UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

# *Simulation methodologies for future large-scale parallel systems*

**Thomas Grass**

UNIVERSITAT POLITÈCNICA DE CATALUNYA

DOCTORAL THESIS

---

# Simulation Methodologies for Future Large-Scale Parallel Systems

---

*Author:*
Thomas GRASS

*Supervisors:*
Dr. Marc CASAS GUIX
Dr. Miquel MORETÓ PLANAS

*A thesis submitted in fulfillment of the requirements*
*for the degree of Doctor of Philosophy*

*in the*

Facultat d'Informàtica de Barcelona
Departament d'Arquitectura de Computadors

July 17, 2017

# Declaration of Authorship

I, Thomas GRASS, declare that this thesis titled "Simulation Methodologies for Future Large-Scale Parallel Systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

---

Date:

---

# Abstract

Since the early 2000s, computer systems have seen a transition from single-core to multi-core systems. While single-core systems included only one processor core on a chip, current multi-core processors include up to tens of cores on a single chip, a trend which is likely to continue in the future. Today, multi-core processors are ubiquitous. They are used in all classes of computing systems, ranging from low-cost mobile phones to high-end High-Performance Computing (HPC) systems. Designing future multi-core systems is a major challenge [12]. The primary design tool used by computer architects in academia and industry is architectural simulation. Simulating a computer system executing a program is typically several orders of magnitude slower than running the program on a real system. Therefore, new techniques are needed to speed up simulation and allow the exploration of large design spaces in a reasonable amount of time.

One way of increasing simulation speed is sampling. Sampling reduces simulation time by simulating only a representative subset of a program in detail. In this thesis, we present a workload analysis of a set of task-based programs. We then use the insights from this study to propose TaskPoint, a sampled simulation methodology for task-based programs. Task-based programming models can reduce the synchronization costs of parallel programs on multi-core systems and are becoming increasingly important. Finally, we present MUSA, a simulation methodology for simulating applications running on thousands of cores on a hybrid, distributed shared-memory system. The simulation time required for simulation with MUSA is comparable to the time needed for native execution of the simulated program on a production HPC system.

The techniques developed in the scope of this thesis permit researchers and engineers working in computer architecture to simulate large workloads, which were infeasible to simulate in the past. Our work enables architectural research in the fields of future large-scale shared-memory and hybrid, distributed shared-memory systems.

# Resum

Des dels principis dels anys 2000, els sistemes d'ordinadors han experimentat una transició de sistemes d'un sol nucli a sistemes de múltiples nuclis. Mentre els sistemes d'un sol nucli incloïen només un nucli en un xip, els sistemes actuals de múltiples nuclis n'inclouen desenes, una tendència que probablement continuarà en el futur. Avui en dia, els processadors de múltiples nuclis són omnipresents. Es fan servir en totes les classes de sistemes de computació, de telèfons mòbils de baix cost fins a sistemes de computació d'alt rendiment. Dissenyar els futurs sistemes de múltiples nuclis és un repte important [12]. L'eina principal usada pels arquitectes de computadors, tant a l'acadèmia com a la indústria, és la simulació. Simular un ordinador executant un programa típicament és múltiples ordres de magnitut més lent que executar el mateix programa en un sistema real. Per tant, es necessiten noves tècniques per accelerar la simulació i permetre l'exploració de grans espais de disseny en un temps raonable.

Una manera d'accelerar la velocitat de simulació és la *simulació mostrejada*. La simulació mostrejada redueix el temps de simulació simulant en detall només un subconjunt representatiu d'un programa. En aquesta tesi es presenta una anàlisi de rendiment d'una col·lecció de programes basats en tasques. Com a resultat d'aquesta anàlisi, proposem TaskPoint, una metodologia de simulació mostrejada per programes basats en tasques. Els models de programació basats en tasques poden reduir els costos de sincronització de programes paral·lels executats en sistemes de múltiples nuclis i actualment estan guanyant importància. Finalment, presentem MUSA, una metodologia de simulació per simular aplicacions executant-se en milers de nuclis d'un sistema híbrid, que consisteix en nodes de memòria compartida que formen un sistema de memòria distribuïda. El temps que requereixen les simulacions amb MUSA és comparable amb el temps que triga l'execució nativa en un sistema d'alt rendiment en producció.

Les tècniques desenvolupades al llarg d'aquesta tesi permeten simular execucions de programes que abans no eren viables, tant als investigadors com als enginyers que treballen en l'arquitectura de computadors. Per tant, aquest treball habilita futura recerca en el camp d'arquitectura de sistemes de memòria compartida o distribuïda, o bé de sistemes híbrids, a gran escala.

# Resumen

A principios de los años 2000, los sistemas de ordenadores experimentaron una transición de sistemas con un núcleo a sistemas con múltiples núcleos. Mientras los sistemas *single-core* incluían un sólo núcleo, los sistemas *multi-core* incluyen decenas de núcleos en el mismo chip, una tendencia que probablemente continuará en el futuro. Hoy en día, los procesadores *multi-core* son omnipresentes. Se utilizan en todas las clases de sistemas de computación, de teléfonos móviles de bajo coste hasta sistemas de alto rendimiento. Diseñar sistemas *multi-core* del futuro es un reto importante. La herramienta principal usada por arquitectos de computadores, tanto en la academia como en la industria, es la simulación. Simular un computador ejecutando un programa típicamente es múltiples ordenes de magnitud más lento que ejecutar el mismo programa en un sistema real. Por ese motivo se necesitan nuevas técnicas para acelerar la simulación y permitir la exploración de grandes espacios de diseño dentro de un tiempo razonable.

Una manera de aumentar la velocidad de simulación es la *simulación muestreada*. La simulación muestreada reduce el tiempo de simulación simulando en detalle sólo un subconjunto representativo de la ejecución entera de un programa. En esta tesis presentamos un análisis de rendimiento de una colección de programas basados en tareas. Como resultado de este análisis presentamos TaskPoint, una metodología de simulación muestreada para programas basados en tareas. Los modelos de programación basados en tareas pueden reducir los costes de sincronización de programas paralelos ejecutados en sistemas *multi-core* y actualmente están ganando importancia. Finalmente, presentamos MUSA, una metodología para simular aplicaciones ejecutadas en miles de núcleos de un sistema híbrido, compuesto de nodos de memoria compartida que forman un sistema de memoria distribuida. El tiempo de simulación que requieren las simulaciones con MUSA es comparable con el tiempo necesario para la ejecución del programa simulado en un sistema de alto rendimiento en producción.

Las técnicas desarolladas al largo de esta tesis permiten a los investigadores e ingenieros trabajando en la arquitectura de computadores simular ejecuciones largas, que antes no se podían simular. Nuestro trabajo facilita nuevos caminos de investigación en los campos de sistemas de memoria compartida o distribuida y en sistemas híbridos.

# Acknowledgements

First and foremost, I would like to thank my thesis directors, Marc Casas and Miquel Moretó, for their continuous support and their guidance during these last years. After my initial group ceased to exist, they offered me to work with them on the RoMoL project, an opportunity for which I am very grateful. Without Miquel and Marc this thesis would not have been possible.

Between 2013 and 2016, I was partially supported by the AGAUR of the Generalitat de Catalunya (grant 2013FI B 0058). Furthermore, this work was supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project received funding from the EUs Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610402 and from the EUs H2020 Framework Programme (H2020/2014-2020) under grant agreement no. 671697.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **BBV** | **B**asic **B**lock **V**ector |
| **ccNUMA** | **c**ache-**c**oherent **N**on-**U**niform **M**emory **A**ccess |
| **CMP** | **C**hip **M**ulti-**P**rocessor |
| **CPU** | **C**entral **P**rocessing **U**nit |
| **DRAM** | **D**ynamic **R**andom **A**ccess **M**emory |
| **FPGA** | **F**ield-**P**rogrammable **G**ate **A**rray |
| **GPGPU** | **G**eneral **P**urpose **G**raphics **P**rocessing **U**nit |
| **ILP** | **I**nstruction-**L**evel **P**arallelism |
| **IPC** | **I**nstructions **P**er **C**ycle |
| **ISA** | **I**nstruction **S**et **A**rchitecture |
| **HPC** | **H**igh-**P**erformance **C**omputing |
| **LLC** | **L**ast-**L**evel **C**ache |
| **LRU** | **L**east **R**ecently **U**sed |
| **MLP** | **M**emory **L**evel **P**arallelism |
| **NoC** | **N**etwork-**o**n-**C**hip |
| **NUCA** | **N**on-**U**niform **C**ache **A**rchitecture |
| **PMU** | **P**erformance **M**onitoring **U**nit |
| **ROB** | **R**e**O**rder-**B**uffer |
| **RTL** | **R**egister-**T**ransfer **L**evel |
| **SIMD** | **S**ingle **I**nstruction, **M**ultiple **D**ata |
| **SMT** | **S**imultaneous **M**ulti-**T**hreading |
| **SoC** | **S**ystem-**O**n-a-**C**hip |
| **SPMD** | **S**ingle **P**rogram, **M**ultiple **D**ata |
| **TLP** | **T**hread-**L**evel **P**aralellism |
| **UMA** | **U**niform **M**emory **A**ccess |

# Chapter 1

# Introduction

The first commercial microprocessor, the Intel 4004, was introduced in 1971 and consisted of approximately 2,300 transistors integrated on a single chip [7]. It required a variety of other integrated circuits to function. The Intel 4004 operated at a clock frequency of 740kHz and was able to execute one instruction every eight clock cycles or a total of up to 92,600 instructions per second. Ever since the introduction of the Intel 4004, performance and complexity of computer systems have been increasing exponentially over time.

In comparison, the Intel Xeon E5-2699 v4 processor [64], released early in 2016, integrates approximately 7.2 billion transistors on a single chip. The chip contains 22 active cores[1], 55MB of last-level cache and a variety of circuitry to interface with the rest of the system. The cores operate at 2.2GHz and can process up to 4 instructions per clock cycle, resulting in a theoretical maximum of 193.6 billion instructions per second for the full processor. Thus, since the arrival of the Intel 4004, processor performance has improved by a factor of more than 2 million.

The massive increase in processor performance over time has been possible because the manufacturing processes for integrated circuits have been continuously improving, combined with architectural enhancements. Over time, these improvements allowed to integrate an ever larger number of transistors on a single chip. Gordon Moore observed in 1965 that the number of transistors had been doubling every two years and he projected this trend into the future [84]. His observation became known as *Moore's Law*.

During the first three decades of the microprocessor's history, computer architects used the increasing numbers of transistors coming along with Moore's Law primarily to improve single-thread performance. As the feature sizes of integrated circuits shrank, it was possible to increase the frequency of operation, and thus the instruction throughput. Besides, the newly available transistors allowed to employ more sophisticated techniques to exploit *instruction-level parallelism* (ILP), allowing a single processor to execute multiple instructions per cycle, potentially out of program order. These efforts culminated in the *Prescott* micro-architecture, used by the

[1] The chip contains 24 cores, two of which are deactivated.

Intel Pentium 4 processor. The Prescott micro-architecture used 31 pipeline stages to achieve a high clock frequency. Pentium 4 Prescott processors were able to run at up to 3.8GHz, consuming more than 100W in power. It became apparent that further increasing the operation frequency would lead to unacceptable thermal power dissipation. At the same time, deeper pipelines would exacerbate the already high penalties of pipeline flushes, e.g. in the case of branch misspeculation.

In the early 2000's, the major processor manufacturers started to use the increased transistor counts to implement *chip multi-processors* (CMPs), i.e. processors integrating several processor cores on a single chip. These processor cores ran at a lower operation voltage and clock frequency than their single-core predecessors. CMPs achieved higher total performance while consuming less power. This performance increase has been achieved by exploiting *thread-level parallelism* (TLP). Instead of relying on ever more sophisticated techniques to detect and exploit ILP, CMPs execute multiple execution *threads* simultaneously.

Exploiting TLP on CMPs typically requires support in programming languages and runtime environments. Programmers need to take care of efficiently exposing TLP in a program. Despite these inconveniences, CMPs are prevalent today. Intel's current high-end Xeon E5-2699 v4 processor has 22 cores, whereas Intel's Xeon Phi 7120X systems even include 61 cores. An end to the continuously increasing core counts in this multi-core era is currently not in sight.

Architectural simulation is a key tool for computer architects and application developers. By relying on simulation, computer architects can evaluate the performance and power consumption of a benchmark on a variety of design choices without actually building costly hardware prototypes. Application developers use simulation to develop and optimize system software and applications so that the software is ready once a new machine hits the market.

High-performance computing (HPC) systems typically consist of a large number of shared memory nodes, each composed of multiple processors or sockets. In recent years, the number of cores per socket is continuously increasing. The TOP 500 list [113] lists the 500 fastest HPC systems in the world and is updated twice a year. Figure 1.1 shows the percentage of systems listed in the TOP 500 list for selected numbers of cores per socket. The figure clearly demonstrates that, since the advent of the dual-core processor, which reached its peak early in 2007, single-core processors have practically vanished. Instead, systems are built with ever increasing numbers of cores, and currently, there is no reason to assume that this trend will change in the near future.

The impact that the increasing core count has on architectural simulation is twofold. First, a larger amount of simulated, state-holding hardware requires the simulation of larger workloads to stress the simulated design meaningfully. Second, a

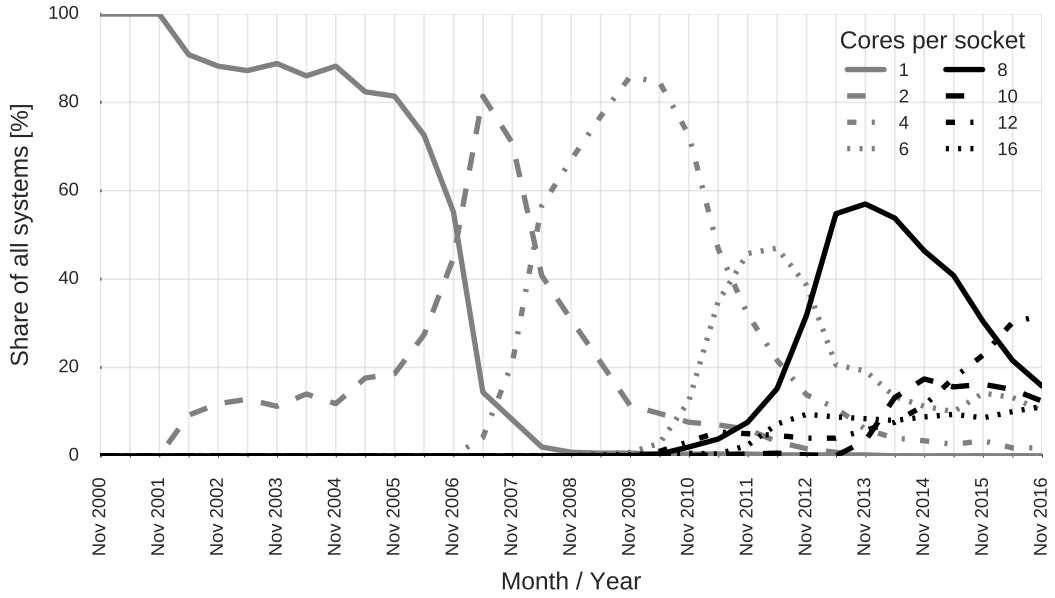



FIGURE 1.1: Evolution of the number of cores per socket since November 2000, as observed in the systems listed in the TOP 500 list of the fastest HPC systems.

simulation of a system with multiple cores requires simulating the interactions of these cores in shared system resources, e.g. last-level caches or the on-chip interconnect subsystem. Both the larger design complexity and the more complex system behavior increase simulation complexity and thus simulation time. However, the simulation speed of contemporary detailed architectural simulators has not increased to the same extent.

The size of HPC systems is also increasing in terms of the number of nodes. Consequently, system-level simulations need to simulate not only a larger number of total cores, but also increasingly large interconnection networks. Existing simulators for such machines either use high-level models or are prohibitively slow. While high-level models achieve high simulation speed, they do so by sacrificing simulation detail. Detailed simulation of programs executing on thousands of cores in a distributed memory system is very accurate but infeasible due to its excessive simulation time.

## 1.1 Thesis Contributions

In the following, we list the contributions we make in the different chapters of this thesis:

### 1.1.1 Execution Time Predictability of Task-Based Programs

- In Chapter 4, we analyze performance variability across instances of the same task type in a set of task-based programs executing on multi-core systems. This analysis shows the variability on an instance-by-instance basis.
- We identify different sources of execution time variability on instances of the same task type, namely input dependence, multiple behaviors per task type, and contention on shared system resources.
- We present a low-complexity model based on linear interpolation for predicting the execution time of a task instance as a function of its instruction count.
- We use a clustering algorithm to identify different classes of behavior in the same task type. In our example, we successfully classify task instances into clusters, each of which exhibits regular performance.

The content of Chapter 4 has been published under the title *"Evaluating Execution Time Predictability of Task-Based Programs on Multi-Core Processors"* at the MuCoCoS workshop, which was held in conjunction with Euro-Par 2014 in Porto, Portugal.

### 1.1.2 Sampled Simulation of Task-Based Programs

- In Chapter 5 we use the insights from Chapter 4 and present TaskPoint, a sampled simulation technique for multi-core architectures programmed with a dynamically scheduled, task-based programming model. We propose a mechanism to accurately fast-forward an architectural simulation of a task-based program. During fast-forward, we model the performance of a given task instance based on previous instances of the same task type. We account for different task input sizes across the application execution by factoring in the number of instructions of the given task instance accordingly.
- For applications with varying behavior across instances of the same task type, we employ basic block vectors (BBVs) and clustering to identify classes of similar behavior. We show how we (i) identify multiple classes of behavior among task instances of the same task type, and (ii) merge task instances with similar behavior belonging to different types. We use an analytical performance model to improve simulation accuracy during simulation in fast-forward mode. Our approach combines the speed of analytical models with the accuracy of detailed simulation.
- We evaluate TaskPoint simulating 27 task-based parallel benchmarks, including the PARSEC benchmark suite. We evaluate the sensitivity of TaskPoint

to different architectures by testing different numbers of simulated threads on two configurations covering the opposite extremes of the multi-core design space: high-performance and low power.

The content of Chapter 5 has been published under the title *"TaskPoint: Sampled Simulation of Task-Based Programs"* at the 2016 International Symposium on Performance Analysis of Systems and Software (ISPASS 2016) which was held in Uppsala, Sweden. An extended version is currently under submission at IEEE Transactions on Computers (TC).

### 1.1.3 Multi-Level Simulation of Hybrid Programs

- In Chapter 6 we present MUSA, a Multi-Scale Simulation Approach that enables fast and accurate performance estimations of next-generation HPC machines. Our methodology seamlessly captures inter-node communication as well as intra-node microarchitectural and system software interactions, improving usability and simplifying the simulation workflow. MUSA relies on native execution traces with two levels of detail to allow simulation of different communication networks, numbers of cores per node, and relevant microarchitectural parameters. MUSA optionally employs TaskPoint, our sampled simulation methodology for task-based programs presented in Chapter 5.

- We validate MUSA using the NAS Multi-Zone Parallel Benchmark suite [116], and then evaluate three large-scale case studies (with up to 16,384 cores) using BT-MZ, HYDRO [75], and SPECFEM3D [72]. Our evaluation shows that MUSA provides accurate performance predictions by combining information at different levels of granularity. When comparing native executions and MUSA simulations with up to 2,048 cores, we achieve relative errors within 10% in the common case, demonstrating that our detailed model is able to capture microarchitectural and system software effects. Besides, we show that our simulations complete in an affordable amount of time, i.e. less than a day of total aggregated CPU time for detailed 16,384-core simulations. This allows to quickly identify scalability problems in the targeted case studies.

- Finally, we perform a design space exploration analysis using high-performance, low-power, and die-stacked DRAM processor profiles on a system with 16,384 cores. We find that for one of the evaluated HPC applications, HYDRO, the low-power processor can achieve on par performance even with the same number of cores, because the high-performance memory hierarchy and aggressive microarchitecture are over-dimensioned. In contrast, the other two applications benefit from an aggressive out-of-order microarchitecture, and

SPECFEM3D achieves better scalability by exploiting the higher memory bandwidth provided by die-stacked DRAM technology.

The content of Chapter 6 has been published under the title *"MUSA: A Multi-Level Simulation Approach for Next-Generation HPC Machines"* at the International Conference for High Performance Computing, Networking, Storage and Analysis 2016 (SC16), which was held in Salt Lake City, Utah, United States of America.

MUSA permits performing design space exploration of future HPC systems. To this end, it has been used in the projects Mont-Blanc 2 and Mont-Blanc 3 to determine architectural features required to build an *exascale* system. An exascale system is an HPC machine with a peak performance of least one exaFLOPS, i.e. one quintillion ($10^{18}$) floating-point operations per second.

## 1.2 Thesis Outline

The remainder of this thesis is organized as follows: in Chapter 2, we provide background and present the state-of-the-art of simulation of shared- and distributed-memory systems. Afterwards, in Chapter 3 we present our experimental setup. Then, in Chapter 4, we present an analysis of execution time predictability of task-based programs. Next, in Chapter 5, we leverage the results of this analysis and propose TaskPoint, a sampled simulation technique for task-based programs. Then, in Chapter 6, we extend a high-level simulator for distributed memory systems by a detailed simulator which includes TaskPoint. The result is MUSA, a multi-level simulation approach for hybrid applications. We conclude in Chapter 7 and outline future work in Chapter 8.

# Chapter 2

# Background

In this chapter, we provide background and present the state-of-the-art related to the research conducted in the course of this thesis. First, in Section 2.1, we introduce parallel computer systems. Then, in Section 2.2, we introduce the prevalent parallel programming models for shared- and distributed-memory machines, as they are frequently used in HPC systems. Starting with traditional programming models, we move on to more recent ones targeting modern multi-core machines and allowing for increased programmer productivity. Afterwards, we review message passing, the prevalent programming model for distributed-memory systems. Finally, we introduce hybrid programming models, which are a combination of shared- and distributed-memory models.

Next, in Section 2.3, we introduce the concept of architectural simulation in the context of computer architecture research. We discuss different techniques for simulation of shared- and distributed-memory systems and point out the lack of techniques targeting hybrid programming models.

Finally, in Section 2.4, we present techniques for accelerating architectural simulation. After reviewing techniques for simulations of single-threaded systems, we discuss the issues which arise when moving to the simulation of multi-threaded systems and how they are addressed by different recent techniques. We point out why existing accelerating techniques are not directly applicable to dynamically scheduled, task-based programming models and motivate the work performed during the course of this thesis.

## 2.1 Parallel Systems

In this section, we introduce the prevalent system architectures used in parallel computing. First, we introduce shared-memory systems, in which all cores of a system can communicate by accessing the same memory address space. Afterwards, we present distributed-memory systems, in which processors have disjoint memory address spaces and communicate through message passing. We then introduce heterogeneous systems. Finally, we show, how current HPC systems combine the shared-

and distributed-memory approaches.

### 2.1.1 Shared-Memory Systems

In a shared-memory system, as illustrated in Figure 2.1, memory can be accessed by multiple processors using the same physical address space. Current high-end multi-core processors typically feature multiple cores or central processing units (CPUs), each of which with private L1 data- and instruction caches, and a unified L2 cache. A cache-coherent on-chip network connects the private L2 caches to an L3 cache, which is shared among all cores. The L3 cache interfaces to the DRAM subsystem.

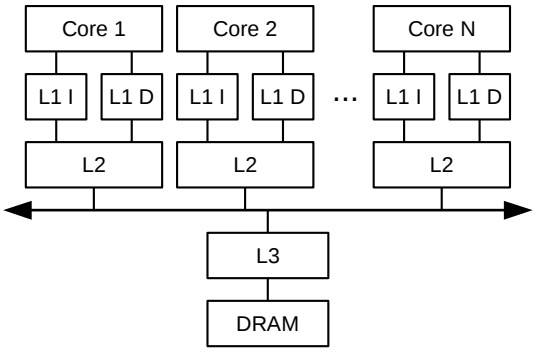



FIGURE 2.1: Shared-memory UMA system with two levels of private cache memories, shared L3 cache and DRAM subsystem.

Because all cores can access arbitrary memory locations with the same average bandwidth and latency, the system is also referred to as a *Uniform Memory Access* (UMA) system. However, modern multi-core systems include routed on-chip interconnect networks and banked last-level caches. As a result, the average memory bandwidth and latency measured on a particular core varies depending on the physical location of the accessed memory. Therefore, current multi-core systems do no longer fall into the class of UMA systems. Instead, they belong to the class of *Non-Uniform Cache Architecture* (NUCA) systems.

High-performance shared-memory systems can consist of multiple sockets, as illustrated in Figure 2.2. Each socket itself is a shared-memory system. The different sockets are connected via a cache-coherent interconnect, ensuring cache coherence across the entire system. In such a system, bandwidth and latency of memory accesses depend on whether the data resides in the same or a different socket. Systems of this type are referred to as *cache-coherent Non-Uniform Memory Access* (ccNUMA) systems.

### 2.1.2 Distributed-Memory Systems

In a shared-memory system, all processors share a common logical memory address space. In a *distributed-memory system*, as illustrated in Figure 2.3, each processor has

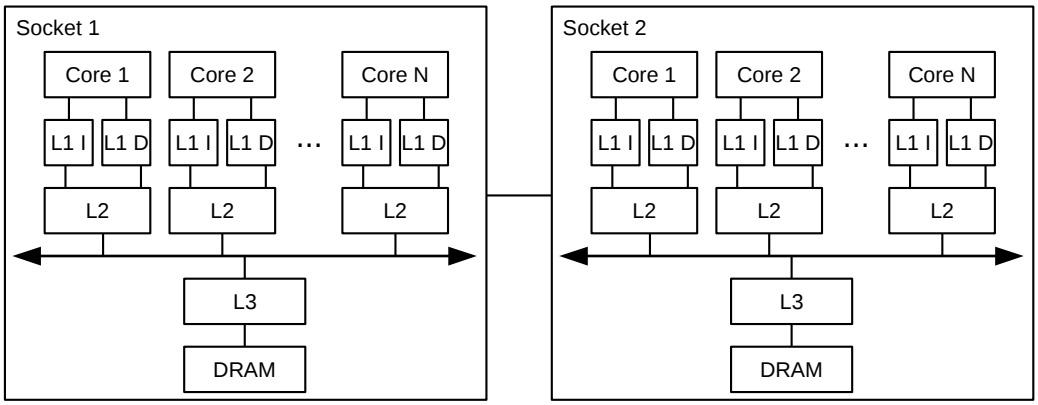



FIGURE 2.2: Shared-memory ccNUMA System with two sockets of N cores each. The sockets are connected via a cache-coherent interconnect.

its own memory address space. The memories of other processors are not directly reachable and different processors communicate via an interconnection network.

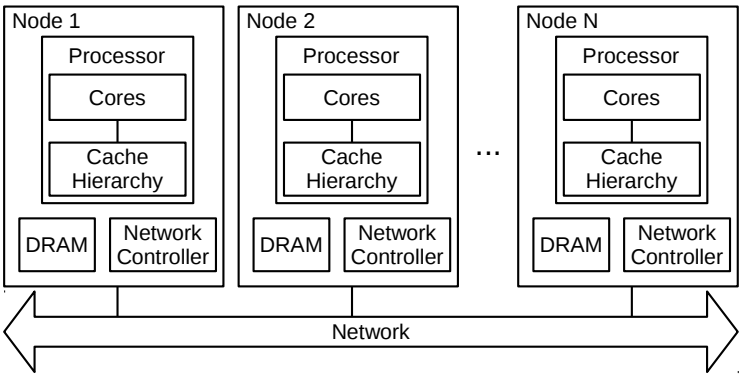



FIGURE 2.3: Distributed-memory system; processors communicate via message-passing over a network.

According to the TOP500 list of November 2016 [113], the two most widely used interconnect network families, namely InfiniBand (37.4%) and 10 Gigabit Ethernet (35.6%), together account for more than two-thirds of the systems on the current TOP500 list. For a distributed system to deliver high performance across a wide range of applications, it is critical that the interconnect system provides high bandwidth and low message latency. Due to the large number of nodes in current HPC systems, interconnect networks are typically organized in a hierarchical fashion.

The currently fastest system in the world, the *Sunway TaihuLight*, consists of nodes featuring 260 processor cores each. The different cores, which are all integrated on the same chip, communicate via a network-on-chip (NoC). 256 nodes form a supernode. The entire system consists of 160 supernodes. Nodes and supernodes are interconnected in a tree topology.

The *K computer*, still one of the ten fastest HPC systems in the world, uses a hybrid mesh-torus interconnect network. Nodes are organized into node groups of 12

nodes. Within a node group, nodes are interconnected using a three-dimensional mesh topology. Different node groups are interconnected in a three-dimensional torus topology. An advantage of this interconnect architecture is that there are multiple paths between any pair of nodes. As a result, the system can tolerate a small number of link failures and still be operational.

The examples mentioned above illustrate that the interconnect network has a significant influence on how applications are efficiently mapped to the nodes of a system. Both mentioned systems provide higher communication bandwidth and lower latency between processes running in the same supernode or node group, respectively. Besides, the K computer's mesh-torus interconnect is well-suited for applications in which processes mainly communicate with their immediate neighbors. On the other hand, all-to-all communications or communications between distant nodes can be costly in terms of latency. Communication between distant nodes additionally causes network contention all along the message path.

### 2.1.3 Hybrid Systems

Current HPC systems cannot be classified into either shared- or distributed-memory systems. Instead, they follow a hybrid design approach, as illustrated in Figure 2.4: each multi-core processor is part of a *UMA socket*. Multiple UMA sockets together form a shared-memory *ccNUMA node*. An HPC system consists of multiple ccNUMA nodes, potentially thousands to tens of thousands. Each ccNUMA node is connected to a network, forming a (hybrid) distributed-memory system comprised of shared-memory nodes.

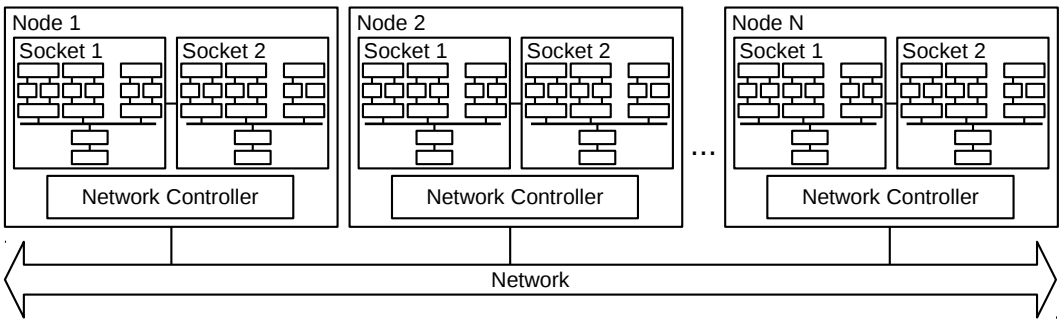



FIGURE 2.4: Hybrid system; shared-memory ccNUMA nodes arranged in a distributed-memory configuration.

### 2.1.4 Heterogeneous Systems

To achieve higher performance and better energy efficiency systems can incorporate multiple types of processors. The different processor types can differ in a variety of characteristics. They can use the same or different instruction set architectures (ISAs) and vary in performance and power consumption. An example is systems

consisting of both multi-core processors and *general-purpose graphics processing units* (GPGPUs). If a processor is optimized for a certain class of computations it is also referred to as an *accelerator*.

The TOP 500 list [113] lists the 500 fastest HPC systems in the world and is updated every six months. Figure 2.5 displays the percentage of different types of accelerators used by systems in the TOP 500 list over time. Some accelerator types, e.g. the IBM Cell processor or ATI Radeon GPGPUs, had some significance in the past but disappeared afterwards. As can be seen in the figure, today the market for accelerators in HPC is dominated by NVIDIA GPGPUs and Intel MIC accelerators. Interestingly, lately, the total amount of system using accelerators is decreasing.

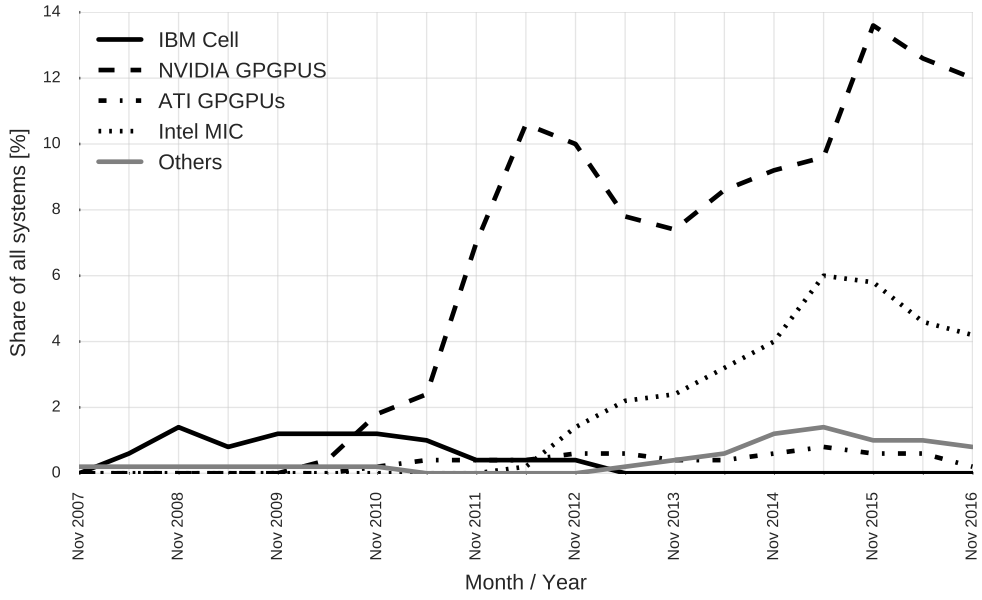



FIGURE 2.5: Percentage of systems in the TOP 500 list using accelerators over time.

Another example of a heterogeneous system is the *ARM big.LITTLE* architecture, designed to improve energy efficiency of battery-powered mobile devices. In big.LITTLE, fast, more power-consuming ("big") cores are combined with slower, more energy-efficient ("little") cores. Early implementations had restrictions, such as only allowing to use either all big or all little cores. Starting with the Samsung Exynos 5420 SoC, these limitations have been overcome. A big.LITTLE system can also include a GPGPU. While big and little cores have the same ISA, the GPGPU usually has a different ISA. However, big cores, little cores and the GPGPU share the same physical memory address space, thus forming a ccNUMA system.

### 2.1.5 Implications on Design Techniques for Future Systems

In the previous subsections we pointed out how the hardware complexity of modern computer systems is continuously increasing. In HPC systems, complexity increases both at the intra-node and the inter-node levels. At the intra-node level core counts and core heterogeneity are increasing over time. At the inter-node level, larger numbers of nodes need to be interconnected, requiring new network technologies.

The increase in overall system complexity exceeds the capabilities of existing simulation techniques. New, advanced techniques are needed to simulate future systems at a good accuracy and in a reasonable amount of simulation time. In this thesis, we present techniques for improving simulation speed at the intra-node and inter-node levels, while maintaining high simulation accuracy.

## 2.2 Parallel Programming Models

Parallel programming models can be classified according to a variety of different criteria. In this section, we present a classification along two orthogonal dimensions. The first dimension is the way in which a programmer manages different parallel execution threads. The second dimension consists in the way the programmer decomposes a problem in order to make it suitable for parallel execution.

### 2.2.1 Shared-Memory Programming Models

A parallel programming model for shared-memory machines provides a means to create several execution threads that share all or part of their memory address space. A program using more than one thread is referred to as a *multi-threaded program*. The decomposition of a program into multiple threads can generate *race conditions*, under which the outcome of a program depends on the order in which the different threads access shared data. Undesired race conditions can be avoided with the aid of synchronization primitives, namely *locks* and *semaphores*.

In traditional parallel programming models for shared-memory systems, like *POSIX Threads* (Pthreads) [19], the programmer explicitly decomposes an application into concurrent instruction streams and manages synchronization between those. These instruction streams are processed simultaneously by different threads. While Pthreads gives the application developer a large degree of control, the resulting programs are difficult to maintain. This is mainly due to the low degree of similarity between the parallel code and a sequential implementation, which results in low code readability. Furthermore, Pthreads programs can be tailored to a particular architecture, hindering performance portability.

The most prevalent shared-memory programming model today is *OpenMP* [39]. OpenMP supports the C, C++ and Fortran programming languages. It allows the programmer to parallelize a sequential version of a program by adding preprocessor directives. An advantage of OpenMP is that it allows a programmer to write a parallel program in an incremental fashion, starting with a sequential implementation. This increases programmer productivity and improves code readability.

LISTING 2.1: Dense matrix-matrix multiplication in OpenMP

```
1  [sequential code]
2
3  #pragma omp parallel for
4  for ( i = 0; i < n; i++ ) {
5      for ( j = 0; j < n; j++ ) {
6          c[i][j] = 0.0;
7          for ( k = 0; k < n; k++ ) {
8              c[i][j] = c[i][j] + a[i][k] * b[k][j];
9          }
10     }
11 }
12
13 [more sequential code]
```

OpenMP is an example of a programming model supporting the *fork-join* paradigm. The code fragment in Listing 2.1 shows a case of a multiplication of two matrices of dimension $n \times n$, implemented using OpenMP. Lines 4 to 11 implement the actual matrix multiplication. The declaration in line 3 fulfills two functions: first, it creates a *thread team*, consisting of a user-specified number of worker threads. If no thread count is specified, the number of worker threads is equal to the number of hardware threads of the host machine. Second, the statement in line 3 causes the iteration space of the outermost for-loop to be split into equally-sized *chunks*. The number of chunks is equal to the number of threads, and each chunk is assigned to a different thread. Once all threads finish the execution of their respective chunks, the worker threads are destroyed, and the main thread continues the sequential execution of the program.

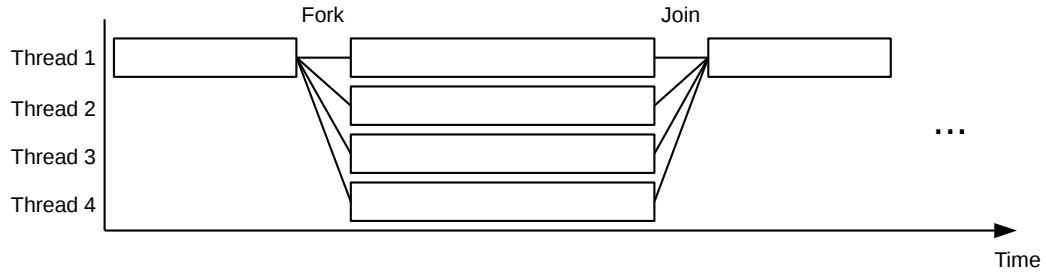



FIGURE 2.6: Illustration of parallel execution in an OpenMP application with four execution threads

Figure 2.6 illustrates the parallel execution of the previous example with four threads. In the beginning, one thread executes the sequential portion of the program. When entering the parallel section, three more threads are created. Afterwards, all threads participate in the parallel computation. Threads can communicate explicitly by accessing the same, shared memory of the application. At the end of the parallel phase, all threads implicitly synchronize, and the additionally created threads are destroyed. Then, the main thread continues with the execution of the sequential parts of the application.

The aforementioned mechanism of distributing work across several threads is referred to as *work sharing*. OpenMP also supports *tasking*, which is introduced in Subsection 2.2.4 later in this section. Other examples of shared-memory programming models are *Cilk* [15] and *Intel Thread Building Blocks* (TBB) [96].

### 2.2.2 Message Passing Programming Models

In message passing programming models, threads have only private memory. Communication is achieved by means of messages exchanged between threads. Typically, message passing relies on parallelization across different processes or *ranks*, in contrast to the threads used by shared-memory programs. The most widespread representative of this class of programming models is the *Message Passing Interface* (MPI) [56, 82]. MPI offers a variety of functions for sending and receiving messages between two processes (point-to-point communication). There are also communication primitives involving more than two processes, e.g. all-to-all communications, in which each process communicates with all other processes. Other examples are one-to-all and all-to-one communications. They are also referred to as scatter and gather operations, respectively.

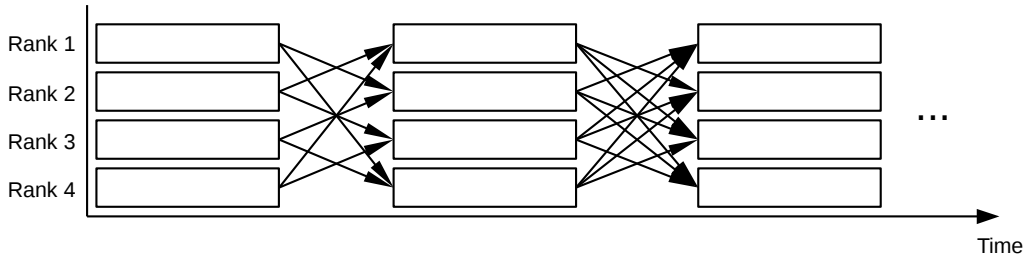



FIGURE 2.7: Illustration of computation phases (boxes) and messages (arrows) in an MPI application with four ranks

Figure 2.7 illustrates an MPI application executed with four ranks, each of which runs on a different node of a cluster. In the beginning, all ranks perform computations. At some point, each rank sends messages to its immediate neighbors. This communication scheme is referred to as point-to-point communication. When the communication is complete, all ranks resume computation. After some time, all

ranks exchange messages with each other, performing an all-to-all communication, before once more they resume computation.

In MPI, a program is parallelized across different processes, each of which runs on a different core. The cores can be located in different nodes, in different sockets of the same node, or in the same socket. Processes typically communicate via a high-bandwidth, low-latency network. If two communicating processes are located on the same node, current MPI implementations avoid using the network interface for communication. Instead, communication is achieved via shared memory in a way which is transparent to the user, managed by the MPI library.

Note that the distinction between shared-memory and message passing programming models is based on the programmer's view of memory. Shared-memory programming models map naturally to shared-memory systems, e.g. CMPs. On the other hand, message passing maps naturally to distributed-memory machines, such as clusters. However, it is also possible to program shared-memory machines using message passing or to program distributed-memory machines with shared-memory programming models. For example, Intel's programming model *Cluster OpenMP* hides explicit message passing from the programmer by creating the illusion of a single address space encompassing the entire system's memory [60].

### 2.2.3 Functional Parallelism vs. Data Parallelism

In the previous sections, we classify programming models into shared-memory and message passing programming models, according to the programmer's view of a system's memory. In this section we consider a different classification, according to the way in which a program exposes parallelism. Note that this classification is orthogonal to the one presented in the previous section.

The implementation of a parallel program requires the decomposition of a problem so that its computation can be accelerated by using multiple threads. The two dominant types of parallelism are *functional parallelism* and *data parallelism*. Exploiting these different kinds of parallelism requires different programming strategies.

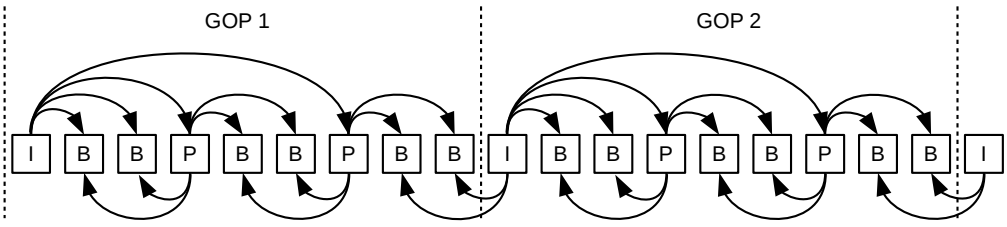



FIGURE 2.8: Example of functional parallelism: Dependencies between different frames in an H.264 video stream

A program is said to contain functional parallelism performs several tasks sequentially, which could partially or entirely be executed in parallel without violating

program correctness. An example is decoding a video encoded in the *H.264* standard [120]. As illustrated in Figure 2.8, a Group-of-Pictures (GOP) in an H.264 video contains so-called I-, P- and B-Frames. I-Frames are independent of other frames and can be decoded in parallel. P-Frames depend on the previous I-Frame. Once this I-Frame is decoded, all P-Frames of the GOP can be decoded in parallel. Finally, B-Frames depend on the surrounding non-B-Frames. Once those are decoded, all B-Frames in a subsequence can be decoded in parallel.

A program containing data parallelism, repeatedly performs the same operation on different elements or ranges of its data, whereas these operations could be executed in parallel without violating program correctness. Several parallel execution paradigms exploit data parallelism.

A *Single Instruction, Multiple Data* (SIMD) machine can exploit data parallelism which is known at compile time. Special machine instructions operate on multiple data items at a time. For example, Intel's AVX512 instruction set extensions provide instructions which can operate on up to 64 byte-sized operands with a single instruction. A second example is GPGPUs, in which multiple hardware threads share the same control logic and operate in lockstep on different data. Finally, the matrix multiplication example in Section 2.2.1 also relies on data parallelism.

Data parallelism can also be exploited by independent processors executing the same program operating on different parts of the data domain, resulting in a *Single Program, Multiple Data* (SPMD) execution scheme. An advantage of SPMD is that data parallelism does not need to be known at compile time. Furthermore, SPMD machines show a higher tolerance for control flow divergence, which occurs when different threads follow different control paths due to data dependence of the control flow.

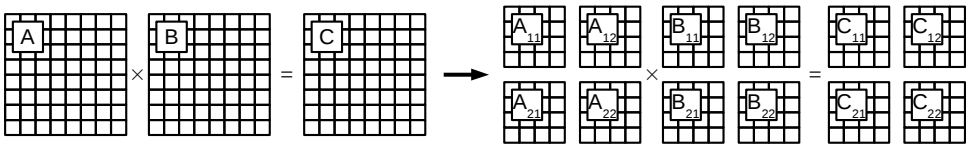



FIGURE 2.9: Example of exploiting data parallelism through domain decomposition: blocked matrix-matrix multiplication

One way to exploit data parallelism in SPMD machines is *domain decomposition*, the decomposition of the problem domain into *blocks* or *tiles*, which can be processed independently. Different threads can operate on different blocks simultaneously, accelerating the overall program execution. Figure 2.9 illustrates three matrices. The matrices $A$ and $B$ are to be multiplied, and the result stored in matrix $C$. After splitting the three matrices into sub-matrices, the multiplication rules for blocked matrices can be applied and the sub-matrices of matrix $C$ can be calculated in parallel,

according to Equations 2.1 to 2.4.

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} \tag{2.1}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21} \tag{2.2}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22} \tag{2.3}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22} \tag{2.4}$$

### 2.2.4 Task-Based Programming Models

A multi-threaded execution is said to be load-balanced if all threads reach a synchronization point at the same time. The absence of load balance is referred to as *load imbalance*. Load imbalance is a common problem with multi-threaded programs since it limits parallel efficiency and, consequently, application scalability. Task-based programming models have the potential to alleviate load imbalance and thus increase parallel efficiency. When implementing a parallel program using a task-based programming model, the programmer specifies program parts as *tasks* and, optionally, data dependencies between these tasks. Tasks are instantiated many times during the execution of a program, resulting in a large number of task instances. A runtime environment dynamically schedules task instances to available execution threads, taking into account the dependencies between different task instances.

Due to a fine-grained *over-decomposition* of the application, there are ideally more task instances ready for execution than there are threads. This allows the runtime environment to balance the workload assigned to each thread dynamically [79]. Further optimizations are possible if the architecture interfaces directly with the runtime environment [30, 114].

In this work, we differentiate between *task types* and *task instances*. Every execution of a task declaration statement at runtime results in the creation of a task instance. All task instances resulting from the same task declaration statement in the source code are said to be of the same task type. In a typical task-based program, the number of task types is small, i.e. up to a few tens. On the other hand, the number of task instances can lie in the order of thousands to millions.

The decomposition of a sequential program into several task types can be seen as a manner of functional decomposition, while the repeated instantiation of a task type can be regarded as domain decomposition. Thus, task-based programming models allow the programmer to exploit both functional and data parallelism.

Figure 2.10 shows a task dependency graph of a task-based implementation of a Cholesky decomposition. The program consists of four task types, which are calls to the functions `dpotrf`, `dtrsm`, `dsyrk` and `dgemm` of the *Level 3 Basic Linear Algebra Subprograms* (BLAS) [76]. In the beginning, only one task instance can be executed,

since it gives a dependency of all other task instances. When this instance finishes execution, more parallelism becomes available. Note that the last four task instances need to be executed sequentially due to data dependencies.

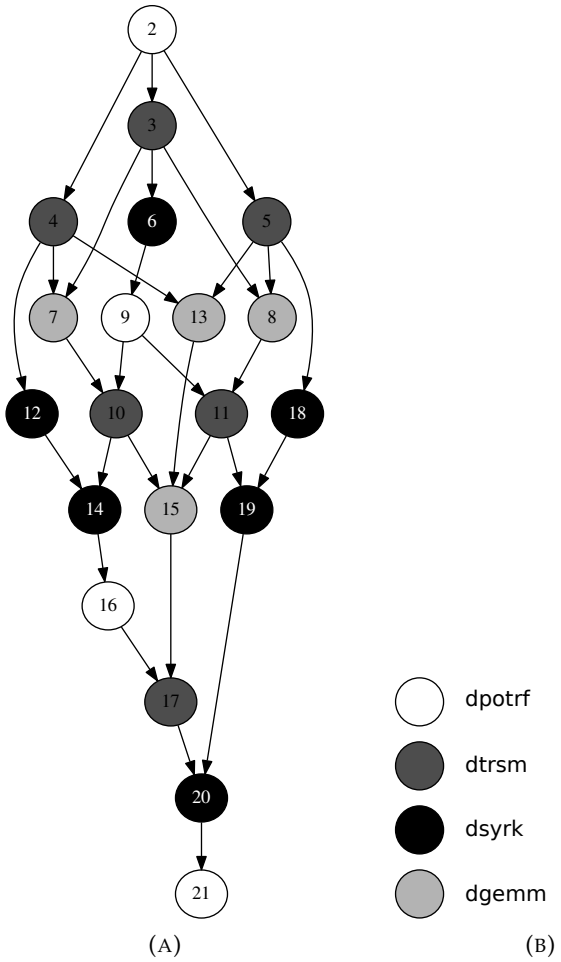



FIGURE 2.10: Dependency graph of task-based implementation of Cholesky decomposition

An example of a programming model supporting tasks is OpenMP [39]. Starting with a sequential version of a program, the programmer adds source code annotations to indicate which parts of the program are to be considered as tasks. These tasks are annotated with the data read and written by each instance of the task. The resulting parallel program is typically more intuitive to programmers, compared to low-level parallel programming models. Besides, development and debugging techniques are similar to the methods for the development of single threaded applications. Other parallel programming models supporting tasks are Intel Thread Building Blocks (TBB) [96] and OmpSs [42].

### 2.2.5 Hybrid Programming Models

As stated in Section 2.1.3, current HPC systems consist of a large number of nodes, forming a distributed-memory system. Each node itself is a shared-memory multi-core system. One way to program these machines is the use of *hybrid programming models*. In a hybrid programming model, a distributed-memory programming model is employed to perform a coarse-grain parallelization of the workload across the different nodes of the system. A shared-memory programming model further parallelizes the workload across the different processors within a node. A widely used hybrid programming model is MPI+OpenMP [92].

Hybrid programming models seem to be a natural fit for the hybrid nature of current HPC systems. However, they also introduce more variables which need to be tuned by system users. Traditional distributed-memory programs, relying purely on message passing for parallelization, are typically run with one process per processor. In a hybrid program, a single process can run on multiple threads. An application might scale well with the number of processes, but not with the number of threads per process and vice versa. It is up to the user to determine the ideal numbers of processes and threads per process.

## 2.3 Architectural Simulation

Architectural simulation is an important tool for computer architects in academic research and industry [2]. Simulation allows evaluating architectural features and their impact on performance and power consumption without actually building a costly prototype of the proposed design. For example, *out-of-order execution*, a feature used by virtually every modern processor from mobile to high-performance systems, was first evaluated in simulation [63]. Also, developers of system software and applications resort to simulation while real hardware is not yet available.

The requirements of computer architects to a simulator are typically different from the requirements of software developers. While the architect is mainly interested in accurately modeling the relevant hardware structures and their impact on performance, the software developer wants a simulator that supports the instruction set architecture (ISA) of the future machine.

### 2.3.1 Functional vs. Performance Simulation

According to the different needs of computer architects and software developers relying on simulation, simulators can be classified roughly into two classes, namely

*functional simulators* and *performance simulators*. However, current simulators frequently offer both functional and performance simulation modes and thus cannot be classified into one of those two categories.

Functional simulation, also referred to as *emulation*, aims to provide a means to execute software designed for a computer system which is not available. Examples are new systems which are not yet available on the market, systems which are too expensive to buy, or legacy systems which are no longer available to the software developer. Functional simulators can be classified into *interpreters* and simulators based on *native execution*. Interpreters simulate the execution of a program instruction-by-instruction within the simulator program. Simulators based on native execution, on the other hand, execute the instruction of the simulated program on the host machine the simulator is running on. This results in a higher simulation speed, compared to interpreters. Functional simulators do not model micro-architectural details. Therefore they are not suitable for performance estimations of the simulated system.

Performance simulation, also called *timing simulation*, aims to accurately predict performance metrics and in some cases power consumption of a computer system. Performance simulators include models of the relevant hardware structures that affect performance. Typically, performance depends on a variety of system components and performance bottlenecks can occur in different parts of the system, according to the workload currently being executed and the historic state of the machine. Therefore, performance simulation usually relies on detailed simulation models of the processor cores, the memory hierarchy and the on-chip interconnect network.

Performance simulators can be further sub-classified into instruction schedulers and cycle timers. Instruction schedulers model the propagation of machine instructions through the processor pipeline. Thereby, the simulator models the effect of each instruction on the architectural and the micro-architectural state. In contrast, cycle timers only model timing of a component of interest.

The highest degree of detail is achieved with models at the register-transfer level (RTL), which describes the combinational and sequential logic of the simulated system in its entirety. However, RTL models have several drawbacks. First, due to their high level of detail, their development is very time intensive and therefore difficult to manage in an academic research environment. Second, RTL simulations are orders of magnitude slower than their more abstract, higher-level counterparts. Therefore, in academic studies, simulating an entire system in an RTL simulation plays a minor role. However, RTL simulations are used in industry due to their higher accuracy and for verifying a design before tape-out.

Researchers in academia tend to rely on a variety of simulators, many of which are distributed under open-source licenses and have an active developer community.

Instead of describing the simulated design at the bit-level using a hardware description language, they use higher-level data types and software engineering techniques found in modern programming languages, e.g. C++.

### 2.3.2 Simulation of Shared-Memory Systems

There is a variety of simulators for shared-memory multi-core systems, offering different tradeoffs between simulation speed and detail. In this section, we introduce several state-of-the-art multi-core simulators. Table 2.1 gives a quick overview of the different simulators.

TABLE 2.1: Classification of shared-memory multi-core simulators

| Name of Simulator | Trace- / execution-driven | Functional / performance simulator | Instruction scheduler / Cycle timer |
|---|---|---|---|
| gem5 | Both | Both | Instruction scheduler |
| Graphite | Execution driven | Performance | Cycle timer |
| Sniper | Execution driven | Performance | Cycle timer |
| TaskSim | Trace driven | Performance | Cycle timer |
| ZSim | Execution driven | Performance | Cycle timer |
| COTSon | Execution driven | Both | Cycle timer |
| ESESC | Execution driven | Performance | Cycle timer |

The *gem5* simulator [14] is a full-system simulator, i.e., it models an entire computer system including devices like I/O controllers and system timers. This allows gem5 to run unmodified versions of different operating systems on the simulated hardware. Besides, gem5 features core models at several levels of detail, ranging from a model employing virtualization and running at near-native speed [104] to a detailed model of a superscalar out-of-order core. Amongst others, gem5 supports the x86 and ARM architectures, which are the most common architectures today.

*Graphite* [83] is a simulator for shared- and distributed-memory systems. It achieves high simulation speed by parallelizing a simulation across multiple cores of the host system, or even across multiple systems. Graphite uses dynamic binary translation to perform functional simulation of the simulated application.

The binary translator instruments all instructions of the simulated program and feeds each thread's instruction stream to an analytical core performance model. Memory requests from the application are serviced by Graphite's simulated memory hierarchy. First, this provides the input to the performance models of the memory hierarchy, e.g. caches, on-chip interconnect and DRAM. Second, this approach decouples the memory address space of the simulated system from the simulation host machine and allows to parallelize the simulation of a shared-memory system across multiple hosts of a distributed-memory system.

*Sniper* [21], proposed by Carlson et al., is a simulator for shared-memory systems based on the Graphite simulator. Carlson et al. show that overly simplistic core performance models can introduce high simulation errors and extend Graphite by adding the *interval model* [48] as an improvement over Graphite's core models processing a fixed number of instructions per cycle. These models are also referred to as fixed-IPC or one-IPC models, since they model program execution at an IPC of one.

The interval model allows to simulate processors with superscalar out-of-order execution, whereas the one-IPC model assumes in-order instruction issue and commit stages and a scalar execution pipeline. The interval model assumes out-of-order execution at the maximum steady-state IPC, which is interrupted by *miss events*. If during steady-state execution a branch predictor miss or a cache miss occurs, the model accounts for the number of cycles which are required to resolve the miss. Afterwards, execution at steady-state IPC is resumed. Consecutive, dependent misses are accounted for separately. The higher level of abstraction of interval simulation is directly reflected in a higher simulation speed, compared to more detailed models.

*TaskSim* [99, 100] is a trace-based simulator, meaning that a trace of the simulated application is generated before simulation. This trace is afterwards used by all simulations involving the corresponding application. The TaskSim tracer traces the computation phases of an application and the parallelism management operations, e.g. work creation and scheduling primitives in the runtime system. This allows the tracer to be single-threaded, while a trace can be used to simulate the execution of the application with an arbitrary number of execution threads. Another advantage is that also the simulator can be a single-threaded process since it does not need to perform functional simulation of the simulated application. TaskSim interfaces with an unmodified instance of the OmpSs runtime system. TaskSim exposes the simulated cores to the runtime system, which then schedules work units for execution on those simulated cores. The instruction streams of these work units are read from the application trace.

TaskSim features a detailed and an abstract simulation mode. The detailed mode,

also referred to as *Memory* mode, models a superscalar processor core featuring out-of-order execution, based on the Reorder-Buffer Occupancy Analysis technique [77]. The core of this technique is a model of the reorder-buffer. According to the specified issue width of the simulated processor, a number of instructions is inserted into the head of the reorder-buffer in every cycle. If the reorder-buffer is full, the issue stage is halted. At the same time, instructions are committed from the tail of the reorder-buffer at a rate equivalent to the specified commit rate. Memory accesses are issued to an external model of the memory hierarchy, containing one or more levels of private cache, on-chip interconnect structures, shared caches, and DRAM.

In the abstract simulation mode, also called *Burst* mode, TaskSim employs a high-level core performance model. In Burst mode, computational phases are assumed to have the same duration as during trace generation. Optionally, these durations can be scaled by a user-defined factor. Microarchitectural core structures, as well as the components of the memory hierarchy, are not simulated. Therefore, Burst mode simulations do not capture contention on shared system resources. Instead, they allow evaluating an application's algorithmic scalability limit and its best-case scalability, assuming that the application does not cause significant contention on shared resources.

The *ZSim* simulator [103], proposed by Sanchez et al., relies on parallel simulation in order to achieve high simulation speed. ZSim achieves good parallel simulation scalability by relaxing synchronization between simulated cores. To this end, simulated time is split into windows of typically 10,000 cycles. In each window, the different threads are simulated without synchronization, and a per-core event trace is generated.

At the end of each window, a dependency graph of all events is constructed, and a timing model is invoked in order to determine the actual interleaving of the per-core events. This timing model is also executed in parallel. The event dependency graph is partitioned into different domains, and the simulation is synchronized only in case of an event dependency crossing different domains.

Sanchez et al. report a simulation speed of 1,500 MIPS for simulations of a thousand-core system. Although ZSim shows absolute performance prediction errors of up to 20%, it achieves errors of less than 5% for scalability predictions of benchmarks of the PARSEC benchmark suite [13].

*COTSon* [6] is a full-system simulator decoupling functional and timing simulation. Functional simulation relies on just-in-time compilation of the simulated program. COTSon features simulation models at several levels of detail and supports sampling. Sampling reduces simulation time by simulating in detail only the representative phases of a program and is introduced in detail later in this chapter.

In addition to performance, *ESESC* [5] also simulates a future design's power

consumption and thermal behavior. ESESC, an extension of the SESC simulator, is the first simulator applying time-based sampling to the simulation of multi-threaded applications. We elaborate more on time-based sampling in Section 2.4.2.

### 2.3.3 Simulation of Distributed-Memory Systems

The simulation of an application executing on a distributed-memory system is considerably more complex, and therefore time intensive than the simulation of a shared-memory multi-core system. First, it requires the simulation of all nodes involved in the computation. Second, besides the computation, also the network used for message passing between nodes needs to be simulated.

The complexity of large distributed memory systems requires using abstract simulation models, or even simplistic models for system components which are not relevant for the conducted study. E.g., when analyzing network performance, an application's computation phases are often modeled as durations, or *CPU bursts*. On the other hand, when studying the intra-node architecture and its impact on an application's computation phases, the network is frequently modeled by a high-level, analytical model.

The *SST/gem5* simulation framework [61] combines the *Structural Simulation Toolkit* (SST) [101] with the gem5 simulator [14]. SST is a scalable simulator for distributed-memory systems. However, its core models lack the amount of detail necessary for detailed architectural studies.

This lack of detail is addressed by the integration with gem5. While MPI communications are simulated by SST's network simulator component, the intra-node communications and computations are simulated using gem5. SST/gem5 supports checkpoints to resume a running simulation after storing the simulated system state on disk.

*Dimemas* [49, 73] is a trace-based simulator for distributed-memory systems. In a first step, an application trace is generated using the *Extrae* instrumentation library. Extrae intercepts all calls to the MPI library and generates a trace containing information on communication type (e.g. point-to-point or collective communication) and size. Besides, the resulting trace contains the duration of computation phases spent outside the MPI library.

The application trace generated with Extrae is afterwards used as an input to the Dimemas simulator. Dimemas uses different analytical models to simulate the performance of MPI communications, depending on a communication's type. For example, the duration $T$ of a point-to-point communication is modeled based on the message size $S$, and the bandwidth $B$ and latency $L$ of the link between sender and receiver, according to Eqn. 2.5:

$$T = L + \frac{S}{B} \tag{2.5}$$

Network contention is modeled by limiting the number of simultaneous links sharing a network connection to a user-specified value.

Collective communications, such as *one-to-all*, *all-to-one* or *all-to-all*, are modeled using a more elaborate model, taking into account that a communication can require multiple successive steps. Communications between processors in the same node are modeled using a different set of parameters, reflecting the difference in bandwidth and latency between inter- and intra-node communication.

Dimemas models different node architectures by applying a speed ratio to the computation phases, whose duration is recorded in the input trace. Hence, the speed ratio is the relative performance difference between the system used for trace generation and the simulated system. It is also possible to cluster computation phases and apply per-cluster speed ratios [51].

*SimGrid* [26] also relies purely on analytical models. Like Dimemas, SimGrid is trace-based and allows the user to provide speed ratios for modeling node architectures different from the system used for trace generation. SimGrid employs a piece-wise linear flow-based model [37] to account for network contention, which is not modeled by Dimemas.

### 2.3.4 Simulation of Hybrid Distributed-Shared-Memory Systems

Distributed-memory systems currently used in HPC consist of multi-core shared-memory nodes, which are interconnected by a high-speed, low-latency interconnect network. Therefore, these systems can be considered hybrid distributed-shared-memory systems. The simulation of an application executing on such a system involves the simulation of all nodes, and the interconnect network, which makes it much more complicated than the simulation of a single, shared-memory multi-core system.

Typically, simulation frameworks for hybrid systems consist of a network simulator, which models the communication between the different nodes of the simulated systems, and a multi-core simulator, which is responsible for modeling the single nodes of the system. An example of a simulator supporting simulations of hybrid systems is SST/gem5, introduced in the previous subsection.

## 2.4 Acceleration Techniques for Architectural Simulation

Over the last decade, simulation of shared-memory systems has become increasingly time-consuming. Since the advent of CMPs, the number of processor cores integrated on a single chip is continuously increasing. Simulations of designs including a larger number of cores also require simulating larger numbers of instructions in order to meaningfully stimulate the simulated systems. The communication

between cores and the contention on shared system resources additionally increase simulation complexity.

### 2.4.1 Checkpointing

Oftentimes, it is desirable to simulate not the entire execution of a benchmark, but only a *region of interest*. However, especially execution-driven simulators require the simulation of a benchmark from the beginning. Consequently, all program parts leading to the region of interest, e.g. initialization of data structures, are simulated out of necessity. A solution to this problem is *checkpointing*. When using checkpointing, an image of the architectural state of the simulated system is stored on the simulation host, together with the state of the simulator. This image is referred to as a *checkpoint*. Checkpoints can be restored, allowing to resume a previously checkpointed simulation. Architectural simulation can be accelerated by creating a checkpoint before the region of interest. Successive simulations can start from this checkpoint. The technique can also be applied to multiple regions of interest.

### 2.4.2 Sampling

Sampling techniques accelerate architectural simulation by performing detailed performance simulation only on a subset, or *sample*, of a simulated program. The program parts not belonging to the sample are either simulated in a faster, functional-only simulation mode or even omitted. Finally, the performance metrics of the entire simulation are extrapolated, based on the performance information obtained during detailed simulation of the sample.

#### Single-Threaded Simulation Sampling

In their *SimPoint* methodology [107], Sherwood et al. use *basic block vectors* (BBVs) to identify the representative parts of a program's execution. Afterwards, only these representative parts are simulated in detail. Finally, the performance metrics of the entire program execution are extrapolated.

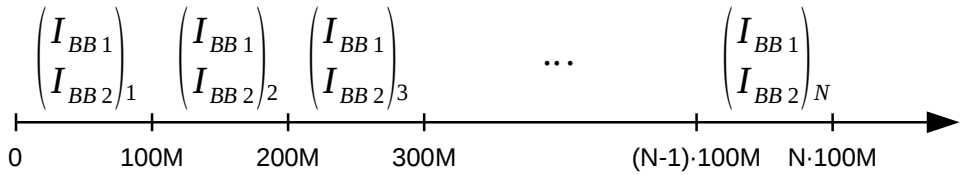



FIGURE 2.11: Generation of basic block vectors (BBVs) for a program consisting of two basic blocks

The first step of applying SimPoint is the generation of BBVs of the program which is to be simulated. While the program is executed natively or in a simulator,

the dynamic instruction stream is split into intervals of typically 100 million instructions, as illustrated in Figure 2.11. For each interval, a vector $BBV_i$ with as many dimensions as the total number of basic blocks in the program is constructed. The figure illustrates a hypothetic example of a program consisting of only two basic blocks. Each dimension of this vector is indexed by a different basic block and contains a counter for the number of dynamic instructions, belonging to the corresponding basic block, which are executed during each 100 million instruction interval.

The key idea behind SimPoint is that if two intervals have similar BBVs, they are similar in terms of the instructions the program executes during this interval and, hence, are likely to have similar performance. This similarity between BBVs is detected by applying *k-means* clustering [81] to the set of all BBVs. The resulting clusters contain BBVs of similar performance. By selecting one representative BBV of each cluster, one can obtain a set of 100 million instruction intervals capturing the entire behavior of the simulated program.

Detailed simulation is performed only on the representative intervals, also referred to as simulation points, while the remainder of the application up to the last detailed interval is simulated in a faster, functional simulation mode. After simulation, the performance metrics of all intervals are used to extrapolate the performance of the full detailed simulation, depending on the number of BBVs in each cluster. Sherwood et al. report an average IPC error of 3.0% [107].

Perelman et al. propose a technique to select statistically valid simulation points early in time [90]. The original SimPoint methodology spends a significant amount of time in functional simulation between simulation points. By choosing simulation points early in time, the amount of functional simulation leading up to the latest simulation point is significantly reduced.

The *Sampling Microarchitecture Simulation* (SMARTS) framework [121], proposed by Wunderlich et al., switches periodically between warmup, detailed simulation and simulation in fast-forward mode, as illustrated in Figure 2.12. During a warmup phase, $W$ instructions are simulated in detail, but the simulation statistics are ignored. The reason for this is that at the beginning of the simulation, the simulated architectural structures, like branch predictors and caches, are in their initial, *cold* state. Also, after a fast-forward phase, the micro-architectural state is stale and is brought up-to-date during the warmup phase.

Once the warmup phase is complete, the simulator starts measuring micro-architectural performance metrics while simulating $U$ instructions during the detailed simulation phase. At the end of the detailed phase, the simulation is switched to fast-forward mode, which executes the simulated program in a purely functional simulation mode without updating the micro-architectural state of the simulation.

Wunderlich et al. report warmup intervals of up to $W = 4000$ instructions

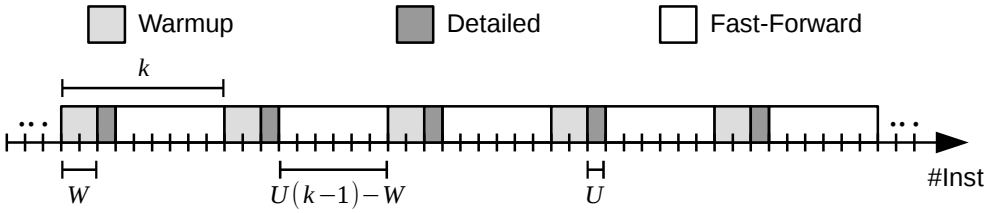



FIGURE 2.12: Periodic switching between warmup, detailed and fast-forward simulation modes in SMARTS

and detailed simulation intervals of $U = 1000$ instructions. The length of the fast-forward interval is set to a value which results in a total number of 10,000 intervals. Consequently, the total number of instructions simulated in detail, including warmup and detailed simulation, amounts to 500,000 instructions per benchmark, or less than 0.1% of the total instruction count across all benchmarks of the SPEC2000 benchmark suite [59]. Simulations using SMARTS are 60 times faster than full detailed simulations, which shows that simulation speedup is mainly limited by the speed of functional simulation during the fast-forward phases.

With *TurboSMARTS* [118], the same group proposes an extension to SMARTS. TurboSMARTS eliminates the functional simulation phases during the fast-forward intervals. In an apriori step, TurboSMARTS generates a checkpoint library of the simulated program, which can afterwards be used for all simulations of the program. Before each warmup interval, the correct architectural state is restored from this checkpoint library. The authors report simulation times of less than 2 minutes across all SPEC2000 benchmarks.

#### Multi-Threaded Simulation Sampling

The previously introduced sampling techniques for simulations of single-threaded architectures can not be directly applied to simulations of multi-threaded systems. In a single-threaded program, progress can be measured in terms of committed instructions. However, this is generally not valid in multi-threaded programs. Different threads of a multi-threaded program can progress at different rates, e.g. due to the inhomogeneous nature of the workload. Another example is the lack of fairness accessing shared system resources, e.g. when one thread monopolizes the last-level cache. Finally, a thread can be executing instructions that do not contribute to the progress of the program, e.g. while spinning on a lock.

For the reasons mentioned above, at any point in time, the different threads have typically executed different numbers of useful instructions in the past. Hence, the instruction count can not serve as a metric for measuring progress and identifying common points in time across multiple threads. The sampled simulation techniques

introduced in Section 2.4.2 rely on instruction count to measure progress and determine phase boundaries in simulated programs. Instead, a technique targeting simulation of multi-threaded programs must measure progress in terms of cycles, i.e. time, which is the only common metric across all threads. In the following, we present several techniques which are based on this insight.

Carlson et al. [24] apply periodic time-based sampling [5, 29] to parallel programs. Short, detailed simulation phases take turns with longer fast-forward phases, resulting in an overall reduction of simulation time. The duration of detailed- and fast-forward intervals is determined based on the periodicities of the simulated application and are measured in terms of cycles.

During detailed simulation, the performance metrics of interest are measured in a timing simulation of the different threads and their interactions with one another. During fast-forward phases, Carlson et al. employ functional simulation. All memory accesses are simulated by simulation models of the memory hierarchy, ensuring that the simulated caches are always in a representative state at the transition to detailed simulation. As the authors rightfully point out, it is possible to employ more elaborate warmup techniques before detailed simulation and thus eliminate the need for functional cache warmup during fast-forwarding.

As stated earlier, different threads of a multi-threaded program can progress at different rates. Detailed simulation is used to model each thread's interaction with the system resources and also its interaction with other threads. Therefore, the progress of each thread is modeled correctly. Fast-forwarding the simulation using functional simulation at a constant IPC, as it is typical in architectural simulators, would result in incorrect thread progress. At the beginning of the next detailed simulation interval, the thread interleaving would not be the same as if the entire simulation would have been run in detail. Carlson et al. minimize this problem by fast-forwarding each thread at the average IPC of the last detailed simulation interval. The technique achieves an average simulation speedup of 2.9 with an average execution time error of 3.5%.

One of the primary advantages of the technique is that it is not tied to a particular programming model. However, it is not directly applicable to task-based programs. The sampling paramters are determined apriori in a profiling run. During simulation, the correct sampling paramters can change due to different decisions of the dynamic scheduler.

BarrierPoint [23], also proposed by Carlson et al., first analyzes micro-architecture independent performance metrics of program sections between global barriers. Afterwards, the SimPoint infrastructure [107] identifies clusters of those inter-barrier regions with similar performance. Simulation time is reduced by simulating only one representative out of each cluster. BarrierPoint exploits the fact that all threads

synchronize at a global barrier and, hence, at the beginning of each inter-barrier region are aligned correctly.

Instead of characterizing program phases based on BBVs, BarrierPoint uses *signature vectors* (SVs). SVs are a combination of BBVs and *stack distance histograms*. The stack distance is the number of memory accesses to unique addresses between two accesses to the same memory location. A stack distance histogram is a histogram of the stack distances observed between the beginning of the program and the end of each inter-barrier region. Thus, stack distance histograms capture the historic behavior of all previous inter-barrier regions. In the case of BarrierPoint, the bins of the stack distance histogram are spaced according to powers of two. This allows for better resolution of smaller stack distances. Finally, for each inter-barrier region, the BBV and the stack distance histogram are combined, either by addition or by concatenation, to form an SV.

Once the SVs are generated, they are clustered using the existing, publicly available SimPoint tool [107]. SVs, in contrast to BBVs, capture the historic state of the memory hierarchy. Therefore, they allow detecting inter-barrier regions with different performance due to a different state of the memory hierarchy, which would go undetected when only using BBVs.

Finally, one representative inter-barrier region of each cluster is simulated in detail, and the overall program performance is extrapolated by applying weights to the per-region simulation statistics. BarrierPoint achieves an average simulation speedup of 24.7 with an average execution time error of 0.9%. In comparison to the aforementioned technique targeting general parallel applications, this shows that leveraging the nature of a parallel programming model can lead to significantly higher simulation speedup.

Task-based programs aim at avoiding global barriers. Instead, synchronization is achieved by dynamically scheduling different task instances in a valid execution order. For this reason, the BarrierPoint technique is not generally applicable to task-based programs.

In their *Multilevel Simulation* technique, Gonzalez et al. [51] identify representative phases (*CPU bursts*) of programs implemented in MPI programming model. These representative CPU bursts are identified during profiling before simulation and are afterwards simulated in detail. The obtained performance information is then used to extrapolate the overall program performance.

#### Warmup in Multi-Threaded Simulations

Warmup for single-threaded simulations has been extensively studied [38, 43, 58, 107, 117, 121]. The technique used by BarrierPoint combines two existing methodologies, namely functional warmup [38] and checkpointing [117]. The resulting technique uses dynamic instrumentation to track the most recent memory accesses on a per-cache-line basis. Afterwards, this information is used to restore cache state at the beginning of each detailed simulation interval.

Luo et al. [124] propose *Self-Monitored Adaptive Cache Warm-Up* (SMA), a technique not requiring profiling before simulation. Every cache in a simulated system monitors its fraction of used lines over time. When this portion passes a threshold or remains constant during a certain time, a cache is considered warmed. The authors evaluate SMA for single-threaded simulations. However, no fundamental reasons are impeding its applicability to multi-threaded simulations.

### 2.4.3 Statistical Simulation

Detailed simulation of a full program execution can be very time-consuming. Statistical models aim to reduce simulation time by creating a synthetic workload with the same statistical properties as the dynamic instruction stream of a full program execution.

The HLS simulator [85] creates a statistical profile of an application while simulating it in an architectural simulator. For each static instruction of the application, the profile contains the functional unit requirements, miss-rate distributions the instruction causes at the different cache levels, and the distance to other instructions on which the current instruction depends. This analysis is done on a per-basic-block basis. The branch instruction at the end of each basic block is assigned the predictability value observed during profiling in the architectural simulator. Finally, the instruction profile is simulated repetitively, until the simulated IPC converges.

Nussbaum et al. propose a statistical performance model for superscalar processors [87]. First, an instruction trace of the application to be simulated is captured during detailed simulation in an architectural simulator. Afterwards, the instruction mix, namely the percentages of dynamic instructions belonging to each out of 14 different instruction types, is determined. At the same time, a distribution of the lengths of the dependency chains between dynamic instructions is determined. Finally, a synthetic instruction trace with the same instruction mix and inter-instruction dependency distributions. This trace is then used as an input to an architectural simulator.

The aforementioned statistical simulation techniques use micro-architecture dependent information to capture the behavior of the simulated branch predictor. If

the parameters of the branch predictor changes, the application profile needs to be regenerated. Eeckhout et al. [44] improve on this model by using statistical flow graphs. A statistical flow graph of order $n$ predicts the outcome of a branch, depending on the outcomes of the last $n$ executions of that branch, similar to the way actual branch predictors work.

### 2.4.4 Analytical Models

Analytical models follow the goal of avoiding architectural simulation and instead rely on a set of analytical expressions to predict the performance of a system executing a specified workload. The two major classes of analytical performance models are *mechanistic* and *empirical* models. Mechanistic models are built in a constructive way with equations describing how the different architectural structures and their interaction affect performance. Mechanistic models, therefore, allow reasoning about why a particular design is better or worse than another. Empirical models, on the other hand, are usually models developed in the field of machine learning e.g. artificial neural networks of support vector machines. These models are trained on a set of detailed reference simulations.

The *Interval Model* [16], presented by Breughe et al., models a program's execution on a hypothetic system by assuming an execution at the designed steady-state IPC. The execution at this maximum sustainable IPC is disrupted by miss events. Breughe et al. distinguish between miss events occurring in the processor front-end, e.g. branch mispredictions and instruction cache misses, and miss events taking place in the back-end, e.g. last-level cache misses. While misses in the front-end are serialized, long-latency misses in the back-end, e.g. DRAM accesses, can partially overlap. The interval model predicts the overall performance of an application executed on the modeled system. It does not take into account the effects of single instructions.

Genbrugge et al. extend this model for simulations of multi-threaded systems in their *Interval Simulation* methodology [48]. In contrast to the original Interval Model, Interval Simulation takes into account the effects of single instructions and how they interact with each other in shared system resources.

Interval Simulation uses one model instance per simulated processor core. A functional simulator supplies the per-core models with instructions. In the absence of miss events, each model processes instructions at a rate equal to the processor width. Dedicated simulation models simulate the occurrence of miss events. E.g., a branch predictor model predicts if a branch misspredict occurs. If a miss event happens, the interval model of the corresponding core accounts for the latency introduced by the event. Note that, since the application profile is generated on-the-fly, it is regenerated for each combination of evaluated architecture and application.

Van den Steen et al. [115] propose an extension of the Interval Simulation model. Interval Simulation requires micro-architecture dependent input data, namely the number of cache misses per cache level, the number of branch predictor misses and the amount of memory-level parallelism (MLP). The approach presented by Van den Steen et al. eliminates the micro-architecture dependent parts of the model input. Instead, they use a micro-architecture independent application profile and generate the micro-architecture dependent elements of the model input using analytical models for caches, branch predictors and MLP.

Caches are modeled using *StatStack* [46], a technique for modeling arbitrarily sized LRU caches. StatStack's model requires the *reuse distances* of the modeled application as an input. The reuse distance is the number of memory accesses between two accesses to the same cache line. Based on the reuse distance profile, StatStack predicts an application's cache miss rate.

Branch predictors are modeled using the *Linear Entropy* model [91], proposed by Pestel et al. First, the application to be modeled is executed in a profiler which, for each static branch instruction and each history of past branches, counts the number of times the corresponding branch is taken and not taken. This information is afterwards used to calculate each branch's entropy. A linear model predicts the per-branch miss rate for several different branch predictors based on the per-branch entropy.

MLP is the number of simultaneously outstanding LLC misses, i.e. the number of memory accesses which can be served by the DRAM subsystem in parallel. Van den Steen et al. propose an MLP model, which separates MLP calculation into a fraction stemming from LLC cold misses and a portion arising from capacity and conflict misses.

In Chapter 5, we present TaskPoint, a sampled simulation methodology for task-based programs. We show how we use a modified version of the model proposed by Van den Steen et al. to improve the accuracy of TaskPoint.

Casas et al. propose an analytical performance model for MPI applications [28]. First, an execution trace of the application is generated which contains time-stamped information about the occurrence of MPI calls or hardware performance counters, e.g. the number of executed floating point operations per second. This information is converted into a time series. By applying Discrete Wavelet Transform to this time series, Casas et al. identify periodic behavior in the application and filter the application trace for size reduction. Afterwards, they apply an analytical model to predict the application's speedup for different numbers of processors.

### 2.4.5 Reduced Input Sets

A common way to reduce simulation time is to simulate benchmark executions using smaller input sets under the assumption that the performance characteristics of the benchmark are not affected. However, changing the input set frequently changes important properties of a benchmark, e.g. the instruction mix or the amount of parallelism which can be exploited by a multi-core system.

KleinOsowski et al. present *MinneSPEC* [70, 71], a modified input set for the SPEC2000 benchmark suite [59]. The authors show, that across all benchmarks either the percentages of calls to the different functions within a benchmark or the instruction mix do not match the values observed when using the *reference* input set. Hsu et al. [62] confirm that the SPEC2000 benchmarks show different performance for different input sets.

Southern et al. [110] present a study of the scalability of the benchmarks constituting the PARSEC benchmark suite [13]. All PARSEC benchmarks can be executed with alternative input sets which are designed for architectural simulation. However, Southern et al. show that some benchmarks, when using the largest simulation input, achieve a scalability several times lower than the scalability observed for the native execution input.

For the reasons mentioned above, it is often impossible to reduce simulation time by reducing the input size without significantly affecting a benchmark's performance characteristics. Therefore, in the scope of this work, we develop several techniques for simulation time reduction based on sampling. These techniques are presented in Chapters 5 and 6.

### 2.4.6 Parallelization

On a parallel system, architectural simulations can be parallelized in a trivial way by executing multiple instances of a single-threaded simulator simultaneously. While this technique does not reduce the time required for a single simulation, it can significantly increase simulation throughput. This is especially the case during the early phase of design space exploration when tens to hundreds of thousands of simulations need to be executed.

There are also approaches to parallelizing the simulator itself. At first glance, architectural simulators used to simulate multi-core designs seem a natural fit for parallelization on a multi-core shared-memory host. Ideally, each core of the host would handle one or more simulated cores. However, threads running on different cores of the simulated systems compete for shared system resources. Therefore, the different simulation threads are frequently forced to synchronize, limiting scalability. In order to circumvent this problem, parallel simulators frequently employ

techniques to relax the synchronization requirements between different simulator threads.

The *Wisconsin Wind Tunnel II* (WWT2) [86] executes different simulated cores in different threads on the host system. Enforcing cycle-by-cycle synchronization among the simulated cores implies a significant synchronization overhead, which results in low simulation speed improvement for parallel simulations. WWT2 splits simulation time into quanta, during which processors do not affect each other's state. Quanta are executed in parallel. At the end of each quantum, the simulated processors synchronize. WWT2's parallelization approach is conservative, i.e. it always maintains temporal causality between the simulated cores.

*SlackSim* [32] allows the simulated time to diverge by a user-specified number of cycles. This temporal slack reduces synchronization overhead and allows for better simulation scalability. The main difference to WWT2 is that different simulated cores are only throttled if they diverge by more than the user-specified slack, an approach which is not conservative. The authors of SlackSim report a simulation error of up to 0.7% for a maximum slack of 100 cycles. For unlimited slack, the error amounts to up to 4% at only slightly better simulation speedup, compared to a slack of 100 cycles.

### 2.4.7 Hardware Acceleration

The main limit to the scalability of architectural simulators is synchronization between simulation models of tightly synchronized system components, e.g. different cores, cache memories and the on-chip interconnect. In a hardware instance of a system, synchronization between system components happens in parallel via dedicated signal lines, all of which can operate in parallel. In an architectural simulator, synchronization is achieved with the help of software techniques, i.e. locks and semaphores. As a result, many events which happen in parallel in a real system are processed sequentially by the simulator.

There are proposals of using hardware acceleration to circumvent this problem. A promising candidate is *Field-Programmable Gate Arrays* (FPGAs). FPGAs consist of generic logic, storage elements and a configurable interconnect fabric. FPGA vendors provide tools to synthesize RTL descriptions and generate a bitstream with the configuration data for the FPGA. The amount of resources on a single FPGA scales with Moore's Law, as do the systems modeled by computer architects. Therefore, FPGAs are a promising platform for architectural simulation.

The *FPGA-Accelerated Simulation Technologies* (FAST) framework [34] splits the simulation of a program executing on a single-core system into two parts, namely functional and timing simulation. Functional simulation of the simulated program is performed in software using QEMU. The dynamic instruction stream executed by

QEMU is forwarded to a timing model residing in an FPGA. The authors of FAST report an average simulation speed of 1.2 MIPS, which is comparable to the speed of functional-only simulation of simulators performed in software.

*RAMP gold* [111] proposed by Tan et al., also separates functional and timing simulation. In contrast to FAST, RAMP gold is able to simulate target systems with up to 64 cores. Another difference to FAST is that also functional simulation is performed by logic on the FPGA, improving simulation speed and minimizing communication with a host server. Tan et al. report simulation speeds of up to 50 MIPS, which is a significant improvement over FAST.

# Chapter 3

# Experimental Setup

In this chapter, we introduce the experimental setup used for the studies in the subsequent chapters. First, we introduce the OmpSs programming model, which is used throughout this thesis. Then, we present the different multi-core platforms used in our analysis. Afterwards, we introduce the benchmarks used for the evaluation of our sampled simulation methodologies for shared-memory and hybrid systems. Finally, we introduce our methodology for measuring performance of task-based programs in native execution.

## 3.1 The OmpSs Programming Model

For our evaluations we choose the OmpSs programming model [42]. The OmpSs compiler and runtime environment are available as open source. OmpSs allows the programmer to declare tasks and annotate them with data inputs and outputs. Using this information, the OmpSs runtime system schedules task instances taking data dependencies into account and performs synchronization only when necessary. These OmpSs features were included into the specifications of OpenMP 3.0 and 4.0.

OmpSs consists of the Mercurium compiler and the NANOS++ runtime environment. Mercurium is a source-to-source compiler supporting the C, C++ and Fortran programming languages. It translates a program annotated with C-style compiler directives (*pragmas*) into an intermediate representation containing calls to the NANOS++ API for task management and data transfer. This intermediate representation is generated in the same language as the source file. It is compiled with the native C, C++ or Fortran compiler and linked to the NANOS++ library.

NANOS++ is the runtime environment of the OmpSs programming model. Its API includes functions for specifying tasks and their input and output data. When executing an OmpSs program, NANOS++ determines task dependencies based on task input and output data. Task instances that have their dependencies fulfilled are scheduled for execution on available threads, according to a pre-defined or user-defined scheduling policy.

LISTING 3.1: Task-based matrix-matrix multiplication in C

```
1  [sequential code]
2
3  for ( i = 0; i < DIM; i++) {
4    for ( j = 0; j < DIM; j++) {
5      for ( k = 0; k < DIM; k++) {
6  #pragma omp task in([BS][BS] A, [BS][BS] B) inout([BS][BS] C)
7        matmulTask(A[i][k], B[k][j], C[i][j]);
8      }
9    }
10 }
11 #pragma omp taskwait
12
13 [more sequential code]
```

Listing 3.1 shows a code fragment performing a task-based dense matrix-matrix multiplication. Note that this is a blocked implementation of matrix-matrix multiplication with a block size of BS. The function `matmulTask` sequentially multiplies two input sub-matrices $A$ and $B$ and stores the result in matrix $C$. The pragma on line 6 declares the function `matmulTask` as a task. Each call to `matmulTask` creates a task instance which reads the matrices $A[i \dots i + BS - 1[k \dots k + BS - 1]$ and $B[k \dots k + BS - 1][j \dots j + BS - 1]$, as indicated by the `in` statements of the pragram. The matrix $C$ is read and written, which is indicated by the `inout` statement. The statement in line 11 creates a global barrier, forcing all previously generated task instances to finish execution before proceeding with the sequential code.

## 3.2 Investigated Systems

### 3.2.1 Shared-Memory Multi-Core Systems

In Chapter 4, we analyze execution time predictability of task-based programs on four different shared-memory multi-core systems. Table 3.1 gives an overview of the characteristics of the four systems used for this evaluation. The first two platforms are high-end systems used in HPC environments, while the other two are based on low-power mobile systems-on-a-chip (SoCs). This selection of machines covers three of today's most widely-used ISAs: x86-64, POWER ISA, and ARMv7.

The first investigated system is a single node of the MareNostrum 3 HPC system. A node of MareNostrum 3 consists of two sockets in a ccNUMA configuration, each equipped with an Intel Xeon E5-2670 processor based on Intel's Sandy Bridge architecture. Each processor has 8 cores, running at 2.6GHz in normal mode and at 3.3GHz in Turbo Boost mode. Each core has 32KB private L1 data- and instruction cache and 256KB combined L2 cache. All cores of a socket share a 20MB L3 cache.

[1] DDR3L-1600 connected to a 750MHz interface

TABLE 3.1: Investigated machines

| Micro-arch. | Cores per socket | L1 size | L2 size | L3 size | Memory |
|---|---|---|---|---|---|
| Intel Sandy Bridge | 8 | 32KB+32KB per core | 256KB per core | 20MB shared | 32GB 64-bit DDR3-1600 |
| IBM POWER7 | 8 | 32KB+32KB per core | 256KB per core | 32MB shared | 64GB 64-bit DDR3-1600 |
| ARM Cortex-A15 MPCore | 2 | 32KB+32KB per core | 1MB shared | n/a | 2GB 32-bit DDR3L-1600 |
| ARM Cortex-A9 MPCore | 4 | 32KB+32KB per core | 1MB shared | n/a | 2GB 32-bit DDR3L-1600[1] |

Each socket disposes of 32GB DDR3-1600 DRAM. The system is capable of 2-way simultaneous multi-threading (SMT). However, in the MareNostrum 3 system SMT is deactivated.

Next, we investigate an IBM BladeCenter PS701 system, featuring an IBM POWER7 processor. The system features one CPU with 8 cores and can thus be classified as a UMA system. The system runs at 3GHz. The system has 32KB private instruction- and data caches, 256KB private L2 caches and a 32MB L3 cache and contains 64GB of DDR3-1600 DRAM as main memory. The POWER7 architecture supports 4-way simultaneous multi-threading. Although activated, we do not make use of this feature in the scope of this thesis.

Even though ARM microprocessors are not used in production HPC environments yet, there is an increasing interest in integrating ARM chips in future server and HPC machines [57, 94]. Therefore, we investigated two ARM systems with different performance characteristics. The first system is an Arndale Board with a dual-core ARM Cortex-A15 SoC. It features 32KB private instruction- and data cache per core and a shared L2 cache of 1MB. The second ARM-based system is an NVIDIA CARMA DEVKIT with an NVIDIA Tegra 3 SoC, featuring 4 ARM Cortex-A9 cores. Each core has 32 KB of private L1 instruction- and data cache. All cores share 1MB of L2 cache. Both system are connected to 2GB of DDR3L-1600 memory, and neither system supports SMT.

These four machines cover a wide range of performance levels as well as different ISAs, CPU, cache and memory technologies.

### 3.2.2 Hybrid Distributed Shared-Memory System

In Chapter 6, we present MUSA, our multi-level simulation approach for hybrid systems. We validate MUSA against MareNostrum 3. The characteristics of a single

node of MareNostrum 3 are listed in Tab. 3.1. MareNostrum 3 has three interconnect networks:

1. An InfiniBand FDR10 network for MPI communication
2. A 10Gb/s Ethernet network for file system access
3. A 10Gb/s Ethernet network for system management and maintenance

## 3.3 The TaskSim Multi-Core Simulator

In Chapter 5, we present TaskPoint, our sampled simulation methodology for task-based programs. We evaluate TaskPoint using the TaskSim simulator [99, 100]. We also use TaskSim to evaluate MUSA, our multi-level simulation approach for hybrid systems, presented in Chapter 6. TaskSim is a cycle-accurate, trace-driven performance simulator for multi-core architectures. It interfaces with an unmodified version of the OmpSs runtime system. The runtime system schedules the task instances of the simulated application for execution on the simulated processor cores.

The main difference between trace-based and execution-driven simulation is that a trace-based simulator needs to functionally execute a simulated application only once, whereas an execution-driven simulator functionally executes an application in each simulation. Representative applications can have a significant memory footprint, limiting the number of simulations which can be run simultaneously on a multi-core host. Trace-based simulation does not have this restriction. Once an application trace is generated, all simulations of this application, potentially with different architectural configurations, have a significantly lower memory footprint. In the scope of this work, we have frequently run 16 simulations simultaneously on simulation hosts with 16 cores.

TaskSim is designed for the exploration of large design spaces requiring large numbers of simulations, i.e. hundreds to thousands. Although TaskSim is a single-threaded simulator, design space explorations can be parallelized by executing multiple simulations in the same host in parallel. This is possible because, due to its trace-based design, TaskSim has low memory requirements. All simulations conducted in the scope of this work show memory footprints of less than 200MB.

Figure 3.1a illustrates the trace generation process required before conducting a simulation with TaskSim. First, the application to be simulated needs to be compiled and linked against the OmpSs runtime system. As illustrated in the figure, the application is executed twice. In the first execution, a runtime system plugin intercepts the application's calls to the runtime system and stores them in the trace. In the second execution, the application is executed with a dynamic binary instrumentation tool. Currently, TaskSim supports Intel's *PIN* tool [80]. Support for the open source

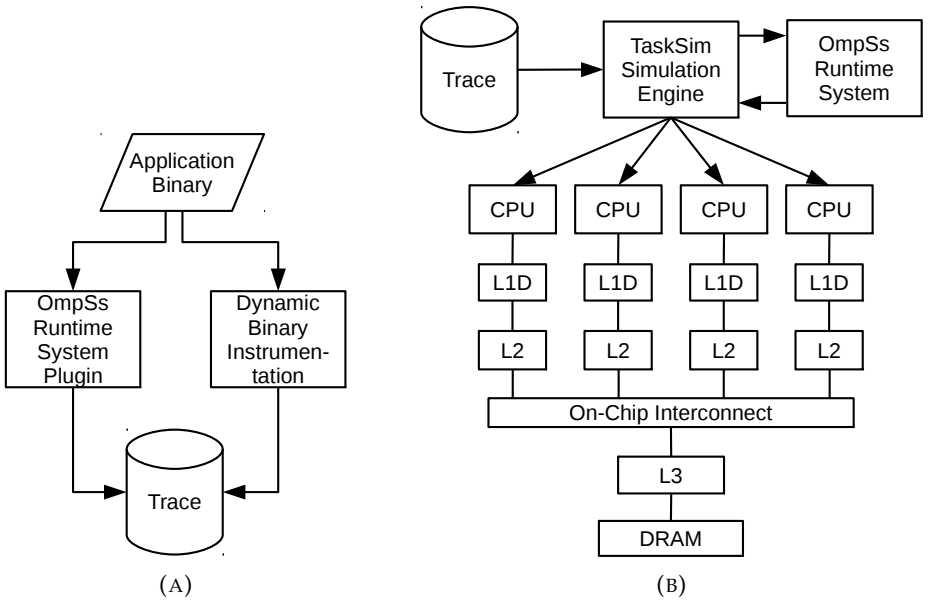



FIGURE 3.1: Overview of TaskSim simulation infrastructure: trace generation (A) and simulation (B)

*DynamoRIO* tool [18] is currently under development. All executed instructions are decoded using *PTLSim*'s x86 decoder [123]. Afterwards, the micro-instructions resulting from the decoding step are added to the trace.

Figure 3.1b shows how, during simulation, the TaskSim simulation engine reads the application trace and forwards the runtime system events to an unmodified instance of the OmpSs runtime system. The runtime system manages creation and scheduling of parallel work units and communicates scheduling decisions back to the simulation engine. According to these scheduling decisions, the simulated application's instructions are retrieved from the trace and processed by the simulated CPUs.

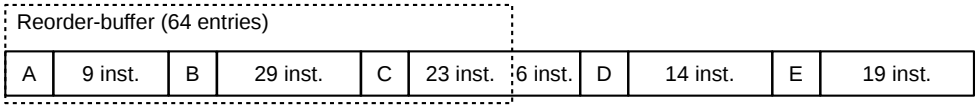



FIGURE 3.2: Illustration of Reorder-Buffer Occupancy Analysis: instructions streamed through 64-entry reorder-buffer. Original figure by Lee et al. [77].

TaskSim has a detailed and an abstract CPU model. The detailed CPU model is based on the *Reorder-Buffer Occupancy Analysis* technique proposed by Lee et al. [77]. Figure 3.2 illustrates an instruction stream flowing through a 64-entry reorder-buffer (ROB). The boxes labeled A through E indicate instructions accessing memory, whereas A is the oldest instruction in the ROB. Between two memory instructions, the figure indicates the number of other instructions of any type. In each cycle, the Reorder-buffer Occupancy Analysis model issues one or more instructions to the head of the

ROB, according to the issue width of the simulated processor. When the ROB is fully occupied, the issue stage is stalled.

When a memory instruction reaches the tail of the ROB, TaskSim creates a memory request and sends it to the CPU-side port of the corresponding core's L1 cache. Detailed timing models of caches, on-chip interconnect structures and DRAM simulate the path of the memory request through the memory hierarchy. Eventually, a response message arrives at the CPU model where the request originated. The memory instruction is committed and leaves the ROB. Afterwards, non-memory instructions are commited at the specified commit rate, until the next memory instruction is encountered or the simulation finishes.

In contrast, TaskSim's abstract CPU model only accounts for the duration, measured in cycles, of computation phases and calls to the runtime system. In the existing implementation, TaskSim reads a task instance's cycle count from the application trace. In Chapter 5 we extend TaskSim with a fast-forward mechanism capable of simulating execution at an arbitrary, user-defined IPC. Furthermore, we add support to switch between different simulation modes at runtime.

## 3.4 Benchmarks

In this section, we give an overview of the benchmarks used for the evaluation of the simulation methodologies developed in the scope of this thesis. First, we present the task-based benchmarks used in our evaluation of TaskPoint. Afterwards, we introduce the hybrid (MPI+OpenMP and MPI+OmpSs) benchmarks used for our evaluation of MUSA.

### 3.4.1 Task-based Benchmarks

In our evaluation of TaskPoint in Chapter 5, we investigate a set of 27 task-based parallel benchmarks implemented using the OmpSs programming model. The benchmarks and their key characteristics are listed in Tab. 3.2. They cover a broad range of algorithms widely used in scientific HPC applications and include programs with different compute-to-memory ratios, different memory access patterns and different amounts of parallelism and synchronization. Benchmarks 1 to 11 have been successfully used in previous works to evaluate HPC clusters [93, 94]. Benchmarks 12 to 16 are in-house implementations of algorithms frequently ocurring in scientific computing. Finally, benchmarks 17 to 27 are part of the PARSEC benchmark suite [13], which is widely used to evaluate the performance of parallel systems.

Whenever possible, we generate traces equivalent to at least ten seconds of single-threaded execution on a state-of-the-art machine. For the PARSEC benchmarks, we use the *simlarge* input sets. Table 3.2 lists the number of task types and task instances

and the time required for a detailed simulation of the entire benchmark for 1 and 64 simulated threads using the TaskSim simulator.

We classified the benchmarks according to whether they are compute-intensive or not. Because the working sets of all concurrently executing task instances fit into the last level cache, we considered the following benchmarks as compute-intensive: *2d-convolution*, *3d-stencil*, *atomic-monte-carlo-dynamics*, *merge-sort*, *dense-matrix-multiplication*, *fluidanimate* and *swaptions*.

We optimized compute-intensive benchmarks by adjusting the task working set to fit into the on-chip last-level cache. This is one of the most straightforward optimizations applied by programmers in blocked numerical algorithms. The most cache constrained configuration is the Cortex-A9 running with four threads. Therefore, we adjusted the task working set to fit into a quarter of the last-level cache in the Cortex-A9 chip. We use the same configuration for all platforms to have the same basis for comparison.

For the remaining benchmarks, we configure the task granularity for the resulting task instances to be at least 100,000 instructions long. By doing so, we ensure that the time spent in task execution is significantly larger than the time spent in performance measurement code or in the runtime environment. The number of task instances per application is adjusted to a large enough number so there is enough parallelism to use all threads at all times.

### 3.4.2 Hybrid MPI+OpenMP Benchmarks

The *NAS parallel benchmarks* [9] have been widely used to evaluate the performance of HPC systems. In this thesis, we use the *Multizone* versions [116] of the NAS parallel benchmarks *BT*, *SP* and *LU*, named *BT-MZ*, *SP-MZ* and *LU-MZ*, respectively. All three benchmarks compute the solution of the unsteady, compressible Navier-Stokes equations of a three-dimensional problem. To this end, the different benchmarks employ different mathematical solvers. The benchmarks perform multiple iterations, whereas the number of iterations depends on the input size. Each iteration represents a time step, at the end of which neighboring zones perform a boundary exchange.

All three NAS multi-zone benchmarks partition the global problem domain into blocks referred to as *zones*. Partitioning is done along the horizontal axes. *LU-MZ* and *SP-MZ* work with zones of equal size. In *BT-MZ* the sizes of adjacent zones approximately form a geometric series. In other words, moving along one of the horizontal axes, the distance between adjacent zone boundaries grows by an approximately constant factor. During execution, different zones are typically processed by MPI processes running on different cluster nodes. Each MPI process can further exploit parallelism by relying on OpenMP for intra-node parallelization.

TABLE 3.2: Task-based parallel benchmarks used for the evaluation of TaskPoint

| # | Benchmark | # Task types | # Task instances | Simulation time $[h:min]$ | | Properties |
|---|---|---|---|---|---|---|
| | | | | 1 Thread | 64 Threads | |
| 1 | 2d-convolution | 1 | 16384 | 31:37 | 59:34 | Kernel: strided memory accesses |
| 2 | 3d-stencil | 1 | 16370 | 9:12 | 40:51 | Kernel: strided memory accesses |
| 3 | atomic-monte-carlo-dynamics | 1 | 16384 | 8:38 | 15:16 | Kernel: embarrassingly parallel |
| 4 | dense-matrix-multiplication | 1 | 17576 | 70:14 | 127:10 | Kernel: high data reuse, compute bound |
| 5 | fft | 8 | 25024 | 31:57 | 110:47 | Kernel: variable stride memory accesses |
| 6 | histogram | 1 | 16384 | 6:02 | 12:13 | Kernel: atomic operations |
| 7 | merge-sort | 4 | 20480 | 12:39 | 33:23 | Kernel: recursive task instantiation |
| 8 | n-body | 2 | 25000 | 8:15 | 12:31 | Kernel: irregular memory accesses |
| 9 | reduction | 2 | 16384 | 1:51 | 5:15 | Kernel: parallelism decreases over time |
| 10 | sparse-matrix-vector-multiplication | 1 | 1024 | 0:33 | 1:26 | Kernel: load imbalance, memory bound |
| 11 | vector-operation | 1 | 16400 | 24:25 | 191:00 | Kernel: regular, memory bound |
| 12 | sparseLU | 11 | 22058 | 7:25 | 17:17 | Decomposition of large, sparse matrices |
| 13 | cholesky | 4 | 19600 | 33:42 | 59:29 | Decomposition of Hermitian positive-definite matrices |
| 14 | jacobi | 9 | 20480 | 19:07 | 19:54 | Jacobi iterative method |
| 15 | kmeans | 6 | 16337 | 75:21 | 141:02 | Clustering based on Lloyd's algorithm |
| 16 | knn | 2 | 18400 | 31:28 | 65:27 | Instance-based machine learning algorithm |
| 17 | blackscholes | 2 | 24500 | 8:42 | 17:19 | Option price calculation |
| 18 | bodytrack | 7 | 21439 | 15:24 | 31:28 | Human body tracking with multiple cameras |
| 19 | canneal | 1 | 16384 | 11:13 | 29:38 | Cache-aware simulated annealing |
| 20 | dedup | 4 | 15738 | 10:08 | 23:32 | Deduplication: combination of global and local compression |
| 21 | facesim | 12 | 20086 | 15:13 | 22:15 | Physical modeling of human face |
| 22 | ferret | 6 | 12288 | 58:34 | 115:22 | Image similarity search |
| 23 | fluidanimate | 9 | 8225 | 25:15 | 46:03 | Simulation of incompressible fluids |
| 24 | freqmine | 7 | 1932 | 23:52 | 34:13 | Frequent Pattern Growth method for Frequent Item Mining |
| 25 | streamcluster | 10 | 14656 | 21:09 | 37:41 | Online clustering algorithm |
| 26 | swaptions | 1 | 16384 | 29:27 | 70:25 | Monte-Carlo simulation to calculate swaption prices |
| 27 | x264 | 3 | 383 | 121:47 | 122:25 | Video compression according to H.264 standard |

Besides the NAS multi-zone benchmarks, we use the *HYDRO* and *SPECFEM3D* proxy applications. In the following, we summarize the key properties of all hybrid benchmarks used in this thesis:

- **BT-MZ** employs a block tridiagonal solver based on Gaussian elimination. Due to the irregular spacing between zones, the total size of the largest zone is approximately 20 times larger than the size of the smallest zone, resulting in different amounts of work assigned to different MPI processes. This makes it difficult to achieve good load balance and, thus, high parallel efficiency.
- **SP-MZ** decomposes the problem domain into equally-sized zones, resulting in approximately the same amount of work per MPI process. The number of zones increases with the input size. This makes it easier to balance load across different MPI processes. *SP-MZ* uses a scalar pentadiagonal solver.
- **LU-MZ** uses a *lower-upper symmetric Gauss-Seidel* solver [122]. In contrast to *BT-MZ* and *SP-MZ*, the number of zones in *LU-MZ* is limited to 16. Therefore, in order to scale to a large number of processors, *LU-MZ* needs to rely on intra-node shared-memory parallelism.
- The **HYDRO** benchmark [75] is a proxy application based on the *RAMSES* application [112]. *RAMSES* uses techniques from computational fluid dynamics to model galaxy formation. *HYDRO* captures the key performance characteristics of *RAMSES*, but at significantly less code complexity. *RAMSES* employs adaptive mesh refinement to rebalance the computation as the mass distribution in the simulated universe evolves. *HYDRO*, on the other hand, assumes a fixed cartesian mesh.
- **SPECFEM3D** [72] is an application for modeling seismic wave propagation. *SPECFEM3D* uses the continuous Galerkin spectral-element method to simulate forward and adjoint seismic wave propagation on arbitrary unstructured hexahedral meshes.

## 3.5 Performance Measurement in Native Execution

In Chapter 4, we investigate the performance predictability of task-based programs. We show, that performance predictability is related to performance regularity. We measure performance regularity using hardware performance counters.

### 3.5.1 Hardware Performance Counters

Modern processors include dedicated hardware for counting performance-related events occurring in the processor. This hardware is referred to as the *performance*

*monitoring unit* (PMU). The PMU can be configured by the user to count a variety of events, e.g. the number of executed instructions, elapsed CPU cycles, or hits and misses in the different cache levels or the branch predictor. The events monitored by the PMU are accessible via a set of registers. The PMU registers can be accessed only in privileged mode. Starting with kernel version 2.6.31, Linux includes kernel support in order to access the PMU from user-space via system calls.

The exact set of observable PMU events and the number of simultaneously available counter registers depend on the processor model. The PMU used in Intel's Sandy Bridge architecture features 11 counter registers, whereas the number of counters on the ARM Cortex-A9 processor is limited to 6. Thus, out of the hundreds of available PMU events on a modern processor the aforementioned platforms can simultaneously monitor up to 11 or 6, respectively.

Accessing the PMU via system calls is a tedious process. Since performance measurement code depends on the ISA and the exact processor model, it is not portable. The *Performance Application Programming Interface* (PAPI) library [17] provides a layer of abstraction decoupling performance measurement code from architectural implementation details. PAPI defines a set of events, many of which exist on all modern processors. As a result, performance measurement code can interface with PAPI in a consistent, architecture independent way, as long as the underlying architecture supports the measured events and provides enough counter registers.

### 3.5.2 Performance Measurement of Task-Based Programs

In Chapter 4, we investigate performance regularity of task-based programs in native execution. We measure cycle count, instruction count and numbers of L1 (data), L2 (data) and L3 cache misses using hardware performance counters. To this end we use the Mercurium compiler, which automatically inserts calls to a low-overhead instrumentation library at the beginning and the end of each task instance. Internally, this library interfaces to the performance counter subsystem via the PAPI library.

In OmpSs, a task instance can be suspended before it finishes execution. In particular, when a task instance executes a call to the runtime system, it is not guaranteed that control is immediately returned to the calling task instance. Instead, the runtime system can schedule another task instance for execution first, and at return to the original task instance in the future. The instrumentation library used in this work takes this into account by maintaining per-task-instance statistics.

# Chapter 4

# Execution Time Predictability of Task-Based Programs

## 4.1 Introduction

Multi-core systems are integrating an increasing number of processor cores on a single chip. This makes it difficult for programmers to exploit the available on-chip thread-level parallelism.

Task-based programming models allow the programmer to specify program parts as so-called *tasks*. Tasks may execute concurrently and are typically instantiated many times during execution. A runtime environment dynamically maps task instances to threads. The intuitive program partitioning improves programmability. At the same time, dynamic task scheduling reduces the inherent synchronization costs of other shared memory programming models thanks to a better load balancing [3].

The fact that all instances of the same task type consist of the same static code suggests that they should exhibit similar performance and execution time and, therefore, execution time should be predictable. In this chapter, we investigate the execution time predictability of task-based programs based on performance regularity. We carry out a performance analysis on four different state-of-the-art multi-core machines. Two machines are based on ARM Cortex-A9 MPCore and Cortex-A15 MPCore mobile CPUs. The other two are based on high-end Intel Sandy Bridge and IBM POWER7 CPUs, respectively. This allows us to investigate if performance regularity depends on the architecture. We expect performance variability to increase when increasing the number of execution threads competing for shared resources. To this end, we analyze performance variability on a per-task-instance basis for thread counts ranging from one up to the number of cores on each machine. We reach similar conclusions for the different machines, but find that architectures with more aggressive performance optimizations show a higher performance variability.

We identify three sources of variability across instances of the same task type: (i) input dependence, (ii) multiple classes of behavior, and (iii) contention on accessing

shared resources. For programs suffering from resource contention, we investigate how sharing decreases performance and increases performance variability. We also present a model based on linear interpolation to predict execution time of input dependent task types. Furthermore, we use a clustering algorithm to identify different behaviors in the same task type. Using our interpolation model and clustering algorithm, we dramatically increase the accuracy of execution time prediction. Prediction errors over 80% are reduced to less than 12% for input dependent cases and less than 2% on the presence of multiple behaviors.

In this chapter, we make the following contributions:

- An analysis of performance variability across instances of the same task type in task-based programs executing on multi-core systems. This analysis shows the variability on an instance-by-instance basis.
- A classification of the different sources of execution time variability on instances of the same task type.
- A low-complexity model based on linear interpolation for predicting the execution time of a task instance as a function of its instruction count.
- The use of a clustering algorithm to identify different classes of behavior in the same task type. In our example, we successfully classify task instances into clusters, each of which exhibits regular performance.

## 4.2 Execution Time Predictability of Task-Based Programs

Many parallel implementations of numerical algorithms decompose the problem domain into sub-domains called *blocks* or *tiles*. In task-based programming models the programmer specifies parts of a program as work units called *tasks*, each one to perform a different operation. A task is usually instantiated many times, each instance performing the common operation of the task on a separate block or tile. Task instances can be scheduled to threads whenever they have their dependencies satisfied. Typically, a thread executes many task instances before reaching a synchronization point. Task-based programming models are a programming paradigm relying on the exploitation of functional- and data parallelism. For background on different types of parallelism we would like to refer to Chapter 2.2.3. Background on parallel programming of shared-memory systems is provided in Chapter 2.1.1.

The fact that instances of the same task type consist of the same code leads us to the assumption that they consist of similar numbers of instructions, exhibit similar performance and therefore their execution time is predictable. However, this assumption turns out to be wrong in some cases. Figure 4.1 shows the total execution time prediction error for a set of task based programs, assuming the time of

the first or the second executed instance for all instances of a task type. The error is calculated according to Equation 4.1, with T the set of task instances of the same task type, $C_{Sample}$ the cycle count of the sample task instance and $C_i$ the cycle count of task instance $i$. We only investigate time spent in task execution and ignore operating system and runtime system overheads.

$$Err = \left(1 - \left|\frac{\sum_{i \in T} C_{Sample}}{\sum_{i \in T} C_i}\right|\right) \cdot 100\% \tag{4.1}$$

Before conducting our detailed analysis, we envision three potential sources of performance variability that potentially degrade performance predictability:

- *Input dependence*: The behavior of a task instance depends on the task instance's input data. An example is sparse algorithms, in which task instances perform different amounts of computation or exhibit different memory access patterns, due to the nature of the sparse input data.
- *Several types of behavior per task type*: Task instances of the same type perform one out of several possible types of computation. An example is recursive algorithms, in which some task instances create more child tasks, while others perform the actual computation when the recursion terminates.
- *Contention on shared resources*: Multiple threads interfere with each other when accessing shared system resources. Different instances of the same task type may suffer from different degrees of interference caused by other threads running in the system and accessing shared resources. This includes shared caches, interconnect structures and memory bandwidth.

## 4.3 Evaluation

The results of the experiments conducted in the scope of this chapter show that, despite the obvious intuition, performance can be irregular across instances of the same task type. This directly affects execution time prediction (shown in Figure 4.1). In this section, we first show the results of our performance analysis on a per-task-instance basis. Afterwards, we present a case of input dependent task behavior and present a model to estimate the execution time of a task instance as a function of its instruction count. We also show a case of multiple classes of behavior within a single task type. We use a clustering technique to distinguish these different classes of behavior and improve execution time predictability. Finally, we explain how resource sharing affects performance regularity and analyze contention on different shared resources in the memory hierarchy.

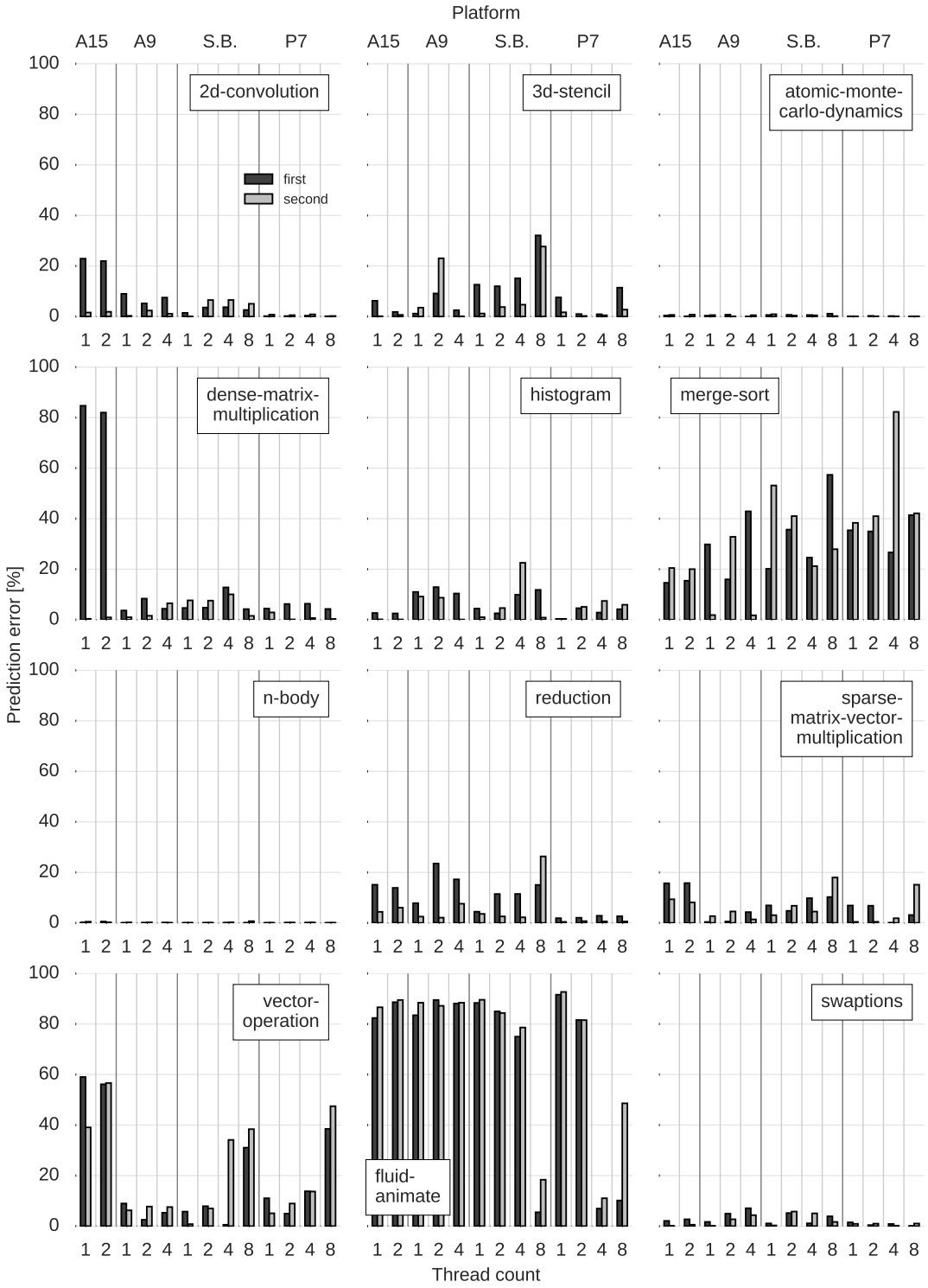



FIGURE 4.1: Percent error when assuming the execution time of the first / second executed task instance for all task instances to predict total execution time. Results shown for four different machines (see Tab. 3.1) and different thread counts.

### 4.3.1 Per-Task-Instance Performance Analysis

Figure 4.2 shows boxplots of the measured instructions per cycle (IPC) per task type. Each chart corresponds to one task type and shows the measured results on four different platforms. Only one thread per core is executed in each experiment, which limits the configurations to two threads (Cortex-A15), four threads (Cortex-A9), and eight threads (Intel Sandy Bridge and IBM POWER7). The solid box contains the interquartile range of the measured IPC values of all instances of the respective task type, i.e., 50% of the observations are within this range. The horizontal line within the box indicates the median. The whiskers extend from the 5th to the 95th percentile. The lower and upper 5% of the measured IPC values are treated as outliers and are not shown in the plot.

Most of the investigated benchmarks only have one task type, whereas *merge-sort*, *n-body* and *reduction* have two and *fluidanimate* has eight. The different task types of *fluidanimate* show similar performance variability. Therefore, we limit our evaluations to the task type `ComputeForcesMT`, which accounts for 40% of *fluidanimate*'s total instruction count.

In our results, we observe two general classes of behavior. The first class consists of benchmarks for which IPC does not significantly degrade when increasing the number of execution threads. This behaviour is exposed by the benchmarks *2d-convolution*, *atomic-monte-carlo-dynamics*, *merge-sort* (both task types), *n-body* (both task types), *reduction* (both task types), *fluidanimate* (all task types) and *swaptions*. We make the important observation that *2d-convolution*, *atomic-monte-carlo-dynamics* and *n-body* (task type 1) present a nearly constant IPC with very low variability. This behavior is persistent across the different platforms.

The second class of behavior consists of the benchmarks, for which IPC degrades when increasing the number of execution threads. This phenomenon is known as work time inflation [88]. In our benchmark suite, this behavior is exposed by the benchmarks *3d-stencil*, *histogram*, *sparse-matrix-vector-multipli-cation* and *vector-operation*. For these benchmarks, besides work time inflation, we also observe an increasing performance variability. Note that the variability shown in Figure 4.2 directly relates to the prediction error shown in Figure 4.1.

### 4.3.2 Predictability of Irregular Behavior

In this subsection, we identify three sources of irregular behavior, namely input dependence, multiple classes of behavior per task type and resource sharing. We predict execution time of task types with input dependent behavior using an interpolation-based model. For task types with several classes of behavior we use a clustering

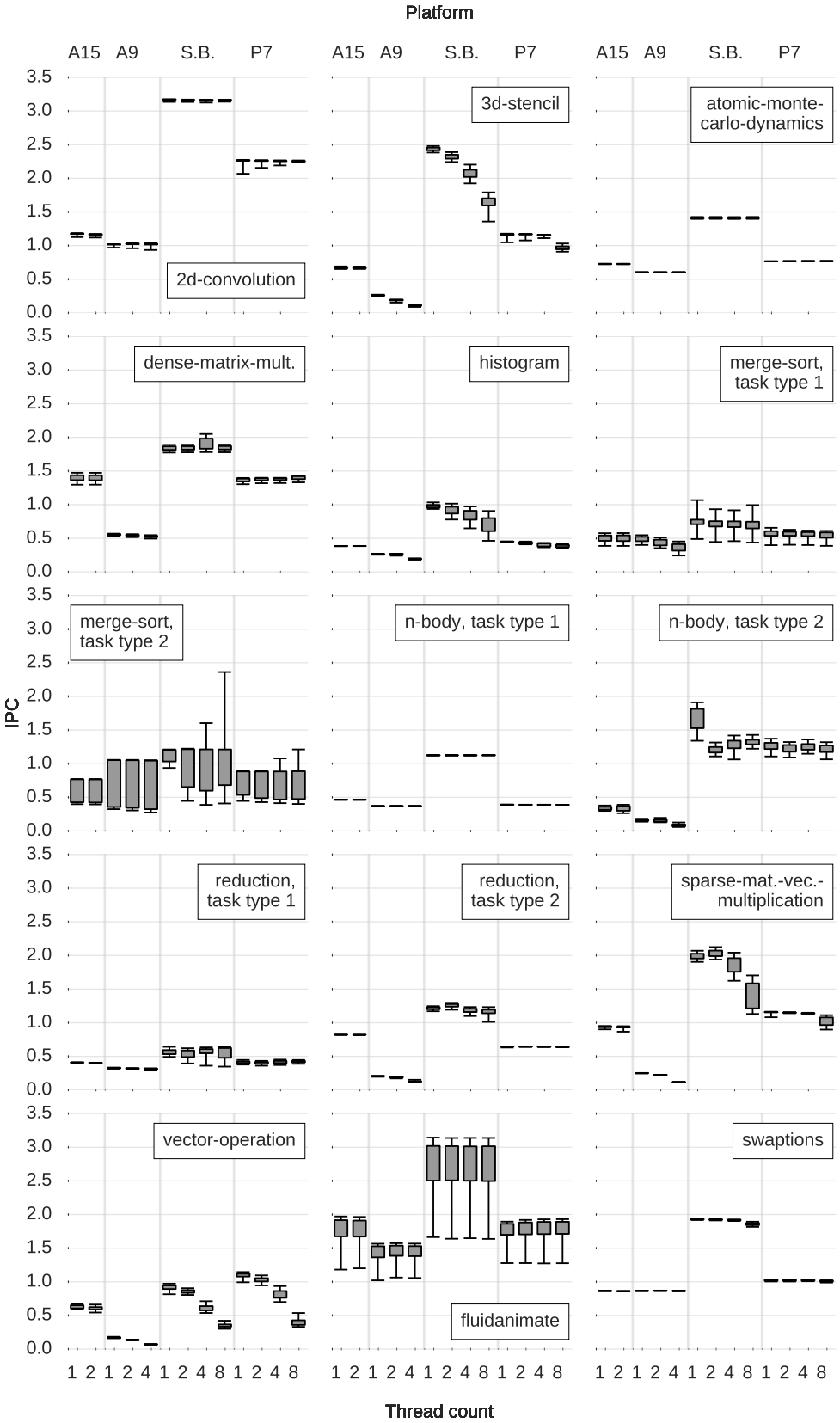



FIGURE 4.2: IPC variation per task type on four different platforms (ARM Cortex-A9 and A15, Intel Sandy Bridge and IBM POWER7)

algorithm to detect clusters of similar behavior and predict execution time on a per-cluster basis. Finally, we analyze the impact of resource sharing on performance predictability.

**Input Dependence:**

Input dependence is the dependence of the control flow of a task instance on the input data. Figure 4.3 shows heatmaps of the programs *fluidanimate* and *merge-sort*. Heatmaps are a graphical representation of a histogram of two independent variables. Both the horizontal and vertical axes are split into bins. For each combination of horizontal and vertical bin, colours indicate how many task instances have a certain instruction count and a certain IPC.

In the case of *fluidanimate*, the instruction count of task instances varies between 1 million and 70 million instructions, while IPC tends to be higher for higher instruction counts. This results in different numbers of execution cycles. Assuming the same cycle count for all task instances leads to the prediction error shown in Figure 4.1 which reaches over 80%. The instruction count and IPC variation is caused by the fact that all task instances perform an index computation that is highly inefficient for high indexes. We want to emphasize that this index computation is part of the default implementation of the *fluidanimate* benchmark and is not caused by porting the benchmark to the OmpSs programming model.

For the programs *fluidanimate* and *merge-sort* (task type 1), we apply a sampling-based model to predict execution time as a function of instruction count for all task instances. This model assumes that the instruction count of each task instance is known apriori and works as follows. First, we add instruction count and execution time of the first executed task instance to the (empty) set of support points. Afterwards, for each encountered task instance we check if its instruction count is less than $90\%$ of the smallest or greater than $110\%$ of the largest instruction count in the set of support points. If this is the case, we add it to the set of support points. Otherwise, we predict the execution time by linear interpolation within the set of support points or by constant extrapolation in the range outside the support points. Figure 4.4 shows that the error of the total execution time prediction based on this model stays below 12% for all configurations on the Intel Sandy Bridge system.

**Multiple Behaviors Per Task Type:**

For *merge-sort* (task type 2) we observe two clusters in the heatmap plot, indicating two different behaviors. Strictly speaking, this is also a case of input dependence. However, the difference to the type of input dependence covered in the previous section is that there are multiple, clearly distinct classes of behavior. This is caused

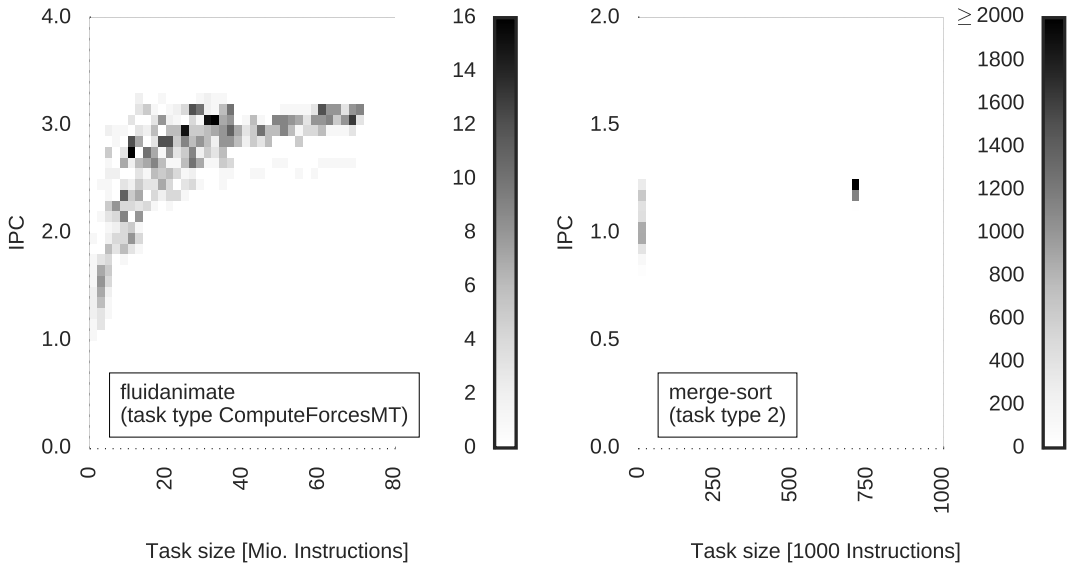



FIGURE 4.3: Instruction count vs. IPC histogram of benchmarks *fluidanimate* (task type ComputeForcesMT) and *merge-sort* (task type 2)

by the recursive implementation of the merge-sort algorithm. A task instance either creates two child instances, resulting in the cluster on the left, or it performs a sorting operation, resulting in the cluster on the right. Predicting execution time based on the assumption of regular execution time and IPC leads to the error shown in Figure 4.1.

For the aforementioned case, we perform a k-means clustering of all task instances into two clusters, according to their instruction count. For each resulting cluster, we determine the centroid and chose the task instance closest to the centroid as a representative of the respective cluster. Finally, we estimate the total execution time of each cluster by multiplying the execution time of the representative by the number of task instances in the cluster. Figure 4.4 shows, that the error of the total execution time prediction based on this method is smaller than 2% for all configurations on the Intel Sandy Bridge system.

**Resource Sharing:**

The third source of irregular behavior we identified is resource sharing. In the following, we present four examples of resource sharing. These examples have in common that contention on shared resources affects the performance of task instances of the same task type to a different extent. This increases performance variability and thus decreases performance predictability. Figure 4.5 shows boxplots of L2 data cache and L3 cache misses per 1000 executed instructions (misses per kilo-instruction, MPKI) of the benchmarks for which we observe a decrease of IPC for increasing thread counts. The measured number of L3 cache misses includes misses

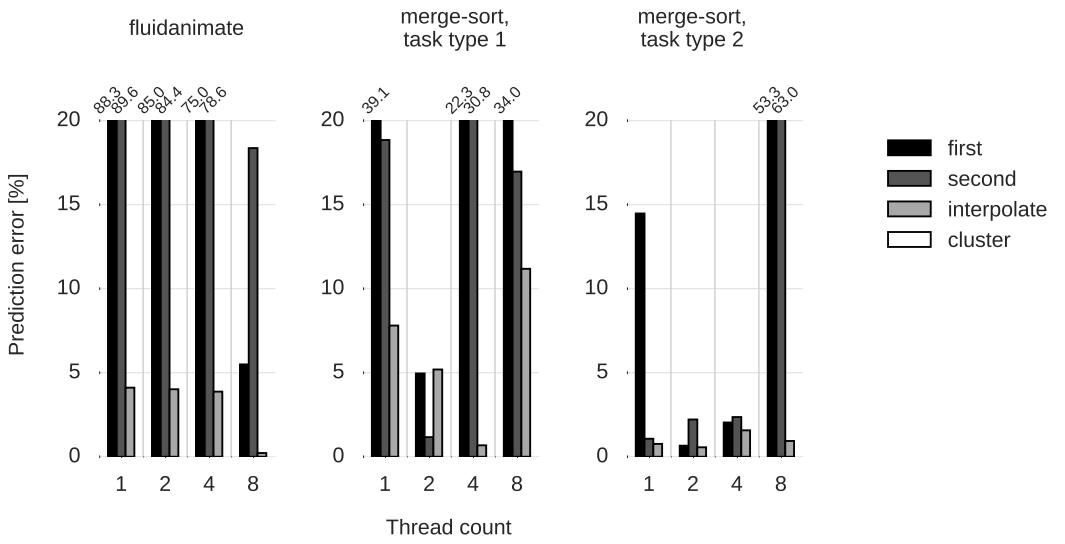



FIGURE 4.4: Execution time prediction error using interpolation model (*fluidanimate* and *merge-sort*, task type 1) and clustering (*merge-sort*, task type 2)

caused by L2 data cache misses, due to the limitations of the available hardware performance counters.

For *3d-stencil*, we observe an increase of L2 MPKI when increasing the number of threads. However, L3 MPKI stays nearly constantly low. Our theory is that the increased L2 MPKI is caused by invalidations of data residing in the private L2 caches by other threads.

The *histogram* benchmark shows not only an increase of L2 MPKI for increasing thread counts, but also an increase in L2 MPKI variability. For increasing thread counts, there might be several threads competing to execute an atomic operation, resulting in higher contention. Furthermore, the execution of the atomic operation itself can invalidate data in other threads' private caches.

In case of *sparse-matrix-vector-multiplication*, L2 MPKI and L3 MPKI are nearly constant for increasing thread counts. Since in this benchmark there is no data sharing between different task instances, the decrease in IPC has to occur due to the limited capacity of shared resources, e.g. memory bandwidth or cache bandwidth.

For *vector-operation*, we observe a decrease of L2 MPKI when increasing the number of execution threads. As memory bandwidth saturates for increasing thread counts, threads progress at a slower rate and thus cause less demand misses in the L2 cache.

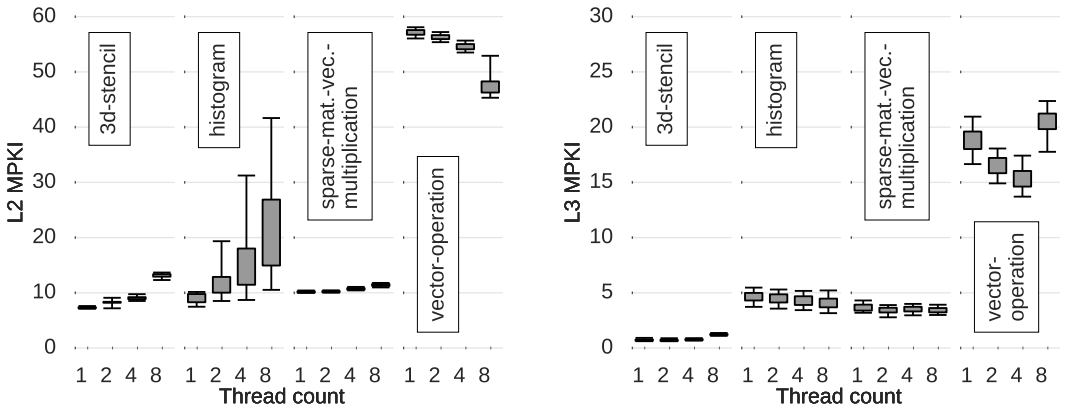



FIGURE 4.5: L2 data and L3 cache misses per 1000 instructions (MPKI) for *3d-stencil*, *histogram*, *sparse-matrix-vector-multiplication* and *vector-operation*, executed on Intel Sandy Bridge with 1, 2, 4 and 8 threads

## 4.4 Related Work

To the best of our knowledge, this is the first analysis of execution time predictability of task-based programs. However, there are other performance analyses of task-based programs focusing on other aspects.

Duran et al. [41] present a benchmark suite consisting of task-based OpenMP programs. They give examples for different kinds of performance analyses of these benchmarks. They evaluate total execution time as a function of various parameters such as processor count and task creation cut-off parameters. Other works [98, 105] investigate task granularity and task creation cost as performance-limiting factors in task-based programs. However, these works neither analyze performance on a per-task-instance basis nor task execution time predictability.

There are other works that use analytical models to predict execution time [48, 67, 87]. These works use mathematical models to compute the delays of certain events during execution. Most past works compute delays for events at the instruction-level, such as instruction issue and commit, branch mispredictions and cache misses. Our model works at a coarser granularity by computing the delay of whole individual task instances.

Performance predictability of parallel applications on large HPC systems has been explored from many perspectives. Some approaches combine the efficiency of analytical models with the accuracy of simulation to generate accurate and fast performance predictions [108]. Other approaches [68] explore performance predictability by developing application-specific performance models, which are formulated from an analysis of the code, inspection of key data structures, and analysis of traces gathered at runtime. While this methodology provides fast and accurate predictions, it is application specific and it requires a deep understanding of the scientific

codes. These works target MPI applications, while the work in this chapter focuses on shared-memory task-based programs.

## 4.5 Summary

The analysis in this chapter shows that the naive assumption of regular performance across instances of the same task type is not always valid. However, we show that accurate performance predictions can be derived from detailed performance information of a relatively small number of task instances.

We present techniques to improve the accuracy of execution time predictions for task types with irregular performance. These techniques are based on linear interpolation and clustering. The execution time prediction error is reduced from more than 80% to less than 12% for input dependent cases and to less than 2% for task types exposing multiple classes of behavior. Further research is needed to improve execution time predictability of task-based programs experiencing contention on shared resources.

In Chapter 5 we leverage the insights from this chapter and present TaskPoint, a sampled simulation methodology for task-based programs executed on multicore systems. We envision another potential application in the field of dynamic task scheduling: apriori-knowledge of the execution time of a task instance would allow for new, smart scheduling techniques.

# Chapter 5

# Sampled Simulation of Task-Based Programs

## 5.1 Introduction

Computer architecture research heavily relies on simulation. Increasing design complexity and increasing core counts in modern multi-core processors present new challenges to architectural simulation. First, simulating a more complex design requires more time for a given workload. Second, the more complex a design, the larger the simulated workload needs to be in order to meaningfully stress the design.

One technique to reduce simulation time is sampling. Sampled simulation reduces simulation time by only simulating a fraction of a workload. Sampling is a well-established technique for simulation of single-threaded architectures. The prevalent techniques perform detailed simulation of either only the representative program parts identified in profiling [107] or switch periodically between detailed and fast-forwarding mode in time-based sampling [121].

While sampled simulation is a well-established technique for single-threaded architectures, techniques targeting multi-threaded architectures have only been recently proposed. The main challenge in sampling multi-threaded simulations is to ensure that at the beginning of each detailed simulation interval all threads have made the same amount of progress as in a full detailed simulation. A technique proposed by Carlson et al. [24] achieves this by selecting a periodic sampling interval during offline profiling and, during simulation, estimating the rate at which to fast-forward each thread between intervals of detailed simulation. Carlson et al. [22] also propose a technique based on the insight that after a global barrier all threads are synchronized and resume execution simultaneously. The technique leverages the inter-barrier regions in barrier synchronized programs as sampling units.

Task-based programming models have been proposed to reduce load imbalance and thus increase parallel efficiency of future large-scale multi-core machines [79]. A task-based programming model allows the programmer to specify program parts

as *tasks* and to specify dependencies between those tasks. Tasks are typically instantiated many times during the execution of a program. *Over-decomposition* ensures that there are many more task instances than there are execution threads. The over-decomposition of a parallel program into tasks, together with dynamic scheduling of task instances to threads, dynamically balances the amount of work assigned to each thread. Inter-task dependencies enforce synchronization only when necessary. The lack of global barriers and the dynamically scheduled execution of task-based programs make them unsuitable for existing sampled simulation techniques.

In this work we present TaskPoint, a sampled simulation methodology for dynamically scheduled task-based programs executed on shared memory multi-core machines. TaskPoint leverages task instances as sampling units and only simulates a small number of them in detail. The remaining task instances are simulated in a faster simulation mode, ensuring that progress in different threads is modelled correctly.

In this chapter, we make the following contributions:

- We compare the performance variation of task-based programs in native execution and architectural simulation. This motivates the design of our TaskPoint methodology, its sampling policies and its fast-forwarding methodology.

- We present TaskPoint, a sampled simulation technique for multi-core architectures programmed with a dynamically scheduled, task-based programming model. In this context, we introduce two sampling policies, *periodic sampling* and *lazy sampling*. Lazy sampling simulates task instances in detail based on their type while periodic sampling considers their type and distribution over time.

- We propose a mechanism to accurately fast-forward an architectural simulation of a task-based program. During fast-forward, we model the performance of a given task instance based on previous instances of the same task type. We account for different task input sizes across the application execution by factoring in the number of instructions of the given task instance accordingly.

- We employ basic-block vectors (BBVs) and clustering to identify classes of behavior among task instances of an application. We show, how we (i) identify multiple classes of behavior among task instances of the same task type and (ii) merge task instances with similar behavior belonging to different types.

- We use an analytical performance model to improve simulation accuracy during simulation in fast-forward mode. Our approach combines the speed of analytical models with the accuracy of detailed simulation.

- We evaluate TaskPoint simulating 27 task-based parallel benchmarks, including the PARSEC benchmark suite. We evaluate the sensitivity of TaskPoint to different architectures by testing different numbers of simulated threads on two different configurations covering the opposite extremes of the multi-core design space: high-performance and low power.

The remainder of this chapter is organized as follows. In Section 5.2, we provide background and motivation of our work. In Section 5.3, we present our TaskPoint methodology. We evaluate TaskPoint in Section 5.4. Finally, we conclude in Section 5.5.

## 5.2 Background and Motivation

This section provides background on task-based programming models. We then motivate our work with an analysis of performance variation in native execution of 27 task-based parallel benchmarks.

### 5.2.1 Parallel Programming Models

In traditional parallel programming models for shared memory systems, like *POSIX Threads* [19], the programmer explicitly decomposes an application into concurrent instruction streams and manages synchronization between those. These instruction streams are processed simultaneously by different threads. A common problem with multi-threaded programs is load imbalance. Load imbalance occurs when different threads reach a synchronization point at different points in time.

Task-based programming models have the potential to alleviate load imbalance and thus increase parallel efficiency. When implementing a parallel program using a task-based programming model, the programmer specifies program parts as *tasks* and, optionally, data dependencies between these tasks. Tasks are instantiated many times during the execution of a program, resulting in a large number of task instances, each of which operates on different data. A runtime environment dynamically schedules task instances to execution threads.

Due to a fine-grained *over-decomposition* of the application, there are ideally more task instances ready for execution than there are threads. This allows the runtime environment to dynamically balance the workload assigned to each thread [79]. Further optimizations are possible if the architecture interfaces directly with the runtime environment [30, 114].

In this work, we differentiate between *task types* and *task instances*. Every execution of a task declaration statement at runtime results in the creation of a task instance. All task instances resulting from the same task declaration statement in

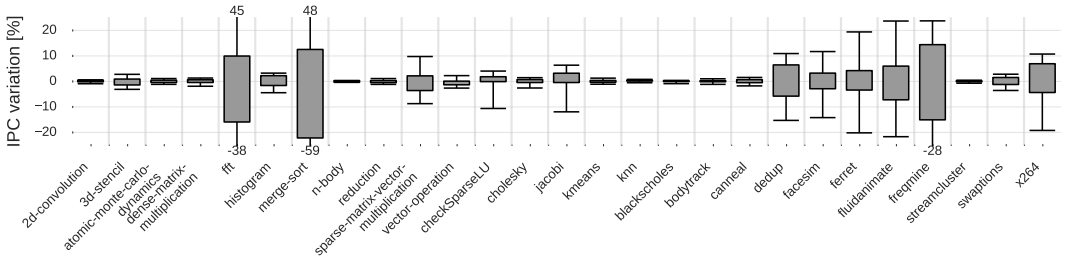



FIGURE 5.1: IPC variation across all task instances for native execution with 8 threads, normalized per task type

the source code are said to be of the same task type. In a typical task-based program, the number of task types is small, whereas the number of task instances lies in the order of thousands.

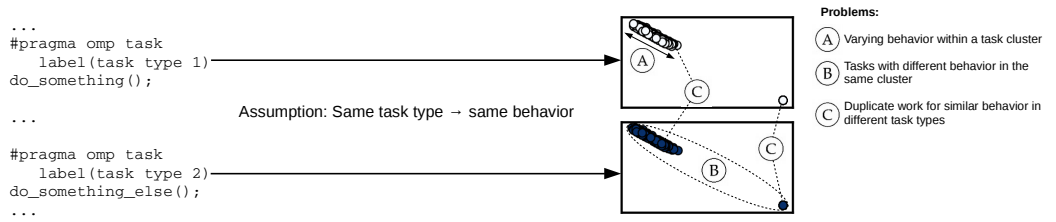
### 5.2.2 Performance Variation of Task-Based Programs

In order to motivate TaskPoint, our sampled simulation technique for task-based parallel programs, we analyze performance variation in native execution of 27 benchmarks. The investigated benchmarks are introduced in Section 3.4.1.

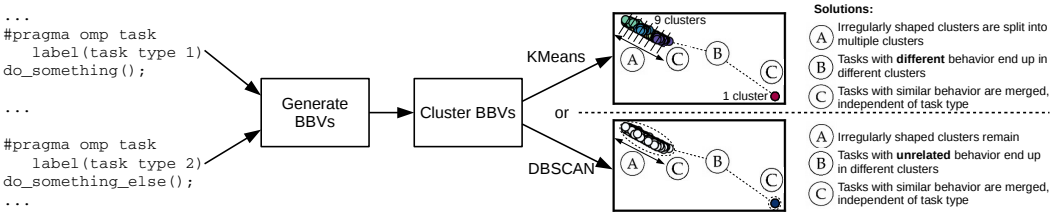
Different benchmarks, and even different task types of the same benchmark, generally show different average instructions per cycle (IPC). For an easy comparison of performance variation across benchmarks, we normalize the IPC of all task instances to the average IPC of their respective task type. For each benchmark, we use one box plot of these normalized IPC values to visualize performance variation across task instances.

Figure 5.1 shows IPC variation across task instances observed in a native execution with 8 threads on a system with an Intel SandyBridge-EP E5-2670 CPU running at 2.6 GHz and 128 GB of DDR3-1600 as main memory. The solid box of each box plot indicates the range from the first to the third quartile of the normalized IPC values, while the whiskers extend from the fifth to the 95th percentile. IPC values of task instances below the fifth and above the 95th percentile are treated as outliers. The Figure shows that for 16 out of 27 benchmarks performance variation lies within $\pm 5\%$.

We motivate TaskPoint based on the insight that performance of task-based programs is, in many cases, regular across instances of the same task type. For the remaining cases, our improved version of TaskPoint automatically detects classes of task instances with similar behavior using basic-block vectors and clustering. A potential simulation error is compensated with a correction factor derived from performance predictions obtained from an analytical performance model.




(A) TaskPoint without analytical modeling. Task instances of the same type are assumed to have similar behavior.




(B) TaskPoint with analytical modeling. Classes of behavior are identified using clustering.

FIGURE 5.2: Overview of original and improved TaskPoint methodology. The improved methodology uses BBVs to determine classes of similar behavior.

### 5.2.3 Identifying Representative Task Instances

Our analysis of performance regularity on a per-task-type basis in Figure 5.1 shows that, for many applications, task instances of the same task type behave similarly in terms of performance. Therefore, it is reasonable to assume that, in those cases, instances of the same task type can serve as performance samples for one another. However, the figure also shows that some benchmarks expose a significant performance variation among task instances. Examples are *merge-sort*, *fft*, *freqmine* and *dedup*. These applications require more sophisticated techniques to identify classes of task instances which can serve as samples for one another.

Basic-block vectors (BBVs) [107] have been used in the past to characterize phases of a workload and identify representative workload regions. A BBV is a vector with as many dimensions as there are static basic blocks in the simulated application. Each dimension contains the number of executed dynamic instructions of the corresponding basic block during a certain time interval. In this work, we determine one BBV per executed task instance.

Figure 5.2a illustrates our original TaskPoint methodology. The figure shows the BBVs of two task types of *merge-sort*. We apply random projection to two dimensions to the BBVs for visualisation. Each task type consists of two clearly distinct clusters of BBVs, which expose different behavior and performance at runtime. One of this clusters is eccentrically shaped (A), with the result, that task instances which are in the same cluster, but at some distance w.r.t. each other, show different performance. Furthermore, treating both clusters of a task type as if they showed the same performance (B), as in *merge-sort*, leads to a simulation error of more than 40%.

Furthermore, each cluster observed in one task type is similar to a cluster in the other task type Ⓒ, resulting in duplicated work during sampled simulation. An ideal clustering would consist in two clusters, each of which containing two of the pairwise similar clusters shown in Figure 5.2a. This is achieved by the extension of TaskPoint we present in this paper. Note that in Figure 5.2a BBVs are solely used for the purpose illustration.

Figure 5.2b illustrates how we improve TaskPoint by applying BBVs and clustering for the identification of different classes of task instances in an application. To this end, we create a BBV for each task instance. If two task instances behave similarly, they typically have similar BBVs. On the other hand, task instances with dissimilar behavior are likely to also have dissimilar BBVs. The figure illustrates that KMeans tends to split irregularly shaped clusters into many sub-clusters. In the case of DBSCAN, task instances which are connected by a dense region of other task instances are clustered together. In this work, we chose to rely on DBSCAN clustering, because a lower number of clusters requires less detailed simulation and thus allows for a higher simulation speedup.

### 5.2.4 Analytical Performance Modeling

Figure 5.2b illustrates that clusters of task instances, as they are detected by DBSCAN, can have asymmetric shape and large diameter Ⓐ. If this happens, using a sample to predict the performance of a task instance, which is further away in the same cluster, introduces a simulation error. We leverage the relative accuracy of an analytical model to correct this error during simulation.

Analytical performance models have been extensively used for sequential applications [16, 44, 67, 91]. As explained in Section 2.4.4, analytical models can be classified into *empirical* and *mechanistic* models. Empirical models aim at capturing a system's behavior with machine learning techniques, e.g. support vector machines or artificial neural networks. While they can achieve good accuracy, they do not provide much insight into why a certain design achieves better or worse performance than another. Mechanistic models employ mathematical formulas to model the effect of the key architectural parameters on performance. Mechanistic models allow to study the sources of particularly good or bad performance by simply comparing the contribution of the different terms of the model's formula. Since they provide more insight into the sources of performance, in this work we use a mechanistic performance model.

In this work, we use an analytical performance model proposed by Van den Steen et al. [115]. The model is an extension of Interval Simulation [48]. Interval Simulation requires micro-architecture dependent input data, namely the number of cache misses per cache level, the number of branch predictor misses and the

amount of memory-level parallelism (MLP). The approach presented by Van den Steen et al. eliminates the micro-architecture dependent parts of the model input. Instead, they use a micro-architecture independent application profile and generate the micro-architecture dependent elements of the model input using analytical models for caches, branch predictors and MLP.

Caches are modeled using *StatStack* [46], a technique for modeling arbitrarily sized LRU caches. StatStack's model requires the *reuse distances* of the modeled application as an input. The reuse distance is the number of memory accesses between two accesses to the same cache line. Based on the reuse distance profile, StatStack predicts an application's cache miss rate.

Branch predictors are modeled using the *Linear Entropy* model [91], proposed by Pestel et al. First, the application to be modelled is executed in a profiler which, for each static branch instruction and each history of past branches, counts the number of times the corresponding branch is taken and not taken. This information is afterwards used to calculate each branches entropy. A linear model predicts the per-branch missrate for several different branch predictors based on the per-branch entropy.

MLP is the number of simultaneously outstanding LLC misses, i.e. the number of memory accesses which can be served by the DRAM subsystem in parallel. Van den Steen et al. propose an MLP model, which separates MLP calculation into a fraction stemming from LLC cold misses. and a fraction stemming from capacity- and conflict misses.

## 5.3 Sampled Simulation of Task-Based Programs

In this section, we present our TaskPoint methodology. First, we introduce the prerequisites which need to be fulfilled by an architectural simulator in order to serve as an implementation platform for TaskPoint. Next, we present the different phases of TaskPoint's sampling mechanism, namely warm-up, sampling and fast-forwarding. Afterwards, we introduce our periodic sampling policy. The separation into sampling mechanism and policy allows for the integration of other sampling policies with low implementation effort.

### 5.3.1 Requirements for the Architectural Simulator

Our objective is to provide a sampled simulation methodology for task-based programs which does not depend on a specific architectural simulator. Therefore, we keep the requirements for the target simulator to a minimum. In order to serve as a suitable platform for implementing our methodology, a simulator needs to fulfil the following two requirements:

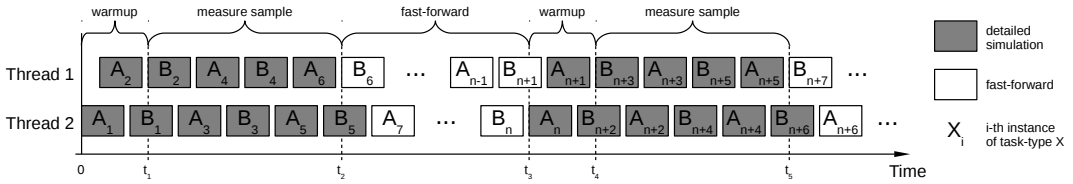



FIGURE 5.3: Initial warmup, sampling, fast-forwarding and resampling in TaskPoint

1. The simulator needs to feature a detailed and a fast simulation mode.

2. The fast mode has to be capable of operating at a user-specified IPC.

Most contemporary architectural simulators feature several levels of detail [6, 14, 103], allowing to trade off speed for accuracy. Thus, we assume the first requirement to be trivially fulfilled. Regarding the second requirement, if a simulator does not support fixed-IPC simulation by default, we consider the implementation of this functionality to be a minor effort.

### 5.3.2 Sampling Mechanism

TaskPoint operates on the level of granularity of task instances. A task instance is simulated either in detailed or in fast mode. Simulation in detailed mode serves for warming architectural state or to measure samples, whereas simulation in fast mode accurately fast-forwards simulation time. Switching between detailed and fast mode only occurs between two consecutive task instances.

Figure 5.3 illustrates the different phases of TaskPoint. For each task type, we maintain two vectors holding the IPC histories of the most recently simulated task instances. The size $H$ of these vectors is a parameter referred to as the *history size*. Both vectors are FIFO buffers in which a newly added element replaces the oldest one. The first vector contains the history of task instances which are valid samples, i.e. which are simulated after warming up architectural state. We refer to it as the *history of valid samples*. The second vector holds the history of all task instances simulated in detailed mode, regardless of the simulation being properly warmed. We refer to it as the *history of all samples*. While the former is the sample history we usually use to determine which IPC to use in fast mode, the latter is needed if there are task types that occur infrequently and can not be sampled in a single sampling interval. We refer to these task types as *rare task types*.

In multi-threaded applications, co-existing threads interfere with each other, e.g. by competing for shared resources, through inter-thread synchronization or by invalidating data residing in remote caches. In order to correctly model thread interference, we simulate all threads either in detailed mode or in fast mode. Since we assume that mode switching only occurs between two consecutive task instances,

there are short phases during which some threads are simulated in fast-forward mode, while others are simulated in detailed mode (see $t_2$, $t_3$ and $t_5$ in Figure 5.3).

#### Simulation Warmup

Before conducting performance measurements, a simulation needs to be *warmed*, i.e. it needs to be put in a representative state. Warming micro-architectural state in sampled simulation is well-studied [38, 58, 107, 117, 121, 124]. In this thesis, we warm the simulation by simulating an empirically determined number of task instances in detail and avoid complex warmup schemes. Instead, we focus on the sampling methodology itself. However, we distinguish between warming at simulation start and warming before resampling after a simulation phase in fast mode. When a task instance simulated for warmup finishes execution, its IPC is added to the history of all samples.

At simulation start, all simulated micro-architectural structures are in their initial (*cold*) state. During detailed simulation, state-holding elements begin to fill until occupancy reaches a steady state. In this work, we assume that simulating $W$ task instances per thread at simulation start is sufficient for putting the simulator into a representative (*warm*) state. We refer to $W$ as the *size of the warm-up interval* and evaluate different values for $W$ in Section 5.4.

After a simulation phase in fast mode, micro-architectural state is stale. Before resampling the simulation, warmup makes sure that micro-architectural state is (approximately) the same as if the whole program was simulated in detail. Before resampling, we perform detailed simulation until every thread has simulated one task instance in detail.

#### Sampling

Like simulation warmup, sampling is performed in detailed simulation mode. When warmup is finished, we start treating the simulated task instances as valid samples. When a valid sample task instance finishes simulation, its average IPC is added to the history of valid samples and to the history of all samples. We trigger the transition to fast mode when one of the following two conditions is fulfilled:

1. The history of valid samples is fully populated.
2. A certain number of task instances has been simulated without encountering any instance of a rare task type whose history of valid samples is not yet fully populated.

The first condition means that all task types are fully sampled. The second condition is needed to avoid spending an excessive amount of time on detailed simulation in

the presence of rare task types. In this paper, we cut off sampling when all threads have simulated 5 task instances without encountering an instance of a previously observed rare task type.

#### Accurate Fast-Forwarding

When the transition to fast mode is triggered, all task instances starting in the future are simulated in fast mode. However, task instances which started in the past are simulated in detailed mode until they complete. Task instances finishing simulation after the transition to fast mode are only added to the history of all samples.

A task instance simulated in fast mode is simulated with the average IPC of the history of valid samples of its task type. If a task instance belongs to a rare task type whose history of valid samples is empty, we use the average IPC of the history of all samples instead. If the history of all samples of the corresponding task type is also empty, we trigger resampling.

Rare task types tend to occur infrequently during the execution of an application. They account only for a small percentage of the total instruction count of an application and are used for infrequent tasks, e.g. setting up and deleting data structures. We find the impact of using non-representative samples for fast simulation of rare task types to be negligible.

One contribution of this paper is the presented fast-forwarding mechanism for architectural simulation of task-based parallel programs. Our technique fast-forwards each thread at a rate depending on the task type of the task instance currently being simulated.

### Clustering Task Instances

Our improved version of TaskPoint identifies classes of task instances with similar behavior prior to simulation. In a profiling step, we determine the BBVs of each task instance of the application. Afterwards, BBVs are normalized and clustered using the DBSCAN algorithm. BBVs are micro-architecture independent. Hence, the costs of BBV generation and clustering are amortized across all simulations of the application. In a trace-based simulation environment, BBVs can be generated together with the application trace.

### Analytical Modeling

Clustering of task instances with DBSCAN typically leads to a smaller number of clusters, compared to clustering based on KMeans. The downside is that DBSCAN can classify task instances with different, but not similar, behavior into the same cluster, as illustrated in Figure 5.2b. For this reason, the performance of a task instance

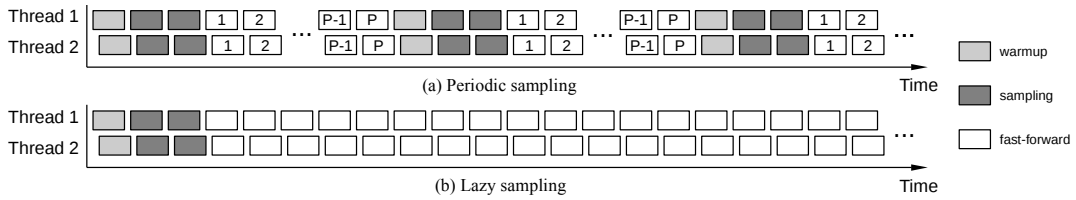



FIGURE 5.4: Illustration of periodic sampling (a) and lazy sampling (b) as a special case of periodic sampling with infinite sampling period $P$

observed in detailed simulation may not be representative for all task instances of a cluster. In our extension of TaskPoint, we employ an analytical performance model to correct this performance difference.

First, we generate a profile of the simulated application. This profile includes the micro-architecture independent performance metrics required as inputs to the analytical model and can be generated in the same profiling run as the per-task-instance BBVs used for task instance clustering. As the BBVs, this profile is only generated once per application.

Afterwards, we evaluate the analytical model for all task instances of the simulated application, assuming the same system configuration as the one used in detailed simulation. For each simulated application, the model is evaluated once per architectural configuration. Changing only the number of simulated threads does not require to reevaluate the model.

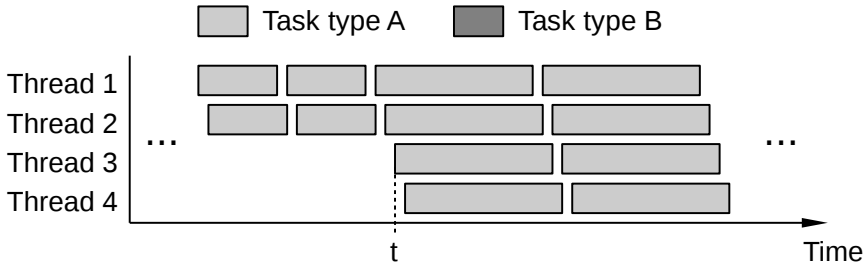
We use the performance information obtained from the analytical model as follows: assume that two task instances $i$ and $j$ belong to the same cluster. Furthermore, $i$ has been simulated in detail $j$ is to be simulated in fast-forward mode, using $i$ as performance sample. Let $IPC_{i,m}$ and $IPC_{j,m}$ be the IPC of task instances $i$ and $j$, respectively, as predicted by the model, and $IPC_{i,d}$ the IPC obtained in detailed simulation of $i$. We estimate the performance $IPC_{j,ff}$ of $j$ in fast-forward mode according to Equation 5.1:

$$IPC_{j,ff} = IPC_{i,d} \cdot \frac{IPC_{j,m}}{IPC_{i,m}} \tag{5.1}$$

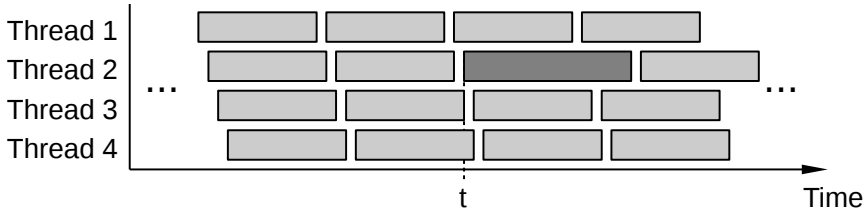
In other words, the IPC of the sample is multiplied with the performance ratio of sample and fast-forwarded task instance. By following this approach, we combine the accuracy of detailed simulation with the relative accuracy of the analytical model.

### 5.3.3 Periodic Sampling Policy

A sampling policy decides when to resample a simulation running in fast-forward mode. The *periodic sampling* policy, illustrated in Figure 5.4a, warms and samples a




(A) Change in number of execution threads at time $t$, thus altering average performance due to resource contention




(B) Instance of rare task type starting execution at time $t$

FIGURE 5.5: Illustration of changing number of execution threads (a) and rare task type (b)

simulation at simulation start. Afterwards, it switches the simulation to fast-forward mode. When a thread has executed a number $P$ of task instances of any task type in fast-forward mode, the simulation is resampled. We refer to the parameter $P$ as the *sampling period*. When a simulation is resampled, the entries of the history of valid samples are discarded. When resampling is complete, the simulation returns to fast-forward mode and the process repeats.

Simulation speedup is determined by the size of the sampling period. The larger the sampling period, the more task instances are simulated in fast mode. In the special case of an infinite sampling period, resampling is never triggered by the sampling policy. We refer to this case as *lazy sampling*. Lazy sampling is illustrated in Figure 5.4b. If the number of task instances of a program is too small or the sampling period is too large, a simulation finishes during the first fast-forward interval, before any thread has simulated $P$ task instances. In this case, periodic sampling is equivalent to lazy sampling.

Besides the aforementioned case of a thread having simulated $P$ task instances in fast mode, resampling is also triggered when it is impossible to accurately simulate a task instance in fast mode. This happens in the following two cases.

Figure 5.5a shows a case where the number of threads participating in task execution changes at runtime, e.g. when the simulated application enters a phase exposing more parallelism. When the number of execution threads changes, so does the contention on shared resources, like shared caches and main memory. This affects per-thread performance and invalidates previously measured samples. Resampling avoids prediction errors due to non-representative samples.

Figure 5.5b shows a case where the first instance of a new task type is encountered while simulating in fast mode. When encountering an instance of a previously unknown task type, the task type's sample history is empty. Therefore, it is impossible to simulate this task instance in fast mode. We circumvent this problem by triggering resampling.

With this resampling strategy, both periodic sampling and lazy sampling account for phase changes in the application. If a new phase is implemented with different task types, the simulation is resampled. The same holds for changes in the available computation resources or the available parallelism.

## 5.4 Evaluation

In this section, we first introduce specific aspects of the experimental setup we use to implement and evaluate TaskPoint. We introduce the two architectures we simulate in our evaluation of TaskPoint. Afterwards, we show, how we extend the TaskSim simulator to enable fast-forwarding a simulation at an arbitrary, user-defined per-thread IPC. A general introduction to TaskSim can be found in Section 3.3. For an introduction to the OmpSs programming model we refer to Section 3.1. The benchmarks used for our evaluation of TaskPoint are presented in Section 3.4.1. After presenting the experimental setup, we proceed with an evaluation of TaskPoint's model parameters, simulation error, and simulation speedup.

#### Simulated Architectures

We evaluate the fidelity of our methodology by investigating simulation speedup and execution time error of multi-threaded simulations of two radically different multi-core architectures. One resembles a server-class system, while the other resembles a low-power mobile platform. Table 5.1 lists the key characteristics of the simulated architectures. The high performance architecture features a large reorder buffer and a three-level cache hierarchy, as found in HPC systems. The low-power architecture has a smaller reorder buffer and two levels of cache memories, as is typical for battery powered mobile systems. Recently, low-power systems are gaining interest for applications in HPC [94].

#### Extension of the TaskSim Simulator

As stated in Section 3.3, TaskSim features a detailed and a fast-forwarding mode. In the fast-forwarding mode, called *burst mode*, TaskSim only accounts for the number of CPU cycles between events, in this case between the beginning and the end of the execution of a task instance. In the existing implementation, TaskSim reads a task

TABLE 5.1: Architectural parameters of high performance and mobile configurations used for model validation

| Parameter | High-perf. | Low-power |
|---|---|---|
| Reorder-buffer size | 168 | 40 |
| Issue width | 4 | 3 |
| Commit rate | 4 | 3 |
| Cache line size | 64 B | 64 B |
| L1 cache | 32 kB private | 32 kB private |
| | 4 cycles latency | 4 cycles latency |
| | 8-way associative | 2-way associative |
| L2 cache | 2 MB private | 1 MB shared |
| | 11 cycles latency | 21 cycles latency |
| | 8-way associative | 16-way associative |
| L3 cache | 20 MB shared | none |
| | 28 cycles latency | |
| | 20-way associative | |

instance's cycle count from the application trace. In the implementation of our fast-forward mechanism, the duration of a task instance is calculated at the beginning of its execution. Using the mean IPC of the sample history of a task instance $i$'s task type $T$ and its dynamic instruction count $I_i$, we estimate its number of execution cycles $C_i$ according to $C_i = \frac{I_i}{IPC_T}$. The result is the number of cycles it takes to execute the task instance at an IPC of $IPC_T$, the average IPC of the instance's task type. The dynamic instruction count is read from the application trace.

In the scope of this work, we extended TaskSim with the capability to switch between detailed and fast-forward mode at runtime. We also extended its fast simulation mode. Instead of using previously recorded cycle counts from a trace, our implementation of fast mode uses cycle counts predicted by our fast-forward mechanism. To the best of our knowledge, this is the first fast-forward mechanism applying different IPCs to different parts of a program. Our mechanism allows fast-forwarding dynamically scheduled parallel programs in which the per-thread instruction stream is a-priori unknown. Next, we evaluate performance variation of task-based programs observed in simulation with TaskSim.

In this section, we conduct a sensitivity analysis of TaskPoint's model parameters. Then, we evaluate execution time error and simulation speedup of periodic sampling and lazy sampling. Finally, we test the robustness of our model by using the same parameters to simulate a low-power architecture.

### 5.4.1 Adjusting the Model Parameters

We determine the optimal model parameters following an incremental approach. First, we determine the optimal number $W$ of task instances needed for warmup at

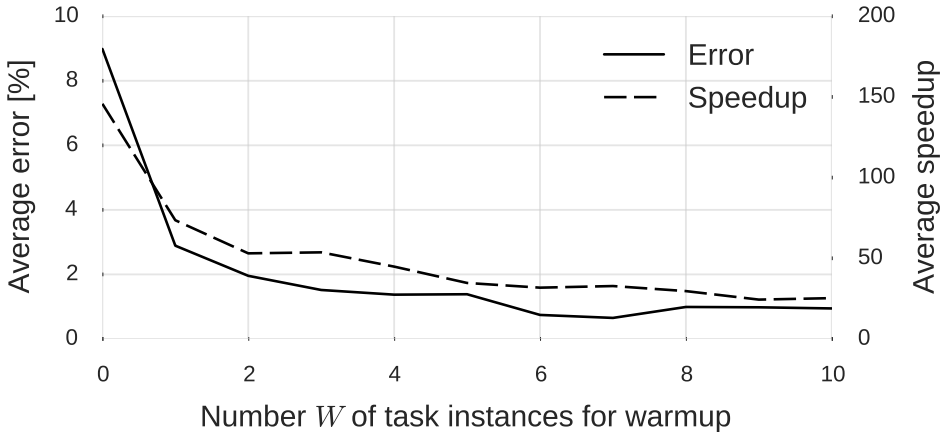



FIGURE 5.6: Error and speedup for different sizes of warmup interval

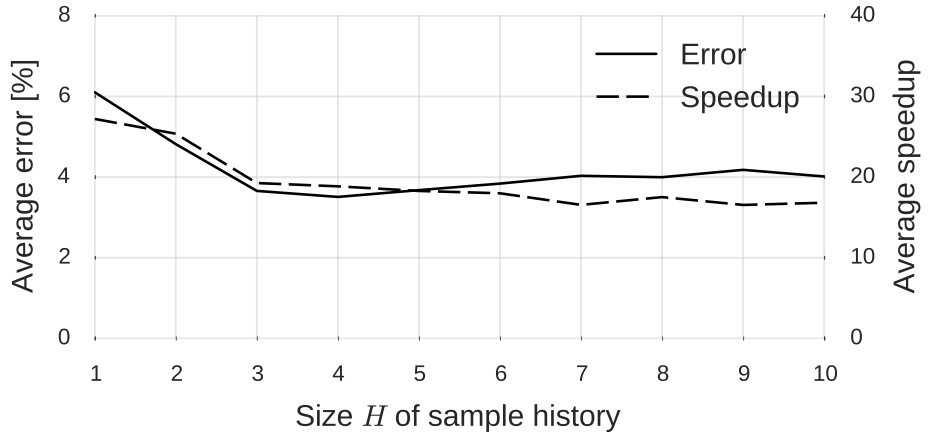



FIGURE 5.7: Error and speedup for different sizes of sample history

simulation start. Afterwards, we consider different numbers of task instances $H$ constituting the sample history. Finally, we explore a range of values for the sampling period $P$.

In order to determine the optimal value for $W$ we set $H = 10$ and $P = \infty$ and evaluate different values ranging from $W = 0$ (no warmup) to $W = 10$. Figure 5.6 shows error and speedup, averaged over simulations with 32 and 64 threads. The reported values are averaged over the benchmarks and kernels with an error $> 5\%$ for at least one value of $H$, namely *2d-convolution*, *3d-stencil*, *atomic-monte-carlo-dynamics*, *knn* and *blackscholes*. We found that $W = 2$ yields an average error of less than 2%. Larger values of $W$ do not significantly reduce the average error, but they reduce simulation speedup. Therefore, for the remainder of this paper, we set $W = 2$.

Next, we evaluate different values for $H$, the size of the sample history. For this purpose, we set $P = \infty$. Note that we already set $W = 2$. Figure 5.7 shows error and speedup for different sizes $H$ of the sample history, averaged over simulations with 32 and 64 threads of the aforementioned benchmarks. We found that $H = 4$ minimizes the average error. This value also minimizes the standard deviation of

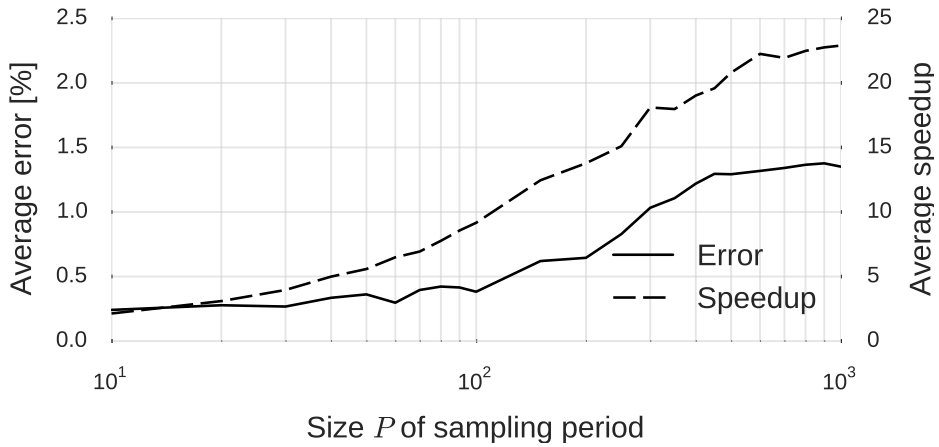



FIGURE 5.8: Error and speedup for different sizes of sampling period

the average error, which is not shown in the Figure. Larger values of $H$ do not only result in a larger average error, but also in lower simulation speedup. Therefore, for the remainder of this thesis, we set $H = 4$.

Finally, we explore different sizes of the sampling period $P$. With $W = 2$ and $H = 4$ already fixed, $P$ is the only remaining parameter. Figure 5.8 shows the average error for values of $P$ ranging from $10$ to $1,000$. We find that average error and speedup increase with the size of the sampling period. The larger the value of $P$, more task instances are simulated in fast mode. Since the total number of task instances of a program is constant, the fraction of detailed simulation decreases, resulting in increasing speedup. For $P \geq 1000$ error and speedup remain constant. At this point, none of the investigated programs has a sufficient number of task instances for resampling the simulation at least once and periodic sampling becomes equivalent to lazy sampling.

We aim for a simulation error of less than 1%. A sampling period $P = 250$ yields an error of 0.8% and a simulation speedup of 15.1x, averaged over the benchmarks used in our sensitivity analysis. In the remainder of this section, we evaluate TaskPoint for periodic sampling with $P = 250$ and for lazy sampling (periodic sampling with $P = \infty$).

### 5.4.2 Periodic Sampling

First, we evaluate periodic sampling, simulating the high-performance architecture in Table 5.1, which we also use to find the sampling parameters. Afterwards, we simulate the low-power architecture using the same sampling parameters.

#### High-Performance Architecture

Figure 5.9 shows execution time error and simulation speedup for all investigated benchmarks, simulated with the parameters $W = 2$, $H = 4$ and $P = 250$.

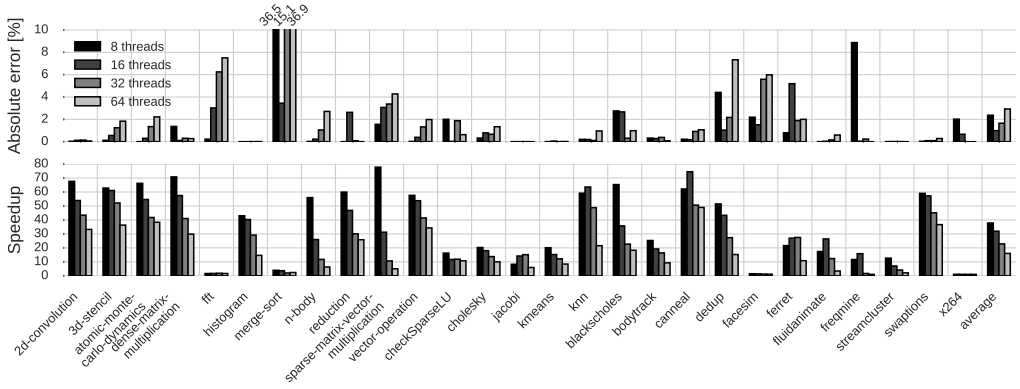



FIGURE 5.9: Error and speedup of periodic sampling; high-performance architecture; $P = 250$

The average execution time error is less than 2% for 8, 16, 32 and 64 simulated threads. The error for 1, 2 and 4 simulated threads is less than 1% and not shown in the Figure. We observe the largest simulation speedup of 76.2 for *sparse-matrix-vector-multiplication* executed with 8 threads.

We observe the highest error of 36.9% for *merge-sort* simulated with 64 threads. We attribute this error to the fact that each of *merge-sort* task types has two distinct classes of behavior, as stated earlier. Later on in this section we show, how model-based simulation improves this error.

The simulation of *freqmine* with 8 threads shows an error of 8.9%. *Freqmine* consists of 7 different task types, one of which accounts for 93% of the total number of dynamic instructions. The dynamic instruction count of the instances of this task type ranges from 490 to 11,000,000. Inspecting the source code reveals a construct of nested `if`-statements in a task declaration. This causes different instances of the same task type to follow completely unrelated control flow paths. The unbalanced size across task instances makes sampling the simulations with 32 and 64 threads ineffective. Since these configurations are simulated almost entirely in detail, the error is negligible and speedup is close to 1.

From this finding, we derive a recommendation to programmers for improving performance predictability of task-based programs: One should avoid large-scale control flow divergence among instances of the same task type. In practice, this is achieved by declaring code performing unrelated work as different task types.

We observe an error of 7.3% in the case of *dedup* with 64 threads. *Dedup* consists of 4 task types, one of which accounts for 99.9% of the dynamic instruction count. The dynamic instruction count of the instances of this task type ranges from 3,500,000 to 25,100,000. The dominating task type performs de-duplication as well as compression, which are highly input dependent operations. Previous work identified input dependence as a source of performance variation [53]. Performance variation makes

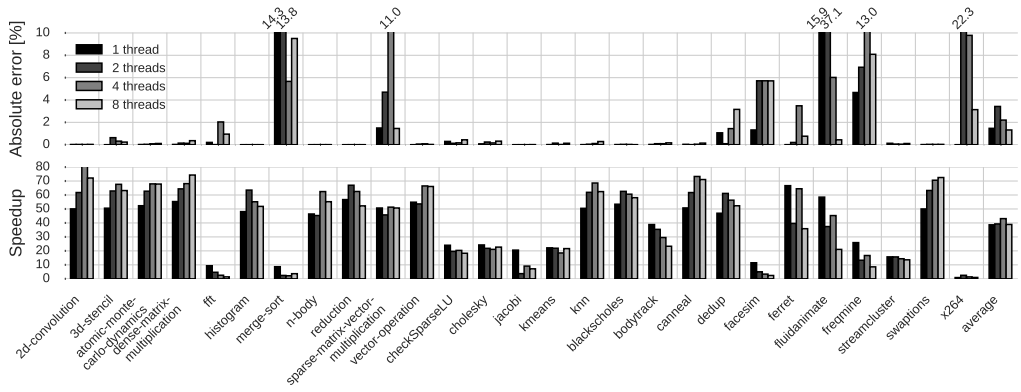



FIGURE 5.10: Error and speedup of periodic sampling; low-power architecture; $P = 250$

it difficult to determine a task type's average performance during sampling.

We recognize that, in certain cases, input dependence can not be avoided. One way to improve the accuracy of sampled simulation of programs showing input dependence is to classify task instances into classes of similar performance. We envision clustering of instances of the same task type based on micro-architecture independent metrics, e.g. instruction count or instruction mix. We leave this for future work.

Next, we evaluate the generalization capability of periodic sampling. We simulate a low-power architecture which is radically different from the high-performance architecture we used to determine the sampling parameters.

### Low-Power Architecture

Figure 5.10 shows execution time error and simulation speedup for simulations of all benchmarks executed on the low-power architecture introduced in Table 5.1 with 1, 2, 4 and 8 threads. We notice that, for increasing thread counts, speedup degrades less than in the case of the high-performance architecture. Since we simulate smaller thread counts, the simulation is resampled more often and the percentage of task instances simulated in fast mode is more similar across different thread counts.

With an error of 37.1% for 2 threads, *fluidanimate* is the benchmark with the highest error. In the case of *fluidanimate*, the instruction count of task instances varies between 1 million and 70 million, whereas task instances with more instructions tend to execute at a higher IPC. The instruction count and IPC variation is caused by the fact that all task instances perform an index computation that is highly inefficient for high indexes. The assumption, that task instances of the same type have similar performance is thus not fulfilled.

*x264* shows an error of up to 22.3%. This benchmark performs video transcoding, which is a highly input-dependent operation. Each frame is processed by a different

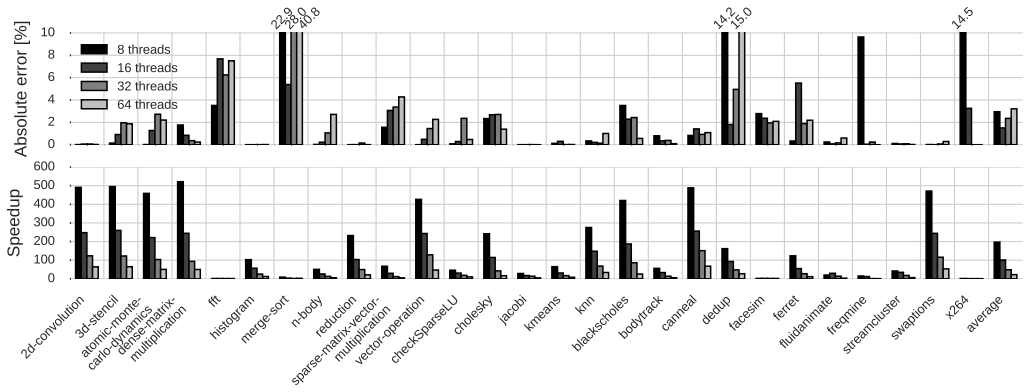



FIGURE 5.11: Error and speed-up of lazy sampling; high-performance architecture

task instance. Depending on the properties of a frame, performance can vary in a wide range, resulting in a large simulation error.

*Merge-sort* and *freqmine* are other benchmarks with significant errors of up to 14.3% and 13.0%, respectively. This is consistent with the simulation of the high-performance architecture. We attribute this error to the same reason as in the case of the high-performance architecture, namely inconsistent behavior among task instances belonging to the same task type.

Interestingly, with 11.0%, *sparse-matrix-vector-multiplication* shows a larger error for the low-power architecture than for the high-performance architecture. Depending on the structure of the input matrix, memory accesses are more or less regular [52]. We conclude that, due to the two-level cache hierarchy, the smaller last-level cache and the lower memory bandwidth, this has a higher impact on performance variation than in the high-performance architecture. This is another example of input dependence, similar to the case of *dedup* explained in the previous section.

### 5.4.3 Lazy Sampling

For our evaluation of lazy sampling, we set $W = 2$, $H = 4$ and $P = \infty$. We simulate the benchmarks listed in Table 3.2 executing on the high performance architecture and the low-power architecture listed in Table 5.1.

#### High-Performance Architecture

Figure 5.11 shows execution time error and simulation speedup of the lazy sampling policy for the investigated benchmarks executed on the high-performance architecture. The average error is less than 3.5% for all simulated thread counts (including 1, 2, and 4 threads, which are not shown in the Figure).

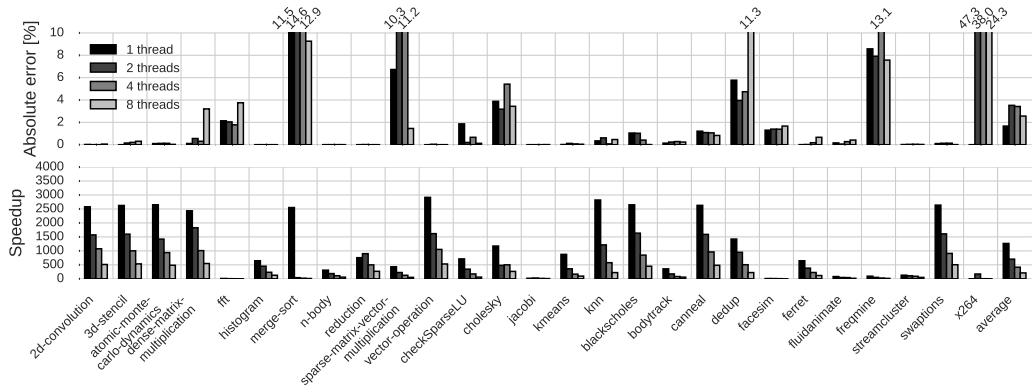



FIGURE 5.12: Error and speed-up of lazy sampling; low-power architecture

*Merge-sort* and *freqmine* are still among the benchmarks showing the highest error. Compared to periodic sampling, the highest observed error of *merge-sort* increases from 36.9% to 40.8% for the simulation with 64 threads. In the case of *freqmine*, the highest observed error increases from 8.9% to 9.6% for the simulation with 8 threads.

With up to 15.0% and 14.5%, *dedup* and *x264* show considerably larger errors compared to periodic sampling. This indicates that by resampling the simulation periodic sampling is able to reduce the error for benchmarks with irregular behavior.

While the average error of lazy sampling is comparable to the error of periodic sampling, we observe a significant increase of average simulation speedup. Compared to periodic sampling, we observe the largest increase from 37.8 to 197.0 for the average speedup of the simulations with 8 threads. The smallest gain in speedup is observed for the simulations with 64 threads, in which speedup increases from 16.0 to 22.5. For 1 thread, which is not shown in the Figure, speedup increases from 35.2 to 1244.5.

### Low-Power Architecture

Figure 5.12 shows execution time error and simulation speedup for the low-power architecture. We observe a marginal increase of the maximum error of *merge-sort*, *sparse-matrix-vector-multiplication* and *freqmine*. For *dedup* and *x264*, the error increases for all simulated thread counts. We observe the highest increase, from 22.3% to 47.3%, for the simulation of *x264* with 2 threads.

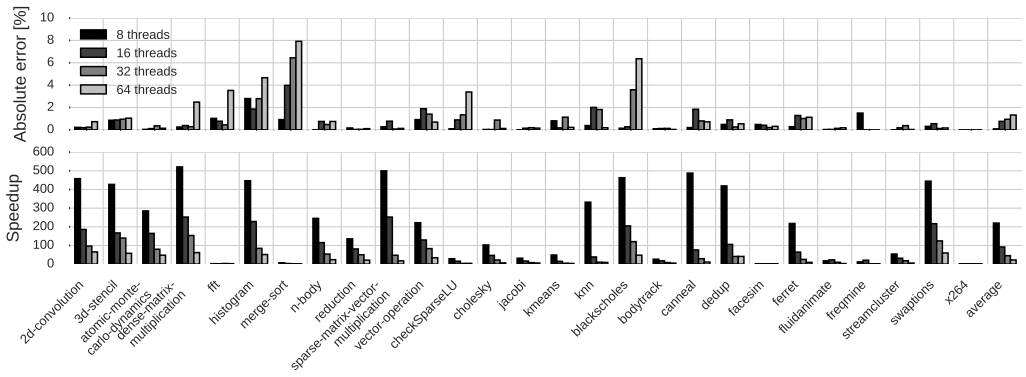



FIGURE 5.13: Error and speed-up with analytical model; high-performance architecture

#### Limitations of Lazy Sampling

Our results show, that lazy sampling achieves significantly higher simulation speedup, compared to periodic sampling. However, lazy sampling can lead to simulation errors of more than 40%, especially if the simulated application contains task types whose instances expose varying behavior. The most notable cases are *merge-sort*, *dedup*, *freqmine* and *x264*. This motivates the use of smarter clustering techniques to detect classes of task instances with related behavior. As stated earlier, we use DBSCAN clustering in order to avoid eccentrically shaped clusters being split into multiple sub-clusters. We correct the resulting simulation error using performance predictions obtained from an analytical model.

### 5.4.4 Analytical modeling

For our evaluation of model-based simulation, we assume the same sampling parameters as for lazy sampling, i.e. $W = 2$, $H = 4$ and $P = \infty$. The application profiles are generated together with the application traces before simulation. For each simulated architecture, the model is evaluated once per benchmark.

#### High-Performance Architecture

Figure 5.13 shows error and speedup for the model-based simulations of the high-performance architecture. The average error ranges from 0.09% for 8 threads to 1.32% for 64 threads. In comparison, lazy sampling shows average errors of almost 3% for 8 and 64 simulated threads.

For 21 out of 27 benchmarks, we observe errors of less than 2% across all simulated thread counts. The highest error is 8% in the case of *merge-sort*, which is a significant improvement over the 40.8% observed in the case of lazy sampling.

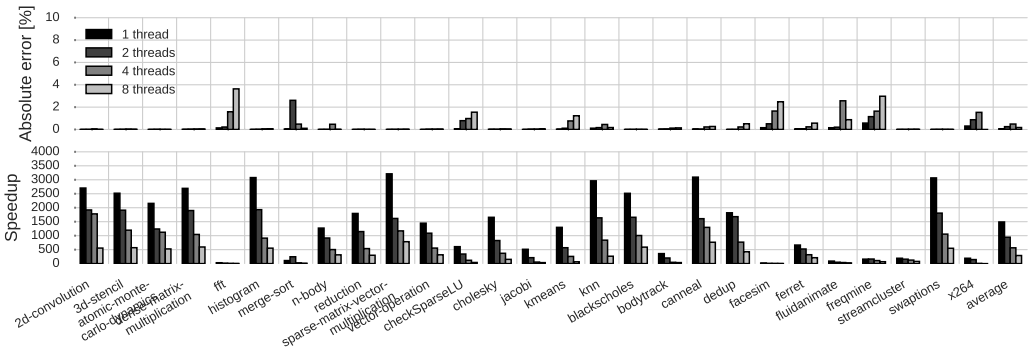



FIGURE 5.14: Error and speed-up with analytical model; low-power architecture

For some benchmarks, the simulation error increases for increasing thread counts. In particular, this happens for the benchmarks *dense-matrix-multiplication*, *fft*, *histogram*, *merge-sort*, *checkSparseLU* and *blackscholes*. In our current implementation, the analytical model does not model contention in the shared LLC. For the aforementioned benchmarks, we find the LLC misses per kilo-instruction (MPKI) to increase for an increasing number of threads, which supports our aforementioned hypothesis.

With an average speedup of 220 for 8 threads, model-based simulation is faster than lazy sampling, which achieves a speedup of 200. At the same time, the simulation error is reduced to less than half, compared to lazy sampling. Thus, model-based simulation is superior to lazy sampling both in terms of error and speedup.

### Low-Power Architecture

Figure 5.14 shows error and speedup for the model-based simulations of the low-power architecture. The average error ranges from 0.06% for 1 thread to 0.49% for 4 threads, which is a large improvement over lazy sampling.

For 22 of 27 benchmarks, the maximum error across all thread counts is less than 2%. For 13 of these benchmarks the error is even less than 0.1%. The largest error of 3.6% occurs in the case of *fft*. As in the case of the high-performance architecture, for some benchmarks the error increases when increasing the number of simulated threads.

The average simulation speedup ranges from 290 for 8 threads to 1490 for 1 thread. Lazy sampling achieves average speedups of 240 and 1300, respectively. Thus, as in the case of the high-performance architecture, model-based simulation achieves superior simulation accuracy and speed.

**Summary**

The results of our evaluation show that TaskPoint accurately predicts execution time of task-based programs. For lazy sampling, the average error is 3.2% with a maximum error of 40.8% and a simulation speedup of 22.5. We show that, for most benchmarks, periodic sampling leads to a smaller simulation error, however, at the expense of simulation speedup. With per-task-instance BBVs, DBSCAN clustering and analytical modelling, we reduce the simulation error across all benchmarks. For 64 simulated threads, model-based simulation leads to an average error of 1.3% with a maximum error of 7.9%. With 22.3, the simulation speedup is only slightly lower than the speedup of 22.5 of lazy sampling.

## 5.5 Summary

Previous sampled simulation techniques for parallel programs rely on profiling to identify the parameters of the sampling mechanism. Although those techniques have been proven to be accurate for statically scheduled fork-join based programs, they are not directly applicable to dynamically scheduled task-based parallel programs.

The proposed methodology enables sampled simulation of task-based parallel programs. Sampling units are identified based on the partitioning into tasks provided by the programmer. Between detailed simulation phases, we employ a novel fast-forward mechanism, which correctly reflects the different progress rates of task instances belonging to different task types and adapts to phase changes in the simulated application.

In this chapter, we extend our original methodology with BBVs and clustering to automatically determine classes of similar task instances. We correct simulation inaccuracies by applying correction factors obtained from an analytical performance model.

We assessed TaskPoint's generalization capability by using two radically different architectures to select sampling parameters and to run simulations. The evaluation results are satisfactory across a wide range of benchmarks, different numbers of simulated threads and different architecture models. The average simulation error is less than 2% at an average speedup ranging from 19× for 64 threads to 1019× for 1 thread.

# Chapter 6

# Multi-Level Simulation of Hybrid Programs

## 6.1 Introduction

The process of designing next-generation High Performance Computing (HPC) machines is extremely challenging. The increasing amount of computational resources each new generation of HPC systems integrates makes this challenge even more difficult. In addition, the trend to use commodity server processors as the common choice for designing such machines is changing, as processors with leaner core designs that feature significantly different microarchirectural characteristics are starting to make their debut in the HPC market [95, 119, 125]. Consequently, the design space for next-generation HPC machines is expanding. Novel solutions are required in order to quickly predict the performance of current and future scientific applications on those systems and to identify the best design points.

Besides taking into account the hardware, it is also important to consider its interactions with the system software (e.g. operating system, runtime system) [30, 114]. Hybrid programming models are pervasive nowadays, employing MPI for inter-node communication and a shared-memory programming model for node-level parallelism. Motivated by larger core counts within the same node, sophisticated ways of handling shared memory parallelism are becoming increasingly attractive to reduce load imbalance and thus improve parallel efficiency in large shared-memory multi-core configurations [42, 66, 79]. For example, OpenMP, the most popular approach for shared memory programming, has significantly evolved and currently incorporates advanced features such as tasking support [8, 89]. For all these reasons, parallel operations such as scheduling and synchronization are expected to become key system software components. As a result, simulators targeting next-generation HPC systems must take into account such parallel operations performed at the runtime system level.

Existing tools make simulation of large-scale HPC machines with thousands of cores unfeasible. Conventional cycle-accurate architectural simulators offer a great

level of detail, but make simulation times impractical when simulating more than a few tens [14, 21, 117] or a few hundreds of cores [103]. Higher-level simulators are able to simulate thousands of cores at the cost of not modelling any microarchitectural details or the impact of the system software [4, 40, 126]. Raising the level of abstraction is necessary, but needs to be done to an appropriate degree. Hence, it is critical to develop flexible simulation infrastructures that allow to quickly trim the vast design space while still capturing the impact of the simulated microarchitecture and system software.

In this chapter, we make the following contributions:

- We present MUSA, a multi-scale simulation approach that enables fast and accurate performance estimations of next-generation HPC machines. Our methodology seamlessly captures inter-node communication as well as intra-node microarchitectural and system software interactions, improving usability and simplifying the simulation workflow. MUSA relies on native execution traces with two levels of detail to allow simulation of different communication networks, numbers of cores per node, and relevant microarchitectural parameters.

- We validate MUSA using the NAS Multi-Zone Parallel Benchmark suite [116], and then evaluate three large-scale case studies (with up to 16,384 cores) using *BT-MZ*, *HYDRO* [75], and *SPECFEM3D* [72]. Our evaluation shows that MUSA provides accurate performance predictions by combining information at different levels of granularity. When comparing native executions and MUSA simulations with up to 2,048 cores, we achieve relative errors within 10% in the common case, demonstrating that our detailed model is able to capture microarchitectural and system software effects. In addition, we show that our simulations complete in an affordable amount of time, i.e. less than a day of total aggregated CPU time for detailed 16,384-core simulations. This allows to quickly identify scalability problems in the targeted case studies.

- Finally, we perform a design space exploration analysis using high-performance, low-power, and die-stacked DRAM processor profiles on a system with 16,384 cores. We find that for one of the evaluated HPC applications, *HYDRO*, the low-power processor can achieve on par performance even with the same number of cores, as the high-performance memory hierarchy and aggressive microarchitecture are over-dimensioned. In contrast, the other two applications benefit from an aggressive out-of-order microarchitecture design, and *SPECFEM3D* achieves better scalability by exploiting the higher memory bandwidth provided by die-stacked DRAM technology.

## 6.2 Background and Motivation

This section describes the co-design challenges in next-generation HPC systems. Afterwards, we discuss the difficulties of simulating large HPC applications and the limitations this imposes in exploring designs for future systems.

### 6.2.1 Co-Design of HPC Applications and Systems

In current HPC applications, the Message Passing Interface (MPI) is the most common way to expose parallelism across multiple computing nodes. As the number of nodes increases with the deployment of new HPC systems, node-to-node communication costs become more relevant and need further consideration when designing such systems. For example, certain applications might experience communication time overheads in the presence of load imbalance across different nodes. Finding the right ratio between the number of nodes and the number of processing units per node is a primary design decision that can greatly impact application performance. Hence, exploring such trade offs beforehand is a fundamental step when designing a new system.

In current HPC systems, nodes typically consist of a small number of sockets with shared memory. Shared-memory programming models such as OpenMP are the most common approach to express parallelism within a node. Recently, advanced tasking features or support for accelerators and SIMD constructs have been included in OpenMP. These features allow to exploit the computational power of the node while increasing programmer productivity [8, 42, 89]. In next-generation HPC systems, an appropriate amount of cache per core and enough memory bandwidth are paramount to achieve the desired performance within a node when running one of the targeted applications. Therefore, provisioning a node with enough resources to fit such demands is a design decision that needs to be considered when designing an HPC system.

Hybrid programming models simultaneously employ different paradigms to exploit both inter- and intra-node parallelism, e.g. MPI and OpenMP. To achieve peak performance it is important to have an even amount of computation distributed across the different nodes, and that the available parallelism within a node maps well to the available resources. By properly dimensioning a system the node-to-node communication overheads can be minimized, while at the same time achieving the desired node level performance.

### 6.2.2 Challenges Simulating Large HPC Applications

Simulation is a key tool in order to design next-generation HPC systems and applications. However, simulating future HPC systems at a meaningful scale is challenging

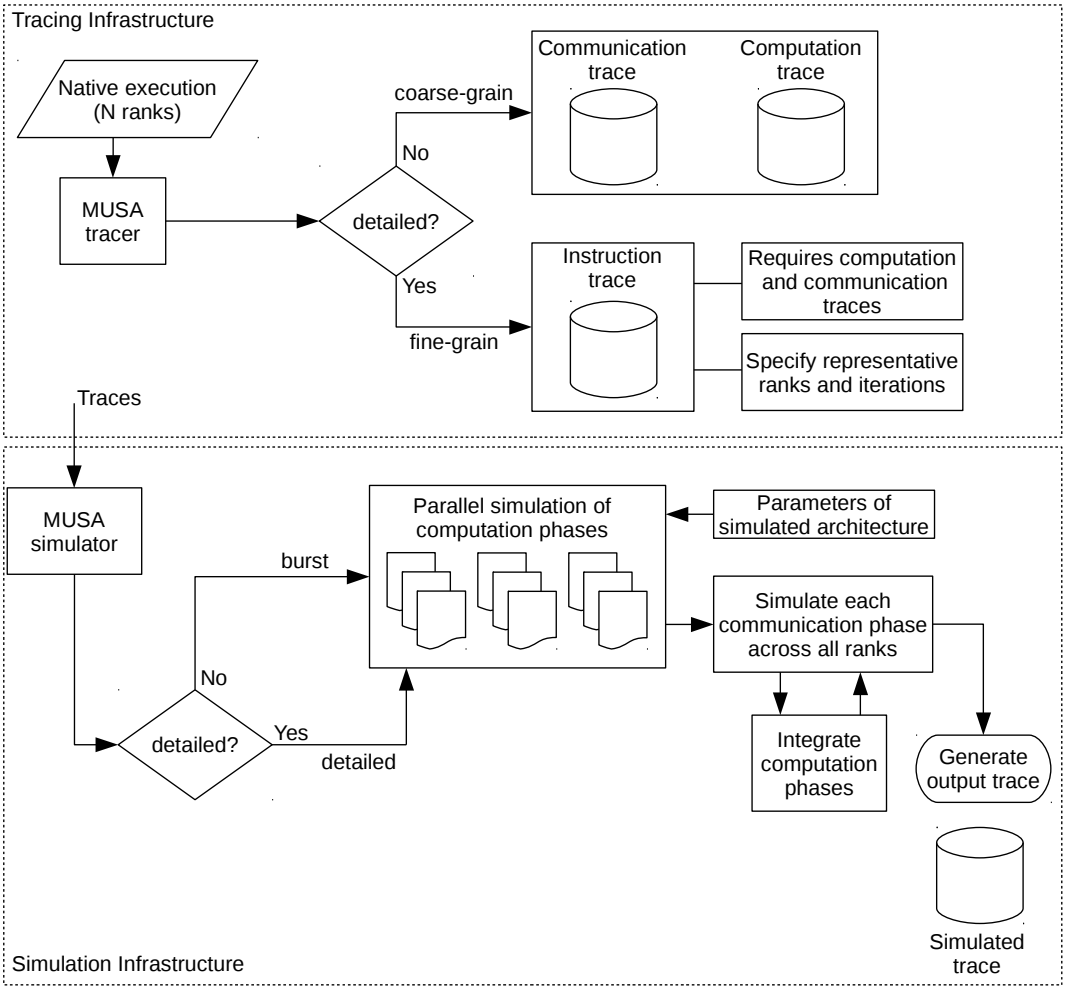



FIGURE 6.1: MUSA tracing and simulation methodology.

due to the large number of components that need to be considered. Consenquently, HPC system designers have to constantly trade off accuracy for simulation speed. As explained before, the number of nodes in the system and the amount of resources within a node can create performance bottlenecks at the inter-node and intra-node levels. Hence, scaling down the simulated system or focusing only on the node level may lead to suboptimal design decisions. Moreover, applications used in large-scale systems exhibit long execution times and downsizing the input sets to make them more manageable can change the application's characteristics, i.e. the amount of cache or memory bandwidth needed to perform well under the original input sets.

In order to simulate such large HPC systems, new methodologies are needed to gauge the necessary requirements both at the overall inter-node level as well as the intra-node level. In this paper we propose MUSA, a multi-level simulation infrastructure capable of simulating large-scale HPC systems. MUSA combines different levels of abstraction to provide insights on the expected performance of an application on a hypothetical HPC system. The following section describes the proposed infrastructure in detail.

## 6.3 Multi-Level Simulation Approach

In this section, we present MUSA, our multi-level simulation infrastructure for hybrid programs running on next-generation HPC systems.

### 6.3.1 MUSA - General Overview

MUSA is an end-to-end methodology that uses traces to enable large-scale simulations with different communication networks, numbers of cores per node, and microarchitectural parameters in a comprehensive HPC environment that considers the effects of system software. To this end, MUSA employs two components:

1. A tracing infrastructure that captures communication, computation and runtime system events

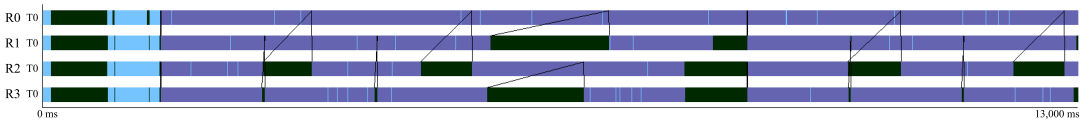
2. A simulation infrastructure that leverages these traces for simulation at multiple levels

Figure 6.1 illustrates our modular methodology that provides a streamlined workflow from tracing to the final simulation output.

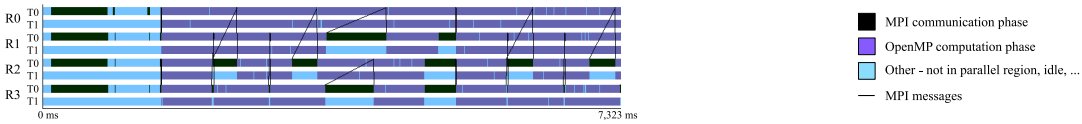
HPC applications stress a system at multiple levels, including both the hardware (i.e. pipeline, core, chip, node, network) and the software (i.e. scheduling, synchronization, communication and computation phases). Using a single simulation approach across all levels would be too rigid to adapt to the degree of detail appropriate for each level. For this reason, MUSA's simulation infrastructure is capable of changing the level of simulation detail, from cycle-accurate microarchitectural simulations to high-level analytical models. The methodology allows to combine detailed (higher computational cost) and high-level (higher simulation speed) simulations, enabling simulation of large-scale machines with thousands of cores in a reasonable amount of computational time, while guaranteeing a high degree of accuracy. The rest of this section provides further details on the tracing and simulation infrastructures.

### 6.3.2 Tracing - Capture Multi-Level Behavior

The initial step is to trace an application's execution at multiple levels. Given our targeted hybrid programming model, we start tracing each MPI process representing a *rank*. Within a rank multiple threads running in parallel may coexist, managed by a runtime system. As shown in Figure 6.1, MUSA traces an application by running it natively with the number of ranks to be used in future simulations and instructs the runtime system to execute each rank using a single thread.




(A) Coarse-grain instrumentation trace of HYDRO with 4 ranks. Delimits computation and communication phases and includes runtime events.




(B) Simulation output trace for the above input trace when simulating a system with 2 cores per rank. The runtime system is faithfully modeled.

FIGURE 6.2: Traces used in MUSA's methodology: (a) tracing infrastructure and (b) simulation infrastructure output. Traces are shown using the same time scale.

The tracer then generates a file with the communication and computation information per rank. This trace file contains information about the MPI communication phases, including:

1. Timestamps of beginning and end of each communication phase for all ranks
2. The type of each communication (e.g. collective or point-to-point)
3. The size of the data to be sent.

At the same time the computation information for each rank is recorded, storing timestamps for each computation phase and multiple runtime events such as creation and synchronization of parallel sections. The instrumentation required to obtain these traces is coarse-grained, leading to a small overhead that does not significantly affect the application's behavior.

In order to simulate a node in detail, MUSA requires additional instruction-level instrumentation for computational phases; such as the operation code, the program counter and the involved registers and memory addresses. Such detailed instrumentation is deferred to a separate native execution due to its higher overhead that might alter application behavior. Hence, when tracing in detailed mode, the timestamps taken in the first trace are used to correct any deviation in the behavior of the application introduced in the detailed trace step.

Figure 6.2a shows a trace generated by MUSA's tracing infrastructure with communication and computation information for a fraction of an application's execution time. The tracing methodology generates traces that allow simulations even if the characteristics of the simulated computational node (e.g. the number of cores, the memory hierarchy) or the communication network change. As a result, we can perform architectural analysis of a large design space using the same set of traces, reducing trace generation time and storage requirements. Section 6.4.3 contains further details on the employed tracing tools and their overheads.

### 6.3.3 Simulation - Leverage Multi-level Traces

MUSA's simulation step employs the communication and computation events gathered in the tracing step. As shown in Figure 6.1, the methodology initially identifies the different computation phases for each rank, which are independent and can be simulated in parallel. Each of these rank level computation phases is simulated with the specified number of cores and parameters of the microarchitecture and the memory hierarchy. However, MUSA is able to simulate an arbitrary number of cores per rank. To accomplish this, MUSA injects runtime system API calls by using the runtime system events recorded in the trace, effectively simulating the runtime system, including scheduling and synchronization for the number of simulated cores. The architectural simulator we employ can perform simulations either in *burst* or *detailed* mode, which allow from faster than native simulation speeds to slower but more detailed design space exploration studies, respectively. Details about the chosen architectural and network simulators can be found in Section 6.4.3.

**Burst mode simulation:** Simulations using burst mode replay the computation events traced during native execution with coarse grained instrumentation. Burst mode simulations do not take into account the contention that the memory hierarchy of a node might experience when running multiple threads, hence the obtained performance estimations are to be treated as upper bounds. However, MUSA allows the user to specify IPC correction factors to account for the impact of inter-thread contention if there is any application-specific knowledge, making burst simulations more accurate and flexible. Burst mode simulations allow faster than native simulation speeds, thus enabling quick design space exploration studies with a variable number of cores per rank and different communication networks.

**Detailed mode simulation:** When simulating a computation phase in detail, MUSA also uses the detailed traces, enabling cycle-accurate simulations with detailed models for microarchitecture and memory hierarchy. The detailed information in the instruction-level trace allows to use different cycle-accurate simulators, ranging from component-specific simulators, such as main memory, cache hierarchy, or interconnects, to detailed pipeline microarchitecture simulators. Detailed simulations can be time consuming and an appropriate simulator has to be chosen depending on the envisioned target study.

Simulating all computation phases of an application in detail is feasible for small systems and short execution times. However, when going into the domain of thousands of cores, full detailed simulation becomes prohibitive both in terms of trace size and simulation time. Fortunately, HPC applications follow certain execution patterns that are easy to identify with our visual trace format. We can leverage this information by specifying a subset of the ranks, or even a subset of the iteration phases within a rank, to be traced and simulated in detailed mode. Therefore,

MUSA allows the user to define such bounds as input parameters, giving great flexibility in deciding which computation phases are to be simulated in detail, while the performance of the remaining phases is extrapolated. Section 6.3.4 details how MUSA performs sampling of computation phases at different levels.

**Network simulation and final output:** After the computation phases have been simulated, MUSA replays the execution of the communication trace events in order to simulate the communication network and generate the final output trace of the simulation. During this process, the durations of the computation phases are replaced by the results obtained in the simulations (either in burst or detailed mode), and the communication phases are simulated using a network simulator. At the end of this process the entire simulation is complete and the output trace is generated for visualization.

Figure 6.2b shows an output simulation trace generated by MUSA when simulating two cores per rank. The simulation models the OpenMP scheduling events by calling the actual runtime system through inserted API calls for the traced events, faithfully modeling the impact of having two cores on each node. The MPI communication is processed by thread *T0* on each rank, while the computation phase load of each rank is distributed across the two cores.

### 6.3.4 Sampling - Reducing Simulation Time

Accurate microarchitectural simulation is time consuming. Conventional simulators achieve simulation speeds of 100 to 1000 KIPS [14, 99, 103]. As a consequence, detailed simulation of large systems or long-running applications becomes infeasible. While MUSA allows simulations at different levels of detail, it still requires to simulate some computation phases in detail. In an HPC application, these phases typically run for a few seconds, before starting a new communication phase.

A common technique for reducing simulation time is sampling. Sampling can be employed to allow detailed simulation of larger portions of an application or to further reduce simulation time. Sampling seeks to minimize the amount of detailed simulation by only simulating the representative parts of an application. In the following, we point out how MUSA employs sampling at three orthogonal levels of granularity in an application, namely (i) the whole application, (ii) a single MPI rank, and (iii) a computation phase within an MPI rank. For a more thorough introduction to sampled simulation we refer to Section 2.4.2.

**Application level:** Many applications in HPC show iterative behavior, with each iteration representing a step in time or space. In many cases, different iterations show very similar performance. Automatic techniques to identify iterations based on performance monitoring counters or traces of logical events have been proposed in the past [27, 65]. However, the simplest approach relies on directly analyzing the

code of the application, annotating the start and end of an iteration. When sampling at the application level, MUSA leverages these techniques to identify repetitive behavior and select a small number of iterations for detailed simulation.

**MPI rank level:** As described in Section 6.2, a common programming technique in HPC applications is the division of the problem domain into *blocks*. Afterwards, each block is processed by a different MPI rank. Often, different MPI ranks show similar performance across all processes. Consequently, MUSA can select a subset of the MPI ranks for detailed simulation at the microarchitecture level. MUSA adopts a simple approach consisting in simulating one out of every $N$ MPI ranks (periodic sampling). There are existing techniques to automatically select representative computation phases of an application [50, 106].

**Computation phase level:** After selecting a subset of iterations and MPI ranks, all computation phases have to be simulated in detail. Identifying representative sections of a computation phase can be done automatically [107, 121], and applied to parallel applications with barriers [22], as is the case of typical OpenMP programs with parallel loops. In the case of task-based programs, MUSA allows to perform simulations with TaskPoint [54], the sampled simulation methodology for task-based programs presented in Chapter 5 of this thesis.

## 6.4 Evaluation

In this section, we present our evaluation of MUSA. First, we introduce the applications we use to evaluate MUSA, and the native HPC system used for validation. Afterwards, we introduce MUSA's tracing infrastructure. Then, we validate MUSA, before we apply our methodology to detect scalability bottlenecks in hybrid applications both at the algorithmic level, due to the lack of parallelism, and at the hardware level, due to contention on shared resources. Finally, we also present simulation time results and a design space exploration analysis.

### 6.4.1 Applications

To validate MUSA we use the NAS multi-zone benchmarks [116]: *BT-MZ*, *SP-MZ* and *LU-MZ*. The benchmarks are introduced in Section 3.4.2. For this validation step we use 16 MPI ranks with a mapping of one rank per node. Simulations are performed with 1 to 8 cores per node. We run the benchmarks with the input class D, for which we observe enough parallelism for the 16 MPI ranks employed.

In order to illustrate the potential of MUSA, we evaluate large-scale machines using *HYDRO* [75], *BT-MZ* with the large input class E, and *SPECFEM3D* [72]. The benchmarks *HYDRO* and *SPECFEM3D* are also introduced in Section 3.4.2. For the

TABLE 6.1: Application characteristics.

| Benchmark | | Characteristics | | | |
|---|---|---|---|---|---|
| Name | Input | Ranks | Tasks/rank | Iterations | Regions/iteration |
| BT-MZ | Class D | 16 | 2.3M | 250 | 1 |
| SP-MZ | Class D | 16 | 131K | 500 | 1 |
| LU-MZ | Class D | 16 | 1.3M | 300 | 1 |
| HYDRO | big | 256 | 1.0M | 200 | 2 |
| BT-MZ | Class E | 256 | 1.3M | 250 | 1 |
| SPECFEM3D | n/a | 256 | 1.9M | 10700 | 1 |

TABLE 6.2: Application tracing statistics.

| Benchmark | | Tracing | | |
|---|---|---|---|---|
| Name | Input | Overhead | Burst Trace | Detailed Trace |
| BT-MZ | Class D | 3.4% | 5.6 GB | 53.3 GB |
| SP-MZ | Class D | 1.2% | 0.4 GB | 13.7 GB |
| LU-MZ | Class D | 1.0% | 3.2 GB | 12.5 GB |
| HYDRO | big | 6.0% | 16.1 GB | 16.9 GB |
| BT-MZ | Class E | 8.5% | 57.4 GB | 120.0 GB |
| SPECFEM3D | n/a | 9.3% | 101.4 GB | 106.4 GB |

large-scale simulations we employ 256 MPI ranks, one per node, and up to 64 cores per node, resulting in simulations of up to 16,384 cores.

All applications use a hybrid programming model based on MPI [56, 82] for inter-node parallelization and a task-based programming model, OmpSs [42], for intra-node parallelization. MPI and OmpSs are introduced in Sections 2.2.2 and 2.2.4, respectively.

Table 6.1 summarizes the main characteristics of each application. It includes the number of MPI ranks, the total number of tasks per MPI rank, the number of iterations of the application and the number of parallel regions within an iteration. For example, in the case of *BT-MZ* with input class E there is an average of 5,200 tasks per parallel region ($\frac{tasks/rank}{iterations \times regions}$).

Table 6.2 lists the trace sizes of the investigated applications. Burst traces contain only MPI and OpenMP runtime system events, but no detailed instruction trace. The table clearly shows, that detailed traces can be up to an order of magnitude larger than burst traces. The table also lists the tracing overhead for generating burst traces, i.e. the application slowdown caused by the instrumentation tool. Generating a detailed trace introduces an overhead of up to three orders of magnitude, which is not shown in the table.

### 6.4.2 Native HPC Infrastructure

We validate MUSA against the MareNostrum 3 supercomputer. Each node has two sockets with an Intel Xeon E5-2670 featuring eight cores running at 2.6GHz. The

cores implement aggressive superscalar capabilities, have private L1 and L2 caches, and a shared 20MB L3 cache. The nodes are connected via a high-bandwidth InfiniBand FDR10 network. To validate MUSA, we simulate the same HPC infrastructure.

For the native executions, we present results with up to eight cores per node, making use of a single socket. This avoids factoring in non-uniform memory access timings that may bias the results. In addition, we run each native experiment five times and select the measurement that presents the lowest amount of interference due to current system load.

### 6.4.3 Tracing and Simulation Infrastructure

Our multi-level simulation infrastructure is based on two main components:

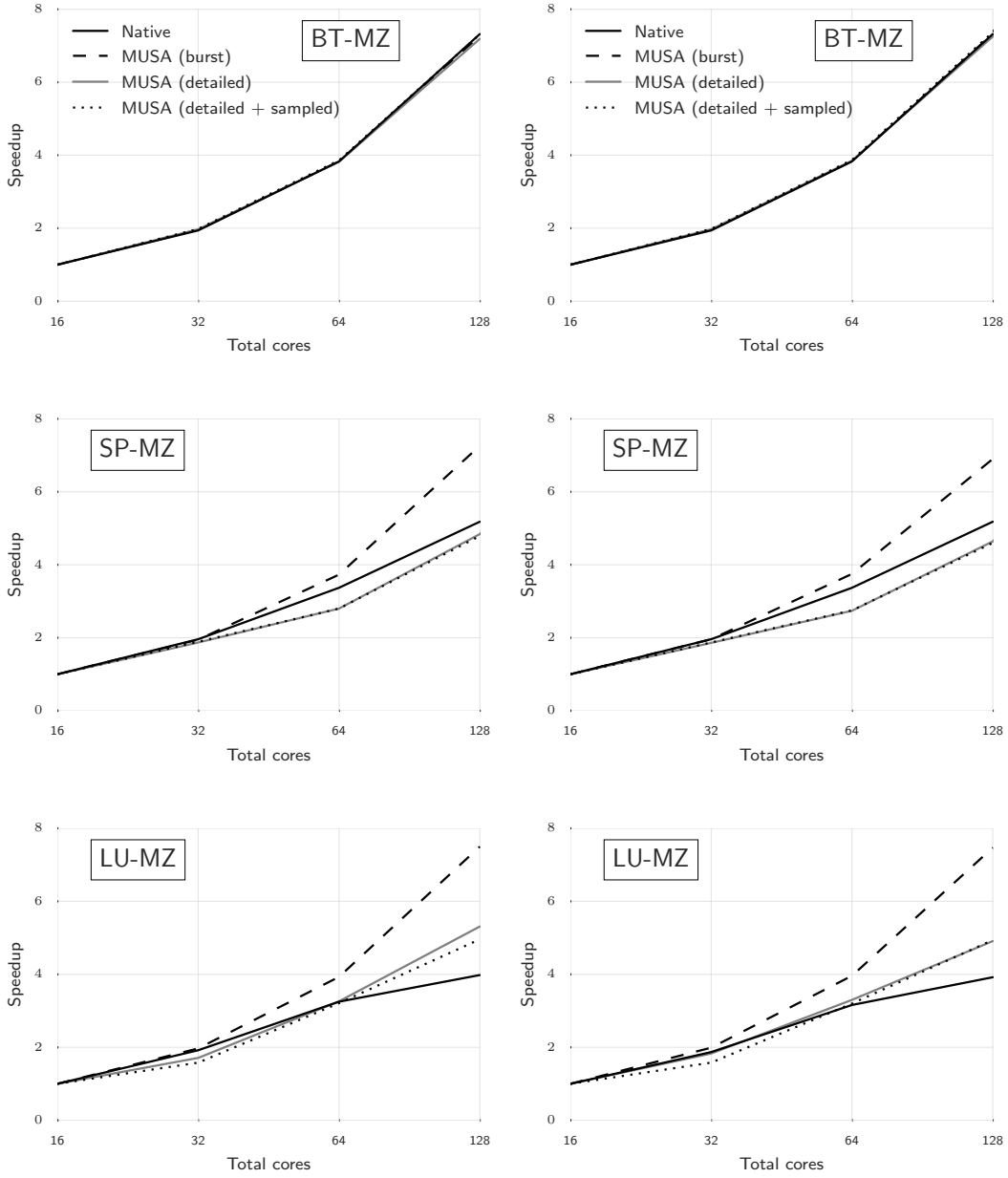
1. *Dimemas*, a high-level simulator able to model MPI communication phases using analytical models [49] (introduced in Section 2.3.3)

2. *TaskSim*, a detailed multi-core simulator with accurate memory models [99, 100] (introduced in Section 3.3)

Performing application simulations requires two steps. In the first step we generate traces that allow execution replay even if the characteristics of the simulated computational node change, e.g. the number of cores or the memory hierarchy description. Hence we can perform design space architectural analysis using the same set of traces, reducing trace storage requirements.

Traces are obtained using different lightweight tracing tools based on *extrae* [10] and *PIN* [80]. To obtain the traces for an application, we instrument a native execution that runs only a single thread per node, i.e. per MPI rank. Extrae generates the high-level trace (*burst trace*) using coarse-grain instrumentation. The tracer instruments the entire application, i.e. all ranks and iterations. However, for the detailed trace, such an approach would be impractical and require too much storage. For the evaluated set of applications, we observe that tracing the second iteration of a single MPI rank is enough to later reconstruct an application's entire execution using this information and the burst trace. This allows for manageable tracing times and storage requirements.

Table 6.2 details the overhead of generating traces at burst level, and the sizes of the burst and detailed traces for each application. The overheads include the trace disk I/O costs, which actually do not affect the application behavior, as I/O is performed at points where the application is halted by the tracer. In terms of trace sizes, burst traces are relatively small, while covering the entire execution of applications running for several minutes on the real machine. On the other hand, detailed traces are bigger, even though they only cover the second iteration of a single MPI rank. Note that a detailed trace for the entire *BT-MZ* application with input class D would




(a) A single iteration of the benchmark

(b) Entire execution of the benchmark

FIGURE 6.3: MUSA validation using the NAS Multi-Zone Parallel Benchmarks: BT-MZ (left), SP-MZ (middle) and LU-MZ (right). Benchmarks are run natively and simulated using MUSA with 16 MPI ranks and up to eight cores per node.

require more than 200 terabytes of storage. The obtained detailed traces are manageable while still allowing MUSA to perform meaningful detailed microarchitectural simulations.

Our methodology requires both an architectural and a communication network

simulator. To simulate the computation phases we use *TaskSim*, a detailed multi-core simulator with two operation modes, a fast exploration mode based on pre-calculated computation phase execution times (*burst*) and a *detailed* mode with accurate microarchitecture and memory models [99, 100]. For the network we employ *Dimemas*, which is able to model MPI communication primitives using analytical models [49]. However, we strongly believe that the MUSA methodology can be applied to nearly any simulator currently available in the community.

### 6.4.4 Validation

We validate MUSA by performing several experiments with the NAS Multi-Zone benchmarks. As described in Section 6.4.1, all validation experiments are done with 16 MPI ranks and the class D input set, always assuming a single MPI rank per node. Figure 6.3 shows the speedup for a single iteration (Figure 6.3a), and for the entire application (Figure 6.3b) when increasing the number of cores per MPI rank. Having both figures is very useful, as the overall execution time of the whole application or a single iteration can be biased by the sequential execution of a particular phase of the application, such as reading input files, initializing data structures or writing output files.

Native executions are performed with up to eight cores per rank, as this is the number of cores per socket on the available machine. Consequently, in our validation we use up to 128 cores, with parallel efficiencies that range from 48% (*LU-MZ*) to 92% (*BT-MZ*). Using a performance visualization tool, we observe that in all benchmarks the first iteration is less representative than the others. We therefore chose to trace the second iteration in detail to avoid capturing the impact of cold hardware structures in the processor. Figure 6.3 shows that scalability in native and simulated executions closely match when comparing a single iteration and the entire application.

First, we evaluate the accuracy of MUSA with burst simulations, denoted *MUSA (burst)* in the figure. A first observation is that burst simulations accurately model the system for *BT-MZ*, with negligible relative errors. This is due to the fact that *BT-MZ* is compute bound and contention on shared resources does not increase significantly with larger core counts, leading to a speedup of 7.3× on an 8 core node. However, *SP-MZ* and *LU-MZ* have higher memory contention and performance predictions start to differ from the native execution as the number of cores per node increases. For *SP-LU* and *LU-MZ*, *MUSA (burst)* predicts speedups of 6.9× and 7.5× with relative errors of 33% and 88% with respect to native runs.

The results obtained in burst simulation clearly indicate that, as we scale the number of cores in the system, cycle-accurate memory simulations are necessary to capture contention on shared resources. We perform a second set of simulations

with MUSA using detailed microarchitectural and memory models, denoted *MUSA (detailed)* in the figure. In this case, MUSA simulates one iteration of a single MPI rank and extrapolates the results to the remaining MPI ranks and iterations.

*MUSA (detailed)* improves accuracy with respect to *MUSA (burst)* for both *SP-MZ* and *LU-MZ* when simulating a system with 128 cores. In the case of *SP-MZ*, the relative error is reduced from 33% to 10%, capturing the trend observed in native execution. For *LU-MZ* the error is reduced from 88% to 25%. However, the trend is not captured as accurately as in the other two benchmarks due to modeling inaccuracies in the simulated DRAM subsystem. *LU-MZ* has poor row-buffer locality and internal bank conflicts, and thus needs a detailed component-specific simulator to capture these behavior. Therefore, for this application we would suggest to use tools like DRAMSim2 [102] or Ramulator [69]. In the case of *BT-MZ*, the error is negligible as happens in the burst simulation and, as expected, the performance is again accurately predicted.

Next, we evaluate the accuracy of MUSA using TaskPoint [54] to speed up detailed simulation, denoted *MUSA (detailed+sampling)* in the figure. In this case, we only perform detailed microarchitectural simulation on a fraction of the task instances of the application. We apply TaskPoint's default parameters: first, we simulate 2 task instances in each thread in order to warm up microarchitectural state. Afterwards, we simulate a total of 4 task instances of each task type as samples. This reduces the total simulation time by a factor of 2.5× in *BT-MZ*, 1.9× in *SP-MZ*, and 3.0× in *LU-MZ*. As shown in Figure 6.3b, *MUSA (detailed+sampling)* predicts nearly the same speedups as *MUSA (detailed)*. The average difference between these approaches is less than 3%. These results are consistent with previously published results with TaskPoint [54].

Our validation shows that MUSA provides accurate performance predictions by combining information at different levels of granularity. When comparing native executions of the entire application with MUSA simulations, we can see that the relative errors are low and that the detailed models are able to capture microarchitectural details such as memory contention. In addition, we can do this in an affordable amount of time, as even detailed simulations complete within a few hours. A more comprehensive study in terms of simulation time is shown for our large-scale simulations in Section 6.4.6.

### 6.4.5 Large-scale Simulations

We present large-scale simulations of *BT-MZ* with input class E, *HYDRO* and *SPECFEM3D* for the entire application. Table 6.1 lists the relevant application characteristics. We employ 256 MPI ranks, one per node, with up to 8 cores per node (2,048 cores) for native executions and up to 64 cores for simulations with MUSA (16,386 cores).

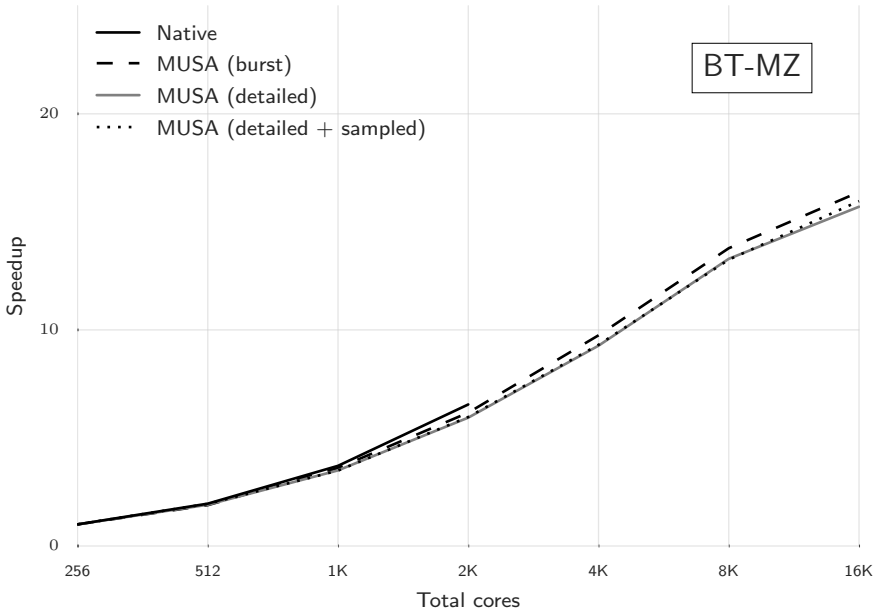



FIGURE 6.4: Performance estimations of BT-MZ with input class E for the entire application on 256 MPI ranks. Native runs with up to 8 cores per node (2,048 cores), and simulated runs with MUSA on up to 64 cores per node (16,384 cores).

These simulations allow us to identify scalability bottlenecks occurring for large core counts per node, a trend that continues to manifest.

Figure 6.4 shows speedup estimations for *BT-MZ*. Results with up to 8 cores per node (2,048 total) are validated against the native execution of the application, showing a good level of accuracy. With 8 cores per node, the parallel efficiency reaches 82% for the overall execution of the native application, and MUSA predicts the parallel efficiency with an error of less than 5% for all simulation modes.

When performing burst simulations with larger core counts, the parallel efficiency significantly degrades, reaching 26% for 64 cores (16× speedup). We analyze if task management is the limiting factor to scalability. To this end, we run the master thread with a significantly higher speed and observe no significant change in scalability. From this experiment we conclude that *BT-MZ* does not expose sufficient task parallelism to achieve a higher parallel efficiency at large core counts. One possible solution is to reduce task granularity and thus increase the number of task instances. As this approach also increases the task management overhead, it poses an interesting optimization problem. MUSA predicts similar scalability trends with all simulation modes because this application is not memory intensive, as stated in the previous subsection.

In conclusion, we identify that *BT-MZ* lacks task parallelism and thus shows limited scalability in executions with larger core counts per MPI rank. Scalability can be improved by reducing task granularity, but only if this does not increase the effort of task management to a point where it becomes the new limiting factor to

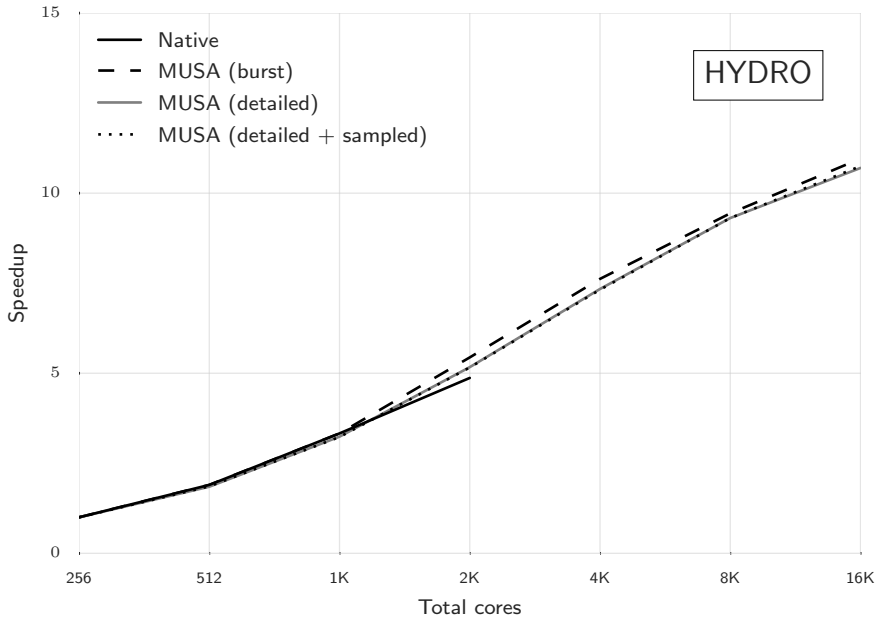



FIGURE 6.5: Performance estimations of HYDRO for the entire application on 256 MPI ranks. Native runs with up to 8 cores per node (2,048 cores), and simulated runs with MUSA on up to 64 cores per node (16,384 cores).

scalability.

Figure 6.5 shows speedup estimations for *HYDRO*. Results with up to 8 cores per node (2,048 total) are validated against the native execution of the application. For up to 8 cores, detailed simulation modes predict parallel efficiency with an error of less than 8%. For higher core counts, all simulation modes predict similar results. We attribute this to *HYDRO*'s low memory intensity.

As we increase the number of cores, parallel efficiency significantly degrades, reaching a value of only 17% at 64 cores per node. A significant percentage of parallel efficiency is lost due to communication (MPI) overheads. We find the parallel efficiency of the computation phases to be 31% when communication is ignored. Therefore, the computational part of the application has room for improvement. With the help of conventional performance analysis tools for MPI applications, we observe that the sequential part in each iteration is limiting the scalability of the application for core counts larger than 8. To avoid this limitation, the application needs to be restructured to reduce the amount of sequential computation.

Furthermore, for 32 and 64 cores per node the time devoted to task creation and scheduling limits the scalability of the application. There are multiple solutions to alleviate this problem. The first solution consists in increasing the granularity of the executed tasks, as this reduces the total number of task instances and thus the management effort. A second option is having multiple threads creating and scheduling tasks using nested parallelism. Finally, a third alternative consists in using hardware support for the runtime system [47].

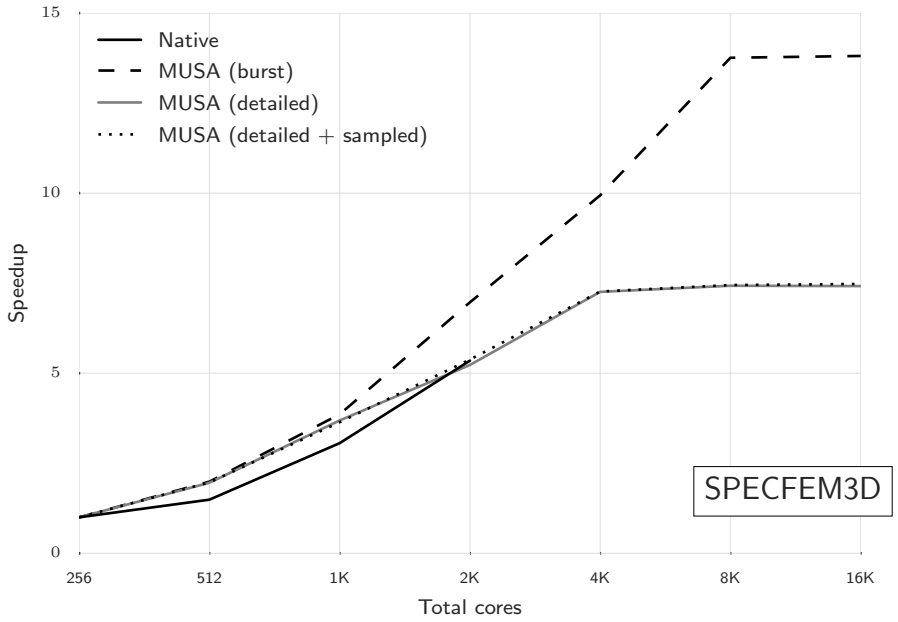



FIGURE 6.6: Performance estimations of SPECFEM3D for the entire application on 256 MPI ranks. Native runs with up to 8 cores per node (2,048 cores), and simulated runs with MUSA on up to 64 cores per node (16,384 cores).

Figure 6.6 shows speedup estimations for *SPECFEM3D*. Results for up to 8 cores per node (2,048 total) are compared to the native execution of the application. For 2 and 4 cores per node, we observe notable relative errors when comparing MUSA simulation modes and native execution. However, for 8 cores per node the detailed simulation modes predict parallel efficiency with an error of less than 3%. In addition, we observe that for core counts per node of 8 and more, performance estimations with burst and detailed mode differ significantly due to increasing off-chip memory contention, leading to performance overestimations in burst mode.

As we increase the core count in burst simulation mode, we observe that the application's scalability suddenly saturates from 32 to 64 cores per node. We find that this is because the number of task instances for this application is small, less than 200 per parallel region (see Table 6.1). Moreover, there are several task types that feature significantly different execution times, which eventually leads to severe load imbalance, limiting scalability. Since MUSA faithfully models task scheduling in burst mode, we correctly identify this bottleneck.

However, for detailed simulations we see that the performance actually saturates when moving from 16 to 32 cores per node. This is due to the combined effect of load imbalance and significant off-chip memory contention, which especially penalizes long running tasks that now execute for an even longer period of time, exacerbating load imbalance. With MUSA we are able to identify a bottleneck that manifests due to the combination of two factors, and gain insight on the performance penalty each factor imposes.

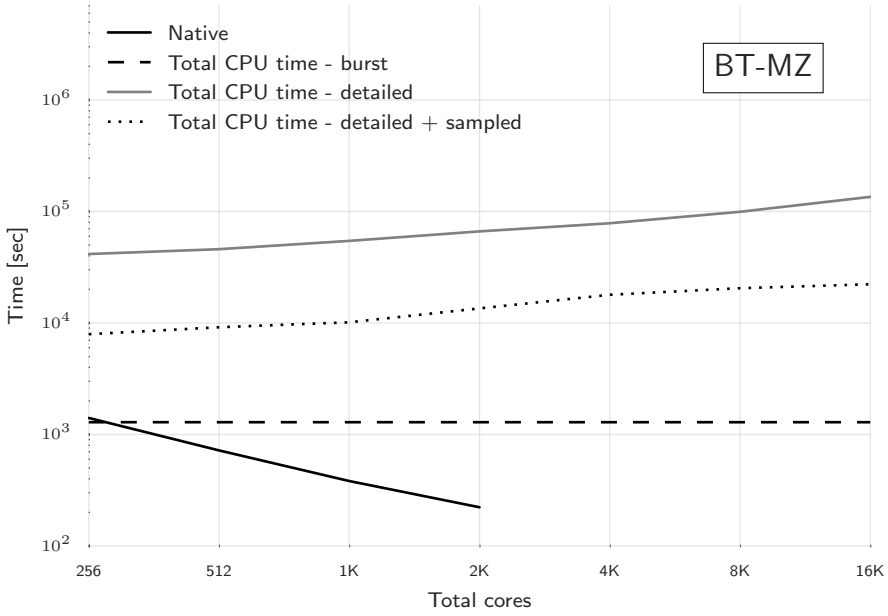



FIGURE 6.7: Total aggregated CPU time for MUSA simulations versus time-to-solution for native executions, *BT-MZ*.

### 6.4.6 Simulation Time Cost Analysis

Figure 6.7 shows the time required to run native and simulated executions for *BT-MZ* (input class E) with 256 ranks. We plot time-to-solution for native executions and total aggregated CPU time for simulated runs with MUSA. The total CPU time required for simulations in burst mode is nearly constant and comparable to the native execution with 1 thread per rank, as it uses pre-calculated task execution times. Speedup of sampled over detailed simulation remains constant, providing around one order of magnitude simulation time improvements. A sampled simulation for 16,384 cores requires less than 6 hours of total CPU time, while the native execution for 1 thread per rank takes about 24 minutes - only one order of magnitude of slowdown, even when considering sequential simulation.

Figure 6.8 shows the same data for *HYDRO* with 256 ranks. Again, the simulation time in burst mode is nearly independent from the number of simulated cores. A detailed simulation of *HYDRO* on 16,384 cores requires less than 3 hours. This time is reduced to less than an hour when performing sampled simulation. We observe that the speedup of sampled over detailed simulation decreases with increasing core counts. *HYDRO* has two computation phases per iteration. Therefore, architectural warmup and measuring of samples is performed twice per iteration. In addition, the number of tasks per computational phase is lower than in the case of *BT-MZ*. Both aforementioned effects hinder effective simulation sampling.

Figure 6.9 shows similar data for *SPECFEM3D* with 256 ranks. In this case we see that burst and detailed executions take a similar amount of time. This is because this application has a large number of iterations (i.e. 10,700). However, only one

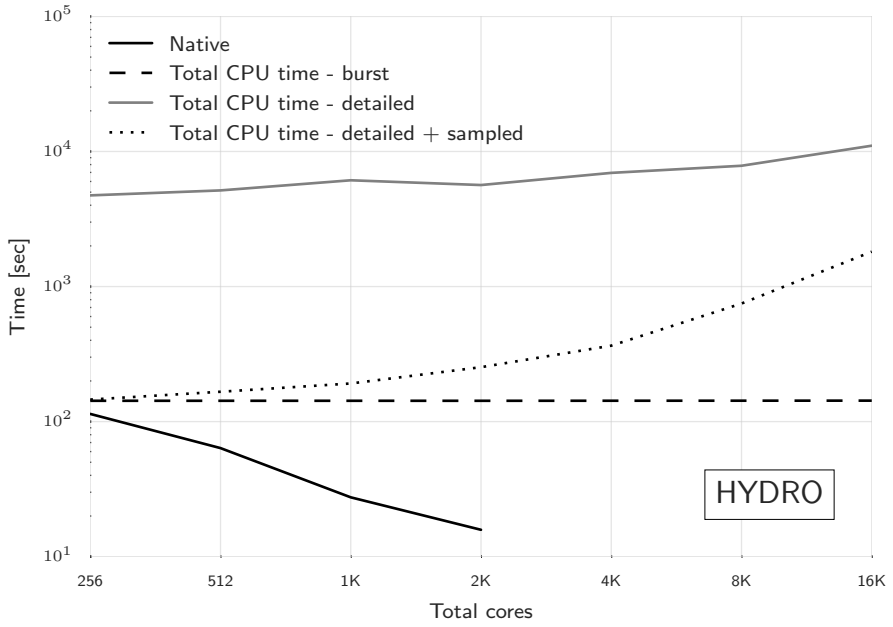



FIGURE 6.8: Total aggregated CPU time for MUSA simulations versus time-to-solution for native executions, *HYDRO*.

iteration is simulated in detailed mode. As a consequence, the time it takes to simulate the burst trace for the entire application is similar. Also note that sampling is not effective and its simulation time eventually converges to the detailed simulation time. The number of tasks per computational phase is so small that all of them are simulated in detail as samples. For 16,384 cores detailed simulation and native execution with 1 thread require 7.3 hours and 5.6 hours, respectively.

### 6.4.7 Design Space Exploration

We demonstrate the usefulnes of the MUSA infrastructure by performing a design space exploration study. Prior simulations focused on increasing the core count per node while leaving microarchitectural and memory parameters unchanged. Given that the trend to use commodity server processors is starting to change and that new technologies like die-stacked DRAM start to be available [109], we show how MUSA can aid to explore this vast design space with simulations using 16,384 cores - i.e. 256 MPI ranks and 64 cores per node - on *BT-MZ* with input class E, *HYDRO* and *SPECFEM3D*.

With this objective, we study the performance of these applications on three different multi-core architectures. The first system resembles a high-end server-class processor with a large reorder buffer and a three-level cache hierarchy, as found in traditional HPC environments. The second configuration is inspired by a low-power mobile platform. It has a smaller reorder buffer and only two levels of cache, as is typical for battery-powered mobile systems. The third configuration represents an emerging many-core chip with die-stacked DRAM, featuring medium cores and

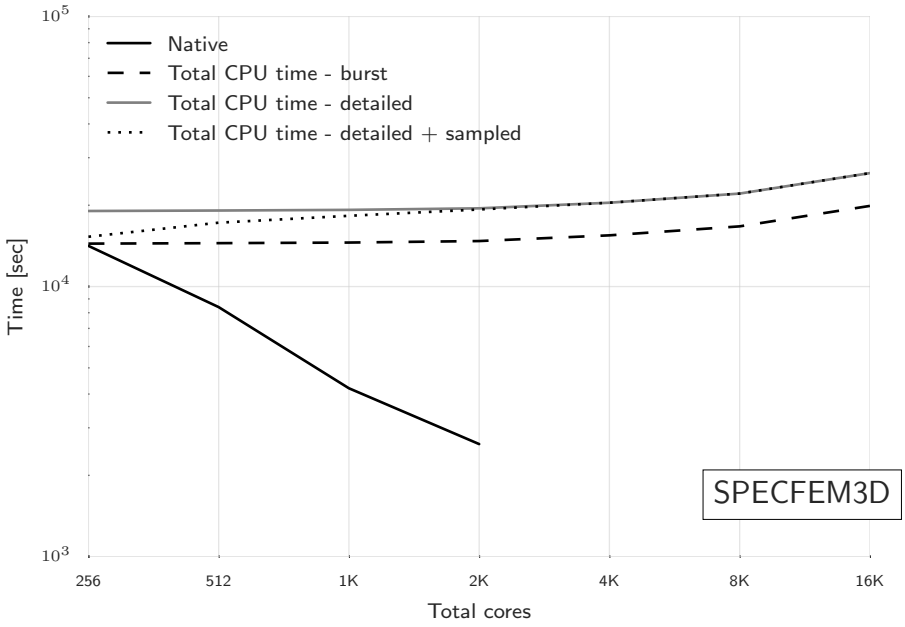



FIGURE 6.9: Total aggregated CPU time for MUSA simulations versus time-to-solution for native executions, *SPECFEM3D*.

TABLE 6.3: Architectural parameters of high-performance, low-power and die-stacked DRAM configurations for a 64 core processor.

| **Parameter** | **High-perf.** | **Low-power** | **Stacked DRAM** |
|---|---|---|---|
| ROB | 168 entries | 40 entries | 72 entries |
| Issue width | 1/2/4 | 1/2/4 | 1/2/4 |
| L1 cache | 32KB private<br>4 cycles<br>8-way | 32KB private<br>4 cycles<br>2-way | 32KB private<br>4 cycles<br>8-way |
| L2 cache | 256KB private<br>11 cycles<br>8-way | 8MB shared<br>21 cycles<br>16-way | 32MB shared<br>16 cycles<br>16-way |
| L3 cache | 128MB shared<br>28 cycles<br>20-way | none | none |
| DRAM | off-chip<br>4 channels<br>DDR3-1600 | off-chip<br>3 channels<br>DDR3-1600 | die-stacked<br>8 channels<br>DDR3-3200 |

moderate LLC capacity, but lower latency and higher bandwidth access to DRAM. Table 6.3 lists the key characteristics of the simulated architectures.

Figure 6.10 shows the predicted performance on these platforms for different issue width values of 1, 2, and 4 instructions per cycle. The reported speedup is normalized to an execution with one thread per rank using the high-performance configuration. The evaluated applications show very different behavior. *BT-MZ* benefits from running on a high-performance processor, achieving more than 35% additional performance compared to the speedup of the low-power processor for an issue width of 4. This compute intensive application favors the combination of a

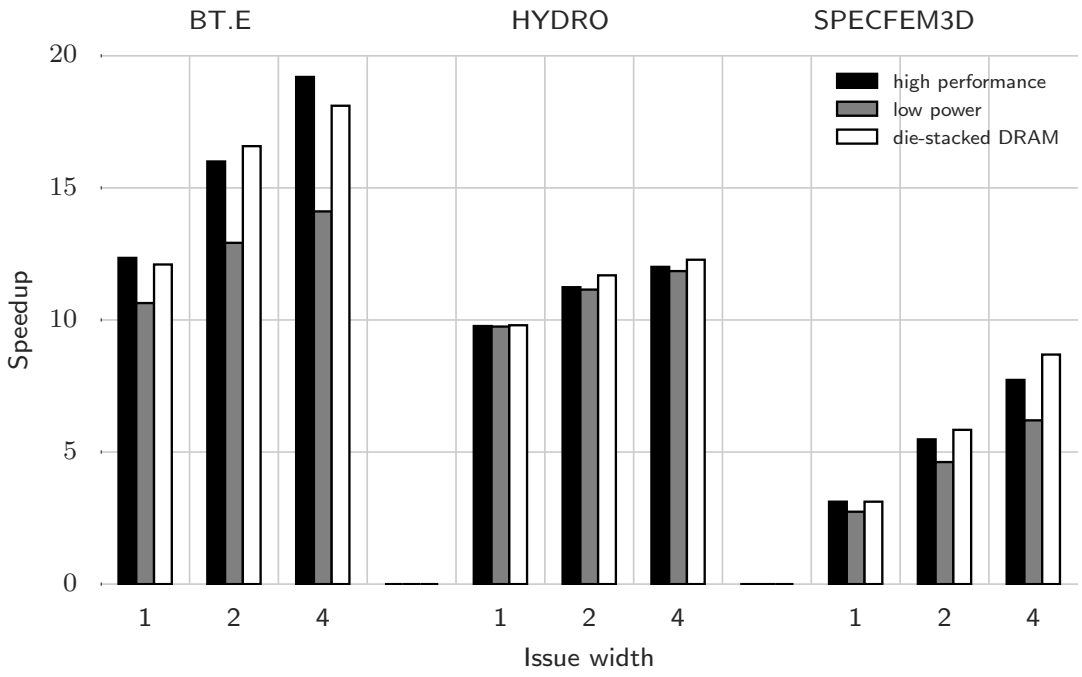



FIGURE 6.10: Design space exploration of BT-MZ, HYDRO and SPECFEM3D for different issue widths and processor profiles for 256 MPI ranks (16,384 cores).

large reorder buffer with quad-issue width, which also outperforms the die-stacked DRAM configuration that has a medium sized reorder buffer. A final observation is that, for the low-power configuration, increasing the issue width from 2 to 4 improves performance by merely 6%, while significantly increasing the complexity of the core.

In contrast, *HYDRO* shows a completely different behavior. The speedup achieved by the low-power processor nearly matches the speedup of the high-performance and die-stacked DRAM configurations. Since *HYDRO* has low memory intensity, deep cache hierarchies or low-latency and high-bandwidth DRAM memory does not improve performance significantly. Moreover, as explained in Section 6.4.5; existing factors that limit the scalability of the application, such as communication overheads and sequential code, hinder the performance of the aggressive cores. Furthermore, HYDRO benefits much less from an increased issue width - performance improves by less than 25% when increasing the issue width from 1 to 4 instructions per cycle. Thus, we conclude that the much simpler low-power architecture can deliver competitive performance for *HYDRO*.

Finally, for *SPECFEM3D* we observe that for issue widths of 2 and 4, the low-power configuration falls behind due to a less performing memory hierarchy. This application has a significant degree of memory contention. For this reason, the die-stacked DRAM configuration is able to outperform the high-performance configuration even though it features a less agressive core. However, the gains are not as

significant as one might expect. This is due to the fact that the performance is limited by severe load imbalance at the node level due to the small number of tasks per parallel region, as explained in Section 6.4.5. Nontheless, we conclude that die-stacked DRAM is beneficial over an agressive core design for *SPECFEM3D*.

## 6.5 Related Work

In this section, we review prior work on simulation of both shared and distributed memory machines as well as techniques to speed up simulation of parallel applications.

**Simulating distributed machines**: Prior work proposed simulation methodologies to evaluate the performance of large-scale parallel applications. Some proposals also employ a multi-level approach, combining different simulation layers. However, only a few evaluate scenarios with thousands of cores, but at the cost of not modeling microarchitectural details or system software interaction [40, 55, 126]. The other proposals evaluate lower core counts [1], while also lacking important features, e.g. detailed microarchitectural simulation [25], or support to capture operating system or runtime system interactions [51]. Finally, in other infrastructures each simulation requires a large computational effort due to the use of full system simulation [61] or the lack of sampling techniques [78], making them impractical for large-scale studies.

The usefulness of parametric models based on basic machine performance metrics and application characteristics has also been explored [11, 68]. These models are applied to understand the performance of current systems, to unveil bottlenecks, and to show where tuning efforts can be useful, but are tailored to specific applications.

**Simulating shared-memory systems**: Most simulation infrastructures at this level tend to be cycle-accurate to faithfully model the processing cores and the memory hierarchy. However, this level of detail comes at a significant slowdown, making simulations with more than a few tens or hundreds of cores impractical [14, 21, 103].

**Sampling techniques**: To reduce simulation time, statistical sampling is applied to identify a representative section of an application or even a synthetic trace, much shorter than the original one [24, 45, 74, 106, 121]. This representative section is then executed in a cycle-accurate simulator. However, the accuracy of these simulations is tied to the quality of the selected representative section of the application.

Finally, to further reduce simulation time and allow the simulation of larger multi-core processors, parallel simulators have been proposed [6, 21, 33, 83, 97, 103]. The main drawback of these proposals lies in the synchronization overhead. This

overhead can be reduced at the expense of sacrificing accuracy in the final results of the simulation.

## 6.6 Summary

In this Chapter we have introduced MUSA, a multi-level simulation approach that enables fast and accurate performance estimations of large-scale next-generation HPC machines. MUSA can model microarchitectural and runtime system effects by leveraging multi-level traces. These traces also allow for different simulation modes and execution replay to quickly extrapolate results of entire hybrid applications running on tens of thousands of cores.

MUSA has been validated using a production supercomputer with up to 2,048 cores showing high accuracy, with relative errors below 10% in the common case. For native codes that run for several minutes, MUSA allows detailed simulation of systems with more than ten thousand cores within a few hours of total aggregated CPU time. Our 16,384-core simulations revealed scalability bottlenecks in the evaluated applications that were easily identifiable using the simulation output trace and conventional performance analysis tools.

The main advantage of MUSA is that it provides results not only across known systems, but also for future systems not yet available on the market. Our design space exploration analysis provides useful insights on the different microarchitectural requirements of three applications to achieve good scalability, showing the potential MUSA offers in predicting the performance of applications on next-generation HPC machines.

# Chapter 7

# Conclusions

In this thesis, we present a study of execution time predictability of task-based programs. The results of this study are the motivation to develop TaskPoint, our sampled simulation methodology for task-based programs executed on shared-memory multi-core systems. Finally, we present MUSA, our multi-level simulation approach for hybrid applications. MUSA includes TaskPoint to speed up simulations at the shared-memory node level.

## 7.1 Execution Time Predictability of Task-Based Programs

Task-based programming models are a promising way to efficiently program future shared-memory systems with large core counts. In a task-based programming model, the programmer declares program parts as tasks, which are instantiated many times during the execution of the program. A runtime system calculates data dependencies between task instances. Task instances which have their dependencies fulfilled are scheduled to available execution threads.

In Chapter 4 we present an analysis of execution time predictability of task-based programs. To this end, we evaluate performance variability across different instances of the same task type and find that the naive assumption of regular performance across instances of the same task type is not always valid.

We show that accurate performance predictions can be derived from detailed performance information of a relatively small number of task instances. We present techniques to improve the accuracy of execution time predictions for task types with irregular performance. These techniques are based on linear interpolation and clustering. The execution time prediction error is reduced from more than 80% to less than 12% for input dependent cases and to less than 2% for task types exposing multiple classes of behavior.

## 7.2 Sampled Simulation of Task-Based Programs

Architectural simulation of future multi-core systems is becoming increasingly challenging. Due to the increasing total size of on-chip caches, larger workloads need to be simulated in order to meaningfully stress a design. Furthermore, the increasing core counts in multi-core designs require longer simulations in order to stress shared system resources and simulate interactions of different threads in a meaningful way.

Previous sampled simulation techniques for parallel programs rely on the assumption, that the sequence of useful instructions, i.e. the application's instructions excluding runtime system activity and synchronization, does not change across different executions of the application. Although those existing techniques have been proven to be accurate for statically scheduled fork-join based programs, they are not directly applicable to dynamically scheduled task-based parallel programs. In task-based programs, the execution order of task instances can change due to the dynamic scheduler of the runtime system.

In Chapter 5 we present TaskPoint, a methodology for sampled simulation of task-based parallel programs. Sampling units are identified based on the partitioning into tasks provided by the programmer. Between detailed simulation phases, we employ a novel fast-forward mechanism, which correctly reflects the different progress rates of task instances belonging to different task types and adapts to phase changes in the simulated application.

We improve the original TaskPoint methodology by automatically clustering task instances using BBVs and DBSCAN clustering. After creating a BBV for each task instance, DBSCAN clustering identifies clusters of task instances with similar behavior. This has two advantages: first, different task types can have task instances with similar behavior. Our improved approach merges these task instances into a single cluster, reducing the amount of detailed simulation required for sampled simulation. Second, a task type can have task instances with different classes of behavior. Our new approach also identifies these cases and clusters the task instances accordingly.

For some applications, clustering with DBSCAN results in clusters with large diameters, i.e. clusters containing task instances which are dissimilar, but connected by a chain of task instances similar to their respective neighbors. Our improved version of TaskPoint uses an analytical performance model to achieve accurate performance predictions in the aforementioned cases.

We assess TaskPoints generalization capability by using two radically different architectures to select sampling parameters and to run simulations. The evaluation results are satisfactory across a wide range of benchmarks, different numbers of simulated threads and different architecture models. The average simulation error of our model-based simulation mode ranges from 0.1% for 1 simulated thread to 1.3%

for 64 simulated threads. The simulation speedup ranges from 22.3×to 1,490×for thread counts of 64 and 1, respectively.

## 7.3 Multi-Level Simulation of Hybrid Programs

The process of designing future HPC systems is extremely challenging. The ever increasing system complexity, in terms of processors per node and nodes per system, makes architectural simulation of entire systems prohibitively time consuming. Furthermore, program execution on future systems is likely to be managed by system software, e.g. a runtime environment. A simulation methodology for future HPC systems ideally allows to perform detailed large-scale architectural simulations, while taking the effects of the system software into account.

In Chapter 6 of this thesis we introduce MUSA, a multi-level simulation approach for future HPC systems programmed with hybrid programming models which enables fast and accurate performance estimations of large-scale next-generation HPC machines. MUSA can model microarchitectural and runtime system effects by leveraging multi-level traces. These traces also allow for different simulation modes and execution replay to quickly extrapolate results of entire hybrid applications running on tens of thousands of cores.

MUSA has been validated using a production supercomputer with up to 2,048 cores showing high accuracy, with relative errors below 10% in the common case. For native codes that run for several minutes, MUSA allows detailed simulation of systems with more than ten thousand cores within a few hours of total aggregated CPU time. Our 16,384-core simulations revealed scalability bottlenecks in the evaluated applications that were easily identifiable using the simulation output trace and conventional performance analysis tools.

The main advantage of MUSA is that it provides results not only across known systems, but also for future systems not yet available on the market. Our design space exploration analysis provides useful insights on the different microarchitectural requirements of three applications to achieve good scalability, showing the potential MUSA offers in predicting the performance of applications on next-generation HPC machines.

# Chapter 8

# Future Work

## 8.1 Scheduling Task-Based Programs Using Execution Time Predictability

In our evaluation of execution time predictability of task-based programs in Chapter 4 we showed that execution time of task-based programs is predictable. In Chapter 5, we leverage this insight and propose TaskPoint, our sampled simulation methodology for task-based programs executed on multi-core systems.

We envision another potential application of the insights of this work in the field of dynamic scheduling of task instances in task-based programming models. In a task-based programming model, a runtime system schedules task instances which are ready for execution to available execution threads. The performance of each task instance, and thus the overall program performance, can depend on the exact schedule.

Scheduling task instances which share data closely after each other is typically benefitial in order to achieve maximum performance. If a consumer task instance is not yet ready, the optimal scheduling decision can be to schedule other task instances in the meanwhile, as long as this does not cause the data accessed by the consumer to be evicted from the shared last level cache [20, 31]. At the same time, it is desirable to not increase the length of the critical path of an application's task dependency graph [35, 36].

Knowing a task instance's execution time in advance has the potential to enable a scheduler to make informed scheduling decisions. We believe that it is worthwhile to investigate the potential of execution time predictability for improving dynamic task scheduling policies.

## 8.2 Sampled Simulation of Task-Based Programs

In Chapter 5 we present TaskPoint, a sampled simulation methodology for dynamically scheduled task-based programs. TaskPoint optionally uses clustering and analytical performance modeling to improve simulation error and speedup, especially for irregular applications.

As we show in our evaluation, our model-based simulation approach does not take into account the contention on the shared LLC of a simulated, multi-threaded system. In the future, we plan to fully integrate the analytical model with our simulation environment, allowing to get more accurate performance prediction by taking LLC contention into account.

Currently, model-based simulations with TaskPoint require detailed simulation of a small number of sample task instances. For the future, we plan to eliminate the need for detailed simulation. While we would still use a simulator to model the effects of the runtime environment, performance estimations would rely purely on analytical modeling. We are confident that this approach will achieve larger simulation speeds and, equally important, improve the scalability of simulation speed when increasing the number of simulated threads.

## 8.3 Multi-Level Simulation of Hybrid Programs

In Chapter 6 of this thesis we present MUSA, our multi-level simulation approach for hybrid systems. We validate MUSA and perform large-scale architectural simulations with up to 16,384 simulated cores. We also conduct a case study, in which we simulate the performance of several large-scale hybrid applications on different architectures, namely a state-of-the-art high-performance architecture, a low-power architecture and an architecture featuring die-stacked DRAM. We find that some applications, e.g. *BT-MZ*, benefit from being run on a system with aggressive out-of-order processors and are not very sensitive to the performance of the DRAM subsystem. Other applications, e.g. *SPECFEM3D*, clearly benefit from being run on a system with high-bandwidth, die-stacked DRAM. The design space exploration presented in this thesis is only an example to show the usefulness of MUSA. We envision a more thorough study of future architectures using MUSA.

Currently, MUSA can only simulate systems consisting of single-socket nodes. However, many current HPC systems consist of nodes containing two or more sockets. We see potential for future work in extending MUSA for it to support multi-socket nodes. This would allow to use MUSA to study the impact of how processor cores are distributed across different sockets on performance.

Our current implementation of MUSA cannot simulate communications overlapped with computation, i.e., before sending or receiving an MPI message, all threads of a rank need to synchronize. Due to the ever increasing number of processor cores in a single HPC system, this can be a limiting factor to performance. In the future, we would like to add support for simulating communications overlapped with computation. This would allow to study the performance benefits of a more asynchronous execution model allowed by less synchronizations of the simulated application.

# Appendix A

# Publications

## A.1 Conference Publications

- *"MUSA: A Multi-Level Simulation Approach for Next-Generation HPC Machines"*. **Thomas Grass**, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesús Labarta, Mateo Valero, Marc Casas, Miquel Moreto. Published in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2016 (SC16). Salt Lake City, Utah, United States of America. November 2016.
- *"TaskPoint: Sampled Simulation of Task-Based Programs"*. **Thomas Grass**, Alejandro Rico, Marc Casas, Miquel Moreto, Eduard Ayguadé. Published in Proceedings of the 2016 International Symposium on Performance Analysis of Systems and Software (ISPASS 2016). Uppsala, Sweden. March 2016.

## A.2 Journal Publications

- *"Sampled Simulation of Task-Based Programs"*. **Thomas Grass**, Germán Ceballos, Trevor Carlson, Alejandro Rico, Eduard Ayguadé, Miquel Moretó, Marc Casas. Under submission at IEEE Transactions on Computers (TC).

## A.3 Workshop Publications

- *"Evaluating Execution Time Predictability of Task-Based Programs on Multi-Core Processors"*. **Thomas Grass**, Alejandro Rico, Marc Casas, Miquel Moreto, Alex Ramirez. Published in Proceedings of Euro-Par 2014: Parallel Processing Workshops (MuCoCoS 2014). Porto, Portugal. August 2014.

## A.4 Poster Presentations

- *"Evaluating Execution Time Predictability of Task-Based Programs"*. **Thomas Grass**, Alejandro Rico, Miquel Moreto, Marc Casas, Alex Ramirez. Presented at Tenth

International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2014). Fiuggi, Italy. July 2014.

- *"Task Sampling: Computer Architecture Simulation in the Many-Core Era"*. **Thomas Grass**. Published in Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT 2013). Edinburgh, Scotland, United Kingdom. October 2013.

## A.5 Other Publications (Not as First Author)

- *"TaskInsight: Understanding Task Schedules Effects on Memory and Performance"*. Germán Ceballos, **Thomas Grass**, Andra Hugo, David Black-Schaffer. Published in Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2017). Austin, Texas, United States of America. February 2017.

- *"Characterizing Task Scheduling Performance Based on Data Reuse"*. Germán Ceballos, **Thomas Grass**, David Black-Schaffer, Andra Hugo. Published in Proceedings of the 9th Nordic Workshop on Multi-Core Computing (MCC 2016). Trondheim, Norway. November 2016.

- *"Evaluating the Effect of Last-Level Cache Sharing on Integrated GPU-CPU Systems with Heterogeneous Applications"*. Víctor García, Juan Gómez-Luna, **Thomas Grass**, Alejandro Rico, Eduard Ayguade, Antonio J. Peña. Published in Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC 2016). Providence, Rhode Island, United States of America. September 2016.

# Bibliography

[1] V. S. Adve. "POEMS: end-to-end performance design of large parallel adaptive computational systems". In: *IEEE Transactions on Software Engineering* 26.11 (2000), pp. 1027–1048.

[2] M. Al-Manasia and Z. Chaczko. "An Overview of Chip Multi-Processors Simulators Technology". In: *Progress in Systems Engineering*. Springer, 2015, pp. 877–884.

[3] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balaji, P. C. Diniz, A. Koniges, M. Snir, and S. R. Sachs. *ASCR Programming Challenges for Exascale Computing*. Tech. rep. U.S. Department of Energy, 2011.

[4] E. Anger, S. Yalamanchili, D. Dechev, G. Hendry, and J. Wilke. "Application modeling for scalable simulation of massively parallel systems". In: *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 2015, pp. 238–247.

[5] E. K. Ardestani and J. Renau. "ESESC: A fast multicore simulator using Time-Based Sampling". In: *Proceedings - International Symposium on High-Performance Computer Architecture*. IEEE, 2013, pp. 448–459.

[6] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. "COTSon: infrastructure for full system simulation". In: *ACM SIGOPS Operating Systems Review* 43.1 (2009), pp. 52–61.

[7] W. Aspray. "The Intel 4004 microprocessor: What constituted invention?" In: *IEEE Annals of the History of Computing* 19.3 (1997), pp. 4–15.

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. "The design of OpenMP tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 404–418.

[9] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. "The NAS Parallel Benchmarks". In: *International Journal of High Performance Computing Applications* 5.3 (1991), pp. 63–73.

[10] Barcelona Supercomputing Center. *Extrae User guide manual for version 2.3*. Tech. rep. 2012.

[11] 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: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2008, pp. 1–11.

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. "Exascale computing study: Technology challenges in achieving exascale systems". In: *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* 15 (2008).

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2008, pp. 72–81.

[14] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. Hill, D. A. D. A. Wood, B. Beckmann, G. Black, S. K. S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, A. Basil, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. Hill, and D. A. D. A. Wood. "The gem5 Simulator". In: *Computer Architecture News* 39.2 (2011), pp. 1–7.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: an efficient multithreaded runtime system". In: *ACM SIGPLAN Notices* 30.8 (1995), pp. 207–216.

[16] M. Breughe, S. Eyerman, and L. Eeckhout. "A mechanistic performance model for superscalar in-order processors". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2012, pp. 14–24.

[17] S. Browne, J. Dongarra, N. Garner, J. London, and P. Mucci. "A Portable Programming Interface for Performance Evaluation on Modern Processors". In: *International Journal of High Performance Computing Applications* 14.3 (2000), pp. 189–204.

[18] D. L. Bruening. "Efficient, Transparent and Comprehensive Runtime Code Manipulation". PhD Thesis. Massachusetts Institute of Technology, 2004.

[19] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[20] P. Caheny, M. Casas, M. Moretó, H. Gloaguen, M. Saintes, E. Ayguadé, J. Labarta, and M. Valero. "Reducing cache coherence traffic with hierarchical directory cache and NUMA-aware runtime scheduling". In: *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE. 2016, pp. 275–286.

[21] T. E. Carlson, W. Heirman, and L. Eeckhout. "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2011, p. 1.

[22] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. "An Evaluation of High-Level Mechanistic Core Models". In: *ACM Transactions on Architecture and Code Optimization* 11.3 (2014), Article No. 28.

[23] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. "BarrierPoint: Sampled Simulation of Multi-Threaded Applications". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2014, pp. 2–12.

[24] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. "Sampled simulation of Multi-Threaded Applications". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2013, pp. 2–12.

[25] L. Carrington, A. Snavely, X. Gao, and N. Wolter. "A Performance Prediction Framework for Scientific Applications". In: *International Conference on Computational Science*. Springer Berlin Heidelberg, 2003, pp. 926–935.

[26] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2899–2917.

[27] M. Casas, R. M. Badia, and J. Labarta. "Automatic Phase Detection and Structure Extraction of MPI Applications". In: *International Journal of High Performance Computing Applications* 24.3 (2010), pp. 335–360.

[28] M. Casas, R. Badia, and J. Labarta. "Automatic analysis of speedup of MPI applications". In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM. 2008, pp. 349–358.

[29] M. Casas, H. Servat, R. M. Badia, and J. Labarta. "Extracting the optimal sampling frequency of applications using spectral analysis". In: *Concurrency and Computation: Practice and Experience* 24.3 (2012), pp. 237–259.

[30] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, E. Ayguade, J. Labarta, and M. Valero.

"Runtime-Aware Architectures". In: *Euro-Par 2015: Parallel processing: 21st International Conference on Parallel and Distributed Computing*. Vol. 9233. 2015, pp. 16–27.

[31] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer. "TaskInsight: Understanding Task Schedules Effects on Memory and Performance". In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM Press, 2017, pp. 11–20.

[32] J. Chen, M. Annavaram, and M. Dubois. "SlackSim: A Platform for Parallel Simulations of CMPs on CMPs". In: *ACM SIGARCH Computer Architecture News* 37.2 (2009), pp. 20–29.

[33] 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 Annual International Symposium on Microarchitecture, MICRO* (2010), pp. 523–534.

[34] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators". In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (2007), pp. 249–261.

[35] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures". In: *Proceedings of the 29th International Conference on Supercomputing*. 2015, pp. 329–338.

[36] K. Chronaki, A. Rico, M. Casas, M. Moreto, R. Badia, E. Ayguade, J. Labarta, and M. Valero. "Task Scheduling Techniques for Asymmetric Multi-core Systems". In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2016), pp. 1–1.

[37] 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: *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*. 2011, pp. 664–675.

[38] T. Conte, M. Hirsch, and W.-M. Hwu. "Combining trace sampling with single pass methods for efficient cache simulation". In: *IEEE Transactions on Computers* 47.6 (1998), pp. 714–720.

[39] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.

[40] W. E. Denzel, Jian Li, P. Walker, and Yuho Jin. "A Framework for End-to-End Simulation of High-performance Computing Systems". In: *Simulation* 86.5-6 (2008), pp. 331–350.

[41] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. "Barcelona openMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openMP". In: *Proceedings of the International Conference on Parallel Processing*. 2009, pp. 124–131.

[42] 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". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[43] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. "BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation". In: *The Computer Journal* 48.4 (2005), pp. 451–459.

[44] L Eeckhout, R. Bell, B Stougie, K De Bosschere, and L. K. John. "Control flow modeling in statistical simulation for accurate and efficient processor design studies". In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. 2004, pp. 350–361.

[45] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. "Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox". In: *IEEE Micro* 23.5 (2003), pp. 26–38.

[46] D. Eklov and E. Hagersten. "StatStack: Efficient modeling of LRU caches". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2010, pp. 55–65.

[47] 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 Annual International Symposium on Microarchitecture, MICRO* (2010), pp. 89–100.

[48] D. Genbrugge, S. Eyerman, and L. Eeckhout. "Interval simulation: Raising the level of abstraction in architectural simulation". In: *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*. 2010, pp. 1–12.

[49] S. Girona, J. Labarta, and R. M. Badia. "Validation of Dimemas communication model for MPI collective operations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface* 1908 (2000), pp. 39–46.

[50] J. Gonzalez, J. Gimenez, and J. Labarta. "Automatic evaluation of the computation structure of parallel applications". In: *International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2009, pp. 138–145.

[51] J. Gonzalez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. "Simulating whole supercomputer applications". In: *IEEE Micro* 31.3 (2011), pp. 32–45.

[52] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. "Understanding the performance of sparse matrix-vector multiplication". In: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (2008), pp. 283–292.

[53] T. Grass, A. Rico, M. Casas, M. Moreto, and A. Ramirez. "Evaluating Execution Time Predictability of Task-Based Programs on Multi-Core Processors". In: *Euro-Par 2014: Parallel Processing Workshops*. 2014, pp. 218–229.

[54] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguad?? "TaskPoint: Sampled simulation of task-based programs". In: *International Symposium on Performance Analysis of Systems and Software*. 2016, pp. 296–306.

[55] E. Grobelny, D. Bueno, I. Troxel, a. D. George, and J. S. Vetter. "FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications". In: *Simulation* 83.10 (2007), pp. 721–745.

[56] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[57] T. R. Halfhill. "ARM's 64-Bit Makeover". In: *The Linley Group Newsletters* (2012).

[58] J. Haskins and K. Skadron. "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation". In: *International Symposium on Performance Analysis of Systems and Software*. IEEE, 2003, pp. 195–203.

[59] J. L. Henning. "SPEC CPU2000: Measuring CPU performance in the new millenium". In: *IEEE Computer* 33.7 (2000), pp. 28–35.

[60] J. P. Hoeflinger. "Extending OpenMP to Clusters". In: *Intel Corporation white paper* (2006).

[61] M. Hsieh, J. Meng, M. Levenhagen, K. Pedretti, A. Coskun, and A. Rodrigues. "SST + gem5 = A scalable simulation infrastructure for high performance computing". In: *Proceedings of the Fifth International Conference on Simulation Tools and Techniques*. 2012, pp. 196–201.

[62] W. C. Hsu, H. Chen, P. C. Yew, and D.-y. Chen. "On the predictability of program behavior using different input data sets". In: *Proceedings Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*. 2002, pp. 45–53.

[63] W.-m. Hwu and Y. N. Patt. "HPSm, a high performance restricted data flow architecture having minimal functionality". In: *ACM SIGARCH Computer Architecture News* 14.2 (1986), pp. 297–306.

[64] *Intel Xeon Processor E5-2699 v4*. https://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz. Accessed: 2017-05-22.

[65] K. E. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Bremer. "Recovering logical structure from Charm++ event traces". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), 49:1–49:12.

[66] L. V. Kale and S. Krishnan. "CHARM++: A portable concurrent object oriented system based on C++". In: *ACM SIGPLAN Notices* 28.10 (1993), pp. 91–108.

[67] T. Karkhanis and J. Smith. "A first-order superscalar processor model". In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. 2004, pp. 338–349.

[68] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. "Predictive performance and scalability modeling of a large-scale application". In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. 2001, pp. 37–37.

[69] Y. Kim, W. Yang, and O. Mutlu. "Ramulator: A fast and extensible DRAM simulator". In: *IEEE Computer Architecture Letters* 15.1 (2016), pp. 45–49.

[70] A. J. KleinOsowski and D. J. Lilja. "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research". In: *IEEE Computer Architecture Letters* 1.1 (2002), p. 7.

[71] A. J. KleinOsowski, J Flynn, N Meares, and D. J. Lilja. "Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research". In: *Workload characterization of emerging computer applications* (2001), pp. 83–100.

[72] D. Komatitsch and J. Tromp. "Introduction to the spectral element method for three-dimensional seismic wave propagation". In: *Geophysical Journal International* 139.3 (1999), pp. 806–822.

[73] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. "DiP: A parallel program development environment". In: *Proceedings of the Second International Euro-Par Conference on Parallel Processing*. April. 1996, pp. 665–674.

[74] T. Lafage and A. Seznec. "Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream". In: *Workload characterization of emerging computer applications*. Springer US, 2001, pp. 145–163.

[75] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays. *Porting and optimizing HYDRO to new platforms and programming paradigms-lessons learnt*. 2012.

[76] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic linear algebra subprograms for fortran usage". In: *ACM Transactions on Mathematical Software* 5.3 (1979), pp. 308–323.

[77] K. Lee, S. Evans, and S. Cho. "Accurately approximating superscalar processor performance from traces". In: *International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 238–248.

[78] E. A. León, R. Riesen, A. B. Maccabe, and P. G. Bridges. "Instruction-level simulation of a cluster at scale". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, p. 1.

[79] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. "Work stealing and persistence-based load balancers for iterative overdecomposed applications". In: *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. 2012, pp. 137–148.

[80] C.-K. Luk, B. C. Ed, F. C. G. Hi, E. D. Q. Rs, A Tu, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. "Pin: Building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Vol. 40. 6. 2005, p. 190.

[81] J. Macqueen. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. 1967, pp. 281–297.

[82] Message Passing Interface Forum. *MPI: A message-passing interface standard*. 2012.

[83] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. "Graphite: A distributed parallel simulator for multicores". In: *Proceedings of the Sixteenth International Symposium on High-Performance Computer Architecture*. 2010, pp. 1–12.

[84] G. E. Moore. "Cramming more components onto integrated circuits". In: *Proceedings Of The IEEE* 86.1 (1965), pp. 82–85.

[85] M.Oskin, F. T. Chong, and M Farrens. "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs". In: *Proceedings of the 27th annual international symposium on Computer architecture*. 2000, pp. 71–82.

[86] 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". In: *IEEE Concurrency* 8.4 (2000), pp. 12–20.

[87] S. Nussbaum and J. Smith. "Modeling superscalar processors via statistical simulation". In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. 2001, pp. 15–24.

[88] S. L. Olivier, B. R. De Supinski, M. Schulz, and J. F. Prins. "Characterizing and mitigating work time inflation in task parallel programs". In: *Scientific Programming* 21.3-4 (2013), pp. 123–136.

[89] OpenMP Architecture Review Board. *OpenMP application program interface version 4.0*. Tech. rep. 2013.

[90] E. Perelman, G. Hamerly, and B. Calder. "Picking statistically valid and early simulation points". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2003, pp. 244–255.

[91] S. D. Pestel, S. Eyerman, and L. Eeckhout. "Micro-Architecture Independent Branch Behavior Characterization". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2015, pp. 135–144.

[92] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes". In: *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 2009, pp. 427–436.

[93] 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*. November. 2013, pp. 464–468.

[94] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. "Supercomputing with commodity CPUs: Are Mobile SoCs Ready for HPC?" In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.

[95] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. "The low power architecture approach towards exascale computing". In: *Journal of Computational Science* 4.6 (2013), pp. 439–443.

[96] J. Reinders. *Intel Thread Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Oreilly Media, Inc., 2007.

[97] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas. "HORNET: A cycle-level multicore simulator". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.6 (2012), pp. 890–903.

[98] A. Rico, A. Ramirez, and M. Valero. "Available task-level parallelism on the Cell BE". In: *Scientific Programming* 17.1-2 (2009), pp. 59–76.

[99] 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". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (2012), pp. 1–20.

[100] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. "Trace-driven simulation of multithreaded applications". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011, pp. 87–96.

[101] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R Risen, J. Cook, P. Rosenfeld, E. Cooper-Balls, et al. "The structural simulation toolkit". In: *ACM SIGMETRICS Performance Evaluation Review* 38.4 (2011), p. 37.

[102] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAMSim2: A cycle accurate memory system simulator". In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[103] D. Sanchez and C. Kozyrakis. "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems". In: *Proceedings of the International Symposium on Computer Architecture*. 2013, pp. 475–486.

[104] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 183–192.

[105] D. Schmidl, P. Philippen, D. Lorenz, C. Rössel, M. Geimer, D. An Mey, B. Mohr, and F. Wolf. "Performance analysis techniques for task-based OpenMP applications". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7312 LNCS.01 (2012), pp. 196–209.

[106] T. Sherwood, E. Perelman, and B. Calder. "Basic block distribution analysis to find periodic behavior and simulation points in applications". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2001, pp. 3–14.

[107] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically characterizing large scale program behavior". In: *ACM SIGOPS Operating Systems Review* 36.5 (2002), pp. 45–57.

[108] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. "A framework for performance modeling and prediction". In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. 2002, pp. 1–17.

[109] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (2016), pp. 34–46.

[110] G. Southern and J. Renau. "Analysis of PARSEC workload scalability". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. 2016, pp. 133–142.

[111] 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: *Proceedings of the 47th ACM/IEEE Design Automation Conference*. 2010, pp. 463–468.

[112] R. Tessier. "Cosmological hydrodynamics with adaptive mesh refinement-A new high resolution code called RAMSES". In: *Astronomy & Astrophysics* 385.1 (2002), pp. 337–364.

[113] *TOP500 Supercomputer Sites*. URL: https://www.top500.org/ (visited on 03/02/2017).

[114] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. "Runtime-Aware Architectures: A First Approach". In: *Supercomputing frontiers and innovations* 1.1 (2014), pp. 29–44.

[115] S. Van Den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. "Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics". In: *IEEE Transactions on Computers* 65.12 (2016), pp. 3537–3551.

[116] R. F. Van der Wijngaart and H. Jin. *NAS Parallel Benchmarks, Multi-Zone Versions*. Tech. rep. 2003.

[117] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. "SimFlex: Statistical Sampling of Computer System Simulation". In: *IEEE Micro* 26.4 (2006), pp. 18–31.

[118] 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*. Vol. 33. 1. 2005, pp. 408–409.

[119] S. White. "The AMD Opteron Seattle: A 64b ARM Dense Server Processor". In: *Hot Chips* (2014).

[120] T. Wiegand. "Overview of the H. 264/AVC video coding standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), pp. 560 – 576.

[121] 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*. 2003, pp. 84–95.

[122] S. Yoon and A. Jameson. "Lower-upper symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations". In: *AIAA journal* 26.9 (1988), pp. 1025–1026.

[123] M. T. Yourst. "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2007, pp. 23–34.

[124] Yue Luo, L. John, and L. Eeckhout. "Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation". In: *16th Symposium on Computer Architecture and High Performance Computing*. IEEE, 2004, pp. 10–17.

[125] C. Zhang. "Mars: A 64-core ARMv8 processor". In: *Proceedings of the 2015 IEEE Hot Chips Symposium*. 2016.

[126] G. Zheng, G. Gupta, E. Bohm, I. Dooley, and L. V. Kale. "Simulating large scale parallel applications using statistical models for sequential execution blocks". In: *Proceedings of the International Conference on Parallel and Distributed Systems*. 2010, pp. 221–228.