J. M. Llabería. Characterization of Apache Web Server with Specweb2005. In *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture*, MEDEA '07, pages 65–72, 2007.

[41] M. Briejer, C. Meenderinck, and B. Juurlink. Extending the Cell SPE with Energy Efficient Branch Prediction. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 304–315, 2010.

[42] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, P. Emma, and M. Rosenfield. New Methodology for Early-Stage, Microarchitecture-Level Power-Performance Analysis of Microprocessors. *IBM Journal of Research and Development*, 47(5.6):653–670, 2003.

[43] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[44] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[45] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy Evaluation of GEM5 Simulator System. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, ReCoSoC '12, pages 1–7, 2012.

[46] M. Butler, L. Barnes, D. Sarma, and B. Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *Micro, IEEE*, 31(2):6–15, 2011.

[47] F. Cabarcas, A. Rico, Y. Etsion, and A. Ramirez. Interleaving Granularity on High Bandwidth Memory Architecture for CMPs. In *International Conference on Embedded Computer Systems*, SAMOS '10, pages 250–257, 2010.

[48] F. Cabarcas, A. Rico, A. Ramirez, Y. Etsion, C. Villavieja, A. J. Vega, M. Pavlovic, A. Quesada, P. Bellens, R. M. Badia, and M. Valero. Castell: A CMP architecture scalable to hundreds of processors. Technical Report UPC-DAC-RR-CAP-2009-33, Universitat Politecnica de Catalunya, Nov. 2009.

[49] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé. A module-based cell processor simulator. In *Advanced Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems 2006 Poster Abstracts*, ACACES '06, pages 237–240, July 2006.

[50] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé. CellSim: A Cell Processor Simulation Infrastructure. In *Advanced Computer Architecture and Compilation for Embedded Systems 2007 Poster Abstracts*, ACACES '07, pages 279–282, July 2007.

[51] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé. CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator. In *XVIII Jornadas de Paralelismo (CEDI 2007: II Congreso Espanol de Informática)*, JP '07, pages 187–188, Sept. 2007.

[52] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, 2011.

[53] A. Carpenter, J. Hu, J. Xu, M. Huang, and H. Wu. A Case for Globally Shared-Medium On-Chip Interconnect. In *Proceedings of the 38th annual International Symposium on Computer Architecture*, ISCA '11, pages 271–282, 2011.

[54] P. Carpenter. *Running Stream-like Programs on Heterogeneous Multi-core Systems*. PhD thesis, Universitat Politècnica de Catalunya, 2011.

[55] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[56] J. Chaoui, K. Cyr, J.-P. Giacalone, S. de Gregorio, Y. Masse, Y. Muthusamy, T. Spits, M. Budagavi, and J. Webb. OMAP: Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals. *White Paper, Texas Instruments*, Dec. 2000.

[57] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, 2005.

[58] J. Chen, M. Annavaram, and M. Dubois. SlackSim: a Platform for Parallel Simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37:20–29, July 2009.

[59] J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. Adaptive and Speculative Slack Simulations of CMPs on CMPs. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 523–534, 2010.

[60] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 377–388, 2008.

[61] C. Ciobanu, G. Kuzmanov, G. Gaydadjiev, and A. Ramirez. A Polymorphic Register File for Matrix Operations. In *International Conference on Embedded Computer Systems*, SAMOS '10, pages 241–249, 2010.

[62] C. B. Ciobanu, X. Martorell, G. K. Kuzmanov, A. Ramirez, and G. N. Gaydadjiev. Scalability Evaluation of A Polymorphic Register File: A CG Case Study. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, ARCS'11, pages 13–25, 2011.

[63] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson. Single Node On-Line Simulation of MPI Applications with SMPI. In *2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS '11, pages 664–675, 2011.

[64] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, 2001.

[65] L. P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. `http://tools.ietf.org/html/rfc1951`, 1996.

[66] J. Donald and M. Martonosi. An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation. *IEEE Comput. Archit. Lett.*, 5(2):14, 2006.

[67] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[68] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[69] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 36:1–36:11, 2008.

[70] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 100–110, 2008.

[71] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Des. Test*, 18(5):21–31, 2001.

[72] L. Eeckhout, S. Nussbaum, J. Smith, and K. De Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *Micro, IEEE*, 23(5):26–38, 2003.

[73] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing Computer Architecture Research Workloads. *Computer*, 36(2):65–71, 2003.

[74] Enciclopedia Britannica. Microprocessor. `http://global.britannica.com/EBchecked/topic/380548/microprocessor`. Accessed: June 6th, 2013.

[75] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task Superscalar: An Out-of-Order Task Pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '10, pages 89–100, 2010.

[76] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr. 2005.

[77] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, page 83, 2006.

[78] E. Fernández Abeledo. Implementation of Nexus: Dynamic Hardware Management Support for Multicore Platforms. Master's thesis, Delft University of Technology, 2010.

[79] M. Floyd, M. Ware, K. Rajamani, T. Gloekler, B. Brock, P. Bose, A. Buyuktosunoglu, J. Rubio, B. Schubert, B. Spruth, J. Tierno, and L. Pesantez. Adaptive Energy-Management Features of the IBM POWER7 Chip. *IBM Journal of Research and Development*, 55(3):8:1–8:18, 2011.

[80] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, 1998.

[81] V. Garcia, A. Rico, C. Villavieja, N. Navarro, and A. Ramirez. The Data Transfer Engine: Towards a Software Controlled Memory Hierarchy. In *Advanced Computer Architecture and Compilation for Embedded Systems 2012 Poster Abstracts*, ACACES '12, pages 215–218, July 2012.

[82] J. Gee and A. Smith. Analysis of multiprocessor memory reference behavior. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, ICCD '94, pages 53–59, 1994.

[83] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *HPCA '10: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan. 2010.

[84] R. Giorgi, Z. Popovic, and N. Puzovic. Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture. In *IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2009, pages 1–8, 2009.

[85] R. Giorgi, Z. Popovic, and N. Puzovic. Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 78–87, 2009.

[86] R. Giorgi, Z. Popovic, and N. Puzovic. Introducing Hardware TLP Support in the Cell Processor. In *International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '09, pages 657–662, 2009.

[87] R. Giorgi, N. Puzovic, and Z. Popovic. Implementing DTA support in CellSim. In *Advanced Computer Architecture and Compilation for Embedded Systems 2008 Poster Abstracts*, ACACES '08, July 2008.

[88] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 146–157, 1993.

[89] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. Simulating Whole Supercomputer Applications. *IEEE Micro*, 31(3):32–45, 2011.

[90] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 179–188, 2010.

[91] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. SimFlex: a Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, 2004.

[92] S. A. Herrod. Tango Lite: A Multiprocessor Simulation Environment, Introduction and User's Guide. `http://www-flash.stanford.edu/herrod/docs/tango-lite.ps`, 1993.

[93] M. Hsieh, K. Pedretti, J. Meng, A. Coskun, M. Levenhagen, and A. Rodrigues. SST + gem5 = A Scalable Simulation Infrastructure for High Performance Computing. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, pages 196–201, 2012.

[94] X. Huang, X. Fan, S. Zhang, and L. Shi. Investigation on Multi-Grain Parallelism in Chip Multiprocessor for Multimedia Application. In *International Conference on Information Engineering and Computer Science*, ICIECS 2009, pages 1–4, 2009.

[95] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A, 3B, and 3C: System Programming Guide, Parts 1 and 2. Technical report, 2011.

[96] S. Isaza, F. Sanchez, F. Cabarcas, A. Ramirez, and G. Gaydadjiev. Parametrizing Multicore Architectures for Multiple Sequence Alignment. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 31:1–31:10, 2011.

[97] A. Jaleel, R. S. Cohn, C.-k. Luk, and B. Jacob. CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs. Technical Report UMDSCA-2006-01, Intel, 2006.

[98] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: a Pin-based On-the-fly Multi-core Cache Simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[99] D. R. Jefferson and H. A. Sowizral. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Rand Note N-1906AF, the Rand Corp., Santa Monica, CA, Dec. 1982.

[100] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, July 2005.

[101] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[102] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *Proceedings of the 31st annual International Symposium on Computer Architecture*, ISCA '04, pages 338–, 2004.

[103] C. D. Kersey, A. Rodrigues, and S. Yalamanchili. A Universal Parallel Front-end for Execution Driven Microarchitecture Simulation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 25–32, 2012.

[104] A. KleinOsowski and D. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1(1):7–7, 2002.

[105] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the Validity of Trace-driven Simulation for Multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 244–253, 1991.

[106] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System in FPGAs. In *International Conference on Field Programmable Logic and Applications, 2007*, FLP '07, pages 54–61, 2007.

[107] J. R. Larus. Abstract execution: a technique for efficiently tracing programs. *Softw. Pract. Exper.*, 20(12):1241–1258, Nov. 1990.

[108] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, and S. Cho. Two-phase trace-driven simulation (TPTS): a fast multicore processor architecture simulation approach. *Softw. Pract. Exper.*, 40:239–258, Mar. 2010.

[109] K. Lee, S. Evans, and S. Cho. Accurately approximating superscalar processor performance from traces. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2009*, ISPASS'09, pages 238–248, Apr. 2009.

[110] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, 2009.

[111] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2009*, ISPASS 2009, pages 53–64, 2009.

[112] D. Ludovici and G. Gaydadjiev. SARC Power Estimation Methodology. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing*, Nov. 2007.

[113] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[114] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[115] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *30th Annual International Symposium on Computer Architecture*, ISCA '30, pages 182–193, 2003.

[116] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[117] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel Scalability of Video Decoders. *J. Signal Process. Syst.*, 57(2):173–194, Nov.

[118] C. Meenderinck and B. Juurlink. A Chip MultiProcessor Accelerator for Video Decoding. In *Proceedings 19th Annual Workshop on Circuits, Systems and Signal Processing*, Nov. 2008.

[119] C. Meenderinck and B. Juurlink. Nexus: Hardware Support for Task-Based Programming. In *14th Euromicro Conference on Digital System Design*, DSD '11, pages 442–445, 2011.

[120] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[121] J. E. Miller, H. Kasture, G. Kurian, N. Beckmann, C. G. III, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. Technical Report MIT-CSAIL-TR-2009-056, Massachusetts Institute of Technology, Nov. 2009.

[122] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, July 2009.

[123] G. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[124] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 19 Apr. 1965.

[125] M. Moudgill. Techniques for Implementing Fast Processor Simulators. In *Proceedings of the The 31st Annual Simulation Symposium*, SS '98, pages 83–, 1998.

[126] M. Moudgill, P. Bose, and J. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. In *IPCCC'99: IEEE International Performance, Computing and Communications Conference*, pages 451–457, Feb. 1999.

[127] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, May 1999.

[128] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8:12–20, Oct. 2000.

[129] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Understanding the effects of wrong-path memory references on processor performance. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 56–64, 2004.

[130] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, 2007.

[131] S. Nussbaum and J. Smith. Modeling Superscalar Processors Via Statistical Simulation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2001*, PACT '01, pages 15–24, 2001.

[132] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, 1996.

[133] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1050–1055, 2011.

[134] M. Pavlovic, Y. Etsion, and A. Ramirez. Can Manycores Support the Memory Requirements of Scientific Applications? In *A4MMC'10: 1st Workshop on Applications for Multi and Many Core Processors*, 2010.

[135] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 318–319, 2003.

[136] J. M. Perez, R. M. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, Sept. 2008.

[137] J. Preshing. A Look Back at Single-Threaded CPU Performance. `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance`, 8 Feb. 2012. Accessed: 31 February 2013.

[138] T. R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, 1985. AAI8509594.

[139] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the Case for an ARM-Based HPC System. *Future Generation Computing Systems*, 2013. Accepted for publication.

[140] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez. Experiences with Mobile Processors for Energy Efficient HPC. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 464–468, 2013.

[141] A. Ramirez, F. Cabarcas, B. Juurlink, A. Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *Micro, IEEE*, 30(5):16–29, 2010.

[142] A. Ramirez, A. Rico, and F. Cabarcas. CellSim: a Modular Simulator for Heterogeneous Chip Multiprocessors. `http://pcsostres.ac.upc.edu/cellsim/doku.php#cellsim_tutorial_documentation`, 15 Sept. 2007. Full-day tutorial at Parallel Architectures and Compilation Techniques (PACT) 2007.

[143] L. Rauchwerger, P. K. Dubey, and R. Nair. Measuring Limits of Parallelism and Characterizing its Vulnerability to Resource Constraints. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, MICRO 26, pages 105–117, 1993.

[144] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[145] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. `http://sesc.sourceforge.net`, Jan. 2005.

[146] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable Simulation of Decoupled Accelerator Architectures. Technical Report UPC-DAC-RR-2010-14, Universitat Politècnica de Catalunya, June 2010.

[147] A. Rico, F. Cabarcas, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé. Implementation and validation of a Cell simulator (Cellsim) using UNISIM. In *3rd HiPEAC Industrial Workshop on Compilers and Architectures*, Apr. 2007.

[148] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, 2012.

[149] A. Rico, J. H. Derby, R. K. Montoye, T. H. Heil, C.-Y. Cher, and P. Bose. Performance and Power Evaluation of an In-line Accelerator. In *Proceedings of the 2010 ACM International Conference on Computing Frontiers*, CF'10, pages 81–82, May 2010.

[150] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven Simulation of Multithreaded Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 87–96, Apr. 2011.

[151] A. Rico, A. Duran, A. Ramirez, and M. Valero. Simulating Dynamically-Scheduled Multithreaded Applications Using Traces. *IEEE Micro*. Submitted for publication.

[152] A. Rico, A. Ramirez, and M. Valero. Task Management Analysis on the Cell BE. In *XIX Jornadas de Paralelismo*, JP '08, pages 271–276, Sept. 2008.

[153] A. Rico, A. Ramirez, and M. Valero. Available Task-level Parallelism on the Cell BE. *Scientific Programming*, 17(1-2):59–76, 2009.

[154] A. Rico, A. Ramirez, and M. Valero. Trace Filtering of Multithreaded Applications for CMP Memory Simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, pages 134–135, Apr. 2013.

[155] F. Ryckbosch, S. Polfliet, and L. Eeckhout. VSim: Simulating Multi-Server Setups at Near Native Hardware Speed. *ACM Trans. Archit. Code Optim.*, 8(4):52:1–52:20, 2012.

[156] A. D. Samples. Mache: no-loss trace compaction. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '89, pages 89–97, 1989.

[157] F. Sanchez, F. Cabarcas, A. Ramirez, and M. Valero. Scalable Multicore Architectures for Long DNA Sequence Comparison. *Concurr. Comput. : Pract. Exper.*, 23(17):2205–2219, Dec. 2011.

[158] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu. Performance Projection of HPC Applications Using SPEC CFP2006 Benchmarks. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, 2009.

[159] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, 2002.

[160] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[161] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, 1994.

[162] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic. RAMP gold: An FPGA-based Architecture Simulator for Multiprocessors. In *47th ACM/IEEE Design Automation Conference (DAC), 2010*, DAC '10, pages 463–468, 2010.

[163] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. A High-Productivity Task-Based Programming Model for Clusters. *Concurr. Comput. : Pract. Exper.*, 24(18):2421–2448, Dec. 2012.

[164] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 135–148, 2009.

[165] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 392–403, 1995.

[166] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *19th International Symposium on Computer Architecture and High Performance Computing, 2007*, SBAC-PAD '07, pages 62–68, 2007.

[167] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29:128–170, June 1997.

[168] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS '94, pages 201–207, 1994.

[169] A. Vega, F. Cabarcas, A. Ramirez, and M. Valero. Breaking the Bandwidth Wall in Chip Multiprocessors. In *International Conference on Embedded Computer Systems*, SAMOS '11, pages 255–262, 2011.

[170] A. Vega, A. Rico, F. Cabarcas, A. Ramírez, and M. Valero. Comparing Last-level Cache Designs for CMP Architectures. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, pages 2:1–2:11, 2010.

[171] N. Vujic, F. Cabarcas, M. Gonzalez Tallada, A. Ramirez, X. Martorell, and E. Ayguade. DMA++: On the Fly Data Realignment for On-Chip Memories. *IEEE Transactions on Computers*, 61(2):237–250, 2012.

[172] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, 1991.

[173] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation method for cache performance analysis. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 27–36, 1990.

[174] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, 2007.

[175] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 408–409, 2005.

[176] Wikipedia. Microprocessor. `http://en.wikipedia.org/wiki/Microprocessor`, 2013. Accessed: June 6th, 2013.

[177] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation Via Rigorous Statistical Sampling. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, ISCA '03, pages 84–97, 2003.

[178] G. Yan, X. Liang, Y. Han, and X. Li. Leveraging the Core-Level Complementary Effects of PVT Variations to Reduce Timing Emergencies in Multi-Core Processors. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 485–496, 2010.

[179] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *11th International Symposium on High-Performance Computer Architecture, 2005*, HPCA '11, pages 266–277, 2005.

[180] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The Future of Simulation: A Field of Dreams. *Computer*, 39(11):22–29, Nov. 2006.

[181] M. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE International Symposium on Performance Analysis of Systems Software 2007*, ISPASS '07, pages 23–34, 2007.

[182] R. Zhong, Y. Zhu, W. Chen, M. Lin, and W.-F. Wong. An Inter-Core Communication Enabled Multi-Core Simulator Based on SimpleScalar. In *21st International Conference on Advanced Information Networking and Applications Workshops, 2007*, volume 1 of *AINAW '07*, pages 758–763, 2007.

[183] P. Zhu, M. Chen, Y. Bao, L. Chen, and Y. Huang. Trace-driven Simulation of Memory System Scheduling in Multithread Application. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 30–37, 2012.