UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1136

Understanding Multicore Performance

Efficient Memory System Modeling and Simulation

ANDREAS SANDBERG

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2014

ISSN 1651-6214
ISBN 978-91-554-8922-9
urn:nbn:se:uu:diva-220652

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, Informationsteknologiskt Centrum, Lägerhyddsvägen 2, Uppsala, Thursday, 22 May 2014 at 09:30 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor David A. Wood (Department of Computer Sciences, University of Wisconsin-Madison).

Abstract

Sandberg, A. 2014. Understanding Multicore Performance: Efficient Memory System Modeling and Simulation. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1136. i–x, 54 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-8922-9.

To increase performance, modern processors employ complex techniques such as out-of-order pipelines and deep cache hierarchies. While the increasing complexity has paid off in performance, it has become harder to accurately predict the effects of hardware/software optimizations in such systems. Traditional microarchitectural simulators typically execute code 10 000×–100 000× slower than native execution, which leads to three problems: First, high simulation overhead makes it hard to use microarchitectural simulators for tasks such as software optimizations where rapid turn-around is required. Second, when multiple cores share the memory system, the resulting performance is sensitive to how memory accesses from the different cores interleave. This requires that applications are simulated multiple times with different interleaving to estimate their performance distribution, which is rarely feasible with today's simulators. Third, the high overhead limits the size of the applications that can be studied. This is usually solved by only simulating a relatively small number of instructions near the start of an application, with the risk of reporting unrepresentative results.

In this thesis we demonstrate three strategies to accurately model multicore processors without the overhead of traditional simulation. First, we show how microarchitecture-independent memory access profiles can be used to drive automatic cache optimizations and to qualitatively classify an application's last-level cache behavior. Second, we demonstrate how high-level performance profiles, that can be measured on existing hardware, can be used to model the behavior of a shared cache. Unlike previous models, we predict the effective amount of cache available to each application and the resulting performance distribution due to different interleaving without requiring a processor model. Third, in order to model future systems, we build an efficient sampling simulator. By using native execution to fast-forward between samples, we reach new samples much faster than a single sample can be simulated. This enables us to simulate multiple samples in parallel, resulting in almost linear scalability and a maximum simulation rate close to native execution.

Keywords: Computer Architecture, Simulation, Modeling, Sampling, Caches, Memory Systems, gem5, Parallel Simulation, Virtualization, Sampling, Multicore

Andreas Sandberg, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Andreas Sandberg 2014

ISSN 1651-6214
ISBN 978-91-554-8922-9
urn:se:uu:diva-220652 (http://urn.kb.se/resolve?urn=urn:se:uu:diva-220652)

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. Andreas Sandberg, David Eklöv, 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)*. 2010. DOI: 10.1109/SC.2010.44

I'm the primary author of this paper. David Eklöv contributed to discussions and ran initial simulations.

II. Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. "Efficient Techniques for Predicting Cache Sharing and Throughput". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 305–314. DOI: 10.1145/2370816.2370861

I'm the primary author of this paper.

III. Andreas Sandberg, Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Modeling Performance Variation Due to Cache Sharing". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2013, pp. 155–166. DOI: 10.1109/HPCA.2013.6522315

I designed and implemented the cache sharing model. Andreas Sembrant contributed to discussions, and provided phase detection software and reference data.

IV. Andreas Sandberg, Erik Hagersten, and David Black-Schaffer. *Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed*. Tech. rep. 2014-005. Department of Information Technology, Uppsala University, Mar. 2014

I'm the primary author of this paper.

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

Other publications not included:

- Andreas Sandberg and Stefanos Kaxiras. "Efficient Detection of Communication in Multi-Cores". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2009, pp. 119–121

  I'm the primary author of this paper.

- Andreas Sandberg, David Eklöv, and Erik Hagersten. "A Software Technique for Reducing Cache Pollution". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2010, pp. 59–62

  I'm the primary author of this paper.

- Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. "A Simple Statistical Cache Sharing Model for Multicores". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2011, pp. 31–36

  I'm the primary author of this paper.

- Muneeb Khan, Andreas Sandberg, and Erik Hagersten. "A Case for Resource Efficient Prefetching in Multicores". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2014, pp. 137–138

  I was involved in discussions throughout the project and wrote some of the software.

Contents

Papers

List of Abbreviations

CMP	chip multiprocessor
CPI	cycles per instruction
CPU	central processing unit
DRAM	dynamic random-access memory
ETM	evict to memory
FIFO	first-in first-out
FSA	full speed ahead
GIPS	giga instructions per cycle
IPC	instructions per cycle
KVM	kernel virtual machine
L1	level one cache
L2	level two cache
L3	level three cache
LLC	last-level cache
LRU	least recently used
MIPS	mega instructions per cycle
MRU	most recently used
OoO	out of order
pFSA	parallel full speed ahead
POI	point of interest
SLLC	shared last-level cache
SMARTS	sampling microarchitecture simulation

1 Introduction

The performance of a computer system is decided by three factors: how fast instructions can be executed, how fast instructions can be delivered to the processor, and how fast data can be delivered to the processor. Due to advances in manufacturing technologies (smaller and faster transistors) and advances in computer architecture (e.g., pipelining, multiple instruction issue, and out-of-order execution), it is generally possible to execute instructions much faster than data and instructions can be delivered from memory. In order to solve the issue of slow memory, architects have resorted to using hierarchies of fast, but small, cache memories to hide the latency of main memory (DRAM) accesses.

In the late 90s, it was clear that optimizations exploiting instruction-level parallelism, such as out-of-order execution, were not going to continue to provide performance improvements. Instead, researchers started to look into the possibility of putting multiple execution cores on the same processor chip, forming a chip multiprocessor or multicore processor. This meant that the previously exclusive cache hierarchy, and often exclusive memory controller, became shared between all cores executing on the same chip. Figure 1.1 shows the memory hierarchy in a typical

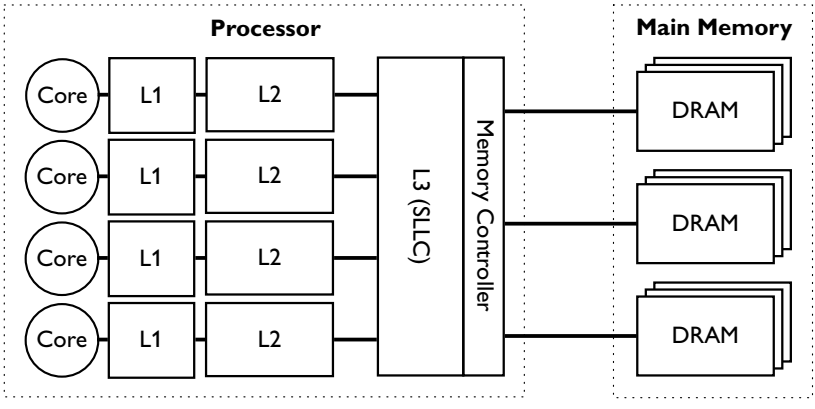


Figure 1.1: Memory hierarchy of a typical multicore processor with two private cache levels and one shared level. The processor has an on-chip memory controller with three memory channels.

multicore processor (e.g., Intel Nehalem). In this case, each core has access to a set of private cache levels (L1 & L2) and all cores on the chip share the last-level cache (L3) and the memory controller. Understanding how these resources are shared has become crucial when analyzing the performance of a modern processor. This thesis focuses on methods to model the behavior of modern multicore processors and their memory systems.

There are different approaches to modeling processors and memory systems. The amount of detail needed from a model depends on how the results are going to be used. When modeling a memory system with the goal of program optimizations, it can be enough to know which instructions are likely to cause cache misses. This can be modeled by statistical models such as StatCache [3] or StatStack [10], which model an application's *miss ratio* (misses per memory access) as a function of cache size. Since miss ratios can be associated with individual instructions, such data can be used to guide cache optimizations. In Paper I, which we describe in Chapter 2, we demonstrate a method that uses StatStack to classify an application's *qualitative* behavior (e.g., which applications are likely to inflict performance problems upon other applications) in the shared last-level cache. We use this classification to reason about applications' suitability for cache optimizations. We also demonstrate a fully automatic method that uses StatStack to find instructions that waste cache space and uses existing hardware support to bypass one or more cache levels to avoid such waste.

The classification from Paper I gives us some, *qualitative*, information about *which* applications are likely to waste cache resources and *which* applications are likely to suffer from such waste. However, it does not *quantify* how the cache is shared and the resulting performance. Several methods [5, 7, 8, 40, 47, 48] have been proposed to quantify the impact of cache sharing. However, they either require expensive stack distance traces or expensive simulation. There have been attempts [9, 48] to quantify cache sharing using high-level application profiles. However, these methods depend on performance models that estimate an applications execution rate from its miss ratio. Such models are usually hard to produce as optimizations in high-performance cores (e.g., overlapping memory accesses and out-of-order execution) make the relationship between miss ratio and performance non-trivial.

In Paper II, we propose a cache sharing model that uses application profiles that can be measured on existing hardware with low overhead [11]. These application profiles treat the core as a black box and incorporate the performance information that would otherwise have had to be estimated. Our model enables us to both predict the amount of cache available to each application and the resulting performance. We

extend this model in Paper III to account for time-varying application behavior. Since cache sharing depends on how memory accesses from different cores interleave, it is no longer sufficient to just look at average performance when analyzing the performance impact of a shared cache. Instead, we look at distributions of performance and show that the expected performance can vary significantly (more than 15%) between two runs of the same set of applications. We describe these techniques in Chapter 3.

In order to model future systems, researchers often resort to using simulators. However, the high execution overhead (often in the order of 10 000× compared to native execution) of traditional simulators limits their usability. The overhead of simulators is especially troublesome when simulating interactions within mixed workloads as the number of combinations of co-executing applications quickly grow out of hand. The cache sharing models from Paper II & III can be used to decrease the amount of simulation needed in a performance study as the simulator only needs to produce one performance profile per application, while the numerous interactions between applications can be estimated using an efficient model. While this approach can decrease the amount of simulation needed in a study, simulating large benchmark suits is still impractical. Sampling, where only a fraction of an application needs to be simulated in detail, has frequently been proposed [1, 6, 12, 37, 43, 44, 46] as a solution to high simulation overhead. However, most sampling approaches either depend on relatively slow (1000×–10× overhead) functional simulation [1, 6, 12, 46] to fast forward between samples or checkpoints of microarchitectural state [37, 43, 44]. Checkpoints can sometimes be amortized over a large number of repeated simulations. However, new checkpoints must be generated if a benchmark, or its input data, changes. Depending on the sampling framework used, checkpoints might even need to be regenerated when changing microarchitectural features such as cache sizes or branch predictor settings.

In order to improve simulator performance and usability, we propose offloading execution to the host processor using hardware virtualization when fast-forwarding simulators. In Paper IV, we demonstrate an extension to the popular gem5 [4] simulator that enables offloading using the standard KVM [16] virtualization interface in Linux, leading to extremely efficient fast-forwarding (10% overhead on average). Native execution in detailed simulators has been proposed before [23–25, 49]. However, existing proposals either use obsolete hardware [24], require dedicated host machines and modified guests [49], or are limited to simulating specific subsystems [23, 25]. Our gem5 extensions run on off-the-shelf hardware, with unmodified host and guest operating systems, in a simulation environment with broad device support. We show how

these extensions, which are now available as a part of the gem5 distribution, can be used to implement highly efficient simulation sampling that has less than 10× slowdown compared to native execution. Additionally, the rapid fast-forwarding makes it possible to reach new samples much faster than a single sample can be simulated, which exposes *sample-level parallelism*. We show how this parallelism can be exploited to further increase simulation speeds. We give an overview of these simulation techniques in Chapter 4 and discuss related ongoing and future research in Chapter 5.

2 Cache Bypass Modeling for Automatic Optimizations

Applications benefit differently from the amount of cache capacity available to them; some are very sensitive, while others are not sensitive at all. In many cases, large amounts of data is installed in the cache and hardly ever reused throughout its lifetime in the cache hierarchy. We refer to cache lines that are infrequently reused as *non-temporal*. Such non-temporal cache lines pollute the cache and waste space that could be used for more frequently reused data.

Figure 2.1 illustrates how a set of benchmarks from SPEC CPU2006 behave with respect to an 8 MB shared last-level cache (SLLC). Applications to the right in this graph install large amounts of data that is seldom, or never, reused while in the cache. This cache pollution can hurt the performance of both the application causing it and co-running applications on the same processor. Applications to the left install less data and their data sets mostly fit in the cache hierarchy, they normally benefit from caching. The applications' benefit from the SLLC is shown on the y-axis, where applications near the top are likely to benefit more from the cache available to them, while applications near the bottom do not benefit as much. If we could identify applications that do not benefit from caching, we could potentially optimize them to decrease their impact on sensitive applications without negatively affecting their own performance.

The applications to the right in Figure 2.1 show the greatest potential for cache optimizations as they install large amounts of non-temporal data that is reused very infrequently. Applications in the lower right corner of the chart are likely to not benefit from caching at all. In fact, almost none of the data they install in the cache is likely to be reused, which means that they tend to pollute the SLLC by wasting large amounts of cache that could have been used by other applications. If we could identify the instructions installing non-temporal data, we could potentially use cache-management instructions to disable fruitless caching and increase the amount of cache available to applications that could benefit from the additional space.

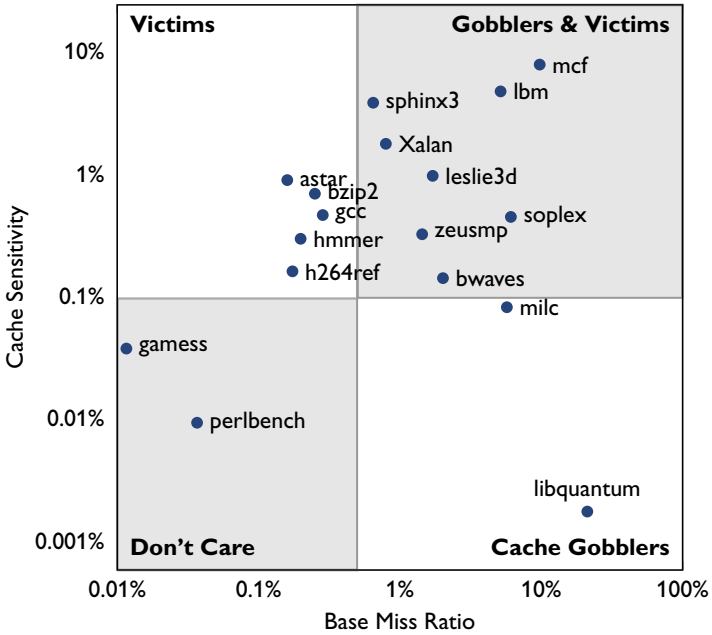


Figure 2.1: Cache usage classification for a subset of the SPEC CPU2006 benchmarks.

In Paper I, we demonstrate a high-level, qualitative, classification scheme that lets us reason about how applications compete for cache resources and which applications are good candidates for cache optimizations. Using a per-instruction cache model we demonstrate how individual instructions can be classified depending on their cache behavior. This enables us to implement a profile-driven optimization technique that uses a statistical cache model [10] and low-overhead profiling [2, 33, 41] to identify which instructions use data that is unlikely to benefit from caching.

Our method uses per-instruction cache reuse information from the cache model to identify instructions installing data in the cache that is unlikely to benefit from caching. We use this information in a compiler optimization pass to automatically modify the offending instructions using existing hardware support. The modified instructions bypass parts of the cache hierarchy, preventing them from polluting the cache. We demonstrate how this optimization can improve performance of mixed workloads running on existing commodity hardware.

Previous research into cache bypassing has mainly focused on hardware techniques [13, 14, 21, 22] to detect and avoid caching of non-temporal data. Such proposals are important for future processors, but

are unlikely to be adopted in commodity processors. There have been some proposals that use software techniques [36, 38, 42, 45] in the past. However, most of these techniques require expensive simulation or hardware extensions, which makes their implementation unlikely. Our technique uses existing hardware support and avoids expensive simulation by instead using low-overhead statistical cache modeling.

2.1 Efficient Cache Modeling

Modern processors often use an approximation of the least-recently used (LRU) replacement policy when deciding which cache line to evict from the cache. A natural starting point when modeling LRU caches is the *stack distance* [18] model. When using the stack distance abstraction, the cache can be thought of as a stack of elements (cache lines). The first time a cache line is accessed, it is pushed onto the stack. When a cache line is reused, it is removed from the stack and pushed onto the top of the stack. A stack distance is defined as *the number of unique cache lines accessed between two successive memory accesses to the same cache line*, which corresponds to the number of elements between the top of the stack and the element that is reused. An access is a hit in the cache if the stack distance is less than the cache size (in cache lines). An application's stack distance distribution can therefore be used to efficiently compute its miss ratio (misses per memory access) for any cache size by computing the fraction of memory accesses with a stack distances greater than the desired cache size.

Measuring stack distances is normally very expensive (unless supported through a hardware extension [35]) since it requires state tracking over a potentially long reuse. In Paper I, we use StatStack [10] to estimate stack distances and miss ratios. StatStack is a statistical model for fully associative caches with LRU replacement. Modeling fully associative LRU caches is, for most applications, a good approximation of the set-associative pseudo-LRU caches implemented in hardware. StatStack estimates an application's stack distances using a sparse sample of the application's *reuse distances*. Unlike stack distances, reuse distances count *all* memory accesses between two accesses to the same cache line. Existing hardware performance counters can therefore be used to measure reuse distances, but not stack distances, since there is no need to keep track of unique accesses. This leads to a very low overhead, implementations have been demonstrated with overheads as low as 20%–40% [2, 33, 41], which is orders of magnitude faster than traditional stack distance profiling.

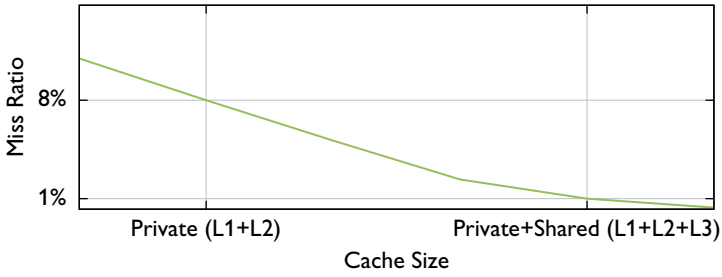


Figure 2.2: Miss ratio curve of an example application. Assuming a three level cache hierarchy (where the last level is shared) that enforces exclusion, the miss ratio of the private caches (i.e., misses being resolved by L3 or memory) is the miss ratio at the size of the combined L1 and L2.

Our cache-bypassing model assumes that the cache hierarchy enforces exclusion (i.e., data is only allowed to exist in one level at a time) and can be modeled as a contiguous stack. We can therefore think of each level in the cache hierarchy as a contiguous segment of the reuse stack. For example, the topmost region of the stack corresponds to L1, the following region to L2, and so on. If we plot an application's *miss ratio curve* (i.e., its miss ratio as a function of cache size), we can visualize how data gets reused from different cache levels. For example, the application in Figure 2.2 reuses 92% of its data from the private caches because its miss ratio at the combined size of the L1 and L2 is 8%. The addition of an L3 cache further decreases the miss ratio to 1%.

2.2 Classifying Cache Behavior

Applications behave differently depending on the amount of cache available to them. Since the last-level cache (LLC) of a multicore is shared, applications effectively get access to different amounts of cache depending on the other applications sharing the same cache. We refer to this competition for the shared cache as *cache contention*. Some applications are very sensitive to cache contention, while others are largely unaffected. For example, the applications in Figure 2.3 behave differently when they are forced to use a smaller part of the LLC. The miss ratio of application 1 is completely unaffected, while application 2 experiences more than 2× increase in miss ratio. This implies that application 2 is likely to suffer a large slowdown due to cache contention, while application 1 is largely

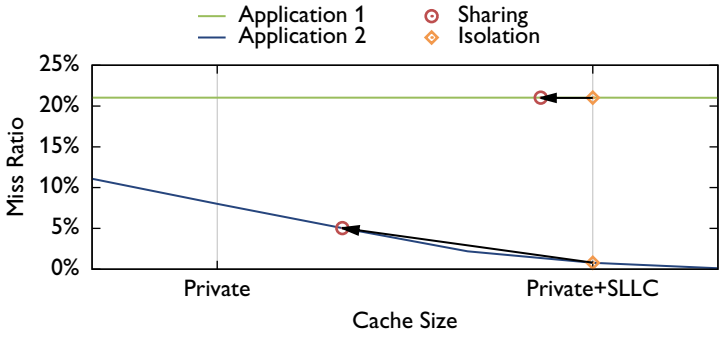


Figure 2.3: Applications benefit differently from caching. Application 1 uses large amounts of cache, but does not benefit from it, while application 2 uses less cache but benefits from it. If the applications were to run together, application 1 is likely to get more cache, negatively impacting the performance of application 2 without noticeable benefit to itself.

unaffected. Despite deriving less benefit from the shared cache, application 2 is likely to keep more of its data in the LLC due to its higher miss ratio.

In order to understand where to focus our cache optimizations, we need a classification scheme to identify which applications are sensitive to cache contention and which are likely to cause cache contention. In Paper I, we introduce a classification scheme that approximates an application's ability to cause cache contention based on its miss ratio when run in isolation (base miss ratio) and the increase in miss ratio when only having access to the private cache. The base miss ratio corresponds to how likely an application is to cause cache contention and the increase in miss ratio to the *cache sensitivity*.

Using an application's sensitivity and base miss ratio, we can reason about its behavior in the shared cache. Figure 2.1 shows the classification of a subset of the SPEC CPU2006 benchmarks. In general, the higher the base miss ratio, the more cache is wasted. Such applications are likely to be good candidates for cache optimizations where one or more cache levels are bypassed to prevent data that is unlikely to be reused from polluting the cache. Applications with a high sensitivity on the other hand are likely to be highly affected by cache contention. In order to quantify the impact of cache contention, we need to predict the cache access rate, which implies that we need a performance model taking cache and core performance into account. Such a quantitative cache sharing model is discussed in Chapter 3.

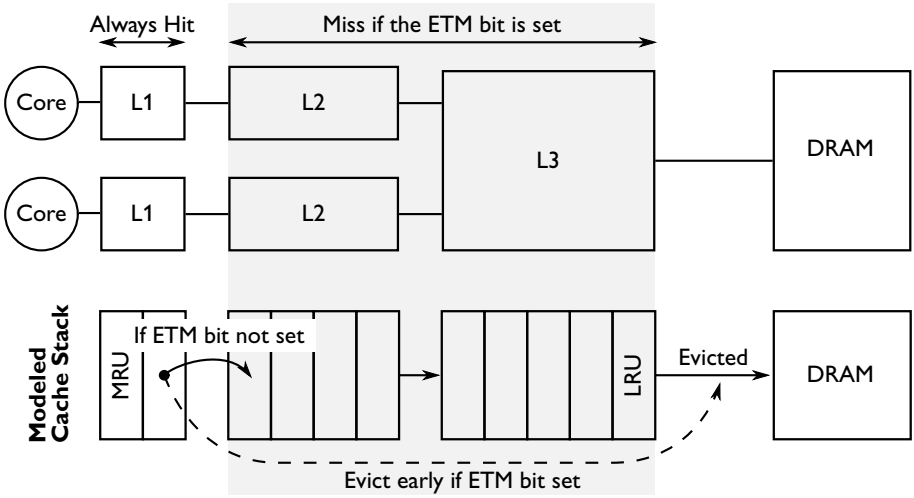


Figure 2.4: A system where data flagged as *evict-to-memory* (ETM) in L1 can be modeled using stack distances. Each level (top) corresponds to a contiguous segment of the cache stack (bottom). Upon eviction, cache lines with the ETM bit set are evicted straight to memory from L1.

2.3 Optimizing Memory Accesses Causing Cache Pollution

Many modern processors implement mechanisms to control where data is allowed to be stored in the cache hierarchy. This is sometimes known as *cache bypassing* as a cache line is prevented from being installed in one or more cache levels, effectively bypassing them. In Paper I, we describe a method to automatically detect which *instructions* cause cache pollution and can be modified to bypass the parts of the cache hierarchy. In order to accurately determine when it is beneficial to bypass caches we need to understand how the hardware handles accesses that are flagged as having a non-temporal behavior. Incorrectly flagging an instruction as non-temporal can lead to bad performance since useful data might be evicted from the cache too early. The behavior we model assumes that data flagged as non-temporal is allowed to reside in the L1 cache, but takes a different path when evicted from L1. Instead of being installed in L2, non-temporal data is evicted straight to memory. For example, some AMD processors treat cache lines flagged as non-temporal this way. We model this behavior by assuming that every cache line has a special bit, the *evict to memory* (ETM) bit, that can be set for non-temporal cache lines. Cache lines with the ETM bit set are evicted from L1 to memory instead of being evicted to the L2 cache. This behavior is illustrated in Figure 2.4.

A compiler can automatically use the knowledge about which instructions cause cache pollution to limit it. In Paper I, we demonstrate a profile-driven optimization pass that automatically sets non-temporal hints on memory accesses that were deemed to have a non-temporal behavior. Using this profile-driven optimization, we were able to demonstrate up to 35% performance improvement for mixed workloads on existing hardware.

Since our optimization work on *static* instructions and stack distances are a property of *dynamic* memory accesses, we need to understand how flagging a static instruction as non-temporal affects future misses. A naïve approach would limit optimizations to static instructions where all stack distances predict a future cache miss. However, this unnecessarily limits optimizations opportunities. In order to make the model easier to follow, we break it into three steps. Each step adds more detail to the model and brings it closer to our reference hardware.

Strictly Non-Temporal Accesses: By looking at an instruction's stack distance distribution, we can determine if the next access to the cache line used by that instruction is likely to be a cache miss. An instruction has non-temporal behavior if all stack distances are larger or equal to the size of the cache. In that case, we know that the next instruction to touch the same data is very likely to be a cache miss. We can therefore flag the instruction as non-temporal and bypass the entire cache hierarchy without incurring additional cache misses.

Handling ETM Bits: Most applications, even purely streaming ones that do not reuse *data*, exhibit spatial locality and reuse *cache lines* (e.g., a reading all words in a cache line sequentially). Hardware implementations of cache bypassing may allow data flagged as non-temporal to live in parts of the cache hierarchy (e.g., L1) to accommodate such behaviors. We model this by assuming that whenever the hardware installs a cache line flagged as non-temporal, it installs it in the MRU position with the ETM bit set. Whenever a normal memory access touches a cache line, the ETM bit is cleared. Cache lines with the ETM bit set are evicted from L1 to memory instead of to the L2 cache, see Figure 2.4. This allows us to consider memory accesses as non-temporal even if they have short reuses that hit in the L1 cache. To flag an instruction as non-temporal, we now require that there is at least one future reuse that will be a miss and that the number of accesses reusing data in the area of the LRU stack bypassed by ETM-flagged cache lines (the gray area in Figure 2.4) is small (i.e., we only tolerate a small number of additional misses).

Handling sticky ETM bits: There exists hardware (e.g., AMD family 10h) that does not reset the ETM bit when a normal instruction reuses an ETM-flagged cache line. This situation can be thought of as *sticky ETM bits*, as they are only reset on cache line evictions. In this case, we

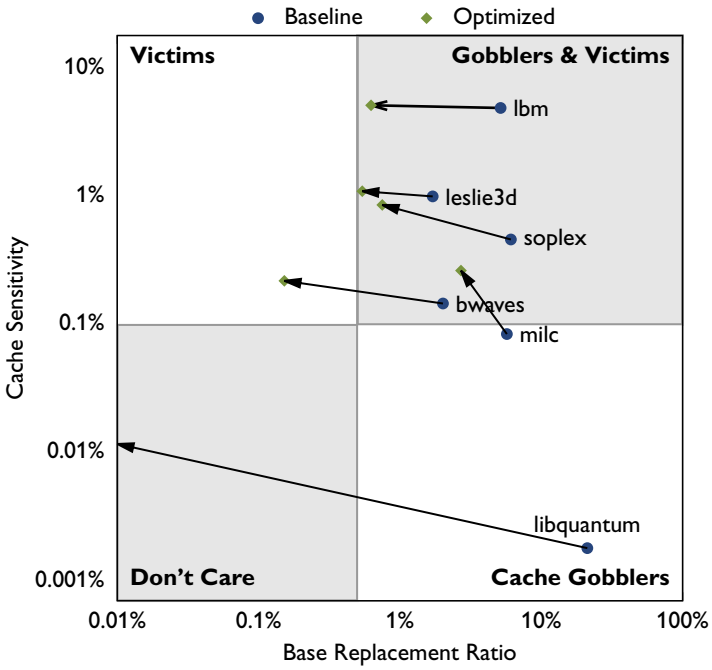


Figure 2.5: Classification of a subset of the SPEC CPU2006 benchmarks after applying our cache optimizations. All of the applications move to the left in the classification chart, which means that they cause less cache pollution.

can no longer just look at the stack distance distribution of the current instruction since the next instruction to reuse the same cache line might result in a reuse from one of the bypassed cache levels. Due to the stickiness of ETM bits, we need to ensure that both the current instruction *and* any future instruction reusing the cache line through L1 will only access it from L1 or memory to prevent additional misses.

2.4 Effects on Benchmark Classification

Bypassing caches for some memory accesses changes how applications compete for shared caches. In cache-optimized applications, some memory accesses fetch data without installing it in one or more cache levels. Since cache contention is caused by cache replacements, we need to reclassify optimized application based on how frequently they cause cache replacements (i.e., their replacement ratio) instead of how frequently they miss in the cache. Figure 2.5 shows how the classification changes for applications that were deemed to be good targets for cache optimizations. In all cases, the number of cache replacements decrease (decreased

replacement ratio), which leads to less cache contention and more space for useful data. In Paper I, we show how these changes in classification translate into performance improvements when running mixed workloads on a real system.

2.5 Summary

In Paper I, we demonstrated a method to classify an application's cache usage behavior from its miss ratio curve. This enables us to reason, qualitatively, about an application's cache behavior. Using this classification scheme, we identified applications that were suitable targets for cache optimizations and demonstrated a profile-based method that automatically detects which instructions bring in data that does not benefit from caching. We show how this per-instruction information can be used by a compiler to automatically insert cache-bypass instructions. Our method uses a statistical cache model together with low-overhead application profiles, making it applicable to real-world applications.

The automatic cache bypassing method in Paper I optimized for the total amount of cache available in the target system. However, this might not be the optimal size. If the applications that are going to run together are known (or if the optimization is done online), we could determine the amount of cache available to each application and apply more aggressive optimizations. A prerequisite for such optimizations is an accurate cache model that can tell us how the cache is divided among applications, which we investigate in Paper II & III (see Chapter 3).

3 Modeling Cache Sharing

When modeling a multicore processor, we need to understand how resources are shared to accurately understand its performance. We might for example want to understand how performance is affected by different thread placements, how software optimizations affect sharing, or how a new memory system performs. The type of model needed can be very different depending on how the output of the model is going to be used. The classification introduced in the previous chapter is an example of a *qualitative model* that identifies high-level properties of an application (e.g., whether it is likely to cause cache contention), but does not *quantify* how the cache is shared and the resulting performance. This type of qualitative classification can be sufficient in some cases, such as when a scheduler decides where to execute a thread. A hardware designer evaluating a new memory system on the other hand will need a *quantitative model* that estimates the performance impact of different design options. However, the additional detail of a quantitative model usually comes with a high overhead.

One of the most common ways of quantifying application performance today is through simulation. This approach unfortunately limits the studies that can be performed due to the overhead imposed by state-of-the-art simulators. For example, the popular gem5 [4] simulator simulates around 0.1 million instructions per second (MIPS) on a system that natively executes the same workload at 3 billion instructions per second (GIPS), which is equivalent to a 30 000× slowdown. This clearly limits the scale of the experiments that can be performed. Additionally, if we are interested in measuring the impact of sharing, the number of combinations of applications running together quickly grows out of hand.

In this chapter, we describe methods to quantify the impact of cache sharing from Paper II & III. These methods enable us to estimate the amount of cache available to each application in a mixed workload as well as per-application execution rates and bandwidth demands.

One approach to model cache sharing is to extend an existing statistical cache model, such as StatStack [10], with support for cache sharing. This approach was taken by Eklöv et al. [9] in StatCC, which combines

StatStack and a simple IPC model to predict the behavior of a shared cache. The drawback of this approach is the need for a reliable performance model that predicts an application's execution rate (IPC) as a function of its miss ratio. Another approach would be to include performance information as a function of cache size in the input data to the model. Such performance profiles can be measured on existing hardware using Cache Pirating [11], which eliminates the need for complex performance models when modeling existing systems. When modeling future systems, we can generate the same profiles through simulation. This reduces the amount of simulation needed as the profiles are measured once per application, while cache sharing and the resulting performance can be estimated by an efficient model. In Paper II, we show that both simulated and measured application profiles can be used to model cache sharing.

In Paper II & III we model both how the cache is divided among co-running applications and how this affects performance. In Paper II, we focus on steady-state behavior where all applications have a time-stable cache behavior. In practice, however, many applications have time varying behavior. In Paper III, we extend the cache sharing model from Paper II to predict how such time-dependent behavior affects cache sharing. We show that looking at average behavior is not enough to accurately predict performance. Instead we look at how performance varies depending on how memory accesses from co-running applications interleave.

3.1 Measuring Cache-Dependent Behavior

Our cache sharing models use application profiles with information about cache misses, cache hits, and performance *as a function of cache size*. Such profiles can be measured on existing hardware using Cache Pirating [11]. This enables us to model existing systems as a black box by measuring how applications behave as a function of cache size on the target machine with low overhead.

Cache Pirating uses hardware performance monitoring facilities to measure target application properties at runtime, such as cache misses, hits, and execution cycles. To measure this information for varying cache sizes, Cache Pirating co-runs a small cache intensive stress application with the target application. The stress application accesses its entire data set in a tight loop, effectively stealing a configurable amount of shared cache from the target application. The amount of shared cache available to the target application is then varied by changing the cache footprint of the stress application. This enables Cache Pirating to measure any performance metric exposed by the target machine as a function of available cache size.

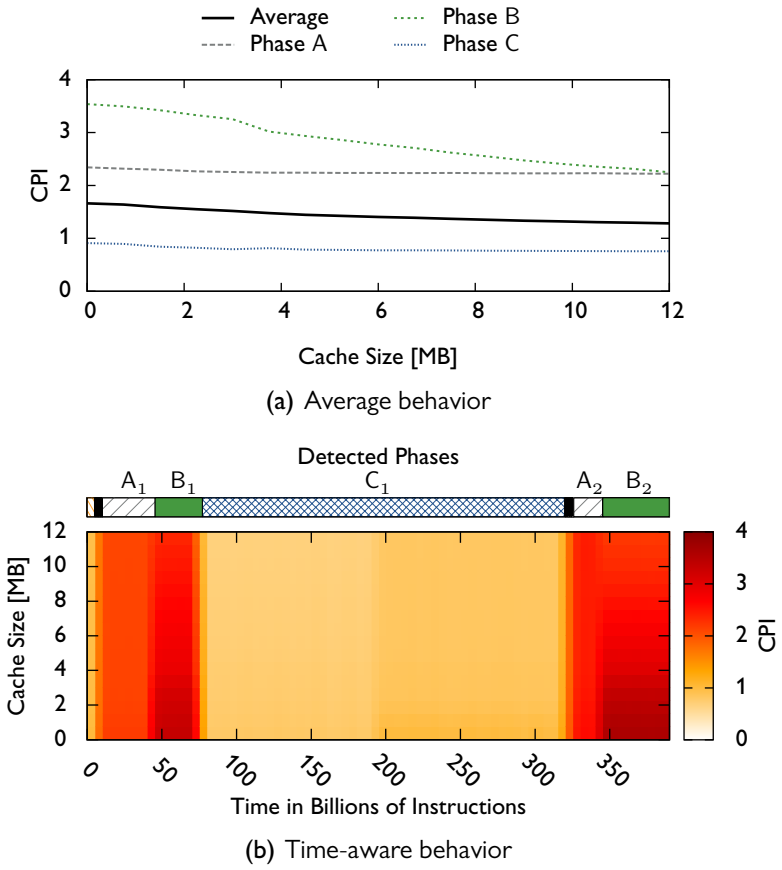


Figure 3.1: Performance (CPI) as a function of cache size as produced by Cache Pirating. Figure (a) shows the time-oblivious application average as a solid line as well as the average behavior of a few significant phases. Figure (b) shows the time-dependent cache sensitivity and the identified phases (above). The behavior of the three largest phases deviate significantly from the global average as can be seen by the dashed lines in Figure (a).

In order to model time-varying behavior, we extend Cache Pirating to measure an application's time-varying behavior. In its simplest form, time-varying behavior is sampled in windows of a fixed number of instructions. Capturing this time-varying behavior is important as very few real-world applications have a constant behavior. For example, the astar benchmark from SPEC CPU2006 has three distinct types of behavior, or phases, with very different performance. This is illustrated in Figure 3.1, which shows: a) the performance (CPI) as a function of cache size for the three different phases and the global average; and b) the time-varying behavior of the application annotated with phases. As seen

in Figure 3.1(a), the average does not accurately represent the behavior of any of the phases in the application.

Phase information can be exploited to improve modeling performance and storage requirements. We extend Cache Pirating to incorporate phase information using the ScarPhase [34] library. ScarPhase is a low-overhead, online, phase-detection library. A crucial property of ScarPhase is that it is execution-history based, which means that the phase classification is independent of cache sharing effects. The phases detected by ScarPhase can be seen in the top bar in Figure 3.1(b) for astar, with major phases labeled. This benchmark has three major phases; A, B and C, all with different cache behaviors. The same phase can occur several times during execution. For example, phase A occurs twice, once at the beginning and once at the end of the execution. We refer to multiple repetitions of the same phase as *instances* of the phase, e.g., A_1 and A_2 in Figure 3.1(b).

3.2 Modeling Cache Sharing

When modeling cache sharing, we look at co-executing application phases and predict the resulting amount of cache per application. We make the basic assumption that the behavior within a phase is time-stable and that sharing will not change as long as the same phases co-execute. We refer to the time-stable behavior when a set of phases co-execute as their *steady state*. When modeling applications with time-varying behavior, we need to predict which phases will co-execute. Knowing which phases will co-execute when the applications start, we model their behavior and use the calculated cache sizes to determine their execution rates from the cache-size dependent application profiles. Using the execution rates, we determine when the next phase transition occurs and redo the calculations for the next set of co-executing phases.

The amount of cache available to an application depends on two factors: The application's behavior and the cache replacement policy. In Paper II we introduce two cache sharing models, one for random replacement and one for LRU replacement. In terms of modeling, these two policies are very different. Unlike random replacement, where a replacement target is picked at random, the LRU policy exploits access history to replace the cache line that has been unused for the longest time.

A cache with random replacement can intuitively be thought of as an overflowing bucket. When two applications share a cache, their behavior can be thought of as two different liquids filling the bucket at different rates (their cache miss rates). The two in-flows correspond to misses that cause data to be installed in the cache and the liquid pouring out of the

bucket corresponds to cache replacements. If the in-flows are constant, the system will eventually reach a steady state. At steady state, the concentrations of the liquids are constant and proportional to their relative inflow rates. Furthermore, the out-flow rates of the different liquids are proportional to their concentrations (fractions of the SLLC). In fact, this very simple analogy correctly describes the behavior of random caches.

The overflowing bucket analogy can be extended to caches that use the LRU replacement policy. Cache lines that are not reused while in the cache can be thought of as the liquid in the bucket, while cache lines that are reused behave like ice cubes that float on top of the liquid and stay in the bucket.

3.3 Modeling LRU Replacement

LRU replacement uses access history to replace the item that has been unused for the longest time. We refer to the amount of time a cache line has been unused as its *age*. Whenever there is a replacement decision, the oldest cache line is replaced.

Since we only use high-level input data, we cannot model the behavior of individual cache lines or sets. Instead, we look at *groups* of cache lines with the same behavior and assume a fully-associative cache. Since the LRU policy always replaces the oldest cache line, we consider a group of cache lines to have the same behavior if they share the same *maximum age*, which enables us to identify the group affected by a replacement. Since the ages of the individual cache lines within a group will be timing-dependent, we model all entries in the cache with the same maximum age as having the same likelihood of replacement.

One of the core insights in Paper II is that we can divide the data in a shared cache into two different categories and use different models depending on their reuse patterns. The first category, *volatile data*, consists of all data that is not reused while in the cache. The second category, *sticky data*, contains all data that is reused in the cache.

The size of each application's volatile data set and sticky data set is cache-size dependent, the more cache available, the more volatile data can be reused and become sticky. Additionally, in a shared cache, the division between sticky and volatile data depends on the maximum age in the volatile group (which is shared between all cores). This means that we have to know the size of the sticky data sets to determine the size of the volatile data set and vice versa. In order to break this dependency, we use a fixed point solver that finds a solution where the ages of sticky and volatile data are balanced.

Modeling Volatile Data

When applications do not reuse their data before it is evicted from the cache, LRU caches degenerate into FIFO queues with data moving from the MRU position to the LRU position before being evicted. Similar to a random cache, the amount of cache allocated to an application is proportional to its miss rate.

Sticky data and volatile data from different applications compete for cache space using age, we therefore need to determine the maximum age of volatile data. Since LRU caches can be modeled as FIFO queues for volatile data, we can determine the maximum age of volatile data using Little's law [17]. Little's law sets up a relationship between the number of elements in a queue (size), the time spent in the queue (maximum age) and the arrival rate (miss rate). The miss rate can be read from application profiles for a given cache size, while the size of the volatile data set is whatever remains of the cache after sticky data has been taken into account.

Modeling Sticky Data

Unlike volatile data, sticky data stays in the cache because it is reused before it grows old enough to become a victim for eviction. When a sticky cache line is not reused frequently enough, it becomes volatile. This happens if a sticky cache line is older than the oldest volatile cache line. In our model, we make the decision to convert sticky data to volatile data for entire groups of cache lines with the same behavior (i.e., having the same maximum age).

In order to determine if a group of sticky data should be reclassified as volatile, we need to know its age. Similar to volatile data, we can model the maximum age of a group of sticky data using Little's law. In this case, each group of sticky cache lines can be thought of as a queue where cache lines get reused when they reach the head of the queue. After a reuse, the reused cache line is moved to the back of the queue.

The amount of sticky data can be estimated from how an application's hit ratio changes with its cache allocation. The relative change in hit ratio is proportional to the relative change in the sticky data. For example, if half of the misses an application currently experiences disappear when the amount of cache available to it is grown by a small amount, half of the application's currently volatile data must have transformed into sticky data.

Both the amount of volatile data and the maximum age of volatile data are described by differential equations. We describe these in detail in Paper II.

Solver

Using the requirements defined above, we can calculate how the cache is shared using a numerical fixed point solver. The solver starts with an initial guess, wherein the application that starts first has access to the entire cache and the other applications do not have access to any cache. The solver then lets all applications compete for cache space by enforcing the age requirement between sticky and volatile cache lines. If the age requirement cannot be satisfied for an application, the solver shrinks that application's cache allocation until the remaining sticky data satisfies the age requirement. The cache freed by the shrinking operation is then distributed among *all* applications by solving the sharing equations for the volatile part of the cache.

The process of shrinking and growing the amount of cache allocated to the applications is repeated until the solution stabilizes (i.e., no application changes its cache allocation significantly). Once the solver has arrived at a solution, we know how the cache is shared between the applications. Using this information, performance metrics (e.g., CPI) can be extracted from cache-size dependent application profiles like the ones used to drive the model.

3.4 Modeling Time

The difficulty in modeling time-dependent cache sharing is to determine which parts of the co-running applications (i.e., windows or phases) will co-execute. Since applications typically execute at different speeds depending on phase, we cannot simply use windows starting at the same dynamic instruction count for each application since they may not overlap. For example, consider two applications with different executions rates (e.g., CPIs of 2 and 4), executing windows of 100 million instructions. The slower application with a CPI of 4 will take twice as long to finish executing its windows as the one with a CPI of 2. Furthermore, when they share a cache they affect each other's execution rates.

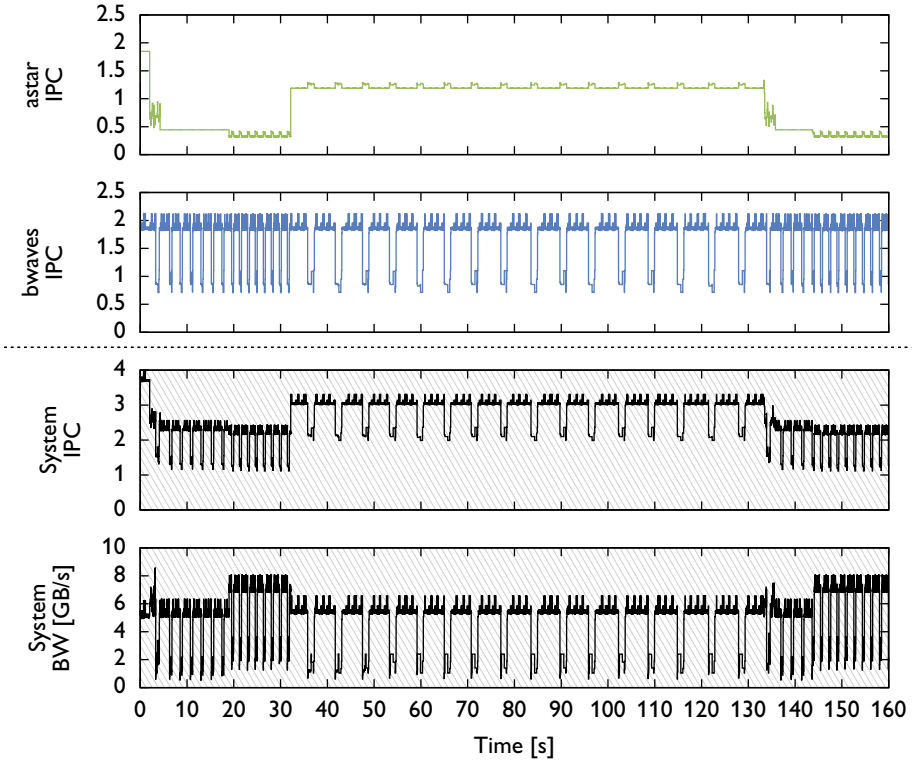
In Paper III, we demonstrate three different methods to handle time. The first, *window-based method* (Window) uses the execution rates of co-running windows to advance each application. The second, *dynamic-window-based method* (Dynamic Window), improves on the window-based method by exploiting basic phase information to merge neighboring windows with the same behavior. The third, *phase-based method* (Phase), exploits the recurring nature of some phases to avoid recalculating previously seen sharing patterns.

Window: To determine which windows are co-executing, we model per-window execution rates and advance applications independently between their windows. Whenever a new combination of windows occurs, we model their interactions to determine the new cache sharing and the resulting execution rates. This means that the cache model needs to be applied several times per window since windows from different applications will not stay aligned when scaled with their execution rates. For example, when modeling the slowdown of astar co-executing with bwaves, we invoke the cache sharing model roughly 13 000 times while astar only has 4 000 windows by itself.

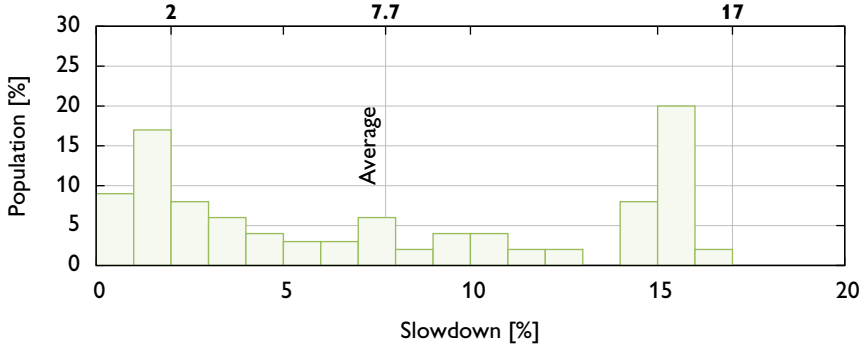
Dynamic Window: To improve the performance of our method, we need to reduce the number of times the cache sharing model is invoked. To do this, we merge multiple adjacent windows belonging to the same phase into a larger window, a dynamic window. For example, in astar (Figure 3.1), we consider all windows in phase A_1 as one unit (i.e., the average of the windows) instead of looking at every individual window within the phase. Compared to the window-based method, this method is dramatically faster. When modeling astar running together with bwaves we reduce the number of times the cache sharing model is used from 13 000 to 520, which leads to 25× speedup over the window-based method.

Phase: The performance can be further improved by merging the data for all instances of a phase. For example, when considering astar (Figure 3.1), we consider all phase instances of A (i.e., $A_1 + A_2$) as one unit. This optimization enables us to reuse cache sharing results for co-executing phases that reappear [39]. For example, when astar's phase A_1 co-executes with bwaves's phase B, we can save the cache sharing results and later reuse them if the second instance of A (A_2) co-executes with phase B in bwaves. In the example with astar and bwaves, we can reuse the results from previous cache sharing solutions 380 times. We therefore only need to run the cache sharing model 140 times. The performance of the phase-based method is highly dependent on an application's phase behavior, but it normally leads to a speed-up of 2–10× over the dynamic-window method.

The main benefit of the phase-based method is when determining performance variability of a mix. In this case, the same mix is modeled several times with slightly different offsets in starting times. The same co-executing phases will usually reappear in different runs. For example, when modeling 100 different runs of astar and bwaves, we need to evaluate 1 400 000 co-executing windows, but with the phase-based method we only need to run the model 939 times.



(a) Modeled performance of a single run



(b) Performance variation across 100 runs

Figure 3.2: Performance of astar co-executing with bwaves. Figure (a) shows the per-application performance (IPC), the aggregate system performance, and the memory bandwidth required to achieve this performance. Figure (b) shows how the performance of astar varies across 100 different runs with bwaves. The high performance variability indicates that we need performance data from many different runs to understand how such application behave.

Time-Varying Application Behavior

When modeling the behavior of a workload, we can predict how multiple performance metrics vary over time. In many ways, this is similar to the information we would get from a simulator running the same set of applications, but much faster. The behavior of two applications from SPEC CPU2006, astar and bwaves, is shown in Figure 3.2(a). The figure shows the performance (IPC) per application when co-scheduled and the aggregate system throughput and bandwidth. As seen in the figure, both applications exhibit time-varying behavior, which means that the aggregate behavior depends on how the applications are co-scheduled. It is therefore not possible to get an accurate description of the workload's behavior from one run, instead we need to look at a performance *distribution* from many runs.

In Paper III, we demonstrate both the importance of looking at performance distributions and an efficient method to model them. For example, looking at the slowdown distribution (performance relative to running in isolation) of astar running together with bwaves (Figure 3.2(b)), we notice that there is a large spread in slowdown. We observe an average slowdown of 8%, but the slowdown can vary between 1% and 17% depending on how the two applications' phases overlap. In fact, the probability of measuring a slowdown of 2% or less is more than 25%.

Measuring performance distributions has traditionally been a tedious task since they require performance measurements for a large number of runs. In the case of simulation, it might not be possible due to excessive simulation times. In fact, it might even be hard to estimate the distribution on real hardware. In Paper III, we show how our cache modeling technique can be used to efficiently estimate these distributions. For example, when measuring the performance distribution in Figure 3.2(b) on our reference system, we had to run both applications with 100 different starting offsets. This lead to a total execution time of almost seven hours. Using our model, we were able to reproduce the same results in less than 40 s (600× improvement).

3.5 Summary

In order to understand the behavior of a multicore processor, we need to understand cache sharing. In Paper II, we demonstrated a cache sharing model that uses high-level application profiles to predict how a cache is shared among a set of applications. In addition to the amount of cache available to each application, we can predict performance and bandwidth requirements. The profiles can either be measured with low overhead on existing systems using Cache Pirating [11] or produced using simulation. When using simulated profiles, the model reduces the amount of simulation needed to predict cache sharing since profiles only need to be created once per application. Interactions, and their effect on performance, between applications can be predicted by the efficient model.

In Paper III, we extended the cache sharing model to applications with time-varying behavior. In this case, it is no longer sufficient to look at average performance since the achieved performance can be highly timing sensitive. Instead our model enables us to look at performance distributions. Generating such distributions using simulation, or even by running the applications on real hardware, has previously been impractical due to large overheads.

When modeling future systems, we still depend on simulation to generate application profiles. Since modern simulators typically execute three to four orders of magnitude slower than the systems they simulate, generating such profiles can be very expensive. In Paper IV (see Chapter 4), we investigate a method to speed up simulation by combining sampled simulation with native execution.

4 Efficient Simulation Techniques

Profile-driven modeling techniques, like the ones presented in Chapter 3, can be used to efficiently predict the behavior of existing hardware without simulation. However, to predict the behavior of future hardware, application profiles need to be created using a simulator. Unfortunately, traditional simulation is very slow. Simulation overheads in the 1 000×–10 000× range compared to native execution are not uncommon. Many common benchmark suits are tuned to assess the performance of real hardware and can take hours to run natively; running them to completion in a simulator is simply not feasible. Figure 4.1 compares execution times of individual benchmarks from SPEC CPU2006 when running natively and projected simulation times using the popular gem5 [4] full-system simulator. While the individual benchmarks take 5–15 minutes to execute natively, they take between a week and more than a month to execute in gem5's fastest simulation mode. Simulating them in detail adds another order of magnitude to the overhead. The slow simulation

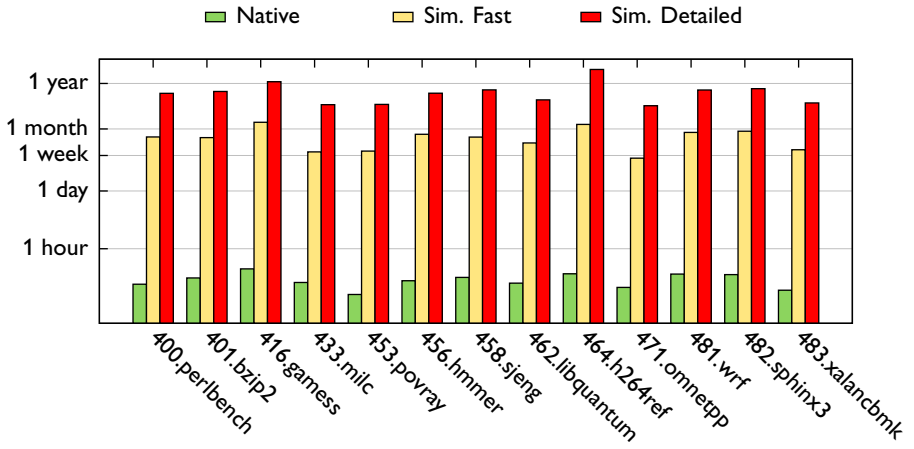


Figure 4.1: Native and projected execution times using gem5's functional and detailed out-of-order CPUs for a selection of SPEC CPU2006 benchmarks.

rate is a severe limitation when evaluating new high-performance computer architectures or researching hardware/software interactions. Faster simulation methods are clearly needed.

Low simulation speed has several undesirable consequences: 1) In order to simulate interesting parts of a benchmark, researchers often fast-forward to a point of interest (POI). In this case, fast forwarding to a new simulation point close to the end of a benchmark takes between a week and a month, which makes this approach painful or even impractical. 2) Since fast-forwarding is relatively slow and a sampling simulator such as SMARTS [46] can never execute faster than the fastest simulation mode, it is often impractical to get good full-application performance estimates using sampling techniques. 3) Interactive use is slow and painful. For example, setting up and debugging a new experiment would be much easier if the simulator could execute at human-usable speeds.

In this chapter, we describe methods from Paper IV to overcome these limitations by extending a classical full-system simulator to use hardware virtualization to execute natively between POIs. Using this extension we implement a sampling framework that enables us to quickly estimate the performance of an application running on a simulated system. The extremely efficient fast-forwarding between samples enables us to reach new sample points more rapidly than a single sample can be simulated. Using an efficient state-copying strategy, we can exploit sample-level parallelism to simulate multiple samples in parallel.

Our implementation targets gem5, which is a modular discrete-event full-system simulator. It provides modules simulating most common components in a modern system. The standard gem5 distribution includes several CPU modules, notably a detailed superscalar *out-of-order CPU module* and a simplified faster *functional CPU module* that can be used to increase simulation speed at a loss of detail. We extended this simulator to add support for native execution using hardware virtualization through a new *virtual CPU module*[1]. The virtual CPU module can be used as a drop-in replacement for other CPU modules in gem5, thereby enabling rapid execution and seamless integration with the rest of the simulator. We demonstrate how this CPU module can be used to implement efficient performance sampling and how the rapid execution between samples exposes parallelism that can be exploited to simulate multiple samples in parallel.

[1]The virtual CPU module, including support for both ARM and x86, has been included in stable gem5 releases since version 2013_10_14.

4.1 Integrating Simulation and Hardware Virtualization

The goals of simulation and virtualization are generally very different. Integrating hardware virtualization into gem5 requires that we ensure consistent handling of 1) simulated devices, 2) time, 3) memory, and 4) processor state. These issues are discussed in detail below:

Consistent Devices: The virtualization layer does not provide any device models, but it does provide an interface to intercept device accesses. A CPU normally communicates with devices through memory mapped IO and devices request service from the CPU through interrupts. Accesses to devices are intercepted by the virtualization layer, which hands over control to gem5. In gem5, we take the information provided by the virtualization layer and synthesize a simulated device access that is inserted into the simulated memory system, allowing it to be seen and handled by gem5's device models. Conversely, when a device requires service, the CPU module sees the interrupt request from the device and injects it into the virtual CPU using KVM's interrupt interface.

Consistent Time: Simulating time is difficult because device models (e.g., timers) execute in simulated time, while the virtual CPU executes in real time. A traditional virtualization environment solves this issue by running device models in real time as well. For example, if a timer is configured to raise an interrupt every second, it would setup a timer on the host system that fires every second and injects an interrupt into the virtual CPU. In a simulator, the timer model inserts an event in the event queue one second into the future and the event queue is executed tick by tick. We bridge the gap between simulated time and the time as perceived by the virtual CPU by restricting the amount of host time the virtual CPU is allowed to execute between simulator events. When the virtual CPU is started, it is allowed to execute until a simulated device requires service. Due to the different execution rates between the simulated CPU and the host CPU (e.g., a server simulating an embedded system), we scale the host time to make asynchronous events (e.g., interrupts) happen with the right frequency relative to the executed instructions.

Consistent Memory: Interfacing between the simulated memory system and the virtualization layer is necessary to transfer state between the virtual CPU module and the simulated CPU modules. Since gem5 stores the simulated system's memory as contiguous blocks of physical memory, we can look at the simulator's internal mappings and install the same mappings in the virtual system. This gives the virtual machine and the simulated CPUs the same view of memory. Additionally, since virtual

CPUs do not use the simulated memory system, we ensure that simulated caches are disabled (i.e., we write back and flush simulated caches) when switching to the virtual CPU module.

Consistent State: Converting between the processor state representation used by the simulator and the virtualization layer, requires detailed understanding of the simulator internals. There are several reasons why a simulator might be storing processor state in a different way than the actual hardware. For example, in gem5, the x86 flag register is split across several internal registers to allow more efficient dependency tracking in the pipeline models. We implement the relevant state conversion, which enables online switching between virtual and simulated CPU modules as well as simulator checkpointing and restarting.

When fast-forwarding to a POI using the virtual CPU module, we execute at 90% of native speed on average across all SPEC CPU2006 benchmarks. This corresponds to a 2 100× performance improvement over the functional CPU module. The much higher execution rate enables us to fast-forward to any point within common benchmark suits in the matter of minutes instead of weeks.

4.2 Hardware-Accelerated Sampling Simulation

To make simulators usable for larger applications, many researchers have proposed methods to sample simulation [6, 37, 43, 44, 46]. With sampling, the simulator can run in a faster, less detailed mode between samples, and only spend time on slower detailed simulation for the individual samples. Design parameters such as sampling frequency, cache warming strategy, and fast forwarding method give the user the ability to control the trade-off between performance and accuracy to meet his or her needs. However, these proposals all depend on comparatively slow functional simulation between samples.

In Paper IV, we implement a sampling simulator inspired by the SMARTS [46] methodology. SMARTS uses three different modes of execution to balance accuracy and simulation overhead. The first mode, *functional warming*, is the fastest functional simulation mode and executes instructions without simulating timing, but still simulates caches and branch predictors to maintain long-lived microarchitectural state. This mode moves the simulator from one sample point to another and executes the bulk of the instructions. The second mode, *detailed warming*, simulates the entire system in detail using an out-of-order CPU model without sampling any statistics. This mode ensures that pipeline structures with short-lived state (e.g., load and store buffers) are in a representative, *warm*, state. The third mode, *detailed sampling*, simulates the

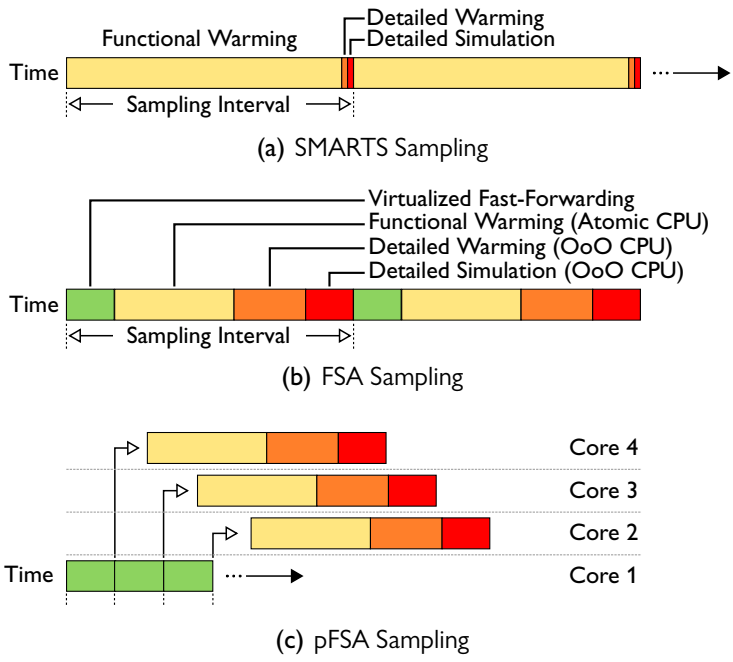


Figure 4.2: Comparison of how different sampling strategies interleave different simulation modes.

system in detail and takes the desired measurements. The interleaving of these simulation modes is shown in Figure 4.2(a).

SMARTS uses a technique known as *always-on* cache and branch predictor warming, which guarantees that these resources are warm when a sample is taken. This makes it trivial to ensure that the long-lived microarchitectural state (e.g., in caches and branch predictors) is warm. However, the overhead of always-on cache warming, which effectively prevents efficient native execution, is significant. We trade-off this guarantee for dramatic performance improvements (on the order of 500×) and demonstrate a technique that can be used to detect and estimate errors due to limited functional warming.

In traditional SMARTS-like sampling, the vast majority of the simulation time is spent in the functional warming mode [43, 46]. To reduce the overhead of this mode, we execute instructions natively on the host CPU using the virtual CPU module when the simulator is executing between samples. However, we cannot directly replace the functional warming mode with native execution, as it cannot warm the simulated caches and branch predictors. Instead, we add a new execution mode, *virtualized fast-forward*, which uses the virtual CPU module to execute between samples. After executing to the next sample in the virtualized

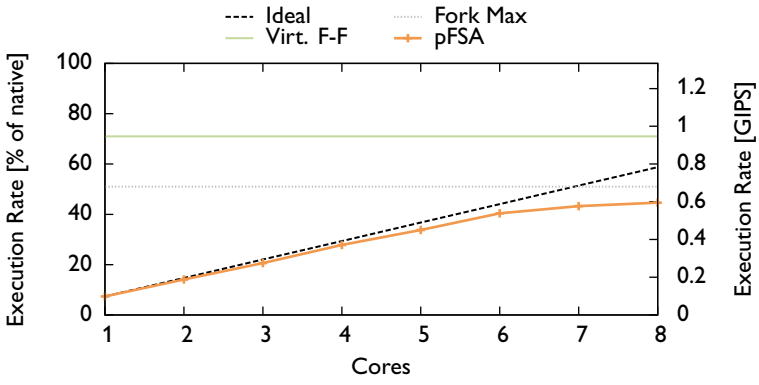


Figure 4.3: Simulator scalability when simulating 471.omnetpp using pFSA with 5 million instructions cache warming.

fast-forward mode, we switch to the functional warming mode, which now only needs to run long enough to warm caches and branch predictors. This enables us to execute the vast majority of our instructions at near native speed through hardware virtualization (Figure 4.2(b)). We call this sampling approach *Full Speed Ahead* (FSA) sampling.

When simulating using SMARTS, the simulation speed is limited by the speed of the functional simulation mode. In practice, our reference implementation of SMARTS in gem5 executes around 1 MIPS. Since FSA uses variable functional warming, its execution rate depends on the simulated cache configuration. Common cache configurations typically result in execution rates around 90–600 MIPS across the SPEC CPU2006 benchmarks.

4.3 Exploiting Sample-Level Parallelism

Despite executing the majority of the *instructions* natively, FSA still spends the majority of its *time* in the non-virtualized simulation modes (typically 75%–95%) to warm and measure sample points. This means that we can reach new sample points much faster than we can simulate a single sample point, which exposes parallelism between samples. To simulate multiple sample points in parallel, we need to do two things: copy the simulator state for each sample point (to enable them to execute independently), and advance the simulator to the next sample point before the previous samples have finished simulating (to generate parallel work). We implement such a parallel simulator by continuously running the simulator in the virtualized fast-forward mode, and cloning the simulator state using the host operating system's copy-on-write functionality

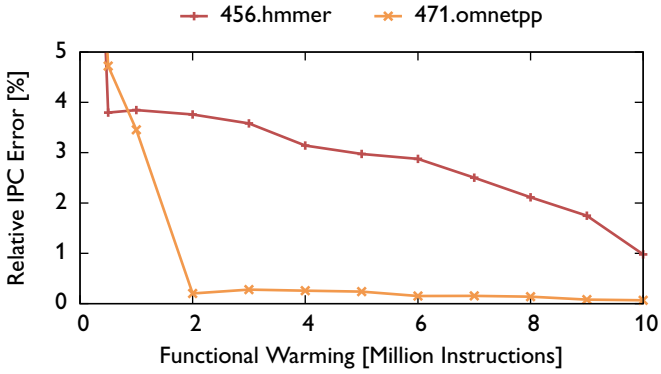


Figure 4.4: Estimated relative IPC error due to insufficient cache warming as a function of functional warming length for 456.hmmer and 471.omnetpp when simulating a system with a 2 MB L2 cache.

(using fork on UNIX) when we want to take a sample. We then simulate the cloned sample in parallel with the continued fast-forwarding of the original execution. We call this simulation mode *Parallel Full Speed Ahead* (pFSA) sampling. pFSA has the same execution modes as FSA, but unlike FSA the functional and detailed modes execute in parallel with the virtualized fast-forward mode (Figure 4.2(c)).

Despite the potentially large amount of state that needs to be copied for each sample, our parallel simulator scales well. Figure 4.3 shows the scalability of the simulator when simulating 471.omnetpp from SPEC CPU2006. This benchmark can be simulated at around 45% of its native execution speed when using eight cores. In this case, the fork overhead was estimated to limit the benchmark to roughly 50% of its native execution speed. In Paper IV, we evaluate the scalability of the sampling methodology when simulating systems on a large machine and show that almost linear scalability can be achieved up to at least 32 cores.

4.4 Estimating Warming Errors

Since FSA and pFSA use limited warming of long-lived architectural state (e.g., caches and branch predictors), there is a risk of insufficient warming, which can lead to incorrect simulation results. To detect and estimate the impact of limited warming, we devise an efficient simulation strategy that enables us to run detailed simulations for both the optimistic (sufficient warming) and pessimistic (insufficient warming) cases. We leverage our efficient state-copying mechanism to create a copy of

the simulator state before the detailed warming mode starts, which enables us to quickly re-run detailed warming and simulation without re-running functional warming. This results in a very small overhead since the simulator typically spends less than 10% of its execution time in the detailed modes. The difference between the pessimistic and optimistic cases gives us insight into the impact of functional warming.

We currently only support error estimation for caches, where the optimistic and pessimistic cases differ in the way we treat *warming misses*, i.e., misses that occur in sets that have not been fully warmed. In the optimistic case, we assume that all warming misses are actual misses (i.e., sufficient warming). This may underestimate the performance of the simulated cache as some of the misses might have been hits had the cache been fully warmed. In the pessimistic case, we assume that warming misses are hits (i.e., worst-case for insufficient warming). This overestimates the performance of the simulated cache since some of the hits might have been capacity misses. When sufficient warming has been applied, the difference between the two should be small. For example, Figure 4.4 shows the relative IPC difference between the two cases (relative IPC error) for two applications as a function of functional warming length. Applications have very different warming behavior due their access patterns. This highlights both the need for sufficient warming and the need to detect if warming was insufficient.

4.5 Summary

In Paper IV, we described an efficient method to fast-forward simulation at near-native speed (90% of native on average) using hardware virtualization. Our implementation extends the gem5 simulation system with a new virtual CPU module that integrates seamlessly with existing device models. This CPU module enables extremely rapid fast-forwarding to a point of interest in simulated applications. We showed how this capability enabled us to implement a highly efficient sampling simulator that maintains good accuracy (2.0% error on average). Due to the extremely rapid fast-forwarding, we can reach the next sample before the simulator finished simulating the previous sample. This exposes *sample-level parallelism* which we exploit to simulate multiple samples in parallel, reaching a simulation rate of up to 63% of native execution. We achieved a speedup of almost three orders of magnitude compared to functional simulation and around four orders of magnitude compared to detailed simulation.

5 Ongoing & Future Work

We are currently working on extensions and improvements to the fast simulation framework presented in Paper IV. The methods presented in this thesis enable extremely efficient simulation (around 400 MIPS for FSA and 2 GIPS for pFSA) of single-core systems with moderately sized caches. In order to efficiently simulate multicore systems, we need to add support for simulating multiple CPUs in a shared-memory configuration using the virtual CPU module. Additionally, to get good performance when simulating future systems with large caches, we need to improve the cache warming strategy. We are currently working on solutions to both of these issues.

5.1 Multicore System Simulation

The virtualization layer assumes that each simulated core runs in its own simulator thread when fast-forwarding a simulated multicore system. This means that devices and CPUs do not necessarily execute in the same thread, which leads to the following two challenges: First, we need to ensure that accesses to simulated devices are synchronized since a simulated core can request service from a device model in a different simulator thread. Second, timing gets more challenging since devices and CPUs live in different threads and simulated event queues.

A first step towards fast-forwarding of multicore systems using the virtual CPU module is to be able to fast-forward multiple systems communicating over a simulated network. This is simpler than true multicore simulation since each system has its own private device models (no synchronization is needed when accessing devices) and communication happens at well-defined places with long latency (easier to control skew between simulator threads). We have implemented support for multi-system simulation using gem5's support for quantum-based parallel discrete-event simulation. However, using quantum-based synchronization can lead to poor scalability since it requires synchronization between communicating threads more frequently than the shortest communication latency between them.

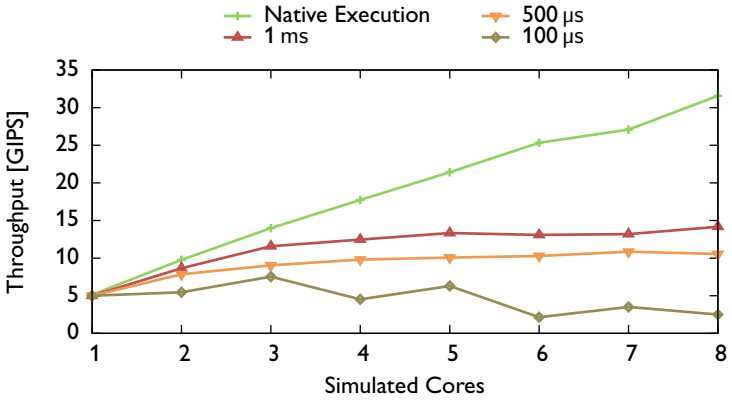


Figure 5.1: Performance (sum of per-core execution rates) of gem5 fast-forwarding a multicore system executing LU from splash2 using the virtual CPU module and relaxed barriers compared to native execution. The synchronization interval controls the trade-off between timing accuracy and speed.

We cannot rely on quantum-based synchronization when simulating multicore systems since this would incur a high synchronization cost and lead to poor scalability. We are currently extending the virtual CPU module to support multicore simulation using relaxed synchronization (similar to the relaxed barriers in Graphite [19]), where barriers are used to synchronize CPUs and devices less frequently than the shortest communication latency. This synchronization approach enables us to control the trade-off between maximum skew between threads and simulation performance without affecting functional correctness. Once these extensions are in place, we plan to extend FSA and pFSA with support for parallel workloads.

Initial performance experiments using the virtual CPU module suggest that we can fast-forward multicore systems at high speeds. Figure 5.1 shows the execution throughput (sum of per-core execution rates) of the parallel LU benchmark from SPLASH2 when running natively and when running in gem5 using the virtual CPU module with different synchronization intervals. Since the synchronization interval controls the maximum skew between simulated cores and devices, we need to ensure that it is low enough to maintain predictable timing (e.g., when delivering timer interrupts), but high enough to get acceptable performance. In practice, synchronizing every 500 µs seems to result in reliable interrupt delivery and a throughput of more than 10 GIPS (almost 35% of native execution) when running on eight cores. Fast-forwarding the system using functional simulation would result in a throughput of around 1 MIPS since gem5 does not (currently) support parallel functional simulation.

5.2 Efficient Cache Warming

When sampling in FSA, the majority of the *instructions* are executed natively. However, the simulator still spends the majority of its *time* warming caches. The time spent warming caches depends on the size of the simulated caches, which is expected to grow for future systems.

Nikoleris et al. [20] recently proposed an efficient method to warm caches using fast statistical cache models based on StatStack [10]. We are currently working on an implementation that collects the necessary input data while executing the system in the virtual CPU. In the case of large caches, the performance improvement is expected to be substantial. For example, when we simulated a 2 MB cache, we needed to execute in functional warming for 5 M instructions. When simulating an 8 MB cache, we had to increase the functional warming to 25 M instructions. Nikoleris et al. have shown that their method can be used to accurately simulate systems with caches of at least 256 MB while only applying 100 k instructions of warming. Assuming a 2× overhead for online profiling (StatStack profiling has been shown to be possible with less than 40% overhead [2, 33, 41]) and 100 k instructions of functional warming, we could potentially reach average simulation speeds of around 1 GIPS (a 10× improvement for 8 MB caches) on a single core for SPEC CPU2006 regardless of cache size.

6 Summary

In today's high-performance processors, an application's performance highly depends on how it uses the processor's cache hierarchy. In a multicore processor where the last-level cache is shared, a single application's cache behavior can affect the throughput of every core sharing the cache. In this thesis, we investigate methods to predict the behavior of applications sharing caches and how this impacts performance.

In Paper I, we demonstrated how low-overhead application profiles can be used to detect instructions that bring data that is unlikely to be reused into the cache and automatically modify the offending instructions to disable caching using existing hardware support. In order to reason about the effects of cache optimizations, we developed a *qualitative* application classification scheme. This classification scheme enables us to predict which applications are *affected by* cache contention and which are *causing cache contention*.

In order to *quantify* the performance of the shared last-level cache in a multicore processor, we need to understand how it is shared. In Paper II we demonstrated a cache model that uses high-level application profiles to predict the steady-state behavior of applications sharing a cache. The model predicts the amount of shared cache available to each of the applications, their individual performance, and the memory bandwidth required to reach that performance. When predicting cache interactions on existing systems, our model uses profiles that we measure with low overhead using existing hardware support. The same profiles can be produced using a simulator when modeling future systems.

In Paper III, we extended the cache sharing model to applications with time-varying behavior. When modeling such applications, it is no longer sufficient to look at average performance since the achieved performance is timing sensitive. Instead, we need to look at performance distributions. Generating such distributions using simulation, or even by running the applications on real hardware, has previously been too time consuming to be practical. Our cache sharing model makes it possible to generate them almost instantly.

Microarchitectural simulation is an important tool to understand the behavior of current and future processors. However, the overhead of

contemporary detailed simulators is prohibitive and often limits the experiments that can be performed. For example, in Paper I & II we used simulation in the evaluation or as an important part of the pre-studies leading to the research. This was no longer possible in Paper III due to the simulation overhead and the larger time-scales needed to simulate time-varying behaviors. Additionally, the methods from Paper II & III would benefit from a fast simulator since it would enabled efficient generation of application profiles when modeling future systems.

In Paper IV we showed how hardware virtualization, detailed simulation, and sampling can be combined to form an extremely fast simulation framework. In its most basic form, our framework enables us to fast-forward to a point of interest at 90% of the native execution rate of the host machine. This enables us to implement an efficient sampling simulator, where only a small number of instructions are simulated in detail and the vast majority of the instructions are executed natively. Since execution between samples is done natively, we reach new samples much faster than a single sample can be simulated. This exposes sample-level parallelism, which we exploit to further increase the performance of the simulator by simulating multiple samples in parallel. Our parallel sampling framework exhibits good scalability and scales almost linearly to approximately 60% of the native execution rate. In our experiments, that translated into an execution rate of 2 GIPS, which is a 19 000× improvement over detailed simulation alone. Compared to a gem5-based reference implementation of the SMARTS [46] sampling methodology, we achieve a 1 500× performance improvement.

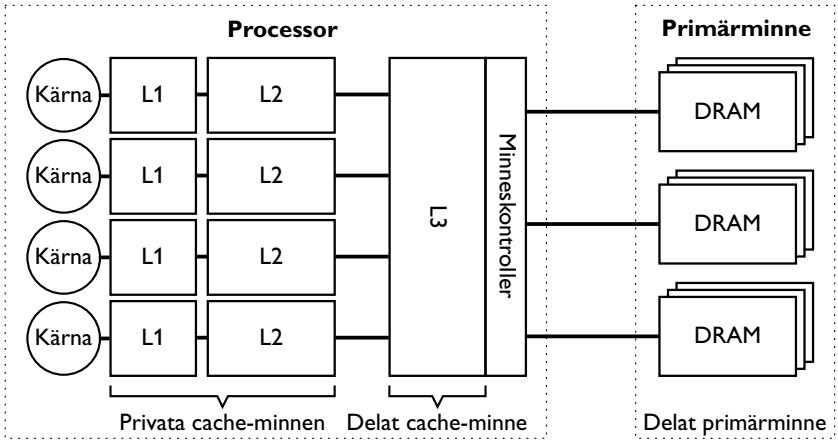
7 Svensk sammanfattning

För att kunna bygga och programmera effektiva datorsystem krävs detaljerad förståelse för hur olika designval påverkar hur systemet kommer att bete sig som en helhet. Den här avhandlingen fokuserar på olika metoder för att effektivt modellera hur program som delar resurser påverkar varandra i existerande datorsystem och hur denna påverkan kan minskas, samt hur prestanda i nya datorsystem kan analyseras på ett effektivt sätt.

7.1 Bakgrund

Utvecklingen av datorer har gått extremt fort sedan introduktionen av de första programmerbara elektroniska datorerna på 1940-talet. Inom en generation har vi upplevt en sällan skådad utveckling där datorer har gått från stora specialbyggda maskiner som fyller hela rum (t.ex. den brittiska Colossus som togs i drift 1944 för att knäcka det tyska Lorenz-kryptot) till dagens ständigt uppkopplade mobila enheter som ryms i fickan. Denna utveckling har möjliggjorts av upptäkter i fysik och elektronik som gett oss mycket små transistorer och integrerade kretsar. En modern processor kan rymma runt 2 miljarder transistorer, medan en tidig dator som Colossus innehöll motsvarande cirka 2 000 transistorer. I samband med att vi får mer och mer avancerade byggstenar och verktyg har också konsten att bygga datorer och datorsystem utvecklats till en egen disciplin, datorarkitektur, som i mycket kan liknas vid traditionell arkitektur. En datorarkitekt, likt en byggnadsarkitekt, sätter samman nya material och komponenter (transistorer, minnesceller och logiska grindar) till en helhet som bildar ett fullt fungerande system.

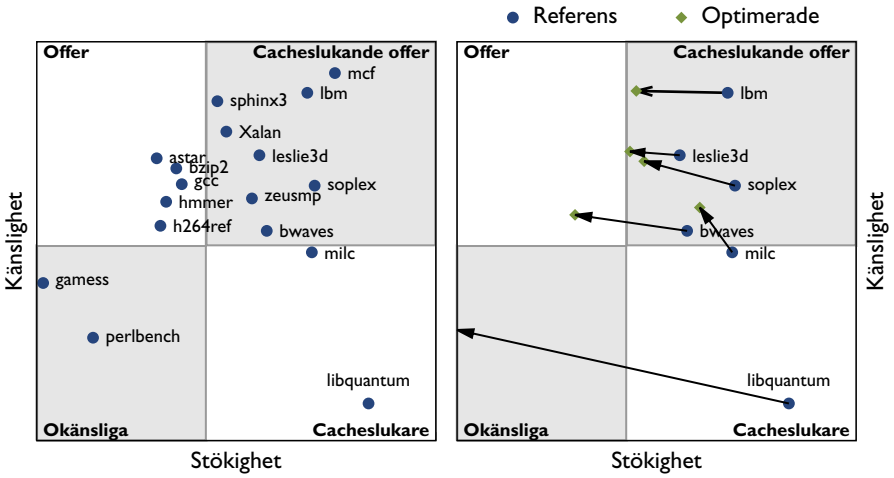
Den grundläggande funktionen hos en datorprocessor består av att hämta instruktioner och data från datorns primärminne samt att antingen manipulera data (binära tal) eller styra vilka instruktioner som ska köras i framtiden. Tiden det tar att köra en instruktion begränsas av två olika faktorer; tiden det tar att hämta data och instruktioner från primärminnet samt tiden det tar att utföra beräkningar. Hastigheten på dagens processorer begränsas oftast av tiden det tar att komma åt datorns primärminne. Det gör att datorarkitekter ofta väljer att använda en stor andel av transistorerna i en processor till att göra minnesåtkomster så effektiva



Figur 7.1: En modern processor innehåller flera olika kärnor som kan köra oberoende instruktioner. För att minska åtkomsttiden till det långsamma primärminnet (s.k. DRAM som nås via en minneskontroller) används en hierarki av små snabba cache-minnen (L1–L3). Till skillnad från enkelkärniga processorer delas delar av processorn (här L3 och minneskontrollern) mellan kärnorna, även om kärnorna beter sig som självständiga processorer.

som möjligt. För att minska tiden det tar att läsa och skriva till minnet använder man en hierarki av små minnen (cache-minnen) som är olika snabba. De snabbaste, men minst energi- och platseffektiva, sitter närmast de delar av processorn som utför själva beräkningarna, medan långsammare, men mer energi- och platseffektiva, minnen sitter längre från processorn. Detta gör att data som används ofta lagras närmare de delar av processorn som använder dem och därför kan hämtas och manipuleras snabbare än annan data.

I takt med att transistorerna blev fler och fler har det blivit svårare och svårare att utnyttja dem effektivt. I början av 2000-talet insåg man att det inte längre gick att göra en enskild processor snabbare på ett effektivt sätt. Istället valde man att integrera flera processorer på samma krets. Sådana ”processorer i processorn” kallas normalt kärnor (eng. *core*) och bildar tillsammans en flerkärnig (eng. *multicore*) processor. Strukturen på en sådan processor illustreras i Figure 7.1. I figuren ses fyra oberoende processorkärnor som har två privata cache-minnen var (L1 & L2) och ett delat cache-minne och primärminne. En processors olika kärnor kör instruktioner helt oberoende av varandra, men kan kommunicera genom datorns minne. Det betyder i praktiken att varje kärna kan köra ett eget program eller att ett program kan använda flera kärnor för att köra fler beräkningar samtidigt. Till skillnad från tidigare processorer är nu bl.a. cache-minnena och anslutningen till minnet delade mellan olika program som kör samtidigt. Det medför att två olika program som körs oberoende



(a) Innan optimering (b) Efter optimering

Figur 7.2: Klassificering med avseende på användning av delad cache för några vanliga testprogram. De olika axlarna visar hur känsliga (y-axeln) programmen är för konkurens och hur mycket de påverkar andra program (x-axeln). Program som slukar mycket cache, höger sida av (a), kan göras snällare (b) genom att använda optimeringen vi föreslog i Artikel I.

av varandra på två olika kärnor kan påverka varandras prestanda genom att de konkurrerar om delade resurser i processorn. Att använda dessa resurser effektivt och förstå hur de delas är alltså av stor vikt för att förstå hur en modern processor beter sig.

7.2 Sammanfattning av forskningen

När man analyserar ett programs beteende med avseende på resursdelning är det helt avgörande att både förstå hur programmet *påverkar* andra och hur det *påverkas av* andra. I den här avhandlingen, som består av fyra olika delarbeten, har vi undersökt hur programs beteende med fokus på resursdelning och hur information om enskilda instruktioners beteende kan användas för att optimera hur programmet beter sig. I avhandlingen presenteras fyra olika delarbeten som fokuserar på olika aspekter kring hur detta kan modelleras effektivt.

I Artikel I fokuserar vi framförallt på att *kvalitativt* beskriva vilka program som *påverkas av* att köra med andra och vilka som *påverkar* andra. Figur 7.2(a) visar hur klassificeringen ser ut för en uppsättning vanliga testprogram. En programs position i x-led beskriver hur mycket den *påverkar* ("stökighet") andra program, medan positionen i y-led indikerar

hur den *påverkas av* (”känslighet”) andra program. Man kan till exempel utläsa att programmet libquantum har en tendens att kraftigt påverka andra program, men påverkas inte av andra i någon större utsträckning.

Den statistiska modellen ger även information per instruktion. Utöver att klassificera program använder vi denna information för att påverka i var i cache-hierarkin data får befinna sig. Detta gör att vi kan ändra många programs beteende och minska påverkan på andra. Detta kan ses i Figure 7.2(b) där libquantum har gått från att vara en stökig cacheslukare som bråkar med andra program till att vara relativt välartad.

Eftersom Artikel I enbart behandlar programs *kvalitativa* beteende ställde vi oss frågan om vi kan *kvantifiera* detta och förutsäga programs faktiska beteende. I Artikel II utvecklar vi en modell som kvantifierar effekterna av cache-delning mellan program som kör på olika kärnor. Istället för att bara förutsäga vilka program som kan ställa till problem för andra program kan vår nya modell avgöra hur mycket cache varje program får tillgång till. Dessutom kan vi förutsäga vad det innebär för programmens prestanda och hur det påverkar datamängden som måste hämtas från primärminnet. För att svara på detta använder vår modell programprofiler som vi kan mäta på riktiga datorsystem.

I Artikel II förutsatte vi att program har ett stabilt beteende som är tidsoberoende. För att göra användbara förutsägelser på realistiska program utökar vi i Artikel III vår cache-delningsmodell till tidsberoende program. Den grundläggande fråga vi svarar på i detta fall är hur cachen fördelas över tid. Det visar sig här att det egentligen inte rätt fråga att ställa när beteendet varierar eftersom en liten skillnad i hur programmen som körs (t.ex. om de inte startas samtidigt) kraftigt kan påverka deras beteende. Därför undersöker vi även hur mycket ett programs prestanda *varierar* när det delar cache med ett annat program. Det enklaste sättet undersöka detta på vore att köra de program man är intresserad av samtidigt flera gånger och mäta hur de beter sig, men detta är oftast för tidskrävande (speciellt när antalet program är stort) för att vara praktiskt. Vår modell beräknar samma information från körprofiler, som vi kan mäta på existerande system, av programmen på under en hundradel av tiden att köra dem.

I de inledande studierna till både Artikel I & II använde vi en relativt detaljerad *simulator*[1] för att djupare förstå systemet vi försökte modellera. I båda fallen såg vi att detaljerad simulering är mycket tidskrävande, den totala simuleringstiden (mätt i tid det skulle ta för *en* kärna i *en* processor att göra alla simuleringar) som har används för artiklarna i den här avhandlingen är uppskattningsvis minst 5 år. I Artikel III var det inte längre möjligt att verifiera modellen med en simulator eftersom det

[1]En simulator är i det här fallet ett program som simulerar ett helt datorsystem, inklusive komplicerade interaktioner i själva processorn.

skulle ha tagit för lång tid. I praktiken begränsas alltså vilka experiment vi kan köra av simulatorn. Detta ledde till Artikel IV där vi undersöker metoder för att effektivisera simulering. I denna studie presenterar vi en snabb simuleringsmetod där vi uppskattar ett systems beteende genom att simulera en liten del av instruktionerna i ett program i detalj. Genom att köra majoriteten av instruktionerna på den riktiga processorn istället för den simulerade kan vi uppskatta det simulerade systemets beteende mycket snabbare än tidigare. Eftersom inte alla instruktioner simuleras introducerar denna typ av simulering två nya felkällor: stickprovsfel och fel på grund av felaktig data i stora strukturer som cacher. Tidigare har man undvikit att göra denna typ av simulering eftersom man inte har kunnat kvantifiera hur resultatet av cache-simuleringen påverkas på ett effektivt sätt. I den här studien visar vi hur felet effektivt kan kvantifieras och hur man kan parallellisera simularingen. Jämfört med detaljerad simulering innebär vår metod nästan 19 000× förbättring, vilket möjliggör helt nya typer av studier.

8 Acknowledgments

Many have played important parts in shaping me into the more independent researcher I am today. I would like to start by thanking my advisor, Erik Hagersten, who guided and focused a general interest in computer architecture from when I was a student in his computer architecture class to when I was an employee in his start-up company, and finally as a PhD student in his research group. You have taught me many things about computer architecture in general and memory systems in particular. I look forward to sailing with you again, Captain! My co-advisor, David Black-Schaffer, for interesting discussions and excellent advice on how to present research. Our research would never have been as understandable, and colorful, if it were not for the countless hours you spent revising article drafts and presentations.

I would also like to thank all the members of the computer architecture group for interesting discussion and ideas. David Eklöv for encouraging me to think critically about my own research. Nikos Nikoleris for interesting technical discussions. Andreas Sembrant and Muneeb Khan for fun and interesting collaborations. Stefanos Kaxiras for interesting discussions about coherence and power. Nina Shariati Nilsson for teaching me new things about integrated circuits.

This thesis would have been considerably worse without the feedback I got from various people, you know who you are, on numerous drafts. I would especially like to thank Magnus Själander for taking the time to provide useful and detailed feedback. By virtue of being the newest member of the group, he provided a much needed pair of fresh eyes. Jessica Bergman and Jonathan Alvarsson, who by being researchers in other fields, helped to make the Swedish popular science summary understandable to people without a CS degree. Thank you!

Research is not just hard work and dusty rooms full of machines that go "ping"; it is largely a social activity where new ideas are born and old ideas die. Not all of these social interactions have lead to formal research collaborations, but they have none the less been interesting and, most importantly, *fun*. I would like to thank Alexandra Jimborean, Germán Ceballos, Konstantinos Koukos, Moncef Mechri, Ricardo Alves, and Vasileios Spiliopoulos for fun and geeky discussions. Special thanks go to

Xiaoyue Pan for acting as the social glue across the division and arranging the yearly Chinese New Year's party. It has been fun!

The research presented in this thesis was made possible through generous financial support from the CoDeR-MP project and the UPMARC research center. Computational infrastructure for reference simulations was provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX), where several CPU year's worth of simulations chewed away in the background while we were having coffee. Initial work on hardware virtualization support in gem5 was sponsored by ARM, where I would especially like to thank Matthew Horsnell, Ali G. Saidi, Andreas Hansson, Marc Zyngier, and Will Deacon for interesting discussions and insights. I am looking forward to working with you again!

There are others who, while not having had a direct impact on the research itself, have made it possible. I would like to start by thanking my parents and my grandparents, who treated me like their own son, for encouraging me to ask questions about the world and fostering an inquisitive mind. And not to mention for putting up with that inquisitive mind when it got *too* inquisitive, the years between learning to take things apart and learning to put them back together must have been challenging… A very special *thank you* goes to Jessica for her infinite love and support, and for standing my somewhat irregular working hours throughout the last year of my PhD. I would also like to thank the rest of my family and all of my friends, you are all very important to me.

Last, but not least, I would like to thank *Coffea arabica*, the humble coffee bean, for making (this) research possible.

9 References

[1] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. "COTSon: Infrastructure for Full System Simulation". In: *ACM SIGOPS Operating Systems Review* 43.1 (Jan. 2009), pp. 52–61. DOI: 10.1145/1496909.1496921.

[2] Erik Berg and Erik Hagersten. "Fast Data-Locality Profiling of Native Execution". In: *ACM SIGMETRICS Performance Evaluation Review* 33.1 (2005), pp. 169–180. DOI: 10.1145/1071690.1064232.

[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)*. 2004, pp. 20–27. DOI: 10.1109/ISPASS.2004.1291352.

[4] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. "The gem5 Simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011). DOI: 10.1145/2024716.2024718.

[5] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2005. DOI: 10.1109/HPCA.2005.27.

[6] Shelley Chen. "Direct SMARTS: Accelerating Microarchitectural Simulation through Direct Execution". MA thesis. Carnegie Mellon University, 2004.

[7] Xi E. Chen and Tor M. Aamodt. "A First-Order Fine-Grained Multithreaded Throughput Model". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2009. DOI: 10.1109/HPCA.2009.4798270.

[8] Xi E. Chen and Tor M. Aamodt. "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors". In: *IEEE Transactions on Computers* PP.99 (2011). DOI: 10.1109/TC.2011.141.

[9] David Eklov, David Black-Schaffer, and Erik Hagersten. "Fast Modeling of Cache Contention in Multicore Systems". In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011, pp. 147–157. DOI: 10.1145/1944862.1944885.

[10] David Eklov and Erik Hagersten. "StatStack: Efficient Modeling of LRU Caches". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2010, pp. 55–65. DOI: 10.1109/ISPASS.2010.5452069.

[11] 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)*. 2011, pp. 165–175. DOI: 10.1109/ICPP.2011.15.

[12] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. "Combining Simulation and Virtualization through Dynamic Sampling". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Apr. 2007, pp. 72–83. DOI: 10.1109/ISPASS.2007.363738.

[13] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Jr Simon Steely, and Joel Emer. "Adaptive Insertion Policies for Managing Shared Caches". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 208–219. DOI: 10.1145/1454115.1454145.

[14] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)". In: *Proc. International Symposium on Computer Architecture (ISCA)*. 2010, pp. 60–71. DOI: 10.1145/1815961.1815971.

[15] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. "A Case for Resource Efficient Prefetching in Multicores". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2014, pp. 137–138.

[16] Avi Kivity, Uri Lublin, and Anthony Liguori. "kvm: the Linux Virtual Machine Monitor". In: *Proc. Linux Symposium*. 2007, pp. 225–230.

[17] John D. C. Little. “A Proof for the Queuing Formula: $L = \lambda W$”. In: *Operations Research* 9.3 (1961), pp. 383–387.

[18] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. “Evaluation techniques in storage hierarchies”. In: *IBM Journal of Research and Development* 9.2 (1970), pp. 78–117.

[19] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. “Graphite: A Distributed Parallel Simulator for Multicores”. In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[20] 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)*. 2014.

[21] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. “Instruction-based Reuse-Distance Prediction for Effective Cache Management”. In: *Proc. Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. 2009, pp. 49–58. DOI: 10.1109/ICSAMOS.2009.5289241.

[22] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. “Adaptive Insertion Policies for High Performance Caching”. In: *Proc. International Symposium on Computer Architecture (ISCA)*. San Diego, California, USA: ACM, 2007, pp. 381–391. DOI: 10.1145/1250662.1250709.

[23] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers”. In: *ACM SIGMETRICS Performance Evaluation Review* 21.1 (June 1993), pp. 48–60. DOI: 10.1145/166962.166979.

[24] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. “Complete Computer System Simulation: The SimOS Approach”. In: *Parallel & Distributed Technology: Systems & Applications* 3.4 (Jan. 1995), pp. 34–43. DOI: 10.1109/88.473612.

[25] Frederick Ryckbosch, Stijn Polfliet, and Lieven Eeckhout. “VSim: Simulating Multi-Server Setups at Near Native Hardware Speed”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8 (2012), 52:1–52:20. DOI: 10.1145/2086696.2086731.

[26] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. "A Simple Statistical Cache Sharing Model for Multicores". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2011, pp. 31–36.

[27] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. "Efficient Techniques for Predicting Cache Sharing and Throughput". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 305–314. DOI: 10.1145/2370816.2370861.

[28] Andreas Sandberg, David Eklöv, and Erik Hagersten. "A Software Technique for Reducing Cache Pollution". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2010, pp. 59–62.

[29] Andreas Sandberg, David Eklöv, 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)*. 2010. DOI: 10.1109/SC.2010.44.

[30] Andreas Sandberg, Erik Hagersten, and David Black-Schaffer. *Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed*. Tech. rep. 2014-005. Department of Information Technology, Uppsala University, Mar. 2014.

[31] Andreas Sandberg and Stefanos Kaxiras. "Efficient Detection of Communication in Multi-Cores". In: *Proc. Swedish Workshop on Multi-Core Computing (MCC)*. 2009, pp. 119–121.

[32] Andreas Sandberg, Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Modeling Performance Variation Due to Cache Sharing". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2013, pp. 155–166. DOI: 10.1109/HPCA.2013.6522315.

[33] 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)*. 2012, pp. 175–185. DOI: 10.1145/2259016.2259040.

[34] Andreas Sembrant, David Eklov, and Erik Hagersten. "Efficient Software-based Online Phase Classification". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011, pp. 104–115. DOI: 10.1109/IISWC.2011.6114207.

[35] Rathijit Sen and David A. Wood. "Reuse-based Online Models for Caches". In: *ACM SIGMETRICS Performance Evaluation Review* 41.1 (June 2013), pp. 279–292. DOI: 10.1145/2494232.2465756.

[36] Timothy Sherwood, Brad Calder, and Joel Emer. "Reducing Cache Misses Using Hardware and Software Page Placement". In: *Proc. International Conference on Supercomputing (ICS)*. Rhodes, Greece, 1999, pp. 155–164. DOI: 10.1145/305138.305189.

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

[38] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. "A Modified Approach to Data Cache Management". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 1995, pp. 93–103. DOI: 10.1109/MICRO.1995.476816.

[39] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004, pp. 45–56. DOI: 10.1109/ISPASS.2004.1291355.

[40] Kenzo Van Craeynest and Lieven Eeckhout. "The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011, pp. 26–37. DOI: 10.1109/IISWC.2011.6114194.

[41] Peter Vestberg. "Low-Overhead Memory Access Sampler: An Efficient Method for Data-Locality Profiling". MA thesis. Uppsala University, 2011.

[42] Zhenlin Wang, McKinley Kathryn S., Arnold L. Rosenberg, and Weems Charles C. "Using the Compiler to Improve Cache Replacement Decisions". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2002, pp. 199–208. DOI: 10.1109/PACT.2002.1106018.

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

[44] 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.

[45] Wayne A. Wong and Jaen-Loup Baer. "Modified LRU Policies for Improving Second-Level Cache Behavior". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2000, pp. 49–60. DOI: 10.1109/HPCA.2000.824338.

[46] 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)*. 2003, pp. 84–95. DOI: 10.1109/ISCA.2003.1206991.

[47] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. "All-Window Profiling and Composable Models of Cache Sharing". In: *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2011. DOI: 10.1145/1941553.1941567.

[48] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. "Cache Contention and Application Performance Prediction for Multi-Core Systems". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452065.

[49] Matt T. Yourst. "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Apr. 2007, pp. 23–34. DOI: 10.1109/ISPASS.2007.363733.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1136

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-220652

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2014

PAPER I

Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses

Andreas Sandberg
David Eklöv
Erik Hagersten

©2010 IEEE. Reprinted, with permission, from *SC'10*, November 13–19, 2010.
DOI: 10.1109/SC.2010.44

Abstract — Contention for shared cache resources has been recognized as a major bottleneck for multicores—especially for mixed workloads of independent applications. While most modern processors implement instructions to manage caches, these instructions are largely unused due to a lack of understanding of how to best leverage them.

This paper introduces a classification of applications into four cache usage categories. We discuss how applications from different categories affect each other's performance indirectly through cache sharing and devise a scheme to optimize such sharing. We also propose a low-overhead method to automatically find the best per-instruction cache management policy.

We demonstrate how the indirect cache-sharing effects of mixed workloads can be tamed by automatically altering some instructions to better manage cache resources. Practical experiments demonstrate that our software-only method can improve application performance up to 35% on x86 multicore hardware.

1 Introduction

The introduction of multicore processors has significantly changed the landscape for most applications. The literature has mostly focused on parallel multithreaded applications. However, multicores are often used to run several independent applications. Such *mixed workloads* are common in a wide range of systems, spanning from cell phones to HPC servers. HPC clusters often run a large number of serial applications in parallel across their physical cores. For example, parameter studies in science and engineering where the same application is run with different input data sets.

When an application shares a multicore with other applications, new types of performance considerations are required for good system throughput. Typically, the co-scheduled applications share resources with limited capacity and bandwidth, such as a shared last-level cache (SLLC) and DRAM interfaces. An application overusing any of these resources can degrade the performance of the other applications sharing the same multicore chip.

Consider a simple example: Application *A* has an active working set that barely fits in the SLLC, and application *B* makes a copy of a data structure much larger than the SLLC. When run together, *B* will use a large portion of the SLLC and will force *A* to miss much more often than when run in isolation. Fortunately, most libraries implementing memory copying routines, e.g. memcpy, have been hand-optimized and use special *cache-bypass* instructions, such as non-temporal reads and writes.

On most implementations, these instructions will avoid allocation of resources in the SLLC and subsequently will not force any replacements of application *A*'s working set in the cache.

In the example above the use of cache bypass instructions may seem obvious, and hand-tuning a common routine, such as memcpy, may motivate the use of special assembler instructions. However, many common programs also have memory accesses that allocate data with little benefit in the SLLC and may slow down co-scheduled applications. Detecting such reckless use is beyond the capability of most application programmers, as is the use of assembly coding. Ideally, both the detection and cache-bypassing should be done automatically using existing hardware support.

Several software techniques for managing caches have been proposed in the past [9, 11–13]. However, most of these methods require an expensive simulation analysis. These techniques assume the existence of heavily specialized instructions [11, 12], or extensions to the cache state and replacement policy [13], none of which can be found in today's processors. Several researchers have proposed hardware improvements to the LRU replacement algorithm [3, 4, 6, 7]. In general, such algorithms tweak the LRU order by including additional predictions about future re-references. Others have tried to predict [14] and quantify [10] interference due to cache sharing.

In this paper, we propose an efficient and practical software-only technique to automatically manage cache sharing to improve the performance of mixed workloads of common applications running on existing x86 hardware. Unlike previously proposed methods, our technique does not rely on any hardware modifications and can be applied to existing applications running on commodity hardware. This paper makes the following contributions:

- We propose a scheme to classify applications according to their impact, and dependence, on the SLLC.
- We propose an automatic and low-overhead method to find instructions that use the SLLC recklessly and automatically introduce cache bypass instructions into the binary.
- We demonstrate how this technique can change the classification of many applications, making them better mixed workload citizens.
- We evaluate the performance gain of the applications and show that their improved behavior is in agreement with the classification.

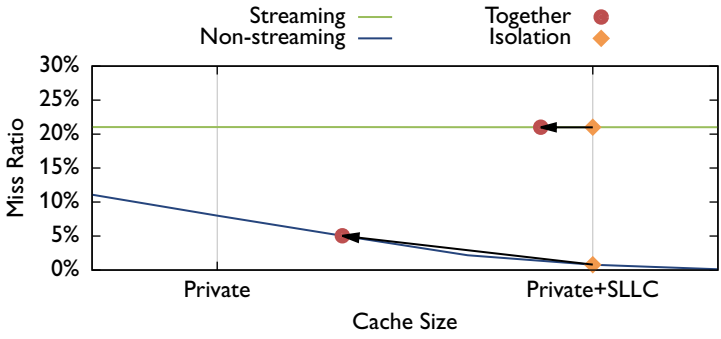


Figure 1: Miss ratio as a function of cache size for an application with streaming behavior and a typical non-streaming application that reuses most of its data. When run in isolation, each application has access to both the private cache and the entire SLLC. Running together causes the non-streaming application to receive a small fraction of the SLLC, while the streaming application receives a large fraction without decreasing its miss ratio. The change in perceived cache size and miss ratio is illustrated by the arrows.

2 Managing caches in software

Application performance on multicores is highly dependent on the activities of the other cores in the same chip due to contention for shared resources. In most modern processors there is no explicit hardware policy to manage these shared resources. However, there are usually instructions to manage these resources in software. By using these instructions properly, it is possible to increase the amount of data that is reused through the cache hierarchy. However, this requires being able to predict which applications, and which instructions, benefit from caching and which do not.

In order to know which application would benefit from using more of the shared cache resources, we need to know the applications' cache usage characteristics. The cache miss ratio as a function of cache size, i.e. the number of cache misses as a fraction of the total number of memory accesses as a function of cache size, is a useful metric to determine such characteristics. Figure 1 shows the miss ratio curves for a typical streaming application and an application that reuses its data. The miss ratio of the non-streaming application decreases as the amount of available cache increases. This occurs because more of the data set fits in the cache. Since the streaming application does not reuse its data, the miss ratio stays constant even when the cache size is increased.

When the applications are run in isolation, they will get access to both the core-local private cache and the SLLC. Assuming that the cache hierarchy is exclusive, the amount of cache available to an application run-

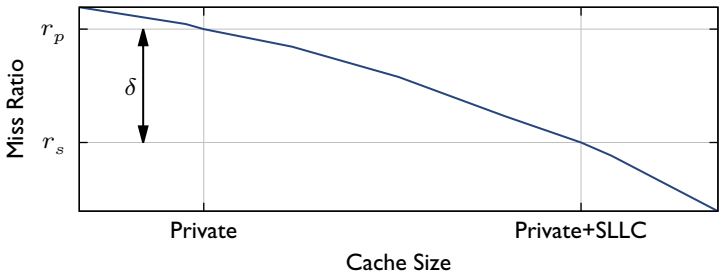


Figure 2: A generalized miss ratio curve for an application. The minimum, i.e. only the private cache, and the maximum, i.e. the private cache and the full shared cache, amount of cache available to an application are shown on the x-axis. The miss ratio when running in isolation (r_s) is the smallest miss ratio that an application can achieve on this system, while the miss ratio when running only in the private cache (r_p) is the worst miss ratio. The δ represents how much an application is affected by competition for the shared cache.

ning in isolation is the sum of the private cache and the SLLC. When two applications share the cache, they will perceive the SLLC as being smaller. In the case illustrated by Figure 1, the streaming application misses much more frequently than the non-streaming application. The frequent misses causes the streaming application to install more data in the cache than the non-streaming application. The non-streaming application will therefore perceive the cache as being much smaller than when run in isolation. The change in perceived cache size, and how this affects miss ratio is illustrated by the arrows in Figure 1.

Decreasing the perceived cache size for the streaming application does not affect its miss ratio. The non-streaming application, however, sees an increased miss ratio when access to the SLLC is restricted. As the number of misses increase, the bandwidth requirements also increase, which affects the performance of all the applications sharing the same memory interface. If we could make sure that the streaming application does not install any of its streaming data into the cache, the miss ratio, and bandwidth requirement, of the non-streaming applications would decrease without sacrificing any performance. In fact, the streaming application would run faster since the total bandwidth requirement would be decreased.

Using the miss ratio curves we can classify applications based on how they affect others and how they are affected by competition for the shared cache. We base this classification on the base miss ratio, r_s, when the application is run in isolation and has access to both its private cache

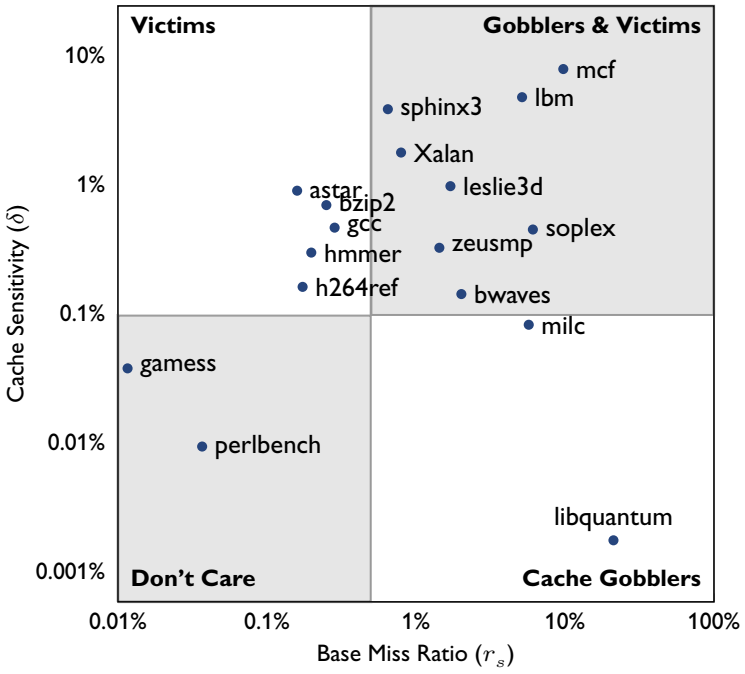


Figure 3: Classification map of a subset of the SPEC2006 benchmarks running with the reference input set on a system with a 576 kB private cache and 6 MB shared cache. The quadrants signify different behaviors when running together with other applications. Applications to the left tend to reuse almost all of their data in the shared cache and generally work well with other applications, applications to the right tend to use large parts of the shared cache for data that is never reused and are generally troublesome in mixes with other applications. Applications in the upper half are sensitive to the amount of data that can be stored in the shared cache, while applications on the bottom are insensitive.

and the entire SLLC, and the miss ratio, r_p, when it only has access to the private cache, see Figure 2. The r_p miss ratio can be thought of as the maximum miss ratio that an application can get due to cache contention, while r_s is the ideal case when the application is run in isolation. To capture the sensitivity to cache contention we define the cache sensitivity, δ, to be the difference between the two miss ratios. A large δ indicates that an application benefits from using the shared cache, while a small δ means that the application exhibits streaming behavior and does not benefit from additional cache resources.

Using the r_s and δ we can classify applications based on how they use the cache. This classification allows us to predict how applications will

affect each other and how the system will be affected by software cache management. We define the following categories:

Don't Care
: Small r_s and small δ—Applications that are largely unaffected by contention for the shared cache level. These applications fit their entire data set in the private cache, they are therefore largely unaffected by contention for the shared cache and memory bandwidth.

Victims
: Small r_s and large δ—Applications that suffer badly if the amount of cache at the shared level is restricted. The data they manage to install in the shared resource is almost always reused. Applications with a working set larger than the private cache, but smaller than the total cache size belong in this category.

Gobblers & Victims
: Large r_s and large δ—Applications that suffer from SLLC cache contention, but store large amounts of data that is never reused in the shared cache. For example, applications traversing a small and a large data structure in parallel may reuse data in the cache when accessing the small structure, while accesses to the large data structure always miss. Disabling caching for the accesses to the large data structure would allow more of the smaller data structure to be cached. Managing the cache for these applications is likely to improve throughput, both when they are running in isolation and in a mix with other applications.

Cache Gobblers
: Large r_s and small δ—Applications that do not benefit from the shared cache at all, but still install large amounts of data in it. Applications in this category work on streaming data or data structures that are much larger than the cache. These applications are good candidates for software cache management. Since they do not reuse the data they install in the shared cache, their throughput is generally not improved when running in isolation. Managing these applications will improve the full system throughput by allowing applications from other categories to use more of the shared cache.

Figure 3 shows the classification of several SPEC2006 benchmarks according to these categories. Applications classified as wasting cache resources, i.e. applications on the right-hand side of the map, are obvious targets for cache management. The large base miss ratio in such applications is due to memory accesses that touch data that is never reused

while it resides in the cache. Disabling caching for such instructions does not introduce new misses since data is not reused, instead it will free up cache space for other accesses.

3 Cache management instructions

Most modern instruction sets include instructions to manage caches. These instructions can typically be classified into three different categories: *non-temporal memory accesses*, *forced cache eviction* and *non-temporal prefetches*. Many processors support at least one of these instruction classes. However, their semantics may not always make them suitable for cache management for performance.

Examples from the first category are the memory accesses in the PA-RISC which can be annotated with caching hints, e.g. only spatial locality or write only. Similar instruction annotations exist for Itanium. Other instruction sets, such as some of the SIMD extensions to the x86, contain completely separate instructions for handling non-temporal data. The hardware may, based on these hints, decide not to install write-only cache lines in the cache and use write-combining buffers instead. Non-temporal reads can be handled using separate non-temporal buffers or by installing the accessed cache line in such a way that it is the next line to be evicted from a set.

Instructions from the second category, *forced cache eviction*, appear in some form in most architectures. However, not all architectures expose such instructions to user space. Yet other implementations may have undesired semantics that limit their usefulness in code optimizations, e.g. the x86 *Flush Cache Line* (`CLFLUSH`) instruction forces *all* caches in a coherence domain to be flushed. There are some architectures that implement instructions in this class that are specifically intended for code optimizations. For example, the Alpha ISA specifies an instruction, *Evict Data Cache Block* (`ECB`), that gives the memory system a hint that a specific cache line will not be reused in the near future. A similar instruction, *Write Hint* (`WH64`), tells the memory subsystem that an entire cache line will be overwritten before being read again, this allows the memory system to allocate the cache line without actually reading its old contents. The `ECB` and `WH64` instructions are in many ways similar to the caching hints in the previous category, but instead of annotating the load or store instruction, the hints are given after or, in case of a store, before the memory accesses in question.

The third category, non-temporal prefetches, is also included in several different ISAs. The SPARC ISA has both read and write prefetch

variants for data that is not temporally reused. Similar prefetch instructions are also available in both Itanium and x86. Some implementations may choose to prefetch into the cache such that the fetched line is the next to be evicted from that set; others may prevent the data from propagating from the L1 to a higher level in the cache hierarchy.

In the remainder of this paper, we will assume an architecture with a non-temporal hint that is implemented such that non-temporal data is fetched into the L1 cache, but never installed in higher levels. This is how the AMD system we target implement support for non-temporal prefetches.

4 Low-overhead cache modeling

A natural starting point for modeling LRU caches is the *stack distance* [5]. A stack distance is the number of unique cache lines accessed between two successive memory accesses to the same cache line. It can be directly used to determine if a memory access results in a cache hit or a cache miss for a fully-associative LRU cache: if the stack distance is less than the cache size, the access will be a hit, otherwise it will miss. Therefore, the stack distance distribution enables the application's miss ratio to be computed for any given cache size, by simply computing the fraction of memory accesses with a stack distances greater than the desired cache size.

In this work, we need to differentiate between what we call backward and forward stack distance. Let A and B be two successive memory accesses to the same cache line. Suppose that there are S unique cache lines accessed by the memory accesses executed between A and B. Here, we say that A has a forward stack distance of S, and that B has a backward stack distance of S.

Measuring stack distances is generally very expensive. In this paper, we use StatStack [2] to estimate stack distances and miss ratios. StatStack is a statistical cache model that models fully associative caches with LRU replacement. Modeling fully associative LRU caches is, for most applications, a good approximation of the set associative pseudo LRU caches implemented in hardware. StatStack estimates an application's stack distances using only a sparse sample of the application's *reuse distances*, i.e. the number of memory accesses performed between two accesses to the same cache line. This approach to modeling caches has been shown to be several orders of magnitude faster than full cache simulation, and almost as accurate. The runtime profile of an application can be collected with an overhead of only 40% [1], and the execution time of the cache model is only a few seconds [2].

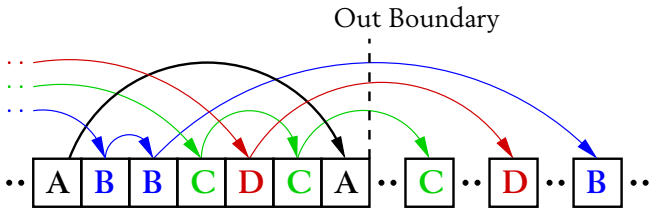


Figure 4: Reuse distance in a memory access stream. The arcs connect successive memory accesses to the same cache line, and represents the reuse of cache lines. The stack distance of the second memory access to *A* is equal to the number of arcs that cross "Out Boundary".

To understand how StatStack works, consider the access sequence shown in Figure 4. Here the arcs connect subsequent accesses to the same cache line, and represent the reuse of data. In this example, the second memory access to cache line *A* has a reuse distance of five, since there are five memory accesses executed between the two accesses to *A*, and a backward stack distance of three, since there are three unique cache lines (*B*, C and *D*) accessed between the two accesses to *A*. Furthermore, we see that there are three arcs that cross the vertical line labeled "Out Boundary", which is the same as the stack distance of the second access to *A*. This observation holds true in general. Based on it we can compute the stack distance of any memory access, given that we know the reuse distances of all memory access performed between it and the previous access to the same cache line.

The input to StatStack is a sparse reuse distance sample that only contains the reuse distances of a sparse random selection of an application's memory accesses, and therefore does not contain enough information for the above observation to be directly applied. Instead, StatStack uses the reuse distance sample to estimate the application's reuse distance distribution. This distribution is then used to estimate the likelihood that a memory access has a reuse distance greater than a given length. Since the length of a reuse distance determines if its outbound arc reaches beyond the "Out Boundary", we can use these likelihoods to estimate the stack distance of any memory access. For example, to estimate the stack distance of the second access to *A* in Figure 4, we sum the estimated likelihoods that the reuse distance of the memory accesses executed between the two accesses to *A* have reuse distances such that their corresponding arcs reach beyond "Out Boundary".

StatStack uses this approach to estimate the stack distances of all memory accesses in a reuse distance sample, effectively estimating a stack distance distribution. StatStack uses this distribution to estimate the

miss ratio for any given cache size, C, as the fraction of stack distances in the estimated stack distance distribution that are greater than C.

5 Identifying non-temporal accesses

Using the stack distance profile of an application we can determine which memory accesses do not benefit from caching. We will refer to memory accessing instructions whose data is never reused during its lifetime in the cache hierarchy as *non-temporal memory accesses*.

If these non-temporal accesses can be identified, the compiler, a post processing pass, or a dynamic instrumentation engine can alter the application to use non-temporal instructions in these locations without hurting performance.

The system we model implements a non-temporal hint that causes a cache line to be installed in the L1, but never in any of the higher cache levels. It turns out that modeling this system is fairly complicated, we will therefore describe our algorithm to find non-temporal accesses in three steps. Each step adds more detail to the model and brings it closer to the hardware. A fourth step is included to take effects from sampled stack distances into account.

A first simplified approach

By looking at the forward stack distances of an instruction we can easily determine if the next access to the data used by that instruction will be a cache miss, i.e. the instruction is non-temporal. An instruction has non-temporal behavior if all forward stack distances, i.e. the number of unique cache lines accessed between this instruction and the next access to the same cache line, are larger or equal to the size of the cache. In that case, we know that the next instruction to touch the same data is very likely to be a cache miss. Therefore, we can use a non-temporal instruction to bypass the entire cache hierarchy for such accesses.

This approach has a major drawback. Most applications, even purely streaming ones that do not reuse data, may still exhibit short temporal reuse, e.g. spatial locality where neighboring data items on the same cache line are accessed in close succession. Since cache management is done at a cache line granularity, this clearly restricts the number of possible instructions that can be treated as non-temporal.

Refining the simple approach

Most hardware implementations of cache management instructions allow the non-temporal data to live in parts of the cache hierarchy, such

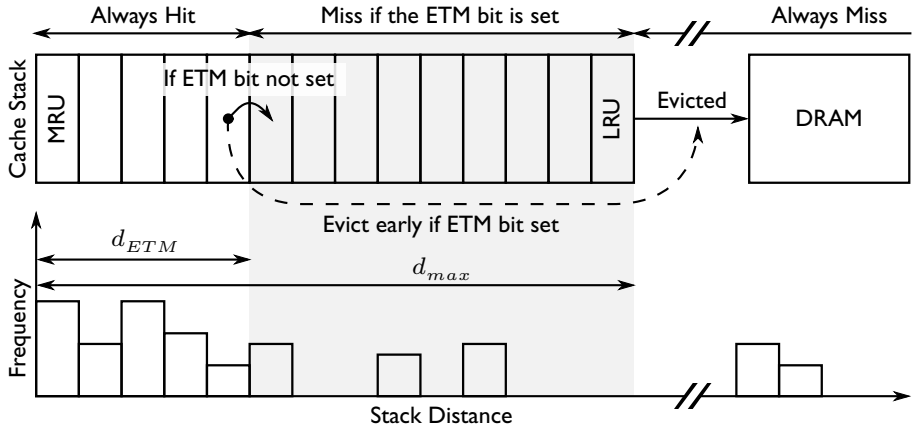


Figure 5: LRU stack (top) and the forward stack distance distribution of a memory accessing instruction (bottom). If the ETM bit is set the cache lines are evicted early to DRAM when they reach d_{ETM}. The bars within the shaded area of the forward stack distances distribution represent memory accesses that will result in cache misses if the ETM bit is set. An instruction is classified as non-temporal if there are less than t_m forward stack distances between d_{ETM} and d_{max} and at least one forward stack distance greater than d_{max}.

as the L1, before it is evicted to memory. We can exploit this to accommodate short temporal reuse of cache lines. We assume that whenever a non-temporal memory access touches a cache line, the cache line is installed in the MRU-position of the LRU stack, and a special bit on the cache line, the *evict to memory* (ETM) bit, is set. Whenever a normal memory access touches a cache line, the ETM bit is cleared. Cache lines with the ETM bit set are evicted earlier than other lines, see Figure 5. Instead of waiting for the line to reach the depth d_{max} it is evicted when it reaches a shallower depth, d_{ETM}. This allows us to model implementations that allow non-temporal data to live in parts of the memory hierarchy. For example, the memory controller in our AMD system evicts ETM tagged cache lines from the L1 to main memory, and would therefore be modeled with d_{ETM} being the size of the L1 and d_{max} the total combined cache size.

The model with the ETM bit allows us to consider memory accesses as non-temporal even if they have short reuses that hit in the small ETM area. Instead of requiring that all forward stack distances are larger than the cache size, we require that there is at least one such access and that the number of accesses that reuse data in the area of the LRU stack outside the ETM area, the gray area in Figure 5, is small, i.e. the number of misses introduced if the access is treated as non-temporal is small. We

thus require that one stack distance is greater or equal to d_{max}, and that the number of stack distances that are larger or equal to d_{ETM} but smaller than d_{max} is smaller than some threshold, t_m. In most implementations t_m will not be a single value for all accesses, but depend on factors such as how many additional cache hits can be created by disabling caching for a memory access.

The hardware we want to model does not, unfortunately, reset the ETM bit when a temporal access reuses ETM data. This new situation can be thought of as sticky ETM bits, as they are only reset on cache line eviction.

Handling sticky ETM bits

When the ETM bit is retained for the cache lines' entire lifetime in the cache, the conditions for a memory accessing instruction to be non-temporal developed in Section 5 are no longer sufficient. If instruction *X* sets the ETM bit on a cache line, then the ETM status applies to all subsequent reuses of the cache line as well. To correctly model this, we need to make sure that the non-temporal condition from Section 5 applies, not only to *X*, but also to all instructions that reuse the cache lines accessed by *X*.

The sticky ETM bit is only a problem for non-temporal accesses that have forward reuse distances less than d_{ETM}. For example, consider a memory accessing instruction, *Y*, that reuses the cache line previously accessed by a non-temporal access *X* (here *Y* is a cache hit). When *Y* accesses the cache line it is moved to the MRU position of the LRU stack, and the sticky ETM bit is retained. Now, since *Y* would have resulted in a cache hit no matter if *X* had set the sticky ETM bit or not, this is the same as if we would have set the sticky ETM bit for the cache line when it was accessed by *Y*.

Therefore, instead of applying the non-temporal condition to a single instruction, we have to apply it to all instructions reusing the cache line accessed by the first instruction.

In a machine, such as our AMD system, where d_{ETM} corresponds to the L1 cache, this new condition allows us to categorize a memory access as non-temporal if all the data it touches is reused through the L1 cache or misses in the entire cache hierarchy. Due to the stickiness of the non-temporal status, this condition must also hold for any memory access that reuses the same data through the L1 cache.

Handling sampled data

To avoid the overhead of measuring exact stack distances, we use StatStack to calculate stack distances from sampled reuse distances. Sampled

Level	Size (kB)	Associativity	Line Size (B)	Shared
1 (data)	64	2	64	No
2	512	16	64	No
3	6144	48	64	Yes

Table 1: Cache properties of the model system (AMD Phenom II X4 920)

stack distances can generally be used in place of a full stack distance trace with only a small decrease in *average* accuracy. However, there is always a risk of missing some critical behavior. This could potentially lead to flagging an access as non-temporal, even though the instruction in fact has some temporal behavior in some cases, and thereby introducing an unwanted cache miss.

In order to reduce the likelihood of introducing misses due to sampling, we need to make sure that flagging an instruction as non-temporal is always based on reliable data. We do this by introducing a sample threshold, t_s, which is the smallest number of samples originating from an instruction that can be considered to be non-temporal.

6 Evaluation methodology

Model system

To evaluate our model we used an x86 based system with an AMD Phenom II X4 920 processor with the AMD family 10h micro-architecture. The processor has 4-cores, each with a private L1 and L2 cache and a shared L3 cache. The processor enforces exclusion between L1 and L2, but not always between L3 and the lower levels if data is shared between cores.

According to the documentation of the `prefetchnta` instruction, data fetched using the non-temporal prefetch is not installed in the L2 unless it was fetched from the L2 in the first place. However, our experiments show that this is not the case. It turns out that data fetched from the L2 cache using the non-temporal prefetch instruction is never re-installed in the L2. The system therefore works like the system modeled in Section 5 where the ETM-bit is sticky.

We used the performance counters in the processor to measure the cycles and instruction counts using the perf framework provided by recent Linux kernels.

Benchmark preparation

The benchmarks were first compiled normally for initial reference runs and sampling. Sampling was done on each benchmark running with the reference input set. Due to the low overhead of the sampler, the benchmarks were run to completion with the sampler attached throughout the entire run. After the initial profile run, we analyzed the profile using the algorithm in the previous section and generated a list of non-temporal memory accesses. The benchmarks were then recompiled taking this profile into account.

The cache managed versions of the benchmarks were compiled using a compiler wrapper script that hooked into the compilation process before the assembler was called. The assembly output was then modified before it was passed to the assembler. Using the debug information from the binary we were able to find the memory accesses in the assembly output corresponding the instruction addresses in the non-temporal list. Before each non-temporal memory access the script inserted a `prefetchnta` instruction to the same memory location as the original access.

Algorithm parameters

We model the cache behavior of our benchmarks using StatStack and a reuse distance sample with 100 000 memory access pairs per benchmark. We use a minimum samples threshold, t_s, of 50 samples. The maximum number of introduced misses, t_m, is set to 0 samples; this may seem strict at first, but remember that we are sampling memory accesses and one sample corresponds to several hundred thousand memory accesses.

Cache exclusivity guarantees that there is at most one copy of a cache line in the caches where exclusivity is enforced. For example, an access to a cache line that resides in the L2 of our system will cause that cache line to be removed from the L2 and installed in the L1. A system where cache exclusivity is not enforced would not remove the copy in the L2. When the cache line is evicted from the L1 cache it is installed in the L2 cache, i.e. it is transferred from the LRU position of the L1 to the MRU position of the L2. This behavior lets us merge the two caches and treat them as one larger LRU stack where each cache level corresponds to a contiguous section of the stack. In the model system, the first 1k lines correspond to the L1 cache, the next 8 kB lines correspond to the L2 and the last 96 kB lines correspond to the L3. This global stack has 105 k lines in total, i.e. the total cache size in lines. We let d_{max} be the depth of this global stack.

Since we are using StatStack we have made the implicit assumption that caches can be modeled to be fully associative, i.e. conflict misses

are insignificant. In most cases this is a valid assumption, especially for large caches with a high degree of associativity. A notable case where this assumption may break is for the L1 cache, which has a low degree of associativity. We therefore have to be more conservative when evaluating stack distances within this range. We use different, conservative, values of d_{ETM}, when calculating the number of introduced misses and handling the stickiness of the ETM bits. We use a d_{ETM} value of twice the L1 size, i.e. 2048 lines, when handling stickiness and half the L1 size, i.e. 512 lines, when calculating the number of misses introduced.

Benchmarks

Using the software classification introduced in Section 2 we selected two benchmarks representing each category for analysis. The number of non-temporal memory accesses and the effect on other applications in the system will depend on a benchmark's position in the classification map. Applications on the left-hand side of the map, *Don't Care* and *Victims*, do not install a significant amount of data in the shared cache and do not disturb other applications running on the system. As expected, our algorithm does not find any non-temporal memory accesses in such applications. Applications on the right-hand side of the map, *Gobblers & Victims* and *Cache Gobblers*, have a high base miss ratio and store a large amount of non-temporal data in the shared cache. We expect such applications to be good candidates for cache management.

There is normally no need to differentiate between cache misses and replacements. Whenever there is a cache miss, a new cache line is installed and another one is replaced. When we start to software manage the cache, we disable caching for certain instructions. This causes misses to occur, but, since the data is never cached, no replacements take place. When we classify managed applications, it therefore makes more sense to use the replacement ratio rather than the miss ratio to better capture the effect on other application.

We extended the StatStack algorithm to handle non-temporal memory accesses to calculate new replacement and miss ratios for managed applications. Looking at Figure 6(a) we see that libquantum's replacement ratio is reduced from approximately 20% to 0% in the shared cache, while the miss ratio stays at 20%. This can be explained by the fact that the instructions installing non-temporal data into the SLLC now bypass the cache. The fact that they bypass the cache leads to a decreased replacement ratio, i.e. fewer cache lines installed in the SLLC. We would normally expect the miss ratio to be decreased due to a reduction of non-temporal data in the SLLC, which would allow more temporal data to be

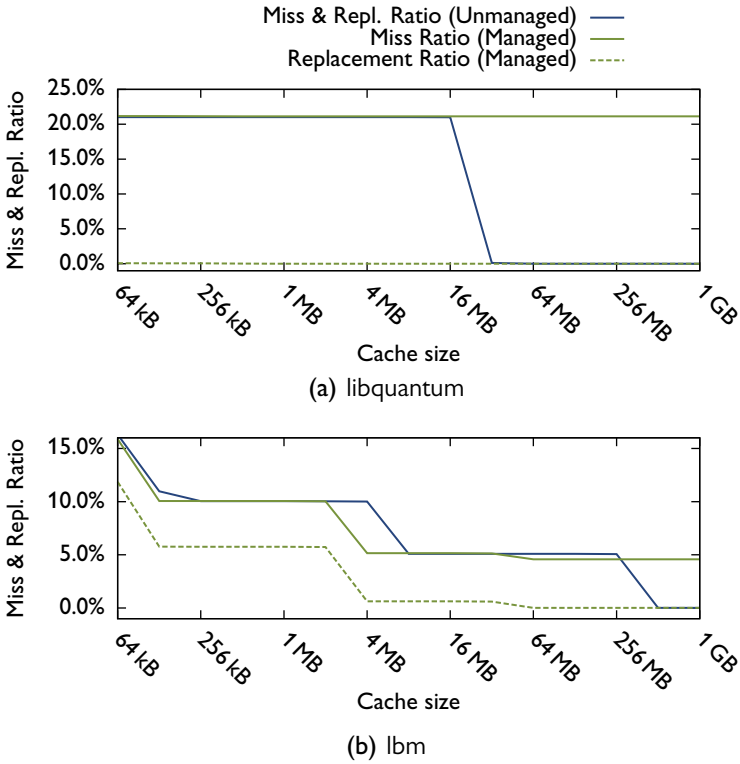


Figure 6: Miss and replacement ratio before and after cache managing the benchmarks to avoid caching of non-temporal data. The number of replacements can be reduced by cache management in both applications. The number of misses in (b) can be reduced, particularly around the target cache size, because a reduction in the number of replacements will allow more temporal data to fit in the cache. The miss ratio normally drops to 0% once the entire data set fits in the cache, this is no longer the case for managed applications, since the non-temporal memory accesses always cause a miss.

stored instead. In the case of libquantum, there is no additional temporal data that can be squeezed into the cache.

When looking at cache sizes larger than d_{max}, another effect of software cache management is seen. As the data set starts to fit in the cache, the miss ratio is normally reduced. This may no longer be the case when applying cache management. When we manage the cache, we force certain accesses to bypass the cache and the reuse to become a miss, independent of cache size.

Lbm has slightly more interesting access patterns than libquantum, which includes a decrease in miss ratio around the target cache size. Looking at Figure 6(b), we notice that, in addition to the features displayed by libquantum, parts of the miss ratio curve have been shifted to

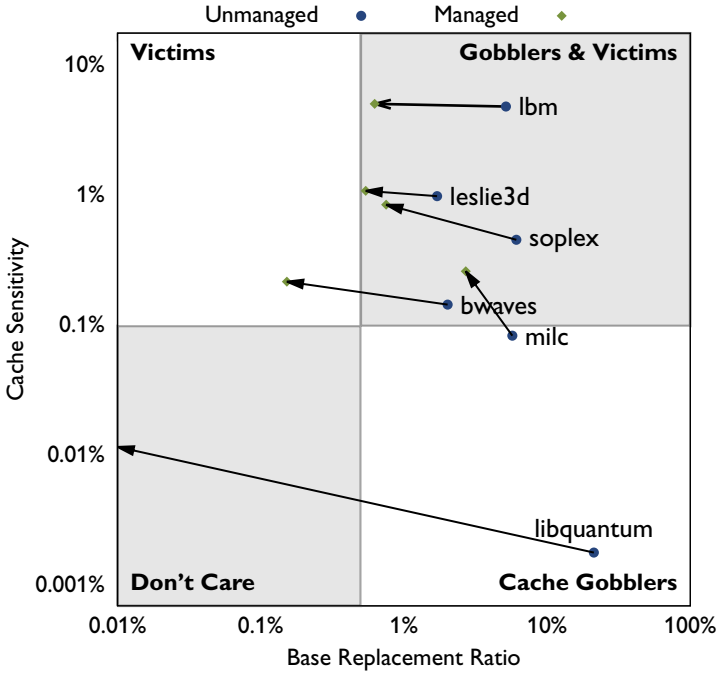


Figure 7: Changes in classification after disabling caching of non-temporal memory accesses. Note that this classification is based on the replacement ratio rather than the miss ratio.

the left. This can be explained by the fact that removing non-temporal data from the cache allows more of the temporal data to fit in the cache.

We reclassify our benchmarks based on their new replacement ratio curves, the new classification allows us to predict how applications affect each other after we introduce the non-temporal memory accesses. The change in classification for the managed benchmarks is shown in Figure 7.

7 Results and analysis

The results for runs of six different mixes of four SPEC2006 benchmarks running with the reference input set, with and without software cache management are shown in Figure 8 and Figure 9. Figure 8 shows a mix of four applications from different categories. Figure 9 shows five different mixes consisting of two pairs of benchmarks from different categories. The instructions per cycle, IPC, is shown for each of the benchmarks, both when running in isolation and when running in the mix and with and without cache management. The cache management instructions

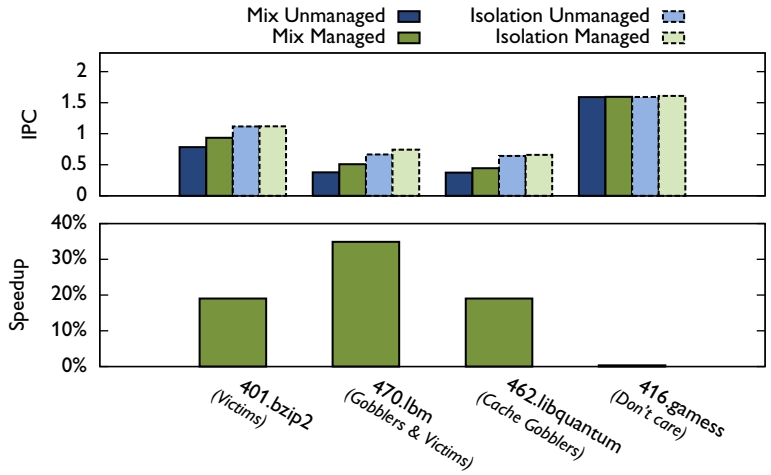


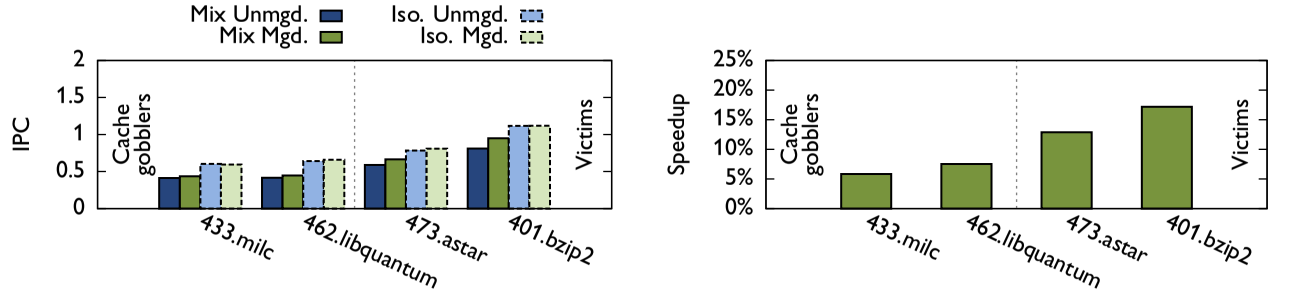
Figure 8: Performance for a mix of four applications, each from a different category. The IPC plot compares the IPC for managed and unmanaged benchmarks, both in a mix and in isolation. The speedup is relative to the unmanaged mix.

are not included in the instruction counts when calculating IPC for managed applications, including them would give an unfair advantage to the managed applications. The speedup is the improvement in IPC over the unmanaged version when running in a mix.

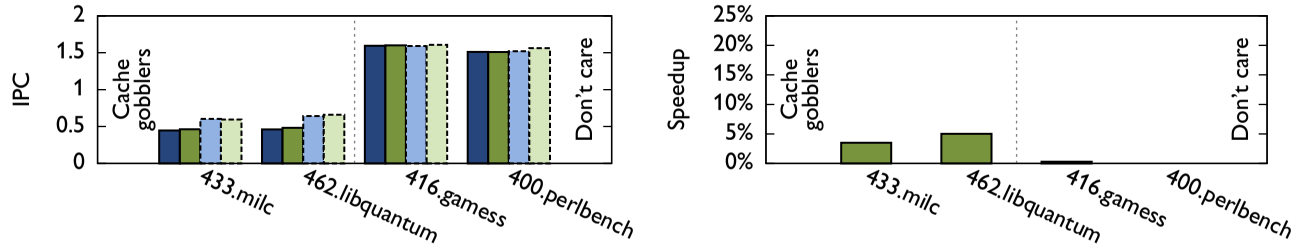
As seen by comparing the IPC for managed and unmanaged applications in isolation, Figure 8 and Figure 9, inserting additional `prefetchnta` instructions does not negatively impact performance in isolation. In fact, the IPC of LBM is increased by approximately 12%. This effect can be explained by Figure 6(b), where a miss ratio knee is clearly visible between 4 MB and 8 MB. Applying software cache management pushes the knee to the left, i.e. towards smaller cache sizes, and decreases the miss ratio for systems with between 4 MB and 8 MB of cache.

Looking at Figure 8 and Figure 9 we see that all applications, except for the applications in the *Don't Care* category, have a lower IPC when running in a mix than running in isolation. This is to be expected since running in a mix means that all the applications compete for a shared cache and shared bandwidth. Applications in the *Don't Care* category fit most of their data in the private cache, which makes their bandwidth and SLLC requirements extremely small. This explains why this category does not benefit from cache management.

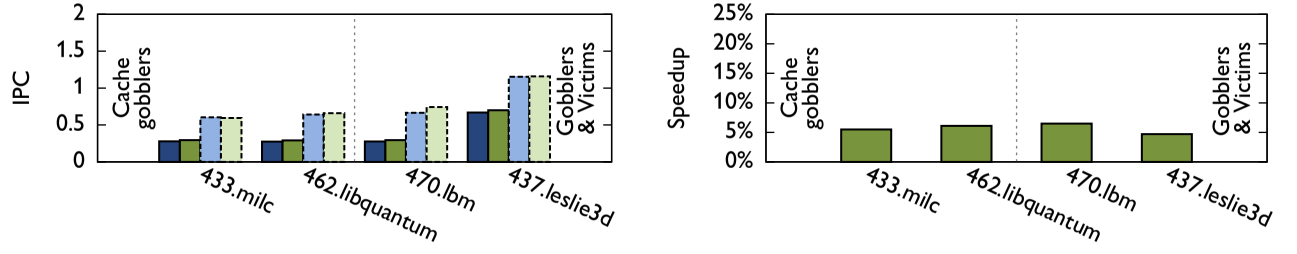
The difference between the *Victims* and the *Don't Care* categories is that the former uses some amounts of L3 cache, while the latter does not. This would suggest that interference between these two categories should be small when running together. This is supported by Figure 8(d), where all applications in this mix run at the same speed as in isolation.



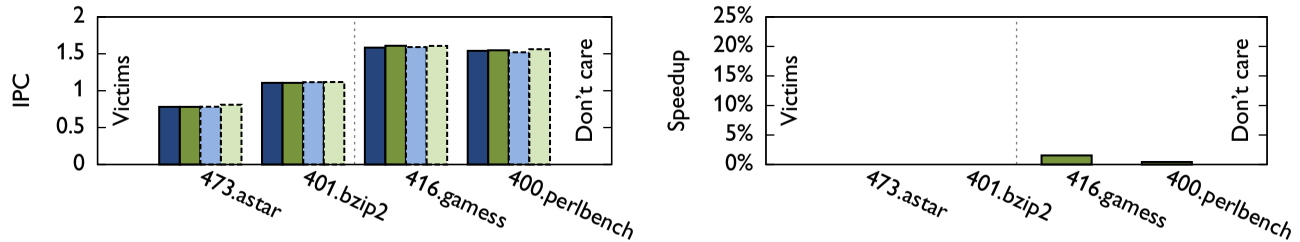
(a) *Cache Gobblers ↔ Victims*



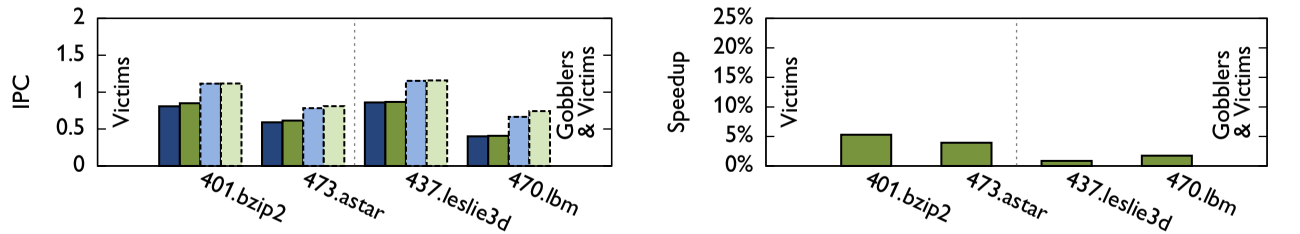
(b) *Cache Gobblers ↔ Don't Care*



(c) *Cache Gobblers ↔ Gobblers & Victims*



(d) *Victims ↔ Don't Care*



(e) *Victims ↔ Gobblers & Victims*

Figure 9: Performance of mixes with benchmarks from two different categories. Benchmarks from different categories are separated by a dotted line. All of the benchmarks, except for the *Don't Care* category, generally run slower in mixes than in isolation. Disabling caching for non-temporal memory accesses regains some of the IPC lost to cache and bandwidth contention without any negative impact on application performance in isolation.

There could still be some interference between applications within the *Victims* category, but this is likely to be very small since these applications have a small bandwidth and cache footprint.

Because applications in the *Cache Gobblers* and *Gobblers & Victims* categories have similar cache and bandwidth pressure they affect other applications in the same way. Looking at Figure 9(a), Figure 9(c) and Figure 8(e) we see that running together with applications from these categories causes a significant decrease in IPC compared to when running in isolation. We expect that managing these categories reduces their cache footprint, and as a consequence reduce their impact on IPC. Our results indicate that some of the performance lost to contention for the shared resources can be regained using software cache management.

A somewhat surprising result might be that applications from the *Cache Gobblers* category benefit from cache management themselves. This is the case for all of the mixes, but is particularly visible in Figure 9(b). The speedup when running with applications from the two victim categories can largely be attributed to a reduction in the total bandwidth requirement of the mix. The speedup when running together with the *Don't Care*, Figure 9(b), is harder to explain, but is likely due to small reductions in the miss ratio in the *Cache Gobblers*.

8 Related work

There has been plenty of research focused on improving cache efficiency. Most of this work has been targeting the miss ratio of private caches [11–13]. Focus has recently started to shift towards shared caches [3, 15]. These methods are either software driven or hardware driven. They all have one thing in common, the need to identify non-temporal data or, as in our case, memory accesses referencing non-temporal data. Once the non-temporal data is identified this information is propagated to the cache, typically by setting a non-temporal bit in the cache tags. This bit is then used by the cache replacement policy to explicitly handle non-temporal data.

Tyson et al. [11] propose a simulation based method to identify non-temporal memory accesses. It can be described in two steps: First, they simulate the cache and identify the instructions with miss ratios above a threshold (25%). Then, for each dynamic execution of these instructions, they keep track of how often the fetched cache line is accessed before being evicted. If this occurs less than 25% of the time the instruction is identified as being non-temporal. Our approach differs on the following key points: 1) It does not require expensive simulations; 2) It considers all instructions as potentially non-temporal, not only the ones with a

miss ratio above a threshold. This increases the potential to reduce the caching of non-temporal data; 3) Our method is tailored to use existing hardware on the x86 architecture.

Wong et al. [13] propose a method to identify non-temporal memory accesses based on Mattson's optimal replacement algorithm (OPT) [5]. Their method is similar to ours in that it uses the forward stack distance distribution of a memory accessing instruction to determine if it is a temporal instruction. However, instead of using LRU stack distances, they use OPT stack distances, which requires expensive simulation.

Several hardware methods have been proposed [3, 11, 13, 15], that dynamically identify non-temporal data. Xie et al. [15] propose a replacement policy, PIPP, to effectively way-partition a shared cache, that explicitly handles non-temporal (streaming) data. To detect non-temporal data, they introduce a set of shadow tags [8] used to count the number of hits to a cache line that would have occurred if the thread was allocated all ways in the cache set. Similarly to Tyson's method [11], they identify a cache line as non-temporal if there are no accesses to it prior to its eviction. This approach is rather course grained, during a time period when the majority of the data accessed by a thread is non-temporal it assumes that all data accessed by the thread is non-temporal.

Qureshi et al. [7] propose an insertion policy (DIP) where on a cache miss to non-temporal data it is installed in the LRU position, instead of the MRU position of the LRU stack. To detect non-temporal data they use two sets of training cache sets. In the first training set, data is installed in the LRU position, and in the other data is installed in the MRU position. The rest of the cache sets use the insertion policy of the training set that currently has the highest hit ratio. This method has been extended [3] to be thread aware (TADIP), by using separate training sets and insertion policies (insertion in LRU or MRU) for the different threads. Both DIP and TADIP exhibit the same course grained time varying behavior as PIPP. A recent extension [4] introduces an additional policy that installs cache lines in the $\mathrm{MRU} - 1$ position.

Petoumenos et al. [6] propose an instruction based reuse distance predictor and a replacement algorithm based on the predicted reuse distances. Their algorithm approximates the optimal algorithm by replacing the cache line that is predicted to be reused furthest into the future.

The time varying behavior of PIPP, DIP and TADIP can be effective to handle the time varying behavior of applications (program phases). However, for applications whose instruction working set is different between program phases, the static classification of memory instructions, used in this and other papers, allows for a more fine grained control, while at the same time following the time varying behavior of the applications.

9 Summary and future work

We describe an application classification framework that allows us to predict how applications affect each other when running on a multicore and a method for finding non-temporal memory accesses. Using a single low-overhead profile run of an application, we can acquire enough information to both classify the application and find non-temporal memory accesses for any combination of shared and private cache sizes. Our method can be used together with contemporary hardware to provide a speedup for existing applications. We show that this is the case for a selection of the SPEC2006 benchmarks. Using a modified StatStack implementation we can reclassify applications based on their replacement ratios after applying cache management, this allows us to reason about how cache management impacts performance.

Future work will explore other hardware mechanism for handling non-temporal data hints from software and possible applications in scheduling.

Acknowledgments

The authors would like to thank Kelly Shaw and David Black-Schaffer for valuable comments and insights that has helped to improve this paper. This work was financially supported by the CoDeR-MP and UPMARC projects.

References

[1] Erik Berg and Erik Hagersten. "Fast Data-Locality Profiling of Native Execution". In: *ACM SIGMETRICS Performance Evaluation Review* 33.1 (2005), pp. 169–180. DOI: 10.1145/1071690.1064232.

[2] David Eklov and Erik Hagersten. "StatStack: Efficient Modeling of LRU Caches". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Mar. 2010, pp. 55–65. DOI: 10.1109/ISPASS.2010.5452069.

[3] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Jr Simon Steely, and Joel Emer. "Adaptive Insertion Policies for Managing Shared Caches". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 208–219. DOI: 10.1145/1454115.1454145.

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)". In: *Proc. International Symposium on Computer Architecture (ISCA)*. 2010, pp. 60–71. DOI: 10.1145/1815961.1815971.

[5] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. "Evaluation techniques in storage hierarchies". In: *IBM Journal of Research and Development* 9.2 (1970), pp. 78–117.

[6] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. "Instruction-based Reuse-Distance Prediction for Effective Cache Management". In: *Proc. Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. 2009, pp. 49–58. DOI: 10.1109/ICSAMOS.2009.5289241.

[7] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. "Adaptive Insertion Policies for High Performance Caching". In: *Proc. International Symposium on Computer Architecture (ISCA)*. San Diego, California, USA: ACM, 2007, pp. 381–391. DOI: 10.1145/1250662.1250709.

[8] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2006, pp. 423–432. DOI: 10.1109/MICRO.2006.49.

[9] Timothy Sherwood, Brad Calder, and Joel Emer. "Reducing Cache Misses Using Hardware and Software Page Placement". In: *Proc. International Conference on Supercomputing (ICS)*. Rhodes, Greece, 1999, pp. 155–164. DOI: 10.1145/305138.305189.

[10] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. "Managing Shared L2 Caches on Multicore Systems in Software". In: *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. 2007.

[11] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. "A Modified Approach to Data Cache Management". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 1995, pp. 93–103. DOI: 10.1109/MICRO.1995.476816.

[12] Zhenlin Wang, McKinley Kathryn S., Arnold L. Rosenberg, and Weems Charles C. "Using the Compiler to Improve Cache Replacement Decisions". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2002, pp. 199–208. DOI: 10.1109/PACT.2002.1106018.

[13] Wayne A. Wong and Jaen-Loup Baer. "Modified LRU Policies for Improving Second-Level Cache Behavior". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2000, pp. 49–60. DOI: 10.1109/HPCA.2000.824338.

[14] Yuejian Xie and Gabriel H Loh. "Dynamic Classification of Program Memory Behaviors in CMPs". In: *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*. 2008.

[15] Yuejian Xie and Gabriel H. Loh. "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches". In: *ACM SIGARCH Computer Architecture News* 37.3 (June 2009), pp. 174–183. DOI: 10.1145/1555815.1555778.

PAPER II

Efficient Techniques for Predicting Cache Sharing and Throughput

Andreas Sandberg
David Black-Schaffer
Erik Hagersten

©2012 ACM, Inc. Reprinted, with permission, from *PACT'12*, September 19–23, 2012.
DOI: 10.1145/2370816.2370861

Abstract — This work addresses the modeling of shared cache contention in multicore systems and its impact on throughput and bandwidth. We develop two simple and fast cache sharing models for accurately predicting shared cache allocations for random and LRU caches.

To accomplish this we use low-overhead input data that captures the behavior of applications running on real hardware as a function of their shared cache allocation. This data enables us to determine how much and how aggressively data is reused by an application depending on how much shared cache it receives. From this we can model how applications compete for cache space, their aggregate performance (throughput), and bandwidth.

We evaluate our models for two- and four-application workloads in simulation and on modern hardware. On a four-core machine, we demonstrate an average relative fetch ratio error of 6.7% for groups of four applications. We are able to predict workload bandwidth with an average relative error of less than 5.2% and throughput with an average error of less than 1.8%. The model can predict cache size with an average error of 1.3% compared to simulation.

1 Introduction

The shared cache in contemporary multicore processors has been repeatedly shown to be a critical resource for application performance [8, 13–15, 18]. This has motivated a significant amount of research into modeling the impact of cache sharing with the goal of understanding applications' interactions through the shared cache and for providing insight to schedulers and runtime systems [10, 11, 15, 20].

This work presents two models for predicting cache allocations, bandwidth requirements, and performance of application mixes in the presence of a shared last-level cache. The models are developed for random replacement and LRU caches, but are shown to be accurate for the pseudo-LRU caches of modern Intel processors.

These models take into account the complexities of modern hardware (such as out-of-order execution and hardware prefetchers) by leveraging input data that incorporates the applications' behavior on real hardware. This input data consist of the applications' fetch and hit rates, IPCs, and hit ratios as a function of their cache allocation, and can be acquired with low overhead on modern multicore machines [6]. This low-overhead data is in contrast to many existing methods for modeling cache sharing which rely on expensive data such as stack distance traces [2–4, 17].

To model cache sharing we use an application's fetch and hit rates as a function of cache size to determine how much of its data is reused for a given cache allocation, and how often that data is reused. With this information we can model how multiple applications compete for shared cache space. The model then uses a numerical solver to find a stable solution that determines the final cache allocations. Once we know the cache allocations we can use our input data to predict performance (throughput) and bandwidth requirements for mixes of co-scheduled applications.

This ability to model cache sharing and predict its impact on performance and bandwidth is important for scheduling and performance analysis on complex systems. Such modeling forms the basis for resource-aware placement in modern memory hierarchies, scheduling on heterogeneous architectures, and for making runtime decisions on future chips in the presence of power constraints (e.g., dark silicon). By combining low-overhead data that reflects the complexities of the real hardware and a simple sharing model we are able to quickly and accurately predict sharing and performance, which is essential for such goals.

This paper makes the following contributions:

- We present a statistical cache-sharing model for random caches that uses high-level, low-overhead input data.
- We extend the model to LRU caches by deriving aggregate data reuse information from the input data, and using this to model competition for cache space based on the data reuse frequencies of each application.
- We demonstrate the accuracy of the model for predicting cache sharing and fetch ratios for mixes of two and four co-scheduled applications through simulation and on real hardware.
- We demonstrate the ability to accurately predict performance (IPC) and bandwidth requirement of application mixes on real hardware.

2 Modeling Cache Sharing

Consider two applications sharing a cache. Their behavior with respect to the shared cache can intuitively be thought of as two flows of liquid filling an overflowing glass. The two in-flows correspond to fetches into the cache and the liquid pouring out of the glass corresponds to the replacement stream from the cache. If the in-flows are constant, the system will

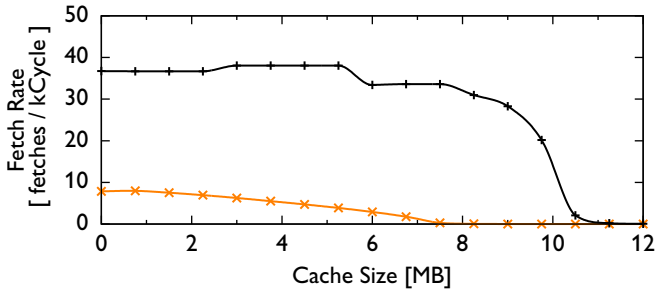


Figure 1: Example Cache Pirate data showing fetch rate as a function of cache size for two applications.

eventually reach a steady state. At steady state, the concentrations of the liquids in the glass are constant and proportional to their relative inflow rates. Furthermore, the outflow rates of the liquids are proportional to their concentrations. This very simple analogy describes the behavior of random caches.

Describing LRU caches requires data reuse to be considered since data reused *frequently enough* will stick in the cache and avoid replacement. In the liquid analogy above, data reused frequently enough can be thought of as ice cubes that never leave the glass. The threshold for how frequently data needs to be reused to exhibit "sticky" behavior varies between sharing situations.

Low-Overhead Input Data

The goal of our cache sharing models is to find the amount of cache allocated to each application in a mix of co-scheduled applications at steady state. This requires per-application information about fetch[1] rate and data reuse characteristics for all applications as well as how that information is affected by cache contention. In order for this information to accurately describe the target system, it needs to take into consideration effects from complex dynamic hardware, such as super-scalar out-of-order execution and hardware prefetching.

Cache Pirating [6] is a method to capture our required input data on the target hardware. In Cache Pirating, the studied application is co-scheduled to share a cache with a small cache-intensive micro benchmark, the Pirate. The Pirate is designed to steal only cache, leaving other shared resources untouched. In a single run, the amount of cache the

[1]The term *fetch* is used extensively in this paper to describe a movement of data from memory to cache caused by either a cache miss or prefetching activity.

Pirate steals is varied, while the effects on the studied application are measured using hardware performance counters. The application's miss rate, fetch rate, hit rate, miss ratio, fetch ratio, hit ratio, memory bandwidth, CPI, etc., *as a function of cache size*, can be measured with an average overhead of 5.5%.

Cache Pirating typically measures sensitivity by stealing a whole way at a time, while our models assume continuous data. We therefore interpolate the measured data using monotone cubic splines [7]. We chose this interpolation method over linear interpolation because the resulting function estimates the behavior in applications with sharp steps in their fetch rate curves (e.g., applications with a small fixed data set) more accurately by making the edges sharper. Figure 1 shows an example of data produced using Cache Pirating: Fetch rate as a function of cache size for two applications, measured in 16 discreet steps and interpolated with monotone cubic splines.

The following section describes how the fetch rate and hit rate information measured using Cache Pirating is used to determine the amount of cache allocated to each application in a sharing situation. Knowing the amount of cache allocated to each application allows us to predict additional performance metrics. In Section 4 we show how throughput and bandwidth demand can be predicted using knowledge about cache allocations and the data from Cache Pirating.

Modeling Random Caches

In random caches, sharing only depends on two events: fetches into the cache and replacements. At steady state, the amount of data an application installs into the cache is equal to the amount of data evicted from that application's cache allocation; that is, the application's fetch rate (fetches per cycle) must equal its replacement rate (replacements per cycle). An application's replacement rate is proportional to the total fetch rate, F, into the cache and its replacement probability. Since replacements are random, the replacement probability is proportional to the amount of cache allocated to the application. For a shared cache of size C and an application, n; the application's fetch rate, f_n, and cache allocation, c_n, are related according to:

$$\begin{cases} \mathrm{f}_n = \mathrm{F}\,\dfrac{\mathrm{c}_n}{\mathrm{C}} \\ \mathrm{C} = \displaystyle\sum_i \mathrm{c}_i \end{cases} \tag{1}$$

We can solve the equation system above, given that we have each application's fetch rate as a function of cache size, using readily available equation system solvers.

Modeling LRU Caches

LRU caches, unlike random caches, use access history to replace the item that has been unused for the longest time. We refer to the duration of time a cache line has been unused as its *age*. Whenever there is a replacement decision, the oldest cache line is replaced.

In practice, we can not determine the age of individual cache lines based on our input data. Instead, we look at *groups* of cache lines with the same maximum *age*[2] and let the groups from different applications compete for cache space.

Cache lines that are fetched into the cache but are evicted before they are reused are put in a separate group and handled differently. After the initial fetch into the cache, the age of these cache lines increases whenever *any* application fetches new data into the cache. This allows us to treat them as one group common to all applications, with one common age. We will refer to these cache lines as *volatile* since they are evicted from the cache before they are reused.

Cache lines which are reused before they are evicted from the cache are referred to as *sticky* cache lines since their reuse makes them resilient to eviction. Normally an application reuses more data in the cache when the amount of cache the application has access to grows. This means that some data is *potentially* sticky and only becomes sticky when the application has access to enough cache. The amount of sticky and volatile data in the cache therefore depends on how much cache an application has access to, which is a function of what other applications are co-executing.

To explain the LRU model, we will first describe how to model sharing within the volatile group and how the age of the group is determined. We will then describe how sticky groups are modeled, and finally how our solver uses this information to determine cache sharing.

Modeling Volatile Data

When applications do not reuse their data before it is evicted from the cache, LRU caches degenerate into FIFO queues with data moving from the MRU position to the LRU position before being evicted. Similar to random replacement, the amount of cache allocated to an application will be proportional to its fetch rate. This observation allows us to use the method devised for random replacement to model cache sharing for volatile data. Assuming that we know the amount of cache available to volatile data[3], C^v, we can solve Equation 1 for the volatile part of the

[2] At any given moment, the cache lines in a group will have different ages. It therefore only makes sense to talk about the maximum age of a group.

[3] We can estimate the amount of sticky data each application has, and therefore whatever is left of the cache is used for volatile data.

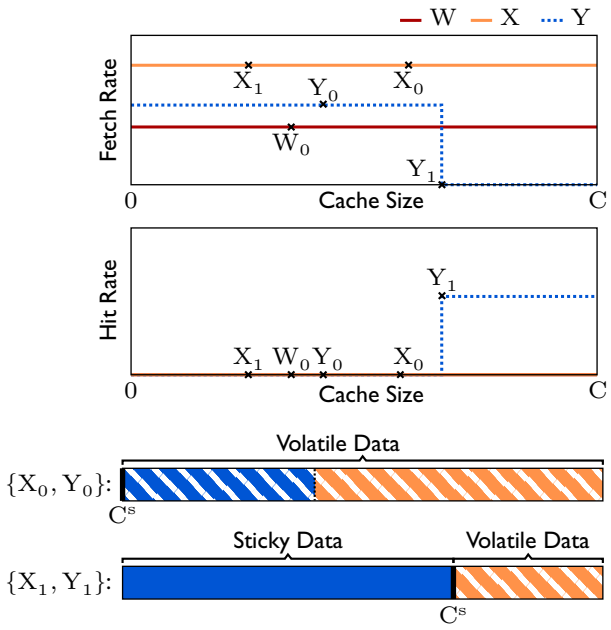


Figure 2: Fetch and hit rate curves for three sample applications. Application W and X always miss in the shared cache, while Y misses only when it has less than $c_n(Y_1)$ space in the cache. The cache allocations for the two stable cache sharing configurations of X and Y are shown below.

cache. This allows us to estimate the amount of volatile data, c^v, each application has.

Since sticky data and volatile data from different applications compete for cache space, we need to be able to compare their maximum age. Because the cache degenerates into a FIFO queue for volatile data, the maximum age of volatile elements can be determined using Little's law [9].[4] Assuming that we know the total size of the volatile part of the cache, C^v, and the *total* fetch rate into the cache, the maximum age of all volatile cache lines, A^v, is:

$$A^v = \frac{C^v}{F} \tag{2}$$

Example 1: Consider application X and application W in Figure 2. Both of the applications have fetch rates that are independent of cache size and X has twice the fetch rate of W. Since the hit rate is zero for both

[4]Little's law sets up a relationship between the number of elements in a queue (size), the time spent in the queue (maximum age) and the arrival rate (fetch rate). The total arrival rate into the queue is the sum of all fetch rates, F, in the system.

of them, neither reuses any data in the cache (i.e., all data is volatile). Using the random replacement model for the volatile data, we conclude that X gets twice the cache allocation of W (i.e., X uses two thirds of the cache) causing the applications to stabilize at the solution $\{X_0, W_0\}$. Since the entire cache is filled with volatile data, the maximum age of volatile elements is described by:

$$A^v = \frac{C^v}{F} = \frac{C}{f(X_0) + f(W_0)}$$

□

Modeling Sticky Data

In most cases, there is both sticky and volatile data in the cache at the same time. Unlike volatile data, sticky data stays in the cache because it is reused while it is in the cache. When sticky and volatile cache lines compete for cache space, the decision to let a sticky cache line remain sticky depends on its age. A sticky cache line becomes volatile if it is older than the oldest volatile cache line. In our model, we make this decision for entire groups of cache lines with the same maximum age. A group of sticky cache lines with the same maximum age, a^s, is allowed to stay in the cache as sticky cache lines if it is younger than the oldest volatile cache line:

$$a^s < A^v \tag{3}$$

Similar to volatile data, we can estimate the maximum age for a group of sticky data using Little's law if we know the size of the group and its reuse rate. This can best be illustrated with an example:

Example 2: Application Y in Figure 2 does not reuse any data (its hit rate is zero) when it has access to less cache than $c(Y_1)$. However, it reuses all its data when it has access to more cache (its fetch rate is zero). This means that it has one group of potentially sticky data. The size of the group is $c(Y_1)$ and the aggregate reuse rate of all elements in the group is its hit rate, $h(Y_1)$.

When Y starts it will bring its entire data set into the cache and start reusing it, causing the data to become sticky. If X is then started, it will first install data into the unused part of the cache. The size of Y's sticky data set will at this point be $c(Y_1)$ and the rest of the cache will be filled with data belonging to X. Since X does not reuse its data, all its data will be volatile. We obtain the ages of sticky, a^s, and volatile, A^v, elements

when they start to compete for cache as follows:

$$\begin{aligned} a^s &= \frac{c(Y_1)}{h(Y_1)} \\ A^v &= \frac{C^v}{F} = \frac{C - c(Y_1)}{f(X_1)} \end{aligned}$$

$\{X_1, Y_1\}$ is a stable cache sharing configuration if Y's sticky cache lines are younger than X's volatile cache lines. This means that Y is reusing its data set frequently enough to prevent X from pushing it out of the cache.

An interesting feature of this benchmark combination is that it can have two stable cache sharing configurations. If X starts first and is allowed to fill the cache with its volatile data, when Y starts, it will have to compete with X to bring its data into the cache. At this point, the entire cache consists of volatile data since Y has not installed enough of its data to be able to reuse it before it is evicted from the cache. Since X has a higher fetch rate than Y, it fetches data faster and will therefore get more cache than Y. In this case, Y will never fit its entire group of potentially sticky data, and its data it will instead remain volatile. Both $\{X_0, Y_0\}$ and $\{X_1, Y_1\}$ are therefore valid sharing configurations, depending on the starting order. □

In the examples so far, both of the benchmarks in a pair have either had sticky or volatile data, but not both. Real applications typically have both sticky and volatile data in the cache at the same time:

Example 3: Application Z in Figure 3 has two drops in its fetch rate curve, which means that it has two groups of potentially sticky data. This can happen in applications reusing two arrays of different size. For a cache size of $c(Z_0)$, the application is able to fit its first group of data in the cache (the fetch rate drops just before $c(Z_0)$) and that group becomes sticky. If the application has access to more cache than $c(Z_1)$, its fetch rate drops to zero and all of its data becomes sticky. In order to calculate the age, a^s, of a group of sticky data, we need to know how much that group contributes to the total hit rate and how big the group is. Assuming that we know the amount of sticky data in an application, c^s, as a function of cache size (we will show how to estimate this in Section 2), we can calculate $a^s(Z_0)$ as:

$$a^s(Z_0) = \frac{c^s(Z_0) - c^s(Z_{0^-})}{h(Z_0) - h(Z_{0^-})}$$

□

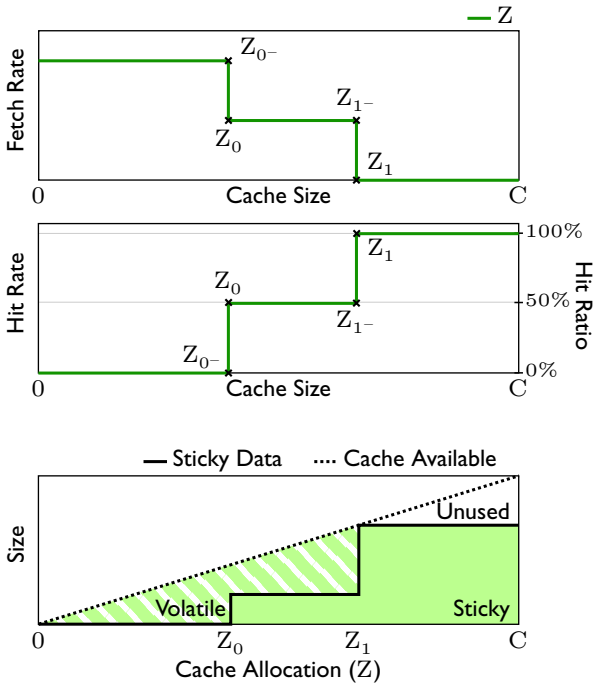


Figure 3: Z has two groups of sticky data of different sizes. When a group of sticky data starts to fit in the cache, the fetch rate starts to drop and the hit ratio increases. Whenever the hit ratio increases, the amount of volatile data at that point decreases (it becomes sticky) by the same relative amount.

The age derived in the example above is simply a finite difference approximation of a differential equation. In general, the access rate for sticky elements is defined as:

$$\mathrm{a}^{\mathrm{s}}(c) = \frac{\mathrm{dc}^{\mathrm{s}}}{\mathrm{dh}} \tag{4}$$

Equation 4 is actually a simplification that assumes that an application's execution rate does not change with cache size. However, execution rate generally increases as an application gets access to more cache. This variation is accurately captured in our Cache Pirate input data. Not compensating for the change in execution rate leads an erroneous estimate of a block's contribution to the total hit rate. We address this by using the difference in *hit ratio* (hits per memory access) which is execution rate independent. We then scale the difference in *hit ratio* with the application's *accesses rate* (which is execution rate dependent) to get the block's contribution to the total hit rate.

Estimating Sticky Group Sizes

The amount of sticky data can be estimated from how an application's hit ratio changes with its cache allocation. The relative change in hit ratio is proportional to the relative change in the sticky data.

Example 4: As seen in Figure 3, Z's hit ratio increases in two steps. This means that it has two groups of potentially sticky data. When it has access to less cache than $\mathrm{c}(\mathrm{Z}_0)$, the hit ratio is zero and it has no sticky data. The amount of sticky data can be broken down into three different cases based on the cache allocation c:

$0 \leq c < \mathrm{c}(\mathrm{Z}_0)$
: Since the hit ratio is zero, there is no sticky data.

$\mathrm{c}(\mathrm{Z}_0) \leq c < \mathrm{c}(\mathrm{Z}_1)$
: When the amount of cache is increased to $\mathrm{c}(\mathrm{Z}_0)$, the hit ratio increases from 0% to 50% (i.e., 50% of the fetches become hits). We would therefore expect 50% of the volatile data to become sticky. Since the amount of volatile data just before Z_0 is $\mathrm{c}(\mathrm{Z}_0)$, the amount of sticky data in this range is $\frac{1}{2}\mathrm{c}(\mathrm{Z}_0)$.

$\mathrm{c}(\mathrm{Z}_1) \leq c$
: At $\mathrm{c}(\mathrm{Z}_1)$, all of the fetches become hits. The sticky data set size is therefore $\mathrm{c}(\mathrm{Z}_1)$.

Using Z's hit ratio function, $\hat{\mathrm{h}}$, the reasoning above can be generalized into:

$$\frac{\mathrm{c^s}(\mathrm{Z}_x) - \mathrm{c^s}(\mathrm{Z}_{x^-})}{\mathrm{c}(\mathrm{Z}_x) - \mathrm{c^s}(\mathrm{Z}_{x^-})} = \frac{\hat{\mathrm{h}}(\mathrm{Z}_x) - \hat{\mathrm{h}}(\mathrm{Z}_{x^-})}{1 - \hat{\mathrm{h}}(\mathrm{Z}_{x^-})}$$

□

The difference approximation above can be generalized into the following differential equation:

$$\frac{\mathrm{dc^s}}{\mathrm{dc}} \frac{1}{\mathrm{c} - \mathrm{c^s}} = \frac{\mathrm{d}\hat{\mathrm{h}}}{\mathrm{dc}} \frac{1}{1 - \hat{\mathrm{h}}} \tag{5}$$

Putting It All Together

We have described how an LRU cache can be modeled by splitting it into groups of cache lines with the same maximum age. Volatile data is treated as a separate group of data where the maximum age is determined by the *total* fetch rate into the cache. Each application's sticky data is allowed to stay in the cache as sticky data as long as its maximum age is

lower than the maximum age of any application's volatile data, otherwise it becomes volatile. This leads to the following requirement, which must hold for every application, n, at steady state:

$$\mathrm{a}_n^{\mathrm{s}} < \mathrm{A}^{\mathrm{v}} \tag{6}$$

If the requirement does not hold for an application, its sticky data that is too old to remain sticky becomes volatile.

Volatile data in the cache can be modeled using the model derived for random replacement (Equation 1), that is:

$$\mathrm{f}_n = \mathrm{F}\,\frac{\mathrm{c}_n^{\mathrm{v}}}{\mathrm{C}^{\mathrm{v}}} \tag{7}$$

The total amount of cache an application has access to is the sum of its sticky and volatile cache allocations:

$$\mathrm{c}_n = \mathrm{c}_n^{\mathrm{s}} + \mathrm{c}_n^{\mathrm{v}} \tag{8}$$

Using the requirements defined above, we can find a sharing solution using a numerical solver. The solver starts with an initial guess, wherein the application that starts first has access to the entire cache[5] and the other applications do not have access to any cache. The initial guess corresponds to the state of the cache just before a new application is started, which enables us to find the correct solution if the application mix has multiple solutions.

The solver then lets all applications compete for cache space by enforcing the age requirement between sticky and volatile cache lines. If the age requirement can not be satisfied for an application, the solver shrinks that application's cache allocation until the remaining sticky data satisfies the age requirement. If multiple applications fail to satisfy the age requirement, we shrink the application with the oldest cache lines. The cache freed by the shrinking operation is then distributed among *all* applications by solving the sharing equations for the volatile part of the cache.

The process of shrinking and growing the amount of cache allocated to the applications is repeated until the solution stabilizes (i.e., no application changes its cache allocation significantly).

Example 5: Assume that we run application Z from Figure 3 and X from Figure 2. Z starts first. The solver will then make the following decisions (illustrated in Figure 4):

[5]The start order can be generalized to more than two applications by first determining the sharing for two applications and using that as the initial guess when start the next application and so on.

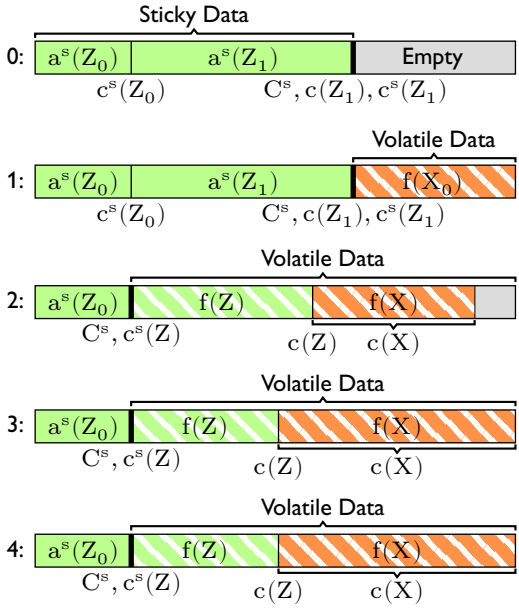


Figure 4: Solver steps for determining cache sharing when running example applications X and Z from Figure 2 and Figure 3 together. (See Example 5.)

0. Since Z starts first, it is given access to all cache and all its sticky data will fit in the cache. This is the initial guess.

1. X starts and the random sharing equations are solved for the volatile part of the cache. X fills the entire volatile part of the cache with its data since Z has a fetch rate of 0.

2. Based on the access rates and group sizes, we compute that Z's oldest sticky cache lines are now older than the oldest volatile cache line (i.e., the age requirement does not hold). Z can therefore not keep all its sticky data in the cache. The solver decides to shrink Z until the age condition can be satisfied. The condition is satisfied when the second sticky group becomes volatile.

3. Sharing in the volatile part of the cache is updated using the random model.

4. When the age conditions are satisfied, applying the random model to the volatile part of the cache does not change cache allocations and a stable solution is found.

□

	L1	L2	Memory
Latency	3 cycles	30 cycles	200 cycles
Size	64 kB	8 MB	
Line Size	64 B	64 B	
Associativity	16	16	

Table 1: M5 Simulator parameters

3 Evaluation (Simulator)

Experimental Setup

To evaluate the quality of the model, we simulated a simple in-order quad-core processor without prefetching using M5 [1]. The simulated processor implemented a snooping MOESI protocol with all L1 caches connected to a shared L2 cache through a common bus. The simulator does not enforce inclusion between cache levels. The detailed parameters are listed in Table 1.

To obtain the input data for the model, we simulated each application running in isolation and changed the L2 size from 512 kB to 8 MB in steps of 512 kB and measured its cache behavior.

We evaluated our models against three different L2 replacement policies: random replacement, LRU, and the pseudo-LRU algorithm used in the Intel Nehalem microarchitecture [6]. We included the pseudo-LRU policy to determine if our LRU model generalizes to the hardware we use for evaluating the model in Section 4. In our experiments, this pseudo-LRU algorithm behaved very similar to normal LRU, so we will limit our discussion to random and LRU.

We selected benchmarks with time-stable behavior from SPEC CPU2006 and PARSEC, and tried to select applications with a wide variety of fetch rate behaviors. Applications with high fetch rates or fetch rates that change significantly when their cache allocation changes are particularly interesting because they are affected by cache contention the most. To avoid unstable start-up behavior, all benchmarks were fast forwarded 85 billion instruction before starting the simulation. The subsequent 2 billion memory accesses were used to drive the simulation.

To further stress our models, we included the two classes of micro-benchmarks shown in Figure 5. The first class, *block*, repeatedly accesses its data in a sequential order. This behavior causes the fetch ratio for LRU caches to drop sharply when the cache size is larger than the data set size. The block benchmark is particularly challenging for the LRU model for two reasons: First, since it has a sharp edge on the fetch ratio curve, estimating group ages is hard as it involves taking a derivative of the curve

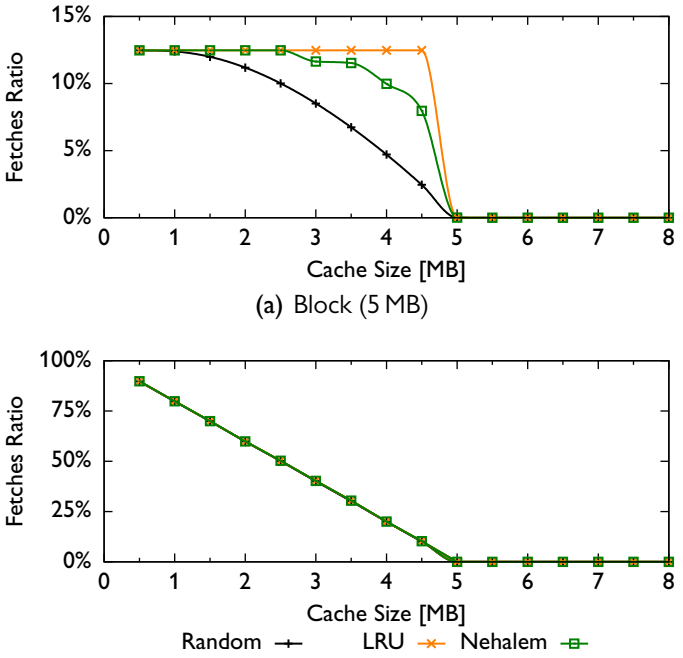


Figure 5: Shared cache fetch ratios for the block and random microbenchmarks.

at the point of the sharpest drop off. Second, as seen in Example 2, such benchmarks have a tendency to induce multiple stable sharing configurations in a given pair of benchmarks. The correct configuration generally depends on which application started first.

The *random* microbenchmark class accesses its entire data set randomly. This causes the fetch ratio to decrease linearly with cache size. An interesting observation is that all three replacement policies behave the same in this case. One of the main reasons to include this benchmark is that its steep fetch ratio curve means that a small error in estimating cache allocation will translate into a large error in fetch ratio. For example, a 1 MB error in predicted cache allocation would lead to a fetch ratio error of 20 percentage points, which is larger than the fetch ratio of most normal applications.

We ran all pairs of the following benchmarks from PARSEC: *blackscholes*, *bodytrack*, *streamcluster*; SPEC CPU2006: *astar*, *LBM*, *leslie3d*, *libquantum*, *soplex*; and the following microbenchmarks: *block (3 MB, 5 MB, 7 MB)*, *random (3 MB, 5 MB, 7 MB)*. Since the simulation time needed to simulate all possible combinations of four applications would be prohibitive, we limited our study to the groups shown in Table 2.

App 0	App 1	App 2	App 3
block 5 MB	random 3 MB	streamcluster	bodytrack
bodytrack	soplex	astar	lbm
bodytrack	streamcluster	blackscholes	lbm
lbm	leslie3d	astar	bodytrack
libquantum	block 5 MB	random 5 MB	random 7 MB
libquantum	lbm	astar	bodytrack
random 5 MB	streamcluster	astar	leslie3d
random 7 MB	lbm	leslie3d	bodytrack
random 3 MB	block 5 MB	lbm	astar
streamcluster	leslie3d	soplex	bodytrack

Table 2: Mixes of four applications

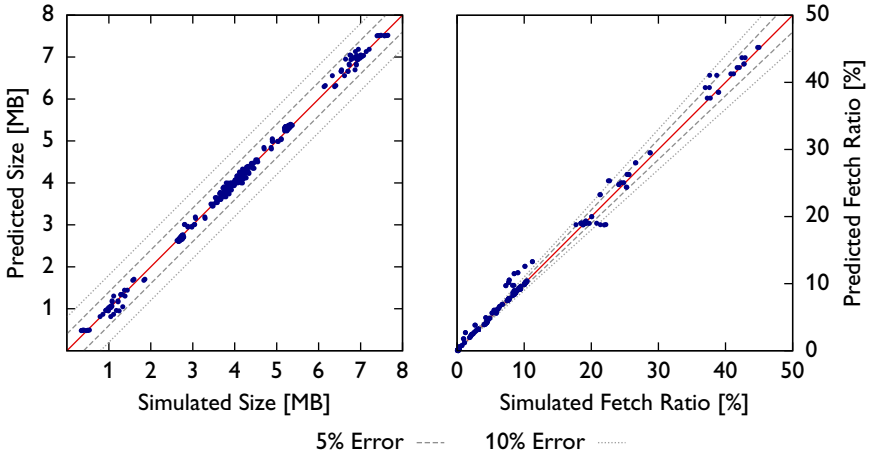
Simulation Results

Random Replacement

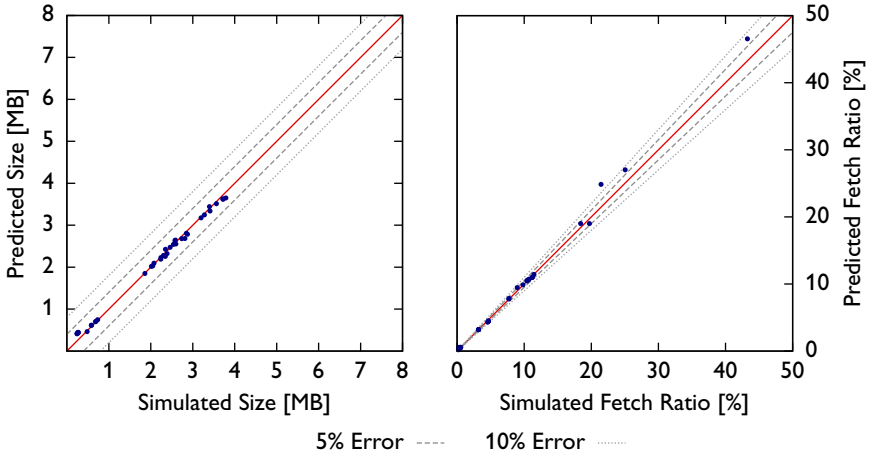
We simulated a random replacement cache and measured the cache allocation and fetch ratio per co-scheduled application. Figure 6(a) shows the predicted cache size versus simulated cache size and predicted fetch ratio versus simulated fetch ratio for all pairs of applications. The better a prediction, the closer it is to the diagonal. As seen in the figure, there is an excellent agreement between the amount of cache used by the applications and that predicted by the random model. The average error in cache size prediction as a fraction of the total cache size was 0.8%.

Some applications, such as the microbenchmarks in Figure 5, change their fetch ratio significantly when there is only a slight change in cache size. The effect an error in cache size has on the memory system will therefore depend on the shape of an application's miss ratio curve. In order to more accurately assess how the model predicts cache performance, we also evaluated how well the model predicts fetch ratio. We define the relative fetch ratio error as the absolute difference in predicted and simulated fetch ratio over the simulated fetch ratio. It makes little sense to look at relative errors for benchmarks with small fetch ratios, since with a fetch ratio close to zero, even an insignificant error will cause the relative error to explode. Excluding benchmarks with a simulated fetch ratio less than 0.5%, we measure a relative fetch ratio error of 6.1%. The average absolute error for the excluded benchmarks was 0.04%, which corresponds to an insignificant difference in performance.

As seen in Figure 6(b), groups of four applications can be predicted with similar accuracy. In this case, the average error in cache size was 0.9% and the average relative error in fetch ratio was 3.3%.

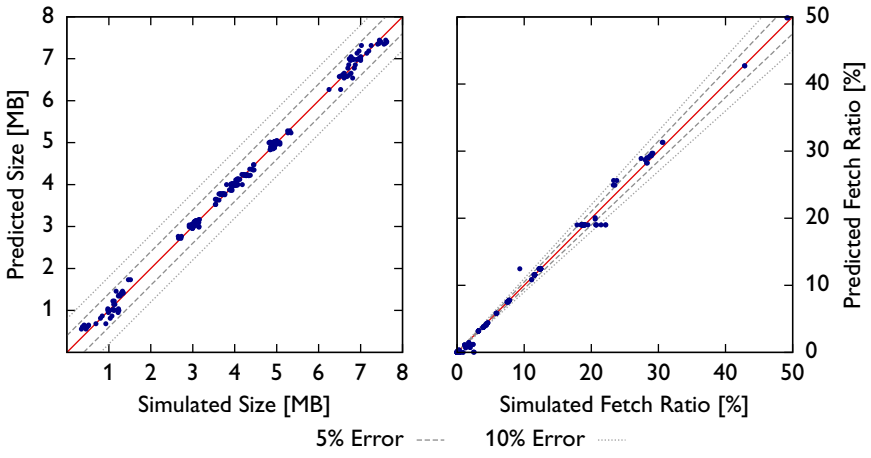


(a) Two co-scheduled benchmarks

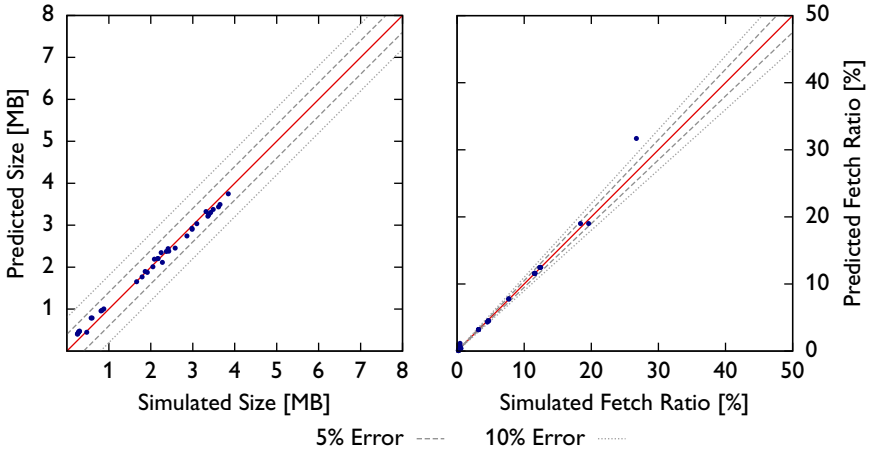


(b) Four co-scheduled benchmarks

Figure 6: Predicted versus simulated sharing and fetch ratio for *random replacement*.



(a) Two co-scheduled benchmarks



(b) Four co-scheduled benchmarks

Figure 7: Predicted versus simulated sharing and fetch ratio for *LRU replacement*.

LRU Replacement

Figure 7(a) shows the predicted and simulated behavior for pairs of applications with LRU replacement. The scatter plot compares the simulator solutions with their predicted counterparts. The average error in predicted cache size was 0.9% and the average relative error for the fetch ratio prediction was 5.4%. Similar to the random replacement case, we excluded applications with a fetch ratio lower than 0.5% from the fetch ratio average. The average absolute error for the excluded benchmarks was 0.05%. The model accurately solves the more complex task

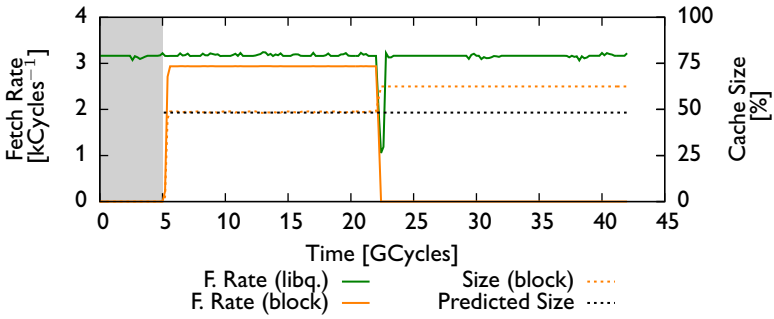


Figure 8: Fetch rate and cache sharing as a function of time for libquantum and the 5 MB block microbenchmark. The shaded part of the graph is the warm-up period where libquantum is executes in isolation.

of modeling cache sharing for LRU caches, even the microbenchmarks with sharp edges in their fetch ratio curves can be handled accurately.

As seen in Figure 7(b), the average error for groups of four applications is similar to when modeling random replacement. The average error in cache size in this case is 1.3% and the average relative error in fetch ratio is 4.2%.

The solver typically finds a solution within 5 to 10 iterations. Our prototype Python-based solver normally finds a solution in less than 100 ms.

Multiple LRU Sharing Solutions

As seen in Section 2, some combinations of applications can result in multiple stable sharing configurations. It turns out that such benchmark combinations are uncommon, and we only observed such behavior for microbenchmarks running in the simulator. In order for a pair of applications to have multiple stable sharing configurations, at least one of the applications must have a sharp knee in its fetch rate curve. In that case, the configuration the simulator finds will depend on the start order of the applications. To find such application pairs, we ran every application pair twice, starting one of the applications 5 billion cycles after the other.

Out of the 98 benchmark pairs, the simulator found multiple stable solutions in three cases, all involving the 5 MB block microbenchmark. The model accurately found both solutions in all of these cases by modeling the start order of the applications.

In two benchmark pairs, the simulator found a stable solution, but later switched to a different solution. This can occur for applications, which despite having fairly time stable behavior, have short hiccups where their fetch rate temporarily drops. Figure 8 shows the 5 MB block microbenchmark running together with libquantum. In this case,

libquantum started first and was allowed to execute in isolation for 5 billion cycles before the block microbenchmark was started. When the block application starts, its fetch rate immediately rises and stays high since it is unable to make its data sticky. Later libquantum has a short drop in its fetch rate which allows the block microbenchmark to install its entire data set into the cache and stabilize at the second solution. The model, being unaware of libquantum's time-varying behavior, predicts that the simulator will stay in the first solution.

In addition to the five benchmark pairs where the simulator found multiple solutions, our model found an additional solution in four other benchmark pairs. As in the cases where the simulator found multiple solutions, all of these cases involved the 5 MB block microbenchmark. The main reason for the additional solutions found by the model is input data limitations. The model uses sparse data collected for a limited set of cache sizes for each benchmark. This makes it feasible to apply the model to real-world systems, but causes numerical problems for benchmarks with sharp edges in their fetch rate curves. For example, inaccuracies in the input data caused the model to predict two sharing configurations when running the 5 MB block microbenchmark together with the 7 MB random microbenchmark.

4 Evaluation (Hardware)

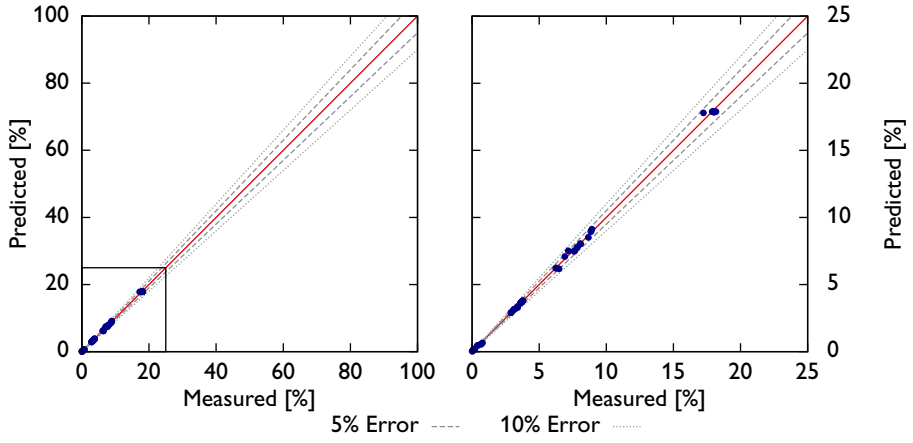
Experimental Setup

Our evaluation system consisted of a 2.4 GHz Intel Xeon E5620 system (Westmere) with 4 cores and 6 GB DDR3 memory. Each core has a private 32 kB L1 data cache and a private 256 kB L2 cache. All four cores share a 12 MB 16-way L3 cache with a pseudo-LRU replacement policy.

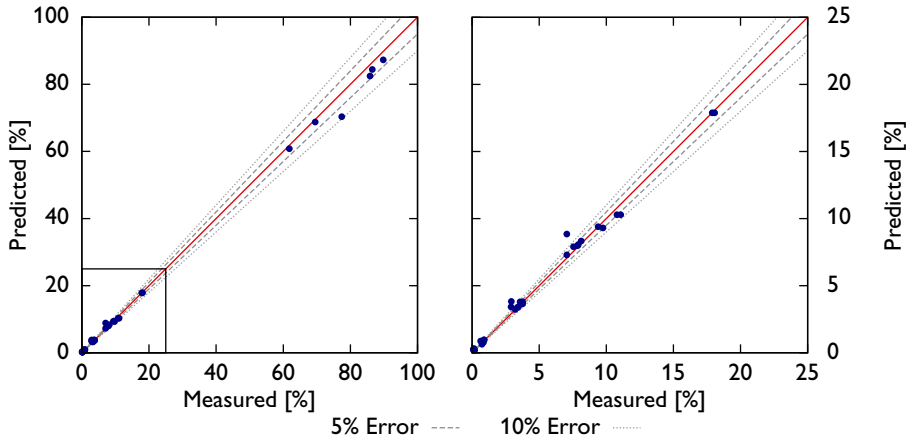
Our cache sharing model requires information about application fetch rate, access rate and hit ratio as a function of cache size. We used Cache Pirating [6] to measure this data for different cache sizes in steps 16 steps of 768 kB (the equivalent of one way) up to 12 MB.

We used the same benchmarks in the hardware study and the simulation study. However, we increased the size of the microbenchmarks' data set by 50% to better stress the 50% larger shared last-level cache.

While measuring the microbenchmarks, we discovered that the Cache Pirate slightly overestimates working set sizes. The average error in working set for the random microbenchmark was 256 kB due to the Pirate application and the monitoring framework using some of the shared cache. We compensated for this error by shrinking the total cache size in the model by this amount and offsetting the input data.



(a) Two co-scheduled benchmarks



(b) Four co-scheduled benchmarks

Figure 9: Predicted vs. measured fetch ratio for applications running on an Intel Xeon E5620 based system.

Results

Figure 9(a) compares the predicted and measured fetch ratio of pairs of co-scheduled applications. We do not show the amount of cache allocated to each application since there is no accurate way to measure this on the hardware. Unlike the simulator, we only found one solution for each benchmark pair. We believe that the reason for this is that the fetch rate curves of the applications running on our reference hardware do not have as sharp edges as in the simulator. The average relative error

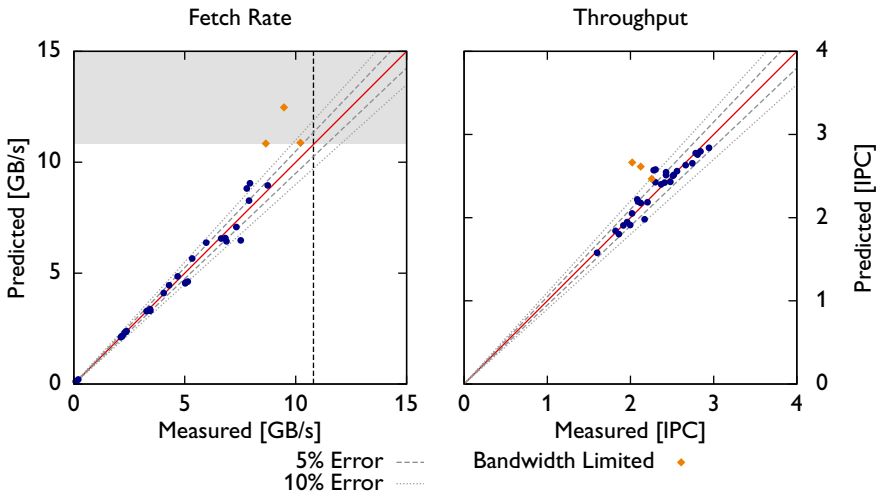


Figure 10: Estimated bandwidth vs. measured bandwidth and estimated IPC vs. measured IPC for pairs of benchmarks running on an Intel Xeon E5620 based system. The gray area indicates the bandwidth limit.

in predicted fetch ratio was 7.1%. Similar to the simulator evaluation, we excluded applications with a fetch ratio lower than 0.5% from the average. Figure 9(b) shows the results for groups of four co-scheduled applications. The average fetch ratio error was 6.7%.

Estimating Bandwidth and Throughput

Knowing how applications share cache allows us to predict other performance metrics, such as bandwidth requirements and throughput. The data measured using Cache Pirating contains information about each application's individual CPI and bandwidth requirement as a function of cache size. Since we can predict each application's cache allocation, we can trivially find its *bandwidth demand*. Assuming that a mix is not bandwidth limited, we can calculate the combined IPC of the mix (throughput) and its expected bandwidth usage.

Knowing the combined bandwidth demand of an application mix can guide a scheduler to avoid mixes with bandwidth demands too close to the system's bandwidth limitation. For mixes well below that limitation, our throughput estimates should be accurate enough to find the best mixes.

We estimated the real-world bandwidth limit of our reference system to approximately 12 GB/s using the STREAM benchmark [12]. We consider an application mix to be bandwidth limited if it uses more than 90% of the maximum bandwidth.

Figure 10 compares the combined bandwidth and throughput of our estimation with the corresponding numbers measured on real hardware. As seen in the figure, as long as the estimated bandwidth is low enough (below 11 GB/s), our bandwidth estimate is quite accurate. Excluding the mixes with a too high bandwidth demand, we can predict the bandwidth of a mix with an average relative error of less than 5.9% and throughput with an average error less than 2.5%.

5 Related Work

Cache sharing models can be divided into two categories: trace driven and high-level data driven. The trace driven models generally use memory access traces or stack distance[6] traces. The benefit of using traces is that they contain detailed information about the execution. Unfortunately, acquiring a memory access trace is slow and storage intensive. Using high-level data, such as sampled memory accesses or statistics provided by performance counters, has become a common approach to reduce data collection overhead.

There are several models [2–4, 17] using stack distance traces. Chandra et al. [2] pioneered the field with a statistical model that estimates the probability that an access becomes a miss by prolonging its stack distance with the expected number of accesses performed by other applications. One drawback with their model is that it assumes that an application's execution rate is independent of the amount of cache it has access to. Chen and Aamodt [3, 4] extended Chandra's model by including variable execution rate. They also improved the accuracy of the model for low cache associativity by taking the access distribution across sets into account.

The method most similar to ours is CAMP [19] by Xu et al. They use high-level input data, similar to the input data used by our model to model sharing among pairs of applications. However, their model depends on a linear approximation of CPI as a function of fetch ratio. Such an approximation is often inaccurate for processors with out-of-order execution and prefetching. We do not need to model execution rate as this is implicit in our input data. Xu et al. also evaluate two simpler models, which assume that an application's cache share is either proportional to its access rate or its fetch rate. The latter is equivalent to the model we use for random caches, but is applied to LRU caches and approximates fetch rates using their linear execution rate model.

[6] A stack distance is the number of *unique* between two accesses to the same cache line.

Eklöv et al. proposed a statistical cache sharing model [5] using memory access samples, which can be measured with low overhead. They use a performance model similar to the one used by Xu et al. to estimate the relative execution rate of co-scheduled applications and merge the sampled access streams from each of them. Unfortunately, since they use a linear approximation of execution rate, they suffer from the same drawbacks as the model by Xu et al.

Two recent works focus on estimating how resource contention affects performance. Mars et al. [10] use a stress benchmark to induce contention in an application and then measure its slowdown. The slowdown is used as a contention-sensitivity metric which can be used to guide schedulers. Unlike our method, they do not try to estimate the performance of specific combinations of applications. Instead they focus on a general classification of applications as either being sensitive or insensitive to contention. The approach by Van Craeynest and Eeckhout [16] is more similar to our method in that they estimate the throughput of mixes of applications. A major difference between our methods is that they depend on a single high-fidelity simulation to generate the application profiles used by their model, whereas we measure our input data with low overhead on the target system.

6 Future Work

We are currently working on extending our models to handle time-varying application behavior. A simple approach would be to slice applications into time windows and estimate sharing between windows. Such an approach would work, however, the amount of data needed would most likely be prohibitively large. Instead, we envision using phase information, which would enable us to analyze larger regions of stable behavior.

Another exciting direction is to extend the model to more accurately predict throughput for bandwidth limited mixes. This, however, most likely requires a detailed analytical performance model of the processor or performance data as a function of bandwidth.

Acknowledgments

The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). This work was financed by the CoDeR-MP project and the UPMARC research center.

References

[1] Nathan L. Binkert, Ron G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. "The M5 Simulator: Modeling Networked Systems". In: *IEEE Micro* 26.4 (2006), pp. 52–60. DOI: 10.1109/MM.2006.82.

[2] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2005. DOI: 10.1109/HPCA.2005.27.

[3] Xi E. Chen and Tor M. Aamodt. "A First-Order Fine-Grained Multithreaded Throughput Model". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2009. DOI: 10.1109/HPCA.2009.4798270.

[4] Xi E. Chen and Tor M. Aamodt. "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors". In: *IEEE Transactions on Computers* PP.99 (2011). DOI: 10.1109/TC.2011.141.

[5] David Eklov, David Black-Schaffer, and Erik Hagersten. "Fast Modeling of Cache Contention in Multicore Systems". In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011, pp. 147–157. DOI: 10.1145/1944862.1944885.

[6] 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)*. 2011, pp. 165–175. DOI: 10.1109/ICPP.2011.15.

[7] F. N. Fritsch and R. E. Carlson. "Monotone Piecewise Cubic Interpolation". In: *SIAM Journal on Numerical Analysis* 17.2 (1980). DOI: 10.1137/0717021.

[8] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Jr Simon Steely, and Joel Emer. "Adaptive Insertion Policies for Managing Shared Caches". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 208–219. DOI: 10.1145/1454115.1454145.

[9] John D. C. Little. "A Proof for the Queuing Formula: $L = \lambda W$". In: *Operations Research* 9.3 (1961), pp. 383–387.

[10] Jason Mars, Lingjia Tang, and Mary Lou Soffa. "Directly Characterizing Cross Core Interference Through Contention Synthesis". In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011. DOI: 10.1145/1944862.1944887.

[11] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. "Contention Aware Execution". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. 2010. DOI: 10.1145/1772954.1772991.

[12] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, 1991–2007. URL: http://www.cs.virginia.edu/stream/.

[13] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2006, pp. 423–432. DOI: 10.1109/MICRO.2006.49.

[14] Andreas Sandberg, David Eklöv, 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)*. 2010. DOI: 10.1109/SC.2010.44.

[15] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. "Managing Shared L2 Caches on Multicore Systems in Software". In: *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. 2007.

[16] Kenzo Van Craeynest and Lieven Eeckhout. "The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011, pp. 26–37. DOI: 10.1109/IISWC.2011.6114194.

[17] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. "All-Window Profiling and Composable Models of Cache Sharing". In: *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2011. DOI: 10.1145/1941553.1941567.

[18] Yuejian Xie and Gabriel H Loh. "Dynamic Classification of Program Memory Behaviors in CMPs". In: *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*. 2008.

[19] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. "Cache Contention and Application Performance Prediction for Multi-Core Systems". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452065.

[20] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010. DOI: 10.1145/1736020.1736036.

PAPER III

Modeling Performance Variation Due to Cache Sharing in Multicore Systems

Andreas Sandberg
Andreas Sembrant
David Black-Schaffer
Erik Hagersten

©2013 IEEE. Reprinted, with permission, from *HPCA'13*, February 23–27, 2013.
DOI: 10.1109/HPCA.2013.6522315

Abstract— Shared cache contention can cause significant variability in the performance of co-running applications from run to run. This variability arises from different overlappings of the applications' phases, which can be the result of offsets in application start times or other delays in the system. Understanding this variability is important for generating an accurate view of the expected impact of cache contention. However, variability effects are typically ignored due to the high overhead of modeling or simulating the many executions needed to expose them.

This paper introduces a method for efficiently investigating the performance variability due to cache contention. Our method relies on input data captured from native execution of applications running in isolation and a fast, phase-aware, cache sharing performance model. This allows us to assess the performance interactions and bandwidth demands of co-running applications by quickly evaluating hundreds of overlappings.

We evaluate our method on a contemporary multicore machine and show that performance and bandwidth demands can vary significantly across runs of the same set of co-running applications. We show that our method can predict application slowdown with an average relative error of 0.41% (maximum 1.8%) as well as bandwidth consumption. Using our method, we can estimate an application pair's performance variation 213x faster, on average, than native execution.

1 Introduction

Shared caches in contemporary multicores have repeatedly been shown to be critical resources for performance [8, 15, 17, 23, 28]. A significant amount of research has investigated the impact of cache sharing on application performance [11, 12, 23, 30]. However, most previous research provides a single value for the slowdown of an application pair due to cache sharing and ignores the variability that occurs across multiple runs. This variability occurs due to different overlappings of application phases that occur when they are offset in time. As the different phases have varying sensitivities to contention for the shared cache, the result is a wide range of slowdowns for the same application pair.

In multicore systems, there can be large performance variations due to cache contention, since an application's performance depends on how its memory accesses are interleaved with other applications' memory accesses. For example, when running astar/lakes and bwaves from SPEC CPU2006, we observe an average slowdown of 8% for astar compared to running it in isolation. However, the slowdown can vary between 1% and 17% depending on how the two applications' phases overlap. Figure 1

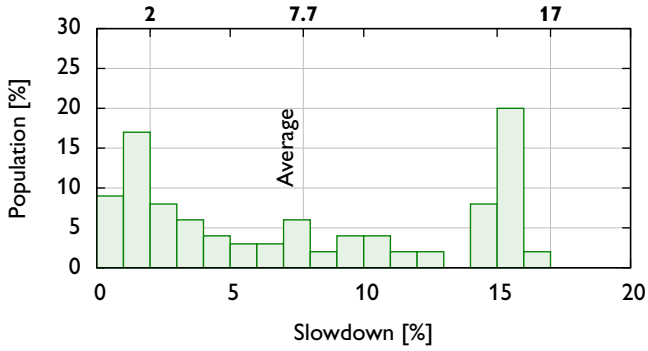


Figure 1: Performance distribution for astar co-running together with bwaves on an Intel Xeon E5620 based system. *Ignoring performance variability can be misleading, since the average (7.7%) hides the fact that the performance can vary between 1% and 17% depending on how the two applications' phases overlap.*

shows astar's slowdown distribution based on 100 runs with different offsets in starting times. A developer assessing the performance of these applications could draw the wrong conclusions from a single run, or even a few runs, since the probability of measuring a slowdown smaller than 2% is more than 25%, while the average slowdown is almost 8% and the maximum slowdown is 17%.

In order to accurately estimate the performance of a mixed workload, we need to run it multiple times and estimate its performance distribution. This is a both time- and resource-consuming process. The distribution in Figure 1 took almost seven hours to generate; our method reproduces the same performance distribution in less than 40 s.

To do this, we combine the cache sharing model proposed by Sandberg et al. [16], the phase detection framework developed by Sembrant et al. [19], and the co-execution phase optimizations proposed by Van Biesbrouck et al. [25]. This allows us to efficiently predict the performance and bandwidth requirements of mixed workloads. In addition, the input data to the cache model is captured using low-overhead profiling [7] of each application running in isolation. This means that only a small number of profiling runs need to be done on the target machine. The modeling can then be performed quickly for a large number of mixed workloads and runs.

The main contributions of this paper are:

- An extension to a statistical cache-sharing model [16] to handle time-dependent execution phases.

- A fast and efficient method to predict the performance variations due to shared cache contention on modern hardware by combining a cache sharing model [16] with phase optimizations [19, 25].
- A comparison with previous cache-sharing methods [16] demonstrating a 2.78× improvement in accuracy (the relative error is reduced from 1.14% to 0.41%) and a 3.5× reduction in maximum error (from 6.3% to 1.8%).
- An analysis of how different types of phase behavior impact the performance variations in mixed workloads.

2 Putting it Together

Our method combines and extends three existing pieces of infrastructure: a cache sharing model [16], a low-overhead cache analysis tool [7], and a phase detection framework [19]. In this section, we describe the different pieces and how we extend them.

Cache Sharing

We use the cache sharing model proposed by Sandberg et al. [16] for cache modeling. It accurately predicts the amount of cache used, CPI, and bandwidth demand for an application in a mixed workload of co-executing single-threaded applications. The input to the model is a set of independent *application profiles*. These profiles contain information about how the *miss rate* (misses per cycle) and *hit rate* (hits per cycle) vary for an application as a function of cache size. We use the Cache Pirating [7] technique (discussed below) to capture the model's input data.

The model conceptually partitions the cache into two parts with different reuse behavior. The model keeps frequently reused data safe from replacements, while less frequently reused data shares the remaining cache space proportionally to its application's miss rate. The partitioning between frequently reused data and infrequently reused data is an application property that is cache size dependent (i.e., the partitioning depends on how much cache an application receives). The model uses an iterative solver that first solves cache sharing for the infrequently reused data and then updates partitioning between frequently reused data and infrequently reused data.

The model however only works on phase-less applications where the average behavior is representative of the entire application. In practice, most applications have phases. To handle this, we extend the model by

slicing applications into multiple small time windows. As long as the windows are short enough, the model's assumption of constant behavior holds within the window. We then apply the model to a set of co-executing windows instead of data averaged across the entire execution.

Cache Pirating

The input to the cache sharing model is an application profile with information about cache miss rates and hit rates *as a function of cache size*. Traditionally, such profiles have been generated through simulation, but such an approach is slow and it is difficult to build accurate simulators for modern processor pipelines and memory systems. Instead, we use Cache Pirating [7] to collect the data. Cache Pirating solves both problems by measuring how an application behaves as a function of cache size on the target machine with very low overhead.

Cache Pirating uses hardware performance monitoring facilities to measure target application properties at runtime, such as cache misses, hits, and execution cycles. To measure this information for varying cache sizes, Cache Pirating co-runs a small cache intensive stress application with the target application. The amount of cache available to the target application is then varied by changing the cache footprint of the stress application. This allows Cache Pirating to measure any performance metric exposed by the target machine as a function of available cache size.

The cache pirate method produces average measurements for an entire application run. This is illustrated in Figure 2(a). It shows CPI as a function of cache size for astar. The solid black line (Average) is the output produced with Cache Pirating.

Just examining the average behavior can however be misleading since most applications have time-dependent behavior. Figure 2(b) instead shows astar's CPI as a function of both time and cache size. As seen in the figure, the application displays three different *phases* of behavior: some parts of the application execute with a very high CPI (phase A & phase B), while other parts execute with a very low CPI (phase C). This information is lost unless time is taken into account.

In this paper, we extend the cache pirate method to produce time-dependent data by dividing the execution into *sample windows* by sampling the performance counters at regular intervals.

Phase Detection

A naive approach to phase-aware cache modeling would be to model the effect of every pair of measured input sample windows. However, to make the analysis more efficient, we incorporate application phase

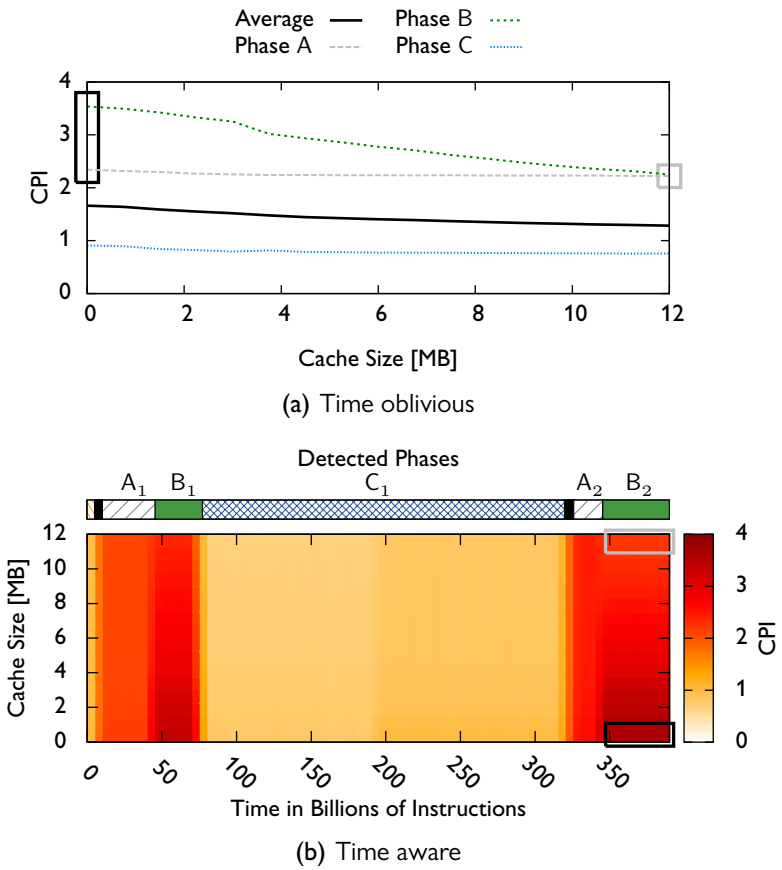


Figure 2: Performance (CPI) as a function of cache size as produced by Cache Pirating. Figure (a) shows the time-oblivious application average as a solid line. Figure (b) shows the time-dependent variability of the cache sensitivity and the phases identified by ScarPhase above. The behavior of the three largest phases vary significantly from the average as can be seen by the dashed lines in Figure (a).

information. This enables us to analyze multiple sample windows with similar behavior at the same time, which reduces the number of times we need to invoke the cache sharing model.

We use the ScarPhase [19] library to detect and classify phases. ScarPhase is an execution-history based, low-overhead (2%), online phase-detection library. It examines the application's execution path to detect hardware independent phases [14, 22]. Such phases can be readily missed by performance counter based phase detection, while changes in executed code reflect changes in many different metrics [5, 9, 18, 20–22]. To leverage this, ScarPhase monitors what code is executed by dividing the application into windows and using hardware performance

counters to sample which branches execute in a window. The address of each branch is hashed into a vector of counters called a basic block vector (BBV) [21]. Each entry in the vector shows how many times its corresponding branches were sampled during the window. The vectors are then used to determine phases by clustering them together using an online clustering algorithm [6]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

The phases detected by ScarPhase can be seen in the top bar in Figure 2(b) for astar, with the longest phases labeled. This benchmark has three major phases; A, B and C, all with different cache behaviors. To highlight the differences in CPI, we have plotted the average CPI of each phase in Figure 2(a). For example, phase A runs slower than C, since it has a higher CPI. Phase B is more sensitive to cache-size changes than phase A since phase B's CPI decreases with more cache.

The same phase can occur several times during execution. For example, phase A recurs two times, once in the beginning and once at the end of the execution. We refer to multiple repetitions of the same phase as *instances* of the same phase, e.g., A_1 and A_2 in Figure 2(b).

In addition, Figure 2(b) also demonstrates the limitation of defining phases based on changes in hardware-specific metrics. For example, the CPI is very similar from 325 to 390 billion instructions when using 12 MB of cache (the gray rectangle), but clearly different when using less than 4 MB (the black rectangle). This difference is even more noticeable in Figure 2(a) when comparing phase A and B. A phase detection method looking at only the CPI would draw the conclusion that phase A and B are the same phase when the application receives 12 MB of cache, while in reality they are two very different phases. It is therefore important to find phases that are independent of the execution environment (e.g., co-scheduling).

3 Time Dependent Cache Sharing

The key difficulty in modeling time-dependent cache sharing is to determine which parts of the application (i.e., sample windows or phases) will co-execute. Since applications typically execute at different speeds depending on phase, we can not simply use the *i*th sample windows for each application since they may not overlap. For example, consider two applications with different executions rates (e.g., CPIs of 2 and 4), executing sample windows of 100 million instructions. The slower application with a CPI of 4 will take twice as long to finish executing its sample windows as the one with a CPI of 2. Furthermore, when they share a cache they impact each others execution rates. Instead, we advance time as follows:

1. Determine the cache sharing using the model for the current windows and the resulting CPI for each application due to its shared cache allocation.

2. Advance the fastest application (i.e., the one with lowest CPI) to its next sample window. The slower applications will not have had time to completely execute their windows. To handle this, their windows are first split into two smaller windows so that the first window ends at the same time the fastest applications sample window. Finally, time is advanced to the beginning of the latter windows.

This means that the cache model is applied several times per sample window, since each window is usually split at least twice. For example, when modeling the slowdown of astar co-executing together with bwaves, we invoke the cache sharing model roughly 13 000 times while astar only has 4 000 sample windows by itself.

We refer to the method described so far as the *window-based method* (Window) in the rest of paper. In the rest of this section, we will introduce two more methods, the *dynamic-window-based method* (Dynamic Window) and the *phase-based method* (Phase), which both use phase information to improve the performance by reducing number of times the cache sharing model needs to be applied[1].

Dynamic-Windows: Merging Sample-Windows

To improve performance we need to reduce the number of times the cache sharing model is invoked. To do this, we merge multiple adjacent sample windows belonging to the same phase into larger windows, a dynamic window. For example, in astar (Figure 2), we consider all sample windows in A_1 as one unit (i.e., the average of the sample windows) instead of looking at every individual sample window within the phase. Merging consecutive windows within a phase assumes that the behavior is stable within a that instance (i.e., all windows have similar behavior). This is usually true and does not significantly affect the accuracy of the method. However, compared to the window-based method, it is dramatically faster. For example, modeling astar running together with bwaves we reduce the number of times the cache sharing model is used from 13 000 to 520, which leads to 25× speedup over the window-based method.

[1] The cache sharing model is implemented in Python and takes approximately 88 ms per invocation on our reference system (see Section 4).

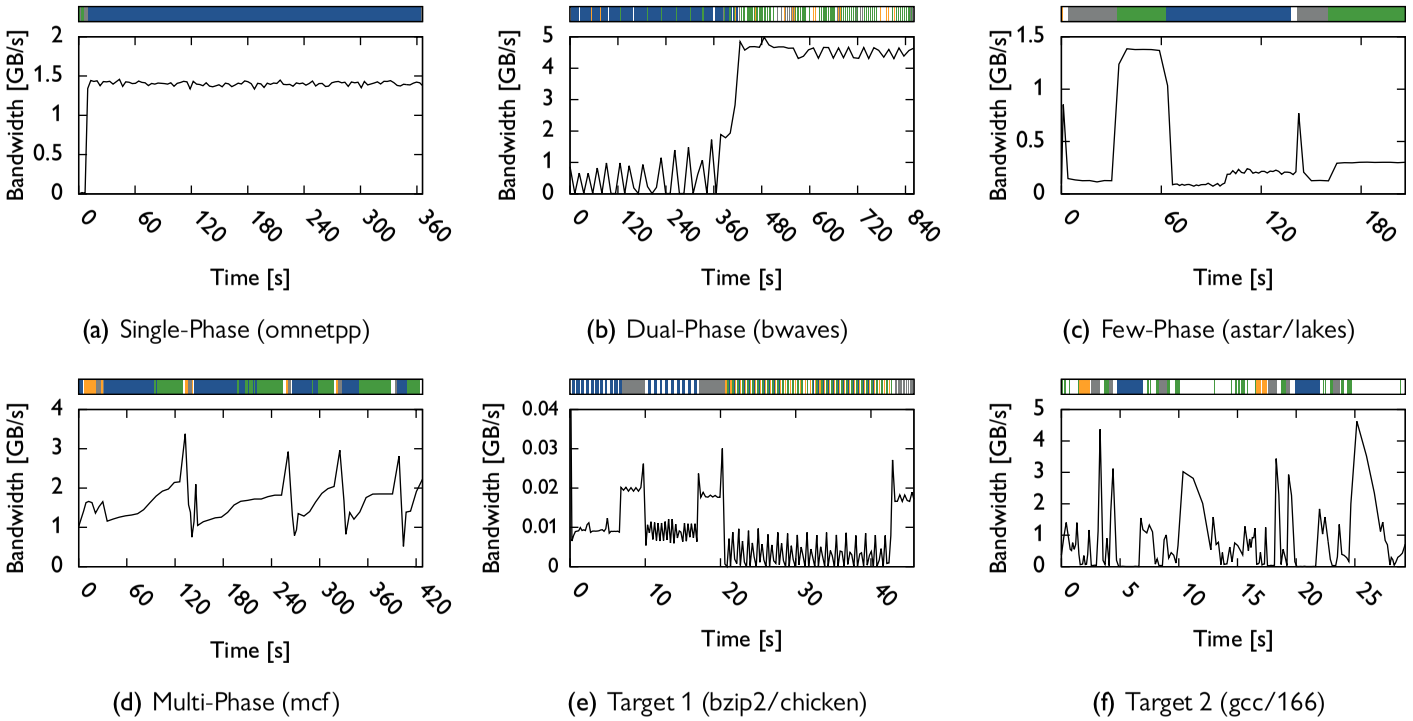


Figure 3: Bandwidth usage across the whole execution of our six benchmark applications, including the four interference applications. Detected phases are shown above. The Single-Phase, Dual-Phase, Few-Phase, and Multi-Phase behavior is clearly visible for the interference applications.

Phase: Reusing Cache-Sharing Results

The performance can be further improved by merging the data for all instances of a phase. For example, when considering astar (Figure 2), we consider all phase instances of A (i.e., A_1 + A_2) as one unit. This makes the assumption that all instances of the same phase have similar behavior in an execution. This is not necessarily true for all applications (e.g., same function but different input data), but works well in practice.

Looking at whole phases does not change the number of times we need to determine an applications cache sharing. It does however enables us to reuse cache sharing results for co-executing phases that reappear later [25]. For example, when astar's phase A_1 co-executes with bwave's phase B, we can save the cache sharing result, and later reuse the result if the second instance (A_2) co-executes with bwaves B.

In the example with astar and bwaves, we can reuse the results from previous cache sharing solutions 380 times. We therefore only need to run the cache sharing model 140 times. The performance of the phase-based method is highly dependent on an application's phase behavior, but it normally leads to a speed-up of 2–10× over the dynamic-window method.

The main benefit of the phase-based method is when determining performance variability of a mix. In this case, the same mix is run several times with slightly different offsets in starting times. The same co-executing phases will usually reappear in different runs. For example, when modeling 100 different runs of astar and bwaves, we need to evaluate 1 400 000 co-executing windows, but with the phase-based method we only need to run the model 939 times.

In addition to reducing the number of model invocations, using phases reduces the amount of data needed to run the model. Instead of storing a profile per sample window, all sample windows in one phase can be merged. This typically leads to a 100–1000× size reduction in input data. For example, bwaves, which is a long running benchmark with a large profile, reduces its profile size from 57 MB to 82 kB.

4 Evaluation

To evaluate our method we compare the overhead and the accuracy against results measured on real hardware. We ran each *target* application together with an *interference* application and measured the behavior of the target application. In order to measure the performance variability, we started the applications with an offset by first starting the interference

	REF	Single-Phase (omnetpp)				Dual-Phase (bwaves)				Few-Phase (astar)				Multi-Phase (mcf)			
	Time	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup
	ISO	W	D	P	P	W	D	P	P	W	D	P	P	W	D	P	P
astar	5.9h	723k	302	84.0	2.1k×	1.4M	123k	938	88.7×	797k	1.8k	460	506×	575k	6.9k	465	435×
bwaves	24.3h	4.9M	62.0k	174	113×	7.3M	2.4M	2.0k	54.9×	5.2M	249k	1.0k	105×	4.3M	973k	1.1k	98.5×
bzip2	1.3h	242k	3.7k	63.0	103×	383k	475k	711	31.6×	272k	17.0k	373	59.7×	213k	64.0k	414	58.3×
gcc	0.8h	119k	870	140	133×	209k	203k	1.5k	18.9×	139k	4.5k	759	38.4×	101k	16.0k	786	36.6×
mcf	12.3h	1.0M	1.2k	97.0	1.9k×	2.3M	578k	1.1k	86.0×	1.2M	8.6k	518	798×	702k	30.9k	589	594×
omnetpp	10.3h	1.1M	33.0	14.0	18.6k×	2.2M	52.8k	172	110×	1.2M	358	85.0	2.5k×	856k	1.4k	99.0	1.8k×
average	9.2h	1.3M	11.4k	95.3	**695×**	2.3M	634k	1.1k	**54.9×**	1.5M	46.9k	540	**250×**	1.1M	182k	582	**215×**
													global average	1.6 M	219 k	572	**213×**

Table 1: Performance statistics for 100 runs with different starting time offsets. The number of model invocations for the three methods (W:Window, D:Dynamic-Window, and P:Phase) is shown along with the speedup for running the phase-based model vs. reference executions on the hardware. We discuss the highlighted results in the text. The model-based approach is on average 213× faster than hardware execution.

application and then waiting for it to execute a predefined number of instructions before starting the target. We then restarted the interference application if it terminated before the target.

In order to get an accurate representation of the performance, we ran each experiment (target-interference pair) 100 times with random start offsets for the target. We used the same starting time offsets for both the hardware reference runs and for the modeled runs.

Experimental Setup

We ran the experiments on a 2.4 GHz Intel Xeon E5620 system (Westmere) with 4 cores and 3×2 GB memory distributed across 3 DDR3 channels. Each core has a private 32 kB L1 data cache and a private 256 kB L2 cache. All four cores share a 12 MB 16-way L3 cache with a pseudo-LRU replacement policy.

The cache sharing model requires information about application fetch rate, access rate and hit rate as a function of cache size and time. We measured cache-size dependent data using cache pirating in 16 steps of 768 kB (the equivalent of one way) up to 12 MB, and used a sample window size of 100 million instructions.

Benchmark Selection

In order to see how time-dependent phase behavior affects cache sharing and performance, we selected benchmarks from SPEC CPU2006 with interesting phase behavior. In addition to interesting phase behavior, we also wanted to select applications that make significant use of the shared L3 cache. For our evaluation, we selected four interference benchmarks that represent four different phase behaviors: Single-Phase (omnetpp), Dual-Phase (bwaves), Few-Phase (astar/lakes) and Multi-Phase (mcf).

Figure 3 shows the interference applications' bandwidth usage (high bandwidth indicates significant use of the shared L3 cache), and the detected phases. In addition to the interference benchmarks, we selected two more benchmarks, gcc/166 and bzip2/chicken, that we only use as targets. These benchmarks have a lower average bandwidth usage than the interference benchmarks, but they are still sensitive to cache contention. For the evaluation, we ran all combinations of the six applications as targets vs. each of the four interference applications.

Performance: Speedup

Table 1 presents the performance of the three methods, Windows (W), Dynamic-Windows (D) and Phase (P) per interference application. The

Model Invocations columns shows the number of times the cache sharing model was invoked. For example, when astar co-executes with bwaves (see the highlighted area), the cache model is invoked 1 400 000, 123 000, and 938 times for the window, the dynamic-window and the phase-based method respectively.

The reference column (REF) shows the execution time to run each target in isolation 100 times. For example, on our system, it takes 5.9 hours to run astar 100 times. The speedup column shows the speedup to model 100 co-executed runs with the phase-based method compared to running the target 100 times[2]. For example, it is 88.7× faster to model 100 co-executions of astar with bwaves than to run astar 100 times in isolation.

Single-Phase: As expected, the speedup is greatest for omnetpp since it consists of just one phase. The dynamic-window method can therefore use a single large window for the whole execution. The phase-based method can then easily reuse cache sharing results whenever the target executes more instances of a phase. The geometric mean of the speedup is 695×, the highest of the four interference benchmarks.

Dual-Phase: In a similar sense, we should expect a high speedup for bwaves as well. However, bwaves executes much longer than the other interference benchmarks. So, even though the phase-based method reduces the number of times the cache sharing model is used, it has a high overhead from reading through all application profile data. On average the speedup is only 54.9×.

Few-Phase and Multi-Phase: The three methods have roughly the same performance for astar and mcf, and fall in between Single-Phase and Dual-Phase in performance. On average the speedup is 250× and 215× for astar and mcf respectively.

It is clear from the table that the phase-based method provides the best performance for all benchmarks, with an average speedup of 213× for all interference benchmarks[3]. Next, we will evaluate the accuracy of three methods to determine if there are any trade-offs associated with the phase-based method.

Accuracy: Average Slowdown Error

Figure 4 presents the relative error when predicting the *average* slowdown for the three methods. On average, the windows-based method

[2]The speedup excludes the time to collect the applications profiles with Cache Pirating. That data is collected only once, and is then used in all the application mixes, and hence not included.

[3]Note that the speedup numbers are based on our Python implementation. A C/C++ implementation would most likely result in greater speedups.

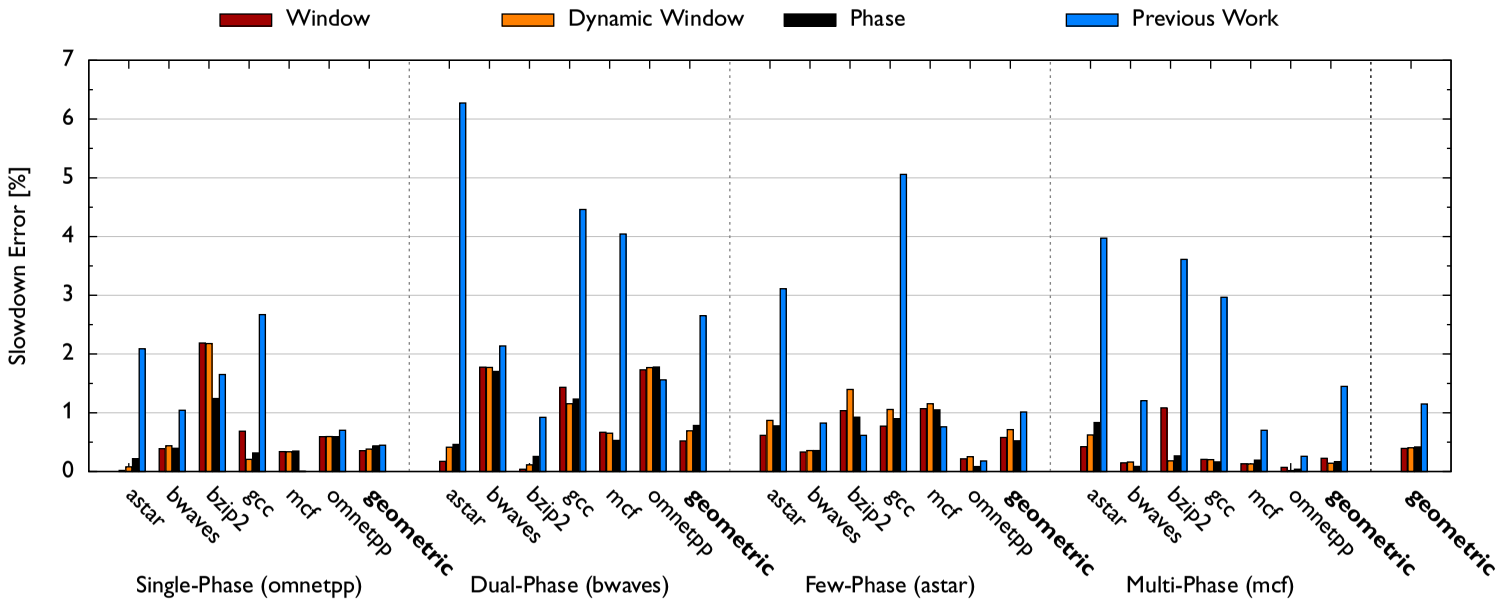


Figure 4: Relative error in predicted slowdown for the three methods and the previous phase-oblivious cache sharing model [16]. *This shows that the phase-based method (the fastest) can be used without lowering the accuracy. In addition, ignoring applications phase behavior will result in noticeable larger prediction errors (e.g., a 6.3% error for the phase-oblivous method when astar co-executes with bwaves).*

has an error of 0.39% and a maximum error of 2.2% (bzip2 + omnetpp), while the phase-based method has an average error of 0.41% and a maximum of 1.8% (omnetpp + bwaves). We can therefore safely use the much faster phase-based method without sacrificing accuracy. In the rest of this paper, we will therefore only look at the phase-based method.

In addition to the three methods, the figure also includes the error of using the previous phase-oblivious cache sharing model [16] that does not take time-varying phase behavior into consideration. The phase oblivious method has a reasonably good accuracy for omnetpp since it only has one phase. However, the error is noticeably larger for applications with more phase behavior. For example, 6.3% when astar co-executes with bwaves. This indicates that even when considering average slowdowns (i.e., ignoring variability), it is still important to consider the time-varying behavior and how the two applications' phases overlap.

On average, our phase-based method is 2.78× more accurate than previous work, with an average error of 0.41% instead of 1.14% and a 3.5× lower maximum error of 1.8% instead of 6.3% (astar + bwaves).

Performance Variability

The average slowdown is a good metric for evaluating the overall accuracy of the different methods. However, it does not take performance variation into consideration. We therefore use another more descriptive metric, the *cumulative slowdown distributions* (CDF), to display the performance variations. Figure 5 presents the CDF for the phase-based method along with results from the reference hardware runs. The graphs with white backgrounds highlight the benchmark pairs with interesting performance variations.

The cumulative slowdown distributions can be interpreted as showing the probability for a certain maximum slowdown. For example, in Figure 5b, when astar is co-running together with bwaves, it has a 50% probability of having a slowdown less than 6.5%. At the same time, there is a 25% probability that the slowdown is larger than 15%.

Single-Phase. The CDF curves are mostly flat when omnetpp is used as a interference application. For example, Figure 5e, where bwaves is co-running with omnetpp, the curve is basically flat at 1% slowdown. This means that there are no performance variations for bwaves co-running with omnetpp, which is to be expected since omnetpp does not have any time-varying behavior.

Dual-Phase. In contrast to omnetpp, bwaves has two phases with very different behavior. The higher bandwidth usage in the second phase indicate that it uses a larger part of the L3 cache, and will thus impose a larger slowdown on the target application. The effect on the target

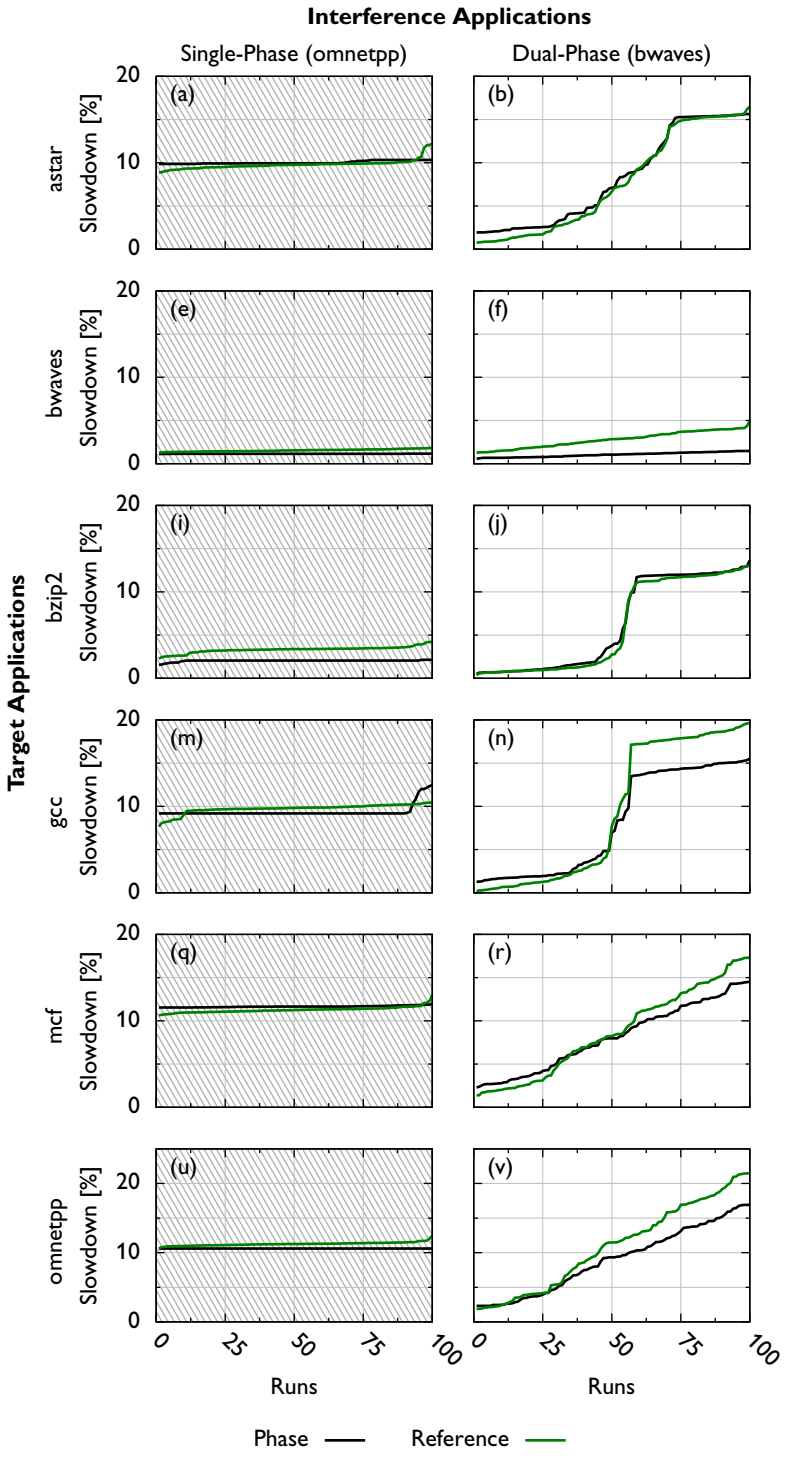
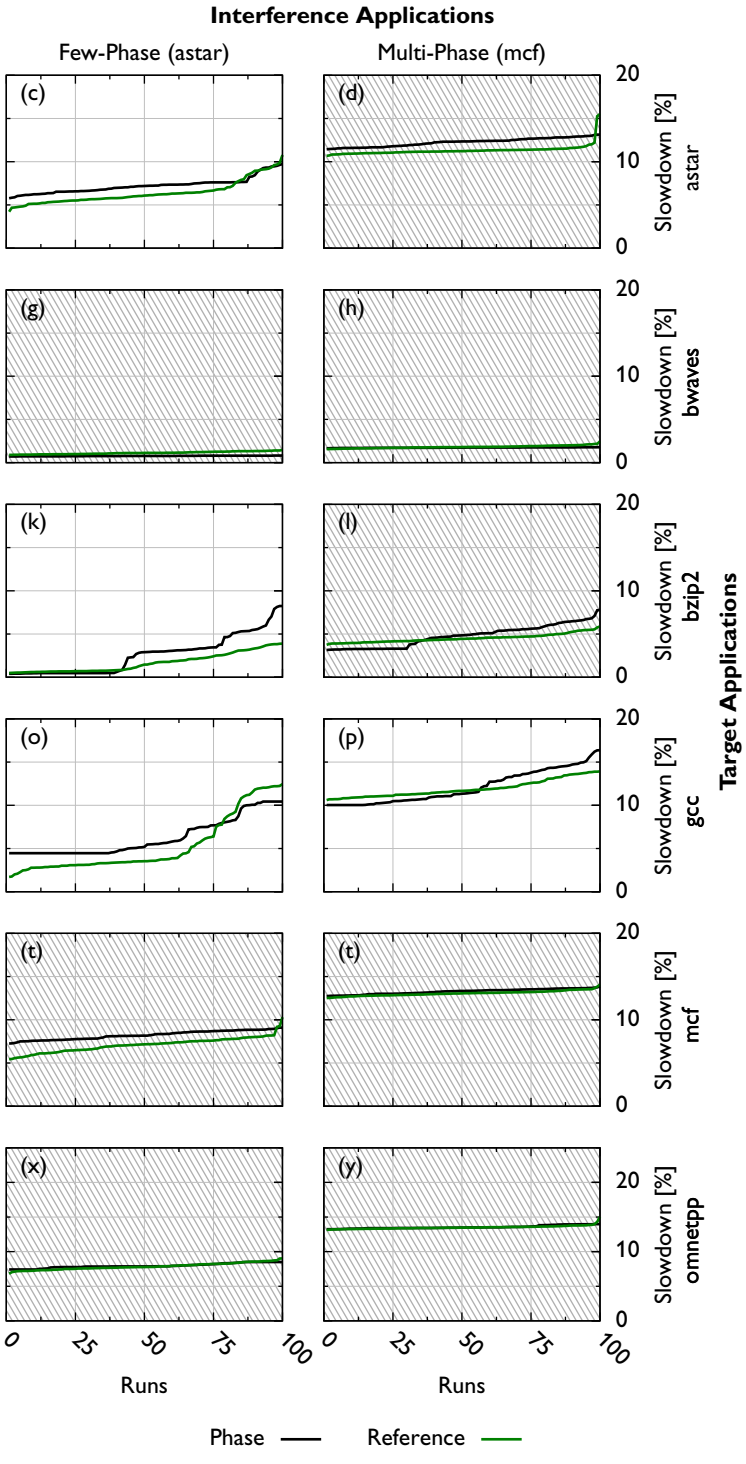


Figure 5: Cumulative distributions of target-slowdowns for 100 runs of each pair of applications with random start time offsets. The 100 application runs were sorted by slowdown, with the largest slowdown on the right. A flat line indicates no performance



variation across the 100 runs. *In general, the performance will vary depending on how the phases overlap.*

applications will therefore depend on the starting offset. Since the two phases have roughly the same length, we expect the target's behavior to depend on how it is aligned with the phase change. For example, short targets (e.g., bzip2 in Figure 5j and gcc in Figure 5n), have a sharp turn in the CDF because their execution is not likely to overlap with the phase change. Longer targets (e.g., mcf in Figure 5r), have smoother distribution since they are more likely to overlap with the phase change, causing the part of the application running before the phase change to have a small slowdown, while the parts after the phase-change have a larger slowdown. Since the position of the phase change relative to the target application will change, the CDFs will tend to become smooth.

Few-Phase. There are both flat and curved CDFs for astar as interference application. This is due to differences in the execution lengths (see Figure 3). The CDF in Figure 5g (bwaves) is flat because astar is much shorter than bwaves. Whenever the interference application terminates, it is restarted. This means that astar will be restarted over and over until bwaves terminates. The phase behavior will therefore appear homogeneous from a distance, and it results in a flat CDF. However, shorter targets (e.g., gcc in Figure 5o) will overlap with different phases in astar. We therefore see different target performance between runs and we find a curved CDF.

Multi-Phase. The CDFs for mcf are similar in shape to astar's for mostly the same reasons. However, mcf has a slightly different phase behavior. The same set of phases reappear several times in mcf (see Figure 3(d)). Since astar takes about half the time to execute, its execution will overlap with several of mcf's phases. Changing offsets in starting time will therfore not change astars performance, since astar will just co-execute with the same set of phases but with different instances of the same phases. We therefore see a flatter curve for astar co-running with mcf (Figure 5d) than with astar (Figure 5c).

Error: Performance Variability

The CDFs produced with the phase-based method have an overall good accuracy, but do not always overlap completely with the reference curves. There are two main sources of error: cache pirating data and bandwidth limitations. We will discuss these two problems in the following sections.

Pirate Data

To measure cache-size dependent data, cache pirating co-executes a cache intensive stress application that tries to steal parts of the cache.

This approach has two limitations: First, if the target is also cache intensive, the pirate will have trouble keeping its working set in the cache. Second, when stealing a large portion of the cache, the pirate will have trouble reusing all of its the data before it is evicted.

Figure 5k shows the CDF for bzip2 when co-executing with astar. The problem here is that bzip2 is cache intensive and only uses a small part of the L3 cache compared to the others (see Figure 3(e)). This makes it hard for Cache Pirating to steal the required cache space. As a consequence, we incorrectly estimated some cache-size dependent effects, which leads to our overestimating the slowdown in the CDF.

One solution would be to instead use more cumbersome and expensive methods to acquire the data. For example, page coloring [10] could be used to limit the amount of cache the target application is allocated.

Bandwidth

The cache sharing model assumes that the system has infinite bandwidth. This is obviously not the case, and as a result the model will underestimate the slowdown whenever the targets need more bandwidth than the system can provide. Figure 5 shows that we tend to underestimate the slowdown of bwaves. The second phase in bwaves (see Figure 3(b)) consumes more bandwidth than the other applications. If this is a problem, we should expect that we will find the largest errors when modeling bwaves, which is indeed the case.

One feature of the cache sharing model is that it can predict the bandwidth an application mix requires to avoid being bandwidth limited. Figure 6 shows the estimated cumulative bandwidth demand[4] and the measured cumulative bandwidth the application mix received during the fastest (i.e., run 1 in Figure 5) and the slowest (i.e., run 100 in Figure 5) runs. We interpret the figures as follows: x percent of the execution has a bandwidth demand of more than y GB/s. For example, during the slowest run with mcf (Figure 6r), 50% of mcf's execution needs more than 7.5 GB/s to avoid slowing down due to bandwidth limitations.

The bandwidth demand is lowest for the fastest run since the target applications is co-running with the first phase in bwaves. Here, the estimated bandwidth demand and the measured bandwidth usage closely match each other. This means that the system can provide the required bandwidth. But also, since we accurately estimate the slowdown, this also implies that the method can accurately estimate the bandwidth demand.

[4]The model produces bandwidth estimates using the input profile to estimate the application's bandwidth consumption for a given cache allocation.

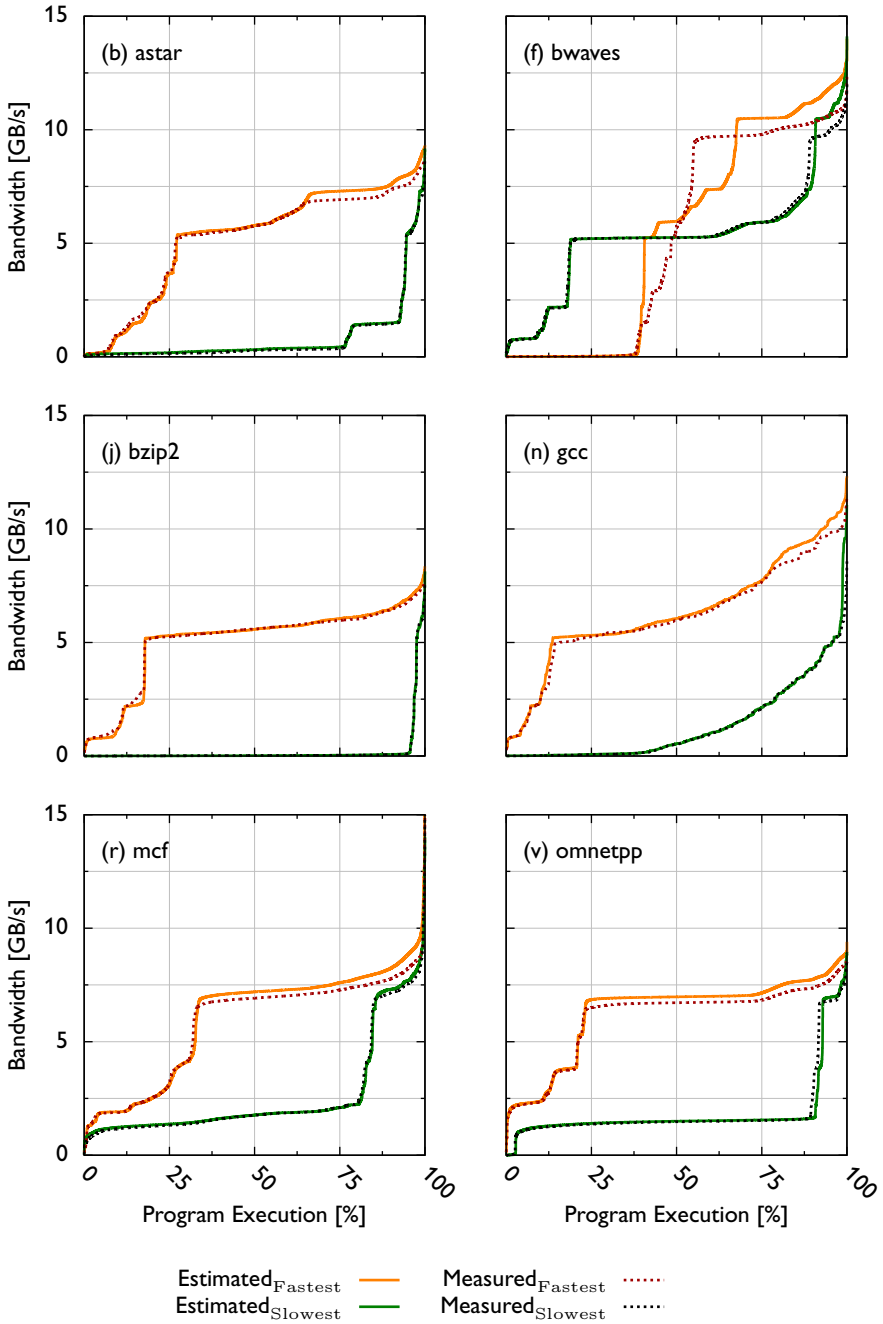


Figure 6: Predicted cumulative bandwidth demand (Estimated) and measured cumulative bandwidth usage (Measured) for the fastest and the slowest run when co-executing with bwaves.

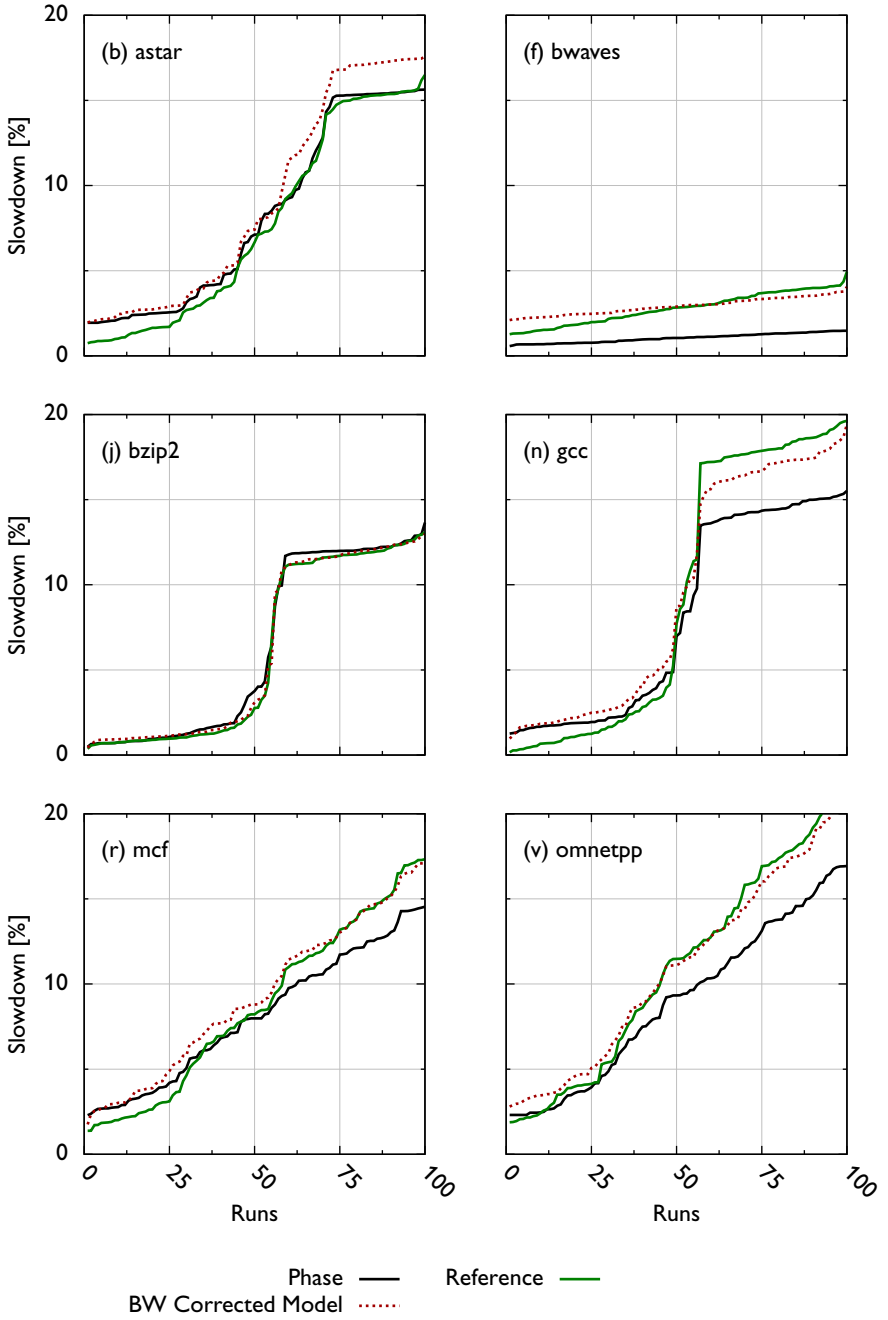


Figure 7: Cumulative slowdown distributions for 100 runs (as in Figure 5) with the bandwidth corrected model. *This shows that the accuracy can be improved by combining a bandwidth model with our cache sharing model to handle both cache sharing and bandwidth.*

The slowest runs occur when the targets are co-running with the second phase in bwaves. Here the bandwidth demand is much higher, and sometimes the estimated bandwidth demand is higher than the measured bandwidth received. This means that the target is slowing down due to bandwidth limitations. To see if we can correct the slowdown estimations by taking this into consideration, we use the measured bandwidth the application mix receives from the reference hardware runs. To do this, we update the estimated number of executed cycles (c_{est}) with the following formula:

$$c_{new\_est} = c_{est} + \frac{(BW_{est} - BW_m) * c_{est}}{BW_{MAX}}$$

where c_{new_est} is the new estimate, c_{est} is the old estimate, BW_{est} is the estimated bandwidth demand, BW_m is the measured bandwidth received and finally, BW_{MAX} is the maximum bandwidth our system can provide[5]. In other words, we extend the modeled execution time by the number of additional cycles incurred by bandwidth limitations.

Figure 7 shows the result of estimating the slowdown with the bandwidth corrected model. This correction reduces the slowdown error for bwaves, mcf, and omnetpp. However, we still underestimate the slowdown slightly for gcc, and now overestimate the slowdown for astar.

Unfortunately, such a bandwidth correction will not work in practice since it uses oracle information (i.e., BW_m), but it illustrates that a better slowdown estimate can be obtained by combining the cache sharing model with a bandwidth model to model both cache sharing and bandwidth limitations. This is a promising direction for future work.

5 Case Study – Modeling Multi-Cores

In the previous section we investigated performance variations of application pairs. However, modern processors have more than two cores. In this section, we perform a small case study to demonstrate that our method can be used to model larger application mixes, and to model system throughput.

Since all of the techniques we integrate in this method scale beyond two cores, we demonstrate that our method can scale as well by estimating the system throughput when co-running a mix of four applications on our four core reference system. To do this we compare the estimated behavior (IPC and bandwidth) to that of the actual behavior for a mix

[5] We estimated the real-world bandwidth limit of our reference system to approximately 12 GB/s using the STREAM benchmark [13].

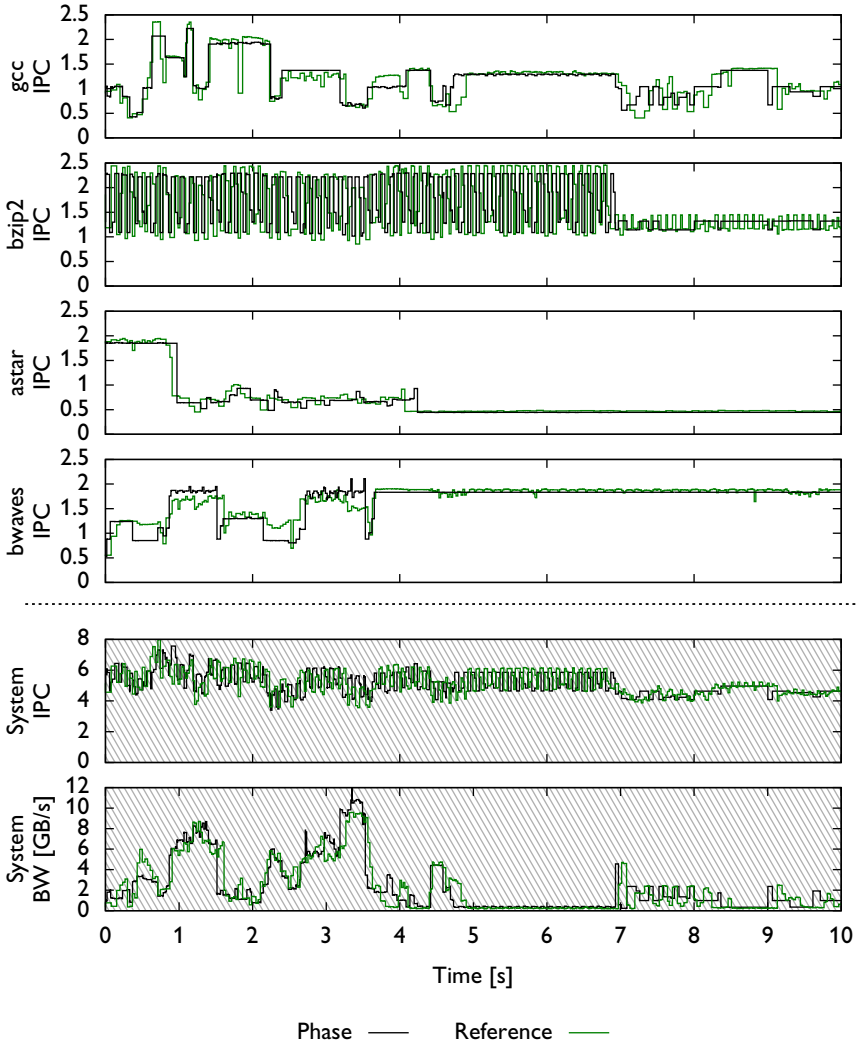


Figure 8: Predicted IPC (Phase) and measured IPC (Reference) for four co-running applications over time on a four-core system, as well as predicted and reference aggregate throughput (IPC) and bandwidth.

of four applications. Figure 8 shows the IPC and system throughput over time for the first ten seconds when co-running gcc, bzip2, astar and bwaves. The figure shows that the estimated IPCs matches the reference well.

The two main sources of error, pirate data and bandwidth, will become more problematic when modeling larger application mixes. The amount of cache available to each application is reduced when adding

more programs to the mix, which puts more pressure on the cache pirate to collect data for smaller cache allocations.

The bandwidth limitation will also become more noticeable for two reasons: First, more applications will contend for bandwidth, and thus lower the amount available to each application. Second, when an application receives less cache space, its bandwidth usage increases since it misses more in L3 and that data needs to be fetch from memory again.

6 Related Work

Techniques to explore and understand multicore performance can generally be divided into three different categories; full system simulation, partial simulation/modeling, and higher level modeling. The most expensive but also the most detail approach is full system simulation [1, 24, 25] where all cores and the entire memory system are simulated. A faster, but less detailed, approach is to only simulate/model parts of the system, and in particular the memory system. Such methods are either trace driven [2–4, 27] or use high-level data [16, 29] similar to the data we use. Finally, the least detailed approach simply aims to identify which applications are sensitive to resource contention [11, 17, 28].

Simulation normally requires combinations of applications to be simulated together, which leads to poor scaling. Van Craeynest and Eeckhout [26] combine simulation and memory system modeling to reduce the cost of simulating co-scheduled applications. Instead of simulating how applications contend for shared resources, they simulate applications running in isolation and use the output from the simulator to drive a cache sharing model. A major difference between our methods is that they depend on a single high-fidelity simulation to generate the application profiles used by their model, whereas we measure our input data with a relatively low overhead on the target system. Also, accurately simulating commodity hardware is often hard, or even impossible, since manufacturers seldom release enough information to implement a cycle-accurate simulator. Additionally, their evaluation focuses on the performance variations of the underlying hardware due to different application mixes, whereas we focus on the performance variations of the individual applications.

The method most similar to ours is the phase guided simulation methods by Van Biesbrouck et al. [24, 25]. Similar to our phase-based method, they use phase information to reuse simulation results. However, since their method relies on simulation they need to find and simulate representative regions (i.e., sample windows) of co-running phases. We do not have this problem since we can use the average behavior for the entire phase in our profiles.

7 Conclusions

In this paper, we have presented an analytical method that predicts performance variability due to the cache sharing effects imposed by other co-running applications. The per-application profile data the method requires can be captured cheaply and accurately during native execution on real hardware for each application in isolation. Three alternative cache-sharing methods with different performance properties were compared. We showed that the fastest method provides excellent accuracy. We have analyzed the performance variations caused by bandwidth sharing and showed that even a simple bandwidth sharing model could explain most of the deviations observed when the bandwidth contention is high. In future work, we plan on extending our analytical method to include such bandwidth-sharing effects.

Due to its speed, simple input data, and accuracy, this method can be used to build efficient tools for software developers or system designers, and is fast enough to be leveraged in scheduling and operating system designs.

References

[1] Nathan L. Binkert, Ron G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. "The M5 Simulator: Modeling Networked Systems". In: *IEEE Micro* 26.4 (2006), pp. 52–60. DOI: 10.1109/MM.2006.82.

[2] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2005. DOI: 10.1109/HPCA.2005.27.

[3] Xi E. Chen and Tor M. Aamodt. "A First-Order Fine-Grained Multithreaded Throughput Model". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2009. DOI: 10.1109/HPCA.2009.4798270.

[4] Xi E. Chen and Tor M. Aamodt. "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors". In: *IEEE Transactions on Computers* PP.99 (2011). DOI: 10.1109/TC.2011.141.

[5] Ashutosh S. Dhodapkar and James E. Smith. "Comparing Program Phase Detection Techniques". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA: IEEE Computer Society, 2003. DOI: 10.1109/MICRO.2003.1253197.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. "Pattern Classification". In: 2nd ed. Wiley-Interscience, 2001. Chap. 10.11. Online Clustering, pp. 559–565.

[7] 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)*. 2011, pp. 165–175. DOI: 10.1109/ICPP.2011.15.

[8] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Jr Simon Steely, and Joel Emer. "Adaptive Insertion Policies for Managing Shared Caches". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 208–219. DOI: 10.1145/1454115.1454145.

[9] J. Lau, S. Schoemackers, and B. Calder. "Structures for Phase Classification". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004, pp. 57–67. DOI: 10.1109/ISPASS.2004.1291356.

[10] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 2008, pp. 367–378. DOI: 10.1109/HPCA.2008.4658653.

[11] Jason Mars, Lingjia Tang, and Mary Lou Soffa. "Directly Characterizing Cross Core Interference Through Contention Synthesis". In: *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 2011. DOI: 10.1145/1944862.1944887.

[12] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. "Contention Aware Execution". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. 2010. DOI: 10.1145/1772954.1772991.

[13] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, 1991–2007. URL: http://www.cs.virginia.edu/stream/.

[14] Nitzan Peleg and Bilha Mendelson. "Detecting Change in Program Behavior for Adaptive Optimization". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2007, pp. 150–162. DOI: 10.1109/PACT.2007.25.

[15] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proc. Annual International Symposium on Microarchitecture (MICRO)*. 2006, pp. 423–432. DOI: 10.1109/MICRO.2006.49.

[16] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. "Efficient Techniques for Predicting Cache Sharing and Throughput". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 305–314. DOI: 10.1145/2370816.2370861.

[17] Andreas Sandberg, David Eklöv, 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)*. 2010. DOI: 10.1109/SC.2010.44.

[18] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. "Phase Behavior in Serial and Parallel Applications". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2012, pp. 47–58. DOI: 10.1109/IISWC.2012.6402900.

[19] Andreas Sembrant, David Eklov, and Erik Hagersten. "Efficient Software-based Online Phase Classification". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011, pp. 104–115. DOI: 10.1109/IISWC.2011.6114207.

[20] T. Sherwood, S. Sair, and B. Calder. "Phase Tracking and Prediction". In: *Proc. International Symposium on Computer Architecture (ISCA)*. 2003, pp. 336–347. DOI: 10.1109/ISCA.2003.1207012.

[21] Timothy Sherwood, Erez Perelman, and Brad Calder. "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications". In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2001, pp. 3–14. DOI: 10.1109/PACT.2001.953283.

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

[23] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. "Managing Shared L2 Caches on Multicore Systems in Software". In: *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. 2007.

[24] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. "Considering All Starting Points for Simultaneous Multithreading Simulation". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2006, pp. 143–153. DOI: 10.1109/ISPASS.2006.1620799.

[25] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2004, pp. 45–56. DOI: 10.1109/ISPASS.2004.1291355.

[26] Kenzo Van Craeynest and Lieven Eeckhout. "The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation". In: *Proc. International Symposium on Workload Characterization (IISWC)*. 2011, pp. 26–37. DOI: 10.1109/IISWC.2011.6114194.

[27] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. "All-Window Profiling and Composable Models of Cache Sharing". In: *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2011. DOI: 10.1145/1941553.1941567.

[28] Yuejian Xie and Gabriel H Loh. "Dynamic Classification of Program Memory Behaviors in CMPs". In: *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*. 2008.

[29] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. "Cache Contention and Application Performance Prediction for Multi-Core Systems". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452065.

[30] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010. DOI: 10.1145/1736020.1736036.

PAPER IV

Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed

Andreas Sandberg
Erik Hagersten
David Black-Schaffer

Technical report 2014-005. Dept. of Information Technology, Uppsala University, March 2014.
URI: urn:nbn:se:uu:diva-220649

Abstract — Popular microarchitecture simulators are typically several orders of magnitude slower than the systems they simulate. This leads to two problems: First, due to the slow simulation rate, simulation studies are usually limited to the first few billion instructions, which corresponds to less than 10% the execution time of many standard benchmarks. Since such studies only cover a small fraction of the applications, they run the risk of reporting unrepresentative application behavior unless sampling strategies are employed. Second, the high overhead of traditional simulators make them unsuitable for hardware/software co-design studies where rapid turn-around is required.

In spite of previous efforts to parallelize simulators, most commonly used full-system simulations remain single threaded. In this paper, we explore a simple and effective way to parallelize sampling full-system simulators. In order to simulate at high speed, we need to be able to efficiently fast-forward between sample points. We demonstrate how hardware virtualization can be used to implement highly efficient fast-forwarding in the standard gem5 simulator and how this enables efficient execution between sample points. This extremely rapid fast-forwarding enables us to reach new sample points much quicker than a single sample can be simulated. Together with efficient copying of simulator state, this enables parallel execution of sample simulation. These techniques allow us to implement a highly scalable sampling simulator that exploits sample-level parallelism.

We demonstrate how virtualization can be used to fast-forward simulators at 90% of native execution speed on average. Using virtualized fast-forwarding, 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. We demonstrate that our parallelization strategy scales almost linearly and simulates one core at up to 93% of its native execution rate, 19 000x faster than detailed simulation, while using 8 cores.

1 Introduction

Simulation is commonly used to evaluate new proposals in computer architecture and to understand complex hardware/software interactions. However, traditional simulation is very slow. While the performance of computer systems have steadily increased, simulators have become increasingly complex, and their performance relative to the simulated systems has decreased. A typical full-system simulator for a single out-of-order (OoO) processor executes around 0.1 million instructions per

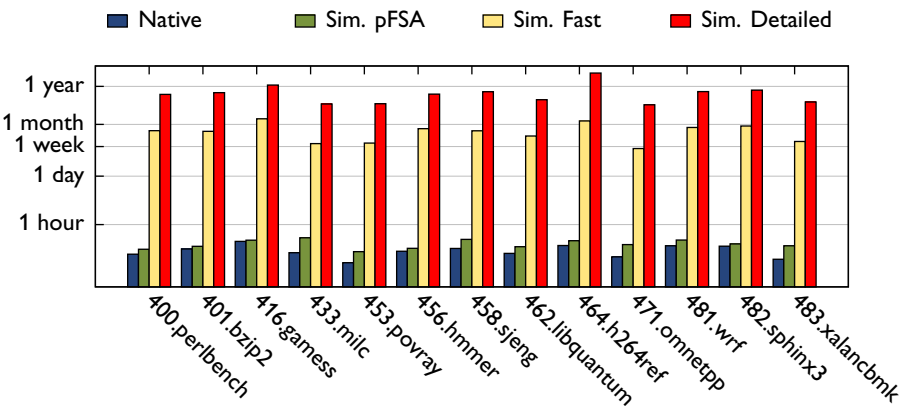


Figure 1: Native, our parallel sampler (pFSA), and projected execution times using gem5's functional and detailed out-of-order CPUs for a selection of SPEC CPU2006 benchmarks.

second (MIPS) on a modern processor that peaks at several billion instructions per second per core. Even fast, simplified, simulation modes typically execute at only 1–10 MIPS. The slow simulation rate is a severe limitation when evaluating new high-performance computer architectures or researching hardware/software interactions. There is therefore a need for efficient sampling simulators that are able to fast-forward simulation at near-native speed.

Many common benchmarks take an exorbitant amount of time to simulate in detail to completion. This is illustrated by Figure 1 which compares execution times of native execution, our parallel sampling method (pFSA), and project simulation times using the popular gem5 [3] full-system simulator. The low simulation speed has several undesirable consequences: 1) In order to simulate interesting parts of a benchmark, researchers often fast-forward to a point of interest (POI). In this case, fast forwarding to a new a simulation point close to the end of a benchmark takes between a week and a month, which makes this approach painful or even impractical. 2) Since fast-forwarding is relatively slow and a sampling simulator can never execute faster than the fastest simulation mode, it is often impractical to get good full-application performance estimates using sampling techniques. 3) Interactive use is slow and painful. For example, setting up and debugging a new experiment would be much easier if the simulator could execute at more human-usable speeds.

Many researchers have worked on improving simulator performance. One popular strategy has been to sample execution. The SMARTS [22] methodology uses periodic sampling, wherein the simulator runs in a fast

mode most of the time and switches to a detailed simulation mode to measure performance. A similar idea, SimPoint [17], uses stored checkpoints of multiple samples that represent the dominant parts of an application.

In this work, we propose a simple but effective parallel sampling methodology that uses *hardware virtualization* to *fast-forward* between samples at near-native speed and parallelization to overlap detailed simulation and fast-forwarding. We demonstrate an implementation of our sampling methodology for the popular gem5 full-system simulation environment. However, the methodology itself is general and can be applied to other simulation environments. In our experiments, we show that our implementation scales almost linearly to close to native speed of execution resulting in a peak performance in excess of 4 GIPS.

To accomplish this, we extend gem5 with a new CPU module that uses the hardware virtualization support available in current ARM- and x86-based hardware to execute directly on the physical host CPU. Our *virtual CPU module* uses standard Linux interfaces, such as the Linux Kernel-based Virtual Machine [7] (KVM) that exposes hardware virtualization to user space. This virtual CPU module is similar to that of PTLsim [23], but differs on two crucial points, both of which stem from PTLsim's use of the Xen para-virtualization environment. Since PTLsim depends on Xen, it presents a para-virtualized environment to the simulated system. This means that the simulated system needs to be aware of the Xen environment to function correctly and it does not simulate many important low-level hardware components, such as interrupt timers or IO devices. In addition, the use of Xen makes it difficult to use PTLsim in a shared environments (e.g., a shared cluster), which is not the case for our KVM-based implementation. Since KVM is provided as a standard component in Linux, we have successfully used our CPU module on shared clusters without modifying the host's operating system.

Having support for virtualization in gem5 enables us to implement extremely efficient *Virtual Fast-Forwarding* (VFF), which executes instructions at close to native speed. By itself, VFF overcomes some of the limitations of traditional simulation environments. Using VFF, we can quickly execute to a POI anywhere in a large application and then switch to a different CPU module for detailed simulation, or take a checkpoint for later use. Due to the its speed, it is feasible to work interactively with the simulator while debugging and setting up the simulation environment.

VFF enables us to rapidly fast-forward the simulator at near-native speed. We use this capability to implement a highly efficient sampling simulator. We demonstrate how this simulator can be parallelized using

standard operating system techniques, which enables us to overlap sample simulation with fast-forwarding. We call this parallel simulator pFSA for Parallel Full Speed Ahead. Similar parallelization techniques [12, 19] have previously been applied to the Pin [8] dynamic instrumentation engine in order to hide instrumentation costs. However, unlike Pin-based approaches, we are not limited to user-space profiling.

Our contributions are:

- We present a new *virtual CPU module* in gem5 which uses standard Linux interfaces and executes code at near-native speed. We are actively working on contributing this to the rest of the gem5 community.
- We demonstrate how hardware virtualization can be used to implement *fast and accurate simulation sampling* (2.0% IPC error on average).
- We present a simple strategy to parallelize sampling simulators which results in almost linear speedup to close to native speed (63% of native execution, 2.0 GIPS on average) for a selection of SPEC CPU2006 benchmarks.
- We present a method that estimates the accuracy loss due to cache warming inaccuracies, and demonstrate how it can be integrated into the sampling framework with low overhead (adding 3.9% overhead on average).

2 Overview of FSA Sampling

Until now, detailed simulation has been painfully slow. To make simulators usable for larger applications, many researchers [4, 17, 20–22] have proposed methods to sample the simulation. With sampling, the simulator can run in a faster, less detailed mode between samples, and only spend time on slower detailed simulation for the individual samples. Design parameters such as sampling frequency, cache warming strategy, and fast forwarding method give the user the ability to control the trade-off between performance and accuracy to meet his or her needs.

SMARTS [22] is a well-known sampling methodology which uses three different modes of execution to balance accuracy and simulation overhead. The first mode, *functional warming*, is the fastest mode and executes instructions without simulating timing, but still simulates caches and branch predictors to maintain long-lasting microarchitectural state. This mode moves the simulator from one sample point to another and

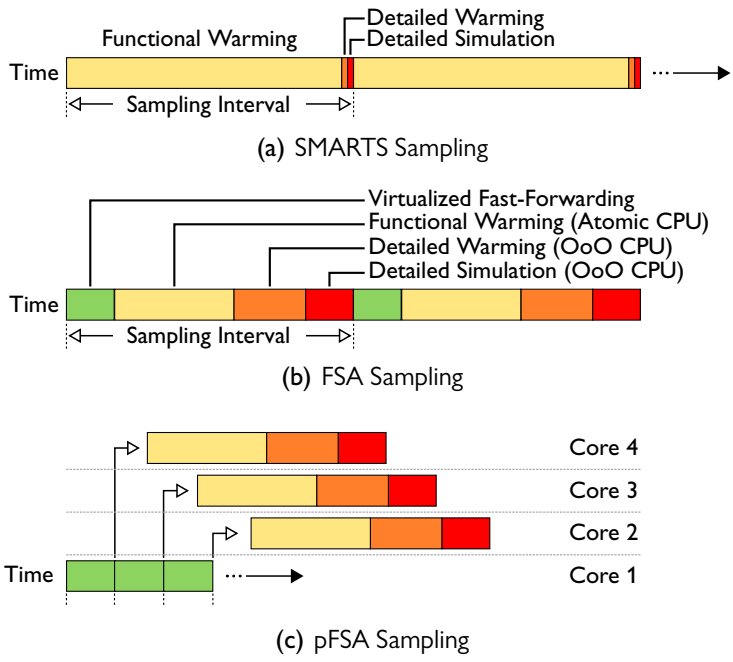


Figure 2: Comparison of how different sampling strategies interleave different simulation modes.

executes the bulk of the instructions. The second mode, *detailed warming*, simulates the entire system in detail using an OoO CPU model and warms the CPU's internal structures (e.g., load and store buffers). The third mode, *detailed sampling*, simulates the system in detail and takes the desired measurements. The interleaving of these sampling modes is shown in Figure 2(a).

SMARTS uses a technique known as *always-on* cache and branch predictor warming, which guarantees that these resources are warm when a sample is taken. This makes it trivial to ensure that the long-lived microarchitectural state (e.g., caches and branch predictors) is warm. However, the overhead of always-on cache warming, which effectively prevents efficient native execution, is significant. We trade-off the guarantees provided by always-on cache and branch predictor warming for dramatic performance improvements (in the order of 1000x) and demonstrate a technique that can be used to detect and estimate warming errors.

In traditional SMARTS-like sampling, the vast majority of the simulation time is spent in the functional warming mode [20, 22] as it executes the vast majority of the instructions. To reduce the overhead of this mode, we use VFF to execute instructions at near-native speed on

the host CPU when the simulator is executing between samples. However, we cannot directly replace the functional warming mode with native execution using VFF, as VFF can not warm the simulated caches and branch predictors. Instead, we add it as a new execution mode, *virtualized fast-forward*, which uses VFF to execute between samples. After executing to the next sample at near-native speed in the virtualized fast-forward mode, we switch to the functional warming mode, which now only needs to run long enough to warm caches and branch predictors. This allows us to execute the vast majority of our instructions at near native speed through hardware virtualization (Figure 2(b)). We call this sampling approach Full Speed Ahead (FSA) sampling. With this approach, the majority of the instructions (typically more than 95%) now execute in the (enormously faster) virtualized fast-forward mode instead of the simulated functional warming mode.

Despite executing the majority of the *instructions* natively, FSA still spends the majority of its *time* in the non-virtualized simulation modes (typically 75%–95%) to warm and measure sample points. To parallelize this sample simulation we need to do two things: copy the simulator state for each sample point (to allow them to execute independently), and advance the simulator to the next simulation point before the previous ones have finished simulating (to generate parallel work). We implement such a parallel simulator by continuously running the simulator in the virtualized fast-forward mode, and cloning the simulator state when we want to take a sample. We then do a detailed simulation of the cloned sample in parallel with the continued fast-forwarding of the original execution. If we can copy the system state with a sufficiently low overhead, the simulator will scale well, and can reach near-native speeds. We call this simulation mode Parallel Full Speed Ahead (pFSA) sampling. pFSA has the same execution modes as FSA, but unlike FSA the functional and detailed modes execute in parallel with the virtualized fast-forward mode (Figure 2(c)).

Since FSA and pFSA use limited warming of caches and branch predictors, there is a risk of insufficient warming which can lead to incorrect simulation results. To detect and estimate the impact of limited warming, we devise a simulation strategy that allows us to run the detailed simulation for both the optimistic (sufficient warming) and pessimistic (insufficient warming) cases. We use our efficient state copying mechanism to quickly re-run detailed warming and simulation without re-running functional warming. This results in a very small overhead since the simulator typically spends less than 10% of its execution time in the detailed modes. The difference between the pessimistic and optimistic cases gives us insight into the impact of functional warming.

3 Background

gem5: Full-System Discrete Event Simulation

Full-system simulators are important tools in computer architecture research as they allow architects to model the performance impact of new features on the whole computer system including the operating system. To accurately simulate the behavior of a system, they must simulate all important components in the system, including CPUs, the memory system, and the I/O and the storage. In most simulators the components are designed as modules, enabling users to plug in new components relatively easily.

Simulators based on discrete event simulation handle time by maintaining a queue of events that happen at specific times. Each event is associated with an event handler that is executed when the event is triggered. For example, an event handler might simulate one clock cycle in a CPU. New events are normally only scheduled (inserted into the queue) by event handlers. The main loop in a discrete event simulator takes the first event from the queue and executes its event handler. It continues to do so until it encounters an exit event or the queue is empty. As a consequence of executing discrete events from a queue, the time in the simulated system progresses in discrete steps of varying length, depending on the time between events in the queue.

gem5 [3] is a discrete event full-system simulator, which provides modules for most components in a modern system. The standard gem5 distribution includes several CPU modules, notably a detailed superscalar *OoO CPU module* and a simplified faster *functional CPU module* that can be used to increase simulation speed at a loss of detail. The simulated CPU modules support common instruction sets such as ARM, SPARC, and x86. Due to the design of the simulator, all of the instruction sets use the same pipeline models. In addition, gem5 includes memory system modules (GEMS [10] or simplified MOESI), as well a DRAM module, and support for common peripherals such as disk controllers, network interfaces, and frame buffers.

In this paper, we extend gem5 to add support for hardware virtualization through a new *virtual CPU module* and leverage the speed of this new module to add support for parallel sampling. The virtual CPU module can be used as a drop-in replacement for other CPU modules in gem5, thereby enabling rapid execution. Since the module supports the same gem5 interfaces as simulated gem5 CPU modules, it can be used for checkpointing and CPU module switching during simulation.

Hardware Virtualization

Virtualization solutions have traditionally been employed to run multiple operating system instances on the same hardware. A layer of software, a *virtual machine monitor* or *VMM*, is used to protect the different operating systems from each other and provide a virtual view of the system. The VMM protects the virtual machines from each other by intercepting instructions that are potentially dangerous, such as IO or privileged instructions. Dangerous instructions are then simulated to give the software running in the virtual machine the illusion of running in isolation on a real machine[1]. Early x86 virtualization solutions (e.g., VMware) used binary rewriting of privileged code to intercept dangerous instructions and complex logic to handle the mapping between addresses in the guest and host system. As virtualization gained popularity, manufacturers started adding hardware virtualization extensions to their processors. These extensions allow the VMM to intercept dangerous instructions without binary rewriting and provide support for multiple layers of address translation (directly translating from guest virtual addresses to host physical addresses). Since these extensions allow most of the code in a virtual machine to execute natively, many workloads execute at native speed.

The goals of virtualization software and traditional computer architecture simulators are very different. One of the major differences is how device models (e.g, disk controllers) are implemented. Traditional virtualization solutions typically prioritize performance, while architecture simulators focus on accurate timing and detailed hardware statistics. Timing sensitive components in virtual machines typically follow the real-time clock in the host, which means that they follow wall-clock time rather than a simulated time base. Integrating support for hardware virtualization into a simulator such as gem5 requires us to ensure that the virtual machine and the simulator have a consistent view of devices, time, memory, and CPU state. We describe these implementation details in Section 4.

4 Implementation

In order to implement a fast sampling simulator, we need to support extremely fast fast-forwarding as most instructions will be executed in the fast-forward mode. We implement rapid fast-forwarding using hardware

[1]This is not strictly true, the virtualization software usually exposes virtual devices that provide more efficient interfaces than simulated hardware devices.

virtualization which executes code natively. To further improve the performance of the simulator, we overlap fast-forwarding and sample simulation by executing them in parallel. This requires efficient cloning of the simulator's internal state, which we implement using copy-on-write techniques. While our implementation is gem5-specific, we believe that the techniques used are portable to other simulation environment.

Hardware Virtualization in gem5

Our goal is to accelerate simulation by off-loading some instructions executed in the simulated system to the hardware CPU. This is accomplished by our virtual CPU module using hardware virtualization extensions to execute code natively at near-native speed. We designed the virtual CPU module to allow it to work as a drop-in replacement for the other CPU modules in gem5 (e.g., the OoO CPU module) and to only require standard features in Linux. This means that it supports gem5 features like CPU module switching during simulation and runs on off-the-shelf Linux distributions.

Integrating hardware virtualization in a discrete event simulator requires that we ensure consistent handling of 1) simulated devices, 2) time, 3) memory, and 4) processor state. First, simulators and traditional virtualization environments both need to provide a device model to make software believe it is running on a real system. We interface the virtual CPU with gem5's device models (e.g., disk controllers, displays, etc.), which allows the virtual CPU to use the same devices as the simulated CPUs. Second, discrete event simulators and traditional virtualization environments handle time in fundamentally different ways. For example, a virtualization environment uses real, wall-clock, time to schedule timer interrupts, whereas a discrete event simulator uses a simulated time base. Interfacing the two requires careful management of the time spent executing in the virtual CPU. Third, full-system simulators and virtualization have different memory system requirements. Most simulators assume that processor modules access memory through a simulated memory system, while the virtual CPU requires direct access to memory. In order to execute correctly, we need to make sure that simulated CPUs and virtual CPUs have a consistent view of memory. Fourth, the state of a simulated CPU is not directly compatible with the real hardware, which makes it hard to transfer state between a virtual CPU and a simulated CPU. These issues are discussed in detail below:

Consistent Devices: The virtualization layer does not provide any device models. A CPU normally communicates with devices through memory mapped IO and devices request service from the CPU through

interrupts. Memory accesses to IO devices (and IO instructions such as `in` and `out`) are intercepted by the virtualization layer, which stops the virtual CPU and hands over control to gem5. In gem5, we synthesize a memory access that is inserted into the simulated memory system, allowing the access to be seen and handled by gem5's device models. IO instructions are treated like normal memory-mapped device accesses, but are mapped to a special address range in gem5's simulated memory system. When the CPU model sees an interrupt from a device, it injects it into the virtual CPU using KVM's interrupt interface.

Consistent Time: Simulating time is difficult because device models (e.g., timers) execute in simulated time, while the virtual CPU executes in real time. A traditional virtualization environment solves this issue by running device models in real time as well. For example, if a timer is configured to raise an interrupt every second, it would setup a timer on the host system that fires every second and injects an interrupt into the virtual CPU. In a simulator, the timer model inserts an event in the event queue one second into the future. The simulator then executes instructions, cycle by cycle, until it reaches the timer event. At this point, the timer model raises an interrupt in the CPU. To make device models work reliably, we need to bridge this gap between simulated time and the time as perceived by the virtual CPU.

We address the difference in timing requirements between the virtual CPU and the gem5 device models by restricting the amount of time the virtual CPU is allowed to execute between simulator events. When the virtual CPU is started, it is allowed to execute until a simulated device requires service (e.g., raises an interrupt or starts a delayed IO transfer). This is accomplished by looking into the event queue before handing over control to the virtual CPU. If there are events scheduled, we use the time until the next event to determine how long the virtual CPU should execute before handling the event. Knowing this, we schedule a timer that interrupts the virtual CPU at the correct time to return control to the simulator, which handles the event.

Due to the different execution rates between the simulated CPU and the host CPU (e.g., a server simulating an embedded system), we need to scale the host time to make asynchronous events, such as interrupts, happen with the right frequency relative to the executed instructions. For example, when simulating a CPU that is slower than the host CPU, we scale time with a factor that is less than one (i.e., we make device time faster relative to the CPU). This makes the host CPU seem slower as timer interrupts happen more frequently relative to the instruction stream. Our current implementation uses a constant conversion factor, but future implementations could determine this value automatically using sampled timing-data from the OoO CPU module.

Consistent Memory: Interfacing between the simulated memory system and the virtualization layer is necessary to transfer state between the virtual CPU module and the simulated CPU modules. First, the virtual machine needs to know where physical memory is located in the simulated system and where it is allocated in the simulator. Since gem5 stores the simulated system's memory as contiguous blocks of physical memory, we can look at the simulator's internal mappings and install the same mappings in the virtual system. This gives the virtual machine and the simulated CPUs the same view of memory. Second, since virtual CPUs do not use the simulated memory system, we need to make sure that simulated caches are disabled when switching to the virtual CPU module. This means that we need to write back and invalidate all simulated caches when switching to the virtual CPU. Third, accesses to memory-mapped IO devices need to be simulated. Since IO accesses are trapped by the virtualization layer, we can translate them into simulated accesses that are inserted into the simulated system to access gem5's simulated devices.

Consistent State: Converting between the processor state representation used by the simulator and the virtualization layer, requires detailed understanding of the simulator internals. There are several reasons why a simulator might be storing processor state in a different way than the actual hardware. For example, in gem5, the x86 flag register is split across several internal registers to allow more efficient dependency tracking in the OoO pipeline model. Another example are the registers in the x87 FPU: the real x87 stores 80-bit floating point values in its registers, while the simulated x87 only stores 64-bit values. Similar difficulties exist in the other direction. For example, only one of the two interfaces used to synchronize FPU state with the kernel updates the SIMD control register correctly. We have implemented state conversion to give gem5 access to the processor state using the same APIs as the simulated CPU modules. This enables online switching between virtual and simulated CPU modules as well as simulator checkpointing and restarting.

Since our virtual CPU module integrates seamlessly with the rest of gem5, we can use it transfer state to and from other simulated CPU modules. This allows the virtual CPU module to be used as a plug-in replacement for the existing CPU modules whenever simulation accuracy can be traded off for execution speed. For example, it can be used to implement efficient performance sampling by fast-forwarding to points of interest far into an application, or interactive debugging during the setup phase of an experiment.

Cloning Simulation State in gem5

Exposing the parallelism available in a sampling simulator requires us to be able to overlap the detailed simulation of multiple samples. When taking a new sample, the simulator needs to be able to start a new worker task (process or thread) that executes the detailed simulation using a copy of the simulator state at the time the sample was taken. Copying the state to the worker can be challenging since the state of the system (registers and RAM) can be large. There are methods to limit the amount of state the worker [20] needs to copy, but these can complicate the handling of miss-speculation. We chose to leverage the host operating system's copy-on-write (CoW) functionality to provide each sample with its own copy of the full system state.

In order to use the CoW functionality in the operating system, we create a copy of the simulator using the fork system call in UNIX whenever we need to simulate a new sample. The semantics of fork gives the new process (the child) a lazy copy (via CoW) of most of the parent process's resources. However, when forking the FSA simulator, we need to solve two problems: shared file handles between the parent and child and the inability of the child to use the same KVM virtual machine that the parent is using for fast-forwarding. The first issue simply requires that the child reopens any file it intends to use. To address the child's inability to use the parent's KVM virtual machine, we need to immediately switch the child to a non-virtualized CPU module upon forking. Since the virtual CPU module used for fast-forwarding can be in an inconsistent state (e.g., when handling IO or delivering interrupts), we need to prepare for the switch in the parent before calling fork (this is known as draining in gem5). By preparing to exit from the virtualized CPU module before forking, we allow the child process to switch to a simulated CPU without having to execute in the virtualized (KVM) CPU module.

One potential problem when using fork to copy the simulation state is that the parent and child will use the same system disk images. Writes from one of the processes could easily affect the other. To avoid this we configure gem5 to use copy-on-write semantics and store the disk writes in RAM.

Our first implementation of the parallel simulator suffered from disappointing scalability. The primary reason for this poor scaling was due to a large number of page faults, and a correspondingly large amount of time spent in the host kernel's page fault handler. These page faults occur as a result of the operating system copying data on writes to uphold the CoW semantics which ensure that the child's copy of the simulated system state is not changed by the fast-forwarding parent. An interesting observation is that most of the cost of copying a page is in the overhead of

simply taking the page fault; the actual copying of data is comparatively cheap. If the executing code exhibits decent spatial locality, we would therefore expect to dramatically reduce the number of page faults and their overhead by increasing the page size. In practice, we experienced much better performance with huge pages enabled.

Warming Error Estimation

We estimate the errors caused by limited warming by re-running detailed warming and simulation without re-running functional warming. We implement this by cloning the warm simulator state (forking) before entering the detailed warming mode. The new child then simulates the pessimistic case (insufficient warming), meanwhile the parent waits for the child to complete. Once the child completes, the parent continues to execute and simulates the optimistic case (sufficient warming).

We currently only support error estimation for caches (we plan to extend this functionality to TLBs and branch predictors), where the optimistic and pessimistic cases differ in the way we treat *warming misses*, i.e. misses that occur in sets that have not been fully warmed. In the optimistic case, we assume all warming misses are actual misses (i.e., sufficient warming). This may underestimate the performance of the simulated cache as some of the misses might have been hits had the cache been fully warmed. In the pessimistic case, we assume that warming misses are hits (i.e., worst-case for insufficient warming). This overestimates the performance of the simulated cache since some of the hits might have been capacity misses.

5 Evaluation

To evaluate our simulator we investigate three key characteristics: functional correctness, accuracy of sampled simulation, and performance. To demonstrate that the virtual CPU module integrates correctly with gem5, we perform two experiments that separately verify integration with gem5's devices and state transfer. These experiments show that we transfer state correctly, but also uncovers several functional bugs in gem5's simulated CPUs. To evaluate the accuracy of our proposed sampling scheme, we compare the results of a traditional, non-sampling, reference simulation of the first 30 billion instructions of the benchmarks to sampling using a gem5-based SMARTS implementation and pFSA. We show that pFSA can estimate the IPC of the simulated applications with an average error of 2.0%. To investigate sources of the error, we

Pipeline	gem5's default OoO CPU	
	Store Queue	64 entries
	Load Queue	64 entries
Branch Predictors	Tournament Predictor	2-bit choice counters, 8 k entries
	Local Predictor	2-bit counters, 2 k entries
	Global Predictor	2-bit counters, 8 k entries
	Branch Target Buffer	4 k entries
Caches	L1I	64 kB, 2-way LRU
	L1D	64 kB, 2-way LRU
	L2	2 MB, 8-way LRU, stride prefetcher

Table 1: Summary of simulation parameters.

investigate the impact of cache warming on accuracy. Finally, we evaluate scalability in a separate experiment where we show that our parallel sampling method scales almost linearly up to 28 cores.

For our experiments we simulated a 64-bit x86 system (Debian Wheezy with Linux 3.2.44) with split 2-way 64 kB L1 instruction and data caches and a unified 8-way 2MB or 8MB L2 cache with a stride prefetcher. The simulated CPU uses gem5's OoO CPU model. See Table 1 for a summary of the important simulation parameters. We compiled all benchmarks with GCC 4.6 in 64-bit mode with x87 code generation disabled[2]. We evaluated the system using the SPEC CPU2006 benchmark suite with the reference data set and the SPEC runtime harness. All simulation runs were started from the same checkpoint of a booted system. Simulation execution rates are shown running on a 2.3 GHz Intel Xeon E5520.

SMARTS, FSA, and pFSA all use a common set of parameters controlling how much time is spent in their different execution modes. In all sampling techniques, we executed 30 000 instructions in the detailed warming mode and 20 000 instructions in the detailed sampling mode. The length of detailed warming was chosen according to the method in the original SMARTS work [22] and ensures that the OoO pipeline is warm. Functional warming for FSA and pFSA was determined heuristically to require 5 million and 25 million instructions for the 2 MB L2 cache and 8 MB L2 cache configurations, respectively. Methods to automatically select appropriate functional warming have been proposed [9, 18] by other authors and we outline a method leveraging our warming error estimates in the future work section. Using these parameters, we

[2]We disabled x87 code generation in the compiler, forcing it to generate SSE code instead, since the simulated gem5 CPUs only support a limited number of x87 instructions.

took 1000 samples per benchmark. Due to the slow reference simulations, we limit accuracy studies to the first 30 billion instructions from each of the benchmarks, which corresponds to roughly a week's worth of simulation time in the OoO reference. For these cases, the sample period was adjusted to ensure 1000 samples in the first 30 billion instructions.

Validating Functional Correctness

For a simulation study to be meaningful, we need to be confident that instructions executed in the simulator produce the right results, i.e., they are functionally correct. Incorrect execution can result in anything from subtle behavior changes to applications crashing in the simulated system. To assess correctness of our gem5 extensions, we rely on SPEC's built-in verification harness, which compares the output of a benchmark to a reference[3]. In order to use this harness, we execute all benchmarks to completion with their reference data sets. We verify that our gem5 additions to support hardware virtualization work correctly by running the benchmarks solely on the virtual CPU module (devices are still simulated by gem5). This experiment ensures that basic features, such as the interaction between the memory and simulated device models work correctly, and that the experimental setup (compilers, OS, and SPEC) is correct. We then verify that our reference simulations are correct by completing and verifying them using VFF.

In the first experiment, we verify that the virtual CPU module works correctly, including its interactions with the virtualized hardware and the simulated system in gem5. To do this, we execute and verify all benchmarks using only the virtual CPU module. In this experiment, all benchmarks executed to completion and completed their verification runs successfully. This demonstrates that: a) our virtual CPU module interacts correctly with the memory system and device models in gem5, and, b) our simulated system and the benchmark setup are working correctly.

In the second experiment, we evaluate the correctness of our reference simulations. In this experiment, we simulated all benchmarks for 30 billion instructions using the detailed CPU module and ran them to completion using VFF. This experiment showed that 9 out of the 29 benchmarks failed before finishing the simulation of the first 30 billion instructions and that another 7 failed to verify after running to completion. In order to verify that the benchmarks that failed after executing the first 30 billion instructions were not caused by incorrect state transfer between the simulated CPU and the virtual CPU module, we set

[3]We realize that this is not sufficient to guarantee functional correctness, but we use SPEC's verification suite here since it is readily available.

Benchmark			Verifies in Reference	Verifies using VFF	Verifies when Switching
400.perlbench	401.bzip2	416.gamess	Yes	Yes	Yes
433.milc	453.povray	456.hmmer			
458.sjeng	462.libquantum	464.h264ref			
471.omnetpp	481.wrf	482.sphinx3			
483.xalancbmk					
410.bwaves	434.zeusmp	435.gromacs	No	Yes	Yes
436.cactusADM	444.namd	459.GemsFDTD			
470.lbm					
445.gobmk	450.soplex	454.calculix	Fatal Error[1]	Yes	Yes
429.mcf	473.astar		Fatal Error[2]	Yes	Yes
437.leslie3d			Fatal Error[3]	Yes	Yes
403.gcc			Fatal Error[4]	Yes	Yes
447.dealII			Fatal Error[5]	Yes	No
465.tonto			Fatal Error[6]	Yes	Yes
Summary:			13/29 verified, 9/29 fatal	29/29 verified	28/29 verified

1. Simulator gets stuck.
2. Triggers a memory leak causing the simulator crash.
3. Terminates prematurely for unknown reason.
4. Fails with internal error. Likely due to unimplemented instructions.
5. Benchmark segfaults due to unimplemented instructions.
6. Terminated by internal benchmark sanity check.

Table 2: Summary of verification results for all benchmarks in SPEC CPU2006. This table is based on three experiments: a reference OoO simulation that is completed using the virtual CPU module, purely running on the virtual CPU module, and repeatedly switching between a simulated OoO CPU and the virtual CPU module.

up another experiment where we switched each benchmark 300 times between the simulated CPU and the virtual CPU module. In this experiment, all benchmarks, with the exception of 447.dealII (which failed because of unimplemented instructions), ran to completion and verified. The results of these experiments are summarized in Table 2. Their results indicate that our virtual CPU module works and transfers state correctly. Unfortunately, they also indicate that the x86 model in gem5 still has some functional correctness issues (in our experience, both the Alpha and ARM models are much more reliable).

Since benchmarks that do not verify take different program paths in the simulator and on real hardware, we exclude them from the rest of the evaluation.

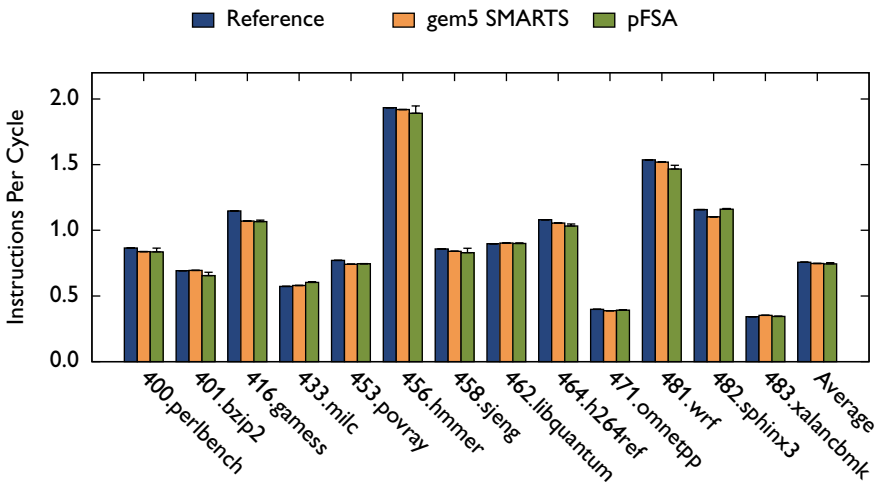
Accuracy

A simulator needs to be accurate in order to be useful. The amount of accuracy needed depends on which question the user is asking. In many cases, especially when sampling, accuracy can be traded off for performance. In this section, we evaluate the accuracy of our proposed parallel sampling methodology. The sampling parameters we use have been selected to strike a balance between accuracy and performance when estimating the average CPI of an application.

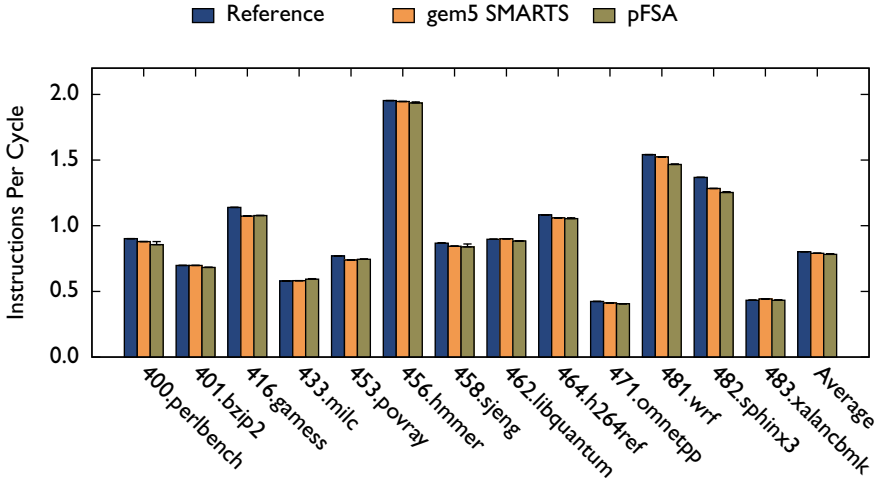
All sampling methodologies that employ functional warming suffer from two main sources of errors: sampling errors and inaccurate warming. Our SMARTS and pFSA experiments have been setup to sample at the same instructions counts, which implies that they should suffer from the same sampling error[4]. Functional warming incurs small variations in the access streams seen by branch predictors and caches since it does not include effects of speculation or reordering. This has can lead to a small error, which has been shown [22] to be in the region of 2%. The error incurred by these factors is the baseline SMARTS error, which in our experiments is 1.87% for a 2 MB L2 cache and 1.18% for an 8 MB L2 cache.

Another source of error is the limited functional warming of branch predictors and caches in FSA and pFSA. In general, our method provides very similar results to our gem5-based SMARTS implementation. However, there are a few cases (e.g., 456.hmmer) when simulating a 2 MB cache where we did not apply enough warming. In these case the IPC predicted by SMARTS is within, or close to, the warming error estimated by our method (Figure 3(a)). A large estimated warming error generally

[4]There might be slight differences when the virtual CPU module is used due to small timing differences when delivering asynchronous events (e.g., interrupts).



(a) 2 MB L2



(b) 8 MB L2

Figure 3: IPC for the first 30 billion instructions of each benchmark as predicted by a reference simulation compared to a our gem5-based SMARTS implementation and pFSA. The error bars extending from the pFSA bars represent warming error estimates.

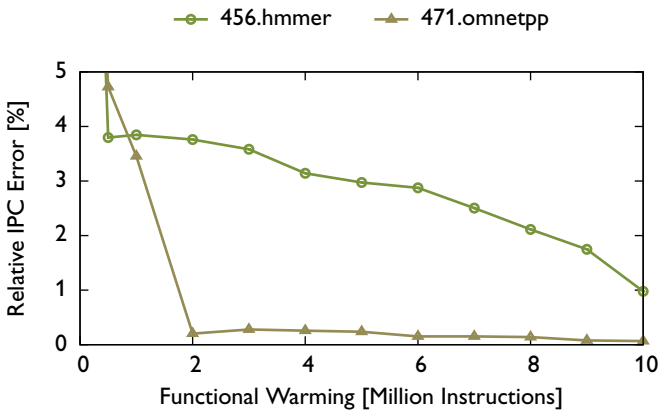


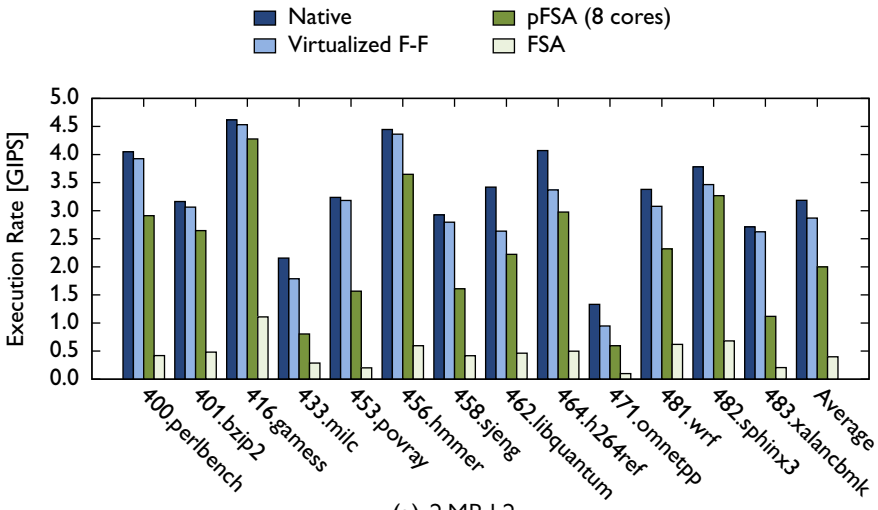
Figure 4: Estimated relative IPC error due to insufficient cache warming as a function of functional warming length for 456.hmmer and 471.omnetpp.

indicate that a benchmark should have had more functional warming applied. Note that we can not just assume the higher IPC since some warming misses are likely to be real misses.

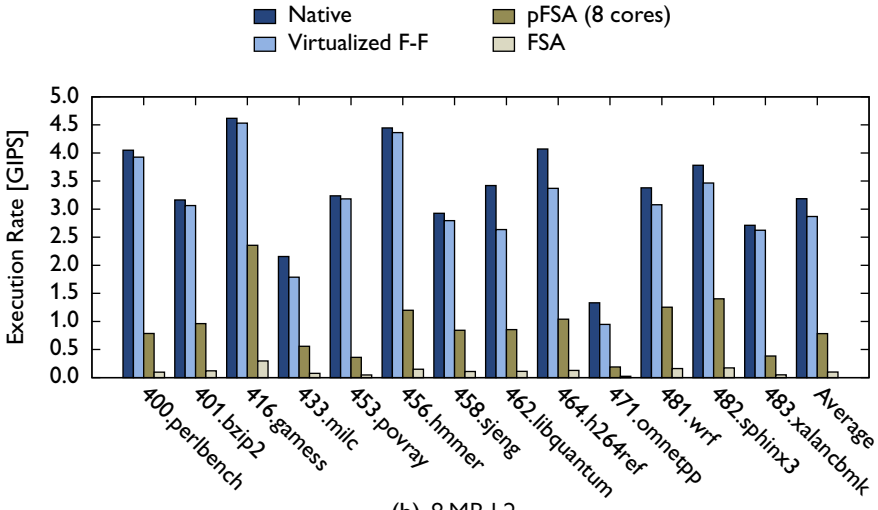
To better understand how warming affects the predicted IPC bound, we simulated two benchmarks (456.hmmer & 471.omnetpp) with different warming behaviors with different amounts of cache warming. Figure 4 shows how their estimated warming error relative to the IPC of the reference simulations shrinks as more cache warming is performed. These two applications have wildly different warming behavior. While 471.omnetpp only requires two million instructions to reach an estimated warming error less than 1%, 456.hmmer requires more than 10 million instructions to reach the same goal.

Performance & Scalability

A simulator is generally more useful the faster it is as high speed enables greater application coverage and quicker simulation turn-around times. Figure 5 compares the execution rates of native execution, VFF, FSA, and pFSA when simulating a system with a 2MB and 8MB last-level cache. The reported performance of pFSA does not include warming error estimation, which adds 3.9% overhead on average. The achieved simulation rate of pFSA depends on three factors. First, fast-forwarding using VFF runs at near-native (90% on average) speed, which means that the simulation rate of an application is limited by its native execution rate regardless of parallelization. Second, each sample incurs a constant cost. The longer a benchmark is, the lower the average overhead. Third,



(a) 2 MB L2



(b) 8 MB L2

Figure 5: Execution rates when simulating a 2MB (a) and 8MB (b) L2 cache for pFSA and FSA compared to native and fast-forwarding using the virtual CPU module.

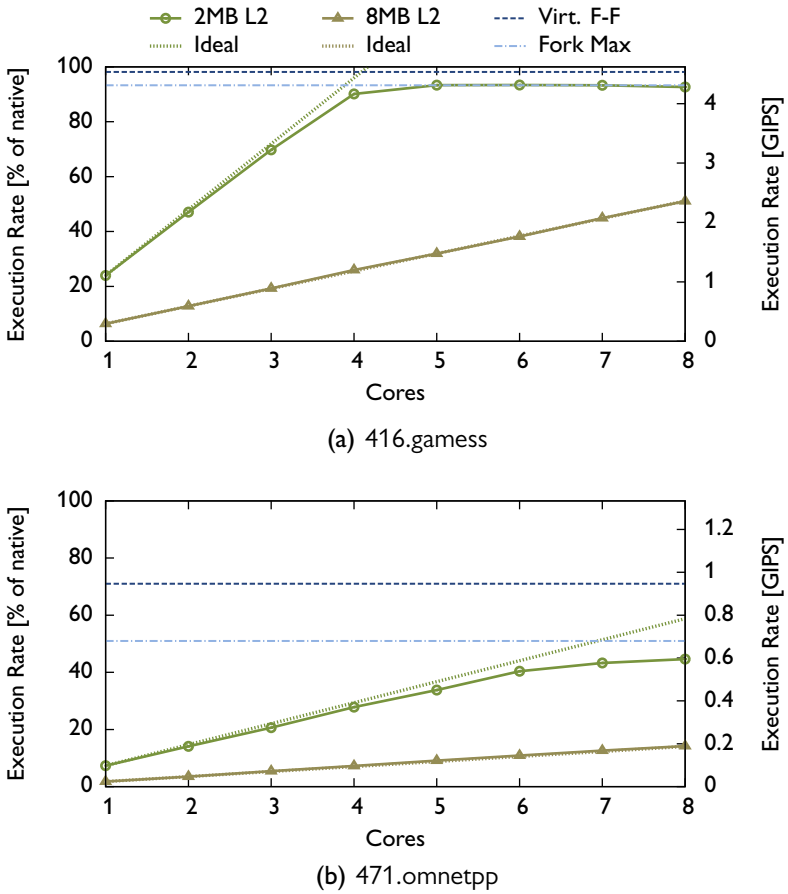
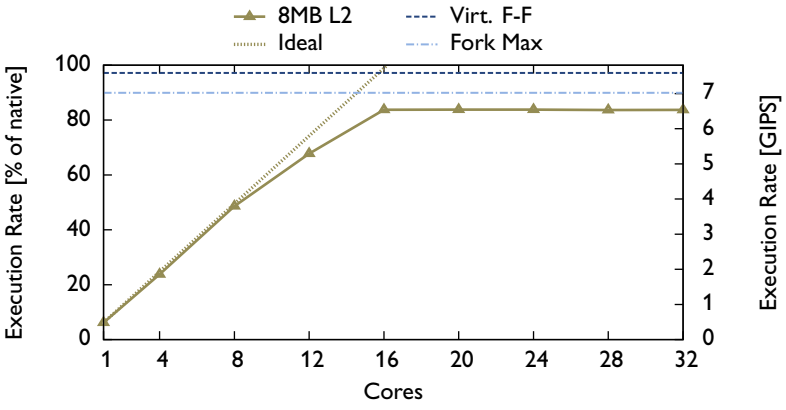


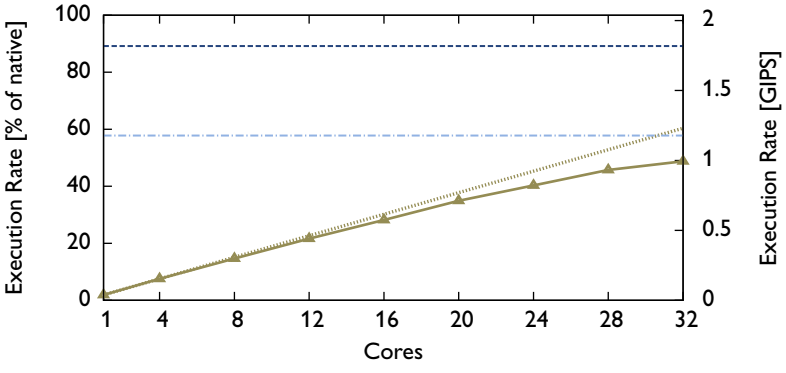
Figure 6: Scalability of 416.gamess (a) and 471.omnetpp (b) running on an 2-socket Intel Xeon E5520.

large caches need more functional warming, and the longer the functional warming, the greater the cost of the sample. As seen when comparing the average simulation rates for a 2 MB cache and an 8 MB cache, simulating a system with larger caches incurs a larger overhead.

The difference in functional warming length results in different simulation rates for 2MB and 8MB caches. While the 8MB cache simulation is slower to simulate than the smaller cache, there is also more parallelism available. Looking at the simulation rate when simulating a 2MB cache as a function of the number of threads used by the simulator (Figure 6) for a fast (416.gamess) and a slow (471.omnetpp) application, we see that both applications scale almost linearly until they reach 93% and 45% of native speed respectively. The larger cache on the other hand starts off at a lower simulation rate and scales linearly until all cores in



(a) 416.gamess



(b) 471.omnetpp

Figure 7: Scalability of 416.gamess (a) and 471.omnetpp (b) running on a 4-socket Intel Xeon E5-4650.

the host system are occupied. We estimate the overhead of copying simulation state (Fork Max) by removing the simulation work in the child and keeping the child process alive to force the parent process to do CoW while fast-forwarding. This is an estimate of the speed limit imposed by parallelization overheads.

In order to understand how pFSA scales on larger systems, we ran the scaling experiment on a 4-socket Intel Xeon E5-4650 with a total of 32 cores. We limited this study to the 8MB cache since simulating a 2MB cache reached near-native speed with only 8 cores. As seen in Figure 7, both 416.gamess and 471.omnetpp scale almost linearly until they reach their maximum simulation rate, peaking at 84% and 48.8% of native speed, respectively.

6 Related Work

Our parallel sampling methodology builds on ideas from three different simulation and modeling techniques: virtualization, sampling, and parallel profiling. We extend and combine ideas from these areas to form a fully-functional, efficient, and scalable full-system simulator using the well-known gem5 [3] simulation framework.

Virtualization

There have been several earlier attempts at using virtualization for full-system simulation. Rosenblum et al. pioneered the use of virtualization-like techniques with SimOS [14] that ran a slightly modified (relinked to a non-default memory area) version of Irix as a UNIX process and simulated privileged instructions in software. PTLsim [23] by Yourst, used para-virtualization[5] to run the target system natively. Due to the use of para-virtualization, PTLsim requires the simulated operating system to be aware of the simulator. The simulated system must therefore use a special para-virtualization interface to access page tables and certain low-level hardware. This also means that PTLsim does not simulate low-level components like timers and storage components (disks, disk controllers, etc.). A practical draw-back of PTLsim is that it practically requires a dedicated machine since the host operating system must run inside the para-virtualization environment. Both SimOS and PTLsim use a fast virtualized mode for fast-forwarding and support detailed processor performance models. The main difference between them and gem5 with our virtual CPU module is the support for running unmodified guest operating systems. Additionally, since we only depend on KVM, our system can be deployed in shared clusters with unmodified host operating systems.

An interesting new approach to virtualization was taken by Ryckbosch et al. in their VSim [15] proposal. This simulator mainly focuses on IO modeling in cloud-like environment. Their approach employs time dilation to simulate slower or faster CPUs by making interrupts happen more or less frequently relative to the instruction stream. Since the system lacks a detailed CPU model, there are no facilities for detailed simulation or auto-calibration of the time dilation factor. In many ways, the goals of VSim and pFSA are very different: VSim focuses on fast modeling of large IO-bound workloads, while pFSA focuses on sampling of detailed micro-architecture simulation.

[5] There are signs of an unreleased prototype of PTLSim that supports hardware virtualization. However, to the best of our knowledge, no public release has been made nor report published.

Sampling

Techniques for sampling simulation have been proposed many times before [1, 4, 5, 17, 20–22]. The two main techniques are SimPoint [17] and SMARTS [22]. While both are based on sampling, SimPoint uses a very different approach compared SMARTS and pFSA that builds on checkpoints of representative regions of an application. Such regions are automatically detected by finding phases of stable behavior. In order to speed up SMARTS, Wenisch et al. proposed TurboSMARTS [20], which uses compressed checkpoints that include cache state and branch predictor state. A drawback of all checkpoint-based techniques is long turn-around time if the *simulated software* changes due to the need to collect new checkpoints. This makes them particularly unsuitable for many applications, such as hardware-software co-design or operating system development. Since pFSA uses virtualization instead of checkpoints to fast-forward between samples, there is no need to perform costly simulations to regenerate checkpoints when making changes in the simulated system.

SMARTS has the nice property of providing statistical guarantees on sampling accuracy. These guarantees assure users who strictly follow the SMARTS methodology that their sampled IPC will not deviate more than, for example, 2% with 99.7% confidence. Since we do not perform always-on cache and branch predictor warming, we can not provide the same statistical guarantees, but we achieve similar accuracy in practice. To identify problems with insufficient warming, we have proposed a low-overhead approach that can estimate the warming error.

The sampling approach most similar to FSA is the one used in COTSon [1] by HP Labs. COTSon combines AMD SimNow [2] (a JIT:ing functional x86 simulator) with a set of performance models for disks, networks, and CPUs. The simulator achieves good performance by using a dynamic sampling strategy [5] that uses online phase detection to exploit phases of execution in the target. Since the functional simulator they use can not warm microarchitectural state, they employ a two-phase warming strategy similar to FSA. However, unlike FSA, they do not use hardware virtualization to fast-forward execution, instead they rely on much slower (10x overhead [1] compared to 10% using virtualization) functional simulation.

Parallel Simulation

There have been many approaches to parallelizing simulators. We use a coarse-grained high-level approach in which we exploit parallelism between samples. A similar approach was taken in SuperPin [19] and

Shadow Profiling [12], which both use Pin [8] to profile user-space applications and run multiple parts of the application in parallel. Shadow Profiling aims to generate detailed application profiles for profile guided compiler optimizations, while SuperPin is a general-purpose API for parallel profiling in the Pin instrumentation engine. Our approach to parallelization draws inspiration from these two works and uses parallelism to overlap detailed simulation of multiple samples with native execution. The biggest difference is that we apply the technique to full-system simulation instead of user-space profiling.

Another approach to parallelization is to parallelize the core of the simulator. A significant amount of research has been done on parallel discrete event simulation (PDES), each proposal with its own trade-offs [6]. *Optimistic approaches* try to run as much as possible in parallel and rollback whenever there is a conflict. Implementing such approaches can be challenging since they require old state to be saved. *Conservative approaches* typically ensure that there can never be conflicts by synchronizing at regular intervals whose length is determined by the shortest critical path between two components simulated in parallel. The latter approach was used in the Wisconsin Wind Tunnel [13]. More recent systems, for example Graphite [11], relax synchronization even further. They exploit the observation that functional correctness is not affected as long as synchronization instructions (e.g., locks) in the simulated system enforce synchronization between simulated threads. The amount of drift between threads executed in parallel can then be configured to achieve a good trade-off between accuracy and performance.

The recent ZSim [16] simulator takes another fine-grained approach to parallelize the core of the simulator. ZSim simulates applications in two phases, a *bound* and a *weave* phase, the phases are interleaved and only work on a small number of instructions at a time. The bound phase executes first and provides a lower bound on the latency for the simulated block of instructions. Simulated threads can be executed in parallel since no interactions are simulated in this phase. The simulator then executes the weave phase that uses the traces from the bound phase to simulate memory system interactions. This can also be done in parallel since the memory system is divided into domains with a small amount of communication that requires synchronization. Since ZSim is Pin-based, ZSim only supports user-space x86 code and does not simulate any devices (e.g., storage and network). The main focus of ZSim is simulating large parallel systems.

Methods such as PDES or ZSim are all orthogonal to our pFSA method since they work at a completely different level in the simulator. For examples, a simulator using PDES techniques to simulate in parallel could

be combined with pFSA to expose even more parallelism than can be exposed by PDES alone.

7 Future Work

There are several features and ideas we would like to explore in the future. Most notably, we would like add support for running multiple virtual CPUs at the same time in a shared-memory configuration when fast-forwarding. KVM already supports executing multiple CPUs sharing memory by running different CPUs in different threads. Implementing this in gem5 requires support for threading in the core simulator, which is ongoing work from other research groups. We are also looking into ways of extending warming error estimation to TLBs and branch predictors. An interesting application of warming estimation is to quickly profile applications to automatically detect per-application warming settings that meet a given warming error constraint. Additionally, an online implementation of dynamic cache warming could use feedback from previous samples to adjust the functional warming length on the fly and use our efficient state copying mechanism to roll back samples with too short functional warming.

8 Summary

In this paper, we have presented a virtualized CPU module for gem5 that on average runs at 90% of the host's execution rate. This CPU module can be used to efficiently fast-forward simulations to efficiently create checkpoints of points of interest or to implement efficient performance sampling. We have demonstrated how it can be used to implement an efficient parallel sampler, pFSA, which accurately (IPC error of 2.2% and 1.9% when simulating 2MB and 8MB L2 caches respectively) estimates application behavior with high performance (63% or 25% of native depending on cache size). Compared to detailed simulation, our parallel sampling simulator results in 7 000x–19 000x speedup.

Acknowledgments

Initial work on hardware virtualization support in gem5 was sponsored by ARM, where the authors would especially like to thank Matthew Horsnell, Ali G. Saidi, Andreas Hansson, Marc Zyngier, and Will Deacon for valuable discussions and insights. Reference simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). This work was financed by the CoDeR-MP project and the UPMARC research center.

References

[1] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. "COTSon: Infrastructure for Full System Simulation". In: *ACM SIGOPS Operating Systems Review* 43.1 (Jan. 2009), pp. 52–61. DOI: 10.1145/1496909.1496921.

[2] Robert Bedicheck. "SimNow™: Fast Platform Simulation Purely in Software". In: *Hot Chips: A Symposium on High Performance Chips*. Aug. 2005.

[3] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. "The gem5 Simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011). DOI: 10.1145/2024716.2024718.

[4] Shelley Chen. "Direct SMARTS: Accelerating Microarchitectural Simulation through Direct Execution". MA thesis. Carnegie Mellon University, 2004.

[5] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. "Combining Simulation and Virtualization through Dynamic Sampling". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Apr. 2007, pp. 72–83. DOI: 10.1109/ISPASS.2007.363738.

[6] Richard M. Fujimoto. "Parallel Discrete Event Simulation". In: *Communications of the ACM* 33.10 (Oct. 1990), pp. 30–53. DOI: 10.1145/84537.84545.

[7] Avi Kivity, Uri Lublin, and Anthony Liguori. "kvm: the Linux Virtual Machine Monitor". In: *Proc. Linux Symposium*. 2007, pp. 225–230.

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. DOI: 10.1145/1065010.1065034.

[9] Yue Luo, Lizy K. John, and Lieven Eeckhout. "Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation". In: *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2004, pp. 10–17. DOI: 10.1109/SBAC-PAD.2004.38.

[10] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset". In: *ACM SIGARCH Computer Architecture News* 33 (2005), pp. 92–99. DOI: 10.1145/1105734.1105747.

[11] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. "Graphite: A Distributed Parallel Simulator for Multicores". In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[12] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. "Shadow Profiling: Hiding Instrumentation Costs with Parallelism". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar. 2007, pp. 198–208. DOI: 10.1109/CGO.2007.35.

[13] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers". In: *ACM SIGMETRICS Performance Evaluation Review* 21.1 (June 1993), pp. 48–60. DOI: 10.1145/166962.166979.

[14] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. "Complete Computer System Simulation: The SimOS Approach". In: *Parallel & Distributed Technology: Systems & Applications* 3.4 (Jan. 1995), pp. 34–43. DOI: 10.1109/88.473612.

[15] Frederick Ryckbosch, Stijn Polfliet, and Lieven Eeckhout. "VSim: Simulating Multi-Server Setups at Near Native Hardware Speed". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8 (2012), 52:1–52:20. DOI: 10.1145/2086696.2086731.

[16] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems". In: *Proc. International Symposium on Computer Architecture (ISCA)*. July 2013, pp. 475–486. DOI: 10.1145/2485922.2485963.

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

[18] Michael Van Biesbrouck, Brad Calder, and Lieven Eeckhout. "Efficient Sampling Startup for SimPoint". In: *IEEE Micro* 26.4 (July 2006), pp. 32–42. DOI: 10.1109/MM.2006.68.

[19] Steven Wallace and Kim Hazelwood. "SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance". In: *Proc. International Symposium on Code Generation and Optimization (CGO)*. Mar. 2007, pp. 209–220. DOI: 10.1109/CGO.2007.37.

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

[21] 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.

[22] 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)*. 2003, pp. 84–95. DOI: 10.1109/ISCA.2003.1206991.

[23] Matt T. Yourst. "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator". In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. Apr. 2007, pp. 23–34. DOI: 10.1109/ISPASS.2007.363733.