UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1756

Efficient Memory Modeling During Simulation and Native Execution

NIKOS NIKOLERIS

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019

ISSN 1651-6214
ISBN 978-91-513-0538-7
urn:nbn:se:uu:diva-369490

Dissertation presented at Uppsala University to be publicly examined in Sal VIII, Universitetshuset, Biskopsgatan 3, Uppsala, Friday, 15 February 2019 at 09:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Lieven Eeckhout (Ghent University, Department of Electronics and Information Systems.).

Abstract

Nikoleris, N. 2019. Efficient Memory Modeling During Simulation and Native Execution. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1756. 73 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0538-7.

Application performance on computer processors depends on a number of complex architectural and microarchitectural design decisions. Consequently, computer architects rely on performance modeling to improve future processors without building prototypes. This thesis focuses on performance modeling and proposes methods that quantify the impact of the memory system on application performance.

Detailed architectural simulation, a common approach to performance modeling, can be five orders of magnitude slower than execution on the actual processor. At this rate, simulating realistic workloads requires years of CPU time. Prior research uses sampling to speed up simulation. Using sampled simulation, only a number of small but representative portions of the workload are evaluated in detail. To fully exploit the speed potential of sampled simulation, the simulation method has to efficiently reconstruct the architectural and microarchitectural state prior to the simulation samples. Practical approaches to sampled simulation use either functional simulation at the expense of performance or checkpoints at the expense of flexibility. This thesis proposes three approaches that use statistical cache modeling to efficiently address the problem of cache warm up and speed up sampled simulation, without compromising flexibility. The statistical cache model uses sparse memory reuse information obtained with native techniques to model the performance of the cache. The proposed sampled simulation framework evaluates workloads 150 times faster than approaches that use functional simulation to warm up the cache.

Other approaches to performance modeling use analytical models based on data obtained from execution on native hardware. These native techniques allow for better understanding of the performance bottlenecks on existing hardware. Efficient resource utilization in modern multicore processors is necessary to exploit their peak performance. This thesis proposes native methods that characterize shared resource utilization in modern multicores. These methods quantify the impact of cache sharing and off-chip memory sharing on overall application performance. Additionally, they can quantify scalability bottlenecks for data-parallel, symmetric workloads.

Keywords: performance analysis, cache performance, multicore performance, memory system, memory bandwidth, memory contention, performance prediction, multi-threading, multiprocessing systems, program diagnostics, commodity multicores, multithreaded program resource requirements, performance counters, scalability bottleneck, scalability improvement

Nikos Nikoleris, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0538-7
urn:nbn:se:uu:diva-369490 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-369490)

To my parents

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals:

I. Nikos Nikoleris, David Eklov, and Erik Hagersten. “Extending Statistical Cache Models to Support Detailed Pipeline Simulators”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2014, pp. 148–157. DOI: `10.1109/ISPASS.2014.6844479`

   I am the primary author of this paper.

II. Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E. Carlson. “CoolSim: Statistical Techniques to Replace Cache Warming with Efficient, Virtualized Profiling”. In: *Proc. Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. July 2016, pp. 106–115. DOI: `10.1109/SAMOS.2016.7818337`

   I am the primary author of this paper.

III. Nikos Nikoleris, Erik Hagersten, and Trevor E. Carlson. *Delorean: Virtualized Directed Profiling for Cache Modeling in Sampled Simulation*. Tech. rep. 2018-014. Department of Information Technology, Uppsala University, Dec. 2018

   I am the primary author of this paper. At the time of writing, this paper was under review.

IV. David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. “Cache Pirating: Measuring the Curse of the Shared Cache”. In: *Proc. International Conference on Parallel Processing (ICPP)*. ICPP ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 165–175. DOI: `10.1109/ICPP.2011.15`

   I was part of the initial discussion that lead to the idea of stealing cache capacity and performed a study to evaluate the accuracy of the method.

V. David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. “Bandwidth Bandit: Quantitative Characterization of Memory Contention”. In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. CGO ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. DOI: `10.1109/CGO.2013.6494987`

   I was part of the initial discussion that lead to the idea of stealing bandwidth, and prototyped an initial version of the bandit for the purposes of measuring an application’s MLP.

VI. David Eklov, Nikos Nikoleris, and Erik Hagersten. “A Software Based Profiling Method for Obtaining Speedup Stacks on Commodity Multi-cores”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2014, pp. 148–157. DOI: `10.1109/ISPASS.2014.6844479`

   I was part of the discussion throughout the formation of the idea, prototyped the framework for instrumenting pthread and authored parts of the paper.

Reprints were made with permission from the publishers. The papers have been reformatted to fit the single-column format of the thesis.

Other papers not included in this thesis:

VII. Andreas Sandberg, Trevor E. Nikoleris Nikos Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed". In: *Proc. International Symposium on Workload Characterization (IISWC)*. Washington, DC, USA: IEEE Computer Society, Nov. 2015, pp. 183–192. DOI: `10.1109/IISWC.2015.29`

I authored parts of the second version of the paper and run experiments with the SPECjbb benchmarks.

Summary

Detailed micro architectural simulation requires complex models that are thousands of times slower than native execution. Sampled simulation improves simulation speed and can reliably estimate performance by evaluating only a few representative portions of a workload. Sampled simulation performs well for microarchitectural explorations. First, the architectural state prior to the simulation points is checkpointed. Then, the checkpoints are restored multiple times to evaluate different design points. Complex scenarios involving hardware-software co-design (such as co-optimizing both a Java virtual machine and the micro architecture it is running on) cause this methodology to break down, as new architectural checkpoints are needed for each memory hierarchy configuration and software version.

In this work, we present Full Speed Ahead (FSA). A methodology that leverages hardware virtualization to allow for fast-forward between simulation points at near-native speed. FSA enables fast and flexible sampled simulations without the limitations of checkpoints. FSA uses functional simulation to warm up caches and a method that bounds the estimation error when caches have not been sufficiently warmed up. We demonstrate a parallel sampling simulator that can be used to accurately estimate the IPC of standard workloads with an average error of 2.2% while still reaching an execution rate of 2.0 GIPS (63% of native) on average.

VIII. Ricardo Alves, Nikos Nikoleris, Stefanos Kaxiras, and David Black-Schaffer. “Addressing Energy Challenges in Filter Caches”. In: *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Nov. 2017, pp. 49–56. DOI: `10.1109/SBAC-PAD.2017.14`

I contributed to the initial idea.

Summary

Cores rely on fast data caches to hide the latency of a large fraction of memory instructions. Filter caches and way-predictors are common approaches to improve the efficiency and performance of data caches. Filter L0 caches are smaller and provide more efficient and faster access to a small subset of the data. Way-predictors remove the need to read all ways in parallel without increasing latency by speculatively accessing one of the ways.

In this work, we examine how SRAM layout constraints (h-trees and data mapping inside the cache) affect way-predictors and filter caches. We show that accessing the smaller L0 array can be significantly more energy efficient than attempting to read fewer ways from a larger L1 cache; and that the main source of energy inefficiency in filter caches comes from L0 and L1 misses.

We propose a filter cache that shares the tag array between the L0 and the L1, which incurs the overhead of reading the larger tag array on every access, but in return allows us to directly access the correct L1 way on each L0 miss. This optimization does not add any extra latency and counter-intuitively, improves the filter caches overall energy efficiency beyond that of the way-predictor. By combining the low power benefits of a physically smaller L0 with the reduction in miss energy by reading L1 tags upfront in parallel with L0 data, we show that the optimized filter cache reduces the dynamic cache energy compared to a traditional filter cache by 26% while providing the same performance advantage. Compared to a way-predictor, the optimized cache improves performance by 6% and energy by 2%.

Contents

List of Terms

CPI Cycles Per Instruction. 32, 33, 41, 42, 51

DIMM Dual In-line Memory Module. 44

DRAM Dynamic Random Access Memory. 5, 43–45, 47

FSA Full Speed Ahead. 9

IPC Instruction Per Cycle. 38, 54

LRU Least Recently Used. 11, 12, 40

MRI Memory Reuse Information. 18, 24–26, 28–32, 34, 35, 60

MRU Most Recently Used. 40, 41

MSHR Miss Status Holding Register. 20, 22, 24, 27, 29, 30, 32

NVM Non-Volatile Memory. 5

OS Operating System. 10

PC Program Counter. 24, 25, 28, 30–32

PCB Process Control Block. 10

PMC Performance Monitoring Counter. 9

PMU Performance Monitoring Unit. 9, 10, 13, 38, 39, 53

TSC Time Stamp Counter. 9, 55

VDP Virtualized Directed Profiling. 30, 31

VFF Virtualized-Fast Forwarding. 9, 16–18, 26, 29–31, 35, 60

1 Introduction

Over the years, improvements in computers have made them faster and cheaper. Consequently, computers can perform computations and move data faster than ever before. These improvements are neither an esoteric quest of computer architects, nor driven by a small number of uses. Many advances in sciences, education, business and nearly every aspect of our everyday life have come to rely on computers. In many cases, improvements in computer performance have a direct impact on medical research, cosmology, meteorology etc.

Historically, this continuous improvement has been due to a number of bigger and smaller optimizations. Advances in materials, transistors, integrated circuits, microarchitecture, architecture and software have contributed to making computers faster and cheaper.

However, for many years, continuous advances in transistor technology resulted in steady improvements in computer performance. As transistors got smaller, their power density remained constant [9], a phenomenon known as Dennard scaling. The transistor power efficiency gains afforded by Dennard scaling allowed computer architects to raise clock frequencies from one generation to the next, without significantly increasing power consumption.

Frequency scaling continued for more than three decades. However, since around 2006, transistor efficiency gains have become harder due to leakage [34]. Transistors get smaller, but leakage power is increasing and leaving little to no room for further increases in clock frequency.

Advances in transistor technology were not limited to Dennard scaling. As first observed by Moore [22], due to advances in manufacturing, the number of transistors in densely integrated circuits doubled every two years. With every new generation, computer architects had twice as many transistors available to improve computer processors. Microarchitectural improvements, larger memories, more computational resources and aggressive speculation techniques were all possible largely due to Moore's law. Since Dennard scaling broke down, Moore's law allowed for many microarchitectural optimizations, and processor performance became entirely dependent on microarchitectural improvements.

As a result of Moore's law, today's processors implemented in the latest manufacturing processes (e.g., nodes of 7 nm, 10 nm) feature billions of transistors (e.g., 19.2 B transistors in AMD Epyc 7601). These transistors are used to implement a number of cores and a complex memory hierarchy[1].

The design complexity of such a processor cannot be easily overstated. Parameters such as the depth of the pipeline, the width of each pipeline stage, instruction scheduling (e.g., in-order, out-of-order), the branch predictor, the size of the physical register file, the number of integer and floating point units or the size of the processor's private memory resources (e.g., data, instruction and micro-op caches) determine the performance of each of the cores. Parameters such as the size of the shared caches, prefetchers, the coherence protocol, the number of memory controllers and the technology of the off-chip memory (e.g., DDR3, DDR4) determine the performance of the memory system. The interactions of all the cores in the shared memory system only exacerbate the design complexity.

Due to the processor design complexity, computer architects have to use systematic approaches and models to understand the performance of future processors. Typically, these performance models range from high-level analytical formulas, to detailed microarchitectural simulators. Abstract high-level models provide quick insights on general performance trends, whereas detailed low-level performance models are typically slow but provide more accurate performance measurements.

For that reason, performance modeling has received a lot of attention over the years [37, 28, 7, 31, 11, 3, 19, 17, 18, 23, 32, 35] and it is the primary topic of this thesis. In particular, this thesis focuses on the impact of the memory system on application performance.

In the first part, this thesis proposes three techniques that combine statistical cache modeling with architectural simulation. Detailed, architectural simulation, a common approach to performance modeling, can be five orders of magnitude slower than execution on the actual processor. At this rate, simulating realistic workloads requires years of CPU time. To speed up simulation, practical approaches use sampled simulation but they sacrifice modeling flexibility. To enable flexible and efficient sampled simulation, this thesis proposes statistical cache models (Paper I, Paper II and Paper III) that eliminate cache warm up and enable fast and flexible sampled simulation. Sampled simulation with statistical cache modeling is up to 150× faster than common approaches that use functional simulation to warm up the cache.

In the second part, this thesis proposes three techniques that quan-

[1] Typically, processors will also have a number of peripheral devices with similar complexity.

tify the impact of shared memory resources on application performance, using data captured from native execution. Efficient resource utilization in modern multicore processors is necessary to achieve their peak performance. The proposed techniques focus on shared resource utilization in modern multicores. Cache Pirating (Paper IV) proposes a method to quantify the performance impact of cache sharing and the Bandwidth Bandit (Paper V) proposes a method that quantifies the performance impact off-chip memory sharing. Finally, the Speedup Stacks (Paper VI) propose a method that leverages insights from the previous two methods to quantify the scalability bottleneck for parallel, symmetric workloads with no data sharing.

The rest of this thesis is organized as follows. In Chapter 2, we introduce a number of key concepts which are used as a basis for the proposed models. In Chapter 3 (Paper I, Paper II and Paper III), we describe the statistical cache models that enable efficient and flexible sampled simulation. In Chapter 4 (Paper IV, Paper V and Paper VI), we propose novel quantitative approaches to modeling contention for the shared memory resources in modern multicores.

2 Background

2.1 Memory Hierarchy

General purpose computers, since their inception more than half a century ago, use a processing unit (core), a memory to store data and instructions (main memory) and a number of devices for input/output operations (I/O devices).

This thesis focuses on the memory subsystem and its impact on application performance. To this end, it proposes analytical and statistical models to quantify the performance impact of common hardware mechanisms that hide the latency of main memory accesses. In this section, we describe the basics of the memory organization in modern computer processors.

Main Memory

Today's computer systems, typically, use Dynamic Random Access Memory (DRAM) as main memory. DRAM accesses to fetch data, typically, take two orders of magnitude more time, and consume four orders of magnitude more energy, than common arithmetic operations. For many years, researchers and architects have designed mechanisms to bridge this performance gap between core execution rate and main memory performance.

However, this performance gap is not likely to close in the near future. Emerging main memory technologies, e.g., Non-Volatile Memory (NVM), that offer an attractive alternative to DRAM, promise higher memory density, but are expected to have similar, if not worse, performance characteristics. As a result, latency hiding and bandwidth optimization techniques will remain relevant for future processor design due to their impact on application performance.

Memory Hierarchy

To mitigate the time and energy cost of accessing main memory, computer architects design cache memories. A cache memory, or simply cache, is a smaller and faster memory. A cache is typically placed between the core and the main memory, and it stores copies of the data from locations of the main memory. Memory accesses that are satisfied by the cache can be up to two orders of magnitude faster than accesses to main memory.

Caches, like other memories, trade size for speed. As the gap between core and main memory performance has widened through time, architects bridge this gap using a hierarchy of caches. Small and faster caches are placed close to the core, while larger and slower caches are placed further down the memory hierarchy. For example, Intel Xeon E5-1660 v3 has two private, split, Level 1 caches, one for data (D-Cache) and one for instructions (I-Cache); a unified Level 2 cache (L2-Cache), private for each core; and a unified Level 3 cache (L3-Cache) shared among all cores in the processor. While less common, Intel and IBM include a Level 4, large eDRAM cache in some of their higher end processors.

A deep and complex hierarchy of caches is part of every processor which is designed for performance and energy efficiency. As a result, the cache hierarchy takes a significant amount of area. Often the area footprint of the cache hierarchy is comparable to or larger than the footprint of the cores (fig. 21). The reason for this design decision is that caches can bring significant performance improvements to applications that use them efficiently. However, for applications with poor memory locality, caches waste energy due to unnecessary cache look-ups and take up valuable silicon area.

2.2 Modeling Performance

Computer architects use performance modeling to find bottlenecks and optimize hardware. However, detailed models of today's computer processors are hard to implement, often requiring a dedicated team of engineers, and slow, often requiring years of CPU time to evaluate large realistic workloads.

To make performance modeling practical, prior research proposes approaches to speed up simulation either at the expense of modeling accuracy [18, 16, 7] or at the expense of modeling flexibility [37, 35, 28, 33]. For most performance modeling approaches, there is a three way trade-off between simulation overhead, accuracy and flexibility.

The rest of this chapter introduces key concepts and modeling techniques that are relevant to this thesis.

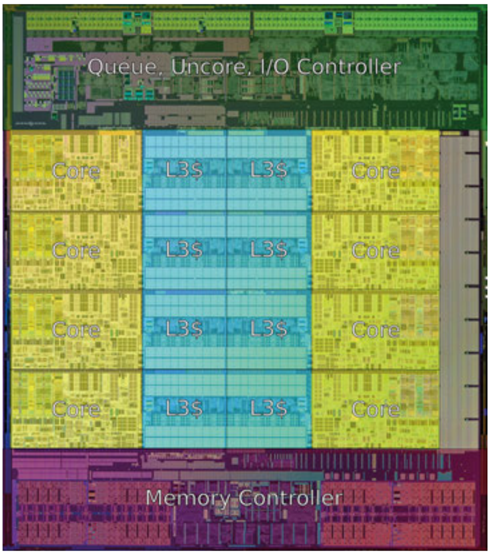


Figure 21: Thermal imaging of the silicon die of an octa-core Intel Haswell processor. The total area of the distributed L3 Cache, highlighted in light blue, is comparable in size with the total area of the eight cores, highlighted in light yellow. Note that the area in yellow, attributed to cores, includes at least a D-Cache, an I-Cache and an L2-Cache. (Image source: WikiChip November 01, 2018; use permitted under the Creative Commons Attribution License CC BY 3.0.)

Architectural Simulation

As in many other sciences, simulation has been instrumental in computer research and engineering. Computer architects use simulation to model hardware and understand performance and power trade-offs. Typical uses range from simulations of particular components (e.g., branch predictor simulators, cache simulators, etc.) to full computer systems. Detailed simulation can result in very accurate measurements, representative of the performance and power characteristics of the actual components or systems.

In practice, however, as computer systems continue to grow in complexity, faithful simulation models become increasingly more challenging. As an example, gem5 [6], a popular and widely used full-system simulator, can evaluate applications at a rate of 100 kIPS. At the same time, common benchmarks such as the SPEC CPU2006 have 327 B to 5189 B instructions [29] and benchmarks in the more recently released SPEC CPU2017 are roughly an order of magnitude larger 1 T to 66 T instructions [27]. Simulating any of these benchmarks would take years worth of simulation time.

A large body of prior research focuses on making simulation faster. Commonly used approaches either tackle a) the problem of modeling hardware complexity by raising the level of abstraction in simulation [18, 16, 7, 18, 17], or b) the problem of software complexity by reducing large, complex applications in a representative sample [37, 35, 28, 33]. In many cases, both approaches are employed to enable simulation of large software stacks on complex computer systems. In this work, we consider primarily the second type of approaches.

Accelerating Simulation with Sampling: Sampled simulation frameworks [37, 33] use statistics to reduce large workloads into a number of small but representative samples. Sherwood et al. [33] capture basic block vectors and use a clustering algorithm to identify regions of the execution with similar code signature, and classify them into phases. Phases exhibit similar performance characteristics. To choose representative samples, they select samples from distinct phases. Using simple statistics (weighted average) on performance metrics obtained by simulating the samples, they can estimate performance metrics of the full application.

Wunderlich et al. [37] use systematic sampling to reduce simulation time. They use rigorous statistics to find the sampling parameters that reduce the total number of simulated instructions, while bounding the estimation error within the desired confidence interval. They demonstrate that for many applications, the optimal sampling unit (or simulation point) is 10 k instructions. Smaller sampling units require a significant increase in the number of simulation points (sample), whereas using bigger sampling units does not allow for a proportionate reduction in the number of simulation points. Their results show that sampled simulation can significantly reduce the simulation overhead, while estimating performance metrics within ±3% with 99.7% confidence.

Accelerating Sampled Simulation with Hardware Virtualization: Sampled simulation achieves significant speedup compared to detailed simulation. The workload execution is fast-forwarded between simulation points using functional simulation, which is typically an order of magnitude faster than detailed simulation. As a result, detailed simulation is only used at the simulation points which account for less than 1% of the total execution [37].

Accounting for 99% of the execution, functional simulation dominates simulation time. For design space explorations, where the same workload is evaluated multiple times with different microarchitectural parameters, practical approaches to sampled simulation use checkpoints. The execution is checkpointed prior to the simulation points and later restored to evaluate the different parameters. This way, the cost of functional simulation, which otherwise dominates performance, is amortized. However, when the workload is not entirely static (e.g., run-time parameter changes, hardware-software co-design, Java workloads, etc.), architectural checkpoints cannot be reused and functional simulation dominates the simulation overhead.

To improve the efficiency of sampled simulation, we propose Full Speed Ahead (FSA) (Paper VII) [31]. FSA uses Virtualized-Fast Forwarding (VFF) instead of functional simulation to advance the execution between simulation points. VFF offloads the execution using hardware virtualization and executes at near-native speed between simulation points. FSA is up to three orders of magnitude faster than traditional sampled simulation without requiring architectural checkpoints. This added flexibility is particularly important when there are changes in the workload and architectural checkpoints cannot be reused.

Native Performance Modeling

A number of tools and techniques allow hardware and software developers to measure performance on existing processors. These native tools and techniques range from low-level, hardware mechanisms like timers (e.g., Time Stamp Counter (TSC)) to high-level tools that combine profiling and debugging technologies (e.g., Linux perf_events) to pinpoint specific application bottlenecks. Native tools can be used to find performance bottlenecks and enable software and hardware optimizations.

Performance Monitoring Unit

A particularly useful mechanism in understanding performance is the Performance Monitoring Unit (PMU), also known as Performance Monitoring Counters (PMCs). The PMU is typically used to measure a number of different architectural and microarchitectural events. These events reveal workload characteristics (e.g., number of executed instructions) and show how well the processor is used (e.g., cache misses).

Initially, the PMU was an undocumented feature in many processors. It was designed by hardware architects for their own use and allowed them to understand key performance characteristics after tape-out. In 1994, Mathisen [20], reversed engineered this undocumented feature for the Intel Pentium processors and published his results. Then, researchers started using the PMU to build tools and techniques, generate insights, and drive software optimization. As the PMU got more attention, processor vendors started documenting and enhancing it.

The PMU in modern processors consists of a set of programmable and fixed-function performance counters. The fixed-function counters allow easy access to frequently used events (e.g., the number of retired instructions) while the programmable counters can be configured to measure the desired event.

The configuration of the PMU and the counters are exposed to the user through memory mapped registers. In many processor architectures, accessing these registers is a privileged operation carried out by the Operating System (OS). The OS has direct accesses to the PMU and provides means for programming and reading the performance counters to user-space applications.

Typically, the OS virtualizes the hardware performance counters to allow for better attribution and per-process measurements. The performance counters are stored in the Process Control Block (PCB) which is updated on every context switch. Additionally, for programmable counters, one can mask events that are generated from user space, kernel space or from the hypervisor.

Hardware Performance Counters Overflows: Performance counters are memory mapped registers which are finite in size (about 40 bits in Intel CPUs). When a counter overflows, it generates an interrupt. This interrupt allows the OS to handle the overflow and support larger measurements.

Performance counter overflows can be used to stop the execution after a predefined number of events. For example, to stop the execution after n instructions, one can program a counter to measure the executed instructions and initialize it to $MAX - n + 1$ — where MAX is the maximum number that the counter can fit (e.g., 2^{40} for a 40 bit counter). The PMU will start counting and after n instructions, the counter will overflow and an interrupt will stop the execution.

Analytical Performance Modeling

Analytical models use low-level metrics obtained from simulation or native execution to estimate higher-level performance, power or energy metrics. For example, one of the simplest analytical models, the *one-IPC model*, assumes that every instruction takes one cycle to execute. The one-IPC model can be used to estimate the execution time (cycles) of an application, using as input the number of executed instructions. Many analytical performance models are used to estimate or approximate metrics. These estimations are not always as accurate as when using simulation, but often they are much faster.

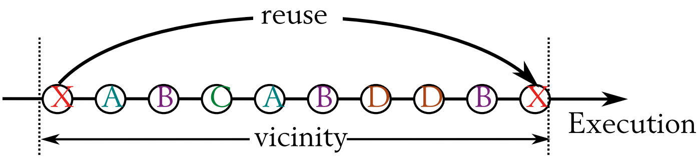


Figure 22: Memory reuse of the cache block X. The reuse distance of the second reference to X is eight, as there are eight memory accesses between the two accesses to X, while its stack distance is four, as there four distinct blocks A, B, C and D accessed between the two uses to X

Analytical Cache Modeling

Analytical cache models are used to understand how well caches are used. Commonly used metrics, indicators of cache performance, are the number of misses, the miss ratio — the number of misses over the number of accesses to the cache, and others.

Memory Stack Distance: The *stack distance* is a metric that can be used to determine whether a memory access hits or misses in the cache. First introduced by Mattson et al. [21], the stack distance of a memory access M that references a cache block X (fig. 22) is defined as the number of distinct cache blocks accessed since the last access to block X. In a fully associative cache with Least Recently Used (LRU) replacement policy, M will hit in the cache if and only if its stack distance is less or equal than the size of the cache in blocks.

Stack distance as a metric is hardware independent. If we execute the same application with the same input set, it will have the same stack distances, regardless of the processor it executes on[1]. This property makes the stack distance a very useful metric for software developers that want to understand memory locality properties without necessarily targeting optimizations for specific processors. Also, it is very useful for hardware architects, who can use it to understand how cache design decisions impact the miss ratio of a set of benchmarks. However, obtaining stack distances is an expensive process [3, 4, 10]. Prior work [10] uses statistical modeling to estimate stack distances. This statistical model is based on memory reuse information which can be obtained more efficiently.

Memory Reuse Distance: The reuse distance of a memory access M to cache block X (fig. 22), is the number of memory accesses since the last access to X. Berg and Hagersten [3] show that a sparse sample of a workload's memory reuse distance information allows for accurate modeling of fully associative caches with random replacement policy.

[1] For processors that speculatively issue memory accesses, stack distances will be the same only if our measurements use the committed memory instructions in program order.

The reuse distance, similarly to the stack distance, is a hardware independent metric and can be used to estimate an application's miss ratio for caches of any size.

Statistical Estimation of Stack Distance: Eklov and Hagersten [11] propose StatStack, an efficient statistical method to estimate stack distances using a sparse sample of reuse distances. StatStack can estimate a stack distance of a memory access M from the reuse distance of M and a histogram of the distribution of reuse distances in the vicinity of M. Eklov and Hagersten use StatStack to accurately estimate an application's overall miss ratio for LRU, fully-associative caches.

Efficient Reuse Distance Profiling: Berg and Hagersten [3] demonstrate a software profiler that samples reuse distances using performance counters on commodity hardware. The reuse sampling rate has a first order effect on the overhead of the sampler. They are able to sample every $10,000$ to $100,000$ memory instructions with an overhead of 40% compared to native execution.

To obtain a representative sample of the application's memory reuse information, the profiler a) selects random memory instructions, and b) for every selected memory instruction, obtains the associated reuse distance.

To select random memory instructions, the profiler uses a random number generator and performance counter overflows. Its goal is to stop the execution after r instructions, where r is a random number that follows an exponential distribution. To achieve this, the profiler programs a performance counter to overflow after r memory instructions. In many modern complex out-of-order CPUs, however, performance counter overflow interrupts are not precise [3, 8]. The processor's pipeline can potentially have many outstanding instructions when the actual overflow happens. Nevetherless, some of the outstanding instructions can potentially commit before the corresponding interrupt is raised. This is typically referred to as skid.

The processor skid can bias the sampling process. As a result of the skid, the processor often stops much later (in program order) than the desired memory instruction that causes the overflow. In theory, the profiler could advance the execution (e.g., by single stepping) and sample the next memory instruction instead. However, in practice, doing so would introduce bias in the sampling process. Often the amount of skid is correlated to the memory latency of the instructions that are executed when the overflow happens, and sampling the memory instruction right after, would significantly bias the sampling process.

To avoid any bias and collect the desired samples, the profiler compensates for the skid. It programs a counter overflow after $r - s$ memory instructions, where s is a compensation factor[2]. When the processor stops after a performance counter overflow, it single steps the execution until r memory instructions have been executed since the last sample. If for any reason the execution stops after the desired memory instruction, then the sample is ignored and the process starts once again to collect the next sample. Compensating for the skid adds significant overhead to the sampling process, as for every sample, the profiler has to single step through a number of instructions.

For any random memory instruction that accesses block X, the profiler measures the number of memory instructions until the next access to X. To make the measurement of long reuses efficient, after locating the random memory instructions, the profiler places a watchpoint on block X and resumes the execution. When the block X is accessed again, the execution stops again. The profile uses the PMU to record the reuse distance to X.

The profiler relies on watchpoints to find reuses. Ideally, the profiler would use hardware watchpoints which have low overhead. Hardware watchpoints, however, are limited in number (e.g., Intel Xeon E5-1660 v3 has four) and size (e.g., up to 8 B) while the profiler needs a large number of watchpoints that can trigger on read or writes on a cache line granularity (64 B). The profiler uses the memory page protection mechanism to implement watchpoints. This implementation allows for a large number of watchpoints (practically unlimited) and watchpoints on cache line granularity.

The profiler protects the memory page against reads and writes and every time a memory instruction touches the protected page, the execution stops. If the memory instruction references a tracked cache line, then it has triggered a watchpoint. Otherwise, the page fault is a false positive and the execution is resumed. False positives add overhead to the sampling process and reducing them can improve the sampler's performance significantly (Paper III).

[2] s is the maximum expected skid.

3 Combining Statistical Memory Modeling with Simulation

Researchers use computer simulation to model future hardware, understand performance trade-offs and, ultimately, optimize hardware and software. Simulators can achieve high modeling fidelity through software models that mimic hardware in detail. The associated overhead, however, can be prohibitively high as detailed simulation of realistic workloads often requires years worth of CPU time.

In this chapter, we describe the contributions of this thesis that make simulation faster. Our work builds on sampled simulation frameworks and combines them with statistical cache modeling. Using statistical cache modeling, we eliminate cache warm-up and make sampled simulation more efficient, even when simulating systems with large caches. Our approach improves the overall modeling flexibility by removing constraints in the modeled memory hierarchy that otherwise would be lifted only at the expense of modeling performance.

3.1 Overhead in Sampled Simulation

Sampled simulation frameworks like SMARTS [37] use rigorous statistics to reduce large workloads to a number of small simulation points. This set of small simulation points is chosen to be representative of the whole workload. Consequently, the evaluation of a workload relies on the accurate evaluation of the simulation points. To accurately evaluate a simulation point, our framework has to:

a) Advance the execution of the workload (fast-forwarding) to the simulation point to find the appropriate *architectural state*; and

b) Warm up large, performance-critical structures such as caches and branch predictors to find the appropriate *microarchitectural state*.

SMARTS [37] satisfies these two requirements using functional simulation to advance to the next simulation point and warm up the caches,

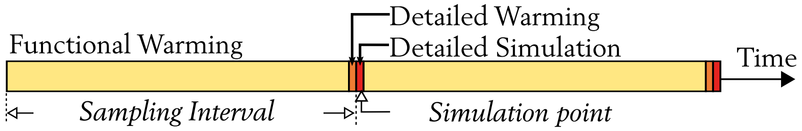


Figure 31: SMARTS: Steps in sampled simulation.

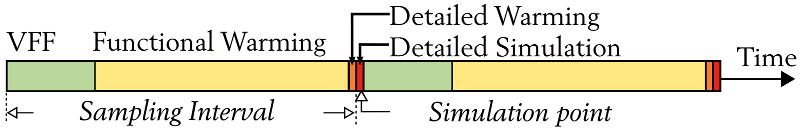


Figure 32: FSA: VFF is used to fast-forward at near native speed.

and a small amount of detailed simulation prior to the simulation point to warm up the rest of the microarchitectural state (fig. 31).

The cost of determining the appropriate architectural and microarchitectural state can be amortized with the use of checkpoints. By checkpointing the system in advance, and restoring it for each experiment, the detailed simulations can readily proceed.

Using checkpoints, sampled simulation can be efficient for evaluating a particular system. However, it requires that the software and most hardware configuration parameters remain fixed. Significant software changes (such as JIT compilation strategies for managed languages, updated compilers or compiler options), or hardware changes (e.g., cache size) require generation of new checkpoints. The regeneration of checkpoints requires functional simulation which can dominate simulation time. Using checkpoints, we can improve the efficiency of sampled simulation, at the expense of modeling flexibility (software and/or hardware changes).

Efficient Fast-Forwarding

Virtualized-Fast Forwarding (VFF) [31] (Paper VII) allows near-native speed fast-forwarding to the next simulation point, without the restrictions of checkpoints. To achieve its efficiency, VFF offloads the execution of the simulated system to real hardware using hardware virtualization. Using VFF, the problem of finding the appropriate architectural state is solved by, simply, executing the simulated workload natively. However, while virtualized execution addresses the problem of efficient fast-forwarding, for accurate sampled simulation, we still need a way to warm-up the microarchitectural state (fig. 32).

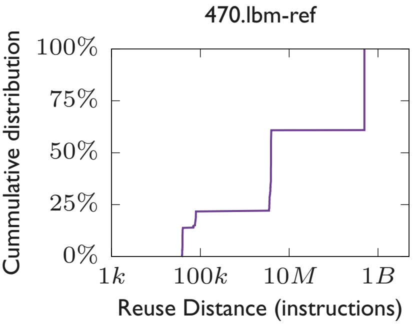


Figure 33: The longest memory reuse for 470.lbm is almost 1 B instructions. Consequently, fully warming up a cache that can fit its working set size will also take almost 1 B instructions.

Efficient Microarchitectural Warming

A large part of the microarchitectural warm up is necessary due to large caches. 470.lbm-ref (fig. 33) requires simulation of almost 1 B instructions to warm up a large DRAM cache that fits its entire working set. If we used functional simulation to warm up 470.lbm-ref, cache warm-up would dominate the overhead of the sampled simulation and leave little to no scope for VFF [31].

3.2 Statistical Cache Modeling in Sampled Simulation

In this work, we propose to use statistical cache modeling to lower the overhead of sampled simulation. Statistical cache models, as we will explain in more detail, remove the requirement for precise history information and therefore lend themselves to efficient sampled simulation.

Typically, architectural simulators use mechanistic cache models to model processors with caches. A mechanistic cache model maintains the state of the cache to determine whether memory accesses hit or miss in it. Every memory access updates the state of the model as it would in a real cache. As a result, to accurately estimate performance, the model relies on the observation of previous memory accesses. This requirement makes mechanistic cache simulators inefficient in sampled simulation. While sampled simulation has the potential to reduce significantly the required simulation, cache warm up is the limiting factor. For the rest of this chapter, we refer to the mechanistic cache model of the simulator as the simulator's cache.

Statistical cache models, on the other hand, rely on sparse information about the workload's memory behavior. First, the workload is profiled to obtain Memory Reuse Information (MRI) and then a rigorous statistical model uses the obtained information to estimate the performance of the cache. This obtained MRI can be fairly sparse, as long as it is representative of the workload. For this reason, statistical cache modeling based on sparse input information can improve the efficiency of sampled simulation.

However, like other statistical approaches, statistical cache modeling works best for estimating aggregate metrics (e.g., cache miss ratio) rather than determining the outcome of a specific event (e.g., whether a specific memory access hits or misses in the cache). For statistical modeling to be useful in sampled simulation, our model has to determine if any given memory accesses executed in the simulation point hits in the cache. Consequently, our statistical cache model has to overcome this limitation to provide accurate information to the simulator.

The rest of this chapter looks into statistical cache modeling and how we can use it with sampled simulation, to enable more efficient architectural simulations. WarmSim (Paper I) explores the feasibility and the accuracy of such a statistical model and sets the requirements for its MRI input. We demonstrate that, using statistical cache modeling based on sparse MRI, we can eliminate cache warm-up in sampled simulation. Then we describe subsequent works that propose efficient MRI profiling technologies. CoolSim (Paper II) introduces an MRI profiler that leverages VFF. Finally, the last work in this chapter, Delorean (Paper III), leverages the speed of VFF to build a multi-pass profiler and optimize further the MRI collection. Delorean's approach to profiling allows us to obtain precise MRI information and greatly simplify its cache model.

3.3 WarmSim: Statistical Cache Modeling in Simulation

WarmSim (Paper I) proposes a statistical cache model that eliminates the need for cache warming in sampled simulation using sparse MRI as input. Rather than completely replacing the simulator's cache, WarmSim uses a statistical cache model to correct it, when its estimations are wrong due to insufficient warm-up. When we eliminate cache warm-up, sampled simulation goes through three distinct steps:

a) Fast-Forwarding: We advance the execution to the next simulation point.

b) Detailed Warm-up: We switch to detailed simulation 10,000 instructions before the simulation point. We reset the state of the simulator and run for 10,000 instructions.

c) Detailed Simulation: At the simulation point, we run for 10,000 instructions and record the desired performance measurements for this simulation point.

The same process repeats until we reach the end of the simulated workload.

To avoid any biases in estimating performance, the Detailed Simulation in step c has to be precisely the same as if the workload had been simulated from the beginning. Any deviations will introduce sampling bias and will perturb our performance estimations.

To achieve this, we perform Detailed Warming in step b to warm up the microarchitectural state of the system. Like in SMARTS [37], Detailed Warming has to be long enough to sufficiently warm the microarchitectural state but not longer than necessary, as it adds overhead. Typically, most of the processor's microarchitectural state is relatively small and simulating for a few thousands of instructions can sufficiently warm it up. However, this is not true for the caches, as their size can be orders of magnitude larger than most of the other structures in the system. As we demonstrated earlier, to sufficiently warm up the cache, we may have to simulate for millions of instructions.

To accurately model the cache performance without warm-up, WarmSim uses statistical cache modeling. WarmSim's cache model comprises of three components (fig. 34) which we describe in the rest of this section.

Leveraging a Lukewarm Cache

When Detailed Simulation starts, the cache is not sufficiently warmed up and, consequently, the simulator can not rely on it. However, the cache is not empty (cold) either, a small amount of simulation (Detailed Warming) has populated part of its state. Further, as simulation progresses through the simulation point, the cache keeps track of the short history of previous memory accesses. At this intermediate state, we refer to the cache as lukewarm.

The lukewarm cache is not sufficiently warmed up and might introduce errors. However, when it determines that a memory access hits in the cache, it is always correct. For example, a memory access m_1 which references a cache line X is determined to be a hit, if and only if, X is part of its state. Earlier (Detailed Warming), a memory access referenced

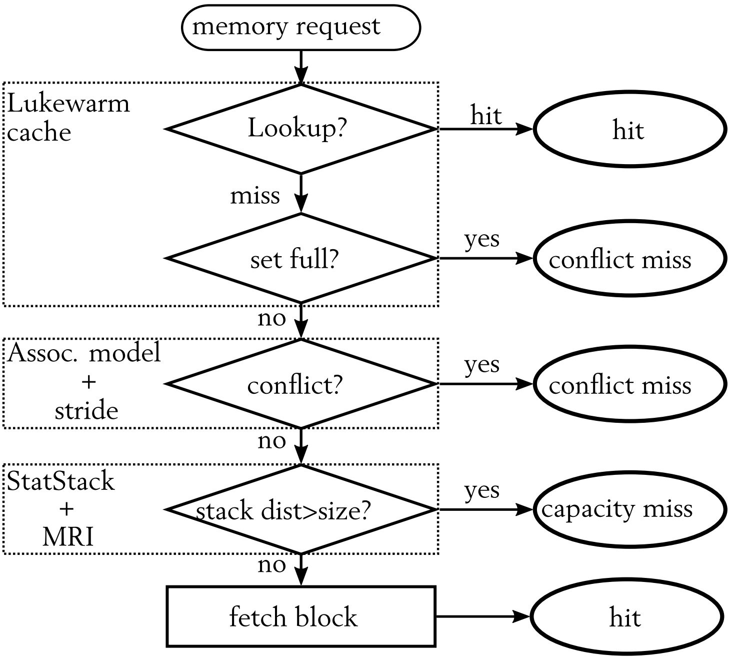


Figure 34: WarmSim's cache model. WarmSim leverages the non-blocking lukewarm cache. Accesses that hit in the lukewarm cache are modeled as cache hits; accesses that hit in the MSHR are modeled as delayed hits; and accesses that miss in the lukewarm cache are handled by the statistical cache model. WarmSim uses StatStack to determine capacity misses, and a limited associativity model to determine conflict misses. All other accesses that appear to be misses are due to insufficient warming and are treated as hits.

cache line X and now it is part of the state of the lukewarm cache. From a modeling perspective, the lukewarm cache will not have any false positives (false hits) and its specificity is 100%.

$$specificity = \frac{true\ negatives}{true\ negatives + false\ positives} \tag{3.1}$$

On the other hand, the lukewarm cache is not always accurate when it determines that a memory access misses in the cache. A cache miss can be either due to the design of the cache (e.g., capacity, associativity) or due to the lack of sufficient warm-up. Therefore, our simulation framework does not rely on the lukewarm cache when it determines a memory access as a cache miss. As a result, the sensitivity of the lukewarm cache is not guaranteed. The lukewarm cache can have many false negatives

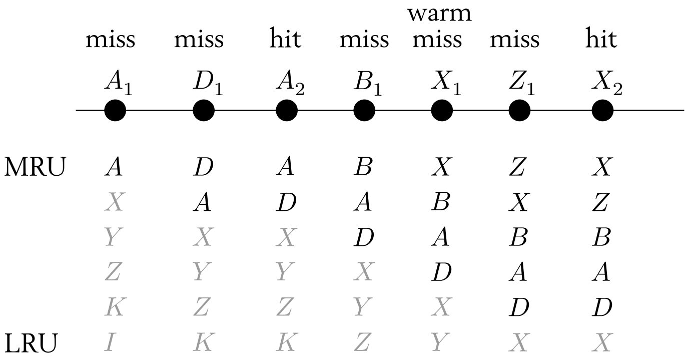


Figure 35: The lukewarm cache has not been sufficiently warmed up and part of its state is unknown (cache lines in gray). However, after a small number of accesses, the same cache lines are reused. These accesses hit in lukewarm cache. In this figure, accesses A_2 and X_2 find the cache line in the lukewarm cache. On the other hand, accesses A_1, D_1, B_1, X_1 and Z_1 miss the lukewarm cache. For the accesses that miss in the lukewarm cache, the statistical cache model has to determine if they are misses due to the limited capacity (capacity misses) due to the limited associativity (conflict misses); or due to the lack of warming (warming misses) in which case they should be modeled as hits.

(false misses).

$$sensitivity = \frac{true\ positives}{true\ positives + false\ negatives} \tag{3.2}$$

In computer architecture terms, an estimation of a cache hit (positive) is based on the presence of a cache line in the lukewarm cache and, consequently, it will always be correct (true). On the other hand, an estimation of a cache miss (negative) is based on the absence of a cache line from the lukewarm cache; this can be either due to insufficient warming (false miss) in which case an access to this cache line in the prior history (before the Detailed Warming) would have fetched the line to the cache; or due to the size and the associativity of the cache (true miss). An illustration of this is also shown in fig. 35.

WarmSim leverages the absence of false positives from the lukewarm cache. When a memory access hits in the lukewarm cache, WarmSim takes no further action. On the other hand, when a memory access misses in the lukewarm cache, WarmSim uses statistical cache modeling to determine if the memory access should be handled as a hit.

To distinguish between the cache misses that occur due to insufficient warm-up from the cache misses that normally occur during execution, we refer to the former as *warming misses*.

Memory accesses perform look-ups in the lukewarm cache. Hits are left unchanged; the simulator models their latency. In fact, the vast majority of memory accesses at the simulation point hit in the lukewarm cache. Our experiments show that for the SPEC CPU 2006 benchmarks, the hit ratio in the lukewarm data cache (64 KiB) is on average 88%. Memory accesses that miss in the lukewarm cache are further handled by WarmSim.

Leveraging the Non-Blocking Cache

Typically, caches can support multiple outstanding requests [2]. When a request misses in a non-blocking cache, a new Miss Status Holding Register (MSHR) is allocated. The MSHR tracks the outstanding request. Subsequent requests that reference the same cache line, when possible[1], will be serviced by the same response. Requests that reference different cache lines, allocate a new MSHR. When the cache receives a response, it services all requests that were tracked in the corresponding MSHR. In such caches, a request can a) hit in the cache, b) miss in the cache but hit in the MSHRs, or c) miss in the cache and miss in the MSHRs.

Due to the inherent spatial locality in most workloads, a significant fraction of memory accesses reference the same or different parts of the same cache line within a short period of time. A number of them will miss in the lukewarm cache while there is already an outstanding request for the same cache line. These accesses are typically referred to as MSHR hits or delayed hits. WarmSim leverages the MSHRs to model these accesses.

Memory accesses that miss in the cache but hit in the MSHRs, i.e., find an in-service MSHR for the requested cache line, are handled by the lukewarm cache. In practice, whenever this happens, WarmSim has already determined that an earlier memory request to the same cache line — the first request of the corresponding MSHR — was a cache miss and subsequent requests to the same cache line will be serviced by the same response.

Our experiments show that for the SPEC CPU 2006 benchmarks, using a 64 KiB data cache after 10,000 instructions, 95% of the requests either hit in the lukewarm data cache or hit in the MSHRs.

By leveraging the lukewarm cache and its MSHRs, WarmSim can quickly estimate the memory latency for a large number of the memory access (cache hit, delayed hit) without performing any statistical modeling. Using the lukewarm cache, WarmSim can readily determine a signif-

[1] A write request might need further coherence actions if the response services a read request.

icant fraction of the total memory accesses. However, for WarmSim to be accurate, it has to statistically model all other memory accesses, detect warming misses, and model them as hits.

Warming Misses

Normally, memory accesses miss in the cache primarily due to three reasons [2]:

- The limited capacity of the cache — capacity misses.
- The limited associativity of the cache — conflict misses.
- The first access to a cache line — compulsory misses.

In our simulation framework, we also expect warming misses; cache misses that are the result of insufficient cache warming. WarmSim's goal is to identify warming misses and model them as hits. For every warming miss identified, WarmSim fetches the referenced line in the cache and then uses the lukewarm cache to handle the latency of the request that now hits. At this point, the cache line is present in the lukewarm cache and potential future accesses to the same cache line will not require any statistical modeling.

Instead of trying to identify warming misses, WarmSim uses statistical cache modeling to identify accesses that are either capacity misses, conflict misses or compulsory misses. Any memory access that is not classified in any of these three categories is classified as a warming miss and it is handled as a hit.

Statistical Modeling: Capacity Misses

To estimate capacity misses, WarmSim uses statistical cache modeling. For any given memory access m, if we know its stack distance s_m, we can determine if it misses in a fully associative LRU cache due to its capacity. However, as we explained in section 2.2, stack distance collection is costly.

To avoid having to profile the application and obtain precise stack distances, we use StatStack. StatStack (section 2.2) can estimate the stack distance s_m using its reuse distance r_m and reuse distances (reuse distance histogram) h_v representative of the memory accesses in the vicinity (v_m) of m. This is a big improvement in the input requirements of our model. Prior work demonstrates efficient native techniques to obtain the reuse

[2] In modern multicore systems cache misses can occur for other reasons as well (e.g., coherence misses).

distance histogram h_v representative of v_m. The exact collection of reuse distance r_m, however, would still incur high profiling overhead[3]. Unfortunately, neither the collection of the stack distance s_m of the memory access m nor its estimation through its reuse distance r_m would fit the efficiency requirements for our sampled simulation framework.

Instead, WarmSim estimates the cache outcome of the memory access m using information about the associated memory instruction i. Often, memory instructions (e.g., inside a loop) are executed more than once. It is statistically more likely to sample instructions that execute repeatedly than sampling specific memory accesses. Conversely, memory instructions with significant contribution to the overall performance will be executed repeatedly. As a result, if we base our statistical analysis on the Program Counter (PC) of the memory instructions, rather than memory accesses, we are more likely to be accurate for memory instructions with significant impact on performance.

We use MRI that we obtain by profiling the workload up to the current simulation point to perform a PC-based stack distance analysis. For every instruction, we estimate its reuse distance histogram. Using the per-PC reuse distance histogram h_{pc} along with the complete reuse distance histogram h, we can use StatStack to estimate the per-PC cache miss ratio. Then, WarmSim predicts whether the memory access m hits in the cache. This prediction is based on the current miss ratio of instruction i based on previous predictions (including those made by the lukewarm cache) and the estimated miss ratio by StatStack. The prediction has to make the estimated miss ratio converge towards the estimated miss ratio.

Statistical Modeling: Conflict Misses

While capacity misses are typically the largest source of cache misses, some workloads have a significant number of conflict misses. To identify conflict misses, WarmSim leverages once more the lukewarm cache. When a memory access misses in the lukewarm cache and the MSHRs, WarmSim inspects the occupancy of the accessed cache set. If the cache set is full, it determines the memory access as a conflict miss. In this case, the information in the lukewarm cache is sufficient to determine the access as a conflict miss.

However, when simulating larger caches, conflicts misses can occur even when the corresponding set in lukewarm cache does not appear

[3] Berg and Hagersten [3] demonstrate an efficient random reuse distance sampler. Their profiling technology can efficiently obtain reuse information for random memory accesses and does not guarantee the collection of the reuse distance of a specific memory access.

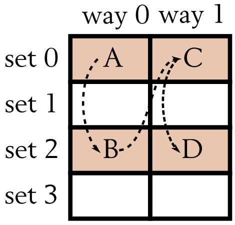


Figure 36: In the above example, a memory instruction repeatedly makes accesses to the cache with a stride (e.g., 128 B) which is twice as large as the size of the cache line. As a result, its working set size can only use half of the cache size.

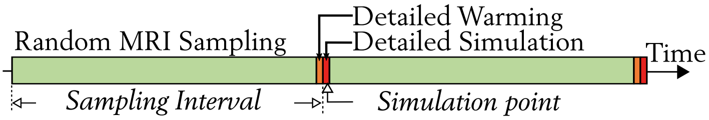


Figure 37: CoolSim

to be full. A typical source of associativity issues is memory instructions with problematic access patterns. As fig. 36 shows, such memory instructions end up using a small fraction of the sets in the cache due to patterns with very regular and large strides. For such memory instructions, WarmSim collects and uses their stride information to estimate the portion of the cache that is used. Assuming 64 B cache line, a memory instruction which performs memory accesses with large strides s, where s is a multiple of the cache line size, will access only a portion p of the cache ($p = s/64$).

This limited associativity model is used to determine the effective cache size that a memory instruction can use. The effective size is an input to StatStack and is used to estimate the per-PC miss ratio.

3.4 CoolSim: Virtualized Memory Reuse Profiling

CoolSim (Paper II) is a simulation framework that uses hardware virtualization and statistical cache modeling to eliminate cache warm-up and enable fast and accurate sampled simulation. CoolSim uses a novel profiler which uses hardware virtualization to collect MRI as the workload is fast-forwarded between simulation points (fig. 37).

MRI Sampling

CoolSim relies on hardware virtualization and VFF to make simulation more efficient. To leverage VFF and unleash its speed and flexibility potential, it obtains its input online, while the workload is fast-forwarded to the next simulation point. CoolSim's input is a sample of the workload's MRI.

As the workload executes using VFF, CoolSim's profiler samples random memory instructions. For any sampled memory instruction m_i that references a cache line c_i, CoolSim records: a) the reuse distance of the next memory access to c_i, and b) the memory stride when the same instruction m_i is executed again. In this section, we describe CoolSim's profiling technology.

Memory Instruction Sampling

To obtain a sparse but representative sample of a workload's MRI, the profiler selects random memory instructions. To achieve this efficiently, CoolSim programs a hardware performance counter to overflow after a given number of memory instructions. The overflow triggers an interrupt which stops the execution.

Unfortunately, our system does not provide an event that counts memory accesses. Instead, it provides events for counting the executed load and store instructions separately. While it is possible to combine the two events to one counter, we found that when we count the two events using the same counter, our processor counts the number of memory instructions that contain either a load or a store. Instructions which contain both loads and stores are accounted for as one event and introduce significant sampling errors (e.g., 462.libquantum). To acquire an unbiased random sample of memory accesses in our experimental setup, we use two separate counters and sample load and store instructions independently at the desired sampling rate.

Additionally, as we are executing in a virtualized system, the performance counters are carefully programmed to exclude events from the host system. As discussed in section 2.2, overflows are not precise and they are affected by the processor skid. As the skid is likely to bias the sampling process, the profiler compensates for it. To do this, it programs the counter overflow to stop a small number of loads/stores earlier. When the interrupt triggers, the profiler switches the execution to functional simulation to execute the remaining number of loads/stores until the desired memory instruction[4].

[4] KVM provides support for single-stepping, but the overhead of context switching from the virtualized system to the host system is prohibitively high.

Memory Reuse Distance Profiling

Once the virtualized simulation has stopped at a randomly selected memory instruction, the profiler uses performance counters to get the current load and store count. It then sets a watchpoint on the accessed cache line. The watchpoint allows the profiler to execute at full speed until the next access to the same cache line. Since most modern processors do not support large enough watchpoints to protect a whole cache line (64 B), CoolSim implements a software watchpoint mechanism that leverages the page protection mechanism.

To set a watchpoint on a given cache line, CoolSim protects the corresponding memory page against reads and writes. When the protected page is accessed, a page fault stops the virtualized simulation. If the triggering access does not reference the watched cache line, i.e., a false positive, the event is ignored, and execution is resumed. If the triggering access references the watched cache line, the profiler records its reuse distance using performance counters. Then, CoolSim restores the original page protection bits, and resumes the execution.

Memory Stride Profiling

When the virtualized simulation stops at a randomly selected memory instruction other than the watchpoint on the cache line, the profiler also sets a breakpoint on the current memory instruction. When the same instruction is executed again, a software interrupt stops the virtualized simulation. The profiler inspects the newly accessed memory location and records the memory stride between the two consecutive executions of the selected memory instruction. CoolSim uses the collected strides as input to the limited associativity model.

Cache Model

CoolSim, like WarmSim, leverages the lukewarm cache. Every request, first performs a lookup in the lukewarm cache. Memory accesses that hit in the lukewarm cache are modeled as cache hits; memory access that hit in the MSHRs are modeled as delayed hits; all other memory accesses that miss in the lukewarm cache and the MSHRs are handled by the cache model to determine whether they are warming misses. CoolSim models memory accesses that are warming misses as if they were cache hits. For any given warming miss, it instantly fetches the cache line from the memory, installs it in the lukewarm cache and serves the memory access as a cache hit.

CoolSim's statistical cache model is an optimized version of the statitstical cache model used in WarmSim. In the rest of this section, we describe the key optimizations we use in CoolSim.

Statistical Cache Model

CoolSim's statistical model uses PC-based stack distance histograms to determine capacity misses. Much like WarmSim, its input is MRI. However, CoolSim improves the accuracy of the cache model for sparser MRI (1/10000 memory instructions).

For every collected reuse distance r_m, we use StatStack to estimate the corresponding stack distance s_m. To estimate the stack distance s_m, StatStack takes as input the reuse distance r_m and the reuse distances of the memory accesses in the close vicinity v_m of memory access m. As our input is a sparse sample of the workload's reuse distances, the exact reuses of the vicinity v_m are not available.

To approximate the reuse distance distribution of v_m, WarmSim uses the complete set of collected reuse distances h. CoolSim, on the other hand, takes another approach in approximating the reuse distance distribution of v_m that improves accuracy given sparser input MRI.

CoolSim creates vicinities which all end at the current simulation point. Any given vicinity needs to be large enough to avoid statistical errors, while not being too large as to blend different phases together. The smallest vicinity will be big enough to have at least 50 reuse distance samples. For every reuse distance r_m, we compute an approximate vicinity v'_m that just about fits r_m. The approximate vicinity v'_m will be bigger than r_m but by no more than 20%. This allows CoolSim to make better use of StatStack and achieve higher accuracy at lower MRI sampling rates than WarmSim.

Progressive MRI Sampling

To accommodate the needs of the statistical model, CoolSim's MRI profiler uses progressively higher sampling rate as we get closer to the next simulation point.

CoolSim's cache model forms multiple vicinities which are as a good fit as possible for every collected reuse distance r_m. To allow for small vicinities which are statistically stable, i.e., have enough samples, the profiler needs to use high sampling rate. Larger vicinities, on the other hand, will have enough samples even at lower sampling rates. CoolSim's profiler progressively adjusts its sampling rate to benefit from the higher performance of sparse sampling far from the simulation point, and the higher statistical stability of dense sampling close to the simulation point.

All reuse distance samples are weighted according to their sampling probability (i.e., sampling rate) to ensure that they are equally represented in the overall sample and avoid any bias.

The profiling phase (fig. 37) is split in three parts which are sampled with a progressive sampling rate; the first part, 75 % of MRI sampling phase, is sampled at a low rate; the second part, 75 % of the remaining, is sampled at a high rate; and the remaining part is sampled at a higher sampling rate.

3.5 Delorean: Multi-pass Virtualized Memory Reuse Profiler

Delorean (Paper III) achieves almost an order of magnitude improvement in simulation speed. Delorean leverages hardware virtualization (VFF) to implement efficient directed profiling and obtains the MRI of the exact memory accesses that we need to model – the memory accesses that are executed in the Detailed Simulation and miss in the lukewarm cache. This provides a large efficiency improvement compared to CoolSim, which obtains MRI randomly. CoolSim's random MRI sample has to be dense enough to be representative and allow for accurate modeling of the memory accesses that miss in the lukewarm cache.

In this section, we present Delorean. First, we describe the directed profiler and, then, the cache model that is used in Detailed Simulation to estimate performance at each simulation point.

Directed MRI Profiling

Delorean's cache model leverages the lukewarm cache in the same way as WarmSim (section 3.3) and CoolSim (section 3.4). Consequently, its cache model needs to model statistically only the memory accesses that miss in the lukewarm cache and the MSHRs. We refer to these memory accesses as the *key accesses* and the cache lines they reference as *key cache lines*.

Delorean's goal is to statistically determine whether a key access is a warming miss, and instead should be modeled as a cache hit. To do that, it uses the obtained MRI. Contrary to WarmSim and CoolSim, that repeatedly sample random memory instructions to obtain MRI that will be representative of the MRI of the key accesess, Delorean's profiler takes a directed approach that aims at obtaining the exact MRI of the key accesses. However, while directed profiling allows for more accurate modeling of the key accesses, its cost with conventional profiling methods is

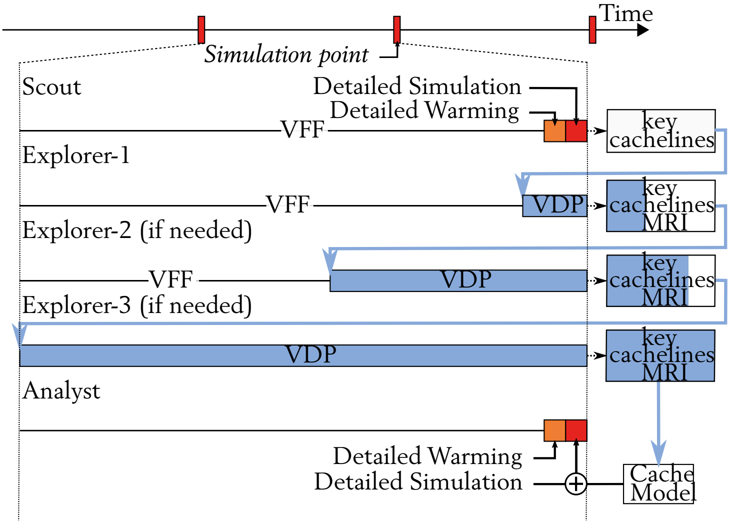


Figure 38: Delorean uses multiple passes. First, the Scout advances to the next simulation point, where it identifies the key memory accesses. Then, the Explorers collect memory reuses for key memory accesses. In the last pass, the Analyst uses the collected MRI with statistical cache modeling and detailed simulation to evaluate the performance at the simulation point.

prohibitively high. Delorean proposes a novel approach that allows for efficient directed MRI profiling.

To efficiently collect MRI for the key accesses, Delorean uses multiple passes. As fig. 38 illustrates, each of the passes uses a separate instance (process) of the same simulation[5].

We refer to the first pass as the *Scout*. The Scout identifies the key accesses. It first advances the execution to the next simulation point using VFF at near-native speed. At the simulation point, it switches to detailed simulation. The Scout monitors memory accesses to the lukewarm cache to identify the key accesses. Recall that key accesses are the memory accesses that miss in the lukewarm cache and its MSHRs. For each of the key accesses, the Scout records the PC, the memory address, the size, and the current instruction count. As we explained earlier, the number of key accesses is small. For the SPEC 2006 benchmarks and for a simulation point of 10,000 instructions, the average number of key accesses is 144.

Another instance, the *Explorer*, uses Virtualized Directed Profiling (VDP) to collect MRI. The Explorer obtains MRI for the key accesses

[5] To obtain multiple instances efficiently, one can either restore from the same checkpoint multiple times, or fork the simulator process to create identical processes.

and a sparse sample of random reuses representative of their vicinity.

To exploit the speed potential of the virtualized execution, the Explorer uses software watchpoints to detect accesses to key cache lines and advances the execution using hardware virtualization. When a memory access references a location protected by a watchpoint, an interrupt stops the virtualized execution. This memory access can be either an access to a key cache line, in which case it is recorded, or a false positive, i.e., an access to another cache line in the same memory page[6], in which case it is ignored. If the inspected access matches a key access, i.e., the exact address, PC, size, and instruction count, the Explorer records its reuse distance.

To achieve its efficiency, the Explorer needs to minimize the number of interrupts during VDP. Watchpoint-hits stop the virtualized simulation and slow down the profiler. While for each of the key accesses two watchpoint hits would suffice (one for the key access and one for the preceding access to the same cache line), many more are likely to occur. Many of the key cache lines are either frequently accessed, or belong to a frequently used page, i.e., are false positives, as the execution advances towards the next simulation point.

Many false positives occur before the Explorer encounters the reuse of the key access. More than 55% of the the key accesses have reuse distance which is smaller than 5 M instructions. As a result, the Explorer sets many watchpoints that aim at capturing small reuses, far from it and closer to the simulation point. When the distance between two consecutive simulation points is in the order of billions of instructions, these watchpoints are likely to fire unnecessarily and stop the execution many times before we find the reuse of the key access.

To avoid such stops and improve the performance of VDP, Delorean exploits the ability to fast-forward at near-native speed and uses multiple Explorers (Explorer-1, Explorer-2, etc.) that profile progressively further away from the simulation point.

Explorer-1 fast-forwards using VFF and switches to VDP, 5 M instructions before the simulation point. By the end of the simulation point, Explorer-1 has obtained MRI for the key accesses that were reused within its region, the last 5 M instructions. Explorer-1 will pass the rest of the key accesses to Explorer-2 which will search for longer reuses.

Explorer-2 has significantly fewer key accesses (about 45% or the original set) to profile. Explorer-2 profiles the last 50 M instructions and feeds the obtained MRI and the remaining key accesses to the next Explorer (Explorer-3). The same process is repeated until the whole set of key

[6] Recall that a software memory watchpoint uses page protection and causes page faults for any accesses to the 4 KiB page, further increasing the number of false positives.

accesses is covered. The optimal number of Explorers depends on the properties of the workload and the parameters of the sampled simulation.

This multi-pass approach allows Delorean to profile only as far back as it is needed, without profiling much further than what is needed and without profiling the same information twice. Also, it reduces the number of unnecessary watchpoint false positives and leverages hardware virtualization to profile MRI one order of magnitude faster than CoolSim.

Finally, the last pass, Explorer-n feeds the obtained MRI to the last instance, the *Analyst*. The Analyst uses detailed simulation and cache modeling with the obtained MRI to evaluate the simulation point without any prior cache warming.

Cache Model

The Analyst leverages the lukewarm cache in the same way as WarmSim and CoolSim. It uses statistical modeling only for accesses that miss in the lukewarm cache and the MSHRs (key accesses). The Analyst uses the same modeling technique to determine conflict misses as WarmSim. A major difference between Delorean's cache model (Analyst) and the cache models in WarmSim and CoolSim is how it determines whether a miss in the lukewarm cache should be modeled as a capacity miss.

Delorean's profiler obtains precise MRI that allows its cache model to estimate the exact stack distance s_m using StatStack. Contrary to our previous approaches that use PC-based analysis, Delorean can estimate the exact stack distance and determine whether the memory access should be modeled as a cache hit ($s_m \leq cache\ size$ [7]) or as a cache miss otherwise.

3.6 Results

In this section, we demonstrate Delorean's accuracy and performance, as it combines and extends the insights from WarmSim and CoolSim and shows the potential of the proposed frameworks. Simulation execution rates are measured on an Intel Xeon E5520.

Accuracy

Figure 39 shows the Cycles Per Instruction (CPI) and the estimation errors for the SPEC CPU2006 benchmarks. 456.hmmer-ref-nph3 and

[7] The cache size is measured in cache lines and might be adjusted by the associativity model.

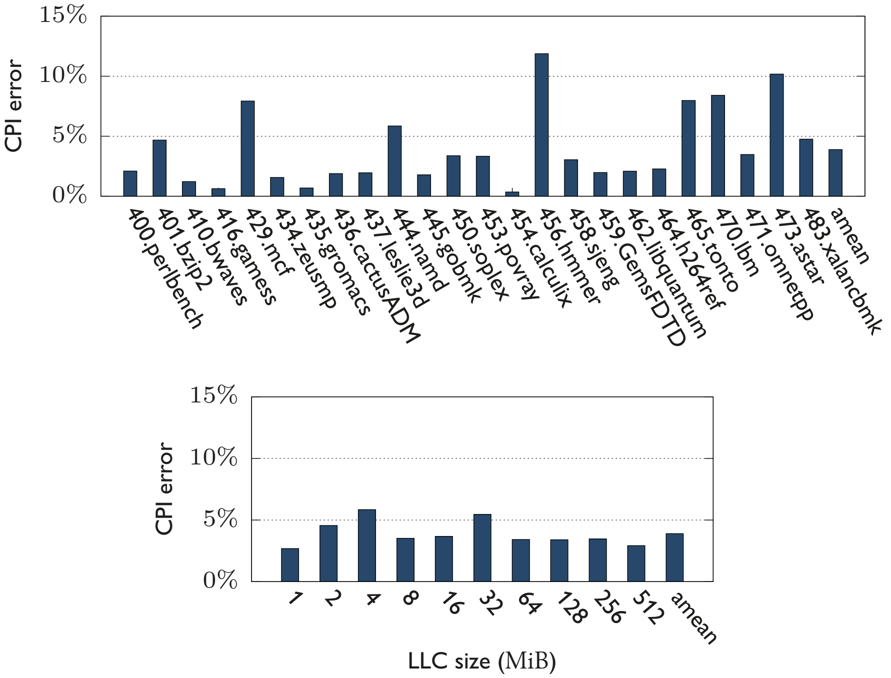


Figure 39: Average CPI error for the 27 SPEC CPU2006 benchmarks and cache sizes from 1 MiB to 512 MiB.

473.astar-BigLakes2048 show the largest average error. For 473.astar-BigLakes2048, the largest part of error comes from the L1D miss ratio. 473.astar-BigLakes2048 is sensitive to the L1D performance. However, the L1D is only 64 KiB where StatStack's statistical estimations show larger errors [11].

Delorean is accurate across the range of LLC cache sizes, with the exception of 429.mcf-ref and 401.bzip2-ref-chicken that show errors for smaller caches. Increasing the density of the vicinity reuses improves the accuracy for these two benchmarks as well as for 456.hmmer-ref-nph3 and brings their average error down from 2.09% to 1.09% for 401.bzip2-ref-chicken, from 7.9% to 4.67% for 429.mcf-ref, and from 11.8% to 2.7% for 456.hmmer-ref-nph3. Overall, the average error across all cache sizes for the SPEC CPU 2006 is 3.89%.

In Paper III, we show the CPI and miss ratio estimations as we vary the size of the LLC for the SPEC CPU2006 benchmarks. In Paper I and Paper II, we show similar results for WarmSim and CoolSim respectively.

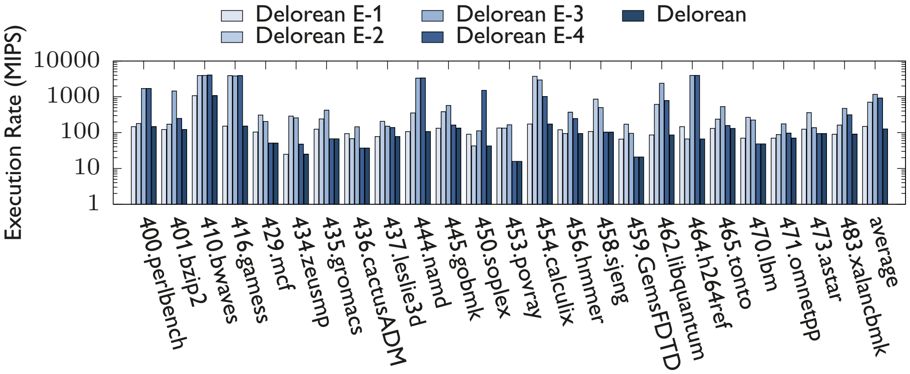


Figure 310: Performance for the different Explorers. For clarity, we omit the performance of the Scout and the Analyst, as they fast-forward between simulation points using and always perform better.

Performance

Delorean combines the insights from WarmSim and CoolSim and improves the efficiency of sampled simulation by an order of magnitude. The efficiency gains come mostly from the multiples passes of its MRI profiler. Figure 310 shows the performance of each of the Explorers. The overall performance for Delorean is determined by the slowest Explorer which runs on average at 148 MIPS. Some Explorers are not necessary; previous Explorers have already collected MRI for the full set of key accesses. For example, Explorer-3 and Explorer-4 run at full speed for 400.perlbench-ref-checkspam, as previously, Explorer-1 and Explorer-2 have already obtained the MRI for the key accesses. Overall, Delorean is 9× faster than CoolSim which runs on average at 17 MIPS.

In Paper III, we also show how the sampling rate of the vicinity MRI affects the accuracy and performance. The evaluation of the paper also shows the reuse distance distribution for the different SPEC CPU2006 benchmarks that led into some of the design choices of Delorean's MRI sampler.

3.7 Summary

Simulation is a very powerful tool in the hands of engineers and researchers who seek to understand performance. However, simulation performance often limits its use to small workloads or very abstract models of the hardware. Simulation overhead is dependent on the performance of the simulator as a model, as well as the size of the workload.

Evaluating large, realistic workloads using detailed microarchitectural models can take months (or even years) worth of CPU time.

Sampled simulation is an effective approach to lower the simulation overhead. Prior sampled simulation frameworks lower the overhead of simulation by an order of magnitude. While much faster than detailed simulation, sampled simulation is four orders of magnitude slower than native execution. The main overhead of sampled simulation is associated with the warm up of the microarchitectural state of the processor. Typically, caches have the largest state and are responsible for the largest part of the warm-up cost.

In this thesis, we use statistical cache modeling to eliminate cache warm-up and speedup sampled simulation by almost two orders of magnitude, compared to sampled simulation that uses functional simulation for cache warm-up. Statistical cache models do not require warm-up and instead rely on sparse MRI.

WarmSim shows that statistical cache modeling can be used in sampled simulation to eliminate the need for cache warm up. WarmSim amends simulator's cache to allow for accurate modeling even in the absence of complete state (lukewarm cache).

CoolSim builds on WarmSim and shows that the input to the statistical cache model can be obtained with low overhead. CoolSim obtains a random MRI sample while the workload is fast-forwarded to the next simulation point. CoolSim's profiler is built on top of virtualized fast-forwarding and uses native techniques to obtain MRI.

Delorean leverages the speed of VFF even further. As its speed is close to native, it uses multiple low-overhead passes to optimize the profiler and speed up the MRI collection process even further. As Delorean obtains directed rather than random MRI, its statistical cache model is significantly simpler and more accurate, compared to WarmSim and CoolSim.

4 Memory Performance Characterization using Native Techniques

Application performance is typically dependent on the processor's memory resources. For example, the size of the cache and the off-chip memory bandwidth determine the latency of memory accesses to fetch data and instructions. In multicore processors, often, some of the memory resources (e.g., shared caches) are shared between all cores.

Private resources such as the processor cores or the private L1 data cache can be studied in isolation. For example, an application with a 6 MiB working set will benefit from a private 8 MiB. The application's working set will fit in the cache and, typically, its cache miss ratio will be low.

However, shared resources, such as the last-level cache or the main memory in multicore systems, can be used by more than one application at the same time. Two applications running on different cores on a multicore processor will have isolated access to the private resources but will compete for the shared cache. An application with an 6 MiB working set might not fit in an 8 MiB shared cache. Its cache allocation will largely depend on the performance characteristics of the other application that competes for the same shared cache [30].

While modeling private resources is not always easy, modeling shared resources is often more challenging due to the complex interactions of the different threads. In this chapter, we propose modeling techniques that allow us to analyze workloads that run on the same multicore processor and compete for shared resources.

In section 4.1, we propose Cache Pirating (Paper IV) to model the contention for the shared cache. Most modern multicore systems feature a large last-level cache. Its large capacity can greatly benefit applications with good memory locality, but it can have very low to no benefit for applications with working sets that do not fit in it. Cache Pirating allows us to model such applications and quantify the impact of the shared cache.

In section 4.2, we propose the Bandwidth Bandit. The Bandwidth Bandit (Paper V) models off-chip memory bandwidth, a shared resource that impacts the performance of memory accesses that miss in the cache. The Bandwidth Bandit quantifies the impact of off-chip memory sharing on application performance.

Finally, in section 4.3, we combine the insights from the Cache Pirate and the Bandwidth Bandit to understand the overall performance impact of shared resources on applications that run on multicore systems. To quantify and correctly attribute the performance impact of shared resources, we demonstrate a method to measure speed-up stacks (Paper VI).

4.1 Cache Pirate: Modeling Cache Contention

Cache Pirating (Paper IV) quantifies the impact of the shared cache on application performance. It is a native modeling method and as such, measurements are performed on real hardware. All performance estimations include the effects of all microarchitectural characteristics of the processor (e.g., hardware prefetchers, cache replacement policies, etc.).

Cache Pirating controls how much of the shared cache capacity an application (the Target) can use. While we control its shared cache allocation, the Target runs on an otherwise unmodified system. Using the hardware PMU, we measure the Target's performance when it gets different allocations in the shared cache. Ultimately, we estimate how a performance metric (e.g., Instruction Per Cycle (IPC)) will change as the Target's allocation in the shared cache changes.

What allows us to control the available shared cache capacity for the Target is to run it concurrently with another application, the Pirate, that occupies a known part of the shared cache. The Pirate retains its working set in the shared cache, stealing cache capacity from the Target. This reduces the shared cache capacity available to the Target, which can only use the remainder of the shared cache.

In the next section, we explain how the Pirate works and how we use it to steal shared cache capacity. The accuracy of Cache Pirating is largely warrantied by the fact that we perform measurements on real hardware. But our measurements need to capture and isolate the performance impact of the contention for the shared cache. As we use the Pirate to create and control the contention for the shared cache, the accuracy of our measurements depend on the Pirate's ability to steal shared cache capacity. To this end, we also demonstrate how we determine that the Pirate was successful at stealing shared cache capacity for the duration of the measurements.

The Pirate

The Pirate creates and controls the desired contention for the shared cache and allows us to quantify the performance impact of shared cache contention. To achieve this, the Pirate has to be successful at stealing the specified amount of shared cache capacity, and avoid creating contention for any other resource that will impact the Target's performance.

The Pirate is an application that performs memory accesses to create contention for the shared cache. We run it on a separate core to minimize its interference with the Target. The Target has exclusive access to some resources such as the core and the private caches (data cache, instruction cache, and the unified L2 cache). On the other hard, the Pirate and the Target share the L3 (shared) cache and the off-chip memory bandwidth. The Pirate has to control the amount of contention in the shared cache, without creating any contention for the off-chip memory.

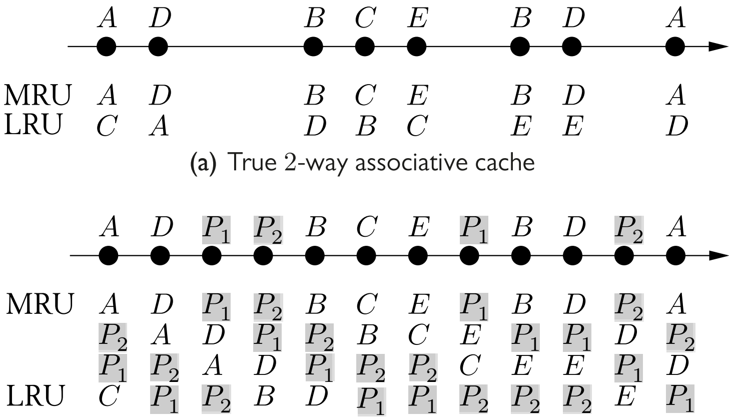
To create contention, the Pirate performs memory accesses, installs its working set in the shared cache and then competes with the Target to retain its working set resident in the L3 cache. To install and retain its working set in the L3, the Pirate needs to iterate repeatedly through its working set. Its accesses need to effectively "bypass" the higher level caches (L1 and L2). In our system, the L2 is exclusive of the L1 and therefore configuring the Pirate's working set size to be larger than the sum of the size of the L1 cache and the size of the L2 cache is sufficient.

To be able to study how different amounts of contention for the shared cache affect the Target's performance, the Pirate needs to be successful at retaining both smaller and larger working sets in the shared cache.

How successful the Pirate is at stealing cache depends on the Target and how much it fights back. When the Pirate retains its working set in the shared cache, all its memory accesses hit in the shared cache. This is a key observation and, in fact, allows us to monitor when the Pirate is successful. When the Pirate retains its working set in the cache, all its memory accesses are satisfied by the shared cache and its traffic to the off-chip memory is zero[1]. We use the hardware PMU to monitor the Pirate's off-chip memory bandwidth. When its bandwidth consumption is (close to) zero, all its accesses are satisfied by the shared cache and therefore its working set is resident in the cache. This feedback allows us to determine when measurements are accurate and representative of a system with the specified shared cache contention.

When the Pirate is successful at retaining its working set in the shared cache, it also satisfies the second requirement; the Pirate does not con-

[1] The memory bandwidth includes hardware prefetching.


A D B C E B D A
MRU A D B C E B D A
LRU C A D B C E E D
A D P1 P2 B C E P1 B D P2 A
MRU A D P1 P2 B C E P1 B D P2 A
P2 A D P1 P2 B C E P1 P1 D P2
P1 P2 A D P1 P2 P2 C E E P1 D
LRU C P1 P2 B D P1 P1 P2 P2 P2 E P1

(a) True 2-way associative cache

(b) 4-way associative cache with the Pirate stealing 1 way

Figure 41: Evolution of the LRU stack of a true 2-way associative cache vs. a 4-way associative cache with the Pirate (P) stealing two ways. The contents and relative ordering of the remaining two ways in the 4-way cache.

sume any off-chip memory bandwidth. As described earlier, the Pirate is an application that creates contention for the shared cache without affecting the off-chip memory bandwidth. As a result, the Target has exclusive access to the off-chip memory.

In summary, the Pirate is an application that runs on its own core to avoid creating contention for resources private to the Target's core(s). To create contention for the shared cache, the Pirate performs memory accesses to install and retain its working set resident in the shared cache. When the Pirate is successful at creating the specified contention, it does not consume any off-chip memory bandwidth. In the next section, we look into the memory access pattern, that allows the Pirate to be highly successful at retaining its working set in the shared cache.

Stealing Capacity in an LRU Cache

In the following paragraphs, we describe the memory access pattern of the Pirate for caches with LRU replacement policy. Cache blocks in a set-associative, LRU cache are organized in sets. Each set has a number (equal to the associativity of the cache) of blocks. A set works like a stack where blocks are ordered based on their age. At the top, we find the Most Recently Used (MRU) block and at the bottom the LRU block. When replacements occur, the block at the bottom (LRU) is evicted and when a cache block is reused, it gets promoted to the top of the stack (MRU) pushing the rest of the blocks by one position down. Figure 41(a) shows how the stack of a 2-way associative cache evolves over time with the access pattern shown at the top. The stack maintains an age ordering of its cache blocks, with the MRU cache block at the top of the stack.

Figure 41(b) shows how the stack evolves for a 4-way associative cache when the Target is running concurrently with the Pirate. In this example, the Pirate is configured to steal two cache blocks per set, thereby leaving two cache blocks in the set to be used by the Target. To avoid having its cache block evicted, the Pirate should access it again, such that it stays as close to the top of the stack as possible. When the Pirate steals more than one cache block per set, the most effective way to achieve this is to always access the "oldest" cache block. As long as the Pirate does this at a high rate, its cache block will not be evicted and its entire working set will stay resident in the cache. Such an access pattern is easily constructed by accessing the first word of successive elements in an array, with an element size equal to the cache block size, and a total size equal to the desired working set. By using this access pattern, we can maximize the Pirate's ability to keep a large working set in the cache.

In summary, the most effective access pattern to achieve the objectives listed above, is a simple linear access pattern. As long as the Pirate's access rate is high enough, it will retain its entire working set in the cache. However, the more cache blocks the Pirate tries to steal, the higher its access rate has to be. Otherwise its blocks can be evicted by the Target. Fortunately, processors features, such as out of order execution and hardware prefetchers, allow the Pirate to achieve a high L3 access rate.

Results

Figure 42 shows the shared cache miss ratio and fetch ratio[2], the bandwidth and performance (CPI) for two SPEC CPU2006 benchmarks as a function of shared cache capacity. All measurements are performed while the Pirate steals the desired amount of shared cache.

The miss ratio for 482.sphinx3 increases significantly as the Pirate steals more shared cache and its allocation in the shared cache reduces. At the same time, as its working set does not fit in the shared cache, it performs more accesses to the off-chip memory. Consequently, its bandwidth consumption increases. This has a significant impact on its performance as its CPI increases by almost 50%.

The miss ratio for 470.lbm seems to be constant across the whole range of shared cache allocations. However, there is a significant increase in the fetch ratio. The prefetcher generates more traffic as the allocation

[2] Fetch ratio is the total number of fetches into the cache including cache blocks fetched by the prefetcher over the total number of accesses.

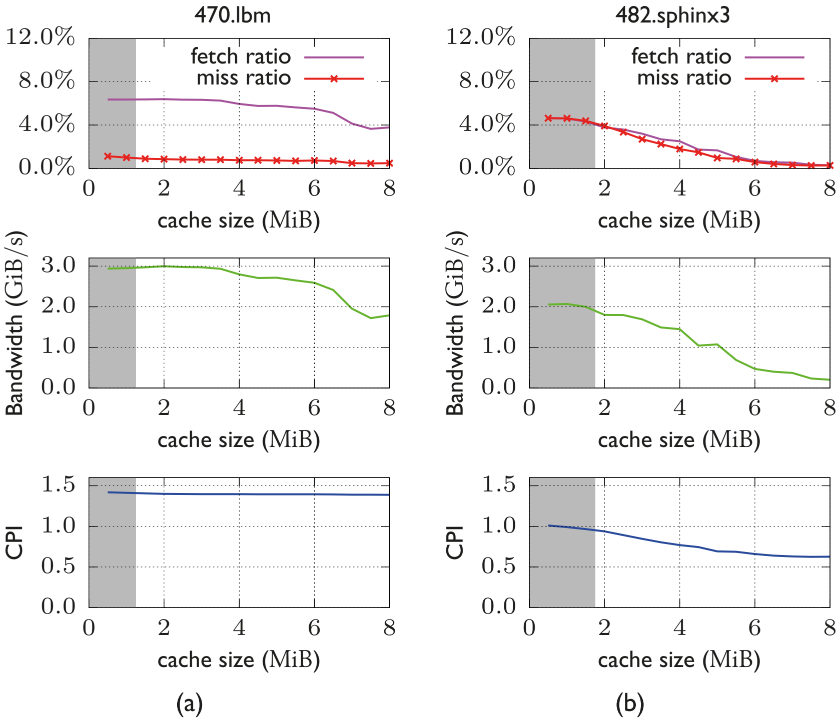


Figure 42: fetch/miss ratios, bandwidth and CPI for 470.lbm **(a)** and 482.sphinx3 **(b)** captured using Cache Pirating. The gray area shows the Target's shared cache capacities for which the Pirate is unable to retain its working set in the shared cache.

for 470.lbm in the shared cache is reduced. The overall performance for 470.lbm is largely unaffected by the allocation in the shared cache due to the prefetcher. The prefetcher seems to be able to hide the latency to fetch from the off-chip memory and makes up for the reduced cache capacity at the expense of higher off-chip memory bandwidth.

4.2 Bandwidth Bandit: Modeling Off-chip Memory Contention

The Bandwidth Bandit (Paper V) allows us to quantify the impact of off-chip memory contention on application performance. The main idea is to run an application (Target) concurrently with a *Bandit* application that generates contention for the shared off-chip memory resources. To accomplish this, the Bandit accesses memory at a specified rate and in a controlled pattern that ensures it generates the desired amount and type of contention. By measuring the Target's performance while varying the amount of contention the Bandit generates, we measure how the Target's performance changes when there is contention for the memory system.

Similarly to Cache Pirating, we want to isolate the performance impact of off-chip memory contention. The Bandit has to create contention for the off-chip memory only, leaving any other resource shared with the Target unaffected. In particular, the Bandit must avoid using the shared cache or any other resource used by the Target. Otherwise, the measured performance impact will include the contention for resources other than the off-chip memory.

There are two different memory performance characteristics that can impact the Target's performance:

a) the *Memory Latency*, as it determines how fast memory accesses can fetch data from the off-chip memory, and

b) the *Memory Bandwidth*, as it determines the maximum number of outstanding requests that can be serviced by the off-chip memory in parallel.

The Bandit has to create different amounts and types of contention for the off-chip memory, and allow us to quantify the application sensitivity to changes in the off-chip memory latency and bandwidth. To better understand the off-chip memory contention that can occur, and the underlying reasons, we will first go through the organization and the basic performance characteristics of typical off-chip memories.

Off-chip Memory Organization

The off-chip memory in today's computers uses DRAM. DRAMs are typically organized in modules. An off-chip memory controller communicates with the DRAM modules over a memory channel. For increased performance, memory controllers might use more than one memory channel.

A DRAM module is organized in several independent memory banks, which can be accessed in parallel when there is no conflict on the address and data buses. The combination of independent memory channels and memory banks allows for a large degree of parallelism in the off-chip memory.

A DRAM bank is organized into rows and columns. To address data, the memory controller has to specify a channel, a bank, a row and a column. To read or write an address, the whole row is first copied into the bank's row buffer (precharge). This single-entry buffer (also known as a page cache) caches the row until a different row in the same bank is accessed. Every request is serviced from the row buffer. When the row buffer already contains the accessed DRAM row, the access is serviced faster. In fact, for every request that is serviced from a DRAM bank, there are three different scenarios:

a) *page-hit*: the accessed row is already in the row buffer and the data can be read/written directly;

b) *page-miss*: the accessed row is different from the one cached in the row buffer. First, the cached row is written back to the memory bank. Then, the newly accessed row is fetched into the row buffer;

c) *page-empty*: the row buffer is empty. The accessed row is copied from the bank before it can be read/written. The memory controller preemptively closes a page that has not been accessed recently to optimistically turn a page-miss into a page-empty.

The latency to service a request to the specified bank depends on its row-buffer locality. A page-hit has the shortest latency, and a page-miss has the longest. Additionally, low overall row-buffer locality increases the service latency of the requests that arrive to the memory controller and limits the sustained throughput. As a result, low row-buffer locality can limit the off-chip memory bandwidth.

The Bandit

To generate realistic contention and allow us to measure reliably the performance impact of off-chip memory contention, the Bandit must be able to create both contention that leads to increased off-chip memory latency and contention that results in reduced off-chip memory bandwidth.

To generate a given amount of memory contention, the Bandit has to be able to generate a number of parallel memory accesses that access a set of banks at different rates. The Bandit has to be able to vary its effect on the row-buffer locality of the Target. To accomplish this, we need a mechanism to control exactly which DRAM bank is accessed to service any given memory request.

In this section, we describe the design of the Bandit. For simplicity, we will reason about our design choices on our experimental setup. The Bandit, however, can be configured to create the desired memory contention on any other system as well. Our system is based on an Intel Xeon E5520 processor. The processor has three memory channels and is populated with $DDR3-1333$ Dual In-line Memory Modules (DIMMs). A DIMM has 16 banks and a row within a bank is 8 KiB.

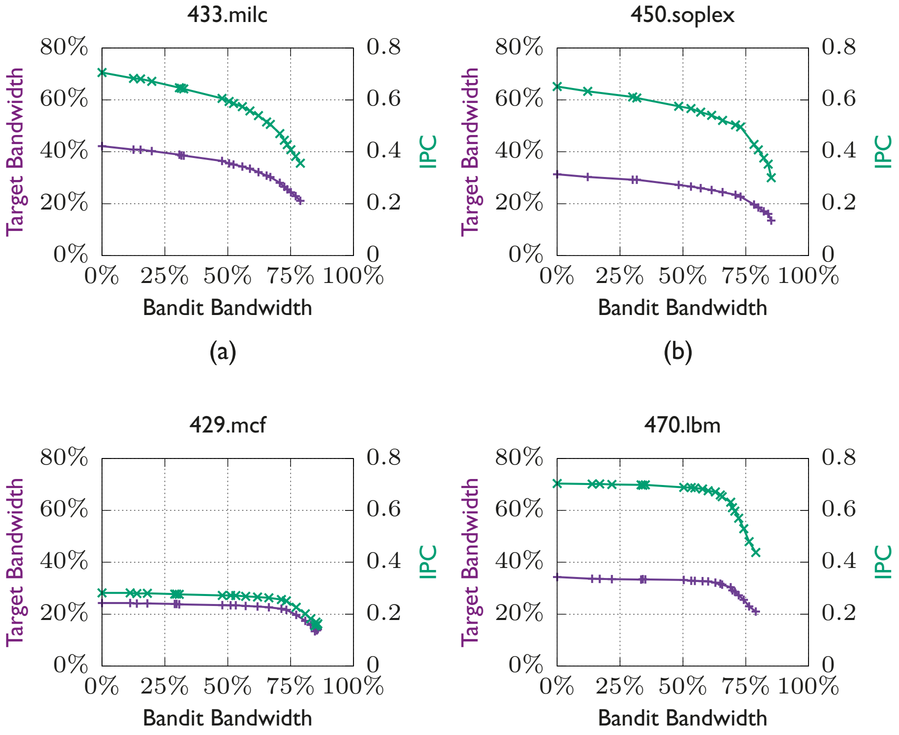
To further simplify our description, we explain how the Bandit works when only one channel is populated. In Paper V, we explain how the Bandit works for systems with more than one memory channel.

The Bandit needs to perform memory accesses that create the desired contention in the DRAM banks. In our setup, an 8 KiB memory region which is aligned[3] and continuous in physical address space maps into a single row. Contiguous 8 KiB memory regions map into the same row of different banks. Since there are 16 banks, an aligned and continuous memory region of 128 KiB maps in the same row of all 16 banks.

The Bandit takes advantage of this mapping to precisely control the contention for DRAM banks. It allocates its working set using large pages (2 MiB). These 2 MiB allocations are continuous both in virtual and physical address space. As 128 KiB are sufficient to access all DRAM banks, the Bandit can be configured to create the desired bank contention.

Apart from precise control of the banks that the Bandit accesses, it also controls precisely the rate at which it performs memory accesses. To achieve that, its working set is organized in linked lists. As it traverses a linked list, its accesses are serialized. To increase the access rate, it traverses multiple linked lists at the same time. For example, we can configure the Bandit to traverse one linked list only where its memory access rate is limited to one outstanding memory request at the same time, or alternatively, we can configure the Bandit to traverse three linked lists where up to three outstanding memory requests can be serviced in parallel.

The Bandit runs on its own core and it has isolated access to its private L1 and L2 caches. However, it shares the shared last-level cache. To avoid creating contention for the shared cache, the Bandit exploits the shared cache organization to minimize its allocation and force all its memory requests to fetch data from the off-chip memory. To achieve 100% miss ratio without trashing the shared cache, the nodes of the linked lists map into a small number of cache set. Consequently, only a small number of sets are trashed. To ensure that all accesses are misses, the nodes in the linked list do not fit in a single set. For example, in an 8-way set associative last-level cache, the Bandit traverses through at least nine distinct nodes in the linked list that map to the same cache set. This way, the Bandit minimizes its use of the shared cache and avoids creating contention other than for the off-chip memory.



(a) (b) (c) (d)

Figure 43: The impact of off-chip memory contention on bandwidth and performance for four SPEC CPU 2006 benchmarks. Bandwidth measurements are presented as a fraction of the maximum total bandwidth achieved in each experiment. At the top, 433.milc (a) and 450.soplex (b) show behavior of latency sensitive applications, as their performance degrades long before the off-chip memory bandwidth is saturated. At the bottom, 429.mcf (c) and 470.lbm (d) appear to be bandwidth sensitive applications. Their performance remains largely unaffected until the sum of the bandit bandwidth and the target bandwidth saturate the system's bandwidth.

Results: Bandwidth and Latency Sensitivity

Figure 43 shows the impact on bandwidth and performance of the contention for the off-chip memory on four SPEC CPU 2006 benchmarks. The off-chip memory contention is created using the Bandit and is shown as a fraction of the system off-chip memory bandwidth that we measured for each Target. All four applications experience performance degradation when the contention for the off-chip memory increases. However, the slowdown for the four applications exhibits different characteristics.

433.milc (fig. 43(a)) and 450.soplex (fig. 43(b)), experience slowdown even when the off-chip memory bandwidth of the Bandit is very low and the total bandwidth (Target and Bandit) is far lower than the maximum total bandwidth. For example, when 433.milc runs on its own, it consumes 2.8 GiB/s. When we run it with the Bandwidth Bandit, the maximum total bandwidth consumption is 6.5 GiB/s. Therefore 433.milc requires only 22% of the maximum off-chip memory bandwidth to achieve its baseline performance. However, when we run it concurrently with the Bandit, its performance degrades long before the total bandwidth reaches its maximum value. As the Bandit increases its memory access rate, it has negative impact on the DRAM bank row locality for 433.milc. As the DRAM bank row locality degrades, the off-chip memory latency increases. The processor is unable to hide the latency increase which impacts the performance of 433.milc. In other words, 433.milc, 450.soplex are latency sensitive.

On the other hand, 429.mcf and 470.lbm are largely unaffected by the Bandit's off-chip memory contention. It is only when the total off-chip memory bandwidth saturates that their performance degrades. At this point, they see a linear decrease in their performance. 429.mcf and 470.lbm are not as sensitive to latency increases and their performance is dependent on their ability to sustain their off-chip memory bandwidth. In other words, 429.mcf, 470.lbm are bandwidth sensitive.

4.3 Speedup Stacks: Quantifying the Scalability Bottlenecks

In this section, we describe a novel method (Paper VI) that leverages the insights from the Cache Pirate and the Bandwidth Bandit, to obtain speedup stacks and quantify the scalability bottlenecks for multi-threaded, data-parallel applications. We leave out the effects of the non-parallelizable phases on multi-threaded performance and instead focus

[3] On an 8 KiB granularity.

only on the parallel phases[4]. However, including these effects in the speedup stack would be a simple exercise.

For clarity, we lump together the scalability bottlenecks into three categories [15]; contention for shared cache, contention for off-chip memory, and synchronization, which includes lock contention, barrier synchronization and load imbalance.

The speedup stack [15] shows the speedup that a parallel application would achieve, if it was run on a hypothetical machine where we can gradually get rid of the individual scalability bottlenecks, but which is otherwise identical to a real, commodity multicore. This hypothetical machine can be thought of as follows; To eliminate the bottlenecks due to synchronization, all threads should perform as if they never had to wait for the other threads. For example, in the case of lock contention, each thread should perform as if the other threads never held the lock in question. To eliminate the scalability bottlenecks due to contention for shared resources (shared cache and off-chip memory), all threads should perform as if they had exclusive access to the relevant shared resource.

As an example, fig. 44 shows the execution of an application's parallel phase on such a hypothetical machine with N cores in four different scenarios. In the top scenario, the hypothetical machine has no scalability bottlenecks, and we achieve perfect (linear) speedup. For each successive scenario, a new scalability bottleneck is introduced. In the bottom scenario, all scalability bottlenecks are present, and the parallel phase therefore achieves the same speedup as it would on the real machine. As the execution time of the parallel phase increases in each successive scenario, the speedup decreases (gets worse).

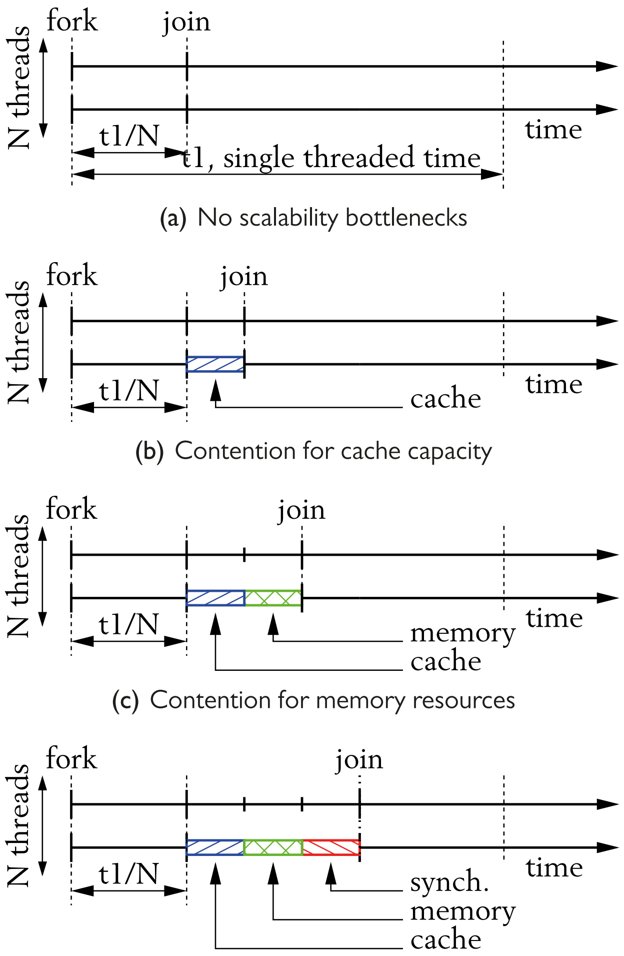
Figure 45(a) shows the speedup curves achieved in the four scenarios. Figure 45(b) shows the corresponding speedup stack. Each component in the speedup stack shows the difference between the performance before and after the corresponding bottleneck was introduced.

In this section, we introduce a method for obtaining the speedup stacks for multiple concurrently running instances of single-threaded applications (section 4.3). Then, we extend this method, first, to multi-threaded applications without synchronization (section 4.3) and, finally, to multi-threaded applications with synchronization (section 4.3).

Single-Threaded Applications

First, we describe how we obtain speedup stacks for multiple concurrently running instances of single-threaded applications. As each instance

[4] For a fork-join parallel program, a parallel phase occurs between a fork and the corresponding join.



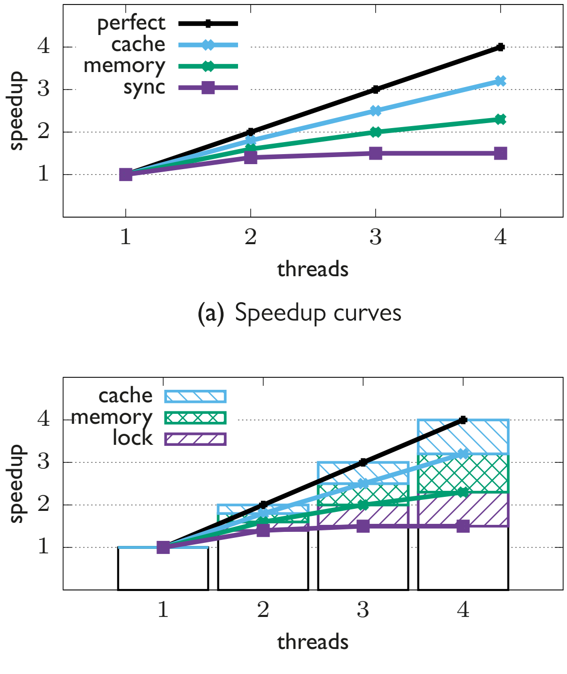
(a) No scalability bottlenecks

(b) Contention for cache capacity

(c) Contention for memory resources

(d) Synchronization overhead

Figure 44: The execution of a parallel phase in four scenarios. For each scenario, a new scalability bottleneck is added to the hypothetical machine.



(a) Speedup curves

(b) Speedup stack

Figure 45: Speedup curves (a) and the corresponding speedup stack (b).

performs the same amount of work no matter how many we run concurrently, we look at their aggregate throughput rather than speedup. Furthermore, as they have separate address spaces and do not synchronize, the only two scalability bottlenecks that can limit throughput are contention for *cache capacity* and contention for the *off-chip memory*. Their speedup stacks will therefore have only two components.

We show how to obtain the speedup stack for one, two and four instances of 403.gcc that run concurrently. To obtain the components of the speedup stack, we have to estimate the performance of the 403.gcc instances in a) a system with contention for the shared cache and the off-chip memory, b) a system with contention for the shared cache but no contention for the off-chip memory, and c) a system without contention for the shared cache or the off-chip memory.

Obtaining performance in a system with contention for the shared cache and the off-chip memory is simple. We run one, two and four instances of 403.gcc in our quad core system. The curve labeled "measured" in fig. 46(a) shows the normalized throughput of 403.gcc on our quad core system. It is immediately obvious that 403.gcc does not achieve

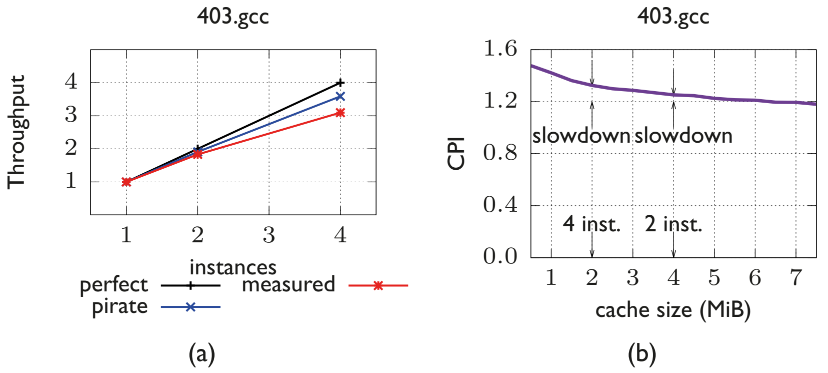


Figure 46: Speedup graph **(a)** and CPI as a function of cache size **(b)** for 403.gcc.

perfect (linear) scaling.

To obtain performance in a system with contention for the shared cache but no contention for the off-chip memory, we use Cache Pirating. Each of the concurrently running instances executes the same code and as result, it has the same demand for shared cache capacity as every other instance. Consequently, each instance receives an equal amount of the shared cache capacity. When we run one instance, it gets up to 8 MiB; when we run two instances, each one gets up to 4 MiB; when we run four instances, each gets up to 2 MiB of the shared cache capacity. The Cache Pirate creates the desired contention in the shared cache and allows us to estimate the performance of 403.gcc as a function of its allocation in the shared cache (fig. 46(b)). In these experiments, 403.gcc has exclusive access to the off-chip memory. This allows us to estimate the CPI of each of the concurrently running instances using the data in fig. 46(b) and then compute the respective aggregate throughput. The curve labeled "pirate" in fig. 46(a) shows the resulting normalized throughput.

Obtaining performance in a system without any contention for the shared cache or the off-chip memory is also simple. Without any contention, the performance of each of the instances of 403.gcc is the same as if we run only one instance. The normalized throughput in such a system would be perfect. The curve labeled "perfect" in fig. 46(a) shows the resulting normalized throughput.

When we run several instances concurrently, they share the off-chip memory resources which can cause significant slowdown. This is the case for 403.gcc. The difference between the curves labeled "perfect" and "pirate" in fig. 46(a), shows the slowdown due to contention for cache capacity; while the difference between the curves labeled "pirate" and "measured" in fig. 46(a), shows the slowdown due to contention for off-

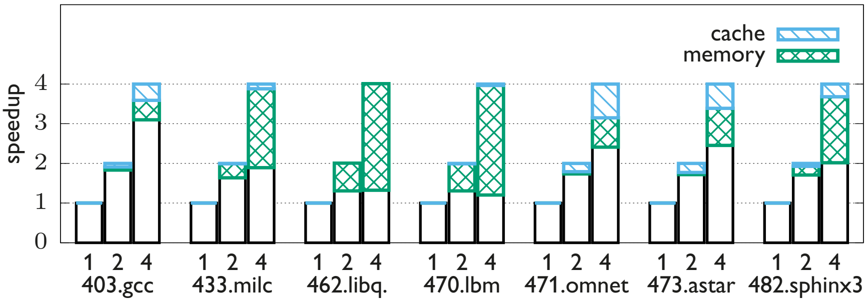


Figure 47: Speedup stacks of seven SPEC CPU2006 benchmarks.

chip memory resources. Figure 47 shows the speedup stacks of 403.gcc when running on our quad core system. Ultimately, fig. 47 shows that the relative throughput of four concurrently running instances can be improved by about 0.4, if we address the cache bottleneck, and by about 0.5, if we address the memory bottleneck.

Multi-threaded Applications

In this section, we show how to obtain speedup stacks for symmetric, fork-join style parallel applications, where all threads execute the same code and have no data sharing. The main bottlenecks that can limit the speedup of such applications can be illustrated in a speedup stack with the following three components; contention for cache capacity, contention for off-chip memory resources, and synchronization, which includes lock contention, barrier synchronization and load imbalance.

First, we show how to obtain speedup stacks for multi-threaded applications without synchronization. Then, we extend this method to handle applications that use synchronization operations.

Multi-threaded Applications without Synchronization

As before, to obtain speedup stacks, we have to emulate a system where we can gradually get rid of the contention for shared resources. Our approach to emulate such a system is similar to the one we used for multiple instances of the same single-threaded application. To obtain performance when there is contention for the shared cache and the off-chip memory ("measured" curve), we run all instances on our quad core system; to obtain performance when there is contention only for the shared cache ("pirate" curve), we use Cache Pirating; finally, performance when there is no contention for the shared resources is same as when we run one isolated instance and its speedup is perfect ("perfect" curve).

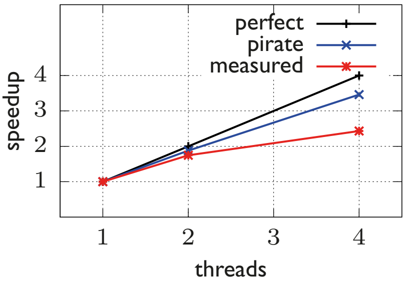


Figure 48: Speedup graph of IS.

System without Off-chip Memory Contention: To emulate a system with contention for the shared cache only, we run one of the threads alone while we halt the other threads. This allows us to profile that thread using Cache Pirating and build the speedup stacks for a multi-threaded application without synchronization in much the same way as we did for single-threaded applications.

First, we start the Target application with a given number of threads. When it reaches the fork-point of the first parallel phase of interest, we stop all threads. We start the Pirate and let it warm up its working set. We let all but one thread run until the join-point of the parallel phase. As with single-threaded applications, using the Pirate, we can measure the performance of the profiled thread in a system where we have created the desired contention for the shared cache but there is no contention for the off-chip memory.

To take control over the threads without any source code modifications, we preload a dynamic library into the Target application's address space. Our dynamic library overloads `pthread_create`, `pthread_join` and `pthread_barrier_wait`. Preloading the library when launching the Target allows us to take control of the threads, start and stop them at the fork, join and barriers as described above. When intercepting calls to `pthread_create` at a fork-point, the library attaches a set of performance counters to the profiled thread. When there is a call to `pthread_join`, we obtain measurements using the hardware PMU. We also carefully monitor the Pirate's fetch ratio to ensure that it steals the desired amount of cache and discard the measurements when its memory traffic is significant.

Figure 48 shows three speedup curves for the most significant parallel phases of IS (integer sort). The ones labeled "perfect" and "measured" report the perfect (linear) speedup and the speedup measured on our quad core system, respectively. To estimate the "pirate" speedup curve,

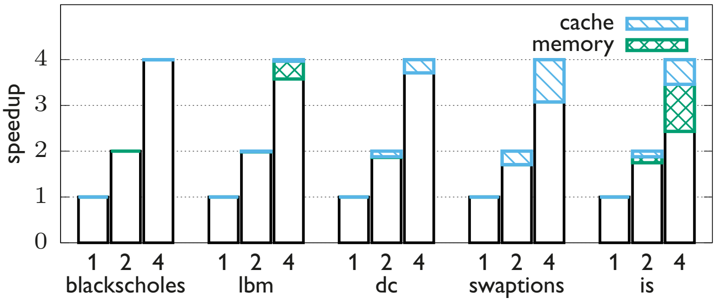


Figure 49: Speedup stacks for five multi-threaded applications without synchronization operations.

we launched IS under the control of the profiling framework presented above with one, two and four threads, and measured the IPC of one thread while giving it access to 8 MiB, 4 MiB and 2 MiB of the shared cache. We only need to profile one thread, since all threads run the same code, they will have approximately the same IPC. Based on this IPC, we then estimate IS's "pirate" speedup curve in the same way as we did for 403.gcc.

Figure 49 shows the speedup stack of the most significant parallel phase of IS (on the far right). Its cache and memory components are computed as the differences between the speedup curves labeled "perfect" and "pirate"; and "pirate" and "measured", respectively.

Multi-threaded Applications with Synchronization

In this section, we extend the method for multi-threaded applications with synchronization. For these applications, the speedup stacks have one more component that accounts for the cost of synchronization. This component shows the slowdown due to lock contention, barrier synchronization and load imbalance.

As before, to obtain speedup stacks, we have to emulate a system where we can gradually get rid of the contention for shared resources. To obtain performance when there is contention for the shared cache and the off-chip memory and lock contention ("measured" curve), we run all instances on our quad core system. To obtain performance when there is contention for the shared cache and the off-chip memory ("sync" curve), we simply measure the time each thread spends waiting for access to critical sections protected by locks, and the time waiting at barriers when the program is running on the real machine. We then use these measurements to estimate the speedup that would be achieved in a scenario where synchronization is for free. To obtain performance when there is contention only for the shared cache (“pirate” curve), we use Cache Pirating. Finally, performance when there is no contention for the shared resources is the same as when we run one isolated instance and its speedup is perfect (“perfect” curve).

Estimating the “measured” and “perfect” curves is trivial, and we use the same approach as above. The rest of this section provides an overview of the method used to estimate the “sync” and “pirate” curves.

System without Lock Contention: Instead of devising an experiment where there is no overhead for synchronization, we run one, two and four threads of the application in an unmodified system and measure the overall performance as well as the time we spend in synchronization operations (e.g., mutexes and barriers). Then, we estimate the performance of the application when there is no contention for synchronization by subtracting the cost of synchronization from the base execution time.

First, for parallel applications without barriers, we measure the execution time (t) for each thread and the time it spends spinning or sleeping to acquire locks (t_l). If the threads did not have to wait to acquire locks, their execution times would be equal to $t - t_l$. However, the execution time of the parallel phase is equal to that of the slowest thread. Therefore, for parallel phases without barriers, the execution time when lock synchronization is for free equals $t - t_l$ of the slowest thread, i.e., the one with the largest $t - t_l$.

Second, for parallel phases with barriers, the synchronization overhead consists of two components: overhead incurred due to lock synchronization and barrier synchronization. While our simple profiling method can be easily extended to provide the information needed to distinguish between the two components, we merge them into a single component for the purpose of building speedup stacks. This allows us to estimate the synchronization component of the speedup stack, simply by measuring the total time each thread spends waiting for both locks and barriers (t_{l+b}), and subtract it from the threads' measured execution times (t). As above, the execution time of the parallel phase is equal to that of the slowest thread, in this case the one with the largest $t - t_{l+b}$.

To measure the time a thread is spinning, waiting for access to a lock, we overload the pthread function that is used to operate on mutexes, spinlocks and barriers. The `pthread_mutex_lock` is overloaded with a function that reads the TSC, using the `rdtscp` instruction, both before (tsc_1) and after (tsc_2) it calls the real `pthread_mutex_lock`. The time spent waiting for access to the lock is then $tsc_2 - tsc_1$. Calls to `pthread_spin_lock` and `pthread_barrier_wait` can be handled in the same way. By accumulating these differences for each thread individually, we get the total time they spend waiting for access to locks.

To account for load imbalance, we treat the join operation at the end

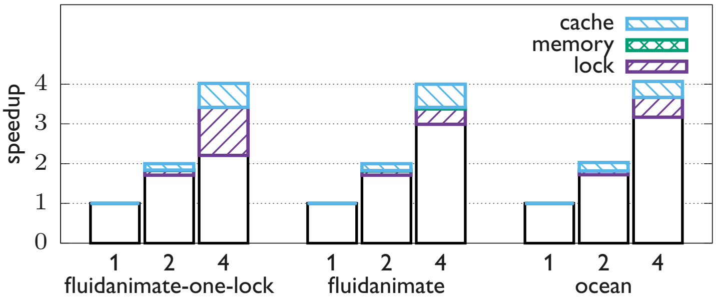


Figure 410: Speedup stacks for three multi-threaded programs with synchronization operations.

of the parallel phase as a barrier. This allows us to handle lock and barrier synchronization; and load imbalance in a uniform way. Our speedup stacks therefore report one component, *synchronization*, which includes all synchronization overheads.

System without Lock and Off-chip Memory Contention: To emulate a system without off-chip memory contention, we use the same method as for multi-threaded applications without locks. By running only one thread a time, we make sure that there is no contention for the off-chip memory. At the same time, the Pirate creates the desired contention for the shared cache. The running thread will find all locks available and therefore will not see any contention for locks.

If the application uses barriers, we repeat the process in between barriers. At the start, we let only one thread run and measure its performance until it reaches a barrier. We record its performance and let the other threads catch up. When all threads reach the barrier, we repeat the same process. We let the profiled thread run and measure its performance until the next barrier. This process is repeated until the end of the parallel phase.

Our goal is to measure the performance of the profiled thread while the pirate creates the desired shared cache contention and there is no contention for any other resource. As we alternate between phases where only the profiled threads run, and phases where all other threads catch up, we need to make sure that the profiled thread can retain its data in the caches. To achieve this and avoid trashing the caches, we run all the non-profiled threads in a separate processor. As the applications we are dealing with have no data sharing, the threads running in the other socket will not affect the profiled thread.

Figure 410 shows speedup stacks for three data parallel benchmarks. We present data for two versions of fluidanimate. Fluidanimate iteratively computes over a set of particles in a three dimensional space which is split into sub-spaces, one for each thread. In a single iteration, several different properties are computed for each particle, between which the threads synchronize using a single barrier. In the original version of fluidanimate (from PARSEC [5]), each particle has its own lock. When a thread computes on a particle residing on a boundary of its sub-space, it has to grab the lock associated with the particle's neighbors. To introduce more lock contention, we created a version of fluidanimate that uses a single lock (fluidanimate-one-lock).

Figure 410 shows that both fluidanimate versions have large lock components. As expected, the lock component in fluidanimate-one-lock version is larger than fluidanimate. The cache component is slightly smaller for fluidanimate. As there is more lock contention in fluidanimate-one-lock, the threads spend more time spinning waiting for access to the lock. During this time, they do not utilize the shared cache capacity, which allows the non-spinning threads access to more cache capacity. This causes fluidanimate-one-lock to have a slightly smaller cache component than fluidanimate.

Finally, we note that all three benchmarks in fig. 410 have virtually zero memory components. This is expected, as their memory bandwidth demands at four threads are low (1.8 GiB/s, 1.9 GiB/s, and 1.7 GB/s, respectively) and can be easily satisfied. For our method to predict a memory component of zero, the speedups estimated using Cache Pirating and the method to estimate synchronization overheads, presented above, must add up. This is indeed the case, and indicates that both methods make accurate estimations.

4.4 Summary

Modern multicore processors feature multiple cores that share part of the memory hierarchy. Often, there is at least one shared cache and all cores use the same memory controller(s) to access the off-chip memory. The shared cache and the off-chip memory are shared resources that determine how application performance scales.

In this chapter, we presented the Cache Pirate and the Bandwidth Bandit. Two native modeling techniques that quantify the impact of contention for the shared cache and the off-chip memory on application performance. As native techniques, they have low overhead and all measurements include the effects of all microarchitectural features.

Finally, the insights from the Cache Pirate and the Bandwidth Bandit allow us to quantify the scalabitity bottlenecks for multiple instances of a single threaded application and data-parallel, symmetric multi-threaded applications. We used Speedup Stacks to attribute the scalability bottlenecks to the shared cache, the off-chip memory and synchronization.

5 Summary

Performance in today's computer processors depends on a number of complex architectural and microarchitectural design decisions. Typically, computer architects rely on performance modeling to make design decisions and improve next generation processors. For example, prior to implementing a new feature, architects try to assess its performance impact. Similarly, when making decisions about the size of a structure (e.g., cache), architects have to determine whether the performance benefit justifies the extra cost.

With the breakdown of Dennard scaling [9, 34] microarchitectural improvements become more important than ever before. At the same time, as Moore's law [22] slows down, microarchitectural optimizations become increasingly more constrained.

Systematic approaches to model and evaluate computer performance can provide many insights for optimizations in processor design. To this end, performance modeling has been the focus of a lot of prior research [37, 28, 7, 31, 11, 3, 19, 17, 18, 23, 32, 35].

This thesis builds on much of this prior research and makes contributions with performance models that quantify the impact of the memory system in applications' performance. These contributions follow primarily two different commonly used approaches to performance modeling. WarmSim (Paper I), CoolSim (Paper II) and Delorean (paper III) propose to use statistical cache modeling to improve simulation performance, while The Cache Pirate (Paper IV), the Bandwidth Bandit (Paper V) and Speedup Stacks (Paper VI) quantify the impact of shared resources on application performance with modeling techniques that run on native hardware.

Combining Statistical Cache Modeling with Simulation

Simulation is commonly used by architects to evaluate the performance characteristics of a computer processor without building a prototype. The fidelity of the simulator can be as close to the actual design as needed. For example, a simulator can be bit-accurate (its output matches the exact output of the actual processor) or cycle-accurate (the simulator models the processor on a cycle granularity). However, higher accuracy typically results in higher -simulation overhead.

To speed up simulation, prior research [37, 33, 31] uses sampled simulation to reduce large workloads into small representative simulation points. Using sampled simulation, the amount of detailed simulation is limited to about 1% of the total workload. As a result, sampled simulation can significantly speed up performance evaluations.

Practical approaches to sampled simulation either first restore from architectural checkpoints, then use functional simulation to warm up the microarchitectural state and detailed simulation to evaluate the simulation point, or restore from microarchitectural checkpoints [35, 36] and use detailed simulation to evaluate the simulation point.

More recent approaches use virtualization (Paper VII) to fast-forward the execution between simulation points at near-native speed, functional simulation to warm up the microarchitectural state and then detailed simulation to evaluate the simulation point. All approaches advance the state of the art in simulation methodology, but trade flexibility for lower simulation overhead.

The first three papers in thesis propose efficient simulation methods without limiting flexibility. WarmSim proposes a statistical cache model that eliminates cache warm-up, typically, the most costly part of the microarchitectural warm-up. Statistical cache modeling relies on sparse MRI. In contrast to prior approaches to sampled simulation, WarmSim's performance does not degrade when simulating processors with larger caches. In fact, WarmSim uses the same MRI to simulate systems with different cache sizes, and its performance remains constant, even when simulating very large caches (e.g.,512 MiB). WarmSim shows that statistical cache modeling can eliminate the need for cache warm-up in sampled simulation.

CoolSim proposes an efficient MRI profiler that leverages VFF (Paper VII). CoolSim improves WarmSim's statistical cache modeling to build a practical and integrated simulation framework. As a result, it can accurately simulate at a rate of 17 MIPS, 19 × faster than sampled simulation that uses functional warming between simulation points.

Delorean proposes a multi-pass approach to improve the efficiency of the MRI collection process. This approach allows for targeted MRI collection, which has lower overhead than CoolSim's random approach to MRI collection. In fact, the targeted MRI allows Delorean to use a significantly simpler statistical cache model than the one used by CoolSim. Delorean's integrated sampled simulation framework simulates at a rate of 148 MIPS, 9× faster than CoolSim and 150× faster than sampled simulation that uses function simulation between simulation points.

Memory Performance Characterization using Native Techniques

Performance models that rely on data obtained from native execution are typically much faster than simulation. While simulators require validation to ensure their accuracy, native modeling is by construction representative of the given hardware. While native methods do not always allow for accurate modeling of future processor designs, insights on the performance bottlenecks in existing processors enable architects to better focus their optimization goals for future designs. Furthermore, software developers can use native modeling to optimize operating systems, run-time systems, applications, etc.

This thesis proposes three novel methods to quantify the impact of shared resources on application performance using data collected on native execution. Many modern multicore systems have a last-level cache and one or multiple memory controllers to the off-chip memory which are shared across all cores. Understanding contention for these shared resources is key to efficiently utilizing multicore processors.

The Cache Pirate (Paper IV) proposes a method to analyze the impact of cache sharing on application performance. The Pirate is a separate application that can effectively control the amount of shared cache capacity it uses. The target application is run concurrently with the Pirate, which limits the available shared cache capacity. This allows us to measure different performance metrics for the target application using hardware performance counters while we control the amount of shared cache available using the Pirate.

The Bandwidth Bandit (Paper V) proposes a method to quantify the effect of sharing the off-chip memory resources on application performance. Similarly to the Pirate, a Bandit application creates and controls the off-chip memory contention in the system. A target application is run concurrently with the Bandit, which allows us to measure performance under any given contention for the off-chip memory.

Finally, the Speed-up Stacks (Paper VI) combine the insights of the Cache Pirate and the Bandwidth Bandit to provide an integrated method for understanding the scalability bottlenecks of symmetric parallel workloads with no data sharing. Using the Speed-up Stacks, we can quantify the performance loss and attribute it to cache contention, off-chip memory contention and synchronization.

6 Svensk Sammanfattning

Prestanda i moderna processorer beror av ett antal komplexa designbeslut som påverkar både arkitektur och mikroarkitekur. Vanligtvis använder processorarkitekter prestandamodellering som stöd för sådana designbeslut. Till exempel så använder arkitekter ofta modellering för att bedöma hur en ny funktion påverkar prestanda innan den byggs. När storleken av en komponent (t ex en cache) ska avgöras, måste en eventuell prestandaförbättring vägas mot de kostnader som beslutet innebär.

När Dennard-skalning inte längre gäller [9, 34] blir mikroarkitektursförbättringar viktigare än någonsin tidigare. Kombinerat med att processförbättringarna som har följt Moores lag [22] mattas av blir de möjliga optimeringarna av mikroarkitekturen alltmer begränsade.

Systematiska metoder för att modellera och utvärdera datorprestanda kan ge värdefull information om hur en processor kan konstrueras. Prestandamodellering har därför varit fokus för omfattande forskning [37, 28, 7, 31, 11, 3, 19, 17, 18, 23, 32, 35].

Denna avhandling bygger på mycket av denna tidigare forskning och bidrar med prestandamodeller som kvantifierar minnessystemets inverkan på applikationsprestanda. Dessa bidrag följer i första hand två olika metoder för prestandamodellering. WarmSim (Paper I), CoolSim (Paper II), och Delorean (Paper III) beskriver hur man använder en statistisk cache-modell för att förbättra simuleringsprestanda. Cache Pirate (Paper IV), Bandwidth Bandit (Paper V), och Speedup Stacks (Paper VI) beskriver hur man kvantifierar hur delade resurser påverkar prestanda med hjälp av modelleringstekniker som bygger på existerande hårdvara.

Kombinerad Statistisk Cachemodellering och Simulering

Datorarkitekter använder ofta simulerade modeller för att utvärdera prestanda hos en ny processor utan att behöva bygga en prototyp av processorn. Simulatorn kan ge resultat med så stor noggrannhet som ett designbeslut kräver. Exempelvis kan en simulator producera exakta resultat på bit-nivå (resultatet matchar exakt hur den faktiska hårdvaran skulle bete sig) eller en cykel-baserad approximation (simulatorn modellerar hårdvaran på cykelnivå). En högre noggrannhet resulterar emellertid oftast i en högre simuleringskostnad. En detaljerade simulator som gem5 [6] kan simulera ett system med cirka 100 kIPS, ca 5 storleksordningar långsammare än hårdvaran simulatorn körs på. Vid denna hastighet kan en simulering av verkliga applikationer ta över ett år

För att öka simuleringshastigheten använder tidigare forskning [37, 33, 31] samplad simulering som reducerar stora applikationer till små representativa simuleringspunkter. Med hjälp av samplad simulering kan behovet av detaljerat simulering begränsas till cirka 1% av den totala applikationen. Samplad simulering kan alltså ge resultat avsevärt snabbare än traditionell simulering.

Metoder för samplad simulering kan utgå från ett begränsat sparat systemtillstånd (precis det som behövs för att återstarta applikationen) och använda en snabb typ av simulator som värmer upp vissa viktiga hårdvarustrukturer för att sedan använda en detaljerad simulator för att utvärdera simuleringspunkten. Alternativt kan man utgå från mer detaljerat sparat tillstånd (tillståndet i applikationen och viktiga komponenter som påverkar prestandan) [35, 36] för gå direkt till den detaljerade simulatorn för att utvärdera simuleringspunkten.

Nyare metoder använder hårdvaruvirtualisering (Paper VII) för att snabbspola exekveringen mellan simuleringspunkter, snabb simulering för att värma upp viktiga hårdvarustrukturer, och sedan detaljerad simulering för att utvärdera simuleringspunkten. Alla dessa metoder har drivit forskningsfronten framåt, men offrar flexibilitet för lägre simuleringskostnader.

I avhandlingens tre första artiklar beskrivs effektiva simuleringsmetoder som inte begränsar flexibiliteten. WarmSim beskriver en statistisk cache-modell som eliminerar cache-värmning, vanligtvis den dyraste delen i uppvärmningsfasen. Statistisk cache-modellering är beroende av minnesprofiler som effektivt kan mätas på existerande hårdvara [3]. I motsats till tidigare metoder för samplad simulering försämras inte WarmSims prestanda när man simulerar processorer med större cacher. I själva verket använder WarmSim samma minnesprofil för att simulera system med olika cache-storlekar. Detta gör att dess prestanda förblir konstant även när man simulerar mycket stora cacher (t ex 512 MiB). WarmSim visar att statistisk cache-modellering kan eliminera behovet av cache-värmning i samplad simulering.

CoolSim är en effektiv metod för att ta fram minnesprofiler genom att använda hårdvaruvirtualisering (Paper VII). CoolSim förbättrar WarmSims statistiska cache modeller för att bygga en integrerad simuleringsmiljö. Detta gör att CoolSim kan exekvera simulerade applikationer med en hastighet på 17 MIPS. Detta är 19 × snabbare än samplad simulering som använder traditionell värmning mellan simuleringspunkterna.

Delorean är en flerstegsmetod för att göra minnesprofileringen ännu effektivare. Metoden möjliggör riktade sampling som har lägre kostnader än CoolSims slumpmässiga sampling. Faktum är att den riktade samplingen gör det möjligt att använda en betydligt enklare statistisk cache-modell än den som används av CoolSim. Deloreans integrerade simuleringsmiljö kan exekvera en simulerade applikation med en hastighet av 148 MIPS, 9x snabbare än CoolSim och 150x snabbare än samplad simulering som använder traditionell värmning mellan simuleringspunkter.

Karakterisering av Minnesprestanda med Existerande Hårdvara

Prestandamodeller som bygger på data från existerande hårdvara är vanligtvis mycket snabbare än simulering. Simulatorer kräver validering för att säkerställa deras noggrannhet medans mätningar på existerande hårdvara per definition representerar den givna hårdvaran. Mätningar på existerande hårdvara kan inte alltid förutsäga hur framtida hårdvara kommer att bete sig, men det kan ge insikter om flaskhalsarna som påverkar prestanda i dessa processorer. Kännedom om dessa flaskhalsar gör att de kan optimeras i framtida designer. Dessutom kan mjukvaruutvecklare använda denna typ av modellering för att optimera till exempel operativsystem, exekveringsmiljöer, och applikationer.

I denna avhandling beskrivs tre nya metoder för att kvantifiera effekten av delade resurser på applikationsprestanda med hjälp av data som samlats in på existerande hårdvara. Många av dagens flerkärniga datorsystem har en delad cache och en eller flera minnes-interface som delas av alla kärnor. Att förstå hur olika kärnor konkurerar om dessa delade resurser är nyckeln till effektiv användning av flerkärniga processorer.

Cache Pirate (Paper IV) är en metod för att analysera hur applikationsprestanda påverkas av cache-delning. Piraten är en separat applikation som kan styra mängden delad cache den använder. Applikationen som ska profileras körs tillsammans med Piraten som begränsar mängden tillgängliga cache. Eftersom Piraten kontrollerar hur mycket cache som finns tillgängligt kan en applikations beteende mätas som en funktion av tillgänglig cache.

Bandwidth Bandit (Paper V) är en metod för att kvantifiera effekten av delad minnesbandbredd. Funktionen hos Banditen påminner om Piraten, men tillskillnad från Pirat-applikationen begränsar Banditen mängden tillgänglig minnesbandbredd i systemet. När en applikation körs tillsammans med Banditen kan vi mäta prestanda för olika mängd tillgänglig bandbredd.

Speed-up Stacks (Paper VI) kombinerar Cache Pirate och Bandwidth Bandit till en integrerad metod som gör det möjligt att förstå flaskhalsar symmetriska, parallella, applikationer som saknar datadelning. Med hjälp av Speed-up Stacks kan vi mäta prestandakostnaden för delning av cache, delning av minnesbandbredd, och synkronisering.

7 Acknowledgments

This thesis is the end of a long journey. From the start, I have shared every step of the way with people who have inspired me and supported me until I have finally reached the port of destination. I am grateful to each and every one of you!

First and foremost, I would like to thank my advisor Erik Hagersten, who guided me through this journey, taught me how to steer and slowly handed over the rudder to me as I was reaching the destination. Our brainstorming on statistical models on the whiteboard has been fun and inspiring. Thank you Erik! I would also like to thank my co-advisor Konstantinos (Kostis) Sagonas for his support, when I needed it the most. Kostis introduced me to the post graduate program at Uppsala University and helped me choose my research topic at the start of this journey.

I would like to thank my co-authors for all the contributions and stimulating discussions. My research would have been a lot less fruitful without you. David Eklov, for helping me in my first steps; Andreas Sandberg, my friend and colleague. Andreas' technical expertise has been both an inspiration and an invaluable contribution in many of my research projects. Muneeb Khan, who started his PhD studies at same time as me and with whom we shared our experiences as junior researchers. I would also like to thank Trevor E. Carlson, who taught me a lot about writing and how to present my ideas. Stefanos Kaxiras, for his invaluable contributions to many cool ideas.

The PhD students from other groups in the department, who inspired me to think outside the box and see the "bigger picture". My friend, colleague and fellow snowboarder Jonatan (Tant) Linden, who introduced to the Swedish culture; Pan Xiaoyue for all the fun discussions; Mikael (Lax) Laaksoharju for all the philosophical discussions and Jonathan Cederberg for all the political discussions!

All the students of UART 3.0, Vasilis Spiliopoulos, Konstantinos Koukos, Moncef Mechri, Ricardo Alves and German Ceballos. You have all been such a refreshing addition to the group. As well as the newer members of the group Mehdi Alipour, Kim-Anh Tran, Gustaf Borgstrom and Johan Janzen. All the senior members of the group, Alexandra Jimborean, Magnus Sjalander and Alberto Ros, your feedback in many presentations

has been invaluable. It has been a great privilege working with all of you!

A large part of this work was finished while I was with Arm Research and many of my colleagues there have also contributed to this. Their support and belief in me has helped me refine existing ideas from a different perspective. I want to thank especially my great colleagues Stephan Diestelhort, Andreas Sandberg (yes again!), Ilias Vougioukas and Radhika Jagtap.

The research presented in this thesis has been generously funded and supported by the UPMARC research center. Computational infrastructure for the countless hours of the reference simulations was provided by the Swedish National Infrastructure for Computing (SNIC) at the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). Also invaluable to my curiosity about computers and later to my research has been the open-source software and open-access research.

Computers and research is a big part of my life, but without my family and friends I would have never been able to get here. First and foremost, and above all, I would like to thank my parents and my brother. I am very grateful for your endless support.

I would like to thank my friend, Anastassios (Tassos) Nanos for the countless things he taught me and for the many opportunities he has offered me from the very start of my career. My friend Dimitris (paliofilos) Bampalikis for all the supportive beers and much more. The Kendo crew for all the fights and the trips. The climbing crew for all the fun climbs. My friends in Uppsala, most of them PhD students and researchers in very different fields. My great friends in Greece, every time I met with you, it was like I never left home.

Last but not least, my partner Sara, for all her love. Sara never stopped believing in me and supporting me in every possible way, until the very end.

8 References

[1] Ricardo Alves, Nikos Nikoleris, Stefanos Kaxiras, and David Black-Schaffer. “Addressing Energy Challenges in Filter Caches”. In: *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Nov. 2017, pp. 49–56. DOI: 10.1109/SBAC-PAD.2017.14.

[2] Samson Belayneh and David R. Kaeli. “A Discussion on Non-blocking/Lockup-free Caches”. In: *ACM SIGARCH Computer Architecture News* 24.3 (June 1996), pp. 18–25. DOI: 10.1145/381718.381727.

[3] Erik Berg and Erik Hagersten. “StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. ISPASS ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 20–27.

[4] Kristof Beyls and Erik D’Hollander. “Discovery of Locality-Improving Refactorings by Reuse Path Analysis”. In: *Proc. International Conference on High Performance Computing and Communications (HPCC)*. HPCC ’06. Munich, Germany: Springer-Verlag, 2006, pp. 220–229. DOI: 10.1007/11847366_23.

[5] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton, NJ, USA, 2011.

[6] 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. “The Gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. “Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation”. In: *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*. SC ’11. Seattle, Washington: ACM, 2011, 52:1–52:12. DOI: 10.1145/2063384.2063454.

[8] Dehao Chen, N. Vachharajani, R. Hundt, Xinliang Li, S. Eranian, Wenguang Chen, and Weimin Zheng. "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations". In: *IEEE Transactions on Computers* 62.2 (Feb. 2013), pp. 376–389. DOI: 10.1109/TC.2011.233.

[9] Robbert H. Dennard, Fritz H. Gaensslen, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical imensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[10] David Eklov, David Black-Schaffer, and Erik Hagersten. "Fast Modeling of Shared Caches in Multicore Systems". In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. HiPEAC '11. Heraklion, Greece: ACM, 2011, pp. 147–157. DOI: 10.1145/1944862.1944885.

[11] David Eklov and Erik Hagersten. "StatStack: Efficient Modeling of LRU Caches". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. ISPASS '10. Mar. 2010. DOI: 10.1109/ISPASS.2010.5452069.

[12] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. "Bandwidth Bandit: Quantitative Characterization of Memory Contention". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. DOI: 10.1109/CGO.2013.6494987.

[13] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. "Cache Pirating: Measuring the Curse of the Shared Cache". In: *Proc. International Conference on Parallel Processing (ICPP)*. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 165–175. DOI: 10.1109/ICPP.2011.15.

[14] David Eklov, Nikos Nikoleris, and Erik Hagersten. "A Software Based Profiling Method for Obtaining Speedup Stacks on Commodity Multi-cores". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2014, pp. 148–157. DOI: 10.1109/ISPASS.2014.6844479.

[15] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. "Speedup Stacks: Identifying Scaling Bottlenecks in Multi-threaded Applications". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 145–155. DOI: 10.1109/ISPASS.2012.6189221.

[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. “A Mechanistic Performance Model for Superscalar Out-of-order Processors”. In: *ACM Transactions on Computer Systems (TOCS)* 27.2 (May 2009), 3:1–3:37. DOI: `10.1145/1534909.1534910`.

[17] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. “Interval Simulation: Raising the Level of Abstraction in Architectural Simulation”. In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. HPCA, ’10. IEEE Computer Society, 2010, pp. 307–318.

[18] Tejas S. Karkhanis and James E. Smith. “A First-Order Superscalar Processor Model”. In: *Proc. International Symposium on Computer Architecture (ISCA)*. ISCA ’04. Munchen, Germany: IEEE Computer Society, 2004, pp. 338–349.

[19] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. “Interval-Based Models for Run-Time DVFS Orchestration in Superscalar Processors”. In: *Proc. International Conference on Computing Frontiers (CF)*. Proc.7th ACM international conference on Computing frontiers. Bertinoro, Italy: ACM, 2010, pp. 287–296. DOI: `http://doi.acm.org/10.1145/1787275.1787338`.

[20] Terje Mathisen. “Pentium Secrets: Undocumented features of the Intel Pentium can give you all the information you need to optimize Pentium code”. In: *Byte Magazine* 19.7 (July 1994), 191–??

[21] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. “Evaluation Techniques for Storage Hierarchies”. In: *IBM Syst. J.* 9.2 (June 1970), pp. 78–117. DOI: `10.1147/sj.92.0078`.

[22] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics Magazine* 38.8 (Apr. 1965), pp. 114–117.

[23] Todd C. Mowry and Chi-Keung Luk. “Predicting Data Cache Misses in Non-Numeric Applications through Correlation Profiling”. In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. MICRO 30. Research Triangle Park, North Carolina, United States: IEEE Computer Society, 1997, pp. 314–320.

[24] Nikos Nikoleris, David Eklov, and Erik Hagersten. “Extending Statistical Cache Models to Support Detailed Pipeline Simulators”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2014, pp. 148–157. DOI: `10.1109/ISPASS.2014.6844479`.

[25] Nikos Nikoleris, Erik Hagersten, and Trevor E. Carlson. *Delorean: Virtualized Directed Profiling for Cache Modeling in Sampled Simulation*. Tech. rep. 2018-014. Department of Information Technology, Uppsala University, Dec. 2018.

[26] Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E. Carlson. "CoolSim: Statistical Techniques to Replace Cache Warming with Efficient, Virtualized Profiling". In: *Proc. Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. July 2016, pp. 106–115. DOI: 10.1109/SAMOS.2016.7818337.

[27] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 271–282. DOI: 10.1109/HPCA.2018.00032.

[28] Erez Perelman, Greg Hamerly, and Brad Calder. "Picking Statistically Valid and Early Simulation Points". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. PACT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 244–255.

[29] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite". In: *Proc. International Symposium on Computer Architecture (ISCA)*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 412–423. DOI: 10.1145/1250662.1250713.

[30] Andreas Sandberg, David Eklov, and Erik Hagersten. "Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses". In: *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. DOI: 10.1109/SC.2010.44.

[31] Andreas Sandberg, Trevor E. Nikoleris Nikos Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed". In: *Proc. International Symposium on Workload Characterization (IISWC)*. Washington, DC, USA: IEEE Computer Society, Nov. 2015, pp. 183–192. DOI: 10.1109/IISWC.2015.29.

[32] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Phase Guided Profiling for Fast Cache Modeling". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. CGO '12. San Jose, California: ACM, 2012, pp. 175–185. DOI: 10.1145/2259016.2259040.

[33] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. "Automatically Characterizing Large Scale Program Behavior". In: *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS X. San Jose, California: ACM, 2002, pp. 45–57. DOI: 10.1145/605397.605403.

[34] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (2005), pp. 202–210.

[35] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. "TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes". In: *ACM SIGMETRICS Performance Evaluation Review* 33.1 (June 2005), pp. 408–409. DOI: 10.1145/1071690.1064278.

[36] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. "SimFlex: Statistical Sampling of Computer System Simulation". In: *IEEE Micro* 26.4 (July 2006), pp. 18–31. DOI: 10.1109/MM.2006.79.

[37] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling". In: *Proc. International Symposium on Computer Architecture (ISCA)*. ISCA '03. San Diego, California: ACM, 2003, pp. 84–97. DOI: 10.1145/859618.859629.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1756

Editor: The Dean of the Faculty of Science and Technology

Distribution: publications.uu.se
urn:nbn:se:uu:diva-369490

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019