HPC memory systems: Implications on system simulation and check-pointing

Rommel Sánchez Verdejo

Supervisor: Dr. Petar Radojković

Advisor: Dr. Eduard Ayguadé

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

Barcelona, 2021

From the past and to the future,   for my family friends and colleagues.

Acknowledgements

In the first place, I would like to thank my supervisor, Petar Radojković, for the trust he deposited on me to become member of the memory systems for High Performance Computing (HPC) group; his detail-oriented guidance, support, and honest feedback had complemented my professional growth since our first conversations up to the complete realization of this work.

To the Barcelona Supercomputing Center (BSC), for creating the world-class research institution that enable all of its members to perform at their best.

To the Universitat Politècnica de Catalunya (UPC) and the efforts that had established the bond with BSC. Particularly, to Eduard Ayguadé for keeping those bonds alive along with his strong supervision in all of our work as Ph.D. students. To Joana Munuera and Nuria Sirvent, who year to year, help us to clear all the paperwork and the administrative tasks.

To Leonardo Bautista-Gómez, for the opportunity to work closely under his advise extending the Fault Tolerant Interface (FTI) library so that I could follow my research project.

To the members of my pre-defense committee: Adrià Armejach, Eduardo Quiñones, and Lluc Álvarez who amid the COVID-19 pandemic, pushed me further to deliver the best out of this thesis.

To my BSC Ph.D. peers: Kazzi Asifuzzaman, Milan Radulovic, Darko Živanović, and David Zaragoza who helped me at different points and levels on the road of the Ph.D. formation. To Kallia Kronacli, Rajiv Nishtala, Sicong Zhuang, Renan Fisher, Tomasz Patejko and Isaac Boixaderas for their friendship along the way.

To all the authors in the free software community that have written millions of lines of code and documentation that we all use in our daily basis to complete our goals for work and fun. To finish with the professional acknowledgements, to all the authors in the indexed engineering and scientific databases that allows knowledge to keep spreading.

On the personal side, the first and most important person I have to acknowledge is to my mother, María del Carmen Yolanda Verdejo Aguilar, for whose infinite love and support had allowed me reach this far.

To the extended members of my family: Alfonso, Félix and Aarón Verdejo for their extensive support throughout my lifetime. To my father, who taught me that among humans, reason is always stronger than physical strength.

To my friends, Eduardo Ruiz Duarte and Omar Lara Salazar, whose (un)human and philosophical advises, had being there for some time now.

To Elizabeth Lozano Lozano, who had showed me the true value of honesty and love.

To Eria López (Siri en el exilio), her long-standing friendship had always shown me a different perspective from my natural approach. Also, to demonstrate to me that with tight discipline, goals are achieved, despite the injuries along the way.

To Szófia Bendefy for her long standing love and support through the last year of this process.

To the Spanish Government, the Generalitat de Catalunya, the province of Barcelona and the Sant Just Desvern Ajuntament to provide me with a wonderful experience living abroad.

Part of this work was supported by the collaboration agreement between Samsung Electronics Co., Ltd. and the Barcelona Supercomputing Center (BSC); Spanish Ministry of Science and Technology (project TIN2015-65316-P); Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272); the Severo Ochoa Programme (SEV-2015-0493) of the Spanish Government.

Abstract

The memory system is a significant contributor for most of the current challenges in computer architecture: application performance bottlenecks, and operational costs in large data-centers as HPC supercomputers. With the advent of emerging memory technologies, the exploration for novel designs on the memory hierarchy for HPC systems is an open invitation for computer architecture researchers to improve and optimize current designs and deployments. System simulation is the preferred approach to perform architectural explorations due to the low cost to prototype hardware systems, acceptable performance estimates, and accurate energy consumption predictions. Despite the broad presence and extensive usage of system simulators, their validation is not standardized; either because the main purpose of the simulator is not meant to mimic real hardware, or because the design assumptions are too narrow on a particular computer architecture topic.

This thesis provides the first steps for a systematic methodology to validate system simulators when compared to real systems. We unveil real-machine's micro-architectural parameters through a set of specially crafted micro-benchmarks. The unveiled parameters are used to upgrade the simulation infrastructure in order to obtain higher accuracy in the simulation domain. Next, to evaluate the accuracy on the simulation domain with respect to the real machine, we propose the *retirement factor*, an extension to a well-known application's performance methodology. Our proposal provides with a new metric to measure the impact of simulator's parameter-tuning when looking for the most accurate configuration for the simulator's parameters. We further present the *delay queue*, a modification to the memory controller that imposes a configurable delay for all memory transactions that reach the main memory devices. Evaluated using the *retirement factor*, the *delay queue* allows to identify the sources of deviations between the simulator infrastructure and the real system.

Memory accesses directly affect application performance, both in the real-world machine as well as in the simulation accuracy. From single-read access to a unique memory location up to simultaneous read/write operations to a single or multiple memory locations, HPC applications memory usage differs from workload to workload. A property that allows to glimpse on the application's memory usage is the workload's memory footprint. In this work, we found a link between HPC workload's memory footprint and simulation performance.

Actual trends on HPC data-center memory deployments and current HPC application's memory footprint lead us to envision an opportunity for emerging memory technologies: to include them as part of the reliability support on HPC systems. Emerging memory technologies such as 3D-stacked DRAM are getting deployed in current HPC systems but in limited quantities in comparison with standard DRAM storage; therefore, they are suitable for low memory footprint HPC applications. We exploit and evaluate this characteristic enabling a Checkpoint-Restart (CR) library to support a heterogeneous memory system deployed with an emerging memory technology. Our implementation imposes negligible overhead while offering a simple interface to allocate, manage, and migrate data sets between heterogeneous memory systems. Moreover, we showed that the usage of an emerging memory technology it is not a direct solution to performance bottlenecks; correct data placement and crafted code implementation are critical when comes to obtain the best computing performance.

Overall, this thesis provides a technique for validating main memory system simulators when integrated in a simulation infrastructure and compared to real systems. In addition, we explored a link between the workload's memory footprint and simulation performance on current HPC workloads. Finally, we enabled low memory footprint HPC applications with resilience support while transparently profiting from the usage of emerging memory deployments.

Contents

CHAPTER 1

Introduction

Supercomputers, also known as High Performance Computing (HPC) systems, had become a technology pillar of our society's lifestyle. Governments and enterprises through public and private funded institutions, make available cutting-edge computing systems to a multi-disciplinary group of researchers to solve the most challenging defiances of humanity. For example, to find appropriate treatments and a possible vaccine for the COVID-19 throughout the SARS-CoV-2 outbreak in Europe, researchers from different scientific fields requested High Performance Computing (HPC) resources that were quickly allocated from the European Union representatives [42].

A HPC system comprises large clusters of thousands of single-computers connected through high-speed networks. Each one of these single-computers has a computing performance peak, which combined adds to a greater computing power. The preferred metric to measure computing power is Floating-point Operations per Second (FLOPS). During the 2010's decade, the performance of HPC systems went from a few peta FLOPS (PFLOPS: 10^{15} FLOPS) up to hundreds of PFLOPS: as per June 2020, the fastest supercomputer peaks at 415.5 PFLOPS [40]. It is expected that in 2021 the HPC road-map will meet a tangible milestone: the power-on of the first exascale computer [5][1]. An exascale supercomputer is a system that could compute at least a single exaFLOPS (EFLOPS): 10^{18} FLOPS. To reach the global peak computing power, every processor in the supercomputer must maximize its throughput by minimizing idle time. Code and data must be delivered to the processor at the right moment in the right shape. Bringing and returning data into and out of the CPUs cores are challenging tasks that are constantly revisited by the research community.

[1]There is a chance for China's Tianhe-3 supercomputer will be powered-on in late 2020. On July 2020, China's National University of Defense Technology (NUDT), the research institution that will host the supercomputer, have not disclosed any further information.

From the abstraction of an infinite tape in the formal definition of a Turing Machine up to today's banks of electronic memory cells, the memory subsystem is an inherent piece of a computing system. Nowadays, the memory subsystem comprises a multi-level memory hierarchy where the closest levels to the CPU have faster access time than the subsequent levels. In fact, CPU silicon vendors built the first levels of the memory hierarchy within the same silicon package; these levels are known as *memory caches*. The memory that is found outside the silicon package is known as *main memory*.

The different access times on the memory hierarchy generally translates to idle times for the processor. As the manufacturing processes had evolved, the processor could compute data faster than what the memory subsystem could load or store. This behavior was first named the *memory wall* [145] and studied in subsequent years [58]. Even with CPU micro-architectural techniques such as CPU's Out of Order (OoO) execution or data-prefetching as attempts to minimize processor's idle time, sometimes the processor must stall its execution until the memory transactions completes, making the memory subsystem a major contributor to bottlenecks for computing performance.

1.1 Current challenges for Memory Systems in HPC

Throughout the last 40 years, in a combination of cost and performance [77, 63], the predominant technology used for main memory is DRAM. Unfortunately, application requirements are pushing DRAM to its physical limits: as memory capacity rises, the energy consumption also increases; the manufacturing miniaturization process affects reliability and therefore security; and lastly, the DRAM protocol imposes bandwidth limitations.

Memory capacity: The trend to have more data-intensive applications in the HPC data-centers is growing [153, 118, 25]. To run such applications, larger amounts of memory are required to be physically installed in the computing node. Due to the internal structure of DRAM, which includes a capacitor that must be constantly refreshed even when the memory cells are not used, the main memory is responsible for an approximate two-thirds of the power consumption of a computing system [23]. This represents a major challenge for scalability with respect to power efficiency in HPC data centers: the more DRAM devices, the higher the power consumption will be.

Memory reliability and security: In an attempt to reduce the number of main memory devices in a system, DRAM manufacturers have increased the density of the memory cells within the same device; an approach that has strong consequences:

uncorrected error rates in DRAM devices increase when manufacturing technology scales down [150]. Researchers have analyzed the impact of the miniaturization process finding serious implications for data integrity. Particularly, Rowhammer [84] is a technique that exploits the electrical characteristics of DRAM devices leading to critical system failures, or even computer security breaches [109]. Even with the presence of Error Correcting Codes (ECC) logic and software-counter measures in HPC computing systems [60], reliability in DRAM devices is still an open challenge.

Memory bandwidth: The inclusion of several cores into the same processing unit allows HPC applications to exploit parallel processing; multiple cores might request for several memory transactions at the same time. The amount of in-flight requests in the memory hierarchy stresses the cache coherency protocol increasing the latency per memory request. Each one of the processing cores within a processor is generally designed so that their first and second levels of memory caches are not shared with the other cores. The capacity of such caches holds from some KiB up to a few MiB. Moreover, in HPC systems, a third level of cache (commonly referred as Last Level Cache (LLC)), is found to be shared among all cores in the same processor package with capacities that hold up to tens of MiB. In multi-threaded applications, the Operating System (OS) schedules the execution of threads trough context switching. If the HPC application wisely exploits memory locality and memory level parallelism, the theoretical maximum bandwidth for DDR3 and DDR4 might be quickly achieved while the CPU cores could still have idle cycles due to in-flight memory transactions.

Software interfaces: Furthermore, as the emerging memory technologies continue to evolve, so must the software interfaces. A common rule from OS designers is to provide the application developer with hardware-abstraction layers that require little or no understanding of the underlying hardware. Unfortunately, for emerging heterogeneous memory technologies, this is not always the case. Most HPC applications are not prepared to support emerging memory technologies integration. Although some important efforts have been made to bring automatic assignment for memory resources to the applications, developers must know the low-level details of the memory hierarchy to exploit emerging memory technologies; in most cases, this is a specialized knowledge that ends up in a computer programmer specialist coding for this purpose.

1.2 Thesis Contributions

System simulation is the preferred option for computer architecture researchers to propose and evaluate novel designs to solve memory hierarchy challenges. This thesis analyzes the implications of system simulation for memory exploration in HPC systems, and proposes a methodology for their validation in regards of real systems. Moreover, we analyzed the performance impact of enabling HPC applications to use an emerging memory system.

1.2.1 System Simulation for Memory Exploration

Nowadays, system simulators require the modeling of sophisticated features found in real-world computing platforms such as multi-core CPUs, different memory devices and, the interconnects which couple them together. In this work we describe a methodology to validate system simulators in regards to real-world hardware so that results are suitable for memory exploration.

Using a real machine as the target system to mimic in simulation, we proposed a set of micro-benchmarks to discover micro-architectural parameters on both, the CPU and the memory hierarchy. Then, we integrated the obtained parameters on the real machine into the simulator infrastructure so that the simulator infrastructure behaves accordingly. After an initial evaluation, we discovered a discrepancy on memory simulation, particularly on workloads with high memory usage. In order to locate the source of the differences between the real machine and the simulator infrastructure, we proposed an architectural modification to the memory controller that inserts a delay of the memory transactions that goes to main memory. To identify the impacts of such change, we further proposed an extension to a well-known Cycle Per Instruction (CPI) stack analysis to evaluate different configuration scenarios. Finally, we analyzed the effects of hardware prefetching and address translation. By analyzing the address translation and virtual memory usage, our results pushed us to further explore the memory footprint of HPC applications where we found a link with simulation performance.

1.2.2 Opportunities for Emerging Memory Technologies

One way to overcome hardware reliability problems in HPC systems is done through Checkpoint-Restart (CR). CR is a technique used in current HPC applications where application's data status is saved into a secondary storage so that in the event of a failure, the computation performed up to that moment along with current application's status is not completely lost.

Emerging memory technologies bring improvements to the memory hierarchy either by decreasing access times, reducing power consumption, or some advances in the manufacturing process. Although they are still deployed with in small capacity configurations making their usage suitable for low memory footprint HPC workloads. Given these conditions, we envision an opportunity for these workloads to profit from actual emerging memory technologies such as Multi-Channel DRAM (MCDRAM) integrations found in some HPC data centers. To achieve this goal, we enabled a software CR library to make usage of different memory technologies. If the memory footprint of a given variable within the scope of the HPC application is compact enough to fit into the current deployment of an emerging memory technology, we enable the HPC application with the reliability support while transparently profiting from the usage of such emerging memory technology.

1.3 Thesis Organization

This thesis is comprised by this introduction, 7 chapters, an appendix, and an overall conclusion with the following structure:

- Chapter 2 describes the basic definitions and requirements for the simulation infrastructure. Moreover, introduces the state-of-the-art projects that we considered relevant for this research.
- In Chapter 3, the experimental methodology is described along with a description of the micro-architectural models for the real targeted HPC system as well as the for the two system simulators we used in this work is presented.
- In Chapter 4, the design assumptions and the challenges of the proposed micro-benchmarks for parameter-discovery in the real system CPU and the memory hierarchy are described. The Chapter elaborates on the execution environment for the micro-benchmarks in both systems (real and simulated). Lastly, the Chapter describes the formal mechanism to extract experimental measurements.
- In Chapter 5, a classical approach to validate system simulators in the state-of-the-art is presented. The results are discussed.
- In Chapter 6, we propose the usage of the Top-Down methodology for micro-architectural comparison. Furthermore, we extend the methodology to allow us

measure the differences among the real system and the simulator infrastructure. This extension is named the retirement factor.

- In Chapter 7, we propose an architectural modification to the memory controller to mitigate gaps in memory simulation. We further stretch-out our retirement factor for two purposes: 1. we used our memory simulation mitigation to locate the source of differences between the systems under test, and 2. to measure the impact of hardware prefetchers and address translation in the real system when compared to the simulation infrastructure. Lastly, we found a relation between workload's memory footprint and simulation performance.
- In Chapter 8, we conduct a case-study for low-memory footprint applications to use heterogeneous memory systems while transparently enabling them with reliability support.
- In Chapter 9, the conclusions and further directions of this work are presented.
- Finally, Appendix A extends Chapter 4 with a table that includes the upgraded instruction latencies on the CPU simulator.

CHAPTER 2

State Of The Art

In the last years, the trend to propose new designs in computer architecture is steeply changing. The tendency had been known as a *new golden age in computer architecture* [64]. In the previous decades, the hardware tech giants were the only players in the proposal's stage. Year after year, Independent Hardware Vendors (IHVs) position their latest products to the market so that everyone would have to consume their designs. For computer architecture researchers, it meant a spiral where a product is reviewed extensively, and new proposals arose to extend commercial designs. Unfortunately, industrial hardware designs are not publicly available; researchers had to find an open mechanism to reproduce results from actual hardware. Such restrictions had driven researchers to work in system simulators that enable the community to test and evaluate their designs, comparing them with real-world computers.

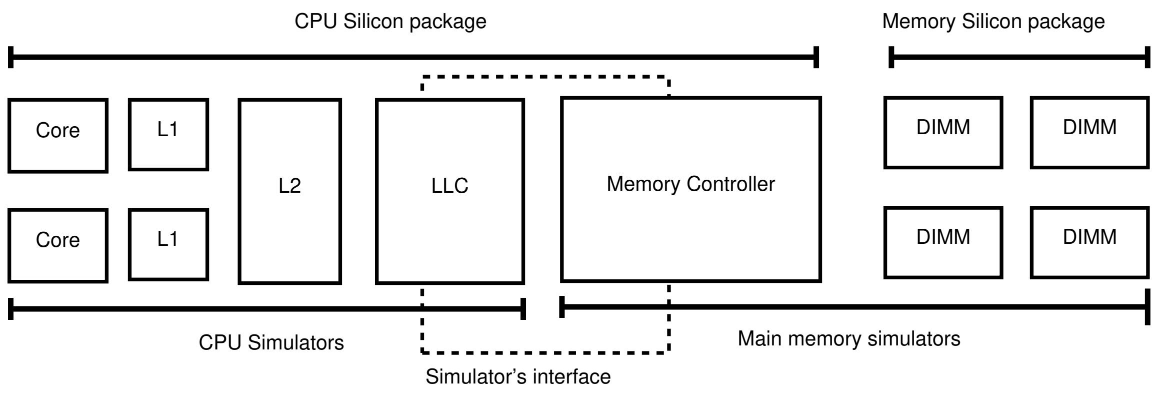


Figure 2.1 When CPU and memory simulators are coupled, the timings of the memory request between the LLC and the memory controller could be easily overlooked or misinterpreted.

Figure 2.1 depicts two major domains of system simulators: CPU system simulators include micro-architectural logic along with its memory caches while memory system simulators integrate the memory controller logic and up to the passive memory devices (the DRAM DIMMs). The significant difference between the simulator domain and the real domain is the memory controller's location. To the best of our knowledge, all current HPC systems integrate the memory controller into the same silicon package as the CPU. The reason relies in the physical improvements to decrease the latency imposed by current DRAM technologies. In the simulation domain, the physical constraints are not always included in the simulated model.

As the exascale era is getting closer [5], cycle-accurate simulators are hitting a wall in simulation time. Researchers showed that a full execution of a single benchmark of the SPEC CPU2006, takes over 1 year in the most detailed configuration of the `gem5` CPU simulator [21] and, over 1 week in the most simplified configuration possible. In contrast, the same workload execution in a native system consumes less than 1 hour [126]. New strategies to speed up CPU and main memory simulation are being proposed [85, 90, 30, 125] in an exchange of cycle-accuracy simulation [21, 122]. Though, every benefit comes with a price; when cycle-accurate simulation is combined with approximate simulation, results must be taken with caution [91, 138].

2.1 Background

A system simulator [1] is generally a software or software-hardware (hybrid) solution that mimics to some extent a hardware design. It provides the researcher with tools to propose changes to the baseline design or extend the characteristics towards a new solution.

According to state-of-the-art definitions [43, 8], system simulators could be categorized as follows:

- Detail of Simulation
  - **Functional simulators**: emulate the behavior of the simulated processor, generally not considering the micro-architectural characteristics. For this reason, the execution is faster than any other simulation.

[1]For the scope of this thesis, we will use the term *computer simulator* or *system simulator* to refer a software entity that provides an evaluation of computer architecture programmed model.

- **Timing simulators**: they consider the micro-architectural states of the processor. According to the details provided in the internal states, they can be sub-categorized as:
  - **Cycle-level**: mimics the micro-architecture at cycle level; although they are not as accurate as a Register-Transfer Level (RTL) cycle-accurate simulation, they could provide information on some of the underlying components of a micro-architecture.
  - **Event-driven** and **Interval-driven**: targets simulation by events instead of cycles. Both simulators could break the simulation in windows of events like cache-misses or number of instructions.
- **Integrated Timing** and **Functional simulators**: in order to speed simulation time, these techniques are mixed to provide a new type of simulator. Two commonly referred techniques are: Functional-first simulator and Timing-first simulator. As an example of the first, a functional simulator in the frontend could feed with execution traces (in memory or through a file system backed-up file) to a backend timing simulator. An example of the second would be to use a functional simulator to validate (within a given error range) the execution of a timing simulator.

• Scope of the target

- **Full-system**: simulates an entire platform allowing the simulator to boot the simulated machine and load an OS.
- **User-level** or **Application-level**: the simulator executes the targeted workload within OS process execution boundaries.

• Input Driven

- **Trace-driven**: the simulator consumes *execution traces* (often, file-system backed-up files) which consist of collected information useful to the simulator to recreate the execution of the targeted workload.
- **Execution-driven**: As oppose to the trace-driven approach, the simulator executes the targeted workload as if it were the targeted machine.

2.1.1 Requirements

At the Barcelona Supercomputing Center (BSC) memory systems group, we are not focused uniquely in the CPU features but also in the memory hierarchy where the interactions with the main memory devices are in our particular interest. The requirements of the *Memory Systems group for HPC* for the simulation study are:

- Simulation time
  - Simulation time: we considered a reasonable time for executing a simulation in less than 72 hours. *e.g.,* it is impractical to wait for a week or more into a single experiment to complete [126].
- HPC applications
  - Simulation of large-scale HPC applications: Message Passing Interface (MPI) applications with tens, hundreds, even thousands of processes [16].
  - HPC applications have repetitive behavior: the main loop of the applications performs similar processing on different data.
  - In an exploratory execution on the real platform, our targeted workloads lasts in average 10 s to complete the 50 billion of instructions.
  - *User-level* simulation (not full-system) is sufficient for our research. We prefer to avoid full-system simulation in order to reduce the simulation time.
- CPU
  - The simulator has to be validated against real hardware. Although the correctness of the system simulator is often provided via validation of real hardware. Most of the CPU simulators in the state-of-the-art, are validated versus outdated micro-architectures.
  - Simulation of high-end `x86` multi-core processors such as Intel's Sandy Bridge: the HPC architecture installed on most of the MareNostrum 3 nodes.
  - We cannot simulate only in-order cores. Our purpose is to stress the memory hierarchy up to the main memory. This decision implies to include OoO cores, to allow the micro-architecture to continue fetch, decode, and dispatch instructions after a single load or store.
  - Simulation of at least one socket. BSC memory group is interested in the memory hierarchy: in current architectures main memory is shared among all the processes that run on a node. In order to have a representative use case, we have to simulate all the processes that execute on a single node. A

possible option is to simulate a single socket and the corresponding memory (the memory directly connected to this socket) claiming that this is good enough because:

    * The HPC system should be advanced enough to allocate application's data into the same Non-Uniform Memory Access (NUMA) node, (see Section 8.2.2).
    * Access to remote memory location causes additional latency and system contention that leads to performance loss [107, 37]. Nodes in the MareNostrum 3 supercomputing cluster are 2-socket systems. Memory transactions originated in on socket with reference to the other, are served through the Intel technology.

- Main memory
    * The primary objective of the simulation infrastructure is the analysis of HPC main memory systems.
    * DRAMsim2 [122] is considered the standard tool for memory system simulation. DRAMsim2 is a cycle-accurate model of a DRAM memory controller, DIMMs, and buses by which they communicate. All major components in a modern memory system are modeled as their own respective objects within the source code, including ranks, banks, command queue, the memory controller, etc. DRAMsim2 was developed by University of Maryland and validated against manufacturer Verilog models. DRAMsim2 can be integrated with various CPU simulators (although not all of them) using a reasonably fairly simple interface.

2.2 CPU Simulators

The most widely used CPU system simulator is **`gem5`** [21]. Since 2002, more than a hundred publications are referred to improve, extend, or use this simulator. It originated as a merge of the CPU's pipeline `M5` simulator [22] and the memory hierarchy inherited from `GEMS` [99]. The `gem5` is a multi-architecture simulator that can perform cycle-accurate, event-driven or hybrid (both, event-driven and cycle-accurate) simulation. Moreover, `gem5` can perform as a full-system or user-level simulator. Regarding the `x86` Instruction Set Architecture (ISA), the micro-architecture implementation is not explicitly bound to a specific Intel's or AMD's product; `gem5` act as a functional ISA simulator. To achieve main memory simulation, `gem5` might use internal memory models or integrate an external memory simulator.

Sniper [30] is among the most used simulators. Sniper is the result of an enhancement of the Graphite parallel simulation infrastructure [106]. The simulator implements several features for the `x86` micro-architecture, such as multi-threading, DVFS support, or the implementation of hardware prefetchers. Sniper is a user-level, event-based (Interval-Core model) simulator [52]. For main memory access, the simulator uses a fixed latency model with the option to add a normal distribution on top of it. Sniper does not provide the possibility to add an external memory simulator.

MARSSx86 [112] is a full-system `x86` simulator built on top of **QEMU** [14] enabling OS execution. For the CPU simulation, MARSSx86 extends **PTLsim** [147], a cycle-accurate simulator that models a generic processor's pipeline. For the `x86` micro-architecture, it aims to behave as an Intel Core 2, Intel Pentium 4, or an AMD K8. In 2005, MARSSx86 was the first available full-system `x86` open-source simulator.

TaskSim [120] is a trace-driven multi-core simulator developed at the BSC aimed to simulate large-scale HPC applications. A task in TaskSim is the computation performed between application's synchronization events; these events are used as milestones in an interval-based simulation. **MUSA** [59] implements a simulation infrastructure that uses TaskSim for the computation phases. Traces are recorded during an execution of a compile-time instrumented binary. Some of the details of the underlying architecture, such as the Instruction Per Cycle (IPC) performance, are measured and recorded so that the simulator uses this information to extrapolate performance metrics. MUSA allows detailed DRAM simulation through external memory simulators.

ZSim [125] is a user-level simulator developed by researchers at Stanford University and the MIT. Currently, it is the fastest CPU simulator attaining up to 300 MIPS being capable of performing a simulation of over a thousand cores. It uses Dynamic Binary Translation (DBT) through Intel PIN [97] allowing faster execution times compared to other simulators. ZSim is an interval-based simulator, where two phases are involved; 1) the *Bound phase*, where every core is simulated in isolation from each other to enable fast parallel simulation, and 2) the *Weave phase*, where the simulation is corrected to account for a potential collision between concurrent memory accesses. ZSim supports two alternatives for the main memory simulation: an internal memory model based on the `M/D/1` queue contention, and a software interface to use an external memory simulator.

In **HSCC** [94], authors propose a mechanism to exploit the memory access patterns on application's memory working-set with a page-locality granularity. In their proposal, the authors have extended ZSim CPU simulator with a one-level Translation Look-aside Buffer (TLB) allowing virtual-to-physical address translations and tracking application's memory requests at page level. In the proposal, a flat memory space is divided between two memory technologies: a DRAM and a non-volatile region, where the DRAM acts as a cache for the non-volatile region. To provision memory heterogeneity, HSCC's researchers had used NVMain2.0 [114] as the external main memory simulator for both DRAM and non-volatile memory.

2.3 Memory Simulators

For a detailed simulation on the DRAM devices, a specialized DRAM simulator has to be integrated with a CPU simulator. On the main memory simulators, an effort has been made to address a detailed simulation of the DRAM devices, the memory controller, and the interconnect to access it. In 2012 ISCA, winners of the Memory Scheduling Championship presented **USIMM** [32]: a trace-based simulation infrastructure for DRAM devices focusing on the memory controller scheduling algorithm. In the same year, Jeong [80] *et al.* had proposed **DrSim**; a traced-based DRAM simulator prepared to interact with a specific version of `gem5`. Unfortunately, to the best of our knowledge, neither of these simulators are maintained any longer.

DRAMsim [142] is the first open-source cycle-accurate DRAM simulator. Developed to explore different physical parameters to achieve optimal memory system performance, authors have abstracted several timing models for technologies such as: SDRAM, DDR, DDR2, DRDRAM, and FB-DIMMs. Limitations mentioned by Wang *et al.* in DRAMsim original work, were addressed in 2011 by Rosenfeld *et al.* in DRAMsim2 [122] proposal. DRAMsim2 provides a detailed timing simulation of main memory following DDR2 and DDR3 standards. DRAMsim2 were validated against Micron's DDR3 DRAM Verilog models [105] where no timing constraints violations were detected.

Ramulator [85] (2016), is the most recent open-source DRAM simulator. Because of their state-machine abstraction model, authors claim to speed-up simulation execution up to 3 times faster with respect to state-of-the-art main memory simulators. Moreover, Ramulator enables some of the high-end DRAM standards and technologies such as GDDR5 or HBM. Regarding validation, DRAM DDR3 timings in Ramulator were also validated against Micron's DDR3 DRAM Verilog models [105], where no constraints violations were detected.

NVmain2.0's [114] main focus is to simulate non-volatile technologies such as STT-MRAM, ReRAM o PCRAM, as well as die-stacked DRAM caches. It is also validated against Micron's Verilog models [105]. The authors claim to outperform DRAMsim2 in speed simulation.

By the moment of the writing of this manuscript [2] researchers released **DRAMSim3** [92], a cycle-accurate DRAM simulator validated against DDR4 Verilog models. The simulator improvements from DRAMsim2 include bank-groups, self-refresh timings, new GDDR5 models, and incorporates performance and thermal co-simulation. Moreover, it includes Hybrid Memory Cube (HMC) logic simulation and High Bandwidth Memory (HBM) dual-command issue.

2.4 Hardware Performance Counters

Hardware Performance Counters (HWPC) are a computer architecture mechanism that allows visibility of particular events within the micro-architecture. HWPC are generally accessible through a set of registers mapped into a specific memory space, or through mailbox interfaces. To access such interfaces, researchers had proposed drivers and libraries that simplify the HWPC abstraction for programming purposes.

In Linux, the proposal that has been widely adopted in the kernel for HWPC access is **`perf_events`** [74]. The initial version to support them was merged into mainline kernel in version 2.6.31 [66], where the interface between user requests and the kernel is enabled through the `sys_perf_event_open` system call[3]. On recent versions of the Linux kernel, different access policies enable HWPCs measurements with different granularity, such as: per process, per thread, per core or socket-wise. Moreover, for security reasons, there are further restrictions according to the user's credentials and access controls.

On top of `sys_perf_event_open`, there are some libraries that enables user-land access to the `perf` subsystem. The Performance Application Programming Interface (**PAPI**) [132], is among the most used libraries that enables an easy access interface to HWPC on diverse CPUs and also, off-chip counters for external devices such as some GPUs. The **Perfmon2** or **libpfm4**, in its latest release [143], supports Linux's `perf_events`, and the abstraction extends portability to Windows and MacOS.

[2]Therefore, not included in the results from our study but deserves to be mentioned in this document.

[3]In Linux 2.6.31 was originally named, `perf_counter_open()` but renamed to `perf_event_open()` in 2.6.32.

EXEgesis [57] is an ambitious project that extracts instruction-level micro-architectural micro-operation (micro-operation) latency solely using vendor's documentation. The researcher's objective is to provide compilers with micro-architectural information for code generation. At the moment of the writing of this thesis, EXEgesis supports only Intel's x86-64 (via an Intel Software Developers Manual [70] parser). EXEgesis also identifies micro-operation execution port scheduling, i.e., execution units where the micro-operations are executed.

Furthermore, HWPCs might be accessed through direct configuration of Model Specific Registers (MSRs). Remarkable initiatives are using this approach: **Agner's Fog** [50], provides source-code of a Linux kernel module for x86-64 machines that enables access to HWPC through a simple interface. **LIKWID** [134] is a framework that provides tools for application performance. One of the tools, `likwid-perfctr`, enables application developers to measure HWPCs. The measurements might be taken externally, through a command line wrapper utility or, through an Application Programmer Interface (API) for specific code sections within the application. HWPCs are read using the `msr` mechanism deployed within the Linux kernel. Intel Architecture Code Analyzer (**IACA**) [71][4] is a tool that provides estimates about execution-units utilization for several families of Intel's micro-architectures. To use IACA, a binary needs to be instrumented at compile time. Once compiled, the binary is statically analyzed. A similar approach is used by **llvm-mca** [38], using the information available in LLVM's compilation phase to measure the processor's throughput and resource utilization.

2.5 CPU Micro-benchmarks

A well-known on-line resource for micro-architectural details is published by **Agner's Fog** [49]. He has historically collected a comprehensive list of latency, throughput, and execution-units utilization of several CPUs from different IHVs. The list is built upon a set of scripts that execute a set of pre-defined micro-benchmarks. The scripts and source code for his micro-benchmarks are available on his research website [50]. His code supports execution on Linux and Windows OSs with support for single and multiple threads. Administrative privileges are required to load and run Agner's micro-benchmarks.

In **`uops.info`** [1] the researchers provide a complete list [3] of instruction latency, throughput, and execution-units utilization for all families of Intel micro-architecture.

[4]In Q3-2019 IACA has been declared End-Of-Life (EOL) by Intel but still available for download.

Their proposal is based on the concept of *blocking instruction*, a construct that enables them to characterize the execution-unit scheduling accurately with respect of the state-of-the-art techniques. Although the interface to execute the micro-benchmark is released (nanoBench [2]), at the moment of writing this document, the source code for such construct it is not publicly available.

`likwid-bench` is a tool within the LIKWID framework [134], that allows an automatic generation of micro-benchmarks. The framework provides a series of generic micro-benchmark kernels which are parameterized by command line switches.

Microprobe [18] is a framework to automatically generate micro-benchmarks. Originally designed to estimate the power consumption of multi-threaded applications running on a multi-core system. Microprobe also allows micro-architectural parameter characterization such as the usage of functional units. The framework is composed of a set of modules that assist the automatic code generation through a comprehensible configuration script. One of the modules, the ISA definition module, is provisioned with details about the targeted micro-architecture. In the original publication [18], researchers used an IBM's POWER7 micro-architecture. Currently, the open-source release [68], supports RISC-V micro-architecture.

2.6 Memory Micro-benchmarks

To the best of our knowledge, the first tool that had helped the memory research is **STREAM** [102]. In STREAM's first version, sustained memory bandwidth is measured using impressively simple kernels written in standard `C` and `Fortran`. To achieve multiprocessing, STREAM uses OpenMP and MPI. On the one hand, these characteristics made STREAM platform-portable; but on the other hand, details and control over the memory accesses are left to the compiler and OpenMP's runtime.

LMBench [103] is constantly referred in the state-of-the-art. LMBench's main focus is to measure transfers between the processors and the members of the memory hierarchy, such as the primary storage or the main memory. As a consequence, it measures main memory latency as well as it performs cache capacity discovery.

RAMspeed [119] by Alasir Enterprises, provides 18 specialized kernels (INT, FLOAT, and SSE), that access data in different reading/writing patterns to measure memory performance. Another tool provided by an enterprise is the Intel Memory Latency Checker (imlc) [141]. The tool is distributed in binary-only form, making it difficult to operate outside the predefined command-line arguments. An essential contribution of this tool is the ability to switch off the hardware prefetchers (privileged credentials are required).

Pmbw [20] collects old requirements from STREAM, LMBench, and imlc, exporting a multi-core, multi-architecture bandwidth-latency memory tool. Pmbw provides 21 hand-crafted assembly-code and compiler-ready kernels to achieve such goal.

In **PROFET** [116], the authors provide two different tools to characterize main memory latency and memory bandwidth. An interesting contribution in the proposal is about memory bandwidth measurements: the tool instruments kernels that deliver a ratio of read/write instructions deploying 25 kernels that achieve bandwidth utilization from 50 % up to 100 % with a granularity of 2 %. A second contribution is cache pollution awareness: to avoid unnecessary misses on the cache hierarchy, write memory transactions make usage of non-temporal ISA instructions.

As an extension **`uops.info`** [1], in **nanoBench** [2] authors provide several Python scripts that explore, discover, and characterize the cache levels of the memory hierarchy. Since the objective for the main research targets CPU micro-architecture, main memory is not covered by nanoBench.

Lastly, a remarkable framework that identifies the complexity and the challenges to write a memory benchmark tool is found in **Hopscotch** [6]. Authors provide tools that characterize the elements on the memory hierarchy such as latency and bandwidth utilization. The tools are deployed with a configurable interface to use different memory access patterns that resemble real-world applications.

2.7 Validation in the State Of The Art

The `gem5` CPU simulator was originally validated against an Alpha machine. For newer versions, the validation of the simulator versus the real hardware has been delegated to the `gem5` community. For instance, Burtko *et al.* [27], validate `gem5` in dual-core configuration versus an `ARM Cortex-A9` where the researchers reported differences in the benchmarks' execution time ranging between 1.39 % and 17.94 %. The benchmarks used in their work are SPLASH-2 [144] and ALPBench [88]. Anoter community validation effort between `gem5` and real hardware is presented by Gutierrez *et al.* [62]. In said work, researchers have modified and extended `gem5` to behave as a `VExpress` development board (an `ARM` processor with `DDR2` as main memory) where they used the `CPU ARM` subsystem within the `gem5`'s `O3 Core Model` and `GEMS`' `SimpleDRAM` for the main memory accesses. The workload chosen to evaluate performance is the SPEC CPU2006 benchmarks. Researchers claimed to achieve an average absolute error of 5 % with this configuration.

Sniper was validated originally against an Intel Core 2 machine [29]. On a further effort, Sniper was validated against an Intel Xeon X5550 processor (Nehalem micro-architecture) using a set of the SPLASH-2 benchmarks [144]. The validation results show that Sniper's IPC error concerning the actual hardware is below 25 %. Let's point out that Sniper uses a fixed memory latency for main memory accesses where a 65 ns value is used for validation in the original paper.

TaskSim was originally validated versus an IBM Cell processor. In MUSA (that uses TaskSim to simulate computation phases), authors validate their infrastructure against an Intel Xeon E5-2670 machine using the NAS Parallel Benchmarks [11], HYDRO [89], and SPECFEM3D [87]. In the study, the authors reported achieving relative errors below 10 %.

The ZSim simulator is validated against an Intel Xeon L5640 machine (Nehalem architecture) with the internal `M/D/1` memory model using SPEC CPU2006 and PARSEC [19] benchmarks. The authors reported IPC errors of below 10 % between the two systems. The ZSim authors clarified that using DRAMsim2 will restrict the simulation to 3 MIPS, landing outside their design goals. Therefore, the validation with DRAMsim2 is not performed in the original paper.

Work introduced by Akram and Sawalha [7] surveys state-of-the-art `x86` CPU simulators. In the experimental section, they tested four simulators: `gem5`, `Multi2sim` [136], `PTLSim` [147] and Sniper. The chosen workloads used for the analysis are SPEC CPU2006 and MiBench [61]. In their configuration, they have not reported the usage of any external memory simulator. For their analysis, they used an Intel Core i7 processor (Haswell micro-architecture) as the targeted system to model. Their results presented a broad set of discrepancies among simulation executions.

DiagSim [81] proposes a method that detects changes in IPC's patterns. The method points out where the differences between the real system and a simulator drift away. The authors categorized micro-architectural features through a set of *diagnostics* bounded together through a *dependency map*. A *diagnose* is in fact, a micro-benchmark for the targeted ISA. By running each *diagnose* through several iterations with different input parameters, micro-benchmarks can *identify* fluctuations in the IPC behavior. The researchers admitted that *identify* the IPC might need manual interaction, and no multi-threading scenarios were considered.

Name	Full-system/ User-level	Trace-/Execution-driven	Multi-threaded/ Sequential	CPU complexity	Validated for	External DRAM simulator
`gem5` [21]	Both	Both	Sequential	OoO	ARM (partial)	Yes
Sniper [30]	User-level	Both	Multi-threaded	OoO	`x86` (Nehalem)	No
ZSim [125]	User-level	Execution	Multi-threaded	OoO	`x86` (Westmere)	Yes
TaskSim(MUSA) [120, 59]	User-level	Trace	Sequential	in-order/OoO	`x86` (SandyBridge)	Yes
MARSSx86 [112]	Full-system	Execution	Sequential	OoO	`x86` (Nehalem)	No

Table 2.1 Overview of reviewed CPU simulators.

Name		Trace-/Execution- Sequential	Multi-threaded/ technologies	Memory
DRAMSim2 [122]	Both	Sequential	No	DDR{2,3}
Ramulator [85]	Both	Sequential	No	DDR{2,3,4}, GDDR5, HBM, WIO1/2
NVMain2.0 [114]	Both	Sequential	No	DDR{2,3}, STT-RAM, PCRAM, ReRAM

Table 2.2 Overview of reviewed DRAM simulators.

2.8 Decision Process

Table 2.1 summarizes the CPU simulators we considered to use for this work. In full-system or hybrid mode simulation, `gem5` lasts for a significant amount of time (several hours up to weeks) [126]. Sniper won't allow co-simulation with external main memory simulator. MUSA extends TaskSim capabilities to perform OoO for multiple ISAs and allows co-simulation with an external DRAM simulator, but it was still in development by the moment we performed the experiments about this thesis. The current implementation of the `x86` ISA in MARSSx86 is outdated, and neither provides an interface to simulate DRAM concurrently. The only simulator that covers our requirements of reasonable execution time and DRAM co-simulation is ZSim.

For main memory simulators, Table 2.2 summarizes our considerations. Because all of them technically satisfy our requirements, the decision to chose one simulator among them was grounded on state-of-the-art reputation. Ramulator and NVmain2.0 are relatively new proposals, whereas DRAMsim2 has been extensively used within the computer architecture since 2011. Up to 2020, the number of paper citations for each one of them in the IEEE site is: 435 for DRAMsim2, 73 for Ramulator, and 54 for NVMain2.0. Moreover, programming interfaces to work with DRAMsim2 are broadly available i.e., ZSim already include support code for DRAMsim2.

Decision: For the CPU we decided to use the execution driven simulator ZSim, and DRAMsim2 for the main memory simulation.

CHAPTER 3

Experimental Methodology

Figure 3.1 introduces the flow to design, upgrade, and tune system simulators. The first step when approaching computer architecture simulation is recognizing the similarities and differences between the real-world system and the state-of-the-art simulators. In this work, we decided to use an `x86` CPU simulator that was designed to behave as a previous generation micro-architecture (Nehalem) from our targeted machine (Sandy Bridge). Therefore, we first identified the differences between the system and the CPU simulator using the vendor's available documentation: the Intel's Software Developer Manual [70] and the Intel's Optimization Guide [69]. Furthermore, we decided to include current reverse engineering efforts published in the state-of-the-art [101]. For main memory, the chosen DRAM simulator is designed to support the DDR2 and DDR3 standards; as our target model uses DDR3 memory devices and is validated versus Verilog models, we assume the memory model is accurate.

The second step consisted of matching and discovering those characteristics beyond vendor's documentation. To achieve this goal, we used the concept of micro-benchmarking: execution of synthetic programs with an iterative loop comprised only of a single instruction. During micro-benchmark execution, we observed the

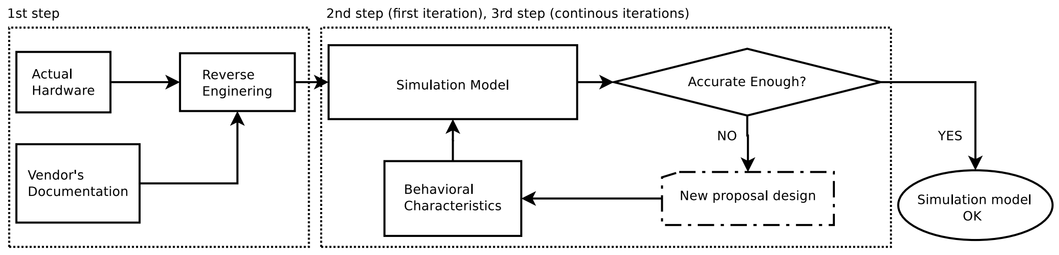


Figure 3.1 Iterative process of integrating and validating new features to a system simulator (simulation model).

micro-architectural resource activity using Hardware Performance Counters (HWPC) in the real machine. Because only one instruction is executed per micro-benchmark, fewer resources are utilized by the underlying micro-architecture, allowing us to discern resource utilization out of the scope of the execution of the micro-benchmark. Moreover, the micro-benchmark kernel is specially crafted to avoid micro-architectural optimization techniques designed to boost single-instruction performance.

To extract information from real machine execution, we used HWPC: special registers designed to provide information about the micro-architecture. If the corresponding HWPCs are not present in the simulator, we implemented them according to vendor's documentation. Implementing a non-existing HWPC in the simulator is not a straightforward task; system simulators are coded in high level languages, where the code logic attempts to mimic the micro-architectural details but it is not the same as the micro-architecture itself. For example, speeding the simulation time became a software-specific feature that imposes serious restrictions when attempting to implement and extract the corresponding micro-architectural details. Finally, the last stage on the second step consists of comparing both systems with their respective HWPC measurements. If discrepancies arose in the simulator, we updated the simulator's logic with values that correspond to those extracted from our micro-benchmark execution in the real machine.

If simulator's accuracy is not sufficient, a third step (optional) on the validation flow is to propose a novel design integrating it back to the simulation infrastructure so that a new comparison and assessment could be made. It is a common practice from IHVs to hide documentation of important architectural features so that they can keep a commercial advantage over competitors. Also, when implementing real-world models, researchers deliberately take some features out, for example, to speed the simulation process. Both conditions leave space for design proposals to explore new alternatives to achieve efficient computing, but also to accurately match existing designs.

3.1 Hardware Platform

The targeted computing system we aimed to simulate is built upon a dual-socket platform; each socket embodies an Intel Xeon E5-2670 Sandy Bridge EP processor operating at 3.0 GHz [72]. By the moment we began this research Sandy Bridge was a micro-architecture still used at large by smaller Tier-0 systems [40]. Specifically, MareNostrum 3 supercomputer was based on over 3,000 nodes with 2x Sandy Bridge EP processors.

In MareNostrum 3 systems, the main memory is populated by 4x 4 GiB DIMMs devices connected to the processor using four `DDR3-1600` channels [124]. Each processor comprises eight cores; the hyper-threading feature has been disabled like in most HPC systems [31, 40]. The running OS is a SUSE Linux distribution running on top of a 3.0 Linux kernel [93].

For the simulation infrastructure, we chose to work with the integration of ZSim and DRAMsim2 system simulators. ZSim [125] is a user-level, execution-driven CPU simulator originally developed to behave as an Intel Westmere micro-architecture. DRAMsim2 [122] is a cycle-accurate model of a DRAM memory controller, the DIMMs passive devices, and buses by which they communicate. It has been validated against the DRAM manufacturer's Verilog models [105].

3.2 Simulation Model

The simulation model of DRAMsim2 is depicted in Figure 3.2 In a simplistic view, the model consist in three queues: 1. the Transaction Queue, 2. the Command Queue, 3. and the Response Queue. When CPU memory transactions arrive, they are submitted to the Transaction Queue. An internal algorithm translates the incoming requests into DRAM commands which fill the Command Queue. To prevent timing violations and data hazards among transactions, the DRAM commands are scheduled according to the Bank State status issuing the corresponding timing simulation in an OoO manner. Once the transaction's simulation is finished, the Response Queue is filled with the simulated transaction and the Memory Model updates the Bank States table while keeping record of latency achieved during transactions. When an external simulator is hooked to DRAMsim2 as the transaction originator, that latency is reported back to the external simulator as the memory access latency; when DRAMsim2 is used in traces mode, the latency is logged accordingly.

Since DRAM is an industry standard for all memory IHVs, the parameters to configure the main memory simulator are mainly the accurate timings for the passive devices. We configured the simulator's technical characteristics found in the vendor's documentation for the DIMM packages [124]. Table 3.1 summarizes the timing parameters for our DDR3-1600 targeted model. Physical parameters such as temperature, might affect the DRAM simulation; on DRAMsim2 they are treated as an external simulation, not affecting the access latency to the passive device. Despite the fact that updating technical characteristics should be enough for DRAMsim2 to get correct timings (latency) between the memory controller and the passive devices, we still need

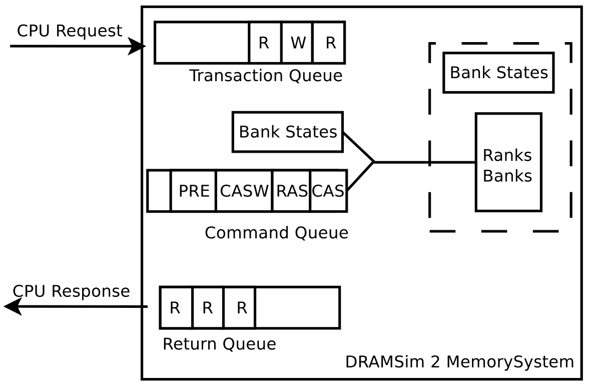


Figure 3.2 DRAMSim2's Memory controller model.

Parameter	Description	Value
tBURST	Burst Length	4
tCAS/tCL	Column Access Strobe latency	11
tRTP	Read To Pre-charge delay	6
tCCD	Column to Column Delay	4
tWTR	Write To Read delay	6
tRTRS	Rank to Rank Switching time	1
tCWD	Column Write Delay	10
tWR	Write Recovery time	12
tCKE	Next power up for an idle device	4
tCMD	Command transport duration	1
tXP	Exit power down with DLL on to any valid command	5
tRCD	Row to Column Delay	11
tRP	Row Pre-charge	11
tRRD	Row activation to Row activation Delay	5
tRFC	Refresh Cycle time	208

Table 3.1 DRAM DDR3-1600 parameters used in our simulated model.

to characterize the maximum access time (worst-case scenario) from the CPU to retire a memory instruction. Since this objective is part of the integration of the CPU and main memory simulators, we designed a set of micro-benchmarks for this purpose.

For the CPU model, Figure 3.3 depicts a diagram of the Sandy Bridge micro-architecture. The diagram is divided in two sections; the *frontend* on the left side, and the *backend* on the right side. On Intel's microprocessors, the *frontend* is the portion of the micro-architecture that takes the byte stream from the Instruction cache, recognizes and tags every instruction to later break them into smaller portions of execution called micro-operations micro-operations. The smaller operations are organized into a queue which serves the Reorder Buffer (ROB) for OoO execution. When possible, the ROB issues up to 4- micro-operations to the micro-operation-scheduler. The micro-operation

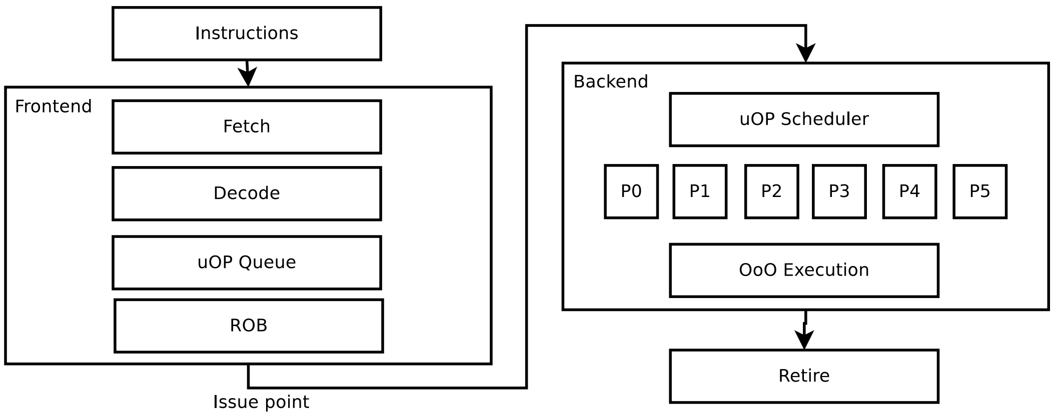


Figure 3.3 Simplified diagram of an Intel's Sandy Bridge micro-architecture.

scheduler along with the execution units (also called *ports of execution*) make up the *backend* portion of the micro-architecture. The instruction retirement takes place when load and store transactions are cleared and data could be written back to the micro-architectural registers.

The chosen simulator was developed with the intent to behave as an `x86` Westmere system. Westmere (2010) is a micro-architecture upgrade of the manufacturing process of the Nehalem micro-architecture (2008) [70]. Sandy Bridge (2011) introduces micro-architectural changes over Westmere, among the most noticeable are:

1. The four branch predictors in Nehalem are improved, non-fully documented but mentioned that number of bits has an extended range.

2. A new *micro-operation cache* holding up to 1536 micro-operations.

3. The `L1` instruction cache is improved in associativity from 4-way to 8-way.

4. The `L1` TLB adds several entries dedicated to large pages, from 7 to 16.

5. The capacity of the ROB is increased from 128 to 160 entries.

6. The number of register files dedicated to renaming is doubled, using 160 entries for integer operations and 144 entries for vector instructions.

7. The execution unit scheduler improved its capacity to hold from 36 to 54 micro-operations entries in the reservation station.

8. From the six execution units available, three are kept for arithmetic operations supporting the new vector extensions and three ports are specialized for memory transactions. On Westmere, two ports were used for store operations and one

micro-architecture feature	**Westmere**	**Sandy Bridge**	**ZSim**
`L1 iTLB` entries	142	144	N/A
micro-operation cache entries	N/A	1536 8-way	N/A
ROB entries	128	160	128 → 160
ROB register files	128	144 int + 160 vector	128 → 144
micro-operation scheduler slots	36	54	36 → 54
execution units for loads	2	2	2
load buffer entries	32	64	32 → 64
execution units for stores	2	1	2 → 1
store buffer entries	32	36	32 → 36
`L1i` size associativity	32 KiB 4-way	32 KiB 8-way	32 KiB* 4-way
`L1d` size associativity	32 KiB 4-way	32 KiB 8-way	32 KiB* 4-way
`L2` size associativity	256 KiB 8-way	256 KiB 8-way	256 KiB* 8-way
`LLC` size associativity	4 MiB-30 MiB 8-way	3 MiB-20 MiB 20-way	20 MiB* 20-way

Table 3.2 Comparison of micro-architectural features. On the ZSim column, the '*' stands for configurable entries, and the '→' for the parameters to update.

for load transactions. On Sandy Bridge, two ports are dedicated to address generation and load requests leaving the execution unit with only one port for store operations.

9. The number of entries of the load and store buffers is increased from 32 to 64 entries for the load buffer, and from 32 to 36 entries for the store buffer.

10. On the most exclusive product line of processors for HPC in Westmere, the LLC size is 30 MiB. In Sandy Bridge, the largest LLC capacity is limited up to 20 MiB with a 20-way policy. Moreover, the LLC is connected through all cores by a ring bus. Maurice *et al.* [101] reversed engineered the hashing algorithm used on Sandy Bridge to reduce congestion and equitably distribute traffic on the LLC.

Table 3.2 introduces a comparison over the two micro-architectures and the CPU simulator. Each row in the table describes a micro-architectural feature. Every column belongs to each one of the compared systems: the ZSim CPU simulator, Westmere (the micro-architecture that ZSim aims to model), and Sandy Bridge (the micro-architecture of the targeted system, also deployed on the MareNostrum 3 computing nodes). On the ZSim's column we also integrate the static changes derived from this upgrading process, meaning the corresponding numbers that the simulators should update to behave as the new targeted system.

CHAPTER 4

Micro-benchmark Design

Our micro-benchmark proposal allows to extract the number of micro-operations and the execution unit utilization. Both parameters are a vital source for accuracy in `x86` CPU simulation since determine the *instruction latency*. Moreover, the micro-benchmark proposal allows us to discover main memory access latency.

Our simulation model targets an Intel's Sandy Bridge micro-architecture. Although Intel's `x86` ISA is known to be a Complex Instruction Set Computer (CISC), the processor's frontend implements a decoder which breaks every instruction into simpler operations that resembles a Reduced Instruction Set Computer (RISC) architecture. Each one of these operations are known as micro-operation (micro-operation). Some instructions are broken into a single micro-operation, while others into more. Originated at the frontend, the micro-operations are delivered to the backend so that the computing units execute them when possible. These units are generally known as *execution units* or *ports of execution*. On modern architectures, some execution units are prepared to compute not only generic processing work but specialized operations such as memory related operations or intense arithmetic computations. To properly assign work to the execution units, a scheduler arbitrates the execution of micro-operations. According to scheduling policies, the execution units might present a non-balanced distribution when executing the micro-operations. Moreover, there are some instructions which need a sequencer intervention leading to a dynamic execution of micro-operations based on instruction's parameters.

In this section, we elaborate the assumptions and design considerations in the proposal of the micro-benchmark for CPU micro-architectural parameter extraction and main memory access latency.

Name	Description	Event	Umask
CPU_CLK_UNHALTED	Cycles where CPU is not halted	0x3C	0x01
INSTRUCTIONS_RETIRED	Instructions until the last micro-operation is retired	0xC0	0x00
UOPS_DISPATCHED_PORT.PORT_0	Number of cycles during one uop is dispatched to the execution unit {0,1,2,3,4,5}	0xA1	0x01
UOPS_DISPATCHED_PORT.PORT_1		0xA1	0x02
UOPS_DISPATCHED_PORT.PORT_2		0xA1	0x0C
UOPS_DISPATCHED_PORT.PORT_3		0xA1	0x30
UOPS_DISPATCHED_PORT.PORT_4		0xA1	0x40
UOPS_DISPATCHED_PORT.PORT_5		0xA1	0x80

Table 4.1 Hardware Performance Monitoring Counters used to characterize instruction latency and execution unit (port) utilization.

4.1 Micro-benchmarks Proposal

Parameters extraction is a task that has been around for some time in the computer architecture community (Section 2.5). Our micro-benchmark proposal allows micro-architectural characterization considering the micro-architecture limitations of our targeted simulator. The micro-benchmarks are designed to run as a user-level application on top of a Linux kernel with non-administrative privileges. The design of the micro-benchmarks consists of two sections: the OS interactions and the micro-benchmark main loop. Figure 4.2 portrays the general characteristics of the micro-benchmark.

A `C` program encloses the OS interactions that provide all functionality such as memory allocation and variable initialization, hardware counters initialization and metrics collection as well all program cleanup. By only counting the execution of the micro-benchmark kernel, the overheads of running on top of an OS are diminished. The micro-benchmark kernel is written directly in `x86` assembly to handcraft the `x86` instruction sequence avoiding any compiler optimization.

There are two modes for the `C` program to wrap the micro-benchmark core: execution in the real machine, and execution within the simulator. The difference between these modes relies on the procedure to collect the measurements.

On the real system, the measurements are obtained via system calls to the Linux `perf` [67] subsystem to access specific HWPC. Table 4.1 shows the HWPCs used in the real machine execution. These calls to perform measurements are made just before entering the main loop and immediately after the iterations are finished. The execution environment for the micro-benchmark plays an important role in micro-benchmark's measurement collection.

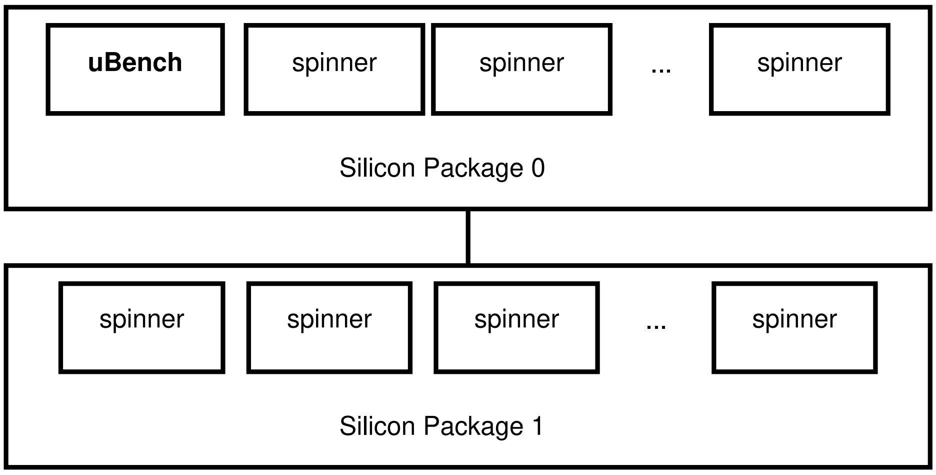


Figure 4.1 Execution environment: spinners in all cores except where the micro-benchmarks is executed.

To the best of our knowledge, we must minimize the effects of the Linux scheduler [96], so that the micro-benchmark is always executed in the same core within the same node. To minimize such effects, we use the `taskset` utility to *pin* the execution of the micro-benchmark, narrowing its execution to a specific core in a particular silicon package.

When processor's load is not high enough, policies for energy efficiency are triggered, affecting process scheduling and adding hardware delays derived from power switching features. To overcome such policies, we conceived an execution environment for the *rest* of the cores in the 2-processor system. We designed a small program (*spinner*) that performs a busy-processing routine with a 100 % of core utilization. In one processor, the spinner program is instantiated 15 times and pinned to each one of the cores where the micro-benchmark is not executed. For the other processor (silicon package), we created 16 instances pinned to each one of the cores in that processor. Figure 4.1 depicts the execution environment. Lastly, although each one of the instances are independent of each other, creating instances of the spinner in the second silicon package is crucial to minimize latency imposed because of the snooping messages between the interconnect on both processors.

Furthermore, we were aware of some glitches on the `perf_events` subsystem on the kernel version we used [39]. To diminish some of these artifacts, we used the `PERF_FORMAT_GROUP` to inquire only once the kernel interface per measurement limiting the number of accessed counters to 4. Being 8 counters needed to extract, we performed twice the execution of the micro-benchmarks with different groups of counters. No syncing between the executions is needed since both executions cover the same micro-benchmark kernel.

On the simulator infrastructure, we use ZSim's *fast-forward* feature that allows skipping the simulation up to a specific point in the program execution. To set the points where the fast-forward feature is turned on/off, a special watermark on the micro-benchmark code is set: `xchg %rcx, %rcx` where the value on `rcx` determine the operation that is *fast-forward on* and *fast-forward off*.

There are two execution watermarks configured in the program simulation; just before entering the micro-benchmark kernel and as soon as the micro-benchmark kernel is finished. The simulation begins in fast-forward mode up to the first watermark is found, from that moment the detailed simulation runs until the next watermark is found. The simulation finishes by enabling back again fast-forward mode. The simulator counters only record what has happened between the two execution watermarks.

To extract parameters from micro-benchmark execution in both cases, the real machine and the simulator, a quotient is necessary:

$$Extracted\_parameter = \frac{HWPCmeasurement}{Number\ of\ executed\ instructions}$$

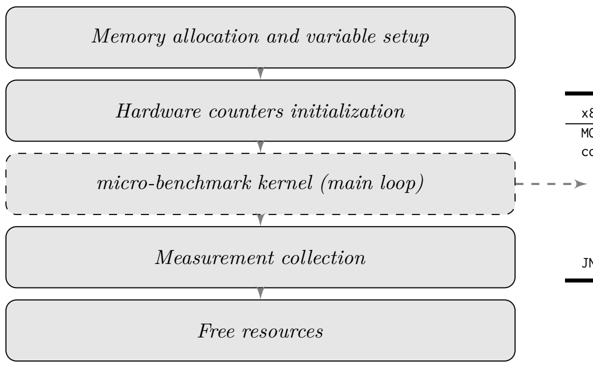
The *number of executed instructions* corresponds to the length of the micro-benchmark kernel and the count of times it has been iterated.

We conceive two categories for the micro-benchmarks; in Section 4.2, we introduce a class of micro-benchmarks that discover per-instruction latency in the processor's execution units utilization, and in Section 4.3, a family of micro-benchmarks that characterizes the memory hierarchy.

4.2 Instruction Latency and Port Utilization

The idea behind the CPU micro-benchmark is to isolate the resource utilization per single `x86` instruction. On the right side of Figure 4.2, a small example of a CPU micro-benchmark kernel is presented. We achieve resource isolation by sequentially executing the same `x86` instruction 10 billion (10^9) times so that the metrics collected directly via HWPC are inevitably distinguishable. First, the iteration count is initialized through the general-purpose register `ecx`: a mandatory requirement in `x86` architectures. Next, the micro-benchmark kernel (labeled as `core_loop`), is a consecutive sequence of `x86` single-instruction statements. Finally, after decreasing the iteration counter, the loop is executed repeatedly until the counter register drops to zero.

Moreover, we proposed a tool that automates the generation of 358 unique cases for different instructions of the `x86` ISA. Extensions to the Intel ISA such as vector


Memory allocation and variable setup
Hardware counters initialization
micro-benchmark kernel (main loop)
Measurement collection
Free resources

x86 pseudo-code	Explanation
`MOV %ecx, $10000`	Initialize loop counter *ecx* to 10,000
`core_loop:`	beginning of the loop
`    XOR %eax, %ebx`	single instruction
`    XOR %eax, %ebx`	single instruction
`    ...`	...
`    XOR %eax, %ebx`	single instruction
`    DEC %ecx`	decrement iteration counter
`JNZ core_loop`	iteration jump

Figure 4.2 The micro-benchmark kernel is isolated by a high-level program that wraps all OS interactions, micro-architectural features are collected immediately after the micro-benchmark is finished.

operations in any version of the `SSE` or `AVX` are not included in this study. We extended the micro-benchmark kernel up to 10,000 `x86` instructions for two reasons:

1. If an instruction is decoded as a single micro-operation, then the Sandy Bridge *micro-operation cache* would be filled with a total of 1,536 instructions. To overcome the *micro-operation cache*, at least 3,072 instructions are needed. Extending the execution in more than three orders of magnitude would allow performance monitoring counters to have reliable measurements.

2. In a 10,000 instruction block, the overhead incurred for the `jnz` (jump if not zero) instruction is negligible.

4.2.1 Micro-benchmark Evaluation

Once we updated the CPU simulator with micro-architectural changes and upgraded the ISA latency discovered by the micro-benchmarks (see Table A.1 in Appendix A for a per-instruction comparison between the targeted systems), we evaluated and compared the enhancements to the simulator and the real machine using the relative CPI simulation error calculated as

$$Relative\_error = \frac{(CPI_{zsim} - CPI_{real})}{CPI_{real}}.$$

Table 4.2 presents the evaluation results. Enhancements significantly improved the overall accuracy, the percentage of instructions that have an absolute error of 2 % increased from 53.9 % to 86 %. The improvement comes mainly from the 26.5 % + 8.7 % (ZSim-original, 2nd and 3rd row) becoming only a 3.5 % + 2 % in our enhanced version. Since ZSim-original holds for latency and execution unit utilization of a previous

	ZSim original		ZSim enhanced	
Error range	# instructions	%instructions	# instructions	% instructions
$(-\infty, -100\%]$	0	0.0%	0	0.0%
$(-100, -50\%]$	95	26.5%	12	3.4%
$(-50, -2\%]$	31	8.7%	7	2.0%
$(-2, 2\%]$	193	53.9%	308	86.0%
$(2, 50\%]$	23	6.4%	19	5.3%
$(50, 100\%]$	11	3.1%	8	2.2%
$(100, \infty]$	5	1.4%	4	1.1%

Table 4.2 Relative CPI Error (summary).

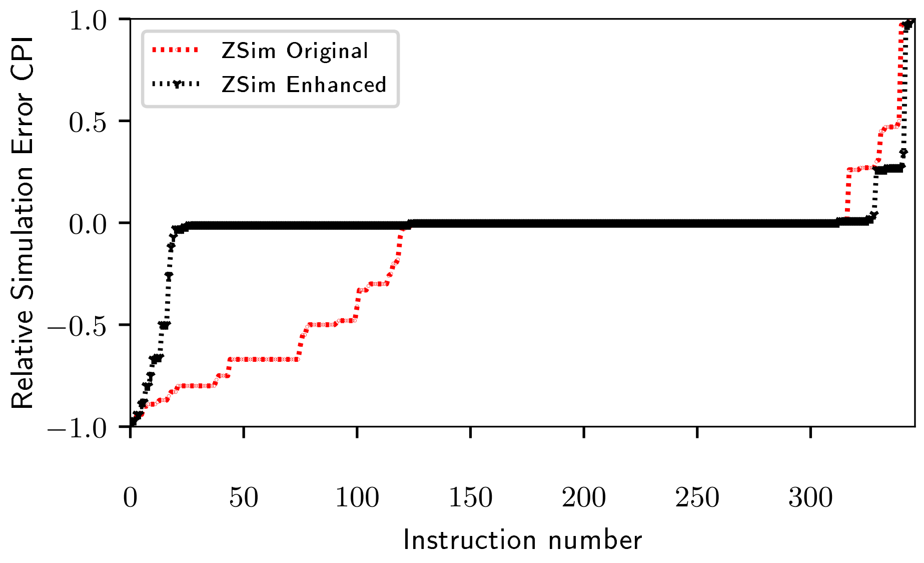


Figure 4.3 ZSim Original: ZSim from current version [152]. ZSim Enhanced: Architectural upgrade from Westmere to Sandy Bridge. The instruction number is an unique identifier per instruction from our micro-benchmark execution.

generation, most of the instructions are underestimated. The overall perspective is presented in Figure 4.3. The X-axis is labeled using an unique number as per-instruction identifier for each one of the 358 `x86` instructions in our study. On the Y-axis, the relative CPI simulation error is used. The values are sorted from left side with negative relative simulation error (underestimation of cycles in the instruction) to the right side, with instructions holding a positive simulation error (overestimation of cycles in the instruction). We identified that most of the instructions in this category fall into the *approximate instruction* ZSim's decoder. The policy for these instructions are a 1-cycle latency and being executed on any execution port. Such conditions generate faster scheduling and retirement from ZSim with respect to the real machine.

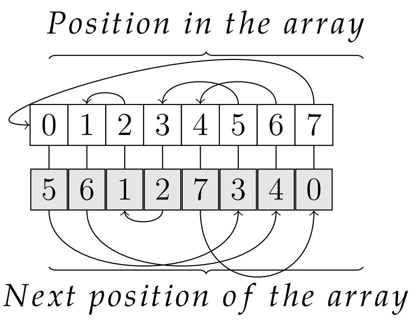


Figure 4.4 System characteristics and illustration of the pointer chasing memory access pattern used in the micro-benchmark.

4.3 Memory Hierarchy Discovery Characterization

The memory micro-benchmarks for main memory latency characterization are designed to stress the cache hierarchy and main memory. Our proposed micro-benchmark [26] follows state-of-the-art efforts (Section 2.6) but extends them in the following scenarios: 1) by overcoming the micro-operation cache, and 2) using the concept of pointer chasing we avoid the effects of hardware prefetchers.

First, on the `C` program preamble, we allocate the required memory space to operate over the pointer chase pattern. Because this memory is allocated through the OS virtual memory allocation mechanism (via `posix_memalign(3)`) this is a contiguous virtual memory region. We fill this region as a standard `C` array, where each element represents a member of a circular list that holds the pointer chase pattern (Figure 4.4 depicts a pointer chasing design using a circular list). An important remark for us is to maximize the number of cache misses, so we align the size of the elements of the array to the size of a *cache line* in the real hardware (64 B).

Once the initialization is set up, the HWPC are set to zero. Then, the `C` program uses inline assembly to hand off to the micro-benchmark kernel the memory addresses of the variables used as counter and the pointer to the first element to access the array.

On a pointer chase design, there is a dependency between two consecutive instructions inducing a CPU in-order execution for the instructions in the sequence. We target any given level of the memory hierarchy (`L1`, `L2`, `L3` or the main memory) by ranging the array size from 4 KiB up to 3.52 GiB. Table 4.3 summarizes the hardware characteristics to explore in both systems, as well as the memory size ranges used to perform the cache latency discovery.

For ZSim to achieve faster simulation times, the execution of the simulated workload is split by a two-phase algorithm: 1. In the *Bound phase*, every memory request from a core is simulated as if it were an standalone transaction. Furthermore, for every

Memory level	Size	Associativity	Latency	Scope	Microbenchmark array size	Number of measurements
L1	32 KiB	8-way	4	Private	4 KiB to 32 KiB	8
L2	256 KiB	8-way	8	Private	60 KiB to 256 KiB	8
L3	20 MiB	20-way	28	Shared	2.17 MiB to 20 MiB	8
Main memory	16 GiB	-	variable	Shared	532 MiB to 3.52 GiB	7

Table 4.3 By setting the micro-benchmark array size we can measure the latency of different levels in the Intel Xeon E5-2670 Sandy Bridge-EP memory hierarchy.

memory request, a fixed latency is assumed. To configure this fixed latency, ZSim enables the *mem.latency* parameter. 2. On the *Weave phase*, ZSim updates the latency achieved by the external main memory simulator (if configured i.e., DRAMsim2) with the corresponding value.

In Figure 4.5, we show a comparison between the real system and the ZSim + DRAMsim2 simulators for the memory subsystem. The X-axis of the figure represents the size of the traversed array, while the Y-axis shows the memory latency in nanoseconds (ns). On default configuration, ZSim is configured to use a *mem.latency* parameter set to 100 cycles (red line). Such configuration creates a gap in memory simulation from around 20 ns. To mitigate the gap found on memory simulation, we chose a value of 170 CPU cycles for the *mem.latency* parameter (green line); we subtract from the real machine's most prolonged latency registered (≈ 210 CPU cycles) the average CPU-to-LLC latency (≈ 40 CPU cycles). The latter provides an upper bound for the minimum latency of a request dispatched from LLC to main memory.

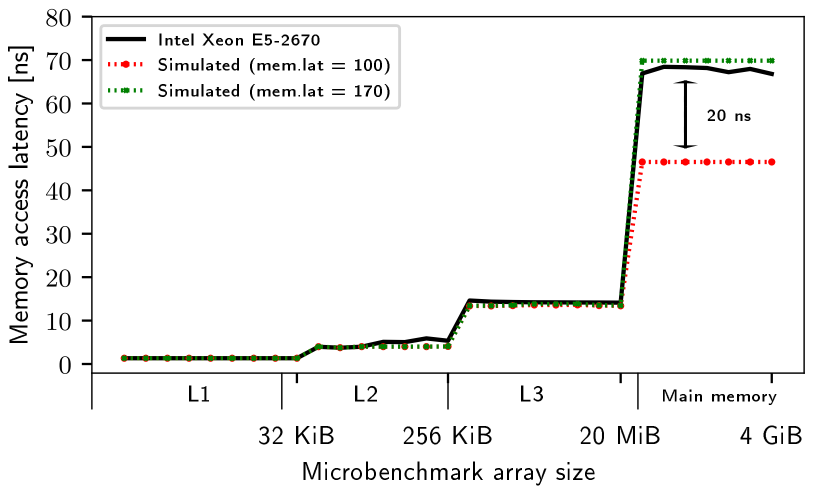


Figure 4.5 Memory discovery: Real machine vs. Simulator infrastructure.

Once that micro-operation port utilization, instruction latency, and main memory access latency are accurate enough with respect to our micro-benchmarks we proceed to evaluate the simulation infrastructure with real-world workloads. In the next chapter, we evaluate the simulation infrastructure versus the real hardware using the SPEC benchmark suite as workload for execution.

4.4 Summary

In this chapter we described the technical assumptions taken to propose our micro-benchmark design. The design was driven to overcome a series of challenges that are not present in the state-of-the-art reverse engineering efforts. For example, our micro-benchmarks are designed to overcome micro-architectural specific features such as the Intel's Sandy Bridge micro-operation cache. Moreover, by isolating the program's preamble and epilogue within the micro-benchmarks, the micro-benchmarks's kernels are prepared to measure exactly *the same* regions of code in both, the simulator infrastructure and the real machine. An important remark regarding the usability in comparison of the state-of-the-art micro-benchmarks, is that our proposal does not require administrative privileges on the execution environment.

After extracting micro-architectural parameters from our micro-benchmark execution and inserting them into the simulator infrastructure, we achieved 32.1 % of performance improvement between the systems under test. Furthermore, when executing our memory micro-benchmarks, we detected a 20 ns gap in the main memory simulation.

CHAPTER 5

Performance Evaluation

In the previous chapter, we analyzed, configured, and proposed specific simulator's parameters. Also, we validated our simulation infrastructure versus real machine with our synthetic benchmark proposal. In this chapter, we further investigate the impact of our enhancements to the simulation infrastructure using the SPEC CPU2006 [65] benchmark suite. Two groups conform the totality of the workloads: the SPEC CPU2006 Integer (SPEC CPU2006 int) and the SPEC CPU2006 Float (SPEC CPU2006 fp). In our study, we executed and evaluated the simulator infrastructure with 12 workloads of the integer set and 17 workloads of the floating-point set.

Using a traditional approach [149, 82, 78], we validated the simulator infrastructure using the relative IPC error when compared to the actual hardware. Next, we set side by side the Misses per Kilo Instruction (MPKI) for all cache levels `L1i`, `L1d`, `L2`, `L3`. For each experiment we targeted an execution of 50 billion of retired instructions. This number is chosen to be congruent with ZSim's original publication [125]. Lastly, we analyzed if the execution is bound to a specific cache level using the CPI error between the real and the simulated system.

5.1 Experimental Environment

Since we decided to execute 50 billion of instructions per benchmark, we needed to detect the moment when the number of instructions has been reached so that the workload execution in the real machine should be stopped. For this purpose, we created a launcher program that wraps benchmark execution to poll for `perf_events` until the 50 billion of instructions condition is met. An important remark is that all the

Name	Description	Event	Umask
L1D_ST_MISSES	perf `L1-data` store misses	N/A	N/A
L1D_LD_MISSES	perf `L1-inst` load misses	N/A	N/A
L1I_ST_MISSES	perf `L1-inst` store misses	N/A	N/A
L1I_LD_MISSES	perf `L1-inst` load misses	N/A	N/A
L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data `L2` Hit Rqsts	0x24	0x01
L2_RQSTS.ALL_DEMAND_DATA_RD	Demand Data `L2` All Rqsts	0x24	0x03
LLC_STORE_MISSES	perf `L3` store misses	N/A	N/A
LLC_LOAD_MISSES	perf `L3` load misses	N/A	N/A

Table 5.1 `perf` and raw Hardware Performance Monitoring Counters used to characterize cache misses.

benchmarks are executed in bootstrap conditions, with no warm-up rounds before execution.

Execution locality is important on this experiment. In the same silicon package, the launcher program pins itself to a core while sets the execution of the benchmark's workload to a different one. Once that execution begins, the launcher program creates a special thread that constantly extracts the `perf_events` values.

Table 6.2 presents the counters and `perf` *macros* used to obtain cache misses values. The `perf` *macros* are a collection of hard-coded values committed to the Linux kernel by Intel developers inside of the `perf` subsystem. For `L2` cache misses there are no immediate `perf` macro or HWPC so we use demand's data as an estimate for the `L2` cache misses:

$$\texttt{L2}\_missess = all\_demand\_data\_\texttt{L2}\_requests - \texttt{L2}\_demand\_data\_hits.$$

We decided to use a sampling period of 500 ms. The responsible thread to read the values from the workload execution operates as follows: 1. An alarm is set for the thread to be awaken after the 500 ms. 2. Once awake, the thread sends a `SIGSTOP` signal to the workload and performs the syscall to stop the count, reads the values for the selected counters, and resets the counter mechanism. 3. When the values are read, a `SIGCONT` signal is sent to the workload to resume execution. Internal variables in the polling thread keep the interval and cumulative sum of all the counters. 4. Particularly for the retired instructions, if they are equal or above the 50 billion instructions, we halt the execution of the benchmark sending a `SIGTERM` signal. 5. To write to the file system the collected data, an additional thread is created in detached state. With this strategy, we delegate file system I/O critical sections to the OS.

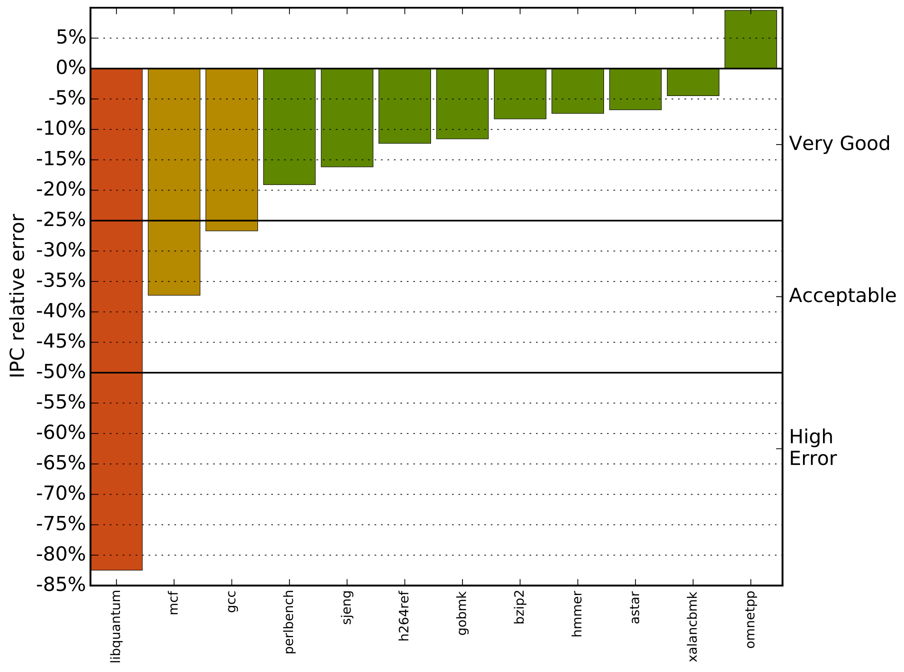


Figure 5.1 SPEC CPU2006 int benchmarks: ZSim+DRAMsim2 relative IPC error.

5.2 SPEC CPU2006 Performance Evaluation

In many computer architecture papers, IPC is the preferred metric to measure processor's performance [9]. We performed a traditional computer architecture evaluation using a variant of the IPC: the relative IPC error between measurements. In Figure 5.1 and Figure 5.2, we present the relative IPC simulation error having as the baseline the real hardware measurements. In both figures, X-axis is labeled with individual benchmark name, and Y-axis shows the relative IPC simulation error calculated as

$$IPC_{relative\_error} = \frac{IPC_{zsim} - IPC_{real}}{IPC_{real}}.$$

A negative IPC simulation error (IPC simulation error < 0) means that the simulator underestimates the IPC of a given benchmark, i.e., the benchmark is simulated to execute slower than on a real system. While a positive IPC simulation error (IPC simulation error > 0) means that the simulator overestimates the IPC of a given benchmark, i.e., the benchmark is simulated so that the executions seems to be performed faster than on a real system. Furthermore, we categorized the relative IPC error in three groups: *very good*, *acceptable*, *high error* when the relative IPC error ranges between 0 % and 25 %, 25 % and 50 %, or above 50 % respectively.

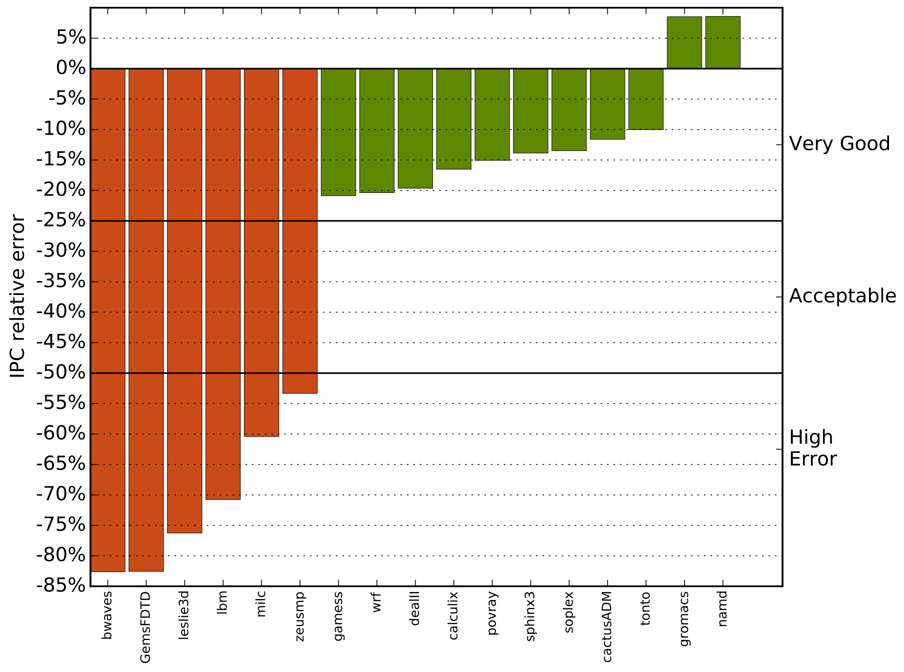


Figure 5.2 SPEC CPU2006 fp benchmarks: ZSim+DRAMsim2 relative IPC error.

The relative IPC error of the integer subset of benchmarks (SPEC CPU2006 int) is depicted in Figure 5.1. In the *very good* category we found 9 out of 12 benchmarks; from these, only one benchmark is also on the overestimating region (simulator perform faster than real hardware) with less than 10 % of absolute IPC error. In the *acceptable* region we found 2 out of 12 benchmarks and just one of benchmarks is in the *high error* category.

The relative IPC error of the floating point subset of benchmarks (SPEC CPU2006 fp) is depicted in Figure 5.2. The simulator shows fairly good accuracy. 11 out of 17 benchmarks falls into the *very good* category. For most of the benchmarks, 15 out of 17, the IPC error is negative, meaning that the simulated performance is slower than one measured on the real system.

An interesting remark is that all benchmarks categorized with high IPC relative error, are also categorized in the state-of-the-art with high bandwidth memory usage. For instance, in a previous study [148], `sphinx3` and `omnetpp` would be the exception for the rule; but in a recent study [116] they showed a higher demand in recent systems. This observation leads us to think that deviations are specifically tied to the memory subsystem, more likely to the main memory accesses.

5.3 SPEC CPU2006 Cache Miss Analysis

First, we analyzed the absolute difference of cache misses at every level of the caches in memory hierarchy using:

$$MKPI_{error} = MKPI_{LX\_Sim} - MKPI_{LX\_RealMachine}$$

where LX represents the analyzed `cache level`. As with the relative IPC error, we picked a 25 % cut-off value that allows us to identify if the metric is suitable for our study. The expected scenario for the $MKPI_{error}$ analysis is to have a difference below the threshold when the relative IPC error is in the very good or acceptable category, and the difference above the threshold when the relative IPC error is in categorized with high error. Otherwise, the $MKPI_{error}$ would be discarded as a useful metric to determine possible deviations between the systems.

Furthermore, we analyzed if the execution is bound to a particular cache level. For this purpose, we calculated the cache miss penalty per instruction (cMPI) for every level of the caches in the memory hierarchy:

$$cMPI = \frac{(\#misses_{zsim} - \#misses_{real}) * cache\_miss\_penalty}{\#instructions}$$

where the values for the penalties at different levels of the cache hierarchy are defined by the results of our micro-benchmark execution (Table 5.2). We continue to use the 25 % rule. If the cache miss penalty per instruction at a given cache level exceeded the threshold of 25 % the real machine's CPI, we considered that the specific cache level might contribute to any deviation of performance between the two systems.

Although cache miss penalty per instruction is a direct measure of an isolated instruction, it is not reflected broadly in the overall benchmark execution time (unless a complete dependency exists in the workload's code). For example, using values in Table 5.2, if a given instruction misses in `L1` and hits `L2`, it will have to wait around 12 cycles for the corresponding data in order to continue the execution. However, this does not mean that the processor will be stalled for these 12 cycles. While a

Memory Level	Latency (CPU cycles)	Miss penalty (CPU cycles)
`L1`	4	8
`L2`	12	28
`L3`	40	214
Main memory	254	N/A

Table 5.2 Cache-miss penalty for different levels of cache hierarchy collected through micro-benchmark execution.

Benchmark	L1d MPKI			L1i MPKI			L2 MPKI			L3 MPKI		
	ZSim	Real	Error	ZSim	Real	Error	ZSim	Real	Error	ZSim	Real	Error
`462.libquantum`	23.37	31.01	-7.64	0.02	0.02	-0.00	23.38	0.51	22.87	23.23	18.68	4.55
`429.mcf`	49.01	79.87	-30.87	0.32	0.19	0.14	44.02	2.24	41.77	13.98	30.24	-16.26
`403.gcc`	26.38	43.85	-17.48	3.44	1.43	2.01	23.08	15.21	7.87	1.49	6.13	-4.65
`400.perlbench`	3.71	6.43	-2.71	3.97	3.50	0.47	1.70	0.94	0.75	0.07	0.91	-0.84
`458.sjeng`	1.50	4.22	-2.72	0.65	0.21	0.44	0.65	0.19	0.47	0.43	0.60	-0.17
`464.h264ref`	3.05	4.88	-1.83	0.37	0.16	0.20	1.77	0.30	1.47	0.01	0.53	-0.52
`445.gobmk`	4.35	8.41	-4.06	7.58	4.28	3.30	2.11	1.15	0.96	0.39	0.78	-0.38
`401.bzip2`	14.56	20.18	-5.63	0.11	0.05	0.06	7.69	2.57	5.12	0.23	1.41	-1.18
`456.hmmer`	5.48	9.47	-3.99	0.11	0.04	0.07	3.30	0.15	3.15	0.02	0.11	-0.09
`473.astar`	26.45	53.49	-27.03	0.18	0.09	0.09	19.84	2.21	17.63	0.92	19.23	-18.31
`483.xalancbmk`	24.74	33.25	-8.51	4.62	2.63	1.98	24.07	1.81	22.26	0.06	12.47	-12.41
`471.omnetpp`	29.54	41.19	-11.65	1.51	0.47	1.05	28.49	1.56	26.94	1.75	12.31	-10.56

Table 5.3 SPEC CPU2006 int benchmarks. MPKI error for all cache levels.

given instruction is waiting for data, the OoO engine executes other (independent) instructions, mitigating the stalled cycles derived from cache misses in the overall execution time. When compared to the overall CPI metrics at every level of the memory hierarchy, the cache miss penalty per instruction provides an idea about the cache level that impacts the most in the workload execution.

5.3.1 SPEC CPU2006 integer set

Table 5.3 presents the MPKI analysis of the SPEC CPU2006 int portion of the benchmarks. The benchmarks are listed in the same order as in Figure 5.1, from lowest (negative) to the highest (positive) simulation error. Following the 25 % rule, we shaded the values beyond that threshold to easily identify them. Moreover, we use a horizontal line to distinguish from the previous simulation error classification: high simulation error (`462.libquantum`), acceptable (`429.mcf`, `403.gcc`), and very good accuracy (`400.perlbench` to `471.omnetpp`).

Since most of the benchmarks are beyond the 25 % rule, we cannot conclude that the MPKI analysis for the integer type of benchmarks is helpful to understand the differences between the two systems.

Furthermore, in Table 5.4 the cache miss penalty per instruction bounds are shown. We use a horizontal line to separate benchmarks with high simulation error (`462.libquantum`), acceptable (`429.mcf` and `403.gcc`) and very good accuracy (`400.perlbench` up to `471.omnetpp`). In the last column of the table, we also included the simulated CPI of the benchmarks.

On `462.libquantum` comparison, the most impacted cache level is `L3`. On the rest of benchmarks, with the exception of `403.gcc`, `483.xalancbmk` and `471.omnetpp` in the `L2`,

Benchmark	cache miss penalty upper bound				Sim CPI	Real CPI	Sim/Real CPI
	L1d	L1i	L2	L3			
`462.libquantum`	0.03	0.00	0.10	0.33	2.81	0.53	5.30
`429.mcf`	0.06	0.00	0.15	0.16	3.59	2.78	1.29
`403.gcc`	0.10	0.01	0.27	0.06	1.05	0.99	1.06
`400.perlbench`	0.02	0.02	0.03	0.00	0.67	0.60	1.12
`458.sjeng`	0.01	0.00	0.01	0.02	0.89	0.75	1.19
`464.h264ref`	0.03	0.00	0.05	0.00	0.46	0.41	1.12
`445.gobmk`	0.02	0.03	0.03	0.02	0.97	0.86	1.13
`401.bzip2`	0.07	0.00	0.12	0.01	0.79	0.79	1.00
`456.hmmer`	0.04	0.00	0.07	0.00	0.54	0.49	1.10
`473.astar`	0.06	0.00	0.13	0.02	1.77	1.83	0.97
`483.xalancbmk`	0.11	0.02	0.33	0.00	0.87	1.53	0.57
`471.omnetpp`	0.11	0.01	0.33	0.07	1.05	1.52	0.69

Table 5.4 SPEC CPU2006 int benchmarks: cache miss penalties for all cache levels.

no other level is affected. This reading appears congruent with the relative IPC error and the link of memory bandwidth utilization [116, 148]; `462.libquantum` is known for high memory bandwidth utilization, `471.omnetpp` and `403.gcc` is around half of memory bandwidth utilization, and `483.xalancbmk` barely uses the memory bandwidth.

In the integer set of benchmarks, the cache miss per instruction metric seems promising, but since there is no clear distinction between `403.gcc`, `483.xalancbmk` and `471.omnetpp`, we can conclude that it is representative only for the `L3`.

Let's remark that the ratio between the simulated CPI and the real CPI is only higher than 1.29 for `462.libquantum` which is also catalogued as with high relative IPC error; the rest of the benchhmarks, catalogued as acceptable or good relative IPC error, are below this value.

5.3.2 SPEC CPU2006 floating-point set

Table 5.5 present the MPKI analysis of the SPEC CPU2006 fp portion of the benchmarks. The results are sorted as they appear in Figure 5.2. Following the 25 % rule, we shaded the values beyond that threshold to easily identify them. Moreover, we use a horizontal line to separate benchmarks with high simulation error (`410.bwaves` to `434.zeusmp`) and low error (from `416.games` to `444.namd`). Just as the integer counterpart, we cannot conclude that the MPKI analysis for the floating-point type of benchmarks is helpful to understand the differences between the simulator and the real system.

The benchmarks are listed in the same order as in Figure 5.2, from lowest (negative) to the highest (positive) simulation error. We use a horizontal line to separate benchmarks with high simulation error (`410.bwaves` to `434.zeusmp`) and low error (`416.games` to `444.namd`). For each level of cache we shade the cache miss penalty error fields with

Benchmark	L1d MPKI			L1i MPKI			L2 MPKI			L3 MPKI		
	ZSim	Real	Error	ZSim	Real	Error	ZSim	Real	Error	ZSim	Real	Error
`410.bwaves`	35.47	36.98	-1.51	0.17	0.06	0.11	20.48	1.93	18.55	18.73	11.11	7.62
`459.GemsFDTD`	30.75	47.34	-16.60	0.28	0.09	0.19	24.89	1.68	23.20	17.75	15.18	2.57
`437.leslie3d`	33.54	48.45	-14.90	0.36	0.04	0.32	24.63	2.39	22.25	14.47	9.72	4.75
`470.lbm`	50.43	92.70	-42.28	0.56	0.05	0.51	31.00	6.87	24.13	15.68	8.46	7.22
`433.milc`	9.12	15.12	-6.00	0.03	0.05	-0.02	9.08	2.98	6.09	8.88	13.88	-5.01
`434.zeusmp`	22.43	35.05	-12.62	0.13	0.03	0.10	6.84	1.22	5.62	5.02	3.72	1.30
`416.gamess`	2.28	6.19	-3.91	0.41	0.42	-0.02	0.14	0.10	0.05	0.00	0.03	-0.03
`481.wrf`	3.46	5.24	-1.78	0.24	0.10	0.14	2.14	0.57	1.56	0.46	0.29	0.17
`447.dealII`	4.96	6.03	-1.07	0.16	0.08	0.08	1.87	0.27	1.61	0.13	0.44	-0.31
`454.calculix`	4.31	5.51	-1.20	0.10	0.05	0.05	2.68	0.15	2.53	0.14	0.24	-0.10
`453.povray`	9.47	14.44	-4.96	0.14	0.23	-0.09	0.01	0.12	-0.11	0.00	0.03	-0.02
`482.sphinx3`	16.16	18.82	-2.65	0.28	0.07	0.22	13.52	0.44	13.08	0.02	4.51	-4.48
`450.soplex`	40.35	61.71	-21.36	0.44	0.17	0.27	34.70	7.73	26.96	0.72	10.68	-9.96
`436.cactusADM`	8.19	24.27	-16.08	0.35	0.06	0.29	5.63	1.08	4.56	2.62	2.99	-0.37
`465.tonto`	3.11	5.40	-2.28	1.03	0.82	0.21	0.48	0.21	0.26	0.00	0.19	-0.19
`435.gromacs`	8.21	13.87	-5.66	0.09	0.04	0.05	1.59	0.10	1.49	0.01	0.22	-0.21
`444.namd`	7.12	12.15	-5.03	0.01	0.03	-0.01	0.18	0.06	0.12	0.02	0.11	-0.09

Table 5.5 SPEC CPU2006 fp MPKI error for all cache levels.

values higher than 25 % of the simulated CPI. Thus, is an arbitrary threshold selected to emphasize the cache miss penalty bounds that we find important to consider.

Despite the fact that the first six listed benchmarks which correspond those on the *high error* category do not show a violation on our arbitrary 25 % rule, we can notice that their `L3` CPI bound are higher than the rest of the benchmarks with the sole exception of `436.cactusADM`, which is in the *very good* category. Moreover, all six benchmarks belong to the high memory bandwidth utilization [116, 148].

Only two of the benchmarks (`482.sphinx3` and `450.soplex`) show a possible source of error based on these cache miss penalties per instruction not in the `L3`, but their relative IPC error is under the *very good* category. Therefore, we can only conclude that the cache miss penalties per instruction metric is only suitable for the `L3`.

Since `L3`'s cache miss penalties range between 15.5 % and 21.5 % with an average of 18.8 %, and the rest of the benchmarks range between 1 % and 13 %, the threshold for the cache miss penalties might be adjusted to the minimum of those categorized with high relative IPC error: 15 %.

Benchmarks with ratio between the simulated CPI and the real CPI higher than 1.29 are catalogued (as their integer counterpart) with high IPC error. The ratio between the simulated CPI and the real CPI seems to provide a hint on where the deviations exists.

Benchmark	cache miss penalty upper bound				Sim CPI	Real CPI	Sim/Real CPI
	L1d	L1i	L2	L3			
`410.bwaves`	0.03	0.00	0.05	0.17	4.65	0.79	5.89
`459.GemsFDTD`	0.03	0.00	0.06	0.15	4.60	0.81	6.91
`437.leslie3d`	0.04	0.00	0.09	0.17	3.44	0.81	4.25
`470.lbm`	0.06	0.00	0.12	0.20	3.20	1.07	2.99
`433.milc`	0.02	0.00	0.04	0.15	2.42	0.95	2.55
`434.zeusmp`	0.06	0.00	0.05	0.13	1.59	0.74	2.15
`416.gamess`	0.02	0.00	0.00	0.00	0.58	0.45	1.29
`481.wrf`	0.02	0.00	0.04	0.03	0.59	0.46	1.28
`447.dealII`	0.03	0.00	0.04	0.01	0.59	0.47	1.26
`454.calculix`	0.04	0.00	0.07	0.01	0.47	0.39	1.21
`453.povray`	0.07	0.00	0.00	0.00	0.58	0.48	1.21
`482.sphinx3`	0.11	0.00	0.27	0.00	0.61	0.63	0.97
`450.soplex`	0.15	0.00	0.38	0.03	1.11	1.28	0.87
`436.cactusADM`	0.04	0.00	0.08	0.12	0.90	0.91	0.99
`465.tonto`	0.02	0.01	0.01	0.00	0.57	0.52	1.10
`435.gromacs`	0.05	0.00	0.03	0.00	0.67	0.72	0.93
`444.namd`	0.06	0.00	0.00	0.00	0.51	0.54	0.94

Table 5.6 SPEC CPU2006 fp benchmarks: cache miss penalties for all cache levels.

5.4 Summary

The MPKI difference between the compared systems is not helpful to identify the performance deviations between the real system and the simulator infrastructure. When using MPKI difference to analyze the possible deviation on both integer and floating-point workloads of the SPEC CPU2006 benchmark suite, the majority of workloads present more than 25 % of deviation with respect of the real machine's measurements for the same cache level. Although the latency is accurately modeled in the simulator, we can only conclude that the behavior of the caches is different in both systems.

Furthermore, the cache miss penalty per instruction is not consistent throughout the different cache levels, being the LLC the only level where it shows consistency. Although it is an important information, it does not differentiate if the differences between the systems are part of the memory subystem or the CPU model.

Lastly, we also showed that the CPI ratio is consistent and supportive since it correlates the difference between the two systems. When a heterogeneous set of instructions is evaluated through the CPI metric, it provides an overall projection of the latency in those instructions. Therefore, we decided to further explore the CPI metric through a method that unfolds micro-architectural details from it.

CHAPTER 6

Top-Down Method and the Retirement Factor

The CPI metric captures overall execution errors. In the previous chapter we found that isolating CPI error bounds per cache-level was not conclusive to find the source of differences between our systems under test, but the CPI ratio between the systems leans towards a path to explore. CPI stacks [48] are a common way to find insights of a workload execution. In this chapter we use a well-known CPI stack analysis to conduct micro-architectural comparison as well as a proposal to the method so that it helps to identify the causes of deviation in main memory simulation between the compared systems.

6.1 The Top-Down Method

Top-Down [146] is a method to conduct performance evaluation. Designed to identify bottlenecks in modern OoO processors, it categorizes application performance in four main groups: *Frontend bound*, *Bad speculation*, *Retiring*, and *Backend bound*. Applying the method and comparing the processed measurements between the real machine and the simulator, we could identify the sources of error between the systems under test.

Top-Down was designed by an Intel researcher, therefore it is targeting an Intel micro-architecture. Ivy Bridge was the chosen micro-architecture used in the research paper. Hence, the conceptualization of a modern OoO CPU was made using `x86` abstractions [69]. In Top-Down, the CPU engine is broken into two major portions: frontend and backend.

The frontend is in charge of decoding instructions taken from the memory and translating them into simple operations called micro-operations (uops). The `x86` ISA is a variable length opcode architecture raging from one up to 15 bytes in length. In the frontend, a pre-decoder, an instruction queue, and specialized decoders are set to recognize and decompose every instruction into one or several micro-operations. Some of the instructions are micro-sequenced: the execution depends on different parameters encoded in the instruction's opcode such as the size and value of the input and output registers. For these instructions, the specialized decoders generate the necessary micro-operations.

The backend executes and retires the work generated as the outcome of scheduling the micro-operations. Within the execution pipeline, the latency for the different types of micro-operations is not fixed. For the frontend to deliver micro-operations, the backend has to free up space in the 168-entry ROB. On the execution stage, a scheduler decides the next micro-operations to be executed, where the execution engine is driven by 6 specialized ports (execution units): port 0 and port 1 are exclusively reserved for arithmetic operations while port 5 also includes branch calculation; port 2 and port 3 are used as Address Generator Units (AGUs) for load and store operations and to load data. Finally, port 4 is only used to store data.

The place where the frontend feeds the backend with micro-operations is known as the *issue point*. Our targeted micro-architecture, Sandy Bridge, is a 4-slot issue-point design. On a simplified CPU architecture, the issue-point has a unique slot, meaning that the frontend can deliver to the backend up to one micro-operation per cycle. On the one hand, if the frontend is ready to deliver a new micro-operation but the slot is not available, it means the backend has not freed the slot from the previous micro-operation. In Top-Down, the time consumed in this transaction would be categorized as *Backend bound*. On the other hand, if the slot is ready but frontend is unable to fill the slot, then the time spent in the transaction would be categorized as *Frontend bound*.

On the backend side, issued micro-operations that are retired at the end of the pipeline are the ones that correspond to the useful pipeline work; Top-Down classifies the time spent in these transactions as part of the *Retiring* category. However, some issued micro-operations are not retired, *e.g.,* because they are part of the miss-predicted branch path. As the branch predictor unit is located in the frontend, generating more micro-operations to keep under pressure the backend translates into less free slots at the issue-point. The difference (the cost in cycles) between the retired micro-operations and the throughput at the end of the pipeline is categorized as *Bad speculation*.

Top-Down builds a hierarchical tree with subcategories beneath the main four. For example, the *Backend bound* category spawns a tree over two categories: *Core* and

Name	Description	Event	Umask
CPU CYCLES	perf-list cycles macro	0x3C	0x01*
INSTRUCTIONS	perf instructions macro	0xC0	0x00*
UOPS_ISSUED.ANY	uops RAT -> RS p/cycle	0x0E	0x01
UOPS_RETIRED.RETIRE_SLOTS	uops retired slots used p/cycle	0xC2	0x02
IDQ_UOPS_NOT_DELIVERED.CORE	uops slots not delivered when no stall in BE	0x9C	0x01
INT_MISC.RECOVERY_CYCLES	stall cycles after machine clears	0x0D	0x03

Table 6.1 Linux perf and raw Hardware Performance Monitoring Counters for Sandy Bridge used for the Top-Down analysis.
RAT: Resource Allocation Table. RS: Reservation Station

Memory. Reciprocally, *Core* splits into *Divider* or *Execution Ports Utilization* while *Memory* breaks for *Stores*, *L1*, *L2*, *L3* and *External Memory*.

6.2 Top-Down Implementation

In the Top-Down's research paper, the corresponding author had given a conceptual approach for each category. From the conceptual description along with Intel's counter definition, we have applied the Top-Down method using the HWPC available in the Sandy Bridge micro-architecture, and implemented an approximate model of the HWPC in the simulator; nevertheless, we did it for the main four categories[1].

Table 6.1, presents the HWPC used for Top-Down analysis on Sandy Bridge. On the simulator, our implementation was done as follows. CPU CYCLES and INSTRUCTIONS are counters natively found in the simulator. The UOPS_ISSUED counter is updated after the decoding stage had been done since the number of micro-operations to be delivered from the frontend to the backend is available. The number of micro-operations that *do* retire is available after asserting if the branch predictors had performed a good estimation, meaning that UOPS_RETIRED is the micro-operations that would have been in branch speculation substracted from the number of total micro-operations. In fact, the INT_MISC.RECOVERY_CYCLES counter is updated by subtracting the cycles spent in the branch speculation from the current simulated cycles. Lastly, we rely on the simulator's micro-operation scheduler to update the IDQ_UOPS_NOT_DELIVERED counter; if there are no stalls in the backend, but the frontend could not update the count of delivered micro-operations, the counter is

[1]Further engineering effort must be considered for implementing the remaining counters if the full Top-Down Method is needed.

Name	Description
Total Slots	CPU_CLK_UNHALTED.THREAD * 4
Slots Issued	UOPS_ISSUED.ANY
Slots Retired	UOPS_RETIRED.RETIRE_SLOTS
Fetch Bubbles	IDQ_UOPS_NOT_DELIVERED.CORE
Recovery Bubbles	INT_MISC.RECOVERY_CYCLES * 4
Retiring	Slots Retired / Total Slots
Bad speculation	(Slots Issued - Slots Retired + Recovery Bubbles) / Total Slots
Frontend Bound	Fetch Bubbles / Total Slots
Backend Bound	1 - (Frontend Bound + Bad speculation + Retiring)

Table 6.2 Top Down Category definitions and association with HWPC needed.

updated. ZSim is an approximate simulator, any attempt to mimic specialized HWPC will require a full refactor of current simulator's code. Regarding the objective to determine the differences between a simulator infrastructure and a real system, the four base categories are sufficient for our study.

In the real machine using a Linux kernel 3.0, the `perf-list` macros used for cycles and instructions translate[2] into the needed counter: CPU_CLK_UNHALTED.THREAD is translated to a Processor Event-Based Sampling (PEBS) event [70] if the *precise bit* is set for the 'cycles' macro; for the 'instructions' macro, INST_RETIRED.ANY is indirectly used to occupy a second PEBS event. PEBS is an Intel technology that allows sampling access to HWPC and internal CPU execution state with high accuracy and low overhead; we are not using PEBS explicitly, but the kernel indirectly uses it to optimize the read of specific HWPC. Next, Table 6.2, depicts the conversion and operations needed to perform the Top-Down analysis.

As we can intuit from the bottom section on Table 6.1, all metrics share a common denominator: the *Total Slots* abstraction. The abstraction translates into the total number of unhalted cycles of the execution thread multiplied by 4-slots, which is the maximum theoretical value for the amount of micro-operations delivered at the issue-point. On the numerator, the *Retiring* and *Frontend bound* are measured in slots at a given point in the pipeline. The *Bad speculation* and the *Backend bound* are expressions resulting as an arithmetic expression from the same units. The Top-Down units are expressed as:

$$TopDown_{units} = \frac{\frac{slots}{cycle}}{cycles{*}4\frac{slots}{cycle}} = \frac{1}{cycles}.$$

[2]See `event_constraint` and `intel_snb_event_constraints` definitions in `arch/x86/kernel/cpu/{perf_event,perf_event_intel}.c` from Linux kernel 3.0 source code [93].

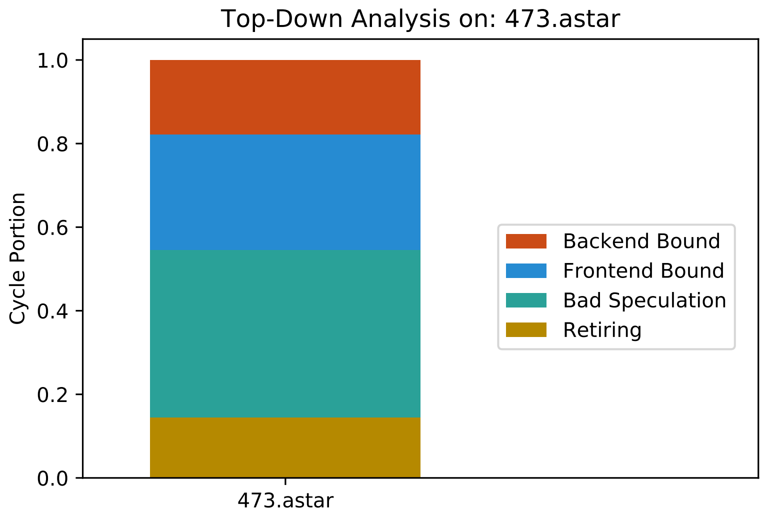


Figure 6.1 Top-Down analysis of the 4 main categories for `473.astar`.

Because the Top-Down is performed after an execution, the resulting tuple of values is an overall representation of how much of a cycle is spent in each one of the main Top-Down categories. Since the slots are used by micro-operations which are transitively the result of decoding instructions, we can say that Top-Down is a special type of a CPI stack. As an example, in Figure 6.1 we present the Top-Down analysis for the SPEC `473.astar` benchmark after a 50 billion instructions execution. The X-axis is just used to horizontally locate the values corresponding to the benchmark. On the Y-axis, 1-overall-cycle is presented from [0 to 1.0]. From Table 6.1 we can see that the values are normalized to 1.0 (Backend bound is expressed: $1 - sum\_of\_the\_others$). To show the powerful insights provided by Top-Down, for our example `473.astar`, we can notice that the Bad speculation is dominating most of the waiting times; a known behavior explored manually in previous research [110].

6.3 Top-Down for Micro-architecture Comparison

In the research publication, Top-Down is mentioned to act as a tool for micro-architecture comparison. Although our simulator attempts to mimic the targeted micro-architecture, we used the Top-Down to dig into the possible locations (Top-Down categories) where the deviations between the systems are present.

Figure 6.2 shows the Top-Down comparison between the real-system measurements and the simulator. At this point, we are not specifying any particular order for the

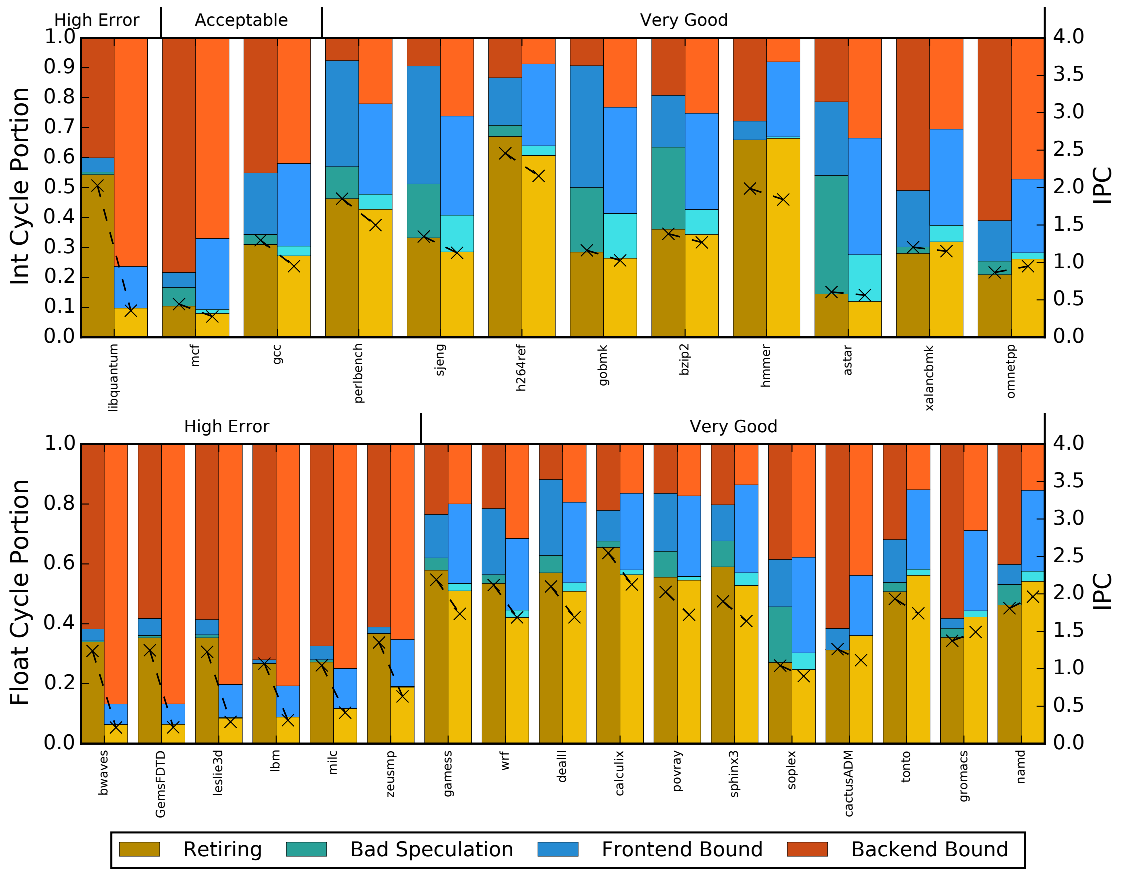


Figure 6.2 Top-Down analysis and IPC comparison. For each benchmark, left bars correspond to real-system measurements, while the right bars correspond to the simulator.

benchmarks. The upper plot shows the Top-Down breakdown for the integer set of benchmarks, while the lower plot shows the floating point set of benchmarks. We considered the plot to be powerful, as it presents condensed information of the overall execution. When used for micro-architectural analysis, side to side comparison allows fast visual inspection of the treated categories. Information on how to read this plots lists as follows:

1. For all experiments, two bars are presented: the left bar for the real system and the right bar for the simulated environment.

2. The X-axis lists the benchmark used in the study.

3. The right-Y-axis are normalized to 1.0 as they represent the per-cycle portion of the given execution.

4. The left-Y-axis are scaled up to 4.0 which means the maximum IPC achievable by the real system.

5. IPC is shown through the black crosses overlapped on each bar. A black-dashed line has also been drawn to visualize the difference between the two obtained metrics.

In Figure 6.2, we can note that for *very good* labeled benchmarks, the comparison between Top-Down categories is acceptable. As the retirement categories are approximately the same we could trust that all other categories are comparable. Unfortunately, we could not state the same for the *high error* labeled benchmarks.

A major drawback using the Top-Down method is that it summarizes the execution in a normalized environment, meaning that the sum of all of its members projects from 0 to 1.0 (in other words, from 0 % to 100 % of an overall cycle execution). For example, if we compare two Top-Down results from the same execution in slightly different conditions where all Top-Down categories are modified, we will not know which one(s) triggered the differences, either because one(s) has grown forcing the others to shrink or vice-versa. To deal with this ambiguity, we propose an extension to the Top-Down method.

6.4 The Retirement Factor

Our proposal to the Top-Down method for micro-architectural comparison requires to limit the execution of a target workload to a specific amount of instructions. In our work, we set up all benchmarks to execute the same number of instructions (50 billion). Our assumption states that since the number of executed instructions are the same in both systems (the real machine and the simulator infrastructure), it also should be the same for the micro-operations that eventually retire. Lets remember that the Top-Down's Retirement category counts the delivered micro-operations which eventually do retire [146]. The assumption is valid because we integrate the differences found for the execution units corresponding to the micro-benchmark execution for the 358 cases of the `x86` ISA. So the number of micro-operations generated by a given instruction is already incorporated into the CPU simulator. Those micro-operations which were not retired because of a bad speculation are not taken in the final count. Therefore, for any given workload in the 50 billion instruction window, the *Retiring* category should be approximately the same for the real system and the simulation

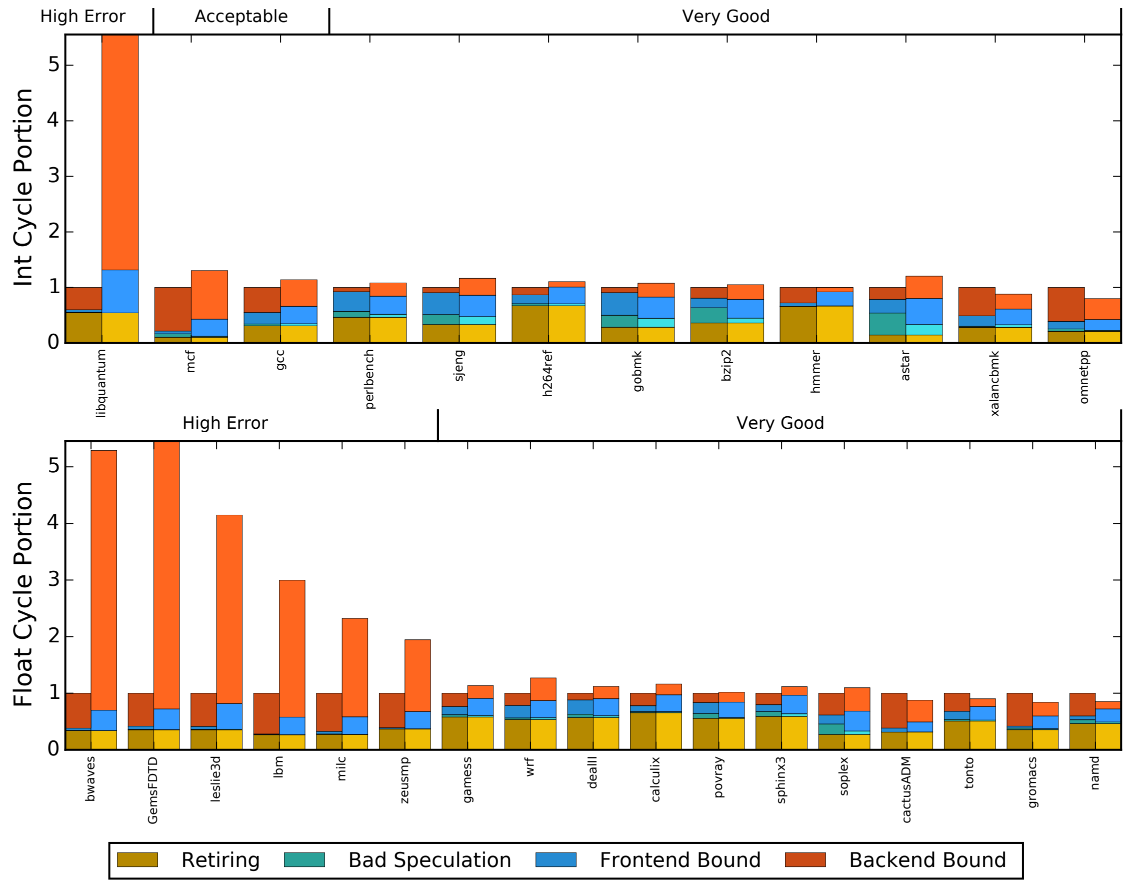


Figure 6.3 Top-Down analysis using the *retirement* factor. For each benchmark, left bar corresponds to real-system measurements while the right bar corresponds to the simulator.

infrastructure. We define the *retirement factor* as the ratio between the *Retiring* category in the real system and the simulation infrastructure:

$$retirement\ factor = \frac{Retirement\ category_{real\ machine}}{Retirement\ category_{simulation\ infrastructure}}.$$

Using the *retirement factor*, we proceeded to scale all the Top-Down categories. In a value-to-value comparison, this means that the *retirement* category will match the same value, but the rest of categories will scale dynamically. As an example, in Figure 6.3, we can notice that the lower level of the stacked bar (the *Retiring* category) is kept the same for all experiments but not in the upper layers of the bars (particularly in the *Backend bound* category).

The *Backend bound* category seems to be the culprit between the two systems. It is affected by the *retirement* factor with up to 9 and 8 times for the integer and floating-point benchmarks accordingly. An important remark linked with the previous research, is that with the exception of `482.sphinx3` and `450soplex`, all *high error* categorized benchmarks are documented as high-bandwidth utilization [148]. The *Frontend bound* category seems to be affected as well with up to 4 times the scaling factor. Unfortunately, even in the Top-Down research paper, it is mentioned that *Frontend bound* is somehow misleading. *Bad speculation* category seems not to play a significant role in the *high error* classified benchmarks. Moreover, it does not seem to be affected by the normalization factor, meaning that the branch predictors in the simulator are a rightful representation of the actual hardware with the exception of `434.zeusmp` that holds a scaling factor of 4 times with respect to the original value.

6.5 Summary

Although the Top-Down method is a tool to conduct an application's performance analysis, it is also well-suited to perform micro-architectural comparison, including simulator's performance. In this chapter, we presented the *retirement factor*, an extension to the Top-Down method that allows us to identify the most suitable micro-architectural section responsible for the differences between a real system and a simulator infrastructure. The *retirement factor* accentuates the differences for each one of the Top-Down categories providing the researcher with a new tool to measure the differences between the systems under test. In our evaluation, the major differences are found in the *Backend bound* category.

Since the benchmarks where the *retirement factor* has a larger magnitude are found in the state-of-the-art as the ones with more interactions with the main memory, the major contributor of the differences must be on the memory subsystem.

CHAPTER 7

The Delay Queue and Top-Down Evaluation

In this chapter, we propose an architectural modification to the memory controller (the `Delay Queue`) that allows us to inject a delay for every memory transaction that reaches the main memory. Let's remember that there are two micro-architectural components that alter the amount of memory transactions during workload execution but are not modeled by ZSim: the hardware prefetchers and the TLBs. To measure the impact of not having neither of these two features in our study, we split the current evaluation setting the real machine in two different scenarios: 1. with the hardware prefetchers **disabled**, and 2. with the hardware prefetchers **enabled**. For the TLB, we measured the address translation impact by using HSCC (a ZSim extension) [94], where a first layer of address translation is modeled in the simulator infrastructure.

Although we were unable to replicate the proper HWPC in the simulator infrastructure to determine the full set of Top-Down categories under the *Backend-bound* branch, we managed to control the external parameters that affect workload behavior; particularly those that affect memory performance. In the real machine, we can alter the amount of memory transactions by disabling and enabling the hardware prefetchers; in the simulator infrastructure, we can modify latency parameters of main memory access including our `Delay Queue` proposal.

The before-mentioned scenarios create a difference in the amount of memory transactions that reached the main memory, jointly with the *retirement factor* allows us to determine the source of deviations between the systems under test.

Description	`mem.latency`	`Delay Queue`
Df.0 - Default configuration	100	0
Df.1 - Only `mem.latency`	170	0
PQ.0 - `Delay Queue` as 15 ns	170	24
PQ.1 - `Delay Queue` as 60 ns	170	100

Table 7.1 Different simulator parameter proposals.

7.1 The Delay Queue

We have enhanced the DRAMsim2 memory controller with a naive model to insert delay cycles in the main memory simulation. We named this proposal as the `Delay Queue`. We categorized the `Delay Queue` per memory request in one of the following scenarios. 1. The request is traversing the Network on a Chip (NoC) to the memory controller, the request is notably longer for write transactions. 2. The request is arbitrating for the right to be en-queued in the memory controller's request queue. 3. The request is potentially stalling if one of the transactions queues are full. 4. At the end of a read request, the multi-cycle cost of transferring the data over the NoC between the memory controller and the core.

To find the best possible theoretical value for DRAMsim2's `Delay Queue`, we use the same strategy to propose values for the ZSim's *mem.latency* parameter. From 70 ns of the worst-case machine latency, we subtract 56 ns from the ZSim's `mem.latency` fixed parameter. These 14 ns should be expressed in the DRAM clock domain (800 MHz), turns into 22.4 clock cycles. To use square numbers, we decided to use 15 ns that becomes 24 DRAM clock cycles.

Furthermore, a remarkable feature using `Delay Queue` in this analysis is that it creates *a bubble* in the simulation environment affecting the Top-down analysis as follows: if the Backend bound category increases, the problem is directly related to the memory subsystem; if the Backend bound category decreases or has little change, we can say that it is a problem of the execution engine. Using the *retirement factor* these changes should be easier to spot.

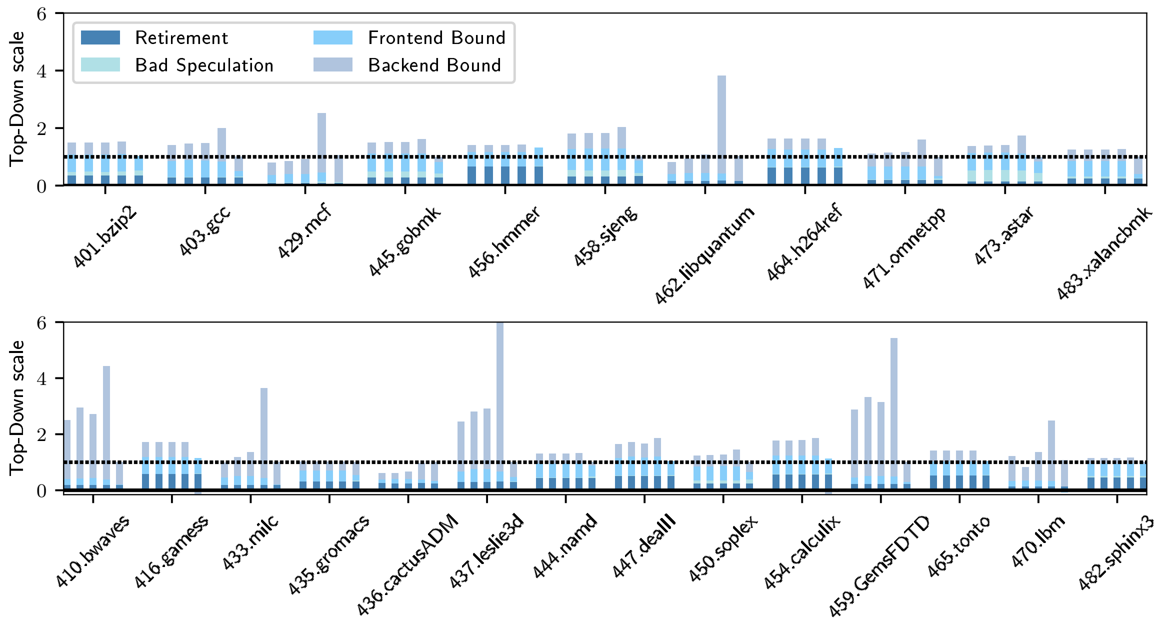


Figure 7.1 *Reriremement factor* Top-Down analysis excluding overheads of hardware prefetchers.

7.2 SPEC CPU2006 Analysis Using Top-Down and the Retirement Factor

The SPEC CPU2006 suite [65] is a well-known collection of real-world benchmarks. We executed and evaluated the simulator infrastructure enhancements on a set of eleven integer and fourteen floating-point benchmarks from the suite.

Figure 7.1 and Figure 7.2 are divided into two sections: integer and floating point set of benchmarks. The upper plot belongs to the integer category and the lower plot belongs to the floating-point category. On the X-axis, we use the canonical name of the SPEC CPU2006 benchmarks. On the Y-axis, the Top-Down scale is used. For each benchmark in the X-axis, we plot five bars representing a combination of simulator parameters. Table 7.1 summarizes these configurations. The order of the bars in both figures from left to right is: `Df0`, `Df.1`, `PQ.0`, `PQ.1`. The 5^{th} bar corresponds to the real machine, this is the reason why it is bound to 1.0.

In Figure 7.1, the *retirement factor* helps us to effortlessly identify two cases of interest in the integer category and five cases in the floating-point category. For memory intensive benchmarks, simulator enhancements based on the changes in the `mem.latency` ZSim parameter (2^{nd} and 3^{rd} bar) show moderate changes with respect to the default ZSim + DRAMsim2 configuration (1^{st} bar). The simulator enhancements based on the `Delay Queue` in the DRAMsim2 (4^{th} bar), show much larger differences.

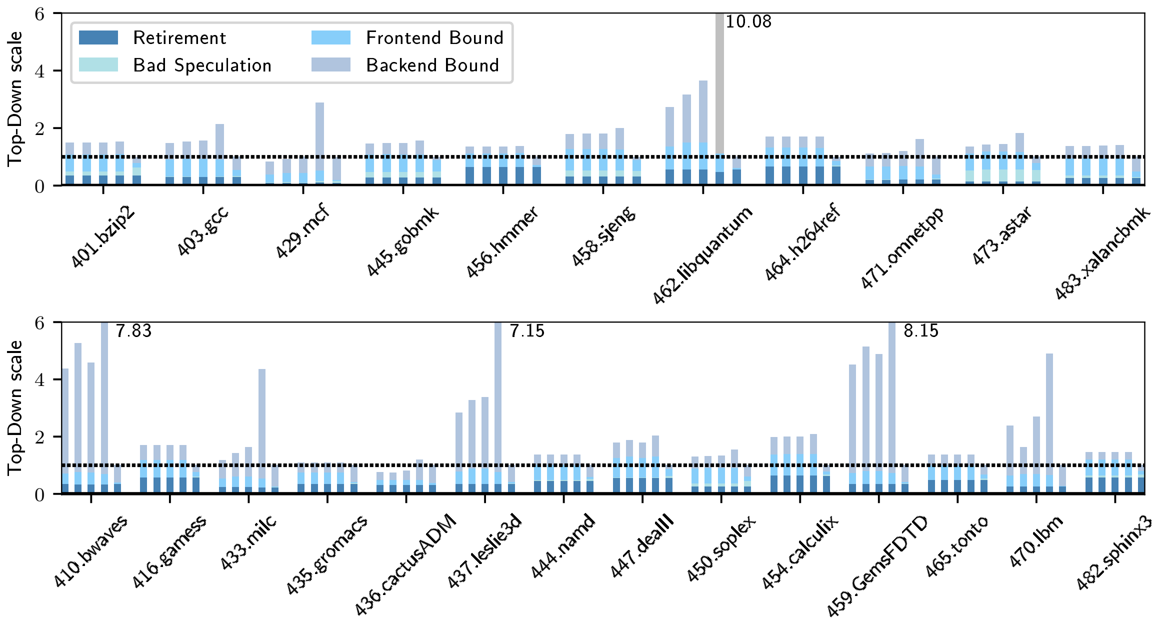


Figure 7.2 *Retirement factor* Top-Down analysis including overheads of hardware prefetchers. Regarding Figure 7.1, the *retirement factor* helps to identify the benchmarks which are sensitive to prefetcher intervention.

We also detected a huge difference in the Backend bound issue stalls between the real platform (5th bar) and all the simulator configurations.

For the integer set, `429.mcf` and `462.libquantum` show a substantial difference in the 4th bar (`Delay Queue`-only scenario). The rest of configurations seems to be stable across all the integer benchmarks, ranging between 1.5 and 2.2 in the Top-Down scale. For the floating-point set, `410.bwaves`, `437.leslie3d`, and `459.GemsFDTD` present a larger difference not only in the 4th bar but also in the remaining of the simulation configurations. An interesting behavior, similar to the integer set, is found in `433.milc` and `470.lbm` that present a moderate increase in the `Delay Queue`-only scenario ranging between 2.0 and 3.0 in the Top-Down scale.

Following our assumptions, we proceeded to enable the hardware prefetchers in the real machine and compare the measurements with the simulation infrastructure. Figure 7.2 presents the results of this comparison. A striking similitude from Figure 7.1, is that for most of the benchmarks, there is a low-to-moderate difference between the different simulator configurations, mainly from the fact that these benchmarks have low memory usage [36]. Moreover, the most impacted benchmarks remain the same as in Figure 7.1.

Between the two comparisons, the Top-Down scale grew on average 0.4 for the integer benchmarks and 0.27 for the floating-point category. For the most impacted benchmarks, the increment in the Top-Down scale for the Backend category between the two real machine scenarios behave on average as follows: 3.62 for `462.libquantum`, and 1.76 , 1.42 and 1.70 for `410.bwaves`, `437.leslie3d` and `459.GemsFDTD` respectively.

Moreover, we can notice that for benchmarks `458.sjeng` on the integer set, `410.bwaves`, `447.dealII`, `459.GemsFDTD`, and `470.lbm` for the floating-point set the 3^{rd} bar (`DQ=24` and `mem.latency=170`) there is a decrement on the *backend* category. This result leads us to think that a discrepancy exists in the execution engine and not solely in the memory simulation.

7.3 Impacts of Hardware Prefetchers and Address Translation

Modern OSs use the Virtual Memory (VM) technique to provide multiple processes execution and shared resources access on the same computer. Process' code and data segments belong to a unique given address space in which each location is known as Virtual Memory Address (VMA). For the running process to reach resources on the machine, the VMAs must be translated into physical locations. All mappings between VMAs and their specific physical locations must be kept for recurrent processes' look-ups. The more resources the process would use, the larger the mapping storage must be. The mapping storage is known as the Address Translation Table (ATT) and is usually located in main memory. If an application constantly accesses the ATT its performance will degrade. The procedure of transforming a processes' virtual addresses into the computer's physical location in main memory is known as Virtual-to-Physical Address Translation (V2PAT). Nowadays in a high-end general-purpose architecture, V2PAT is a process that is achieved jointly between the OS and the hardware through the operation of the Memory Management Unit (MMU). The TLB is the component of the MMU that speeds up the V2PAT acting as an additional memory cache that serves solely for the translation purposes. When the TLB does not hold for the correct record of translation, a TLB miss event is generated triggering a *page walk*. A page walk is a hardware mechanism that looks-up into the main memory's ATTs mappings for the corresponding translation. The TLB is on the critical path so every miss on the TLB (page walk) degrades the overall workload performance. In contemporary hardware more than one level of TLBs can be found, minimizing the amounts of requests to the main memory's ATTs.

Although ZSim provides lightweight user-level virtualization [125], it does not take into account V2PAT, meaning that the simulator ignores TLB overheads along with its corresponding page walks. In HSCC [94], researchers introduced a TLB mechanism

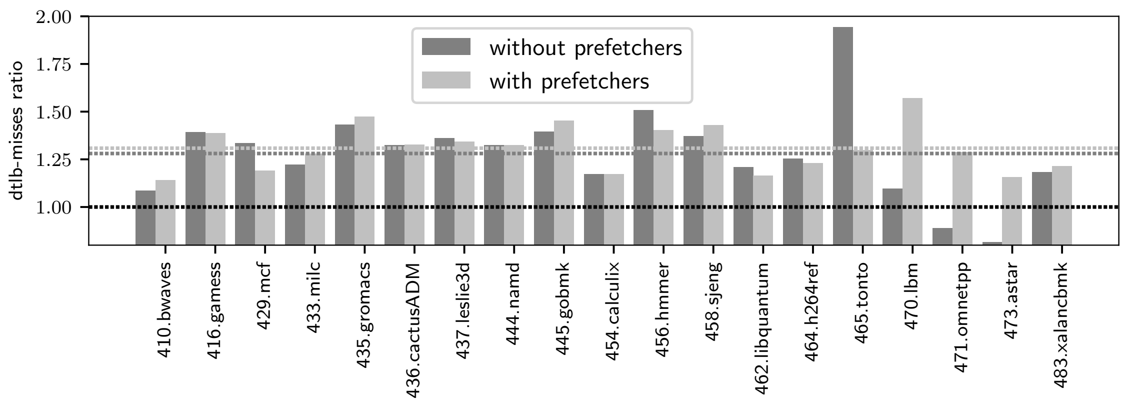


Figure 7.3 `dtlb-access` ratios for the enhanced ZSim-HSCC vs. the Sandy Bridge machine for each of the SPEC CPU2006 benchmarks.

to improve multi-memory management. The authors had implemented a 1-level TLB scheme on top of ZSim. To quantify the impact of address translation, we used and extended their work.

As we mentioned in Section 6, a modification to the source of the simulator implies a projection of new architecture in the real world. Therefore, ZSim-HSCC is an ideal target to test our validation proposal.

First, we integrated our Sandy Bridge upgrades (Section 4) with ZSim-HSCC. Next, using hardware performance counters, we collected and compared the data-TLB (`dtlb-access`) in both architectures. In the simulator, accesses are calculated as `hits + misses`, whereas in the real machine, a HWPC provides this information. Figure 7.3 shows the ratio calculated as

$$dtlb\_ratio = \frac{Simulator\_dtlbaccess}{Sandy\ Bridge\_dtlbaccess}.$$

Each one of the two bars represents the comparison of the simulator and the real machine in two scenarios: on the left bar with the prefetchers disabled and the right bar with the prefetchers enabled. On average[1], ZSim-HSCC overestimates the `dtlb-misses` by 32 %. Moreover, except for five benchmarks (`465.tonto`, `470.lbm`, `471.omnetpp`,`473.astar`, `483.xalancbmk`), we can identify that the address translation is not affected by the prefetcher configuration.

In Figure 7.4, we used the Top-Down retirement factor to identify the effects of address translation. For each benchmark, three bars are presented. The baseline in our work is the real execution with prefetchers enabled (1st bar), the next two bars

[1]From our previous comparison we excluded `401.bzip`, `450.soplex`, `447.dealIII`,`459.GemsFDTD`, and `482.sphinx3` because of timing violations that caused a simulation crash.

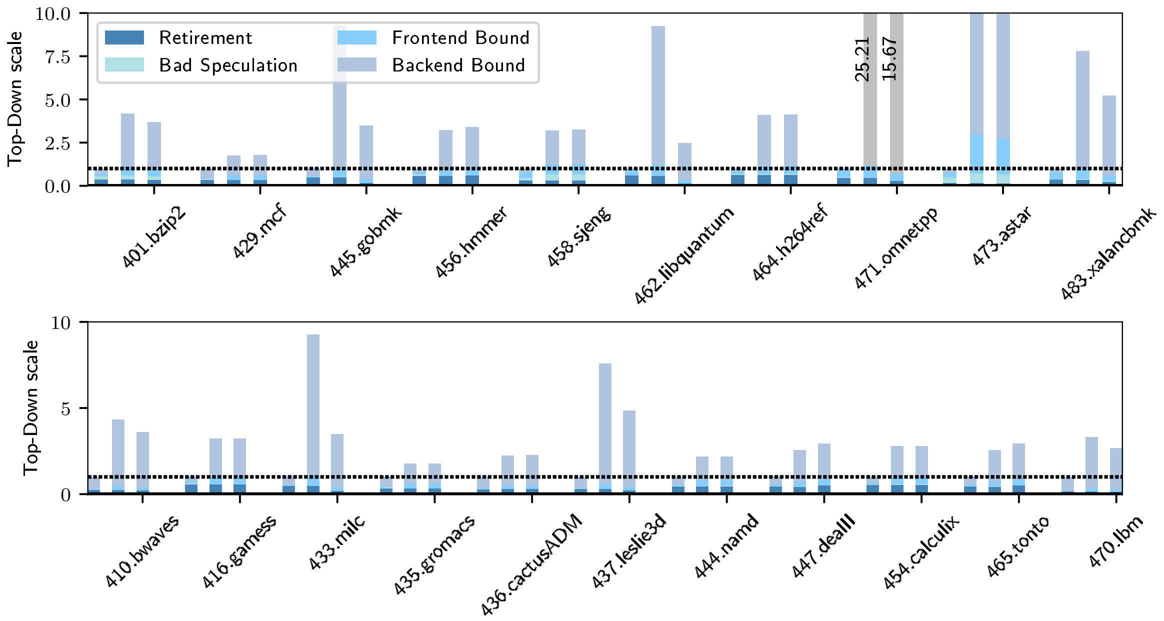


Figure 7.4 Top-Down Analysis for ZSim-HSCC SPEC CPU2006 benchmarks. From left to right: the 1st bar corresponds to the real execution with prefetchers enabled (baseline in our case study), the 2nd bar to the simulation vs. real execution with prefetchers enabled, and the 3rd bar, corresponds to the simulation vs. real machine execution with prefetchers disabled.

correspond to the simulation executions. To use the retirement factor, we compared the simulation with the real machine execution with prefetchers enabled (2nd bar), and with prefetchers disabled (3rd bar).

When prefetchers are enabled, the retirement factor grows bigger than in the opposite case. The average increments in the Backend bound category with respect to the real machine with prefetchers enabled are: for the integer set of benchmarks 6.38 and 2.58 for the floating-point set. When prefetchers are disabled the average decreases to 4.06 for the integer set, and 1.81 for the floating-point set. This result heavily supports the argument that hardware prefetchers improve workload performance.

The memory intensity workloads that appear to be most affected by address translation in both comparisons are: for the integer set, `445.gobmk`, `462.libquantum`, `471.omnetpp`, `473.astar`, and `483.xalacbmk` with Top-Down scale differences ranging from 2.26 up to 9.1; for the floating-point set, `433.milc` and `437.milc` present Top-Down scale differences of 2.47 and 5.14 respectively. The average difference for the benchmarks that shows small effects because of address translation is 0.2 for the integer set, and 0.1 for the floating-point set. This behavior implies a little temporal data and code locality.

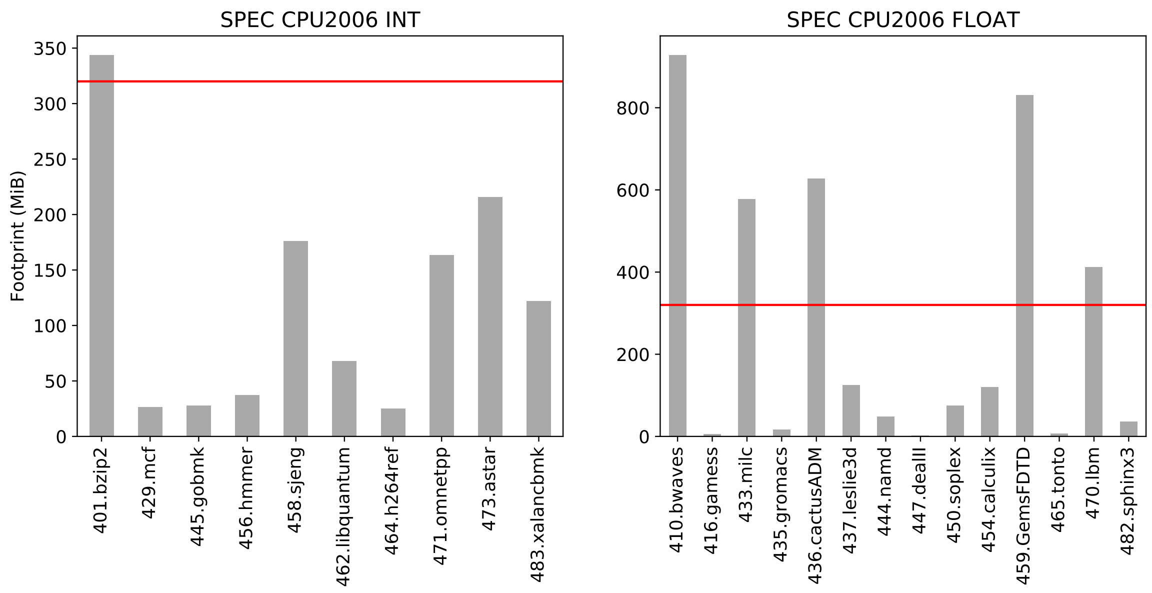


Figure 7.5 SPEC CPU2006 memory footprint for the first 50 billion instructions. The red horizontal lines mark the proposed 320 MiB threshold.

7.4 Memory Utilization

A characteristic not available in baseline ZSim but brought-in by ZSim-HSCC is the memory footprint accountability. In modern OSs, the memory usage, or *memory footprint*, is calculated by the amount of page tables a process requests. For example, given that the default page size for most of the Linux systems is 4 KiB, a process that allocates and uses 10 KiB will report 12 KiB as memory footprint since 3 pages were allocated for the process' request. Taking the same example for depicting purposes, the 2 KiB difference between the reported memory footprint and the *used* memory leaves room for the Resident Set Size (RSS) definition: which is the amount of physical memory used by the process. Furthermore, in most computer programs, several memory allocations and deallocations happen throughout the lifetime of their execution. The peak of RSS memory used by a process is also known as the *high watermark*. This metric is in particularly interest for HPC applications, since it determines the minimal amount of memory expected by the HPC application developer to be physically available in the HPC running node.

Figure 7.5 depicts the memory footprint reported by the ZSim-HSCC simulator during the first 50 billion of instructions execution on different SPEC CPU2006 benchmarks. Let's remark that as opposed to the regular 4 KiB Linux metrics, the memory footprint in ZSim-HSCC is measured with a 64 B granularity. Meaning that

	Integer	Floating-point
mean	120.5 MiB	272.0 MiB
std-dev	105.5 MiB	334.1 MiB
minimum	25.05 MiB	1.8 MiB
maximum	343.8 MiB	928.4 MiB

Table 7.2 Memory footprint characteristics of the first 50 billion instructions execution of the SPEC CPU2006 benchmark suite.

for ZSim-HSCC both terms, *high watermark* and *memory footprint* can be used interchangeably.

In Table 7.2 we displayed the differences between the minimum and maximum memory footprint recorded for the 50 billion of instructions execution. These values are tighter for the integer set than for the floating point set of benchmarks. It is distinguishable that for the SPEC CPU2006 suite, the memory footprint for the integer set of benchmarks does not fluctuate as much as the floating-point counterpart. Since the minimum and maximum values for the integer set of benchmarks are *close* (25.05 MiB, 343.8 MiB) to a known reference value in our simulated platform (the 20 MiB LLC) we decided to establish an arbitrary threshold of 16× the LLC which in this analysis results in 320 MiB.

Using Figure 7.5 and our results from the relative IPC error analysis, we can identify those benchmarks that surpassed the 320 MiB threshold and that were catalogued with *high* relative IPC error. For the floating-point set of benchmarks, there is only one exception, `436.cactusADM` which was cataloged with *very good* relative IPC error but surpasses the threshold. For the integer set of benchmarks, `462.libquantum` have a *high* relative IPC error but it is located under the 320 MiB threshold whereas `401.bzip2` surpasses the threshold but was catalogued with *very good* relative IPC error.

Given that 22 out of 24 benchmarks in the SPEC CPU2006 suite follows the hypothesis that workloads not crossing the 16× the LLC threshold are correlated with good simulation accuracy, we propose our chosen threshold for further simulation validation. For any targeted execution of a workload, if the application's memory footprint is lower than the 16× the LLC threshold, we will have good simulation accuracy for CPU and DRAM co-simulation.

7.4.1 Real-World Memory Footprint and HPC Data Center Deployments

Even with computers' robustness at 2020, categorizing an application with 16× the LLC might sound as a small memory footprint for HPC workloads; but recent studies on the state-of-the-art show otherwise.

The common metric used to characterize memory in regards of CPU perspective is the *memory per core*; nevertheless, there are two contexts where the same concept applies but have different readings. In the context of job allocation for HPC workloads, the *memory per core* ratio is defined as the quotient of all the allocated memory for the job divided by the number of the assigned cores (logical or physical). In the context of data center deployment, the ratio is calculated in a single-node basis being deined as the quotient of the total amount of physical main memory divided by the number of physical cores[2].

Although there is a mild increasing trend in memory footprint throughout time, low memory footprint workloads are not uncommon in HPC data centers. Researchers have found that 48 % of workloads sent to the ARCHER supercomputer (UK) utilizes less than 0.5 GiB per core [135]. This result is compatible with researchers at University of Buffalo analyzing NSF's (USA's National Science Foundation) Innovative HPC resources, where in average HPC workloads do not exceed more than 1.0 GiB per core utilizing only 19 % of available memory [129]. In a separate study, researchers from Virginia Tech (USA) in the Los Alamos National Laboratory HPC, has concluded that HPC workloads used in average 29 % of available memory [111]. Lastly, researchers from BSC (Spain) have found similar results; HPC applications with good scalability present low memory footprint per process with less than 0.5 GiB per core [151]. The study also shows that most HPC applications with a distributed design decrease the memory footprint per working process node, whereas the master process (only 1 node) increases its memory footprint.

A remark found in such studies is that workloads managers in HPC data centers such as `SLURM` are configured by default to allocate an HPC job exclusively on a given node [131]; meaning that no other allocation could use the machine's resources even if those are underutilized. IBM's `lsf` also provides a queue for job exclusiveness in its scheduling policy.

[2]In HPC data centers, Simultaneous multithreading (SMT) technologies such as Intel's Hyper-Threading are disabled by default because of performance loss in HPC applications [31].

Generation	Micro-architecture	Nodes	Sockets	Processors	Cores	Main Memory	Main Memory per Core
1st	IBM PowerPC 970FX	4,812	1	1	4,812	9.6 TiB	2 GiB
2nd	IBM PowerPC 970MP	10,240	1	1	10,240	20 TiB	2 GiB
3rd	Intel SandyBridge-EP E5-2670	2,752	2	16	88,064	176.128 TiB	2 GiB
3rd	Intel SandyBridge-EP E5-2670	128	2	16	4,096	16 TiB	4 GiB
3rd	Intel SandyBridge-EP E5-2670	128	2	16	4,096	32 TiB	8 GiB
3rd	Intel Xeon Phi 7230	16	1	64	1,024	1.5 TiB + 256 GiB	1.5 GiB + 256 MiB
4th	Intel Xeon Platinum 8160	3,240	2	24	155,520	303.75 TiB	2 GiB
4th	Intel Xeon Platinum 8160	216	2	24	10,368	81 TiB	8 GiB
4th	IBM Power9 8335-GTH	52	2	20	2080	26 TiB	12.8 GiB | 3.2 GiB

Table 7.3 MareNostrum core and main memory evolution.

Modern releases of hardware from IHVs and Original Equipment Manufacturers (OEMs) follow the trend in HPC workload's memory footprint. For instance, providing service to 1,059 researchers from 51 research groups [12], the MareNostrum supercomputer is constantly identified as one of the fastest and most robust HPC systems. Currently in its fourth iteration, it has continuously incremented computer power and memory capacity since the previous generations. Table 7.3 presents a brief survey of the technical characteristics and evolution of the MareNostrum supercomputer. Among these, the main memory capacity and the ratio between the main memory and processor count are in our particular interest. Only a subset of nodes of the supercomputer in the last two generations offers a different memory configuration; either with bigger capacity, or access to an emerging memory technology.

In the MareNostrum supercomputer, the memory per core has been a constant of 2 GiB per core. To situate MareNostrum with respect to other HPC systems, in Figure 7.6 we present the evolution of the first 100 systems [3] in the Top500 list [40]. We can notice an existing trend to provision systems with 2 GiB per core and up to 8 GiB per core. Furthermore, the dominant micro-architecture of systems using more than 2 GiB is `x86`.

Lastly, emerging memory technologies such as 3D-stacked DRAM, are becoming a common architectural decision to deploy among HPC OEMs [130, 123, 4]. Albeit emerging memory technologies are being targeted for GPU specific purposes such as for Machine Learning (ML) or faster visualization workloads, attempts such as Intel's Xeon Phi products provisioned with MCDRAM memory gave us a glimpse on how to profit using emerging memory technologies. Notwithstanding, memory capacity on such deployments are still modest with respect to main memory meaning that the machine's memory per core using emerging memory technologies it is also still under our simulation ranges for good accuracy.

[3]There were some systems with incomplete information or that we could not determine if all memory belongs to main memory, or if it is shared with GPU devices. From the 100 list we used: 53, 51, and 51 systems for the years: 2019, 2018, 2017 correspondingly.

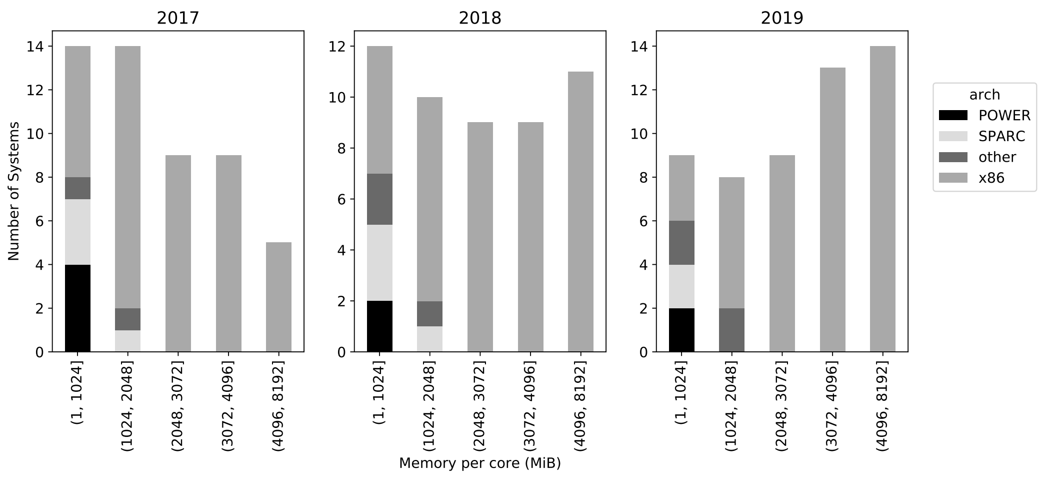


Figure 7.6 Evolution of the RSS/core. First 100 ranked systems in the Top500 list.

7.5 Summary

To identify the source of errors in the memory hierarchy simulation, we proposed an architectural modification that consists of injecting a delay for every memory transaction that crosses the memory controller. We named this proposal the Delay Queue. By injecting a delay for every memory transaction that missed the LLC, we achieved 2 goals: 1. we reached the maximum expected latency in main memory simulation for dependent back-to-back memory transactions and, 2. we created a *bubble* in the CPU executing pipeline that is easy to spot using our retirement factor Top-Down modification.

The Delay Queue allow us to design experiments to identify if the main memory simulation error is in the CPU or DRAM simulation. When comparing the results using the our Top-Down's retirement factor, we could identify that the gap of main memory simulation comes from the CPU simulation. As some of the simulated workloads show a decrease in the *Backend bound* Top-Down category while increasing the latency for memory transactions, we concluded that the simulation error must come from the CPU engine. This line was studied in a separate work: Rethinking Cycle Accurate DRAM Simulation [91].

Moreover, we noticed a link between memory footprint, high memory utilization, and large simulation error. We propose to use the memory footprint threshold of 16× the LLC to estimate if the simulated workload will execute with good simulation accuracy.

CHAPTER 8

Opportunities For Emerging Memory Systems: Check-pointing in Heterogeneous Systems

As we shown in the previous chapter, a portion of the HPC data center time is consumed by low memory footprint workloads (less than 2 GiB per core). Given that some IHVs are deploying small capacity emerging memory technologies into their products, such as the Intel's Xeon Phi Processor that is deployed with 16 GiB of 3D-stacked DRAM, we envision a clear opportunity to use these deployments with low memory footprint HPC applications. The intent for emerging memory technologies is to boost application's performance, or in the case of non-volatile memories, improve reliability support. In this work, we expose a case study where real-world low-memory footprint HPC applications use an emerging memory technology for data reliability, while improving their current execution performance.

Checkpoint-Restart (CR) is a common technique that saves hours of application's execution in the case of a failure event. Currently, CR it is considered a best-practice for HPC programmers. Moreover, some data centers are questioning to impose a CR requirement for HPC applications. Therefore, some HPC applications have incorporated some CR functionality. We extended the CR Fault Tolerant Interface (FTI) library [13] to work with heterogeneous memory subsystems. For those users who already use the library, small changes are needed to explicitly use different memory nodes. For those who do not, the library empowers the application developer with a simple API, that performs transparent data migrations between different memory technologies, consolidating a robust system tolerant to failures.

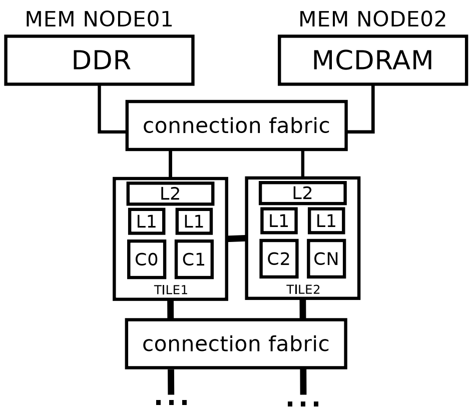


Figure 8.1 An example of a NUMA system: a tile-mesh architecture with two associated memory technologies.

8.1 Background

A computer architecture design that deals with dynamic memory latency, is known as a NUMA architecture. The tile-mesh design shown in Figure 8.1 is an example of a NUMA system. Each tile is composed of compute cores sharing a physical connection with the fabric, where two memory technologies are found: MCDRAM and Double Data Rate (DDR). From the computing cores standpoint, once a memory transaction is issued the obtained latency is variable. The complexity of the fabric adds dynamic delays to the latency of memory transactions: transactions for package routing, data coherency, or data location.

For an application to make usage of the memory, it has to interact with the OS. For the scope of this work, we will use the term OS and Linux interchangeability[1]. The interaction from the application to the Linux kernel, might be summarized as follows. First, the application requests for a memory range via OS system calls. Next, the OS scrubs into its internal structures for an available memory range. Using the concept of virtual memory, the OS is responsible for managing physical-to-logical address mappings, so that all running applications might use portions of the memory as the applications demanded it. If it is possible to fulfill the request, the OS assigns a virtual address which is then delivered to the application request. When the application receives the OS assigned address, two options are available for the application: either begin to use the requested space, or apply for a change of policy of the given space, i.e., being located into a different memory node. For our case study, we use both mechanisms conveniently.

[1]We base our assumption on the fact that as per 2019, all HPC systems listed on the Top500 site are running a software distribution with a Linux kernel [40].

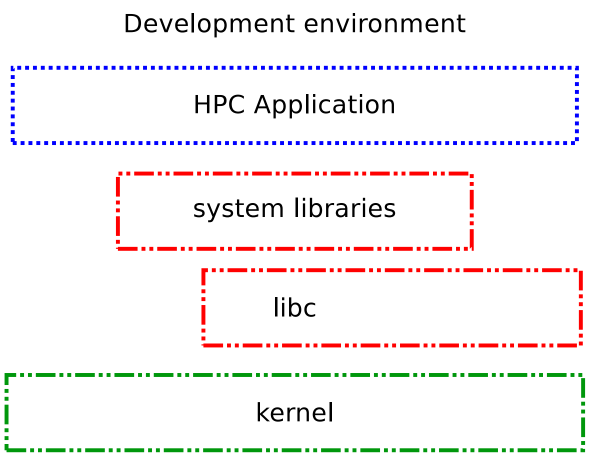


Figure 8.2 Software stack representation. Dependencies exist if the boxes vertically overlap.

Figure 8.2 abstracts the HPC application development environment. Each box represents a software entity of our particular interest. If the boxes vertically intersect, it means that the upper box relies on the support of the layer underneath. For example, the HPC application overlaps completely with the kernel base layer; all applications need support from the kernel. Moreover, the application might use system libraries which iteratively use the `libc`, or they can directly access the syscall kernel interface. The box diagram aids in the understanding of the dependency support between HPC application and the system libraries.

8.2 System Libraries

Most computing applications rely on external pieces of code that abstract some functionality to achieve some goal: reading and writing specific file formats, perform specific data manipulation, or orchestrate distributed environments. These external pieces of code or *libraries* help researchers to focus on the task to solve rather than dealing with computing infrastructure problems. In this section, we will discuss the most important libraries that affects our case study.

8.2.1 libc

The `C` library is the most relevant library among the Linux distributions [51]. The library provides all functionalities imposed by the ISO C standard, the Portable Operating System Interface (POSIX), the Berkeley Unix and System V Interface Description, and the X/Open Portability Guide standard. Nowadays, most of the Linux applications and Linux system libraries are written in languages that somehow are linked against libc. In fact, some of the traditional languages Fortran (GNU) or Perl are directly implemented in `C`. Some other languages implement a bootstrap mechanism: first they are compiled with C/`asm` (therefore using `libc`) to get a small compiler based on the target language; then, the resulting compiler is used to build the rest of the language. Examples like these are: C++, Java (OpenJDK), Python (CPython) or Go. There are some other `libc` implementations, but mostly related to the embedded world.

8.2.2 Computer Topology and System Libraries

In NUMA architectures as the one depicted in Figure 8.1, the OS might assign different memory nodes (also known as NUMA nodes) to the memory allocations requested by the running applications. Moreover, depending on the process-load of the system, the Linux scheduler might migrate a process from one core to another within the same processor or even to another processor's core out of the current silicon package. Such policy directly impacts application's performance either by the variations in the memory access latency, the cost of process migration, or pollution to the memory caches.

To minimize the effects of process migration and cache pollution, a common solution is to *pin* the process to a specific core. For Symmetric Multi Processing (SMP) systems, the support was added in kernel 2.5.8 [121], and wrapped in `libc` in version 2.3.3. For different memory nodes, kernel engineers provide an API to assign certain memory requests to specific memory nodes as the developer feels convenient. The NUMA support was introduced since kernel version `2.5.40` [100], but an API to handle different memories policies was only introduced in kernel `2.6.26` [133]. The user-level library that abstracts the NUMA support is known as **`libnuma`** [86]. The library is part of the `numactl` package, which enables applications to communicate with the kernel to request for specific processor location.

A requirement to use the `libnuma` library is the knowledge of the running computer topology. If the topology is unknown, the Portable Hardware Locality (**`hwloc`**) library [24] aids the application developer to identify processors in the system [55], enumerate and characterize the memory hierarchy [54], as well to perform device tree discovery [53].

Furthermore, the libraries are commonly deployed with command-line tools that help the application developer to further abstract the policies so that no changes to the code are required. These tools wrap the execution of the application by creating an environment where the application is pinned to a particular core binding all the memory transactions to a particular NUMA node. Although that solution might be the right fit for simple applications, HPC applications require stricter granularity for different code regions.

Traditionally, HPC applications scale out into large data centers. To communicate among nodes of the data center cluster, they use a standardized parallel paradigm: the **MPI** [104]. The standard mandates a set of operations that translates into a specific library interface implemented in C and Fortran. The implementation of such standard is carried out by multiple entities: from free and open source ones (MPICH, Open MPI), up to the industry support (Intel MPI or Microsoft MPI). MPI distributions are deployed with tools that ease the compilation process and the correct execution in the supported environments.

8.2.3 Memory Allocation

A Linux application is composed of three memory regions for data: the global area, the stack, and the heap [63]. These three regions are found in the `.data` section of an object file. On the one hand, the global area and the stack, are constrained in size by the micro-architecture and the compiler. On the other hand, once the object file is loaded into application's memory, the heap is the region where the application might request as much memory as the OS' virtual space policies allow.

For the application to use some memory from the heap, it has to issue a special request to the Linux kernel. If the request is valid, the OS responds to the application with the starting address of a memory region along with the right permissions and policies to bind that request. Generally, the request to allocate memory for application usage is performed through the `libc`'s `malloc(3)` or `mmap(2)` function calls. Both family of functions wrap the Linux kernel API for virtual memory allocation: `brk(2)` and `mmap(2)`. These interactions generate on overhead in the total application's performance. Although a single request does not deteriorate the application's performance, a large number of requests might do so. The penalties on performance are associated with the overhead in kernel scrubbing, page allocation, and potential cases of cache pollution.

A solution to minimize performance degradation generated because of the large amount of memory allocation requests is the usage of a heap manager. A heap manager is a user-land interface that aims to optimize the OS virtual memory management; the most used heap manager is provided by the `libc`'s library through the `malloc(3)` family of functions. Despite the convenience to simply use `libc`'s heap manager, researchers have found that, for some cases [95], building a granular manager for memory requests increases application performance. Some the most popular heap managers are:

Hoard [17] Google's TCMalloc [56], Intel's TBBMalloc [73], and Jemalloc [75]. Our particularly interest is in the latter, Jemalloc, since is in constant evolution [76] and is regularly cited as a reference in most of the state-of-the-art works [41, 95, 128, 45, 137, 113, 95].

Jemalloc achieves performance efficiency by reducing the number of kernel interactions [46] by pre-allocating and managing the memory resources with different granularity throughout the lifetime of the application. Granularity is accomplished using the *arena* concept. In Linux, an arena is a contiguous unused memory from the heap region. In `jemalloc`, an *arena* is a structure that categorizes the memory into buckets based on the requested size. Three buckets are provided: small, large, and huge. For small and large portions, `jemalloc` pre-initializes regions of the memory in chunks of 4 MiB that iteratively are split into smaller regions (*chunks*) of 1 MiB. Huge portions are allocated independently through adjacent memory *chunks*. The easiest way to use Jemalloc by the application programmer is to wrap (shim) application's execution using the Linux's `LD_PRELOAD` environment variable. That way, all calls to `libc`'s `malloc(3)` are transparently routed to Jemalloc.

Memkind is a library that enables access to emerging memory technologies. Since the library is mainly maintained by Intel Corp [28], it supports emerging technologies such as non-volatile Intel's Optane (3D-Xpoint), or HBM (MCDRAM) on Intel's KNL products. It is built on top of Jemalloc and Intel's Threading Building Blocks as heap managers, as well as the `hwloc` and `libnuma` libraries. In general, the library might be used for any emerging technology which is exposed as independent NUMA node. One of the most promising features of the library is the *AutoHBW* feature. In a NUMA system, the library transparently allocates memory blocks into the HBM NUMA node without code modifications. Just as with Jemalloc, Memkind uses the Linux's `LD_PRELOAD` environment variable to wrap the application's execution.

Lastly, a library from the Argo project of the LLNL [127], **`AML`** *Building Blocks for Explicit Memory Management*, provides a mechanism that allows the application developer to place, align, and move data through a tiling scheme, exploiting data locality to boost application's performance. `AML` provides interfaces that request and manage the virtual memory from the OS. At the moment of the writing of this work, `AML` uses `jemalloc` [75, 76] to improve virtual memory usage. As opposed to the usage of Memkind or Jemalloc, `AML` needs substantial changes in the source code to exploit data locality; therefore, applications must be compiled and linked with `AML`'s support.

8.2.4 Checkpoint-Restart Libraries

Computing resources are susceptible to failures: manufacturing nuances, thermal conditions in data center, or estimated time of product's life, are among the common sources of errors. In addition, corner cases in software logic or non-validated input conditions might cause software to crash. A portion of these errors are recoverable: some computing devices are designed with ECC for main memory and storage devices, preventing application errors because of data corruption in the memory hierarchy. Unfortunately, not all errors can recover once they happened, i.e., the shutdown of a computing node because of a power surge.

Regularly, distributed executions on several computing nodes stop running when a single instance crashes. In such cases, all previous computation in all nodes is lost if data was not properly saved. Checkpoint-Restart (CR) is a technique to provide applications with resilience support. Given a certain amount of time or when special milestone is met in application's lifetime, a *photograph* of the current application's execution is saved. The *photograph* is the current *checkpoint* of the application. The checkpoint might be composed of all the data of the application including internal application control variables or just a selected subset of the data. In regards of the application's recovery, depending on the set-up of the CR policy, the CR subsystem will read the data from the last *checkpoint* and restart the execution from there.

In the state-of-the-art, CR systems might be categorized as Application-Level, User-Level and System-Level [44]. System-Level checkpointing is out of the scope of our study since it requires administrative privileges on the executing nodes. Application-Level and User-Level checkpointing might overlap in design; the main difference lies in an automated or explicit manner to perform the checkpoint. Two system libraries with a growing user base that provide CR support are: SCR as Application-Level checkpoint, and the FTI library on the User-Level checkpoint.

The **Scalable Checkpoint/Restart** (SCR) for MPI [108] is a multi-level checkpointing library focused to optimize `I/O` operations. The library is extensively used in the LLNL since 2007. SCR is built on top of CRUISE [117]: an in-memory user-level file system for checkpoints that support a variety of storage technologies, including emerging persistent memory. `SCR` integrated with `CRUISE` extends the possibilities of check-pointing where storage resources are insufficient in local nodes. Moreover, SCR enables a cache for fast and slow storage systems. The multi-level checkpoint is provided as a hierarchy of checkpoints where the lower levels are the weaker regarding resiliency, and the higher levels offer a robust checkpoint mechanism. This is because SCR wraps the running environment performing an incremental checkpoint from all application related data. From the application developer standpoint, few changes are needed since only some logic is required to make the application to be aware of the checkpoint.

The **Fault Tolerant Interface (FTI)** library provides a framework to enable low-overhead CR capabilities to an HPC application [13]. The library allows for a multi-level checkpoint scheme mainly on the back storage. At the moment of this implementation, FTI supports a special memory technology aside from the main memory: the GPUs memory. This implementation is different from the scope of this study as the memory from the GPU is not addressable from the CPU, and it relies on the manufacturer's support, *e.g.,* CUDA proprietary drivers from Nvidia [98].

FTI enables a multi-level checkpoint scheme for CR. The levels refer to the secondary storage where the checkpoint is stored: on local storage (*L1*), local storage + partner-copy (*L2*), *L1* with erasure code (*L3*), and on global storage (*L4*). Through a configuration file, the developer controls the library specifying parameters for the general execution of the HPC application, *e.g.,* paths to the local storage (*L1, L2, L3*), paths to the parallel file system (*L4*), number of reserved processes per node, group information (for *L2* and *L3*), or time to automatically perform the check-pointing.

The usage model for FTI is that the HPC application developer performs a call to the FTI library to initialize, protect, and clean the FTI run-time environment. To protect and check-point the data, an explicit call to `FTI_Protect()` and `FTI_Checkpoint()` respectively, is required. Furthermore, the check-point might be configured to happen periodically, indistinctly of the corresponding function call.

We chose to work with FTI due to the user-level design that allows us to explicitly specify the desired NUMA node while controlling the memory footprint with an in-variable granularity.

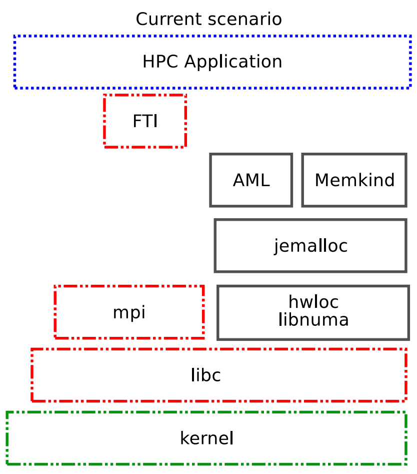


Figure 8.3 Software stack - FTI's current scenario.

8.3 Methodology

In Figure 8.3, the software stack related to FTI's integration with system libraries is depicted. In this scenario, all pieces of the software stack are using `libc` and the HPC application must be compiled with FTI and MPI support.

Several approaches might be taken to support memory heterogeneity in the HPC application. 1. The HPC application directly supports the memory heterogeneity: the application must discover the hardware topology for each one of the MPI processes where the distributed execution is taking place. This means all MPI processes must synchronize to identify concurrent node execution and then split the hardware resources accordingly. 2. The HPC application uses system libraries that manage memory heterogeneity: the application uses data structures proposed by the system libraries modifying the source code accordingly. This approach needs to ensure concurrent execution conditions so that hardware resources do not starve, for example, running out of memory because there are multiple process assiginments on the same NUMA node. 3. The HPC application instructs the system libraries to manage memory heterogeneity: this is a hybrid approach. Although the delegation for managing the different NUMA nodes is passed down to libraries, some parameters also need to be discovered or set up. For example, if Memkind is selected to manage different NUMA nodes, then an execution environment must be discovered and configured so that each MPI process has a maximum amount of memory to be used per NUMA node allocation. 4. The HPC application uses system libraries that transparently manage the memory heterogeneity: as the checkpoint libraries already support the MPI standard and deal with the memory allocations, the libraries can synchronize with all MPI processes, so that hardware

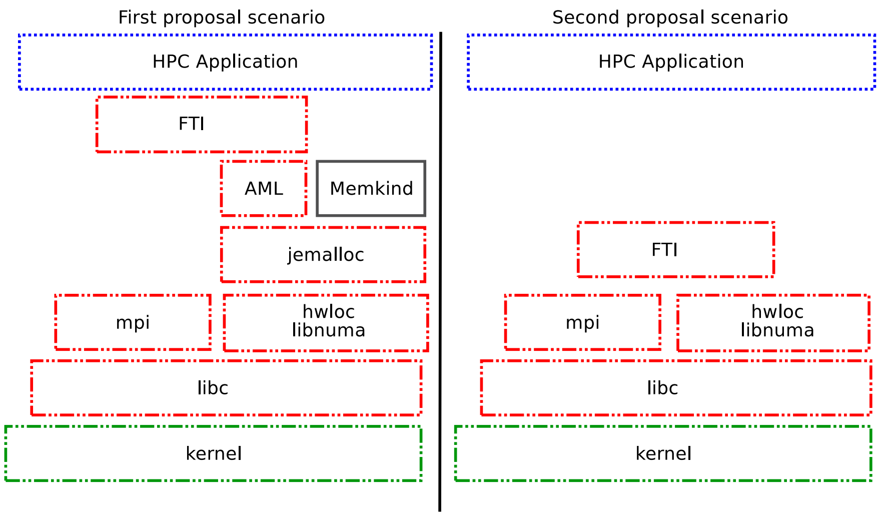


Figure 8.4 Software stack of the FTI's proposed scenarios.

discovery and context set-up might happen without or very little intervention of the HPC application.

We decided to pursue the hybrid approach proposing two scenarios: 1. Extending the FTI library using the `AML` library: `AML` combines memory heterogeneity through `jemalloc` with a data-tiling scheme, making it attractive for the HPC application development. 2. Native support of memory heterogeneity in FTI. Both proposals are depicted in Figure 8.4. For the application developer, few changes are needed in both scenarios, mainly driven by two new FTI function calls; `FTI_ProtectedVariable()`, which reserves the amount of requested memory in the selected NUMA node, and `FTI_Migrate()`, used to move data from one NUMA node to other. Because the latter is the most expensive operation, we elaborated on the implementation. We expect the application developer to use the migration only once.

8.4 Implementation

Within the FTI architecture, the library holds for an internal pointer list of variables to protect. To insert variables in the list, the application developer uses the `FTI_Protect()` call. When the checkpoint takes place, FTI traverses the list of protected variables according to the CR level requested. Therefore, the impact of integrating memory heterogeneity is expected to change the `FTI_Checkpoint()` performance. Furthermore, we evaluated two important characteristics within this proposal: sensitivity to data locality and code implementation. Both aspects are crucial to break ambiguity when more than one NUMA nodes are used i.e., in a heterogeneous memory system.

8.4.1 FTI Integration with `AML`

Our primary objective is to provide HPC applications with a robust failure management system that supports heterogeneous memory. Since `AML` shares the same objective, two significant changes were needed for FTI and `AML` to be integrated. First, we have extended the same concept of `jemalloc` arenas into FTI. Second, we needed to implement `AML`'s memory tiling model into FTI's.

At the moment of the integration, `AML` used `jemalloc` as a heap manager, meaning that `jemalloc`'s workflow heavily influences some design principles in `AML`. For example, an *arena* for every NUMA node has to be initialized and bounded accordingly. To translate FTI's memory model to `AML`'s, we assumed a contiguous virtual memory region for the protected variable to migrate. This assumption allows us to use the `AML`'s 1D-tiling scheme using memory *chunks* of size: 4 KiB (commonly, a single page).

```
AML_DMA_LINUX_SEQ_DECL(dma);
AML_TILING_1D_DECL(tiling);
struct aml_area_linux_mmap_data mmapconfig;
szVariable = sizeof(variableToCkpt);
ptr = aml_area_linux_mmap_generic(&mmapconfig, NULL, checkpointSize);
nRequests = szVariable/pageSize;

aml_tiling_init(&tiling, AML_TILING_TYPE_1D, pageSize, szVariable);
aml_dma_linux_seq_init(&dma, &nRequests);
for( request = 0; request < nRequests; request++ )
    aml_dma_copy(&dma, &tiling, ptr, variableToCkpt,request);
aml_dma_linux_seq_destroy(&dma);
aml_tiling_destroy(&tiling, AML_TILING_TYPE_1D);
```

Listing 8.1 FTI-`AML` enabled pseudo-code for `FTI_Migrate()` procedure.

Listing 8.1 presents a pseudo-code of the process involved to use the `AML` library for migrating a variable between memory nodes. For each of the variables on the FTI protected list, the following procedure is executed. From line 1 to line 6, initialization of variables is required by the `AML` library, where `nRequests` holds for the number of pages the copy has to perform from main memory to the file system for the current variable. Lines 8 and 9 initialize the `AML` tiling scheme. The migration for each variable is done through the loop on lines 10 and 11, which iterates `nRequests` times. Lastly, in lines 13 and 14 for each `AML` structure it is required to clean the state of the used variables.

8.4.2 FTI's Native Implementation

In our proposed implementation, during FTI initialization, the library verifies if the system is provisioned with non-traditional memories. If the technologies are available, and the application developer instructs FTI to use them, the appropriate data structures are configured accordingly.

Two opportunities are available to profit from emerging technologies: 1. If the application developer is still using libc's `malloc(3)`, FTI will manage the allocation, so the application transparently uses the emerging technology. 2. To avoid expensive data migrations, we enforce FTI to align all the protected data, so that the remainder of operations would be as effective as the hardware could perform.

We provisioned FTI with a set of functions similar to libc's `malloc(3)` family of functions: the most critical call translates into a virtual memory request (`mmap(2)`) followed by a NUMA allocation petition (`mbind(2)`). The requests keep the data aligned to the page size (in most of Linux systems this is 4 KiB) to alleviate the check-point operations when writing the data to the file system. Both of our implementations of `FTI_ProtectVariable()` and `FTI_Migrate()` use our `FTI`'s alloc() family of functions making their usage transparent to the application developer.

Listing 8.2 presents a pseudo-code version of the `FTI_Migrate()` function. In lines 1-3, the context pointers and variable holding the control sizes are initialized. In line 5, the function identifies if the memory to migrate is aligned or not. If the memory is not aligned, then the lines 6-9 process the request with two operations. The first copy operation begins with line 6 which contains the maximum aligned pages on the migrated data. The migration process is finalized on line 9, copying the trail of bytes that were not aligned. As the original allocation used a call to `FTI_ZeroAlloc()`, the remainder of space in the last page is padded with zeros. In line 12 if the function identifies that the memory is aligned, the process is performed only using a single function call. Finally, on line 13, a validation check is performed.

```
1  alignedSize = getAlignedSize(size, pageSize);
2  newPtr = FTI_ZeroAlloc(FTI_Align(alignedSize), placement);
3  startPtr = basePointer;
4
5  if ( (remainder = (size % pageSize)) != 0 ) {
6    FTI_Memcpy(newPtr, sourcePtr, alignedSize)
7    newPtr = newPtr + alignedSize;
8    sourcePtr = sourcePtr + alignedSize;
9    FTI_Memcpy(newPtr, sourcePtr, remainder)
10 } else {
11   FTI_Memcpy(newPtr,  startPtr,  pageSize);
12 }
13 FTI_Memcmp(newPtr,  basePointer,  alignedSize);
```

Listing 8.2 `FTI_Mirate()` - Native Implementation. No external overheads.

8.5 Evaluation

8.5.1 Experimental Setup

We evaluated the checkpoint for heterogeneous memory hierarchy in a 4-node Intel Knights Landing (KNL) cluster. The platform remains valid for the scope of this study since 3D-stacked DRAM is becoming a *de facto* component in HPC designs. Each node of the cluster has an Intel Xeon Phi 7230 processor comprising 64 cores operating at 1.30 GHz. The memory available in the system consists of two NUMA nodes: an external DDR4 node populated with 6x16 GiB DIMMs running at 1200 MHz and an in-socket MCDRAM module. The NUMA nodes are distributed through the fabric as eight MCDRAM memory controllers with 2 GiB each, and two DRAM memory controllers managing three channels each.

Three configurations are possible for the Intel KNL to expose the DDR4 and MCDRAM memory to the OS [79]. 1. *Cache mode*: the MCDRAM is used as cache for the DDR4 memory. In this mode, the OS wouldn't notice the MCDRAM's presence. 2. *Flat mode*: both memory technologies are exposed as two different NUMA nodes increasing the amount of total memory available in the system. 3. *Hybrid mode*: a partition is made so that the two previous modes are used in small but dedicated portions (half or quarter of the MCDRAM).

We chose to use the *flat mode* as we are interested in exploring the implications of the usage of MCDRAM in the checkpoint process while exploiting the maximum available memory in the system. The OS used in this evaluation is a SUSE Linux distribution running a Linux kernel version 4.4.21. The cluster resources are managed through the SLURM workload manager.

8.5.2 Sensitivity to Code Implementation and Data Location

In order to test sensitivity to data locality, we use a Double precision General Matrix Multiplication (DGEMM) application where input and output data are bound to a particular NUMA node using our `FTI_ProtectedVariable()` interface. The DGEMM implementation is provided by the Intel Math Kernel Library (MKL) through the `cblas_dgemm()` library call. In this implementation, there are two input matrices (`A`,`B`), and one output matrix (`C`). The size of the matrices ranges from 512 MiB to 1800 MiB. We designed an experiment to find the allocation of input and output matrices that allow the best computing performance. The experiment is about allocating the input and output matrices in different NUMA nodes. We acknowledge that in a scientific

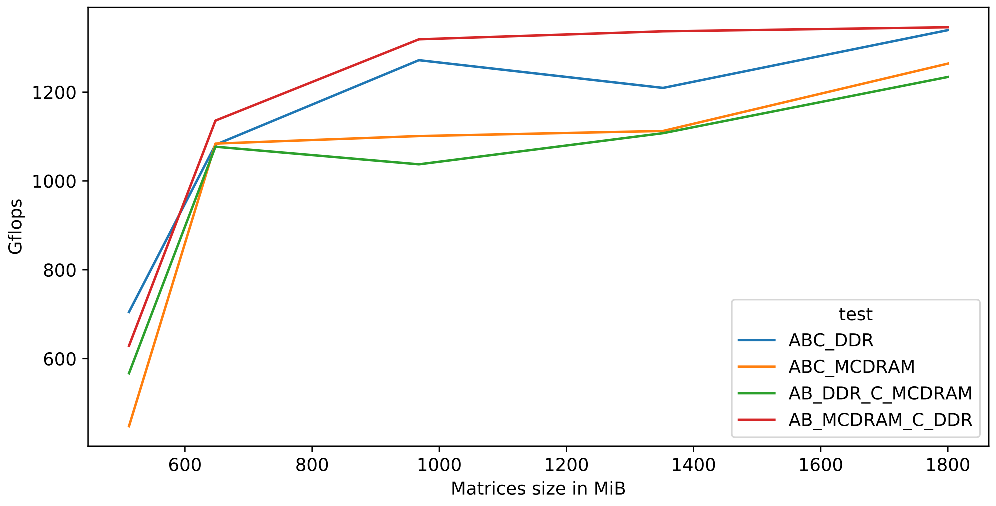


Figure 8.5 Maximum Gflops achieved by allocating different sizes of matrices in different NUMA nodes. `A,B` are the input matrices and `C` is the output matrix.

application such matrices might need to migrate from one NUMA node to another depending on the application workload.

Figure 8.5 presents the results for the best allocation experiment. For different sizes of the matrices, diverse performance metrics are reached. For the smallest size in the experiment (512 MiB), the best allocation corresponds to all three variables placed on the DDR. As the size of the matrices increases, the result is overcome by two allocations: 1. having the input matrices in the MCDRAM and the output matrix in the DDR 2. having all three matrices in the DDR. Overall, the best performance is achieved by allocating the input values in the MCDRAM and the output values in the DDR. The results make sense as the MCDRAM provides an increment in memory bandwidth, so that the Intel MKL kernels, written with `AVX2` extensions, take direct benefit from such feature.

For testing sensitivity to code implementation, two variants of the DGEMM were used: the Intel's MKL `cblas_gemm()` family of functions, and an auto-generated DGEMM algorithm from the BOAST programming framework [140].

Figure 8.6 presents the results for the two code implementations. The first two sets of bars on the left side of the figure correspond to the BOAST implementation, the two sets of the right correspond to the Intel's MKL implementation. For each group, we show four sets of bars that corresponds to different input sizes: 32 MiB, 128 MiB, 512 MiB, and 2048 MiB. The Intel's MKL module is a specialized crafted version of the DGEMM algorithm compiled explicitly for the KNL architecture. In contrast, the

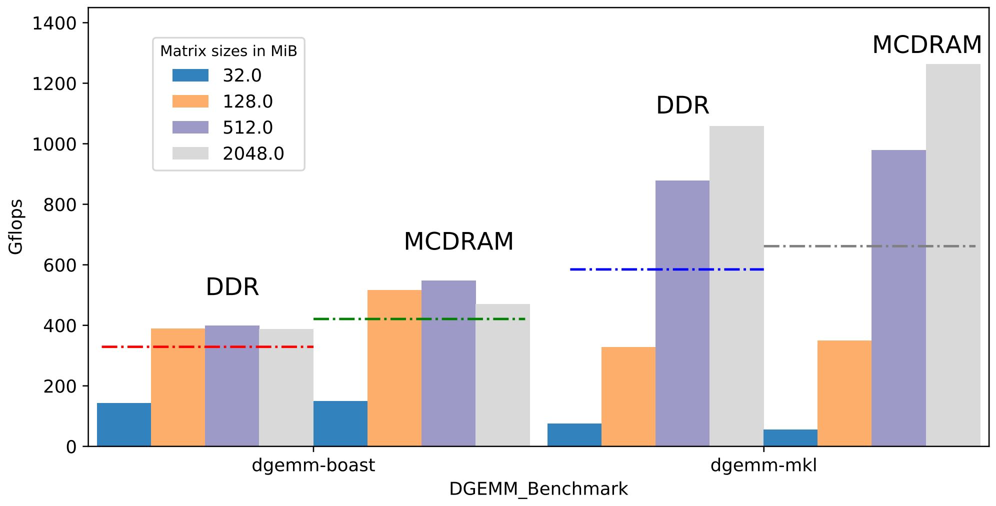


Figure 8.6 Giga FLOPS sustainied for different DGEMM implementations on a 64-core execution. The dotted lines depict the average value for each experiment after 10 time iteration. The size of matrices are expressed in MiB.

auto-generated BOAST implementation leaves the architecture-specific details to the compiler. In both cases, the average improvement of using MCDRAM as input and the DDR node as the output matrix is 23 %.

8.5.3 Implementation Impacts on `FTI_Checkpoint()`

For this evaluation, we used Intel's DGEMM implementation with matrix sizes of 8 MiB, 18 MiB, and 32 MiB. The experiment was executed using the 64-cores on each of the nodes of the KNL cluster. Table 8.1 shows the total memory footprint per-node in the experiment, where the memory footprint of the checkpoint for the three variables is defined as:

$$Checkpoint\_Footprint = (matrix\_elements) * \texttt{sizeof(double)} * (number\_matrixes) * (number\_cores).$$

Number of elements	Total checkpoint size per-node
1024	1536 MiB
1536	3456 MiB
2048	6144 MiB

Table 8.1 Total checkpoint size per-node in a 3 matrix experiment running in 64 cores.

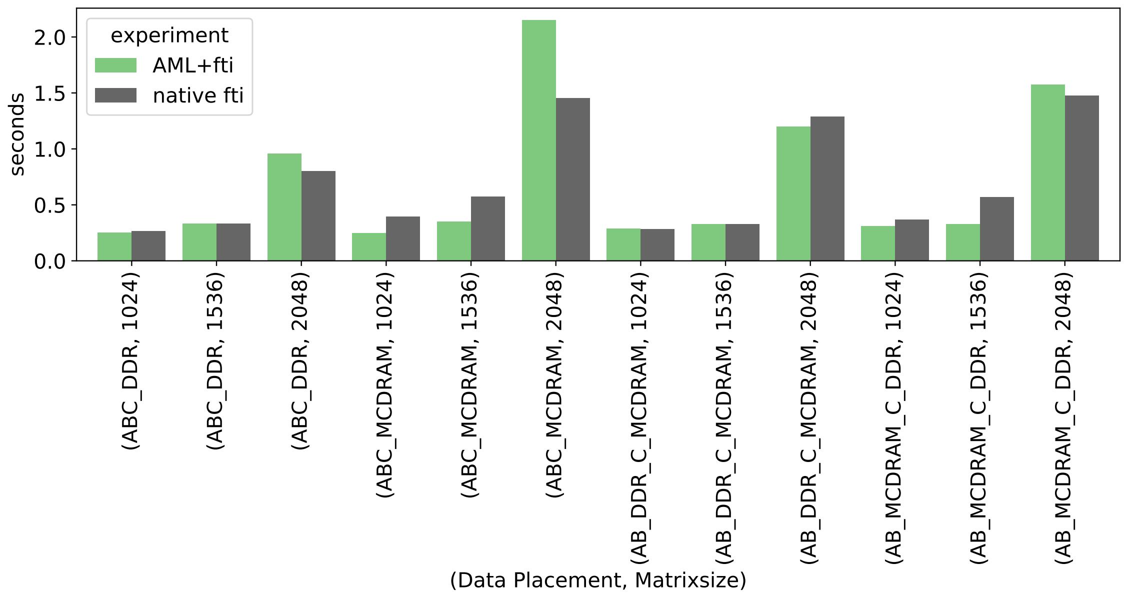


Figure 8.7 Comparison of performance of the `FTI_Checkpoint()` function with different NUMA allocations for the DGEMM test.

The performance impact of the implementation of `FTI` with AML and the native `FTI` is depicted in Figure 8.7. Results are measured in seconds.

Although MCDRAM latency is slightly higher than DDR [115], the Multi-Channel technology allows the checkpoint to benefit from larger bandwidth. The difference is noticeable on larger memory sizes. For small and medium sizes, when the input matrices are located in the DDR, both implementations behave similar. Not the same when when input matrices are located in MCDRAM: the native implementation takes longer. Moreover, when the size of the input matrices is large, the native implementation is considerably faster. The exception happens when input matrices are located in DDR.

8.5.4 Impacts on Real-World HPC Applications

We used three HPC applications to stress the `FTI_Checkpoint()` function call: CoMD [33], CoSP2 [35] and LULESH [83]. LULESH and CoSP2 are part of the ExMatEx [47] project, whereas CoMD is part of the CoPA [34] project. All three applications were executed in the 4-node KNL cluster. CoMD and LULESH ran with 216 MPI processes distributed as 54 MPI processes per node, while CoSP2 ran with 192 MPI processes distributed as 48 MPI processes per node. For each HPC application, we chose up to 4 variables to checkpoint with FTI. We consider two scenarios for the checkpoint: when the variables are placed in the MCDRAM, and when they are placed

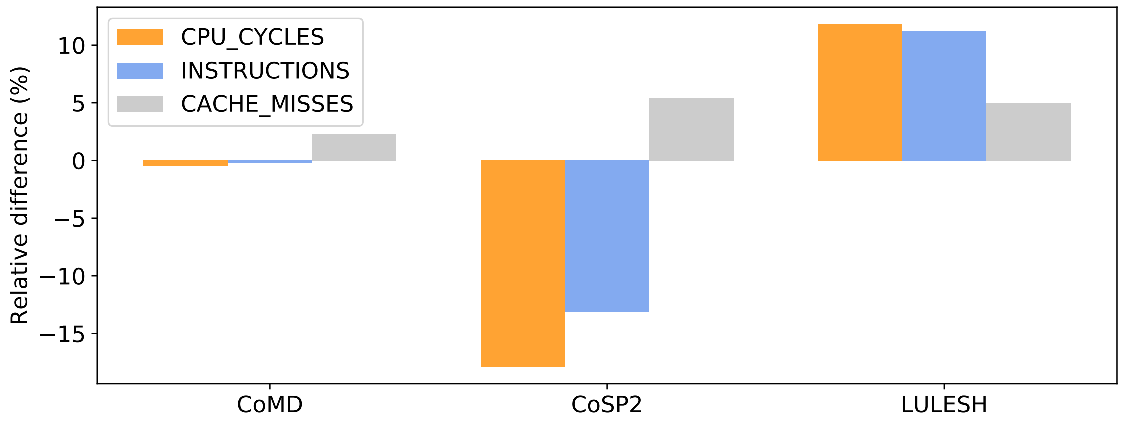


Figure 8.8 Relative difference of counter measurements for `FTI_Checkpoint()` with the baseline variables located in the DDR NUMA node.

in the DDR. The total size of the protected variables (the checkpoint size) is displayed in Table 8.2.

To compare the performance we used HWPC, accessed via Linux perf [67]. We collected measurements for three counters: CPU unhalted cycles (`CPU_CYCLES`), retired instructions (`INSTRUCTIONS`), and `L2` cache misses (`CACHE_MISSES`). We considered the average of all measurements in the cluster. In Figure 8.8, we present the relative difference between measurements taking the checkpoint on DDR as the baseline for the calculation.

On the one hand, CoMD barely presents a significant difference but CoSP2 presents a negative difference of −17.9 % in CPU cycles and −13.4 % for retired instructions, meaning that checkpointing on the MCDRAM is a lighter and faster process. On the other hand, LULESH presents the opposite case. The reason for this is that in the main iteration cycle, LULESH destroys and allocates a portion of memory. On FTI's native implementation we are not using a heap manager, generating an additional overhead on the overall application but not more than 11.8 %. In all three applications,

Application	Size per MPI process	Total checkpoint size per-node
CoMD	53.47 MiB	2887.38 MiB
LULESH	124.81 MiB	6739.74 MiB
CoSP2	48.09 MiB	2308.32 MiB

Table 8.2 Individual checkpoint size and Per-Node total checkpoint size

the overhead of cache misses is around 4.2 %. This is expected as the `L2` cache in all cores mainly contains information from variables stored in the DDR.

8.6 Emerging Hardware and Workload Trends

Although we have worked with a specific memory technology, it is important to remark that the two promising memory technologies are taking orthogonal directions. On the one hand, non-volatile memory (NVM) is increasing the amount of memory available to the application but at latency cost. On the other hand, 3D-stacked DRAM is minimizing the CPU idle time through its high bandwidth, but its capacity is limited.

Turner and McIntosh-Smith [135] explored the HPC application's footprint for over a year (2017) in the ARCHER supercomputer. They found that considering the characteristics of their system, 60 % of all allocated jobs used less than 0.5 GiB per core. In this work, we presented a case study conducted on an Intel KNL cluster, where only 16 GiB of MCDRAM is available throughout the 64 cores in every node, that is 256 MiB per core. Doubling the size available in near memory architectural designs will suffice to replace DRAM for most of the current HPC workloads.

New workloads for ML and data-analytics such as map-reduce are considered to use large amounts of memory. However, the traditional HPC core applications would not increase their memory footprint dramatically in the next years. Specifically, 3D-stacked memory seems to be a good match for HPC core applications, whereas Non-Volatile Memory (NVM) can be a good match for the ML and Big Data workloads. Furthermore, researchers had noticed [15] that there is an incremental adoption of MPI communications in ML frameworks.

Interacting with hybrid memory systems is not always easy; features like the automatic migration technique presented in this work can help users leverage such deep memory hierarchies. The convergence of HPC and industrial data-mining applications is pushing computer designs to adopt hybrid memory architectures where both types of applications can thrive. Therefore, it is crucial to strengthen the existing run-time libraries such as FTI with new hybrid memory technologies as the one presented in this thesis.

8.7 Summary

In this chapter we proposed a solution to support heterogeneous memory systems through a software implementation in a CR library. Moreover, we conducted an evaluation of such implementation targeting small memory footprint HPC applications.

The heterogeneous memory system used for this study is composed of an emerging memory technology (MCDRAM) working together with traditional memory (DRAM). Our software implementation to support heterogeneous memory systems exhibits low overhead and does not translate into any difficulties to the application programmer, as it is exported only through two new function calls within the traditional API of the CR library. We further showed that in-memory data location, and code-specific implementations are important parameters that directly affect the application performance. For instance, using the most efficient in-memory data placement, the performance could improve by 23 % while at the same time for reliability purposes the data are protected and the checkpoint's time is minimized.

New memory technologies are becoming ubiquitous into exascale data centers where resiliency support for HPC applications will not take long to turn into a mandatory requirement. Fortunately, our proposal complies with these requirements: we encourage the community to try out our implementation.

CHAPTER 9

Conclusions

In this thesis we analyzed, through system simulation, the implications of the processor's micro-architectural details concerning main memory. We also explored a link between the HPC application's memory footprint and simulation accuracy. Finally, we proposed and evaluated a software implementation to support memory heterogeneity in HPC applications focusing on low-memory footprint workloads while enabling them with checkpoint-restart capabilities.

9.1 Impacts of System Simulation

Focusing on the memory subsystem, we have shown the importance of validation for computer architecture simulation. When using a system simulator, a myriad of parameters are available to researchers to design diverse experiments. A slight modification at any level of the computing architecture model directly affects the outcome of the simulation, including the performance of the simulated workload.

In this thesis, we validated a simulation infrastructure based of the co-simulation of two state-of-the-art system simulators: ZSim and DRAMsim2. The simulated infrastructure attempts to mimic a real system composed of an Intel Sandy Bridge E5-2670 processor populated with DDR3 memory devices. Our results showed that in comparison to the real machine measurements, the default configuration of the CPU simulator deviates 33 %, and for the main memory simulator approximately 20 ns of the main memory latency was not taken into account.

The memory subsystem behavior is driven by the processor's micro-architectural details such as instruction latency or execution unit utilization. To characterize these parameters, we designed a family of micro-benchmarks that allows us to extract them from the real machine so that we could enhance the simulator infrastructure accordingly.

With the extracted parameters, we upgraded the simulator infrastructure to achieve 29 % of accuracy gain (considering 5 % of relative CPI error) in CPU simulation. To tackle the memory simulation gap, we explored a combination of CPU parameters for memory simulation, along with a simulation model proposal (the `Delay Queue`) to inject configurable delays for every transaction that crosses the memory controller.

Moreover, to compare our measurements, we proposed an extension to the Top-Down CPI stack analysis: the *retirement factor*. Our extension provides us with a metric to measure the differences when comparing two micro-architectures such as our simulation infrastructure and the targeted real machine. We further analyzed the impact of hardware prefetchers and virtual-to-physical address translation in the real system when compared to the simulation infrastructure. Our Top-Down extension helped us to promptly recognize how significant the gap is between the systems under test. We found that six applications of the SPEC CPU2006 (four in the integer domain and two in the floating-point domain) are profoundly affected by the address translation mechanism when prefetchers are disabled, exposing a high-data-locality behavior for these applications. The average difference between the Top-Down's Backend bound category when assessing the impact of hardware prefetcher in the SPEC CPU2006 benchmark suite is: 0.75 for the floating point subset and 1.21 for the integer subset.

Using the `Delay Queue` and comparing the performance executions using the *retirement factor* we found that the responsible for the differences between the simulator infrastructure and the real machine is the approximate simulation technique used in the CPU simulator.

Finally, we found a link between the simulation accuracy and the HPC application's memory footprint. Particularly, our simulation infrastructure shows good accuracy for low memory footprint HPC workloads. The memory footprint of HPC applications is a crucial parameter to consider in supercomputing scalability and data center provisioning; specially when emerging memory technologies are about to be integrated in new deployments.

9.2 Opportunities for Emerging Memory Technologies

We envisioned an opportunity to use emerging memory technologies, particularly for workloads that exhibit a low memory footprint profile. In this thesis, we evaluated the impact of enabling HPC applications to use a heterogeneous memory system provisioned with an emerging memory technology. We implemented support for memory heterogeneity in a CR library for HPC applications. The implementation enables HPC workloads to benefit from the emerging memory technology while endowing reliability support.

We showed that performance improvement in HPC workloads is sensitive to data location and code implementation, particularly if an emerging memory technology is used. We further showed that using a *heap manager* can boost the workload's performance by handling virtual memory for small data-transfers, primarily when memory resources are claimed continuously by the HPC application and returned to the OS; nevertheless, native (direct) OS support is more suitable for large data-transfers. Moreover, we showed that curated data locality favors memory performance: we identified an improvement in one of our experiments with up to 15 % of computing cycles.

9.3 Future Work

In the near future, system simulation will remain to be the chosen approach to conduct computer architecture prototyping and performance exploration. To attain simulation results in a feasible time, models for approximate system simulation need to be explored. We plan to evaluate an approach that accurately couples CPU interval simulation and approximate but detailed main memory simulation. For example, in a collaborative research [91] a novel model according to these principles has already been proposed.

Moreover, we plan to abstract current NUMA functionality into the simulator infrastructure, so that policies for data locality would also be considered when simulating a real workload. Furthermore, models for emerging memory technologies should be available in the simulator infrastructure. For example in a separate work [10], we provided a set of parameters for an emerging non-volatile memory acting as main memory.

In regards of real system comparison with a simulation infrastructure, we plan to generalize our proposed micro-benchmark for different micro-architectures, extended to support SMT and SMP beyond the *spinner* strategy we used in this work.

We also plan to extend our discussion on memory footprint in HPC data centers with real data collected from an HPC cluster. The data collection must consider including information such as: 1. periodic sampling of HWPCs to conduct Top-Down analysis 2. periodic sampling of RSS, the high watermark, and the Working Set Size (WSS) 3. if possible, source code pointers of the algorithms used for the workload execution.

Lastly, we plan to incorporate error injection profiles into the simulation infrastructure and the corresponding support on the CR software library.

APPENDIX A

Instruction Latencies

Table A.1 shows the comparison between the real machine (Intel Xeon E5-2670 Sandy Bridge), ZSim (enhanced) with the parameters extracted from the micro-benchmark execution in the real machine (Sandy Bridge), and ZSim (original) without the micro-architecture upgrade. The table presents the CPI for each one of the targeted instructions. Since the micro-benchmarks are executed only for a specific instruction, this is considered the latency of the instruction. Instructions with fractional CPIs, means that the instruction is composed of more than one micro-operation and effectively distributed into the Execution Unit ports at the same cycle.

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
ADC	2.00	2.00	2.00
ADD	1.00	1.00	1.00
ADDPD	3.00	3.00	3.00
ADDPS	2.99	3.00	3.00
ADDSD	3.00	3.00	3.00
ADDSS	3.00	3.00	3.00
ADDSUBPD	3.00	3.00	3.00
ADDSUBPS	3.00	3.00	3.00
AESDEC	7.97	8.00	1.00
AESDECLAST	7.97	8.00	1.00
AESENC	7.97	8.00	1.00
AESENCLAST	7.97	8.00	1.00
AESIMC	2.04	2.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
AESKEYGENASSIST	7.97	8.00	1.36
AND	1.00	1.00	1.00
ANDNPD	1.01	1.00	1.00
ANDNPS	1.00	1.00	1.00
ANDPD	1.01	1.00	1.00
ANDPS	1.00	1.00	1.00
BLENDPD	1.08	1.37	1.37
BLENDPS	1.08	1.37	1.37
BLENDVPD	2.01	2.00	1.00
BLENDVPS	2.00	2.00	1.00
BSF	2.99	3.00	0.33
BSR	2.99	3.00	0.33
BSWAP	1.00	1.00	1.00
BT	0.97	1.00	0.67
BTC	1.00	1.00	1.00
BTR	1.00	1.00	1.00
BTS	1.00	1.00	1.00
CBW	1.00	1.00	1.00
CDQ	1.00	1.00	0.50
CDQE	1.00	1.00	1.00
CLD	3.99	4.00	1.00
CMC	1.00	1.00	1.00
CMOVB	1.00	1.00	1.00
CMOVBE	1.01	1.00	1.00
CMOVL	1.00	1.00	1.00
CMOVLE	1.00	1.00	1.00
CMOVNB	1.00	1.00	1.00
CMOVNBE	1.01	1.00	1.00
CMOVNL	1.00	1.00	1.00
CMOVNLE	1.00	1.00	1.00
CMOVNO	1.00	1.00	1.00
CMOVNP	1.00	1.00	1.00
CMOVNS	1.00	1.00	1.00
CMOVNZ	1.00	1.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
CMOVO	1.00	1.00	1.00
CMOVP	1.00	1.00	1.00
CMOVS	1.00	1.00	1.00
CMOVZ	1.00	1.00	1.00
CMP	0.34	0.67	0.67
CMPPD	3.02	3.00	3.00
CMPPS	3.01	3.00	3.00
CMPXCHG	4.98	5.00	5.00
COMISD	1.00	1.00	1.00
COMISS	1.00	1.00	1.00
CQO	0.98	1.00	0.50
CRC32	3.01	3.00	3.00
CVTDQ2PD	1.02	1.03	1.49
CVTDQ2PS	1.02	1.00	1.00
CVTPD2DQ	1.01	1.03	1.49
CVTPD2PI	1.01	1.03	1.49
CVTPD2PS	1.01	1.03	1.49
CVTPI2PD	1.02	1.03	1.49
CVTPI2PS	3.98	2.00	1.00
CVTPS2DQ	1.03	1.00	1.00
CVTPS2PD	1.00	1.00	1.00
CVTPS2PI	1.01	1.00	1.00
CVTSD2SI	1.01	1.03	1.00
CVTSD2SS	1.01	1.03	1.49
CVTSI2SD	3.00	1.03	1.49
CVTSI2SS	3.00	1.03	1.00
CVTSS2SD	1.00	1.03	1.00
CVTSS2SI	1.01	1.03	1.00
CVTTPD2DQ	1.01	1.03	1.49
CVTTPD2PI	1.01	1.03	1.49
CVTTPS2DQ	1.03	1.00	1.00
CVTTPS2PI	1.01	1.00	1.00
CVTTSD2SI	1.01	1.03	1.00
CVTTSS2SI	1.01	1.03	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
CWD	1.00	1.00	0.50
CWDE	1.00	1.00	1.00
DEC	1.00	1.00	1.00
DIVPD	10.00	10.00	7.00
DIVPS	9.99	10.00	7.00
DIVSD	10.00	10.00	7.00
DIVSS	10.01	10.00	7.00
DPPD	9.00	9.00	1.36
DPPS	11.97	12.00	1.36
ENTER	9.05	10.82	1.04
EXTRACTPS	1.08	1.36	1.36
FABS	1.00	1.00	1.00
FADD	2.99	3.00	1.00
FCHS	1.00	1.00	1.00
FCMOVB	2.00	1.99	1.04
FCMOVBE	2.00	1.99	1.04
FCMOVE	2.00	1.99	1.04
FCMOVNB	2.00	1.99	1.04
FCMOVNBE	2.00	1.99	1.04
FCMOVNE	2.00	1.99	1.04
FCMOVNU	2.00	1.99	1.04
FCMOVU	2.00	1.99	1.04
FCOM	1.00	1.00	0.67
FCOMI	1.00	1.00	0.67
FDECSTP	1.00	1.00	0.33
FDIV	9.99	10.00	1.00
FDIVR	9.99	10.00	1.00
FFREE	1.00	1.00	0.33
FFREEP	2.00	2.00	0.33
FINCSTP	1.00	1.00	0.33
FMUL	4.98	5.00	1.00
FNOP	1.00	1.00	0.33
FNSTSW	1.00	1.00	0.33
FST	1.00	1.00	0.33

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
FSTP	1.00	1.00	0.33
FSUB	2.99	3.00	1.00
FSUBR	2.99	3.00	1.00
FTST	1.00	1.00	1.00
FUCOM	1.00	1.00	0.67
FUCOMI	1.00	1.00	0.67
FWAIT	1.00	1.00	0.33
FXAM	2.02	2.00	1.00
FXCH	0.50	1.00	1.00
HADDPD	4.98	5.00	1.00
HADDPS	4.98	5.00	1.00
HSUBPD	4.98	5.00	1.00
HSUBPS	4.98	5.00	1.00
IMUL	2.99	3.00	3.00
INC	1.00	1.00	1.00
INSERTPS	1.08	1.36	1.36
LAHF	1.00	1.00	0.33
LZCNT	3.00	3.00	0.74
MAXPD	3.00	3.00	3.00
MAXPS	2.99	3.00	3.00
MAXSD	3.00	3.00	3.00
MAXSS	3.00	3.00	3.00
MFENCE	33.07	0.33	0.33
MINPD	3.00	3.00	3.00
MINPS	2.99	3.00	3.00
MINSD	3.00	3.00	3.00
MINSS	3.00	3.00	3.00
MOV	0.34	0.33	0.33
MOVAPD	1.01	1.00	1.00
MOVAPS	1.00	1.00	1.00
MOVD	1.00	1.00	0.33
MOVDDUP	1.01	1.00	1.00
MOVDQ2Q	1.00	0.74	0.74
MOVDQA	0.33	0.74	0.74

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
MOVDQU	0.33	0.74	0.74
MOVHLPS	1.00	1.00	1.00
MOVLHPS	1.00	1.00	1.00
MOVMSKPD	1.01	1.00	1.00
MOVMSKPS	1.00	1.00	1.00
MOVQ	0.33	0.74	0.74
MOVQ2DQ	0.33	0.74	0.74
MOVSHDUP	1.01	1.00	1.00
MOVSLDUP	1.01	1.00	1.00
MOVSS	1.01	1.00	1.00
MOVSX	1.00	1.00	1.00
MOVSXD	1.00	1.00	1.00
MOVUPD	1.01	1.00	1.00
MOVUPS	1.00	1.00	1.00
MOVZX	1.00	1.00	1.00
MPSADBW	5.99	6.00	1.36
MULPD	4.99	5.00	5.00
MULPS	4.98	5.00	4.00
MULSD	4.99	5.00	5.00
MULSS	4.99	5.00	4.00
NEG	1.00	1.00	1.00
NOT	1.00	1.00	1.00
OR	1.00	1.00	1.00
ORPD	1.01	1.00	1.00
ORPS	1.00	1.00	1.00
PABSB	0.50	1.00	1.00
PABSD	0.50	1.00	1.00
PABSW	0.50	1.00	1.00
PACKSSDW	1.01	1.00	1.00
PACKSSWB	1.01	1.00	1.00
PACKUSDW	1.02	1.00	1.00
PACKUSWB	1.01	1.00	1.00
PADDB	1.01	1.00	1.00
PADDD	1.01	1.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
PADDQ	1.00	1.00	1.00
PADDSB	1.01	1.00	1.00
PADDSW	1.01	1.00	1.00
PADDUSB	1.01	1.00	1.00
PADDUSW	1.01	1.00	1.00
PADDW	1.01	1.00	1.00
PALIGNR	1.08	1.36	1.36
PAND	1.01	1.00	1.00
PANDN	1.01	1.00	1.00
PAUSE	10.95	11.00	9.00
PAVGB	1.01	1.00	1.00
PAVGW	1.01	1.00	1.00
PBLENDVB	1.00	1.00	1.00
PBLENDW	1.08	1.36	1.36
PCMPEQB	1.01	1.00	1.00
PCMPEQD	1.01	1.00	1.00
PCMPEQQ	1.02	1.00	1.00
PCMPEQW	1.01	1.00	1.00
PCMPESTRM	11.01	1.36	1.36
PCMPGTB	1.01	1.00	1.00
PCMPGTD	1.01	1.00	1.00
PCMPGTQ	5.06	5.00	3.00
PCMPGTW	1.01	1.00	1.00
PEXTRB	1.00	1.36	1.36
PEXTRD	1.00	1.36	1.36
PEXTRQ	1.00	1.43	1.43
PEXTRW	1.00	1.00	1.00
PHADDD	2.02	2.00	1.00
PHADDSW	2.02	2.00	1.00
PHADDW	2.02	2.00	1.00
PHMINPOSUW	5.06	5.00	1.00
PHSUBD	2.02	2.00	1.00
PHSUBSW	2.02	2.00	1.00
PHSUBW	2.02	2.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
PINSRB	1.08	1.36	1.36
PINSRD	1.08	1.36	1.36
PINSRQ	1.07	1.43	1.43
PINSRW	1.08	1.00	1.00
PMADDUBSW	4.99	5.00	1.00
PMADDWD	4.99	5.00	1.00
PMAXSB	1.00	1.00	1.00
PMAXSD	1.00	1.00	1.00
PMAXSW	1.01	1.00	1.00
PMAXUB	1.01	1.00	1.00
PMAXUD	1.00	1.00	1.00
PMAXUW	1.00	1.00	1.00
PMINSB	1.00	1.00	1.00
PMINSD	1.00	1.00	1.00
PMINSW	1.01	1.00	1.00
PMINUB	1.01	1.00	1.00
PMINUD	1.00	1.00	1.00
PMINUW	1.00	1.00	1.00
PMOVMSKB	1.01	1.00	1.00
PMOVSXBD	1.00	1.00	1.00
PMOVSXBQ	1.00	1.00	1.00
PMOVSXBW	1.00	1.00	1.00
PMOVSXDQ	1.00	1.00	1.00
PMOVSXWD	1.00	1.00	1.00
PMOVSXWQ	1.00	1.00	1.00
PMOVZXBD	1.00	1.00	1.00
PMOVZXBQ	1.00	1.00	1.00
PMOVZXBW	1.00	1.00	1.00
PMOVZXDQ	1.00	1.00	1.00
PMOVZXWD	1.00	1.00	1.00
PMOVZXWQ	1.00	1.00	1.00
PMULDQ	4.99	5.00	1.00
PMULHRSW	5.00	5.00	1.00
PMULHUW	4.99	5.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
PMULHW	4.99	5.00	1.00
PMULLD	4.99	5.00	1.00
PMULLW	4.99	5.00	1.00
PMULUDQ	4.99	5.00	1.00
POPCNT	3.00	3.00	1.00
POR	1.01	1.00	1.00
PSADBW	4.99	5.00	1.00
PSHUFB	1.02	1.00	1.00
PSHUFD	0.50	1.00	1.00
PSHUFHW	0.50	1.00	1.00
PSHUFLW	0.50	1.00	1.00
PSHUFW	0.50	0.74	0.74
PSIGNB	1.02	1.00	1.00
PSIGND	1.02	1.00	1.00
PSLLD	1.00	1.00	1.00
PSLLDQ	1.02	1.00	1.00
PSLLQ	1.01	1.00	1.00
PSLLW	1.00	1.00	1.00
PSRAD	1.00	1.00	1.00
PSRAW	1.00	1.00	1.00
PSRLD	1.00	1.00	1.00
PSRLDQ	1.02	1.00	1.00
PSRLQ	1.00	1.00	1.00
PSRLW	1.00	1.00	1.00
PSUBB	1.01	1.00	1.00
PSUBD	1.01	1.00	1.00
PSUBQ	1.00	1.00	1.00
PSUBSB	1.01	1.00	1.00
PSUBSW	1.01	1.00	1.00
PSUBUSB	1.01	1.00	1.00
PSUBUSW	1.01	1.00	1.00
PSUBW	1.01	1.00	1.00
PTEST	1.00	1.01	1.01
PUNPCKHBW	1.01	1.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
PUNPCKHDQ	1.01	1.00	1.00
PUNPCKHQDQ	1.01	1.00	1.00
PUNPCKHWD	1.01	1.00	1.00
PUNPCKLBW	1.01	1.00	1.00
PUNPCKLDQ	1.01	1.00	1.00
PUNPCKLQDQ	1.01	1.00	1.00
PUNPCKLWD	1.01	1.00	1.00
PXOR	1.01	1.00	1.00
RCL	2.05	2.00	2.00
RCPPS	1.02	1.00	1.00
RCPSS	5.04	5.00	1.00
RCR	2.06	2.00	2.00
ROL	1.00	1.00	1.00
ROR	1.00	1.00	1.00
ROUNDPD	1.07	1.37	1.37
ROUNDPS	1.08	1.37	1.37
ROUNDSD	3.01	3.00	1.37
ROUNDSS	3.01	3.00	1.37
RSQRTPS	1.02	1.00	2.00
RSQRTSS	4.99	5.00	2.00
SAHF	1.99	1.00	0.33
SAR	1.00	1.00	1.00
SBB	2.00	2.00	2.00
SETB	1.00	1.00	0.33
SETBE	1.00	1.00	0.33
SETL	1.00	1.00	0.33
SETLE	1.00	1.00	0.33
SETNB	1.00	1.00	0.33
SETNBE	1.00	1.00	0.33
SETNL	1.00	1.00	0.33
SETNLE	1.00	1.00	0.33
SETNO	1.00	1.00	0.33
SETNP	1.00	1.00	0.33

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
SETNS	1.00	1.00	0.33
SETNZ	1.00	1.00	0.33
SETO	1.00	1.00	0.33
SETP	1.00	1.00	0.33
SETS	1.00	1.00	0.33
SETZ	1.00	1.00	0.33
SFENCE	5.98	2.00	0.33
SHL	1.00	1.00	1.00
SHLD	1.01	1.00	3.00
SHR	1.00	1.00	1.00
SHRD	1.01	1.00	4.00
SHUFPD	1.02	1.00	1.00
SHUFPS	1.01	1.00	1.00
SLDT	5.97	0.33	0.33
SMSW	9.95	0.33	0.33
SQRTPD	9.96	10.00	7.00
SQRTPS	9.95	10.00	7.00
SQRTSD	10.00	10.00	7.00
SQRTSS	10.00	10.00	7.00
STC	0.34	0.33	0.33
STD	3.98	1.00	1.00
STR	5.97	0.33	0.33
SUB	1.00	1.00	1.00
SUBPD	3.00	3.00	3.00
SUBPS	2.99	3.00	3.00
SUBSD	3.00	3.00	3.00
SUBSS	3.00	3.00	3.00
TEST	0.34	0.67	0.67
TZCNT	3.00	3.00	0.74
UCOMISD	1.00	1.00	1.00
UCOMISS	1.00	1.00	1.00
UNPCKHPD	1.01	1.00	1.00
UNPCKHPS	1.00	1.00	1.00
UNPCKLPD	1.01	1.00	1.00

Instruction	CPI real (Sandy Bridge)	CPI ZSim (Enhanced)	CPI ZSim (Original)
UNPCKLPS	1.00	1.00	1.00
XADD	2.02	2.00	4.00
XCHG	1.53	1.50	1.50
XOR	1.00	1.00	1.00
XORPD	1.01	1.00	1.00
XORPS	1.00	1.00	1.00

Table A.1 Comparison of the Cycles per Instruction (CPI) – Instruction latency – of our 358 targeted `x86` instructions in the real machine, the enhanced version ZSim , and the original version of ZSim.

Publication List

This thesis includes text and images from the author version of the following published papers. URL's and Digital Object Identifier (DOI) to the published versions are provided in the following list:

1. **Rommel Sánchez Verdejo**, Kazzi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, Bruce Jacob, "Microbenchmarks for Detailed Validation and Tuning of Hardware Simulators", in the International Conference on High Perfomance Computing and Simulation (HPCS), July 2017. (doi: 10.1109/HPCS.2017.135 [139]).
2. **Rommel Sánchez Verdejo**, Kazzi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, Bruce Jacob, "Main memory latency simulation: the missing link", in the International Symposium on Memory Systems (MEMSYS), October 2018. (doi: 10.1145/3240302.3240317 [138]).

The following publications are under review:

1. **Rommel Sánchez Verdejo**, Kazzi Asifuzzaman, Milan Radulovic, Petar Radojković, Leonardo Bautista-Gómez, Eduard Ayguadé, Bruce Jacob, "The Elegant Simulation: Tuning System Simulators and the Quest for the Missing Cycles".
2. **Rommel Sánchez Verdejo**, Leonardo Bautista-Gómez, "Checkpointing on heterogeneous memory systems".

Other Publications:

1. Kazzi Asifuzzaman, **Rommel Sánchez Verdejo**, Petar Radojković, "Enabling a reliable STT-MRAM main memory simulation", in the International Symposium on Memory Systems (MEMSYS), October 2017. (doi: 10.1145/3132402.3132416 [10]).
2. Milan Radulovic, **Rommel Sánchez Verdejo**, Petar Radojković, "PROFET Modeling System Performance and Energy Without Simulating the CPU", in the ACM

Special Interest Group on Measurement and Evaluation (SIGMETRICS), December 2019. (doi: 10.1145/3341617.3326149 [116]).

③ Shang Li, **Rommel Sánchez Verdejo**, Petar Radojković, Bruce Jacob"Rethinking cycle accurate DRAM simulation", in the International Symposium on Memory Systems (MEMSYS), October 2019. (doi: 10.1145/3357526.3357539 [91]).

References

[1] Andreas Abel and Jan Reineke. 2019. Uops.Info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[2] Andreas Abel and Jan Reineke. 2020. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. (April 2020). http://arxiv.org/abs/1911.03282

[3] Abel, Andreas and Reineke, Jan. 2019. Table of instruction latency, throughput and micro-operation breakdowns for Intel's micro-architecture. (2019). https://uops.info/table.html

[4] Advanced Micro Devices 2020. AMD HBM Strategy. (2020). Retrieved February, 2021 from https://www.amd.com/en/technologies/hbm

[5] Agonne National Laboratory. 2019. Aurora Press Release. (2019). Retrieved June 2020 from "https://press3.mcs.anl.gov/aurora/"

[6] Alif Ahmed and Kevin Skadron. 2019. Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. ACM, New York, NY, USA, 167–172. https://doi.org/10.1145/3357526.3357574

[7] A. Akram and L. Sawalha. 2016. x86 computer architecture simulators: A comparative study. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 638–645. https://doi.org/10.1109/ICCD.2016.7753351

[8] A. Akram and L. Sawalha. 2019. A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access* 7 (2019), 78120–78145. https://doi.org/10.1109/ACCESS.2019.2917698

[9] A. R. Alameldeen and D. A. Wood. 2006. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro* 26, 4 (July 2006), 8–17. https://doi.org/10.1109/MM.2006.73

[10] Kazi Asifuzzaman, Rommel Sánchez Verdejo, and Petar Radojković. 2017. Enabling a Reliable STT-MRAM Main Memory Simulation. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/3132402.3132416

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

[12] Barcelona Supercomputing Center 2020. MareNostrum 4 (2017) System Architecture. (2020). https://www.bsc.es/marenostrum/marenostrum/technical-information

[13] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. 2011. FTI: High performance Fault Tolerance Interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2063384.2063427

[14] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, USA, 41.

[15] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018). arXiv:1802.09941 http://arxiv.org/abs/1802.09941

[16] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article Article 65 (Aug. 2019), 43 pages. https://doi.org/10.1145/3320060

[17] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[18] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. 2012. Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, USA, 199–211. https://doi.org/10.1109/MICRO.2012.27

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[20] Bingmann,Timo . 2013. Parallel Memory Bandwidth Benchmark / Measurement. (2013). Retrieved Sept 2019 from https://github.com/bingmann/pmbw

[21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[22] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (July 2006), 52–60. https://doi.org/10.1109/MM.2006.82

[23] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. *SIGPLAN Not.* 53, 2 (March 2018), 316–331. https://doi.org/10.1145/3296957.3173177

[24] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE (Ed.). Pisa, Italy. https://doi.org/10.1109/PDP.2010.67

[25] Randal E. Bryant. 2007. *Data-Intensive Supercomputing: The case for DISC*. Technical Report.

[26] BSC Memory Systems Group. 2019. memBench. (2019). https://github.com/bsc-mem/PROFET

[27] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. 2012. Accuracy evaluation of GEM5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. 1–7. https://doi.org/10.1109/ReCoSoC.2012.6322869

[28] C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, S. Hammond 2015. User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies. (2015). http://memkind.github.io/memkind/memkind_arch_20150318.pdf

[29] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

[30] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans. Archit. Code Optim.* 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[31] O. Celebioglu, A. Saify, T. Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini. 2004. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 250–. https://doi.org/10.1109/IPDPS.2004.1303311

[32] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship. (2012).

[33] CoMD main repository 2019. Classical molecular dynamics proxy application. Hash commit: `3d48396b`. (2019). https://github.com/ECP-copa/CoMD

[34] CoPA main repository 2019. COPA: Co-design center for Particle Applications. (2019). https://github.com/ECP-copa

[35] CoSP2 main repository 2019. Second Order Spectral Projection (SP2) for Electronic Structure Calculations. Has commit: `3e07ada0` . (2019). "https://github.com/exmatex/CoSP2"

[36] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1998582.1998590

[37] David Kanter. 2011. *Intel's Quick Path Evolved.* Real world technologies. https://www.realworldtech.com/qpi-evolved/3/.

[38] Di Biagio, Andrea. 2018. llvm-mca: a static performance analysis tool. (2018). https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

[39] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. 2016. Reliable and Efficient Performance Monitoring in Linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Article 34, 13 pages.

[40] Dongarra J., Meuer H. Meuer M., Simon H., Stronhmaier E. 2020. Top 500 Supercomputer sites. (2020). https://www.top500.org/

[41] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*. ACM, New York, NY, USA, Article 21, 3 pages. https://doi.org/10.1145/3329785.3329918

[42] E4C is a public-private consortium supported by the European Commission's Horizon 2020 tender for projects to counter the Coronavirus pandemic and improve the management and care of patients. 2020. The EXSCALATE4CoV (E4C) project. (2020). Retrieved July, 2020 from "https://www.exscalate4cov.eu/"

[43] Lieven Eeckhout. 2010. Computer Architecture Performance Evaluation Methods. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–145. https://doi.org/10.2200/S00273ED1V01Y201006CAC010 arXiv:https://doi.org/10.2200/S00273ED1V01Y201006CAC010

[44] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.* 65, 3 (2013), 1302–1326. https://doi.org/10.1007/s11227-013-0884-0

[45] Diego Elias, Rivalino Matias, Marcia Fernandes, and Lucio Borges. 2014. Experimental and Theoretical Analyses of Memory Allocation Algorithms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*. Association for Computing Machinery, New York, NY, USA, 1545–1546. https://doi.org/10.1145/2554850.2555149

[46] Jason Evans. 2015. Tick Tock, Malloc Needs a Clock. In *Applicative 2015 (Applicative 2015)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/2742580.2742807

[47] ExMatEx main repository 2019. ExMatEx: Exascale Co-design Center for Materials in Extreme Environments. (2019). "https://github.com/exmatex"

[48] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A Performance Counter Architecture for Computing Accurate CPI Components. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 175–184. https://doi.org/10.1145/1168917.1168880

[49] Fog, Agner. 2015. Lists of instruction latency, throughput and micro-operation breakdowns for Intel, AMD and VIA CPUs. (2015). https://agner.org/optimize/instruction_tables.pdf

[50] Fog, Agner. 2015. Software Optimization Resources. (2015). https://agner.org/optimize

[51] Free Software Foundation 2020. libc(7). (2020). Retrieved March, 2020 from "https://www.gnu.org/software/libc/"

[52] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 307–318.

[53] Brice Goglin. 2014. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *International Conference on High Performance Computing & Simulation (HPCS 2014)*. IEEE, Bologna, Italy. https://doi.org/10.1109/HPCSim.2014.6903671

[54] Brice Goglin. 2016. Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/2989081.2989115

[55] Brice Goglin. 2016. *Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc*. Research Report. Inria. https://hal.inria.fr/hal-01400264

[56] Google TCMalloc 2020. Thread-Caching Malloc . (2020). https://github.com/google/tcmalloc

[57] Google's EXEgesis 2019. Google's EXEgesis. (2019). https://github.com/google/EXEgesis

[58] Willian Dally David Ditzel Yale Patt Gordon Bell, Richard Sites. 1996. Architects Look to Processors of Future: Applications, Instruction Sets, Memory Bandwidth Are Key Issues. Retrieved July, 2020 from http://cva.stanford.edu/classes/cs99s/papers/architects_look_to_future.pdf

[59] T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, and M. Moreto. 2016. MUSA: A Multi-level Simulation Approach for Next-Generation HPC Machines. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 526–537. https://doi.org/10.1109/SC.2016.44

[60] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2017. Another Flip in the Wall of Rowhammer Defenses. *CoRR* abs/1710.00551 (2017). arXiv:1710.00551 http://arxiv.org/abs/1710.00551

[61] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[62] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. 2014. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 13–22. https://doi.org/10.1109/ISPASS.2014.6844457

[63] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[64] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[65] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[66] I. Molnar, T. Gleixner, et al. 2009. Performance Counters for Linux. (2009). https://lwn.net/Articles/337493/

[67] I. Molnar, T. Gleixner, et al. 2009. Performance Counters for Linux. (2009). https://lwn.net/Articles/337493/

[68] "IBM's Microprobe public repository" 2020. "Microprobe: Microbenchmark generation framework". (2020). Retrieved March, 2020 from https://github.com/IBM/microprobe

[69] Intel Corporation 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. Order Number: 248966-033.

[70] Intel Corporation 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Combined Volumes*. Intel Corporation. Order Number: 325462-061US.

[71] Intel Corporation 2019. Intel Architecture Code Analyzer. (2019). https://software.intel.com/en-us/articles/intel-architecture-code-analyzer-download/

[72] Intel Corporation 2019. Intel Product Specification site. (2019). Retrieved May, 2019 from https://ark.intel.com/

[73] Intel Corporation 2020. Intel Threading Building Blocks Developer Reference. (2020). https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator

[74] J. Edge 2009. "Perfcounters added to the mainline". (2009). https://lwn.net/Articles/339361/

[75] J. Evans 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006). https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[76] J. Evans 2006. Code repository for jemalloc. (2006). https://github.com/jemalloc/jemalloc

[77] Jacob, Bruce and Ng, Spencer and Wang, David. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[78] Jaleel, Aamer 2007. Memory Characterization of Workloads Using Instrumentation-Driven Simulation. A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites. (2007). Retrieved February 2021 from "http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf"

[79] Jim Jeffers, James Reinders, and Avinash Sodani. 2016. Chapter 4 - Knights Landing architecture. In *Intel Xeon Phi Processor High Performance Programming (Second Edition)* (second edition ed.), Jim Jeffers, James Reinders, and Avinash Sodani (Eds.). Morgan Kaufmann, Boston, 78 – 82. https://doi.org/10.1016/B978-0-12-809194-4.00004-1

[80] Min Kyu Jeong, Doe Hyun Yoon, and Mattan Erez. 2012. DrSim: A Platform for Flexible DRAM System Research. http://lph.ece.utexas.edu/public/DrSim. (2012).

[81] Jae-Eon Jo, Gyu-Hyeon Lee, Hanhwi Jang, Jaewon Lee, Mohammadamin Ajdari, and Jangwoo Kim. 2018. DiagSim: Systematically Diagnosing Simulators for Healthy Simulations. *ACM Trans. Archit. Code Optim.* 15, 1, Article 4 (March 2018), 27 pages. https://doi.org/10.1145/3177959

[82] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. 2014. Performance analysis of the memory management unit under scale-out workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. https://doi.org/10.1109/IISWC.2014.6983034

[83] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. LLNL. 1–9 pages.

[84] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance

Errors. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 361–372. https://doi.org/10.1145/2678373.2665726

[85] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[86] Andi Kleen. 2019. Simple NUMA policy support. (2019). https://github.com/numactl/numactl

[87] D. Komatitsch and J. Tromp. 2002. Spectral-element simulations of global seismic wave propagation-I. Validation. *Geophysical Journal International* 149, 2 (2002), 390–412. https://doi.org/10.1046/j.1365-246X.2002.01653.x

[88] Man lap Li, Ruchira Sasanka, Sarita V. Adve, Yen kuang Chen, and Eric Debes. 2005. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *In Proc. of the IEEE Int. Symp. on Workload Characterization*. 34–45.

[89] Pierre-François Lavallée. 2012. Porting and optimizing HYDRO to new platforms and programming paradigms – lessons learnt. (Dec. 2012). https://doi.org/10.5281/zenodo.814563

[90] Shang Li and Bruce Jacob. 2019. Statistical DRAM Modeling. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. ACM, New York, NY, USA, 521–530. https://doi.org/10.1145/3357526.3357576

[91] Shang Li, Rommel Sánchez Verdejo, Petar Radojković, and Bruce Jacob. 2019. Rethinking Cycle Accurate DRAM Simulation. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 184–191. https://doi.org/10.1145/3357526.3357539

[92] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. 2020. DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* (2020), 1–1. https://doi.org/10.1109/LCA.2020.2973991

[93] Linux Foundation 2018. Official Linux Kernel repositories for version 3.x. (2018). https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/

[94] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/3079079.3079089

[95] Ye Liu, Shinpei Kato, and Masato Edahiro. 2018. Is the Heap Manager Important to Many Cores?. In *Proceedings of the 8th International Workshop*

on Runtime and Operating Systems for Supercomputers (ROSS'18). Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3217189.3217194

[96] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[97] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[98] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda. 2019. Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs (GPGPU '19)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/3300053.3319419

[99] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 92–99. https://doi.org/10.1145/1105734.1105747

[100] Matthew Dobson, Patricia Gaughen, Michael Hohnbaum, Erich Focht 2003. Linux Support for NUMA Hardware. (2003). "https://www.kernel.org/doc/ols/2003/ols2003-pages-169-184.pdf"

[101] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404 (RAID 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[102] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[103] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96)*. USENIX Association, Berkeley, CA, USA, 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[104] Message Passing Interface Forum 2020. "MPI: A Message-Passing Interface Standard (2019 Draft specification)". (2020). https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf

[105] Micron Technology, Inc. 2019. Micron DDR3 SDRAM Verilog Model. (2019). Retrieved Sept. 2019 from https://www.micron.com/-/media/client/global/documents/products/sim-model/dram/ddr3/ddr3-sdram-verilog-model.zip

[106] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416635

[107] Daniel Molka, Daniel Hackenberg, and Robert Schöne. 2014. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. Association for Computing Machinery, New York, NY, USA, Article Article 4, 10 pages. https://doi.org/10.1145/2618128.2618129

[108] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.18

[109] Onur Mutlu and Jeremie S. Kim. 2019. RowHammer: A Retrospective. (2019). arXiv:cs.CR/1904.09724

[110] Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. 2009. Exploring speculative parallelism in SPEC2006. *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (2009), 77–88.

[111] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying Memory Underutilization in HPC Systems and Using It to Improve Performance via Architecture Support. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 821–835. https://doi.org/10.1145/3352460.3358267

[112] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC '11)*. ACM, New York, NY, USA, 1050–1055. https://doi.org/10.1145/2024724.2024954

[113] S. Patel. 2017. Behavioural study of memory allocators for Android platform. In *2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. 52–55. https://doi.org/10.1109/ICCE-ASIA.2017.8309320

[114] M. Poremba, T. Zhang, and Y. Xie. 2015. NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems. *IEEE Computer Architecture Letters* 14, 2 (July 2015), 140–143. https://doi.org/10.1109/LCA.2015.2402435

[115] Milan Radulovic, Rommel Sánchez Verdejo, Paul Carpenter, Petar Radojkovic, Bruce Jacob, and Eduard Ayguadé. 2019. PROFET: Modeling System Performance and Energy Without Simulating the CPU. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '19)*. ACM, New York, NY, USA, 71–72. https://doi.org/10.1145/3309697.3331502

[116] Radulovic, Milan and Sánchez Verdejo, Rommel and Carpenter, Paul and Radojković, Petar and Jacob, Bruce and Ayguadé, Eduard. 2019. PROFET: Modeling System Performance and Energy Without Simulating the CPU. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 34 (June 2019), 33 pages. https://doi.org/10.1145/3341617.3326149

[117] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. (DK) Panda. 2013. A 1 PB/s File System to Checkpoint Three Million MPI Tasks. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/2493123.2462908

[118] Daniel A. Reed and Jack Dongarra. 2015. Exascale Computing and Big Data. *Commun. ACM* 58, 7 (June 2015), 56–68. https://doi.org/10.1145/2699414

[119] Paul V. Bolotoff Rhett M. Hollander. 2002. ramspeed. (2002). Retrieved Apr 2018 from http://alasir.com/software/ramspeed/

[120] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. 2012. On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.* 8, 4, Article 36 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086715

[121] Robert Love 2002. "Task CPU affinity syscalls". (2002). https://lwn.net/2002/0418/a/affinity.php3

[122] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[123] Samsung Electronics 2020. Samsung Flarebolt. (2020). Retrieved February, 2021 from https://www.samsung.com/semiconductor/dram/hbm-flarebolt/

[124] Samsung Electronics Co., Ltd. 2011. *240pin Registered DIMM based on 2Gb D-die.* M393B5273DH0 Datasheet.

[125] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[126] Sandberg Andreas, Diestelhorst Stephan, Wang Willian. ASPLOS Tutorial 2017. Architectural Exploration with gem5. (2017). http://gem5.org/ASPLOS2017_tutorial

[127] Pavan Balaji Cyril Bordage George Bosilca Alex Brooks Adrian Castello Damien Genet Thomas Herault Prateek Jindal Laxmikant V. Kale Sriram Krishnamoorthy Jonathan Lifflander Huiwei Lu Esteban Meneses Marc Snir Yanhua Sun Sangmin Seo, Abdelhalim Amer and Pete Beckman. 2016. Argobots: A Lightweight, Low-Level Threading and Tasking Framework. *Argonne National Laboratory* (2016). http://www.mcs.anl.gov/papers/P5515-0116.pdf

[128] Joe Savage and Timothy M. Jones. 2020. HALO: Post-Link Heap-Layout Optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 94–106. https://doi.org/10.1145/3368826.3377914

[129] Nikolay A. Simakov, Joseph P. White, Robert L. DeLeon, Steven M. Gallo, Matthew D. Jones, Jeffrey T. Palmer, Benjamin Plessinger, and Thomas R. Furlani. 2018. A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD. (2018). arXiv:cs.DC/1801.04306

[130] SK-Hynyx 2020. SK-hynix News-Room - HB2E Mass production. (2020). Retrieved February, 2021 from https://news.skhynix.com/sk-hynix-starts-mass-production-of-high-speed-dram-hbm2e/

[131] SLURM Introduction 2020. SLURM Official Web Site. SLURM Quickstart. (2020). Retrieved March, 2020 from https://slurm.schedmd.com/quickstart.html

[132] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[133] "The kernel development community" 2003. NUMA Memory Policy. (2003). "https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html"

[134] J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA.

[135] Andy Turner and Simon McIntosh-Smith. 2018. A Survey of Application Memory Usage on a National Supercomputer: An Analysis of Memory Requirements on ARCHER. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer International Publishing, Cham, 250–260.

[136] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques.*

[137] A. Umayabara and H. Yamana. 2017. MCMalloc: A scalable memory allocator for multithreaded applications on a many-core shared-memory machine. In *2017 IEEE International Conference on Big Data (Big Data)*. 4846–4848. https://doi.org/10.1109/BigData.2017.8258563

[138] Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, and Bruce Jacob. 2018. Main Memory Latency Simulation: The Missing Link. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. ACM, New York, NY, USA, 107–116. https://doi.org/10.1145/3240302.3240317

[139] R. S. Verdejo and P. Radojković. 2017. Microbenchmarks for Detailed Validation and Tuning of Hardware Simulators. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 881–883. https://doi.org/10.1109/HPCS.2017.135

[140] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. 2018. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 28–44. https://doi.org/10.1177/1094342017718068 arXiv:https://doi.org/10.1177/1094342017718068

[141] Viswanathan, Vish and Kumar, Karthik and Willhalm, Thomas and Lu, Patrick and Filipiak, Blazej and Sakthivelu, Sri. 2019. Intel Memory Latency Checker. (2019). Retrieved June 2019 from https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[142] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A Memory System Simulator. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 100–107. https://doi.org/10.1145/1105734.1105748

[143] William Cohen's Github public repository 2020. Perfmon2 - libpfm4. (2020). Retrieved March, 2020 from "https://github.com/wcohen/libpfm4"

[144] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

[145] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[146] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[147] M. T. Yourst. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 23–34. https://doi.org/10.1109/ISPASS.2007.363733

[148] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 55–64. https://doi.org/10.1109/RTAS.2013.6531079

[149] Fucen Zeng, Lin Qiao, Mingliang Liu, and Zhizhong Tang. 2012. Memory Performance Characterization of SPEC CPU2006 Benchmarks Using TSIM. *Physics Procedia* 33 (2012), 1029–1035. https://doi.org/10.1016/j.phpro.2012.05.169 2012 International Conference on Medical Physics and Biomedical Engineering (ICMPBE2012).

[150] Darko Zivanovic, Pouya Esmaili Dokht, Sergi Moré, Javier Bartolome, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2019. DRAM Errors in the Field: A Statistical Approach. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/3357526.3357558

[151] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé.

2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article Article 3 (March 2017), 26 pages. https://doi.org/10.1145/3023362

[152] Zsim's github main repository 2016. ZSim's hash commit: `fb4d6e04`. (2016). https://github.com/s5z/zsim

[153] Javier Álvarez Cid-Fuentes, Pol Álvarez, Ramon Amela, Kuninori Ishii, Rafael K. Morizawa, and Rosa M. Badia. 2020. Efficient development of high performance data analytics in Python. *Future Generation Computer Systems* 111 (2020), 570 – 581. https://doi.org/10.1016/j.future.2019.09.051

List of Figures

List of Tables

Biographical Sketch

Rommel Sánchez Verdejo received his M.Eng. in Computer Engineering from the Universidad Nacional Autónoma de México, Mexico City (2009).

He had more than 15 years working in the computing industry; throughout 2003 and 2008, Rommel had given several technical talks in diverse Mexican forums. In 2006, he founded a start-up where he also acted as an architect and lead developer of a device that performs Automatic Vehicle Localization (AVL) and remote control over a GPRS network. In 2009 he started working for Intel Corporation in Jalisco, Mexico as a UEFI BIOS Engineer and Software Security Validation Engineer. In 2016, Rommel began research tasks at the Barcelona Supercomputing Center, Spain, along with the Univeristat Politècnica de Catalunya, pursuig a Ph.D. in Computer Architecture. His specialty and area of interest is in the software and hardware interactions targeting impact analysis on memory systems.

Rommel is currently (2021) working for Marvell Technology Group Ltd. as an Embedded Engineer performing tasks in system simulation.