Abstract

To reduce latency and increase bandwidth to memory, modern microprocessors are designed with deep memory hierarchies including several levels of caches. For such microprocessors, the service time for fetching data from off-chip memory is about two orders of magnitude longer than fetching data from the level-one cache. Consequently, the performance of applications is largely determined by how well they utilize the caches in the memory hierarchy, captured by their *miss ratio curves*. However, efficiently obtaining an application's miss ratio curve and interpreting its performance implications is hard. This task becomes even more challenging when analyzing application performance on multi-core processors where several applications/threads share caches and memory bandwidths. To accomplish this, we need powerful techniques that capture applications' cache utilization and provide intuitive performance metrics.

In this thesis we present three techniques for analyzing application performance, StatStack, StatCC and Cache Pirating. Our main focus is on providing memory hierarchy related performance metrics such as *miss ratio*, *fetch ratio* and *bandwidth demand*, but also *execution rate*. These techniques are based on profiling information, requiring both runtime data collection and post processing. For such techniques to be broadly applicable the data collection has to have minimal impact on the profiled application, allow profiling of unmodified binaries, and not depend on custom hardware and/or operating system extensions. Furthermore, the information provided has to be accurate and easy to interpret by programmers, the runtime environment and compilers.

StatStack estimates an application's miss ratio curve, StatCC estimates the miss ratio of co-running application sharing the last-level cache and Cache Pirating measures any desired performance metric available through hardware performance counters as a function of cache size. We have experimentally shown that our methods are both efficient and accurate. The runtime information required by StatStack and StatCC can be collected with an average runtime overhead of 40%. The Cache Pirating method measures the desired performance metrics with an average runtime overhead of 5%.

To Ann-Sofie

Acknowledgment

This thesis is a result of collaboration. It had not been written without the support and encouragement from many. To begin I would like to thank my supervisors Erik Hagersten and David Black-Schaffer: Erik, for all the interesting and challenging discussion, and relentless encouragement. David, for always stressing the importance of putting our work in a larger context and for the countless hours he spent improving the presentation in our papers. Many thanks go to Nikos Nikoleris who provided great insights and spent many late nights working on the Cache Pirating paper. Furthermore, I would like to thank the remaining two members of the Uppsala Architecture Research Team: Andreas Sandberg and Muneeb Khan for all interesting discussions, help with everything from GCC to LaTeX, and for creating an inspiring work environment.

I would also like to thank: Marting Stigge, Guan Nan, Pontus Ekberg, Xiaoyue Pan and Jonathan Lidén for letting me convince them to go to Rullan for lunch every day (except Mondays which, as we all know, is lunchbox day ☹). Last but not least I would like to thank the staff at Rullan for making the tastiest lunch in town.

The work presented in this thesis has been supported by in part by the Swedish Research Council (VR) individual grant, UPMARC VR-Linnaeus grant and by the SSF CoDeR-MP project.

List of Papers

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

I David Eklov and Erik Hagersten. StatStack: Efficient Modeling of LRU Caches. In Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), White Plains, NY, USA, March 2010

II David Eklov, David Black-Schaffer and Erik Hagersten. Fast Modeling of Cache Contention in Multicore Systems. In Proceedings of the the 6th International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC), Heraklion, Crete, Greece, January 2011

III David Eklov, Nikos Nikoleris, David Black-Schaffer and Erik Hagersten. Cache Pirating: The Curse of the Shared Cache. Technical Report 2011-001, Department of Information Technology, Uppsala University, Sweden, December 2010. Submitted for publication.

Comments on My Participation:

I I am the principal author and principal investigator.

II I am the principal author and principal investigator.

III I am the principal author and principal investigator. Nikos Nikoleris contributed to the discussions, and configured and ran all reference cache simulations.

Other publications not included:

- David Eklov, David Black-Schaffer and Erik Hagersten. StatCC: A Statistical Cache Contention Model. In the Proceedings of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, September 2010
- Andreas Sandberg, David Eklov and Erik Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, November 2010

- Erik Hagersten, David Eklov and David Black-Schaffer. Efficient Cache Modeling with Sparse Data. Chapter in “Processor and System-on-Chip Simulation”, editors Olivier Temam and Rainer Leupes. Springer, 2010

Contents

1 Introduction . . . 1
  1.1 Cache Capacity . . . 2
  1.2 Cache Contention . . . 4
  1.3 Performance and Bandwidth . . . 5
2 Contributions . . . 7
3 Background and Related Work . . . 9
  3.1 Cache Simulation . . . 9
  3.2 Stack Distance . . . 11
  3.3 Reuse Distance . . . 12
  3.4 StatCache . . . 13
  3.5 Cache Contention . . . 14
  3.6 Performance Simulation . . . 15
4 Summary of Papers . . . 17
  4.1 StatStack . . . 17
  4.2 StatCC . . . 19
  4.3 Cache Pirating . . . 21
Bibliography . . . 23
Paper I . . . 29
Paper II . . . 57
Paper III . . . 83

List of Abbreviations

CMP	Chip Multiprocessor
CPI	Cycles Per Instructions
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
GB/s	Giga Byte per Second
kB	Kilo Byte
L1	Level One Cache
L2	Level Two Cache
L3	Level Three Cache
LRU	Least Recently Used
MB	Mega Byte
MPA	Misses Per Access
MPC	Misses Per Cycles
MPKI	Misses Per Kilo Instructions
MRC	Miss Ratio Curve
MRU	Most Recently Used
SPEC	Standard Performance Evaluation Corporation

1. Introduction

The long latency and limited bandwidth to off-chip memory have been identified as two potential bottlenecks of present and future computer systems [2, 32]. To reduce the impact of both memory latency and bandwidth issues, high performance computer systems implement deep memory hierarchies with multiple levels of caches. Consequently, application performance is largely determined by how well they utilize the resource in the memory hierarchy. However, understanding the interaction between applications and the memory hierarchy is hard. To deliver high performance software, we therefore need powerful techniques that provide insights into how applications' performance is impacted by their interaction with the memory hierarchy.

The goal of this thesis is to provide efficient and accurate profiling-driven methods for application performance analysis. Such methods have to efficiently capture performance metrics that accurately describe the behavior of the analyzed applications. These metrics should provide insights enabling application developers to easily pinpoint potential performance and scalability bottlenecks.

Application performance analysis is typically performed in the following cycle: First, the application is compiled and profiled. Then, the profiling information is analyzed and interpreted in order to find potential bottlenecks. Once found, an attempt is made at eliminating the bottleneck, and the cycle repeats. This requires the latency for obtaining and analyzing the profiling information to be short. However, most of the existing simulators and performance models target architectural design space explorations. As such, they model the internals of the hardware at a level of detail that makes their execution time impractical for application performance analysis.

This thesis presents three methods specifically designed for application performance analysis. Our main focus is the analysis of applications' interaction with the memory hierarchy and how it impacts the applications' performance. The first method, StatStack (Paper I) is a statistical cache model that accurately and efficiently captures an application's data locality in terms of its *miss ratio* . The second method, StatCC (Paper II), is a statistical cache contention model that models the effects of cache sharing in chip multiprocessors. The third method, Cache Pirating (Paper III), is a measurement technique that measures any performance metric available through hardware performance counters, such as miss ratio and off-chip bandwidth demand, as a function of the cache capacity received by the applications.

1.1 Cache Capacity

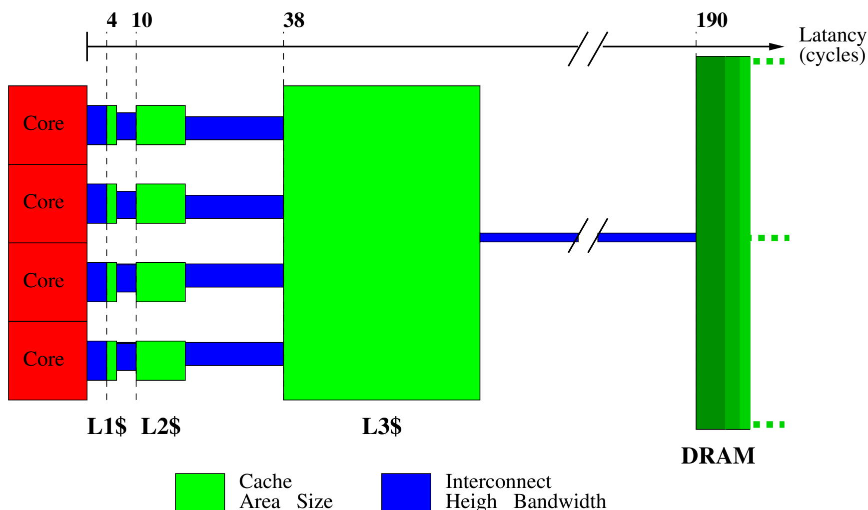


Figure 1.1: Memory hierarchy of a Nehalem based processor. Caches are represented by green boxes whose areas are proportional to their size. Their distances from the cores are proportional to their read latencies. The heights of the interconnects (blue) are proportional to their respective bandwidths. (The L1, L2 and L3 sizes are 64kB, 256kB and 8MB, with read latencies of 4, 10 and 38 cycles and bandwidths of 46, 31 and 10GB/s respectively.) The data is taken from [11].

Modern microprocessors are designed with deep memory hierarchies to reduce latency and increase bandwidth to memory. Figure 1.1 shows a typical memory hierarchy. When a core makes a request for data, the service time (latency) increases the further down the hierarchy (to the right in the figure) the requested data resides. For example, in Figure 1.1, the service time for retrieving data from off-chip memory is about two orders of magnitude longer than fetching data from the level one (L1) cache. For the SPEC CPU2006 benchmark applications, about 30% of the executed instruction are loads [27]. The performance of these applications is therefore largely determined by the memory request service time.

The longer latency incurred in the event of a *cache miss* (i.e., when the referenced data is not resident it the cache) can force the responsible core to stall and stop executing instructions. These stalls contribute to the execution time of the application, reducing its performance. To achieve high performance, it is therefore important to reduce the number of cache misses. Ideally, all the data referenced by an application should be resident in the caches. This however is not always possible as the volume of data referenced by most applications typically exceeds the capacity of the caches.

An important metric for determining how well an application leverages the caches in the memory hierarchy, is the *miss ratio*. In this thesis, we define the *miss*

ratio to be the ratio between the number of cache misses, and the total number of executed memory accesses. A low miss ratio implies that most of the referenced data is present in the cache, while a high miss ratio implies that most of the referenced data is not present in the cache.

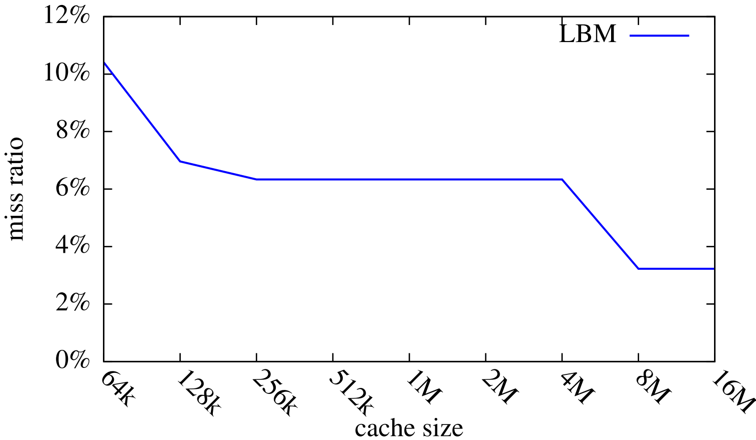


Figure 1.2: Miss ratio curve. MRC of the LBM SPEC CPU2006 benchmark application.

Miss ratio curves (MRCs) show applications' cache miss ratios as a function of the cache space available to the applications, and can be used to determine how much an application benefits/suffers from increasing/reducing its the cache capacity. The MRC is an important tool for assessing applications' data locality. Figure 1.2 shows the MRC of the LBM benchmark from the SPEC CPU2006 benchmark suit. The MRC has two distinct kinks at 256kB and 8MB. This suggests that LBM accesses two data sets, the smaller fits in a cache of 256kB and the larger fits in a cache of 8MB. Furthermore, as the MRC is flat between 256kB and 4MB, LBM does not benefit from having its available cache capacity increased from 256kB to 4MB of cache. Its miss ratio will stay at 6%. However, if its available cache capacity is increased beyond 8MB its miss ratio decreases to 3%.

MRCs have been used in several contexts such as guiding program transformation [5], cache partitioning [21], off-chip bandwidth partitioning [15] and cache contention modeling [8]. However, the main drawback of using MRC is the high cost of capturing them. There are two cost involved in capturing MRCs. 1) Capturing runtime information, such as address traces, and 2) post processing the runtime information to generate the MRC, using for example cache simulation. In this thesis, we address both of these issues by presenting StatStack (Paper I). StatStack is a statistical cache model that efficiently and accurately captures applications' MRCs.

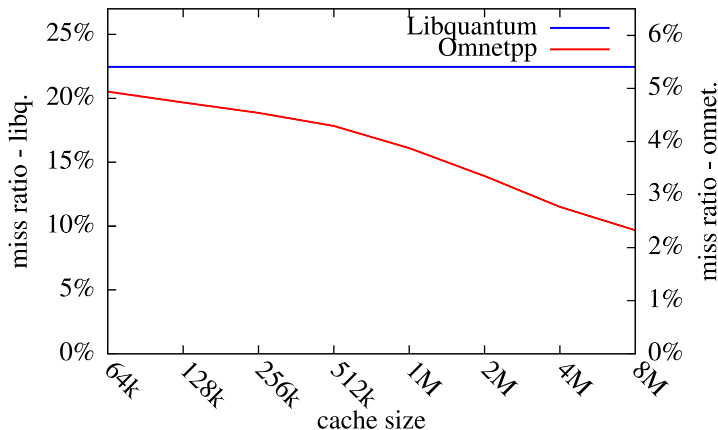


Figure 1.3: Miss ratio curves. MRCs of the Libquantum (left y-axis) and Omnetpp (right y-axis) SPEC CPU2006 benchmark applications.

1.2 Cache Contention

In modern chip multiprocessors (CMPs), cores typically share the last level cache. (For example, see Figure 1.1.) The shared cache operates as if all the data it caches belong to a single application/thread. This results in contention for cache capacity among the applications/threads running on the cores. The amount of cache capacity an application receives depends not only on its own memory access rate and pattern, but also on the applications running on the same core. For example, if an application is co-running with a set of cache-hungry applications, it will likely get a small fraction of the shared cache, while if it is co-running with applications that are not so cache-hungry it gets a larger portion of the cache capacity. Furthermore, it is not necessarily the application that benefits most from more cache capacity that gets most cache capacity allocated.

Figure 1.3 shows the MRC for both Libquantum and Omnetpp. As shown in the figure, Libquantum's MRC is flat, which implies that it does not benefit when its cache capacity is increased (from 16kB to 16MB). Libquantum is a streaming cache hungry application. No matter how much cache capacity it receives its miss ratio stays the same. Omnetpp on the other hand benefits from an increased cache capacity. However, when we co-ran these two applications on a CMP system with a shared cache, Libquantum will receive a large portion of the cache capacity. However, we cannot determine how much cache capacity the co-running applications receive simply by analyzing their MRCs. How much cache they receive does not only depend on their miss ratios but also on their access frequencies, which is not captured in the MRCs.

As mentioned above, Libquantum does not benefit from increased cache capacity butOmnetpp does, yet Libquantum will receive a large portion of the cache

capacity. This type of situation can be avoided by a clever process scheduler. However, to make such decisions the scheduler has to have information about how the co-running applications behave when they share a cache.

To understand and analyze application performance on CMPs with shared caches, we need efficient methods to estimate how much cache capacity the co-running applications receives. For such methods to be broadly applicable, they must be based on information gathered with low overhead from the applications when running independently. This thesis addresses this by presenting StatCC (Paper II). StatCC is a statistical cache contention model. It estimates the miss ratios of co-running applications based on runtime information that is efficiently captured when the applications are running individually.

1.3 Performance and Bandwidth

While the MRC is an important tool for analyzing application data locality, it lacks a fundamental notion of time, and can therefore not be directly used to assess application performance. Two applications can have the same miss ratio but still execute at difference rates. This can occur if the two applications have different instruction mixes, where, for example, one of the applications executes twice as many instructions per memory access as the other. Even if they execute the same number of non-memory instructions per memory reference, these instructions can have different latencies, resulting in the applications having different execution rates. Furthermore, MRCs do not account for latency hiding techniques such as, hardware prefetching, out-of-order execution and memory-level parallelism, and does not account for bandwidth limitations.

The information provided by MRCs is therefore not always sufficient. An important example is when estimating an application's off-chip bandwidth demand. The miss ratio tells us how many cachelines the application requests from off-chip memory (ignoring hardware prefetching), but not at what rate. Ideally, we want to estimate/measure the application's execution rate (CPI) as a function of cache capacity, accounting for all effects of the memory hierarchy, including hardware prefetching, memory-level parallelism and bandwidth limitations.

This thesis addresses these issues by presenting Cache Pirating (Paper III). Cache Pirating is a method to measure any performance metric available through hardware performance counters as a function of how much cache capacity the application receives. For example, it allows us to measure performance (CPI) and bandwidth demand (GB/s).

2. Contributions

This thesis makes the following contributions:

STATSTACK A novel statistical cache model that models caches with LRU replacement. The input to StatStack is a sparse and cheaply collected fingerprint of the target application's memory access stream, and the output is a miss ratio curve. Leveraging previous work [13], the fingerprints are collected with an average runtime overhead of 40%. Once a fingerprint is captured, StatStack produces accurate MRCs in a few seconds. To the best of our knowledge, StatStack is the fastest method to capture applications' MRCs for LRU caches.

STATCC A low-overhead statistical cache contention model. StatCC builds on the StatStack technology, and answers the following question: Given the memory access fingerprints of a set of applications, what are their miss ratios when co-running on chip multiprocessor with a shared last-level cache? For our benchmark applications, StatCC can predict the shared level cache miss ratios of the target applications with an average error of less than 2%.

CACHE PIRATING A method to accurately measure any performance metric available through hardware performance counters as a function of how much shared cache space that is available to the target application. The measurements are taken while the target application is executing on real hardware, and does therefore include all idiosyncrasies of the hardware, such as out-of-order execution, memory level parallelism and hardware prefetching. The focus of this work is on performance metrics related to the memory hierarchy, such as miss ratio (MPKI) and bandwidth demand (GB/s). With the Cache Pirating method we can accurately measure all these performance metrics with an average execution time overhead of 5%.

3. Background and Related Work

In this section we review background material and discuss related work. First, we review trace driven cache simulation (Section 3.1) and stack distance analysis (Section 3.2). An important application of these techniques is to generate MRCs, which relates to StatStack (Paper I). Second, we define the reuse distance (Section 3.3), which is the input to StatStack (Paper I) and StatCC (Paper II), and review a low-overhead method developed by Berg et al. [13] for capturing reuse distance from running applications. Then, StatCache [12, 13] is reviewed (Section 3.4), which influenced our work on StatStack. This is followed by a short discussion about two cache contention models (Section 3.5). Finally, we discuss some of the available simulation techniques for simulating the performance and bandwidth demand of applications for varying cache sizes (Section 3.6).

3.1 Cache Simulation

The straightforward method to generate MRCs is to use trace driven cache simulation. This requires an *address trace* to be collected. An address trace contains a list of the memory addresses reference by an application. A common method to collect address traces is to use binary instrumentation. The applications are instrumented with additional instructions to record the address referenced by the application's memory instructions in a file on disk. There are several tools available for binary instrumentation, both static (ATOM [1] and EEL [18]) and dynamic (Pin [7] and Valgrind [22]). The execution time overhead of collecting address traces can be up to $100\times$ depending on the instrumentation tool and the traced application. Furthermore, the resulting address trace files tend to be huge, even after applying compression. Marathe et al. [17] implement a structural compression algorithm for regular address traces. For traces exhibiting sufficient access regularity, their compression scheme dramatically surpasses both VPC3 [20] and Bzip2 [19]. Another option is to pipe the address trace directly to the cache simulator (e.g. through a POSIX pipe), avoiding the need to store the trace. This however is not always desirable, for example, when address trace is to be distributed and/or used at a later time.

To generate MRCs, the address traces are feed to a cache simulator. Cache simulators range from very simple, only simulating a single cache level with minimal features, to very complex, simulating the full memory hierarchy including multiple

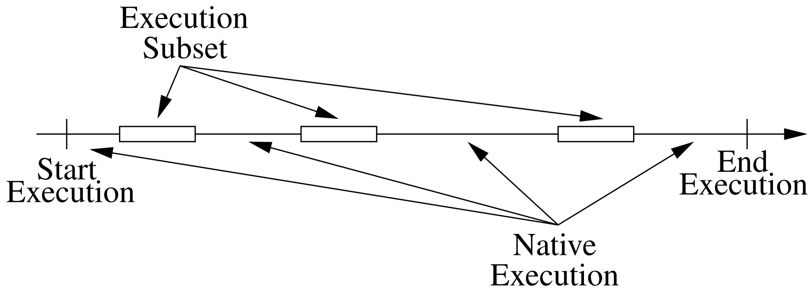


Figure 3.1: Sampled execution. The rectangular boxes on the horizontal time line represents the subsets where address traces are collected. The execution subsets are randomly scattered across the application's execution. The application executes at native speed between the traced subsets.

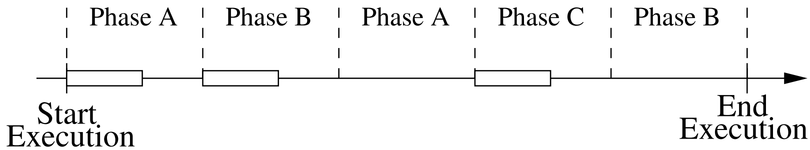


Figure 3.2: Simpoint example. The figure shows the execution of an application with three program phases, A, B and C. The rectangular boxes represent the execution subsets that are traced. As the figure shows, traces are only collected once for each phase.

levels of caches. There is a clear trade-off between accuracy and simulation time. The simple simulators are fast but do not always simulate real hardware caches to the desired level of detail. While complex simulators can be designed to have arbitrary accuracy, they suffer from long simulation times.

Laha et al. [29] proposed using statistical sampling techniques to collect address traces for only a few small subsets of the application's execution. (See Figure 3.1.) In principle, this allows the application to execute at native speed during the parts of execution when address traces are not collected. Their sampling technique reduces the execution time overhead of collecting address traces by two orders of magnitude. Furthermore, the address trace size is reduced by the same factor.

Wunderlich et al. [25] extend Laha's sampling technique to detailed architectural simulations. In the context of this thesis, their most important contribution is a rigorous statistical analysis that allows them to bound the statistical error introduced by sampling. Perelman et al. [14] proposed SimPoint, that uses program phase information to simulate only the representation portions of the application's execution. Instead of randomly selecting execution subsets to simulate, they select subsets from the most representative program phases. (See Figure 3.2.) The size of the simulated execution subsets are chosen to be proportional to the phase sizes.

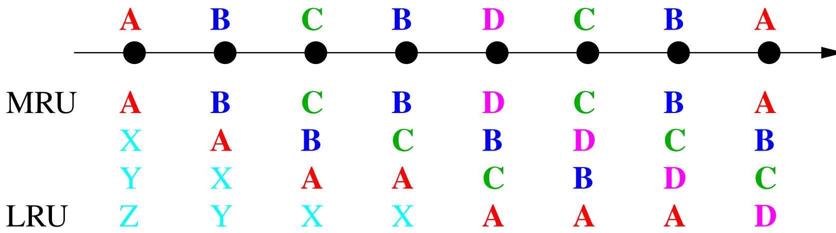


Figure 3.3: Evolution of the LRU stack of a 4-way associative LRU cache. Memory references are represented as circles on the horizontal time line, and are labeled with the accessed cachelines. The content of the LRU stack is shown below the time line, with the most recently used (MRU) cacheline at the top and the least recently used (LRU) cacheline at the bottom.

Tam et al. [10] presented a hardware performance counter based approach to collect address traces efficiently. Their focus is on online generation of miss ratio curves for the last level cache (L2). Using hardware performance counters, they trace only the references that miss in the L1 cache. To further reduce the runtime overhead they use sampling. The result is a very efficient technique for collecting L2 address traces.

Several methods have been proposed to reduce the simulation time. Most of these are based on the *stack distance* concept.

3.2 Stack Distance

For LRU caches we can conceptually think of each cache set as being organized as a finite sized stack, with the most recently used (MRU) cacheline at the top, and the least recently used (LRU) cacheline at the bottom. On a cache miss, the LRU cache-line is evicted to make room for the newly fetched cache-line which is pushed on the top of the stack. On a cache hit, the accessed cache-line is moved to the top. Figure 3.3 shows from left-to-right how the stack of one set in a 4-way associative cache evolves over time with the access pattern shown at the top. The stack maintains an age ordering of its cachelines, with the more recently used cache-lines at the top of the stack.

The stack distances[26] is the number of unique cachelines accessed between two successive memory accesses to the same cacheline. (See Figure 3.4.) The stack distance can be directly used to determine if the 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 stack distances greater than the desired cache size.

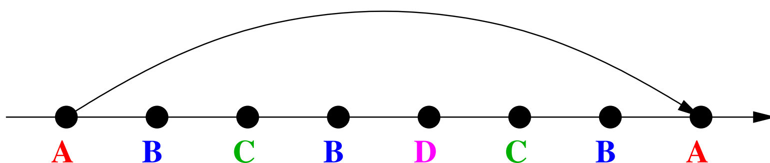


Figure 3.4: Stack distance and reuse distance. Memory references are represented as circles on the horizontal time line, and are labeled with the accessed cachelines. The arc represents the reuse of cacheline A. The second memory reference to cache line A has a *stack distance* of three as there are three unique cachelines (A,B,C) accessed between the two references to A. The *reuse distance* of the second memory reference to A is six since there are six dynamic memory references executed between the two reference to A.

The stack distances of an application's memory references are typically presented in the form of a histogram, called a *stack distance distribution*. An application's MRC can be computed from its stack distance distribution, simply by "integrating" the stack distance distribution. The hit ratio for a cache of size C is the "integral" from zero to C over the stack distance distribution, and the miss ratio is one minus the hit ratio. In fact, MRCs and stack distance distributions are equivalent in the sense that one can be computed from the other. To get the stack distance distribution we take the "derivative" of the MRC.

Since its introduction in the 70s, the stack distance concept has earned plenty of attention from researchers in the quest to find fast methods to obtain stack distances. To improve the performance of Mattson's stack based algorithm Bennett and Kruskal [3] replaced the stack with a m-ary tree. The m-ary tree allows for faster operations than the linked-list stack used by Mattson. To further improve performance, other types of trees have been proposed, for example AVL trees [16] and splay trees [24]. Kim et al. [36] propose an approximate algorithm, in which the stack is sliced up into disjoint ranges. This allows them to use a hash table to search the stack. Other approximate algorithms have also been proposed by Ding and Zong [6] and Shen et al. [35].

3.3 Reuse Distance

The reuse distance of a memory reference a, to cacheline A, is equal to the *number of memory reference* executed between a and the previous memory reference to cacheline A. For example, the reuse distance of the second memory reference to cacheline A in Figure 3.4 is 6 since there are 6 memory reference executed between the two reference to A. Note that the reuse distance and the stack distance are generally not the same. The reuse distance is always greater than or equal to the stack distance.

In the context of LRU caches, the reuse distance is a less powerful measure than the stack distance as it cannot be directly used to determine if a memory access is a hit or a miss. However, the two features that makes the reuse distance interesting are: (1) It is more general than the stack distances. It can be used to (indirectly) model caches with both random (see Section 3.4) and LRU replacement (see Section 4.1). (2) It is much more efficient to measure than the stack distance. Rather than keeping track of the number of accessed cachelines, we only need to keep track of the number of executed memory references. This allows for efficient use of functionality provided by contemporary hardware and operating systems, such as hardware performance counters and watchpoints.

Berg and Hagersten [13] implemented a reuse distance sampler, that measures the reuse distances of randomly selected memory accesses. It has a very small impact on the sampled applications runtime. They report an average slowdown of only 40 percent when hardware and operating system support are used. The reuse distance sampler works as follows. Hardware counter overflow traps are used to halt the execution when a memory reference that has been randomly selected for sampling executes. A watchpoint is then set for the cacheline it accesses. The number of memory accesses performed to date is recorded by reading a hardware counter, after which the execution is continued. The next time the same cacheline is accessed, a watchpoint trap is generated, which again halts the execution. The reuse distance is calculated as the difference between the number of memory accesses executed to date and the previously recorded number.

3.4 StatCache

StatCache [12, 13] is a statistical cache model that estimates the miss ratio for fully associative caches with random replacement policy. The input to StatCache is a sparse reuse distance sample. StatCache accurately estimates miss ratios using reuse distance samples collected with a sample rate of only 10^{-6} [13], i.e. the sample contains the reuse distance of every one millionth memory access. Such sparse reuse distances can be collected with a runtime overhead of only 40% [13]. The time it takes to run the StatCache cache analysis, i.e. the off-line simulation cost, is less than a second. StatCache avoids the problems of trace driven cache simulation: The high cost of collecting runtime information and the high cost of off-line simulation.

StatCache works as follows. For a cache with a random replacement policy the likelihood that a cacheline is evicted is exponentially decreasing with the number of replacements that has taken place since the cacheline was last accessed. The cacheline to be evicted is selected randomly. In the event of an eviction, the likelihood that a given cacheline is evicted is therefore $1/L$, where L is the number of cachelines in the cache. Suppose that we know the average miss ratio of the application, M, then we can estimate the number of evictions since the cacheline

was last accessed. If the last access to the cacheline had a reuse distance of R, we know that R memory references have been executed since the cacheline was last accessed, and the number of replacements is therefore $R \times M$ on average. (Here we assume that every cache miss results in a cacheline being evicted.) The likelihood that the cacheline has been evicted is therefore, $1 - (1 - 1/L)^{R \times M}$. Suppose the reuse distance sample contains N reuse distances, each of length $D(i)$, then we can write the following equation,

$$\sum_{i=1}^{N} \left(1 - \left(1 - \frac{1}{L} \right)^{D(i) \times M} \right) = N \times M. \tag{3.1}$$

This equation can be readily solved for the miss ratio, M using numerical methods. Solving the above equation for different cache sizes, L, we get the application's MRC.

3.5 Cache Contention

In this section we discuss the challenges of modeling contention for shared caches and briefly review two previously proposed cache contention models. The problem of analyzing cache contention has been recognized and addressed in the past [8, 34]. However, one of the major challenges, modeling the coupling between the applications' execution rate and their miss ratios, has not been addressed in previous work. For example, an increased capacity pressure from one application can result in the eviction of cachelines allocated to a second application, thereby reducing its cache allocation and possibly increasing its miss ratio and CPI. This can cause the second application's capacity pressure to increase, which, in turn, can reduce the cache allocation of the first application.

Chandra et al. [8] and Chen et al. [34] propose two cache contention models that estimate the shard-level cache miss ratio of co-running applications. However, Chandra et al. [8] rely on the user to guess the execution rate of the co-running applications. They assume that the co-running application always execute at this rate, which implies that the application's execution rates are independent of their shared cache miss ratios. Hence, they do not consider the coupling between the applications' execution rate and miss ratio. Chen et al. [34] starts with an initial guess and run their model once to compute the shared cache miss ratios. Then they re-compute the execution rates using a very simple performance model (see Section 3.6) and the new miss ratio. To arrive at the final result, they run the model once more using the refined execution rate. However, they do not consider further refinements.

3.6 Performance Simulation

A limitation of trace driven cache simulation and stack distance analysis is that they cannot be directly used to estimate execution-rate-dependent metrics, such as execution rate (CPI), miss rate (MPC) or bandwidth consumptions (BW/s). The reason is that address traces do not contain any timing information. Instead, to obtain execution rate depended metrics as a function of cache size, we need to use timing accurate architectural simulations [30, 4, 23] or performance models [31, 28, 9]. In the case of simulations, the simulation has to be run several times, once for each cache size. This however can be very costly, as the simulation time can be several orders of magnitude longer than the native execution time of the applications.

To reduce the overheads of simulation, several performance models of varying complexity and accuracy have been proposed. A simple model is shown in Eq. 3.2, where CPI_∞ is the CPI when the application is running on a fictional system with an infinite cache and $LATANCY_{DRAM}$ is the average latency to DRAM. The performance model in Eq. 3.2 only models the effects of the last level cache, but it can easily be extended to model all levels. The input to the model is the target application's miss ratio. Given the target application's MRC, we can estimate the its performance as a function of cache size by evaluating the model with the miss ratios at the desired cache sizes.

$$cpi = CPI_\infty + LATENCY_{DRAM} \times miss_ratio \tag{3.2}$$

While the above model is simple and very efficient to evaluate, its accuracy is limited for all but very simple in-order architectures. For example, it does not explicitly model latency hiding techniques, such as out-of-order execution, memory level parallelism or hardware prefetching. To address these accuracy issues, Karkhanis and Smith [31] introduced the interval simulation methodology (later extended by Eyerman et al. [28]). Interval simulation accurately models super scalar processors explicitly including several latency hiding techniques. It has been further extended, for example, to model multi-cores [9] and hardware prefetching [33]. While these models are much faster than architectural simulation (one order of magnitude [9]), they are still too slow to be practical for application performance analysis. To accurately model existing hardware, interval simulation requires several hardware parameters as input. Finding these parameters can be challenging as they are not always disclosed by the manufacturer and are hard to accurately measure. Sometimes, the only way to find these parameters is to run expensive simulations.

Doing application performance analysis with simulators/performance models requires the simulators/performance models to model the target hardware with a sufficient accuracy and detail, but at the same time be efficient enough to allow for the use of large realistic input sets. Furthermore, all their parameters have to be

known with sufficient accuracy. This limits the usefulness of both simulation and interval analysis for application performance analysis.

4. Summary of Papers

4.1 StatStack

StatStack (Paper I) is a statistical cache model that efficiently and accurately estimates MRCs for fully associative LRU caches. The input to StatStack is the same type of profiling data used by StatCache – a sparse sample of an application's reuse distances (see Section 3.4). This allows StatStack to use the same efficient reuse distance sampling technique as StatCache (see Section 3.3), and thereby inherits StatCache's low runtime overhead (40% [13]). The analysis of the profiling data is very fast. Given the input profiling data, StatStack generates MRCs in less than a second. This makes StatStack an efficient and accurate method for capturing MRCs. To the best of our knowledge, StatStack is the fastest available method to capture MRCs for LRU caches.

At a high level, StatStack works as follows. First, it profiles the target application and collects a spare reuse distance sample. Second, it estimates the target application's reuse distance distribution from the reuse distance sample. Then, it uses the reuse distance distribution to estimate the application's stack distances distribution. Finally, it estimates the application's MRC based on the estimated stack distance distribution.

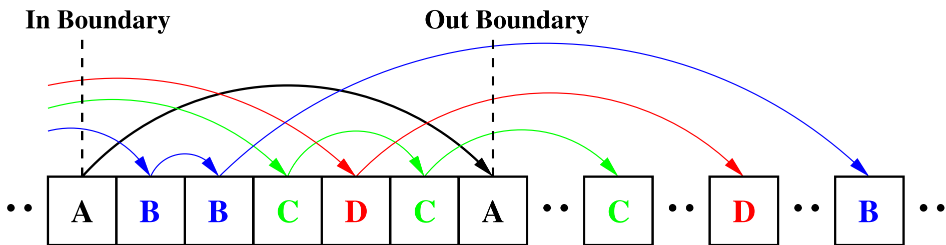


Figure 4.1: Reuse distances in a memory access stream. The arcs connect successive accesses to the same cache-line, and represents reuse of data. The stack distance of the second access to A is equal to the number of arcs that cross the "Out Boundary".

To understand how StatStack works, consider the address trace shown in Figure 4.1. Here the arcs connect subsequent accesses to the same cacheline, and represent the reuse of data. In this example, the second memory access to cacheline A has a reuse distance of five, since there are five memory accesses executed

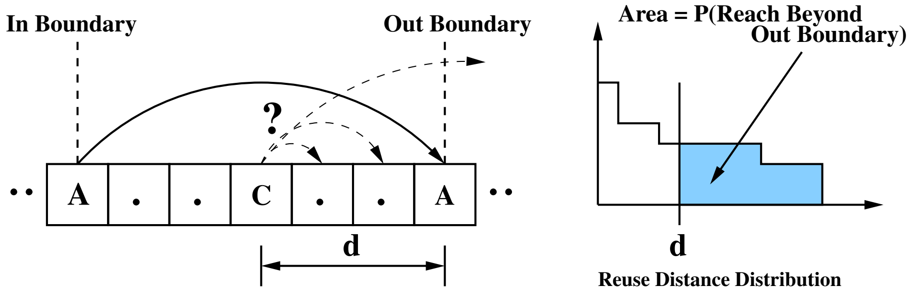


Figure 4.2: Expected stack distance estimation. Consider the arc originating at the memory access to cache-line C. The probability that the arc reaches beyond "Out Boundary" is equal to the fraction of reuse distances in the reuse distance distribution that are greater than d.

between the two accesses to A, and a stack distances of three, since there are three unique cache-lines accessed between the two accesses to A.

In Figure 4.1 we see that, for each unique cacheline that is accessed between the two accesses to A, there is a series of connected arcs that cross both the "In Boundary" and the "Out Boundary". Now, since the stack distance of the second access to A is equal to the number of unique cacheline accessed between the two accesses to A, it is equal to the number arcs that reach beyond the "Out Boundary".

The length of the arcs in Figure 4.1 are determined by the their respective reuse distances. If we know the reuse distance of all memory accesses executed between the two accesses to A, we can determine the stack distance of the second access to A by counting the number of arcs that reach beyond the "Out Boundary". However, the input to StatStack is a sparse distribution of reuse distances, and therefore does not contain this information for every memory access. Instead, we use the reuse distance distribution to compute the likelihood that the memory accesses executed between the two accesses to A have reuse distances such that their arcs reach beyond the "Out Boundary".

Figure 4.2 shows how to compute the probability that the outbound arc of the first memory access to cacheline C reaches beyond the "Out Boundary". In the figure, d denotes the memory access's distance to the "Out Boundary". If the reuse distance of the memory access is greater than d, its arc reaches beyond the "Out Boundary". StatStack makes the approximation that the probability of this is equal to the fraction of memory accesses in the application's reuse distance distribution that have reuse distances greater than d. Using the above method we can compute the probabilities that the arcs originating at the memory accesses between the two memory accesses to A, in Figure 4.2, reach beyond the "Out Boundary". By adding these probabilities we get the expected stack distance of the second memory access to A.

StatStack uses the above approach to compute the expected stack distances of all memory accesses in the input reuse distance distribution, which produces an expected stack distance distribution. From this distribution StatStack can then estimate the miss ratio for any given cache size, C, as the fraction of expected stack distances in the expected stack distance distribution that have stack distances greater than C.

We have evaluated the accuracy of StatStack using the SPEC CPU2006 benchmarks. The results show that Statstack accurately estimates MRCs based on reuse distance samples containing the reuse distance of only one out of every $10,000$ memory references. Furthermore, the execution time of StatStack (not counting the profiling) is less than a second.

4.2 StatCC

StatCC (Paper II) is a statistical cache contention model. It estimates the miss ratio of a set of co-running applications that share the last level cache based on the application's individual reuse distance distributions. StatCC explicitly models the coupling between the co-running applications' miss ratios and execution rates. The input to StatCC is the co-running applications' reuse distance distributions. This allows StatCC to use the same efficient reuse distances measurement technique as StatCache (Section 3.3), and therefore inherits StatCache's low overhead for collecting runtime information. Furthermore, given the co-running applications' reuse distance distributions, the analysis required to estimate their shard cache miss ratios is performed in only a few seconds.

Figure 4.3 shows the architectural model used by StatCC for a dual core system. The applications running on the cores generate individual memory reference streams (Stream 1 and Stream 2). These two streams are interleaved at the shared cache. The shared cache only sees the interleaved reference stream and operates as if it was generated by a single core. Therefore, given that we know the reuse distance distribution of the interleaved reference stream we can use StatStack to estimate its stack distance distribution. StatCC's is based on a transformation that given the execution rates of the co-running applications and their individual reuse distance distributions, estimates the reuse distance distribution of the interleaved reference stream. While doing this, StatCC keeps track of what stream (Stream 1 or Stream 2) the reuse distances in the interleaved reuse distance distributions originate from. This allows StatCC to compute the individual shared cache miss ratios of the co-running applications. However, the above transformation requires the execution rates of the co-running applications to be known. StatCC therefore uses a simple performance model (see Section 3.6) to estimate the execution rates of the

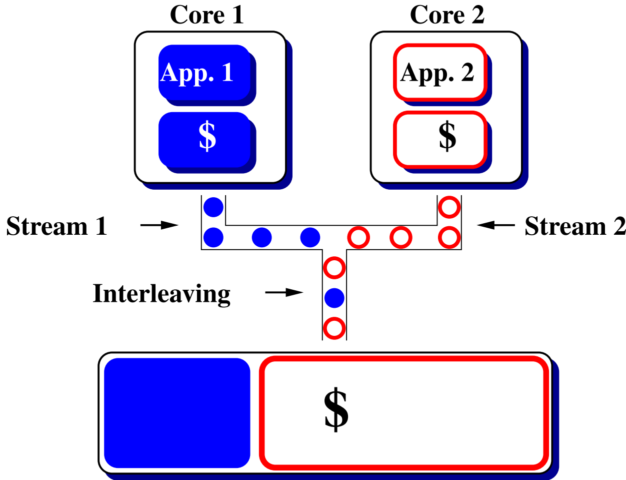


Figure 4.3: Architectural model. Shown here for two cores. The two applications (App. 1 and App. 2) generate two memory reference stream (Stream 1 and Stream 2). These two streams are interleaved at the shared cache.

co-running applications based on their shared cache miss ratios. The result is the following equation system,

$$\begin{cases} cpi_1 = CPI_MODEL\left(miss_ratio_1\left(cpi_1, cpi_2\right)\right) \\ cpi_2 = CPI_MODEL\left(miss_ratio_2\left(cpi_1, cpi_2\right)\right). \end{cases} \tag{4.1}$$

This equation system can be readily solved for the CPIs of the co-scheduled applications (cpi_1 and cpi_2) by a general purpose equation system solver. Then, to get the shared cache miss ratios, we simply plug these CPIs into the $miss_ratio(cpi_1, cpi_2)$ functions. To compute how much cache capacity each of the co-running applications receive, we can use (the inverse of) the applications' individual MRCs to find out at what cache capacity they experience the estimated miss ratios.

We have demonstrated the effectiveness of StatCC trough experimental evaluation using 55 pairs of SPEC CPU2006 applications. However, the equation system (Eq. (4.1)) that describes how co-scheduled applications contend for a shared cache, can easily be extended to any number of threads. We have successfully solved the equation system for several combinations of four co-scheduled synthetic applications, which suggests that the equation system is solvable for more than two co-scheduled (real) applications.

StatCC's low-overhead data collection and fast model evaluation make it well suited for many uses. For example, program cache optimization targeting a shared level cache, long-term cache contention aware job scheduling and processor design space exploration can all benefit significantly from fast data collection.

4.3 Cache Pirating

Cache Pirating (Paper III) is a method that allows us to accurately measure any performance metric available through hardware performance counters as a function of how much shared cache space is available to an application. In particular, we can measure execution rate (CPI), miss ratio (MPKI) and bandwidth demand (GB/s). We do this while the applications is running on real hardware, and therefore account for *all* effects of the memory hierarchy, such as non-standard replacement policies, hardware prefetching and memory level parallelism, which can be hard and expensive to accurately capture with simulation and performance models. Cache Pirating measure the desired performance metrics across a wide range of cache sizes with only a single execution of the measured application, resulting in an average execution time overhead of only 5%.

The basic idea of Cache Pirating is to control exactly how much shared cache space is available to the application under measurement (the Target) by co-running it with a cache-“stealing” application (the Pirate). The Pirate steals cache from the Target by ensuring that its entire working set is always resident in the shared cache. This effectively reduces the cache space available to the Target. Indeed, as long as the Pirate keeps its entire working set in the cache, we know that the Target has exactly the remainder of the cache space.

The Pirate must actively compete with the Target for cache space. For some target applications, it is hard for the Pirate to steal large amounts of cache, limiting the range of cache sizes that we can measure. How successful the Pirate is at stealing cache depends on the Target and how much it fights back. However, using performance counters we can readily determine if the Pirate is successful in stealing the requested amount of cache. When the fetch ratio of the Pirate is below a threshold (close to zero) virtually none of its data is fetched from main memory, and its working set is resident in the cache. To ensure this, we monitor the fetch ratio of the Pirate while measuring the performance of the Target. If the Pirate’s fetch ratio is above the threshold, we know that the Pirate cannot maintain its entire working set in the cache, and we discard the measurement.

We have evaluated the Cache Pirating technique and shown that it allows us to accurately measure the Target application’s performance and bandwidth demand as a function of its available shared cache space. By multi-threading the Pirate and dynamically varying its working set size as the Target runs, we are able to steal on average 6.1MB of a 8MB shared cache with an average runtime overhead of 5%, without impacting the Target’s measured performance. All of this is done without requiring special hardware or modifications to the Target application or operating system. This demonstrates that the Cache Pirating technique is a viable and accurate method for measuring any combination of hardware performance counter statistics for the Target application as a function of its available shared cache space.

Cache Pirating has enabled us collect performance data for real applications running on real hardware. The results show that as the available cache size is decreased

most application's bandwidth increases to compensate, resulting in relatively flat performance curves. To better understand this bandwidth increase, we can examine the fetch and miss ratio curves to determine how much of it is due to hardware prefetchers. The ability to collect and visualize this data allows us to analyze the performance impact of resource sharing in the memory system.

Bibliography

[1] A. Srivastava and A. Eustace. ATOM: a System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[2] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[3] B. T. Bennett and V. J. Kruskal. LRU Stack Processing. *IBM J. Res. Dev.*, 19:353–357, July 1975.

[4] Binkert, N.L., Hallnor, E.G., Reinhardt, S.K. Network-oriented full-system simulation using M5. In *Proceedings of Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2003.

[5] C. Cascaval and D. A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003.

[6] C. Ding and Y. Zhong. Predicting Whole-Program Locality Through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the Intl. Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[9] D. Genbrugge, S. Eyerman and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[10] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[11] R. Schone D. Molka, D. Hackenberg and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[12] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[13] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2005.

[14] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[15] F. Liu, X. Jiang and Y. Solihin. Understanding how Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[16] F. Olken. Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies. Technical Report LBL-12370, Lawrence Berkeley Lab, 1981.

[17] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski and A. Yoo. METRIC: Memory Tracing via Dynamic Binary Rewriting to Identify Cache Inefficiencies. *ACM Trans. Program. Lang. Syst.*, 29, April 2007.

[18] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[19] J. Seward. Bzip2. `http://bzip.org/`.

[20] M. Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[21] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2006.

[22] N. Nethercote and J. Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[23] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, February 2002.

[24] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[26] R. L. Mattson, J. Gecsei, I. L. Traiger D. R. Slutz. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[27] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor. In *Proceedings of the SPEC Benchmark Workshop*, 2007.

[28] S. Eyerman, L. Eeckhout, T. Karkhanis and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Trans. Comput. Syst.*, 27:3:1–3:37, May 2009.

[29] S. Laha, J. H. Patel and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. Comput.*, 37:1325–1336, November 1988.

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

[31] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[32] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995.

[33] X. E. Chen and T. M. Aamodt. Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs. In *International Symposium on Microarchitecture (MICRO)*, 2008.

[34] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[35] X. Shen, J. Shaw, B. Meeker and C. Ding. Locality Approximation Using Time. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[36] Y. H. Kim, M. D. Hill and D. A. Wood. Implementing Stack Simulation for Highly-Associative Memories. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

Paper I

StatStack: Efficient Modeling of LRU Caches

David Eklov and Erik Hagersten
Uppsala University, Department of Information Technology
{david.eklov, eh}@it.uu.se

Abstract

Efficient execution on modern architectures requires good data locality, which can be measured by the powerful *stack distance* abstraction. Based on this abstraction, the miss rate for LRU caches of any size can be predicted. However, measuring stack distance requires the *number of unique memory objects* to be counted between successive accesses to the same data object, which requires complex and inefficient data collection.

This paper presents a new efficient way of estimating the stack distances of an application. Instead of counting the number of unique memory objects touched between successive accesses to the same data, our scheme only requires the *number of memory accesses* to be counted, a task efficiently handled by existing built-in hardware counters. Furthermore, this information only needs to be captured for a small fraction of the memory accesses. A new efficient off-line algorithm is proposed to estimate the corresponding stack distance based on this sparse information.

We evaluate the accuracy of the proposed estimation compared with full stack distance measurements for 28 of the applications in the SPEC CPU2006 benchmark suite. The estimation shows excellent accuracy based on information about only every 10,000th memory access.

I.1 Introduction

The high latency and limited bandwidth of DRAM has been identified as two potential bottlenecks of present and future computer systems [20]. However, both memory latency and bandwidth issues can often be tolerated if an application's data locality is improved, leading to a better use of the caches in the memory hierarchy. To accomplish this, we need powerful techniques to give insight into application data locality. Furthermore, such techniques have to be efficient enough to evaluate long-running applications operating on large input data sets.

Stack distance, introduced by Mattson [17], is the basis for several such techniques. The stack distance is the number of unique memory objects accessed during a *reuse epoch*, where a reuse epoch is the time between two successive memory accesses to the same memory object. For cache analysis, the memory objects are usually cache lines. Based on the distribution of stack distances for all memory operations of an application, the miss ratio of a fully associative LRU cache of any size can be calculated trivially and quickly [17]. Since its introduction, stack distance has been widely used, for example, to model cache reuse [8], to guide program transformation [27] and insertion of cache hints [5], to find locality phases [22], and for modeling of cache contention between parallel processes [6].

While the stack distance abstraction is powerful, the requirement to count the number of unique memory objects accessed during every reuse epoch requires elaborate bookkeeping, which results in complex and slow algorithms. Furthermore, the traditional way of implementing a stack distance algorithms requires full address traces collected from parts of or the full execution of the target application. We have measured slowdowns of $100-1000\times$ while collecting full address traces, using the Pin [14] and Valgrind [18] dynamic instrumentation tools.

The goal of this research is to find an inexpensive way to model LRU caches, based on sparse and easily captured information, such as that used by the StatCache [2][3]. Instead of keeping track of the number of *unique memory objects* accessed during reuse epochs, StatCache simply records the *number of memory accesses* executed during the reuse epochs, called a reuse distance. Reuse distances can be measured with low overhead by leveraging hardware counters and watchpoint mechanisms provided by modern CPUs and operating systems. Furthermore, StatCache only measures the reuse distance for a very sparse selection of memory accesses, for example, only one out of every $10,000$ memory accesses. Practical experiments using this method report an average runtime overhead of 40 percent [3], compared to $100 - 1000\times$ to capture full address traces. StatCache then uses an off-line statistical model to estimate the miss ratio for an arbitrary size cache with a *random-replacement* policy.

The sparsely collected reuse distance information from StatCache appear to be a poor fit for building up the state needed to model LRU caches. However, it is possible to use the sparse reuse distance information to infer much of the lost state. Instead of explicitly trying to count the number of unique memory objects accessed during each reuse epoch, we start off with an algorithm that calculates

stack distances based on the reuse distances of all memory accesses. The stack distances can then be estimated based on sparse reuse distance information by applying several approximations. The validity of these approximations is evaluated through experiments, which show excellent accuracy for reuse distance information for only one out of every 10,000 memory accesses.

The paper continues by reviewing StatCache and its efficient data collection. Next, in Section I.3, we give a rigorous description of StatStack. In Section I.4 and I.5, we discuss possible sources of errors and describe a scheme for collecting runtime data that minimizes these errors. Finally, in Section I.6, we evaluate the accuracy of StatStack by comparing its results with the output from a traditional cache simulator fed with full address traces for 28 applications. We show that a sparse sample of reuse distances can be used to model a fully-associative LRU cache accurately across a wide range of cache sizes.

I.2 StatCache in Review

StatCache is a tool for estimating an application's cache miss ratio [2][3]. It uses an online sampler that attaches to the target application and measures the reuse distances of a set of randomly selected memory references. The reuse distance of a memory reference A is equal to the number of memory references in the target application's address trace between A and the previous memory reference to the same cache line. The sampler records the set of measured reuse distances in a structure called a reuse distance sample or RDS, which is fed as input to an offline statistical cache model. StatCache models a fully associative cache with a random replacement policy, and can accurately estimate miss ratios using RDSs collected with sample rates as low as 10^{-6} [3], i.e., only containing reuse information of every one millionth memory access.

Unlike collection of address traces, the collection of reuse distances allows the sampler to make efficient use of functionality supported by contemporary hardware and operating systems, such as hardware performance counters and watchpoints. The StatCache sampler enables sampling of unmodified binaries and has a relatively small impact on the sampled applications runtime. Berg et al. [3] report an average slowdown of only 40 percent when hardware and operating system support are used.

The StatCache sampler works as follows. Hardware counter overflow traps are used to halt the execution when a memory reference that has been selected for sampling executes. A one-time watchpoint is then set for the address it accesses. The number of memory accesses performed to date is recorded by reading a hardware counter, after which the execution is continued. The next time the same address is accessed, a watchpoint trap is generated, which again halts the execution. The reuse distance is calculated as the difference between the number of memory accesses executed to date and the previously recorded number.

I.3 StatStack

StatStack, like StatCache, is a method for estimating an application's cache miss ratio. StatStack introduces a new statistical cache model that, unlike the statistical cache model of StatCache, models a fully associative cache with an LRU replacement policy. The StatStack cache model uses the same RDS input data as the cache model of StatCache. This allows StatStack to use the same efficient sampling technique as StatCache, and thereby inherit StatCache's low runtime overhead.

A natural approach to modeling of LRU caches is to use the stack distance abstraction. However, the sparse input data used by StatStack does not contain enough information to support traditional methods for computing stack distances. StatStack is based on a new approach that uses the concept of an *expected stack distance*, which is roughly the average number of unique cache lines accessed during a reuse epoch.

Our approach can be summarized as follows: 1) Based on the reuse distance information in an RDS, we estimate the target application's reuse distance distribution; 2) We use that distribution to determine the likelihood that a memory access, to cache line l, is the last of the memory accesses executed in the current reuse epoch to access l; 3) Based on these likelihoods, we estimate the average number of unique cache lines accessed during each reuse epoch recorded in the RDS, which gives us the expected stack distances of the sampled memory accesses; 4) We then use these expected stack distances to estimate the target application's miss ratio.

In the remainder of this section, we develop the StatStack cache model. We start by introducing a fundamental relation between reuse and stack distance (Section I.3.1). Based on this relation we derive an expression for expected stack distances given the knowledge of all reuse distances in the target application's address trace (Section I.3.2). We then introduce a set of approximations that enable us to the estimate expected stack distances based only on the sparse information in a RDS (Section I.3.3). Finally, we put the pieces together and show how the miss ratio of the target application is estimated based on expected stack distances (Section I.3.5).

I.3.1 Stack Reuse Relation

In this section, we introduce a relation between reuse distance and stack distance. This relation allows us to compute the stack distance of a memory access, x, given that we know the reuse distance of all memory accesses executed between x and the previous memory accesses to the same cache line. Before proceeding, we need to make our terminology more precise.

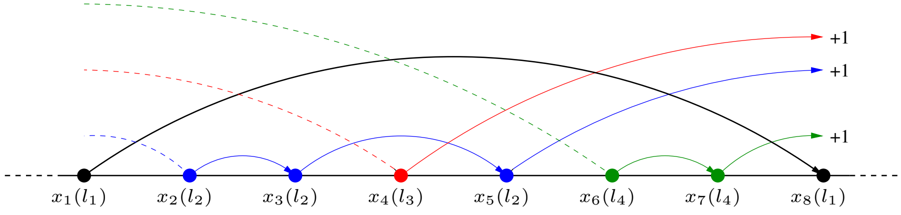


Figure 1: The figure shows a sequence of memory references in an address trace. Memory references are marked by dots on the horizontal time line. The labels below the circles show the name of the memory references and the cache line accessed in parenthesis. The arcs connect successive memory references to the same cache line.

Definition I.3.1 *Let x_i and x_j be two successive memory accesses the same cache line, then*

- *the **reuse distance** of x_j is the number of memory references executed between x_i and x_j,*
- *the **forward reuse distance** of x_i is the number of memory references executed between x_i and x_j,*
- *the **stack distance** of x_j is the number of distinct cache lines accessed by the memory references executed between x_i and x_j.*

The relation between reuse distance and stack distance is best explained by an example.

Example 1: Figure 1 shows a sequence of an address trace, where memory references are marked by dots on the horizontal time axis. The memory references are labeled according to their position in the address trace. The arcs connect successive memory references to the same cache line to indicate a reuse epoch, for example, consider the reuse epoch between x_1 and x_8. The stack distance of x_8 is three, since there are three unique cache lines, l_2, l_3 and l_4, accessed by the memory references executed between x_1 and x_8. The three cache lines, l_2, l_3 and l_4, have to be accessed one and only one last time by the memory references executed between x_1 and x_8. The last of these memory reference to l_2, l_3 and l_4 are x_5, x_4 and x_7, respectively. These are exactly the memory references executed between x_1 and x_8 whose outbound arcs reach beyond x_8, i.e. their forward reuse distance is greater than their distance to x_8. The stack distance of x_8 is therefore equal to the number of memory references executed between x_1 and x_8 with a forward reuse distances greater than their distances to x_8, which is three in this example.

The result of the above example holds true in general. To show this we introduce the following notation: Let $R(x_t)$ and $\vec{R}(x_t)$ denote the reuse distance and the forward reuse distance of x_t, respectively. Further, let $Q(x_t)$ be the set of memory references between x_t and the previous memory reference to the same cache line. For example, in Figure 1, $Q(x_8) = \{x_2, x_3, x_4, x_5, x_6, x_7\}$. We can now generalize the result of Example 1 as follows: The stack distance of a memory reference x_t is

equal to the number of memory references, $x_i \in Q(x_t)$, for which $\vec{R}(x_i) > t - i$. This statement follows from the observation that $\vec{R}(x_i) > t - i$ if and only if x_i is the last of the memory references in $Q(x_t)$ to access cache line l_i. The stack distance of x_t, denoted by $S(x_t)$ can now be expressed as follows,

$$
\begin{aligned}
S(x_t) &= \sum_{i=t-r}^{t-1} \mathbf{1}(\vec{R}(x_i) > t - i) \\
&= \sum_{j=1}^{r} \mathbf{1}(\vec{R}(x_{t-j}) > j)
\end{aligned} \tag{1}
$$

where, $r = R(x_t)$ and $\mathbf{1}(\alpha)$ is defined to be one if α is true and zero otherwise. The second equality in (Eq. 1) follows from the variable substitution $j = t - i$. (Eq. 1) can also be derived from equations presented by Bennett and Kruskal [1] (see Appendix).

I.3.2 Expected Stack Distance

To compute the stack distance of a memory reference x_t using the method of Section I.3.1 we need to know the forward reuse distance of all memory references executed between x_t and the previous memory reference to the same cache line. However, our goal is to approximate an application's miss ratio given only the reuse distance of a sparse set of the application's memory references. To this end, we introduce the concept of an *expected stack distance*. We start by assuming that we know the reuse distance of all memory references in the target application's address trace, and derive an expression for the expected stack distances based on this information. Later, we introduce a set of approximations that enable us to estimate expected stack distances based on the sparse information in an RDS.

The expected stack distance of the memory references with a reuse distance of r is the average stack distance of all memory references with a reuse distance of r. As an example, consider the three memory references x_i, x_j and x_k in Figure 2. Using the method of Section I.3.1, we can compute their stack distances, which are one, two and three, respectively. Assuming that these three memory references are the only memory references with a reuse distance of three, we can compute their expected stack distance, simply by averaging their stack distances, which in this case gives us an expected stack distance of $\frac{1}{3}(1 + 2 + 3) = 2$.

To derive an analytical expression for the expected stack distance we introduce the following notation. Let $T(r)$ be the set of all memory references with a reuse distance of r in the target application's address trace, and let n_r be the number of such memory references. We can now express the expected stack distance of the memory references with a reuse distance of r, denoted $ES(r)$, as follows,

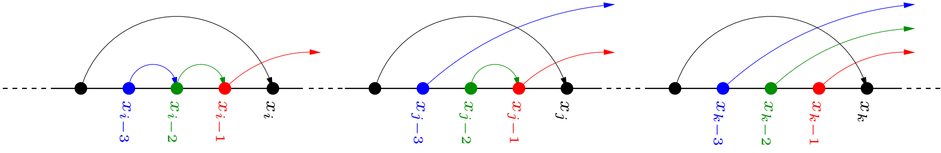


Figure 2: The figure shows a sequence of an address trace. Memory references are marked by circles on the horizontal time axis. The figure shows only the three memory references, x_i, x_j and x_k, with a reuse distance of three

$$
\begin{aligned}
ES(r) &= \frac{1}{n_r} \sum_{x_i \in T(r)} S(x_i) \\
&= \frac{1}{n_r} \sum_{x_i \in T(r)} \sum_{j=1}^{r} \mathbf{1}(\vec{R}(x_{i-j}) > j)
\end{aligned}
\tag{2}
$$

We get the second equality in (Eq. 2) by substituting $S(x_i)$ with the right hand side of (Eq. 1). We again consider the memory reference x_i, x_j and x_k in Figure 2, but this time we compute their expected stack distance using (Eq. 2). By expanding the two nested sums in (Eq. 2), we get the following expression,

$$
\begin{aligned}
&ES(3) = \\
&\frac{1}{n_r} \cdot (\mathbf{1}(\vec{R}(x_{i-1}) > 1) + \mathbf{1}(\vec{R}(x_{j-1}) > 1) + \mathbf{1}(\vec{R}(x_{k-1}) > 1)) + \\
&\frac{1}{n_r} \cdot (\mathbf{1}(\vec{R}(x_{i-2}) > 2) + \mathbf{1}(\vec{R}(x_{j-2}) > 2) + \mathbf{1}(\vec{R}(x_{k-2}) > 2)) + \\
&\frac{1}{n_r} \cdot (\mathbf{1}(\vec{R}(x_{i-3}) > 3) + \mathbf{1}(\vec{R}(x_{j-3}) > 3) + \mathbf{1}(\vec{R}(x_{k-3}) > 3))
\end{aligned}
\tag{3}
$$

Notice the order in which the terms are arranged. For example, the first row in (Eq. 3) is the fraction of the memory references, x_{i-1}, x_{j-1} and x_{k-1}, with a reuse distance greater than one, the second row is the fraction of the memory references x_{i-2}, x_{j-2} and x_{k-2}, with a reuse distance greater than two, and so on. We denote these fractions f_1, f_2 and f_3, respectively. The benefit of rearranging the terms of (Eq. 2), is the following. Given the reuse distance distributions for the memory references appearing in each row, we can compute f_1, f_2 and f_3 simply by computing the fraction of forward reuse distances in the respective distributions that are greater than one, two and three, respectively. This will later allow us to make the approximations necessary to estimate expected stack distances based on the sparse reuse distance information in an RDS. Above, a subscript of i denotes that f_i, is the number of memory references with a forward reuse distances greater than i, at position i counted backwards from the end, in

the reuse epochs of length three. Generalizing the notation by adding a superscript that denotes the length of the reuse epochs in question, allows us to write the following expression for the expected stack distances,

$$ES(r) = \sum_{i=1}^{r} f_i^r. \tag{4}$$

As an aside we note that the f_i^r fractions can be interpreted as a likelihood: If we were to pick one of the memory references x_{i-2}, x_{j-2} and x_{k-2} in Figure 2 at random, we can interpret f_2^3 as the likelihood that reuse distance of the memory reference is greater than two.

I.3.3 RDS Approximation

To compute an expected stack distance using (Eq. 4), we need to compute all the f_i^r fractions. For each of these fractions, we need to know the reuse distances distribution for a specific set of memory references. For example, to compute f_2^3 in Figure 2, we need to know the reuse distances distribution of x_{i-2}, x_{j-2} and x_{k-2}. In this paper, we use RDSs that contain the reuse distances of only one out of every $10,000$ to $50,000$ memory references on average. It is therefore unlikely that these RDSs contain enough reuse distances to accurately estimate the f_i^r fractions. Our approach is to introduce an approximation that relaxes the constraint of having to know the reuse distance distributions of these specific sets of memory references.

The approximation that we use relies on the assumption that the reuse distance distribution observed for any (large enough) set of memory references is the same as the reuse distance distribution all memory references in the address trace. Under this assumption, we can make the following approximation,

$$f_i^r \approx F_i, \tag{Approx. A}$$

where, F_i is the fraction of all memory references with a reuse distance greater than i in the target application's address trace. With the above approximation, we can use the reuse distance distribution of all memory references in the target application to compute the f_i^r fractions. This might seem like a step in the wrong direction, we now need to know the reuse distance of more, in fact all, memory references to compute the f_i^r fractions. However, the benefit of this is that we can now use all reuse distances in an RDS to estimate the reuse distance distribution needed to compute the expected stack distances. Let $\hat{F}_i$ be the fraction of reuse distances in an RDS with a reuse distance greater than i, we can now make the following approximation,

$$F_i \approx \hat{F}_i. \tag{Approx. B}$$

By putting together, (Approx. A), (Approx. B) and (Eq. 4), we get the following approximate expression for expected stack distance,

$$ES(r) \approx \sum_{i=1}^{r} \hat{F}_i. \qquad (7)$$

We have now arrived at an expression that allows us to compute the expected stack distance of any memory reference in the target application's address traces, given only its reuse distance and a sparse RDS collected from the target application.

In order to move forward with the development of the cache model we defer the discussion of the implications of the above approximation to Section I.5. However, it is important to recognize the negative effects that program phase changes [7][23] can have on (Approx. A). Fortunately, the StatCache sampler has a sampling mode, discussed in Section I.4, that allows us to estimate the expected stack distances of memory references executed in different program phases separately. As we will see in Section I.6, this will under most circumstances, eliminate the sensitivity to program phase changes.

I.3.4 Cold Misses

Since the number of cold misses experienced by an application is independent of the replacement policy of the cache, we can use the same method as StatCache to compute the target application's cold miss ratio. This method is based on the concept of dangling samples. When a memory reference that the StatCache sampler has selected for monitoring executes, the sampler sets a watchpoint for the address being accessed. If this address is not accessed again, the watchpoint will never trigger. When the application has finished its execution, the untriggered watchpoints are recorded in the RDS as dangling samples.

The number of cold misses experienced by an applications is proportional to the number of dangling samples [4]. A cold miss occurs when the application accesses cache-line-sized piece of memory for the first time. Every cache-line-sized piece of memory accessed by the application is also accessed one last time and if the last memory references is sampled it will result in a dangling sample. Since the sampler samples all memory accesses with equal probability, the cold miss ratio is equal to the number of dangling samples divided by the number of reuse distances in the RDS.

I.3.5 Cache Model

In this section, we put the pieces together and show how StatStack estimates an application's miss ratio including booth cold and capacity misses from its input data, an RDS collected by the StatCache sampler.

Armed with equation (Eq. 7), we can estimate the expected stack distance distribution of the target application given its RDS as follows. First, we compute the expected stack distance of each distinct reuse distance in the RDS using (Eq. 7). Our implementation of StatStack does this in three steps: First, it sorts the reuse distances in the RDS into a histogram data structure. It then computes all $\hat{F}_i$'s, in a single pass over the histogram's buckets. Finally, it computes the expected stack distance for each distinct reuse distance in the RDS, by computing a running sum over the sorted sequence of $\hat{F}_i$'s. Then, by weighting each of the expected stack distances with the frequency of the corresponding reuse distance in the RDS we get the expected stack distance distribution.

To estimate the target application's capacity miss ratio, we approximate its actual stack distance distribution with its expected stack distance distribution; we discuss this approximation further in Section I.5.1. For a fully associative cache with an LRU replacement policy, a memory reference will result in a cache miss if its stack distances is greater than the cache size, measured in number of cache lines [17]. We can therefore compute the target application's capacity miss ratio for a given cache size C, simply by computing the fraction of stack distances in its expected stack distance distribution that are greater than or equal to C. Finally, by adding the cold miss ratio to the capacity miss ratio we get the target application's miss ratio for a fully associative cache with an LRU replacement policy.

Figure 3 shows the miss ratios estimated by StatStack as a function of cache size for the SPEC CPU2006 benchmarks, together with a reference miss ratio obtained from a trace driven cache simulator. The RDSs used to estimate these miss ratios have been collected using a sample rate of 10^{-4}, i.e. the RDSs contain the reuse distances of only one out of every $10,000$ memory references. These results will be analyzed in Section I.6, where we perform a sensitivity analysis to quantify the impacts of the approximations on the accuracy of the cache model.

I.4 Hierarchical sampling

In this section, we discuss a hierarchical sampling policy implemented by the StatCache sampler. In this work, we use the hierarchical sampling policy for two reasons: First, it reduces StatStack's sensitivity to program phase changes, and second, it reduces the runtime overhead of the sampler.

Under the hierarchical sampling policy, the sampler alternates between two phases, the sampling phase and the hibernation phase. It is only during the sampling phase that the sampler starts new reuse distance measurements. The sampler stays in the sampling phase for a fixed number of memory references. We call the duration of a sampling phase a sampling window. The hibernating phases are of random length; this is to ensure random sampling. When a sampling phase comes to its end, the watch points are kept alive so that no measurements are lost. For most applications, the majority of reuse distances are short so most of the watch

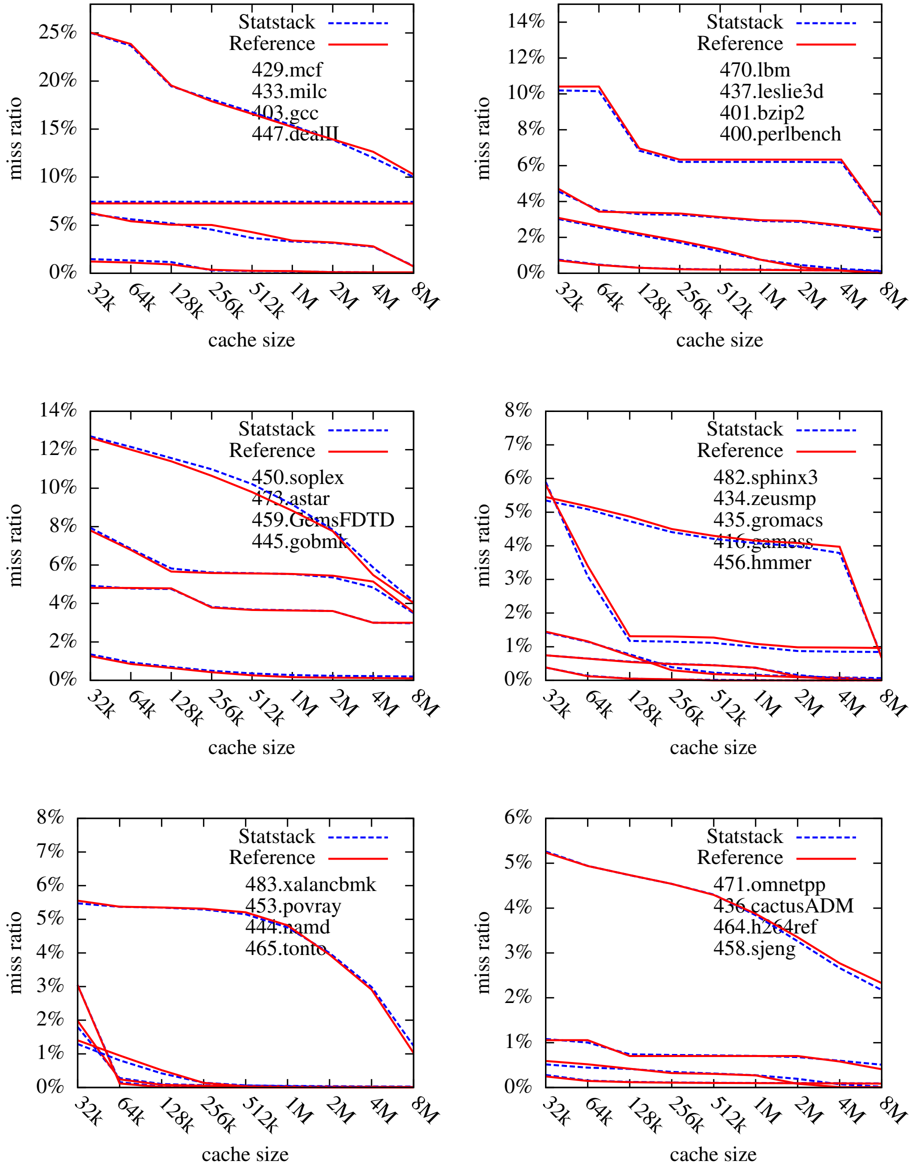


Figure 3: The graph show the miss ratio estimated by StatStack (labeled StatStack) as a function of cache size, together with a reference miss ratio (labeled Reference) obtained from a trace driven cache simulator, for 25 SPEC CPU2006 benchmarks. The graphs are ordered by their y-intercept according to the list on the right. The RDSs used to estimated the miss ratios were collected with a sample rate of 10^{-4}.

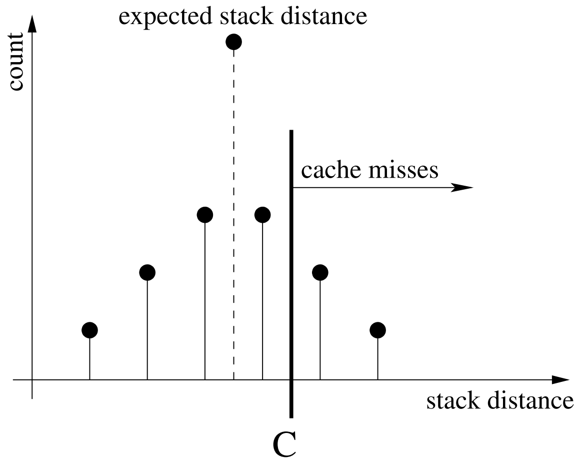


Figure 4: The figure shows the distribution of stack distances of a set of memory references with a reuse distance of r in the form of a histogram. The vertical line labeled C represents the cache size.

points trigger early in the hibernation phase. This allows the target application to execute at close to native speed during the sampler's hibernation phases.

The sampler tags the reuse distances it measures with the id of the window in which the measurement started. This allows the cache model to estimate the miss ratio of each sampling window individually. As we alluded to in Section I.3.3, this is what eliminates the cache model's sensitivity to program phase changes.

The StatCache sampler exposes three parameters to the user, the sampling phase length (s), the average hibernation phase length (h), and the average number of samples per sampling phase (n). The setting of these parameters is important since they affect the accuracy of the cache model and the runtime overhead of the sampler.

I.5 Error Sources

To be able to drive the cache model with the sparse reuse distance information in a RDS we have introduced a number of approximations that are potential sources of errors. We group these error sources into two categories, approximation errors and statistical sampling errors. These categories are further divided into type I and type II approximation error and type I and type II statistical sampling errors. In the remainder of this section we discuss these error sources one by one.

I.5.1 Type I Approximation Error

StatStack computes the expected stack distance distribution of the target application and uses it to approximate the target application's miss ratio. By doing this, StatStack approximates the application's actual stack distance distribution with its expected stack distance distribution. Since these two distributions are generally not the same, this approximation can be a potential source of errors.

Figure 4 shows an example stack distance distribution of memory references with a reuse distances of r and also their expected stack distance. The memory references with a stack distance greater than the cache size will result in a cache miss. As shown in Figure 4 the expected stack distance of the memory references for this distribution is less than the cache size. The miss ratio calculated from the *expected* stack distance distribution is therefore zero. However, since some of the stack distances are larger than the cache size, the miss ratio calculated from the *actual* stack distance distribution is not zero.

The above error tend to be large if the application makes a large number of memory references with the same reuse distances but with largely varying stack distances. This is most likely to be the case for applications having program phases with largely different cache behaviors. However, since, the cache model estimates the miss ratio for each sampling window individually; it is only for the windows that span a program phase change that this is a problem. This error is reduced by having the sampler use a large number of small sampling windows. This results in the portion of windows that span program phase changes being small, and therefore their impact on the overall estimated miss ratio is low.

I.5.2 Type II Approximation Error

In Section I.3.3, we made the assumption that the reuse distance distribution observed for any (large enough) set of memory references is the same as the reuse distance distribution all memory references in the address trace. If this assumption does not hold, the accuracy of the cache model will suffer.

For example, if an application's reuse distance distribution changes over time, the above assumption does not hold. However, for these types of applications, we can reduce the problem, by using a sampling window size small enough so that the reuse distance distribution stays the same within the sampling windows. Another scenario where the assumption does not hold is for applications whose memory references' reuse distances display certain patterns. For example, every memory references with a forward reuse distance of length r is followed by r memory references with a reuse distance of 1. In this case sampling windows does not help. However, as we show in Section I.6, the approximation errors are small, which suggests that reuse distance patterns that compromise the accuracy of the cache model are rare.

We have argued that both the type I and type II sampling errors can be reduced by using short sampling windows. However, too short sampling windows can hurt

the cache model. Consider a memory references with a reuse distance of r, as we showed in Section I.3.1, its stack distance is determined by the forward reuse distance of the r memory references executed before it. If the sampling window size is smaller than r, the sampler cannot capture the reuse behavior of these r memory references. When modeling a cache of size C, the accuracy of the estimated stack distances with lengths close to C will have the largest impact on the overall accuracy of the cache model. Therefore, the window size should typically not be shorter than the reuse distances of the memory accesses with expected stack distances close to the cache size.

I.5.3 Type I Statistical Sampling Error

StatStack estimates the target application's reuse distance distribution with the distribution of reuse distances in a sparse RDS. This gives rise to the type I statistical sampling error. The magnitude of the type I statistical sampling error depends on the number of reuse distance measured in the sampling windows, and the variance of the target application's reuse distance distribution.

I.5.4 Type II Statistical Sampling Error

The hibernation phase of the hierarchical sampling policy introduces a second type of statistical sampling error, which we call the type II statistical sampling error.

The use of non-zero-length hibernation phases as compared to zero-length hibernation phases, can roughly be thought of as randomly selecting only a sub set of the sampling windows for estimating the overall miss ratio. Since the overall miss ratio is the average of the miss ratios for the sampling windows, the type II statistical sampling errors can be analyzed using standard statistical techniques for random sampling. This type of analysis, for the similar method of trace sampling [13][16], has been thoroughly investigated in previous work [26]. In essence, the type II statistical sampling error depends on the average hibernation phase length and how much the target application's miss ratio varies over time.

I.6 Evaluation

In this section, we evaluate the accuracy of the StatStack cache model by comparing its output; an estimated miss ratio, to a reference miss ratio obtained from a traditional trace driven cache simulator, for the SPEC CPU2006 benchmark suite.

I.6.1 Experimental Setup

We use 28 out of the 29 programs in the SPEC CPU2006 benchmark suite (481.wrf did not compile on our system) run with their first reference input sets. All bench-

mark programs are compiled using GCC version 4.1.2 with optimization level 02 targeting an x86_64 system.

In order to evaluate the accuracy of our model, we first collect reference address traces from the benchmarks programs using an in-house instrumentation tool. The address trace collection is started after approximately 60 seconds of uninstrumented execution for all programs except for 403.gcc whose total execution time with its first reference input set is less than 60 seconds. All traces used for evaluation contains five billion memory access. This trace size was chosen for practical reasons.

To obtain reference miss ratios we use a trace-driven cache simulator, configured to simulate a fully associative cache with an LRU replacement policy for cache sizes ranging from 32kB up to 8MB. The reference miss ratios include both cold and capacity misses. We then collect sparse RDSs by measuring the reuse distances of randomly selected memory references in the address traces using the hierarchical sampling policy. We finally use an implementation of the StatStack cache model, as described in Section I.3, to estimate the miss ratios of the traces for cache sizes ranging from 32kB up to 8MB.

I.6.2 Sampler Parameters

As mentioned in Section I.4, the StatCache sampler has three parameters: sampling phase length (s), average hibernation phase length (h) and average number of samples per sampling phase (n). We have empirically found two sets of parameters, S_1: $\{s = 10^6, h = 14 \cdot 10^6, n = 1500\}$ and S_2: $\{s = 10^6, h = 74 \cdot 10^6, n = 1500\}$ that work well for our benchmarks. Here s and h are measured in number of executed memory references. These two sets of parameters result in sample rates of 10^{-4} and $2 \cdot 10^{-5}$ respectively. Unless otherwise noted, these are the parameters used to collect the RDSs in this section. For an in-depth discussion of how we found these sampler parameters please see [11].

I.6.3 Sensitivity Analysis

In this section, we present the results of two of experiments. In the first, we evaluate the cache model's sensitivity to the approximations errors, and in the second, we evaluate the cache model's sensitivity to the sampling errors.

Approximation Sensitivity

To evaluate the cache model's sensitivity to the approximation errors, we collect RDSs that contain the reuse distances of all memory references in the address traces and estimate miss ratios using these RDSs. Since these non-sparse RDSs contain the reuse distance of all memory references, we effectively eliminate the sampling errors. By comparing these miss ratios to the reference miss ratios, we can evaluate the approximation errors in isolation. Note that, even though we mea-

sure the reuse distance of all memory references in the address traces we still use a sampling window of size $s = 10^6$ (the other two parameters are set to $h = 0$ and $n = 10^6$).

Figure 5 shows the estimated miss ratio next to the reference miss ratio for all benchmark (except for 429.mfc[1], 433.milc[1], 471.omnetpp[1], 410.bwaves[2] and 453.povray[2]). The close agreement between the estimated and reference miss ratios indicates that the approximation errors' are relatively small.

The largest discrepancies between the estimated and the reference miss ratios are for small cache sizes of less than 64KB. This is likely due to the type I approximation error: For most applications the distribution of stack distance are heavily weighted towards short stack distances. This makes it likely that there are a large number of memory references with the same reuse distance that display the behavior shown in Figure 4 for small cache sizes. The application with the largest error for large cache sizes is 473.astar for which the estimated miss ratio is somewhat lower than the reference for 4MB caches. This indicates that some portion of the expected stack distances with lengths close to 64k (4MB / line-size) are underestimated, this is likely due to the type II approximation error.

Sampling Sensitivity

In order to evaluate the sampling errors we sampled the address traces, using sampler parameters S_1, and estimated their miss ratios 32 times each. By observing the differences of the estimated miss ratios we can evaluate the sampling errors. Figure 6 shows three graphs for each benchmark application: Statstack-min and Statstack-max, show the minimum and the maximum of the 32 estimated miss ratios, the third graph labeled Reference, show the references miss ratio. The close proximity of the min and max graph indicates that largest differences of the 32 estimated miss ratios are small, which further implies that the sampling errors are small.

We repeated the above experiment, but this time with the sampler parameters S_2, the results are shown in Figure 7. Since RDSs collected using S_2 contains on average 5 times fewer reuse distance than RDSs collected using S_1, we expect the sampling error to be larger in Figure 7 than in Figure 6, which is indeed the case. However, the only difference between the two sets of sampler parameters is that S_2 has longer hibernation periods and therefore contains less sampling windows; the larger sampling error in Figure 7 is therefore due to the type II sampling error.

To gain further insight into the sampling errors, we consider the distributions of errors for the two sets of sampler parameters as shown Figure 8. These error distributions are computed using the same data as above, which contains 32 miss ratios for each application and each cache size. We compute the errors for each of these sets of 32 miss ratios individually as the absolute error with respect to the

[1]Due to the large dispersion of reuse distance, the amount of memory needed for simulation exceeds the amount of memory available on our system.

[2]Both the reference and the estimated miss ratio are zero for cache sizes larger than 32kB.

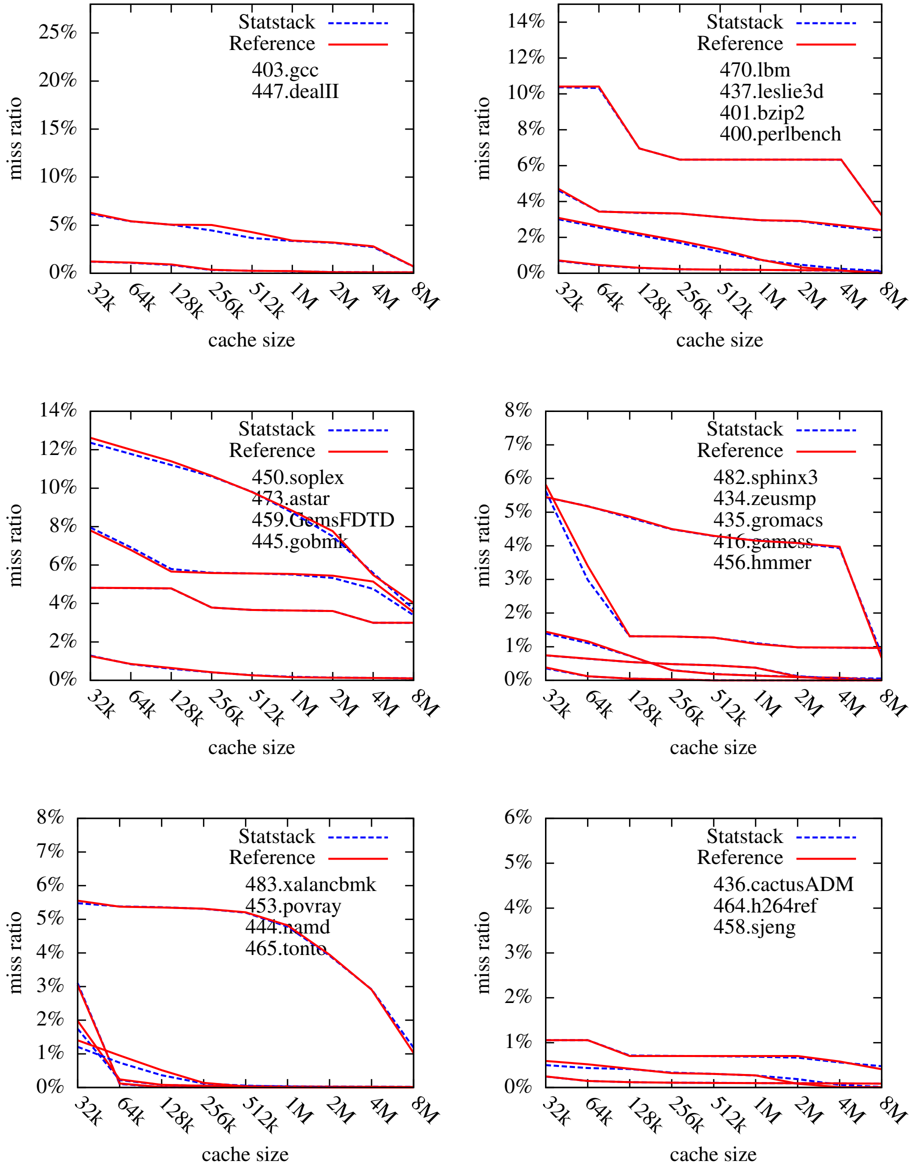


Figure 5: Miss ratio as a function of cache size. The graph labeled StatStack shows a miss ratio estimated using RDSs containing the reuse distance of all memory accesses in the address trace. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

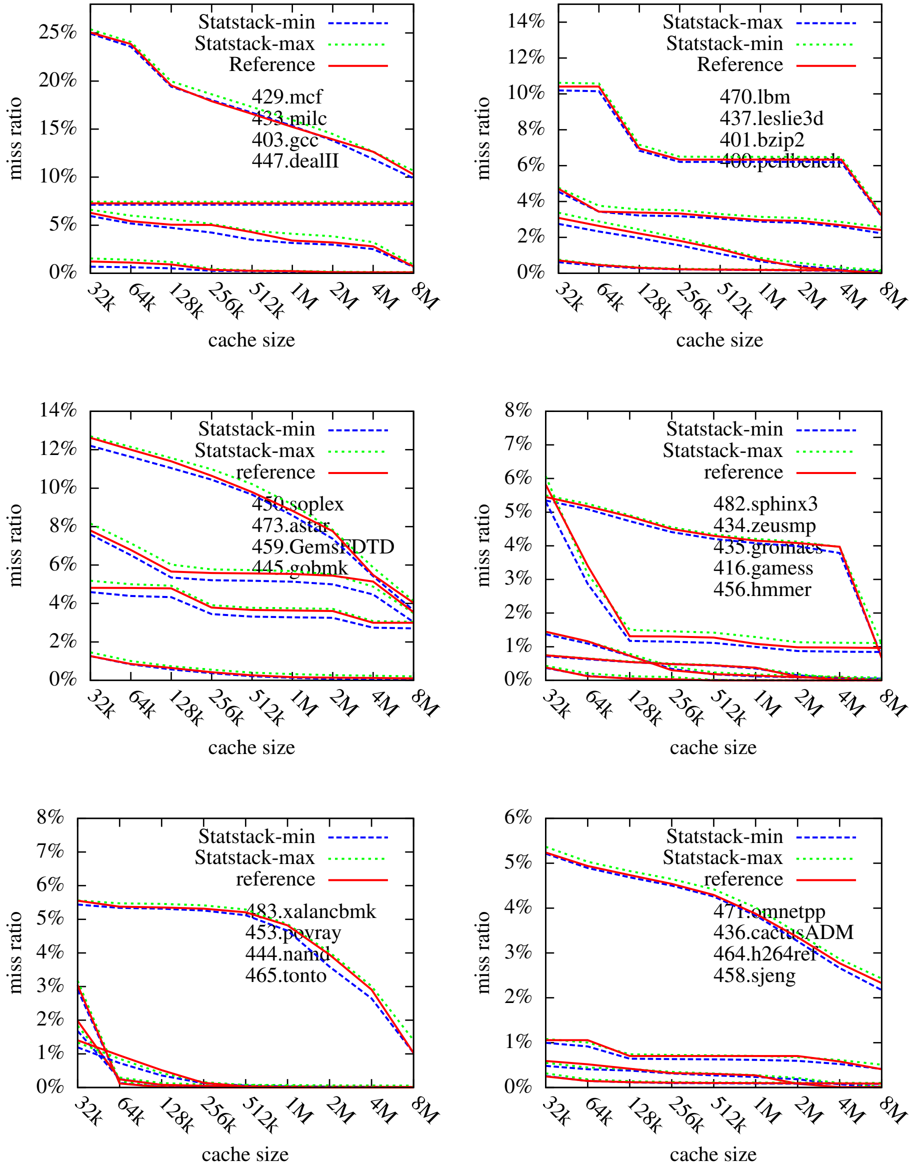


Figure 6: Miss ratios as a function of cache size. The graphs labeled StatStack-min and Statstack-max shows the minimum and maximum of 32 estimated miss ratios, based on RDSs collected with sampler parameters S_1. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

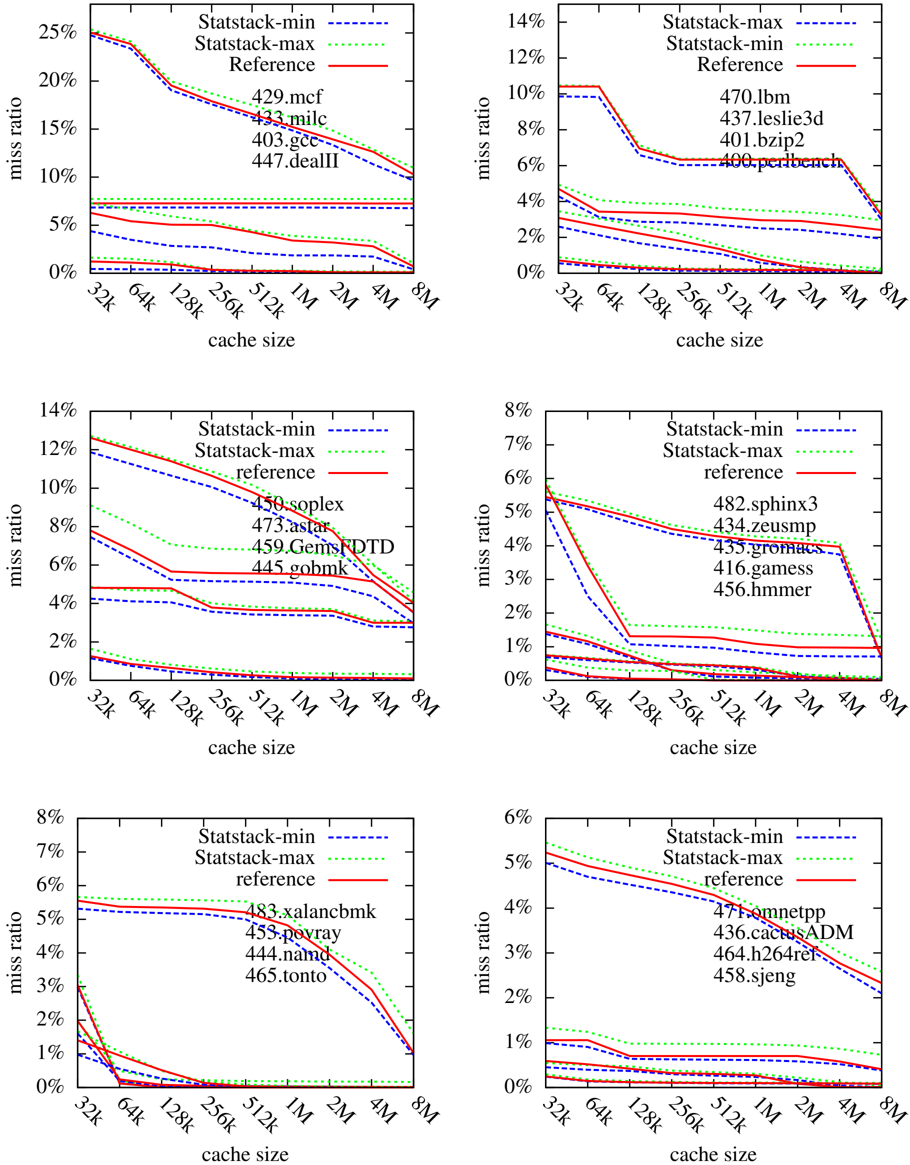


Figure 7: Miss ratios as a function of cache size. The graphs labeled StatStack-min and Statstack-max shows the minimum and maximum of 32 estimated miss ratios, based on RDSs collected with sampler parameters S_2. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

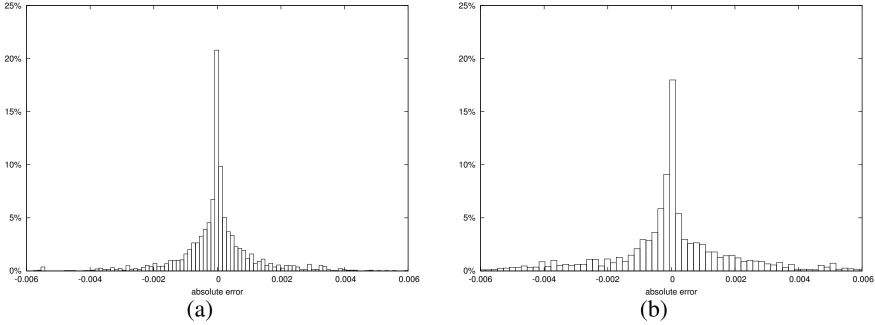


Figure 8: Distribution of absolute errors for RDSs collected using S_1 (a) and S_2 (b). 90% of the estimated miss ratios have a absolute error less than 0.2% in (a) and 0.4% in (b).

average miss ratio. To obtain a large enough data sample to accurately estimate the error distributions, the distributions shown in Figures 8(a) and 8(b) contain the error of all applications and all cache sizes for the miss ratios estimated using the sampler parameters S_1 and S_2, respectively.

The reason for using absolute error as opposed to a relative error is that, many of our benchmark applications have miss ratios that are very low. For example, 456.hmmer has a miss ratio of less than 0.5% for all cache sizes greater than 128KB. The performance gained by reducing a miss ratio this small, by say 50%, is negligible. Furthermore, small absolute errors tend to become large relative errors for small cache sizes, and the other way around for large caches. If we use relative errors, the errors for low miss ratios will be overrepresented and the errors for large miss ratios, for which there are potential performance gains, will be suppressed, we therefore use absolute errors.

As we can see in Figure 8, when RDSs are collected using S_1, 90% of the measured errors are less than 0.2%, for RDSs collected using S_2, 74% of the errors are less than 0.2% and 89% are less than 0.4%.

I.6.4 Cache Model Performance

The overall performance of cache models, like StatStack, has two parts, the performance of the data collection mechanism and the performance of the model itself.

The StatStack cache model uses the StatCache sampler to collect its input data. The sparseness of the collected data in conjunction with the use of hardware and operating system support makes the execution time overhead of the target application as low as 40 percent [3].

The execution time of the cache model, for the RDSs used for evaluation, is only a few seconds. The short execution time is due to the sparseness of the input data. Internally, the cache model stores the reuse distances in a histogram. The most time consuming operation for the cache model is to sort the data in the input RDS

into the histogram, but because of the sparseness of the RDSs this operation is still fast. When the histogram is built, the cache model requires only two passes over its buckets to compute the miss ratio. The number of buckets in the histogram is equal to the number of distinct reuse distances in the RDS.

I.7 Related Work

Since the introduction of stack distance, by Mattson [17] in the early 70s, it has earned plenty of attention from researchers in a quest to find efficient techniques to study cache locality. Mattson proposed a stack based algorithm to compute stack distances. For each memory reference, Mattson's stack based algorithm searches the stack for the accessed address. If the address is in the stack, the algorithm moves the address to the top of the stack, and otherwise, pushes it on the top of the stack. At any time, the number of unique addresses accessed since the last access to an address is the number of addresses above it in the stack. The stack maintains the history of uniquely accessed addresses. This history is sufficient to find the stack distance of memory reference.

To improve the performance of Mattson's algorithm Bennett and Kruskal [1] replace the stack with a m-ary tree. The m-ary tree allows for faster operations than the linked-list stack used by Mattson. To further improve performance, other types of trees have been proposed, for example AVL tress [19] and splay trees [24]. Kim et al. [15] propose an approximate algorithm, in which the stack is sliced up into disjoint ranges and only the address range is tracked. This allows them to use a hash table to search the stack. Other approximate algorithms have also been proposed by Ding and Zong [9] and Shen at al. [21].

All of the above algorithms use full address traces. However, the time it takes to collect and analyze large traces can be prohibitive. Trace sampling is a technique to reduce the size of the traces [13] [16] [26]. Trace sampling only collects and analyzes address traces for chosen sections of the application's execution. By applying statistical techniques, Trace sampling infers the overall miss ratio from that of the short sub traces. A drawback of Trace sampling is the large number of additional memory references needed to regain the access history lost between sub traces. The number of additional memory references required and techniques to reduce them have been investigated [10][12].

Shen at al. [21] proposed a probabilistic model to estimate stack distance distributions. The input to their model is a reuse distance distribution and the size of the memory footprint. StatStack differs from their model in two key aspects. First, StatStack does not require the size of the memory footprint to be known a priori. Secondly, StatStack takes an RDS as input and uses it to estimate the reuse distance distribution, while Shen at al. use the actual reuse distance distribution of all memory references. It appears as they could use an estimated reuse distance distribution, but this is not considered.

Tam at al. [25] present a hardware supported approach to collect address traces efficiently. Their focus is on online generation of miss ratio curves for the last level cache. By using hardware performance counters, they are able to trace only the memory references that miss in higher cache levels. Furthermore, they use trace sampling which in combination with their tracing technique makes their approach very efficient.

I.8 Conclusions

This paper presented StatStack, a new statistical cache model that models a fully associative cache with LRU replacement policy. The input to StatStack cache model is an RDS, which contains the reuse distances of a sparse set of randomly selected memory references. To obtain RDSs, StatStack uses the same sampling technique as StatCache, and therefore inherits StatCache low data collection overhead.

We evaluated the accuracy of StatStack using the SPEC CPU2006 benchmarks. The results show that StatStack accurately estimates miss ratios based on RDSs containing as few as $500,000$ and $100,000$ reuse distances, for which 90% of the estimated miss ratios have absolute errors less than 0.2% and 0.4%, respectively. Furthermore, the execution time of the StatStack cache model is less than a few second.

Acknowledgments

Håkan Zeffer, Erik Berg and Mats Nilsson took active part during the initial discussions about this research direction. David Black-Schaffer added insightful and valuable comments during the creation of this paper. This research was supported in part by the Swedish Research Council (VR) individual grant, UPMARC VR-Linnaeus grant and by the SSF CoReR-MP project.

1. References

[1] B. T. Bennett and V. J. Kruskal. LRU Stack Processing. *IBM Journal of Research and Development*, pages 353–357, 1975.

[2] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, Mar. 2004.

[3] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of ACM SIGMETRICS 2005*, Banff, Canada, June 2005.

[4] E. Berg, H. Zeffer, and E. Hagersten. A Statistical Multiprocessor Cache Model. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006)*, Austin, Texas, USA, Mar. 2006.

[5] K. Beyls and E. H. D'Hollander. Generating Cache Hints for Improved Program Efficiency. *J. Syst. Archit.*, 51(4):223–250, 2005.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351. IEEE Computer Society, 2005.

[7] A. S. Dhodapkar and J. E. Smith. Comparing Program Phase Detection Techniques. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[8] C. Ding and M. Orlovich. The Potential of Computation Regrouping for Improving Locality. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 13. IEEE Computer Society, 2004.

[9] C. Ding and Y. Zhong. Predicting Whole-Program Locality Through Reuse Distance Analysis. *SIGPLAN Not.*, 38(5):245–257, 2003.

[10] L. Eeckhout, S. Niar, and K. D. Bosschere. Optimal Sample Length for Efficient Cache Simulation. *J. Syst. Archit.*, 51(9):513–525, 2005.

[11] D. Eklov and E. Hagersten. Statstack: Efficient Modeling of LRU Caches. Technical Report 2009-019, Department of Information Technology, Uppsala University, July 2009.

[12] L. V. Ertvelde, F. Hellebaut, L. Eeckhout, and K. D. Bosschere. NSL-BLRL: Efficient Cache Warmup for Sampled Processor Simulation. In *ANSS '06: Proceedings of the 39th annual Symposium on Simulation*, pages 168–177, Washington, DC, USA, 2006. IEEE Computer Society.

[13] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Trans. Comput.*, 43(6):664–675, 1994.

[14] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[15] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing Stack Simulation for Highly-Associative Memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1):212–213, 1991.

[16] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. Comput.*, 37(11):1325–1336, 1988.

[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[18] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[19] F. Olken. Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies. Technical report, Lawrence Berkeley Lab, Aug. 1993.

[20] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Comput. Archit. News*, 37(3):371–382, 2009.

[21] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality Approximation Using Time. *SIGPLAN Not.*, 42(1):55–61, 2007.

[22] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 165–176, Boston, MA, USA, 2004. ACM.

[23] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, 2003.

[24] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 24–35, Santa Clara, California, United States, 1993. ACM.

[25] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 121–132, New York, NY, USA, 2009. ACM.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *SIGARCH Comput. Archit. News*, 31(2):84–97, 2003.

[27] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss Rate Prediction Across Program Inputs and Cache Configurations. *IEEE Transactions on Computers*, 56(3):328–343, 2007.

Appendix

Let l_t denote the cache line sized piece of memory accesses by memory reference x_t. We can now write Bennett and Kruskal's equations [1] as follows,

$$B_t(i) = \begin{cases} 1 & \text{if } l_i = l_j \text{ for } j = i+1, ..., t, \\ 0 & \text{otherwise} \end{cases}$$

$$P_t(x_k) = \begin{cases} \max\,(i \le t) & \text{such that } l_i = l_k, \\ -1 & \text{otherwise} \end{cases}$$

$$S(x_t) = \sum_{i=p+1}^{t-1} B_{t-1}(i), \text{ where, } p = P_{t-1}(x_t) \tag{1.1}$$

To show that (Eq. 1) follows from the above equations, we indentify the following equalities, $B_t(i) = 1(\vec{R}(x_i)+i > t)$ and $P_{t-1}(x_t) = t - R(x_t) - 1$. By substituting these equalities into (Eq. 1.1), (Eq. 1) follows.

Paper II

Fast Modeling of Shared Caches in Multicore Systems

David Eklov, David Black-Schaffer and Erik Hagersten
Uppsala University, Department of Information Technology
{david.eklov, david.black-schaffer, eh}@it.uu.se

Abstract

This work presents STATCC, a simple and efficient model for estimating the shared cache miss ratios of co-scheduled applications on architectures with a hierarchy of private and shared caches. STATCC leverages the StatStack cache model to estimate the co-scheduled applications' cache miss ratios from their individual memory reuse distance distributions, and a simple performance model that estimates their CPIs based on the shared cache miss ratios. These methods are combined into a system of equations that explicitly models the CPIs in terms of the shared miss ratios and can be solved to determine both. The result is a fast algorithm with a 2% error across the SPEC CPU2006 benchmark suite compared to a simulated in-order processor and a hierarchy of private and shared caches.

II.1 Introduction

The miss ratio[1] of independent co-scheduled applications on modern architectures is both important and difficult to predict. Depending on the architecture, in particular the degree of cache sharing, different combinations of applications can see dramatic variations in cache miss ratios when co-scheduled or none at all [3, 6]. However, modeling these effects has historically required costly simulation and/or analysis [6, 7]. Providing better insight into co-scheduled performance issues is essential to enable developers to better understand the behavior of their code in increasingly parallel environments. More interestingly, if these effects can be quickly and accurately modeled, it will open up many possibilities for improved run-time scheduling and placement of applications.

The primary difficulty in modeling co-scheduled applications arises from the interleaving of the address streams at the shared cache, and its impact on the execution of each thread. For example, memory accesses from one application can occur between memory accesses from a second application. This causes additional capacity pressure in the shared cache, which can increase the miss ratio for a second application. Such an increased miss ratio causes the CPI of the second application to increase, which reduces the rate at which it sends memory accesses to the shared cache, thereby changing the interleaving further. This coupling between the CPIs and miss ratios makes it challenging to predict the performance of the co-scheduled applications. Further, for such a model to be broadly applicable, it must be based on information gathered from the applications running independently.

StatCC leverages previous work that has shown that an application's miss ratio can be accurately predicted based on its reuse distance distribution for both random replacement (StatCache [4, 5]) and LRU replacement (StatStack [10]) caches. In that work, a reuse distance is defined as the number of memory accesses executed between two successive accesses to the same cache line. This reuse distance should not be confused with the stack distance, which is the number of *unique* cache-lines accessed between two successive accesses to the same-cache line, and which is much more costly to measure. Importantly, an application's reuse distance distribution can be accurately estimated based on a sparse sample of the application's reuse distances [4, 10]. Such sampled data can be gathered at runtime with an overhead of only 40% [5].

In this paper, we develop a method to predict the shared cache miss ratios of a set of co-scheduled applications based on their individual reuse distance distributions. First, we develop a method to predict the shared cache miss ratios of the co-schedule applications, given that we know their relative CPIs. The basis of this method is a transformation of the individual reuse distance distributions of the co-scheduled applications into the reuse distance distribution of the interleaved access stream presented to the shared cache. Once we have this combined distribution, we

[1] In this paper we use the term miss ratio to refer to the number of cache misses per memory accesses (MPA).

can use StatStack to predict the shared cache miss ratio. Of course this method is of little use since we do not know the CPIs of the co-scheduled applications to begin with. Under the assumption that we can predict an application's CPI based upon its miss ratio [1, 11, 12, 20], we can write the following simple equation system, shown here for two co-scheduled applications:

$$\begin{cases} cpi_1 = CPI\left(miss_ratio_1\left(cpi_1, cpi_2\right)\right) \\ cpi_2 = CPI\left(miss_ratio_2\left(cpi_1, cpi_2\right)\right). \end{cases}$$

This equation system can be readily solved for CPIs of the co-scheduled applications (cpi_1 and cpi_2) by a general purpose equation solver. We then use these CPIs to compute the applications' shared cache miss ratios ($miss_ratio_i(cpi_1, cpi_2)$).

To evaluate StatCC, we have run three experiments. The first evaluates the overall accuracy of StatCC and demonstrates an average error of 2% across 55 pairs of co-scheduled SPEC2006 applications. The second experiment evaluates the error introduced by using sparsely sampled reuse distance distributions. With a sampling rate of 10^{-3} (1 of every 1,000 memory accesses), the errors increase less than 2.5% for 95% of the simulations. Finally, the third experiment studies the sensitivity of the StatCC model to the accuracy of the CPI model. The results show that the StatCC suppresses these errors, such that a 10% error in the CPI model results in a change of the predicted CPI of only 1.0%.

Before delving into the details of StatCC, we will discuss related work (Section II.2) and review the principles behind StatStack (Section II.3). With that background, the design and implementation of StatCC can be explained (Section II.4). The subsequent discussion of the experimental setup (Section II.5.1) is proceeded by a description of the methodology and motivation for choosing benchmarks (Section II.5.2). This is particularly important as different benchmark combinations exhibit dramatically different co-scheduling effects, which makes it important to choose appropriate benchmarks to stress the model. Next we discuss how to compute the error for cache contention models (Section II.5.3). We argue that using relative miss ratio error can be misleading since it does not always correlate well with performance changes. The following experimental evaluation (Section II.5.4) discusses our results and the accuracy of StatCC.

II.2 Related Work

The most closely related cache contention models are those proposed by Chandra et al. [6] and Chen et al. [7]. Both of these models predict the miss ratio of co-scheduled applications, based on profiling information collected from the applications when running in isolation. A comparison of the reported errors suggests that StatCC's accuracy is on par with both these models, however, StatCC differs from these models on the following points:

(1) The input to StatCC is an easily collected sparse sample of reuse distance, while the other two models require full stack distance profiles, which are far more expensive to collect (StatCC's input data can be collected with a 40% overhead which is orders of magnitude less expensive than collecting full stack distance profiles). At a first glance, it would appear that the stack distance profiles used by Chandra et al. and Chen et al. could be collected by StatStack, thereby reducing the runtime overhead of their models. Unfortunately, their models rely on information not provided by StatStack[2].

(2) StatCC models the interaction between the co-scheduled applications with an equation system that explicitly encodes the coupling between the co-scheduled applications' miss ratios and CPIs. The other two cache contention models assume that the CPIs of the co-scheduled applications are the same as when they run alone. The model proposed by Chen et al. improves this by refining the initial CPI guess once, using a similar CPI model to the one used in this work.

In addition to these two models, Fedorova et al. [2] propose a cache contention model designed to guide an operating system scheduler. This model attempts to determine the miss rate (misses per cycle) that an application would experience under equal cache sharing, i.e., when the co-scheduled applications are all allocated an equal portion of the shared cache. Using hardware performance counters they measure the miss ratio of the co-scheduled applications. When enough applications have been scheduled together, they fit a linear model to the collected data that is then solved for the miss ratio under equal cache sharing.

There has been much interest in improving the efficiency of algorithms to compute stack distances for private caches. However, the input to these algorithms are either full [8, 16, 19, 18] or partial [13, 22, 21] address traces, which are costly to collect. Instead, StatCC uses StatStack [10] to estimate stack distances based cheaply collected on sparse reuse distance distributions.

Recently, Schuff et al. [17], showed how to efficient parallelize online computation of stack distances for both private and shared caches for multi threaded applications. Their starting point is a naive method that dynamically instruments the target application and synchronizes its threads on every memory accesses to generate a single address trace of all threads' memory accesses, from which they can compute stack distances. This approach is then enhanced properties of the stack distance and the assumed race-freeness of the target application to remove most of the synchronization overhead. It is unclear whether their method can be used for the mulit-programmed workloads that StatCC is targeting, but it is worth noting that their overhead is at least $10\times$ that of StatCC.

[2]The model proposed by Chandra et al. requires the stack distance distributions of memory accesses with the same reuse distance. However, StatStack approximates these distributions with a single expected stack distance [10]. The main motivation behind the model proposed by Chen et al. is to accurately model conflict misses, but since StatStack does not model conflict misses it is a poor fit for their model.

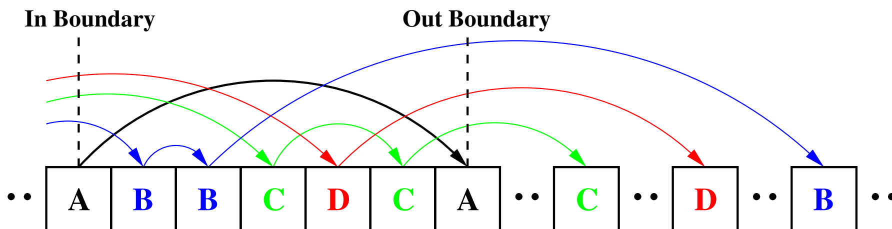


Figure 1: Reuse distances in a memory access stream. The arcs connect successive accesses to the same cache-line, and represents reuse of data. The stack distance of the second access to A is equal to the number of arcs that cross the "Out Boundary".

II.3 StatStack

Before discussing StatCC it is important to introduce the StatStack model upon which it is based. StatStack [10] is a statistical cache model that can predict an application's cache miss ratio for LRU caches of any given size. The input to StatStack is an application's *reuse distance* distribution. The reuse distance is the number of memory accesses executed between two successive memory accesses to the same cache-line. Based on an application's distribution of reuse distances, StatStack estimates the application's *stack distance* distribution. The stack distance is the number of *unique* cache-lines accessed between two successive memory accesses to the same cache-line. The stack distance can be directly used to determine if the 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 [15], by simply computing the fraction of memory accesses with a stack distances greater than the desired cache size.

The input to StatStack is an application's *reuse distance distribution*, which can be accurately estimated based on a sparse sample of the application's reuse distances [4][3]. Using reuse distances allows for very low overhead data collection. Berg et al. [5] implemented a reuse distance sampler that can collect an application's reuse distance distribution with an overhead of only 40%, which is far lower than that required to capture a full stack trace. In addition to efficient runtime data collection, the StatStack model itself can be evaluated in less than a second. This combination of efficient data capture and a fast model evaluation makes for a very flexible cache performance prediction tool.

To understand how StatStack works, consider the access sequence shown in Figure 1. Here the arcs connect subsequent accesses to the same cache-line, and

[3]StatStack uses a two-level sampling approach that first divides the application's memory accesses stream into windows and then selects a portion of the windows for further reuse distance sampling. This is done to address different behavior across application phases. More information can be found in [4] and [10].

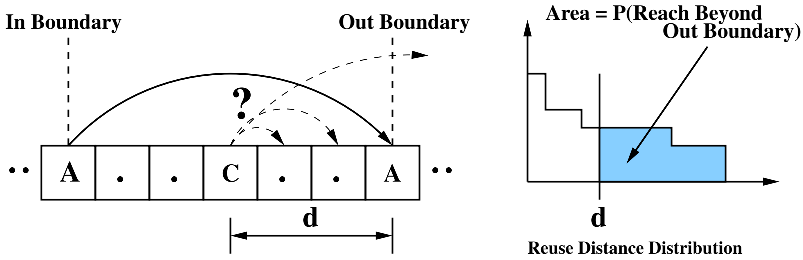


Figure 2: Using the reuse distance distribution to compute the probability that the arc from the memory access to cache-line C crosses the "Out Boundar". This is done by estimating the reuse distance of C from the proportion of reuse distances in the reuse distance distribution that are greater than d.

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 a total of five memory accesses executed between the two accesses to A, and a stack distances of three, since there are three *unique* cache-lines accessed between the two accesses to A. The question that StatStack answers is: How can the stack distance of the second access to A be computed from the reuse distances of the memory accesses executed between the two accesses to A?

In Figure 1 we see that, for each unique cache-line that is accessed between the two accesses to A, there is a sequence of connected arcs that cross both the "In Boundary" and the "Out Boundary". Now, since the stack distance of the second access to A is the number of unique cache-line accessed between the two accesses to A, it is equal to the number arcs that reach beyond the "Out Boundary".

The lengths of the arcs in Figure 1 are determined by the their respective reuse distances. If we know the reuse distance of all memory accesses executed between the two accesses to A, we can determine the stack distance to the second access to A by counting how many of the intervening accesses have outbound arcs that reach beyond the "Out Boundary". However, the input to StatStack is a sparse distribution of reuse distances, and therefore does not contain this information for every memory access. Instead, we use the reuse distance distribution to compute the probability that the memory accesses executed between the two accesses to A have reuse distances such that their arcs reach beyond the "Out Boundary".

Figure 2 shows how to compute the probability that the outbound arc of the memory access to cache-line C reaches beyond the "Out Boundary". In the figure, d denotes the memory access's distance to the "Out Boundary". When the reuse distance of the memory access is greater than d, its arc reaches beyond the "Out Boundary". StatStack makes the approximation that the probability of this is equal to the fraction of memory accesses in the application's reuse distance distribution with a reuse distance greater than d. Using the above method we can compute the probabilities that the arcs originating at the memory accesses between the two memory accesses to A, in Figure 1, reach beyond the "Out Boundary". By adding

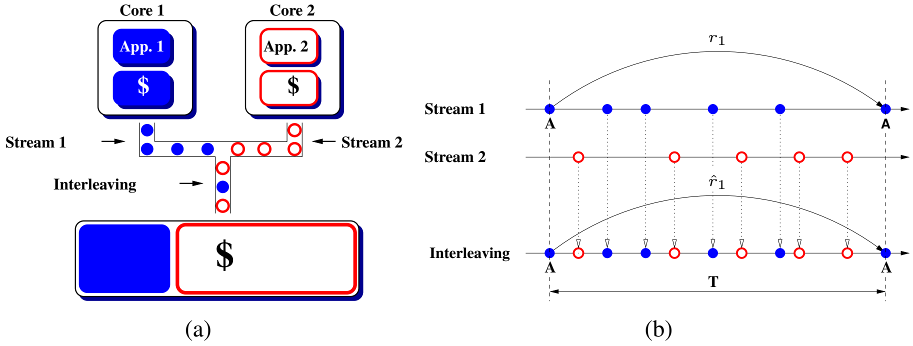


Figure 3: (a) The Architectural model used in this paper. (b) Example of an interleaving of memory access from two threads at the shared cache.

these probabilities we get the *expected stack distance* of the second memory access to A.

StatStack uses this approach to compute the expected stack distances of all memory accesses in the input reuse distance distribution, which produces an expected stack distance distribution. From this distribution StatStack can then estimate the miss ratio for any given cache size, C, as the fraction of expected stack distances in the expected stack distance distribution that have a stack distance greater than C. StatStack has been shown to very accurately model the cache behavior across the SPEC CPU 2006 benchmarks for a wide variety of cache sizes [10].

II.4 StatCC

In the previous section we discussed StatStack, and how it estimates an application's miss ratio given its reuse distance distribution. In this section we show how we can compute the miss ratios of a set of co-scheduled applications that share cache based on their individual reuse distance distributions. To simplify the discussion we will consider only two co-scheduled applications, but the extension to more applications is straightforward.

The architecture model targeted in this paper contains two processors cores, with both private and shared data caches (see Figure 3(a)). The two cores execute two threads with no data-sharing, thereby generating two independent memory access streams. At the shared cache these two independent memory access streams are interleaved due to their co-execution, and it is this interleaved access stream that determines the miss ratio in the shared cache.

We will present StatCC in three steps, summarized here:

1. First, we develop a scaling transformation that, given the two threads' individual reuse distance distributions and their CPIs, approximates the reuse distance distribution of the interleaved access stream.
2. Then, we use StatStack to estimate the shared level cache miss ratios of the two threads based on the approximated reuse distance distributions from the previous step. This gives us the shared level cache miss ratios of the two threads as a function of their CPIs ($miss_ratio_i(cpi_1, cpi_2)$).
3. Finally, we use the $miss_ratio_i(cpi_1, cpi_2)$ functions to setup an equation system whose solution gives us the shared level cache miss ratios of the two threads.

Step 1: Figure 3(b) shows an example of the memory access stream interleaving from two threads. The question to address is: What is the total number of memory accesses seen at the shared cache between two successive accesses to the same cache line from one thread, i.e., the reuse distance observed in the interleaved access stream? For example, in Figure 3(b), the access to cache-line A has a reuse distance of $r_1 = 4$ in the first thread's memory access stream. During the time interval T between the two accesses to A, the second thread executes $r_2 = 5$ memory accesses. Therefore, the reuse distance of cache-line A, as observed in the interleaved access stream at the shared cache, is $\hat{r_1} = 4 + 5 = 9$.

In general we do not know how many memory accesses the second thread generates relative to the first thread. To estimate this, we use the CPI of the first thread to determine the number of cycles taken for a given memory reuse distance, T, and then use the CPI of the second thread to determine how many memory access instructions it executes in the same time. This can be done if we know the relative CPIs of the two threads and each thread's proportion of memory instructions, mix_1 and mix_2, where,

$$mix_i = \frac{\#memory_accesses_i}{\#instructions_i}. \tag{1}$$

With this information we can compute the time interval, T, between the reuse of the cache-line A (reuse distance r_1) in the first thread's memory access stream, as follows:

$$T = \frac{r_1}{mix_1} \times cpi_1,$$

where $\frac{r_1}{mix_1}$ is the number of instructions executed by the first thread during T. Now, we can compute the number of memory accesses executed by the second thread during T as follows,

$$r_2 = \frac{T}{cpi_2} \times mix_2 = \frac{mix_2}{mix_1} \times \frac{cpi_1}{cpi_2} \times r_1.$$

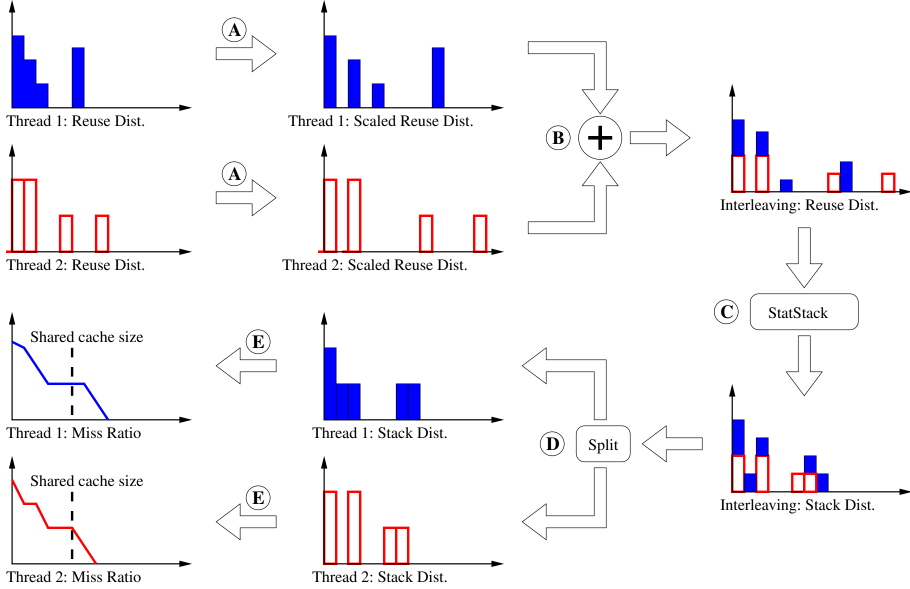


Figure 4: The data-flow of the $miss_ratio_i(cpi_1, cpi_2)$ functions. Eq. (2) is applied to all reuse distances in the reuse distance distributions of the two threads, which produces two scaled reuse distance distributions (A). These two scaled distributions are added together (B), resulting in an approximation of the reuse distance distribution of the interleaved accesses stream, which is then feed to StatStack which produces the stack distance distribution of the interleaved access stream at the shared caches (C). This distribution is then split into two separate stack distances distributions, one for each thread (D). Finally, the miss ratios of the two threads are computed from the two stack distances distributions (E).

Finally, we compute the reuse distance of the memory accesses to cache-line A as observed in the interleaved stream, $\hat{r}_1 = r_1 + r_2$. A similar expression for the second thread can be derived analogously, or simply by interchanging indices, which gives us,

$$\hat{r_1} = r_1 \left(1 + \frac{mix_2}{mix_1} \times \frac{cpi_1}{cpi_2}\right)$$
$$\hat{r_2} = r_2 \left(1 + \frac{mix_1}{mix_2} \times \frac{cpi_2}{cpi_1}\right). \tag{2}$$

By applying Eq. (2) to all reuse distances in the reuse distance distributions of the two threads and combining the result, we create an estimate of the reuse distance distribution observed at the interleaved stream. This transformation is effectively a scaling of the reuse distance distributions that incorporates the effects of the combined execution rates of the two threads.

Step 2: Figure 4 schematically shows the data-flow of the $miss_ratio_i(cpi_1, cpi_2)$ functions. First, we apply the scaling transformation of Eq. (2) to all reuse distances in both threads' individual reuse distance distributions (A). This results in a scaled reuse distance distribution for each thread. These are then added together (B) to get the reuse distance distribution of the interleaved access stream, and additional metadata is kept to indicate which reuse distances originated from which thread. Then, we feed this reuse distance distribution to StatStack (C) which gives us the stack distance distribution of the interleaved access stream. The next step is to use the metadata, recorded earlier, to split the stack distance distribution of the interleaved access stream into two separate stack distance distributions for each thread (D). Finally we compute the miss ratios of the two threads from their individual stack distance distributions (E).

Step 3: In this work we assume that we can model the CPI of an application as a function of its miss ratio. Many such models of varying complexity and accuracy have been proposed, both for, in-order processors [1], and out-of-order processors [11, 12, 20]. Some of the more complex out-of-order models require information other than the application's miss ratios, however this information only needs to be collected once, and, importantly, it is collected when the applications execute alone.

The absolute accuracy of the predicted CPIs is not critical for our approach to work, as StatCC only needs the ratios of the CPIs to calculate the miss ratio (see Eq. (2)). As long as the CPI model's predictions for the two applications are consistent, i.e., the CPI predictions of the two threads are off by the same factor, StatCC is insensitive to the accuracy of the CPI model. In Section II.5.4 we perform a sensitivity analysis to investigate how the accuracy of StatCC is effected by inconsistent errors in the CPI model. It shows that an error of 10%, as achieved by Fedorova et al. [1], is attenuated, and the error of StatCC increases by only 1%.

Armed with the $miss_ratio_i(cpi_1, cpi_2)$ functions, from Step 2, we can now write the following equation system, where $CPI(\cdot)$ represents the CPI model,

$$\begin{cases} cpi_1 = CPI\left(miss_ratio_1\left(cpi_1, cpi_2\right)\right) \\ cpi_2 = CPI\left(miss_ratio_2\left(cpi_1, cpi_2\right)\right). \end{cases} \tag{3}$$

This system can be solved for cpi_1 and cpi_2 with a general purpose equation system solver, for example a fixed-point method. Finally, to get the shared level cache miss ratios of the two threads, we plug the cpi_1 and cpi_2 that we get from solving Eq. (3) into the $miss_ratio_i(cpi_1, cpi_2)$ functions.

To summarize, The model presented here takes two reuse distance distributions and predicts their behavior when co-scheduled with a shared cache. It does this by explicitly modeling the interaction between the CPIs and the miss ratios, and using that model to predict both. Unlike previous models, this approach does not rely on aggregate data from a whole application execution, can function with easily

Table 1: *Memory system configuration*

	Size	Associativity	Latency
L1	32kB	8 way	1 cycle
L2	2MB	16 way	10 cycles
DRAM	4GB		130 cycles

collected input data, and explicitly models the CPI-miss ratio interaction. This leads to a simple, fast, and accurate model for predicting co-scheduled miss ratios and CPIs.

II.5 Evaluation

To evaluate STATCC, we have devised three experiments. The first evaluates the overall accuracy of STATCC across a variety of co-scheduled applications. The second examines the error introduced when using more efficiently gathered sparse reuse distance distributions. And the final experiment studies STATCC's sensitivity to errors in the CPI model.

II.5.1 Experimental Setup

For our experiments we use the Simics [14] multi-core system simulator. We run Simics with two in-order cores and a $base_cpi$ of 1.0, i.e., all non-memory instructions have a latency of one CPU cycle. For memory instructions, a memory system simulator is used to simulate cache and DRAM latencies. The memory simulator models a 32kB, 8-way private L1 data cache with a 64B line size, for each core, a shared 2MB, 16-way L2 data cache with a 64B line size, and a fixed-latency DRAM. The access latencies for the L1, L2, and DRAM are 1 cycle, 10 cycles, and 130 cycles, respectively. The memory simulator forces inclusion in the cache hierarchy by invalidating cache lines in the L1 cache that are evicted from the L2 cache. We do not simulate instruction caches. The configuration of the memory system simulator is summarized in Table 1.

The above experimental setup can be described with a simple and accurate CPI model, which allows us to evaluate STATCC independently from the CPI model. The CPI model for our experimental setup can be written as follows:

$$CPI_i(L2_mr_i) = base_cpi + mix_i \cdot (1 \cdot L1_hr_i + 10 \cdot L2_hr_i + 130 \cdot l2_mr_i)$$

where the suffix $_mr$ stands for miss ratio, the suffix $_hr$ stands for hit ratio, the subscript is the core, and mix is the instruction mix as defined in Eq. (1). The L2 hit ratio is then $L2_hr_i = 1 - L1_hr_i - L2_mr_i$.

Simulations are fast-forward for 5 billion instructions with the memory timing simulator turned off, and then run with the memory timing simulator enabled until one of the co-scheduled applications executes 100 million memory accesses. For the applications considered here, 100 million memory accesses take approximately 300-400 million instructions.

The StatCC model takes two independent reuse distributions from separate threads and predicts their CPIs and miss ratios when run together sharing a cache. To compare this model to the baseline simulation, we need to know which portions of the applications were co-scheduled during the simulation. We gathered this information during the simulation and used it to determine which of the independently collected reuse distributions should be evaluated together in the StatCC model. This allows us to fairly compare StatCC's results to the ground-truth simulation.

II.5.2 Benchmarks

The benchmarks used for this evaluation consist of 55 pairs of applications from SPEC CPU2006 that were explicitly selected to stress the StatCC model. To determine which applications will be sensitive to the resources sharing evaluated by StatCC we first used StatStack to generate miss ratio curves for all 29 SPEC CPU2006 applications. Based on this information we made a rough estimate of how sensitive the applications' miss ratios were to cache sharing. This was done by computing the increase of the applications' miss ratios when their cache size is reduced from 2MB to 1MB. We then selected the seven applications with the largest miss ratio increases (Figure 5(a)-5(c)). For example, 471.omnetpp is the application with the largest miss ratio increase. Its miss ratio increases almost 5% when its cache allocation is reduced from 2MB to 1MB.

This approach gave us the seven applications that present the greatest challenge for the cache sharing model as they exhibit the largest change in miss ratio when they are forced to share the last level cache. For completeness, we added another three applications to our set of benchmarks for which the difference in miss ratio at cache sizes of 1MB and 2MB are small. The miss ratio curves for these three applications are shown in Figure 5(d). From these 10 applications we then generated all 55 possible pairings and used these to evaluate the StatCC model. This selection process allowed us to focus on the applications for which cache sharing effects are most significant.

To validate this selection of benchmarks, Figure 6 shows, for each of the 55 pairs of applications, the relative increase in miss ratio when the application are co-scheduled with respect to when they run alone. As expected, the applications with little change in miss ratio between 1MB and 2MB, 456.hmmer, 458.sjeng and 470.lbm, experience a relatively small increase in miss ratio. For 471.omnetpp, which shows a large increase in miss ratio going from 2MB to 1MB, the miss ratio increases up to about 1000%, depending on which application it is co-scheduled

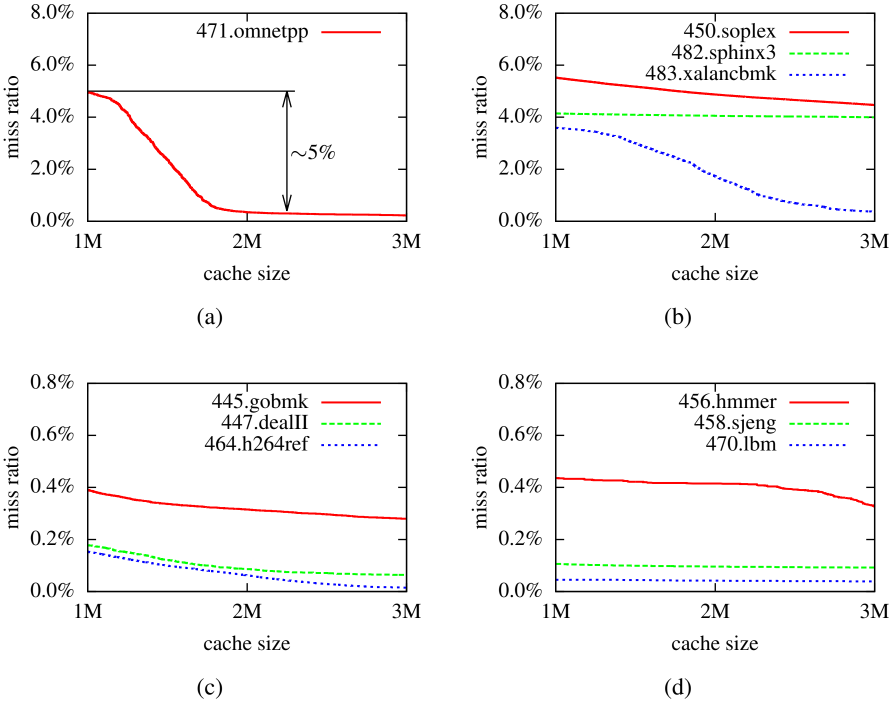


Figure 5: Miss ratio curves for the selected benchmark applications. The applications that exhibit the largest change in miss ratio when their cache size is reduced from 2MB to 1MB are shown in 5(a), 5(b), and 5(c). Figure 5(d) shows the miss ratio curves for the selected benchmarks with little change in miss ratio between 2MB and 1MB.

with. Most importantly, the overall miss ratio increase for the 55 co-schedules is much larger than that for randomly selected co-schedules, thereby making them interesting to study because their resulting performance is harder to predict.

II.5.3 Error Computation

StatCC is designed to predict the miss ratio of co-scheduled applications. However, when evaluating the accuracy of a cache contention model that predicts miss ratios, we argue, that, when there exists a reasonable CPI model for the architecture in question, CPI should be used when computing the error of the prediction. The motivation is that the relative miss ratio increase of an application when it is co-scheduled with other applications compared to running in isolation, is not tightly correlated with the performance change, or CPI increase. Figure 7 shows a scatter plot relating relative miss ratio increase to relative CPI increase for the 55 co-scheduled applications. As we can see, the correlation between the two is not very strong. When the miss ratio of an application with a low miss ratio, say 0.1%,

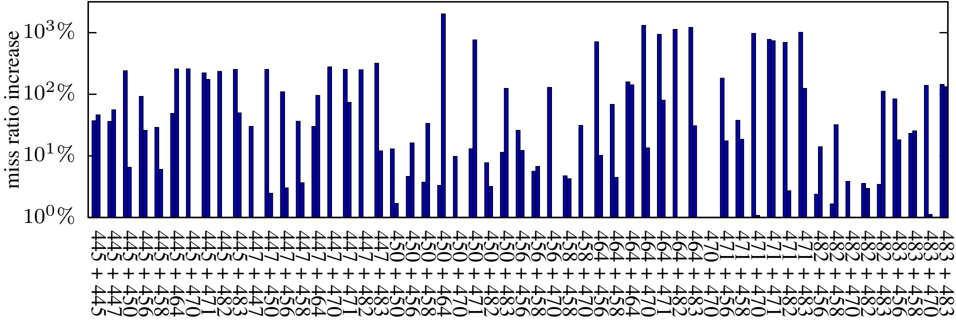


Figure 6: Increase in miss ratio when the applications are co-scheduled compared to when they run alone. Each pair of bars indicate the increase for a pair of co-scheduled applications. The left bar indicates the increase for the first, and the right bar indicates the increase for the second application.

is increased to 0.2% when it is co-scheduled with other applications, the relative miss ratio increase is 100%. However, since the absolute increase of its miss ratio is only 0.1% and the miss ratio itself is small, the increase in CPI due to the miss ratio increase is negligible compared to other factors that effect performance. We therefore use the following equation to compute STATCC errors,

$$error = \frac{CPI\,(mr) - CPI\,(\widehat{mr})}{CPI\,(mr)} \tag{4}$$

where, mr and $\widehat{mr}$ are the reference and the predicted miss ratios, respectively. Here, the miss ratio is filtered through the CPI model, and the large relative miss ratio errors for applications with small miss ratios are attenuated. It is only when the error of the miss ratio prediction would lead to a largely miss-predicted CPI that the error is large.

Figure 7(b) shows the correlation between the relative increase in CPI and the absolute increase in miss ratio, for the same scenario as Figure 7(a). In Figure 7(b), there is a much stronger correlation than in Figure 7(a). Therefore, when it is necessary to report miss ratio errors, for example when no CPI model is available, the absolute miss ratio should be used rather than the relative.

II.5.4 Results

Experiment 1: In this experiment we evaluate the accuracy of STATCC by comparing the predicted results to those obtained from the reference simulation, using all 55 application pairs. For each application pair, we measure the reuse distances of all memory accesses for both applications, and feed them to STATCC. That is,

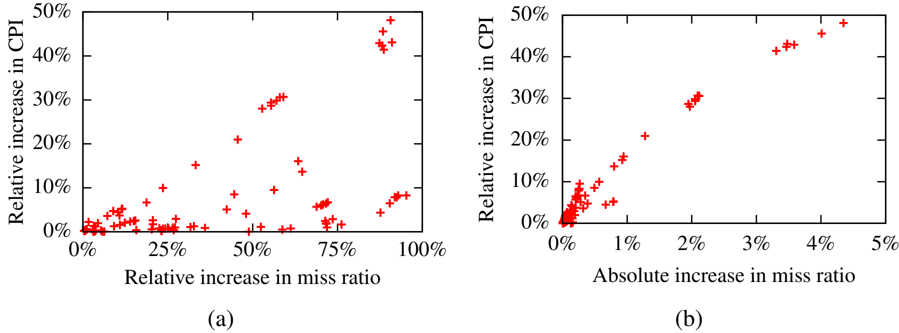


Figure 7: Scatter plot relating (a) the relative miss ratio increase, and (b) the absolute miss ratio increase, to the relative increase in CPI, for the co-scheduled applications when running alone compared to when they are co-scheduled. For each of the 55 pairs of co-schedule applications, there are two data points shown, one for each application.

we do not sample the reuse distributions. This allows us to evaluate the accuracy of StatCC without the effects of any errors introduced by sampling.

Table 2 shows the average, median and standard deviation of StatCC's prediction errors, computed using Eq. (4), for each benchmark application when they are co-scheduled with all other applications including itself. The bottom row shows the overall average, median and standard deviation of the prediction errors for both applications in all 55, which are 1.9%, 0.4% and 3.8% respectively. For the 55 pairs examined, 90% of the prediction errors are less than 5%. These statistics show that StatCC accurately predicts the co-scheduled behavior for seven of the most challenging applications in the SPEC CPU2006 benchmark. The two applications with the largest average errors are 471.omnetpp and 483.xalancbmk. They are the two applications whose miss ratios have the greatest potential to change when experiencing capacity pressure in the shared cache (see Section II.5.2), which makes their miss ratios hard to predict. We will further examine these two applications later in this section.

Figure 8 shows the modeled and simulated results for each application in the 55 pairs of co-schedules. Figure 8(a) shows the reference vs. predicted miss ratios, and Figure 8(b) shows the reference vs. predicted CPI. Ideally, all points should be on a 45° line passing through the origin. When a point is above the 45° line the model overestimates, and when the point is below the 45° the model underestimates. In Figure 8(a), there is a cluster of points between 0.6% and 3%, whose relative miss ratio prediction errors are larger then average. These points correspond to miss ratios of co-schedules containing 471.omnetpp and 483.xalancbmk, which we have identified as being the hardest applications to predict. However, Figure 8(b) shows that when the predicted miss ratios are used to estimate performance, the errors of the performance predictions are not as significant.

Table 2: *Average, median and standard deviation of* StatCC*'s prediction errors, computed using Eq. (4), for each benchmark application when they are co-scheduled with all other applications including themselves.*

	Average	Median	Std. Dev.
445.gobmk	0.7%	0.1%	1.4%
447.dealII	0.2%	0.1%	0.1%
450.soplex	0.5%	0.4%	0.3%
456.hmmer	0.5%	0.1%	1.2%
458.sjeng	0.1%	0.1%	0.1%
464.h264ref	1.7%	1.3%	1.4%
470.lbm	0.8%	0.9%	0.5%
471.omnetpp	10.3%	8.3%	6.3%
482.sphinx3	1.4%	1.5%	0.3%
483.xalancbmk	3.3%	0.9%	3.5%
Overall	1.9%	0.4%	3.8%

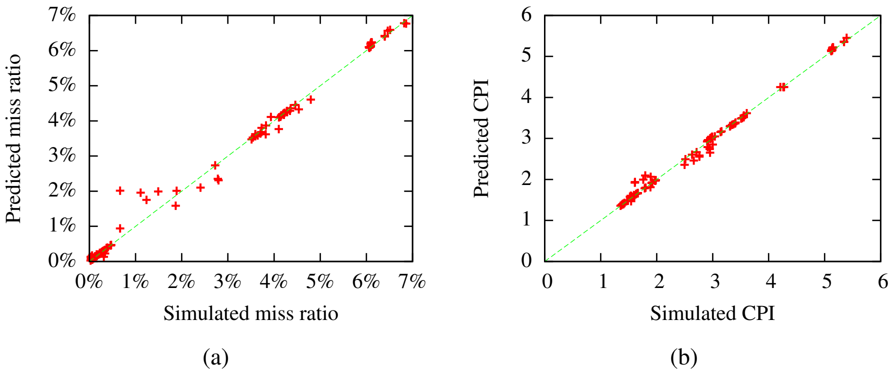
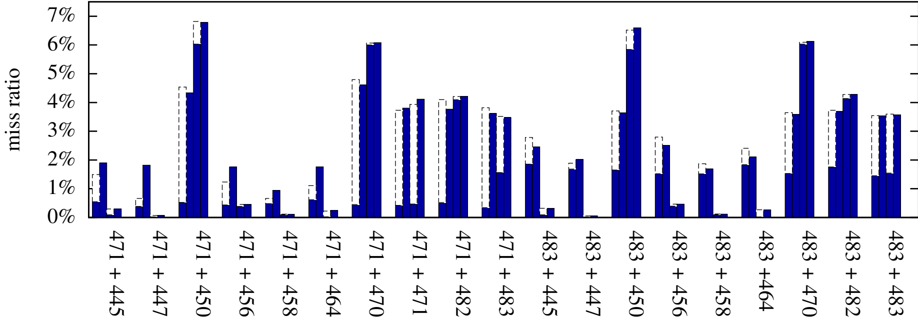
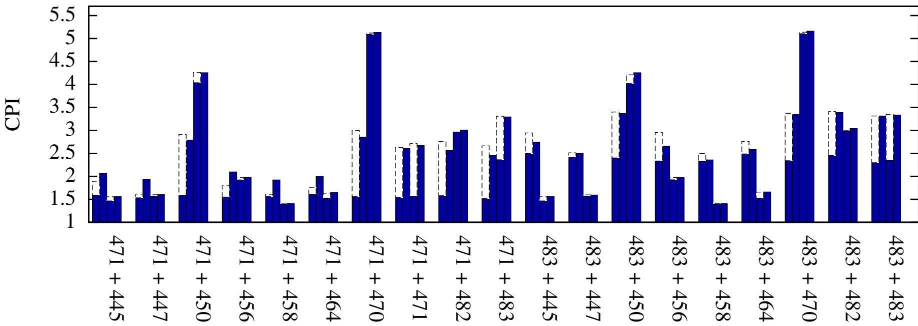


Figure 8: Scatter plot showing, (a) predicted vs. simulated co-scheduled miss ratio and (b) predicted vs. simulated co-scheduled CPI. For each pair of the 55 co-schedules, there are two data points shown, on for each application.

Figure 9(a) shows the reference and the predicted co-scheduled miss ratios for all co-schedules contaning 471.omnetpp and
483.xalancbmk, which are the hardest to predict, and whose average prediction errors are the largest. For each co-schedule there are four bars, with two for each application. The first of each pair is the simulation results and the second is the StatCC prediction. For example, consider the application pair of 471.omnetpp and 483.xalancbmk, which is in the middle of the figure. The leftmost bar indicates the miss ratio of 471.omnetpp when run alone (bottom, dark red portion) and



(a)



(b)

Figure 9: Prediction and simulation results for the two most difficult applications, 471.omnetpp and 483.xalancbmk, for which StatCC's prediction errors are the largest. There are four bars for each application pair. The first and third bars are the simulation results for the first and second applications, respectively. These bars consist of the miss ratio, in Figure 9(a), and CPI, in Figure 9(b), when run individually (bottom, black) and the delta when co-scheduled (top, white). The second and fourth bars are the predicted co-scheduled results from StatCC for the first and second applications, respectively. To evaluate the accuracy of StatCC one should compare the co-scheduled simulation results (first and third bars), including the increases due to co-scheduling (white), to the StatCC results in the second and fourth bars, respectively. The change due to co-scheduling (white) is an indication of how sensitive the application is to other applications running at the same time.

the total height (including the top light green portion) indicates the miss ratio of 471.omnetpp when co-scheduled with 483.xalancbmk. The next bar immediately to its right is the predicted miss ratio from StatCC. Figure 9(b) shows similar results for the predicted CPIs as computed from StatCC's miss ratios.

As Figure 9(a) shows, the largest miss ratio prediction errors, for both 471.omnetpp and 483.xalancbmk, are measured when they are co-scheduled together with applications with low miss ratios. This is likely due StatStack not being able to accurately estimate the miss ratios for applications with low miss ratios [10]. This is especially true when their miss ratio curves have knees, as 471.omnetpp has at a cache size of about 1.7MB (see Figure 5(a)). For example, consider the co-schedule 471.omnetpp and 447.dealII. 447.dealII has a low miss ratios for cache sizes between 1MB and 2MB (see Figure 5(c). Applications with low miss ratios typically only occupy a small portion of the cache when co-scheduled with other applications. This results in 471.omnetpp's cache occupancy being close to the knee in its miss ratio curve, which makes it hard for StatStack to estimate its miss ratio.

Experiment 2: In this experiment, we evaluate STATCC's sensitivity to using sparsely sampled reuse distance distributions. Again, we use the 55 pairs of applications, but this time we only collect reuse distance information from randomly sampled memory accesses. Each pair of co-scheduled applications is sampled 32 times, which gives us $32 \cdot 2 \cdot 55 = 3520$ sparse reuse distance distributions. We then feed these distributions to STATCC and compute the errors for the resulting predictions. By examining the distributions of these errors, we can quantify STATCC's sensitivity to sampling.

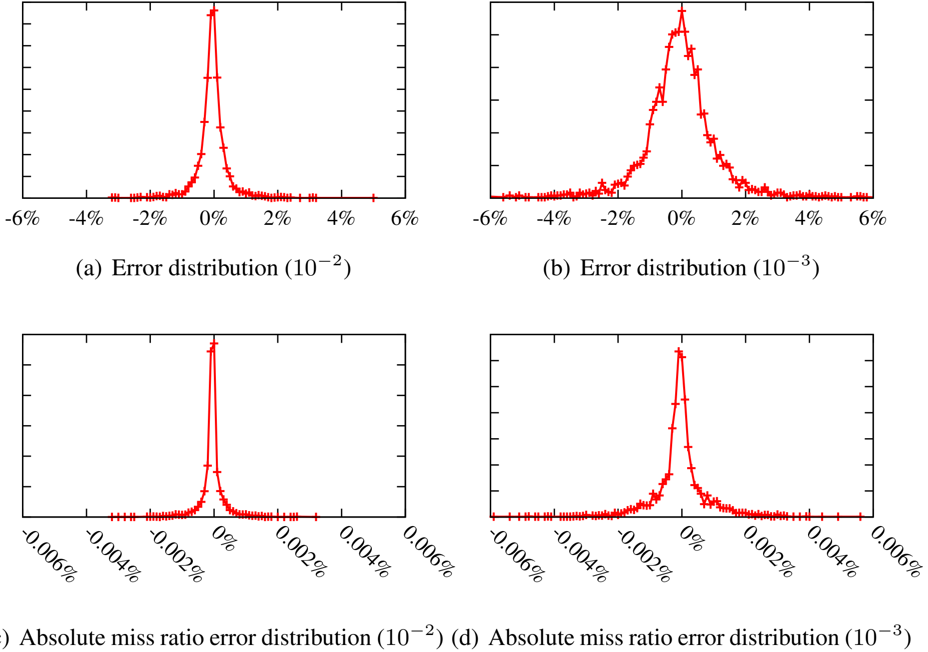
Figures 10(a) and 10(b) show the distribution of the errors, computed with Eq. (4) for sample rates of 10^{-2} (1 of every 100 memory accesses) and 10^{-3} (1 of every 1,000 memory accesses), respectively, normalized to their means. For a sample rate of 10^{-2}, 95% of the errors are within $\pm 1.0\%$, and for a sample rate of 10^{-3}, 95% of the errors are within $\pm 2.5\%$. This means that, for a sample rate as low as 10^{-3}, we can expect the errors to increase no more than 2.5% (on top of the errors reported in Experiment 1).

Similarly, Figures 10(c) and 10(d) show the distribution of the absolute miss ratio errors, for sample rates of 10^{-2} and 10^{-3}, respectively. For a sample rate of 10^{-2}, 97% of the errors are within $\pm 0.1\%$, and for a sample rate of 10^{-3}, 97% of the errors are within $\pm 0.25\%$.

Experiment 3: In this experiment, we evaluate STATCC's sensitivity to errors in the CPI model. To do so, we introduce an error in the CPI model by multiplying it by a factor α, in the following manner:

$$\begin{cases} cpi_1 = \alpha_1 \cdot CPI_1\left(mratio_1\left(cpi_1, cpi_2\right)\right) \\ cpi_2 = \alpha_2 \cdot CPI_2\left(mratio_2\left(cpi_1, cpi_2\right)\right). \end{cases}$$

By solving this equation system for cpi_1 and cpi_2 and then plugging them into the $miss_ratio_i(cpi_1, cpi_2)$ functions we obtain miss ratio predictions. By analyzing the error of these miss ratio predictions as a function of α_1 and α_2, we can quantify the miss ratio predictions sensitivity to errors in the CPI model.



(a) Error distribution (10^{-2}) (b) Error distribution (10^{-3})

(c) Absolute miss ratio error distribution (10^{-2}) (d) Absolute miss ratio error distribution (10^{-3})

Figure 10: Distribution of the additional errors (on top of the errors reported in Experiment 1) introduced by sparse sampling. 10(a) and 10(b), shows the error distributions, computed using Eq. (4), for sample rates of 10^{-2} and 10^{-3} respectively. 10(c) and 10(d), shows the distribution of absolute miss ratio errors, for sample rates of 10^{-2} and 10^{-3} respectively.

Table 3 shows the average increase in StatCC's absolute miss ratio error for relative errors in the CPI model of up to 10%, as reported by Fedorova et al. [1]. The largest increase is observed for $\alpha_1 = 0.9$ and $\alpha_2 = 1.1$ and is 0.08 percentage points. This means that if the CPI model were to over estimate the CPI by 10% for one of the co-scheduled applications, and at the same time under estimate the CPI of the other application by 10%, the error of StatCC's miss ratio prediction would increase by 0.08%. However, this absolute error increase is hard to relate to the 10% relative error introduced to the CPI model. Therefore, we use the method to compute miss ratio errors discussed in Section II.5.3. Table 3 shows these errors, where the largest increase in relative CPI error is 1.0%, for $\alpha_1 = 0.9$ and $\alpha_2 = 1.1$, this corresponds to an average increase from 1.9% to 2.9%.

II.5.5 Overhead and Scalability

In this evaluation we have demonstrated the effectiveness of StatCC for two co-scheduled applications. The equation system (Eq. (3)) that describes how co-scheduled applications contend for a shared cache can easily be extended to any

Table 3: *The increase in error due to a 10% error in the CPI model. The right side, shows the average increase of the absolute miss ratio error, and the left side, shows the average increase of the relative CPI error.*

Absolute Miss Ratio Error

α_1 \ α_2	0.9	1.0	1.1
0.9	-	0.04	0.08
1.0	0.02	-	0.03
1.1	0.06	0.05	-

Relative CPI Error

α_1 \ α_2	0.9	1.0	1.1
0.9	-	0.4	1.0
1.0	0.1	-	0.1
1.1	0.6	0.4	-

number of threads. We have successfully solved the equation system for several combinations of four co-scheduled synthetic applications, which suggests that the equation system is solvable for more than two co-scheduled applications.

We have used a simple unoptimized fixed-point equation system solver implemented in Python. The time it takes to solve the equation system (Eq. (3)) is on the order of a few seconds. However, this time can easily be reduced by an order of magnitude by implementing an optimized solver in a compiled language.

II.6 Summary

This work has presented StatCC, a statistical cache contention model that uses sparsely collected runtime information to model the cache contention of co-scheduled applications in terms of their miss ratios. The input to StatCC is a reuse distance distribution that can be collected with an overhead of only 40% [5] and independently of the co-scheduled execution. StatCC explicitly models the interaction between miss ratio and CPI, and leverages the StatStack cache model and a simple CPI model to efficiently and accurately predict the miss ratio of co-scheduled applications.

StatCC's one-time, low-overhead data collection and fast model evaluation make it well suited for many uses. For example, program cache optimization targeting a shared level cache, long-term cache aware job scheduling and processor design space exploration can all benefit significantly from fast data collection.

In a multiprogramed environment, StatCC can be used to predict the effectiveness of cache optimization targeting the shared level cache. Here, the low-overhead data collection significantly reduces the turnaround of the profile-optimize-test cycle. Long-term job scheduling is another interesting use case for StatCC. For workloads where new programs are frequently submitted to the job queue and only resubmitted a few times, the data collection overhead 40% might be hard to justify. However, for workloads where programs are frequently resubmitted, for example in HPC cluster, the cost of the data collection can easily be amortized.

For design space exploration, both the low-overhead data collection and the fast evaluation time of StatCC allows for larger search spaces.

To further reduce the data collection overhead, we are investigating the use of hardware debug registers (instead of page protection) to implement the watchpoint mechanism used by the StatCache sampler. This could potentially reduce the data collection overhead by an order of magnitude. Such a reduction would enable more uses of StatCC, such as guiding cache contention aware kernel level thread scheduling.

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research trough CoDeR-MP, and by the Swedish Research Council through UPMARC, the Uppsala Programming for Multicore Architectures Research Center. We would especially like to thank the anonymous reviewers for their helpful suggestions.

1. References

[1] A. Fedorova, M. Seltzer and M. D. Smith. A Non-Work-Conserving Operating System Scheduler for SMT Processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-33*, Boston, MA, USA, June 2006.

[2] A. Fedorova, M. Seltzer and M. D. Smith. Cache-fair Thread Scheduling for Multicore Processors. Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University, Oct. 2006.

[3] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum. Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 2005. USENIX Association.

[4] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, Mar. 2004.

[5] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of ACM SIGMETRICS 2005*, Banff, Canada, June 2005.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[7] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *IEEE International Symposium on High Performance Computer Architecture (HPCA 2009)*, pages 329–340, Raleigh, North Carolina, USA, Feb. 2009.

[8] C. Ding and Y. Zhong. Predicting Whole-Program Locality Through Reuse Distance Analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–257, New York, NY, USA, 2003. ACM.

[9] D. Eklov, D. Black-Schaffer, and E. Hagersten. StatCC: A Statistical Cache Contention Model. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 551–552, Vienna, Austria, Sept. 2010.

[10] D. Eklov and E. Hagersten. StatStack: Efficient Modeling of LRU caches. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2010)*, White Plains, NY, USA, Mar. 2010.

[11] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar out-of-order Processors. *ACM Trans. Comput. Syst.*, 27(2):1–37, 2009.

[12] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. *SIGARCH Comput. Archit. News*, 32(2):338, 2004.

[13] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.*, 43(6):664–675, 1994.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, 2002.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[16] F. Olken. Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies. Technical Report LBL-12370, Lawrence Berkeley Lab Berkeley, May 1981.

[17] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, Sept. 2010.

[18] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–61, New York, NY, USA, 2007. ACM.

[19] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. *SIGMETRICS Perform. Eval. Rev.*, 21(1):24–35, 1993.

[20] T. M. Taha and S. Wills. An Instruction Throughput Model of Superscalar Processors. *IEEE Trans. Comput.*, 57(3):389–403, 2008.

[21] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, pages 84–95, 2003.

[22] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 91–100, New York, NY, USA, 2008. ACM.

Paper III

Cache Pirating: Measuring The Curse of the Shared Cache

David Eklov, Nikos Nikoleris,
David Black-Schaffer and Erik Hagersten
Uppsala University, Department of Information Technology
{david.eklov, nikos.nikoleris, david.black-schaffer, eh}@it.uu.se

Abstract

We present a low-overhead method for accurately measuring application performance (CPI) and off-chip bandwidth (GB/s) as a function of its the available shared cache capacity, on real hardware, with no modifications to the application or operating system. We accomplish this by co-running a Pirate application that “steals" cache space with the Target application. By adjusting how much space the Pirate steals during the Target’s execution, and using hardware performance counters to record the Target’s performance, we can accurately and efficiently capture performance data for the Target application as a function of its available shared cache. At the same time we use performance counters to monitor the Pirate to ensure that it is successfully stealing the desired amount of cache.

To evaluate this approach, we show that 1) the cache available to the Target behaves as expected, 2) the Pirate steals the desired amount of cache, and 3) the Pirate does not impact the Target’s performance. As a result, we are able to accurately measure the Target’s performance while stealing between 0MB and an average of 6.1MB of the 8MB of cache on our Nehalem based test system with an average measurement overhead of only 5.5%.

III.1 Introduction

The increasing core count of modern processors has not been met with a corresponding increase in off-chip bandwidth [2]. Instead, modern CMPs have come to rely on large on-chip caches to reduce off-chip bandwidth demand. As these resources are typically shared across multiple cores, the amount of each resource available to an individual core may vary with workload. As application performance is believed to be strongly influenced by the available cache and bandwidth [2, 11, 17], understanding performance as a function of the available shared memory system resources is increasingly important for performance analysis.

Our long term goal is to understand the impact of shared memory system resources on the performance and scalability of multithreaded data parallel programs. For such programs, the similar execution of each thread leads to an equal distribution of memory resources [7, 10]. This suggests that if we knew how a single thread's performance and off-chip bandwidth demand change as a function of its shared cache space allocation, we could determine the performance and bandwidth demand as a function of the number of threads, and therefore predict how the application will scale. As a first step towards this goal, this paper presents a very accurate, low-overhead technique for measuring performance and bandwidth as a function how much cache space is available to the application.

To be successful, this method must exhibit three key characteristics: accuracy, efficiency, and simplicity. Accuracy means that we need to reveal the idiosyncrasies of the real hardware. As we will show later, details such as hardware prefetchers and non-standard cache replacement policies have significant impacts on performance and must be included. Efficiency translates into speed, and is essential to enable us to evaluate real applications with real workloads. If the overhead of collecting the performance data is too high, then the technique becomes impractical. And finally, simplicity is important to enable broad use of the technique. If we rely on non-standard hardware extensions, operating system modifications, or application changes to collect our data, we not only run the risk of unknowingly impacting the results, but we significantly raise the bar for adoption. The technique presented here achieves accuracy through the use of real hardware, efficiency through the use of performance counters, and simplicity by requiring no operating system or application modifications.

Our approach is simply to measure the performance of a Target application while it is co-run with a Pirate application that intentionally “steals" space in the shared cache. To ensure accurate measurements we need to design the Pirate to steal cache without making excessive use of other shared resources as this can adversely impact the performance of the Target application. Such a Pirate will allow us to evaluate the Target's performance as a function of cache size by varying the amount of space the Pirate steals and measuring the Target's resulting performance. Central to the success of this method is our ability to easily monitor the Pirate while it is executing to determine if it is stealing as much cache as we expect. With this ca-

pability we can assure the accuracy of our measurements regardless of the Target application's behavior.

The result is the *Cache Pirating* method which allows us to accurately, and efficiently, evaluate the performance of an application as a function of its available shared cache. We demonstrate that we can achieve an average absolute error of 0.2% (compared to reference fetch ratios) with an average measurement overhead of only 5.5%. Furthermore, this is accomplished running on standard hardware with no modifications to the Target application or the operating system.

III.2 Background and Motivation

III.2.1 Motivation

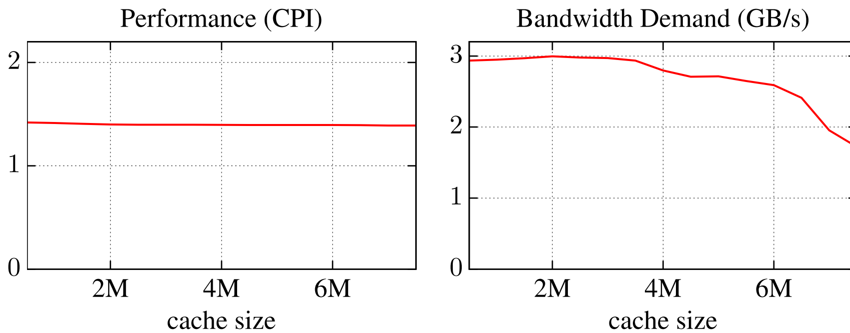


Figure 1: Sample application performance and bandwidth demand as a function of cache size.

The performance and bandwidth requirement of an application as a function of cache size are fundamental properties of an application. These properties are critical for analyzing performance and scaling. Consider, for example, the performance data shown in Figure 1. Here we see that the application's performance does not change as its available cache capacity changes. This implies that running more instances (or threads) of this application will linearly increase the performance. Although we expect each thread to receive less cache as the number of threads is increased, the data clearly indicates that this will not result in a decreased per-thread performance.

However, if we examine the bandwidth requirement for the application, the situation changes. From the bandwidth data we clearly see that reducing the available cache space results in an increase in the bandwidth requirement. That is, for this application there is a tradeoff between the available cache space and required bandwidth. The performance curve as a function of cache size is flat because as we reduce the amount of cache space, the application makes up for it by increasing its off-chip bandwidth consumption.

From this information we can analyze the expected performance scaling as we increase the number of threads or application instances. With each thread consuming an equal portion of the shared cache[1], we can determine the expected throughput by determining the per-thread performance based on the per-thread cache allocation. Similarly, we can determine the total required bandwidth from the per-thread bandwidth data. If the total bandwidth requirement is less than the system's bandwidth, then we can expect to achieve the predicted performance. However, if the total required bandwidth is greater than the system's bandwidth, we can conclude that with this number of threads the application is bandwidth-limited, and will not achieve the expected throughput.

To predict performance on real hardware we need to be able to collect realistic and accurate data. This implies that we need to be able to collect data fast enough to measure the performance of applications processing real datasets and that the data we collect needs to accurately reflect the idiosyncrasies of the target hardware. The focus of this paper is on how to collect this data accurately, efficiently, and simply.

III.2.2 Simulation

The standard approach for collecting detailed performance data is through simulation. Simulator precision is limited only by the detail with which they model the targeted hardware. Unfortunately this accuracy comes at the cost of performance. Detailed simulations typically take orders of magnitude longer than native execution. To address this, a substantial amount of work has been done to improve simulation performance [12, 16, 21]. In particular, accuracy and detail can be traded for performance by the use of analytical performance models [18, 22].

However, for simulation to accurately report performance, the models used to represent the targeted hardware must be sufficiently detailed to reliably capture the true behavior of the system. To avoid both this difficulty and the overhead incurred in simulation, our method measures performance metrics of the target application while it is executing on real hardware using performance counters, and therefore captures the true behavior of the hardware with very low overhead.

III.2.3 Miss Ratio Curves

A common approach to understand how the amount of available cache space impacts an application's performance is to analyze its *miss ratio curve* (MRC). MRCs capture an application's cache miss ratio as a function of the cache space available to the applications. MRCs can be generated fairly cheaply [9, 6, 8], and have

[1]The motivation of this work is the study of the performance and scalability of data parallel programs whose threads executed the same code but working on different parts of the application's data. For such applications, the threads equal demand for resources results in a equal distribution of cache capacity across the threads [7, 10].

been used in contexts such as cache partitioning [14], off-chip bandwidth partitioning [11] and cache contention modeling [7].

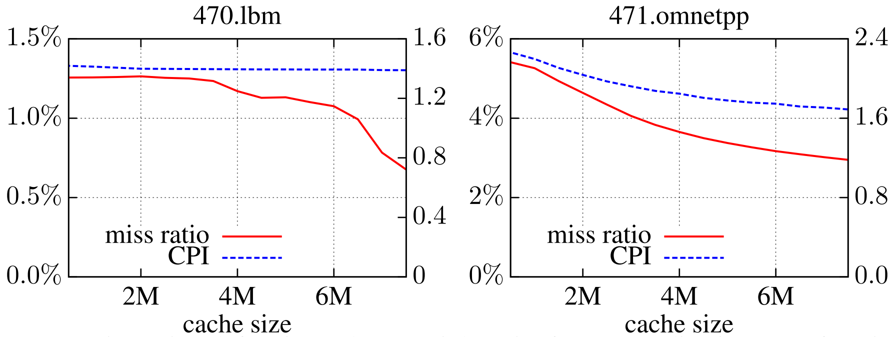


Figure 2: Miss ratios (left axis) and CPIs (right axis) for two applications as a function of cache size.

However, while MRCs provide significant insight into the miss ratios and data locality of applications, they are limited in their ability to predict performance. Consider the MRCs and CPIs shown for two applications in Figure 2. While 470.lbm's miss ratio changes by almost a factor of two over the displayed cache range, its CPI is nearly constant. For 471.omnetpp, however, the CPI curve qualitatively follows the miss ratio curve, with a higher miss ratio corresponding in decreased performance. This data indicate that miss ratio alone is not enough to analyze the performance impact of reduced cache space due to shared resources.

III.2.4 Understanding Cache Performance Metrics

When evaluating cache performance it is essential to distinguish between two categories of events: *misses* and *fetches*. We define *fetches* as the total number of cache-lines fetched from main memory while *misses* is the number of cache misses. Fetches and misses are not always the same. (See Figure 3.) For example, consider the impact of hardware prefetchers. When the prefetchers fetch data from memory that is later accessed by the program, the number of misses is reduced, while the number of fetches stays unchanged. However, if the hardware prefetchers fetch data that is not

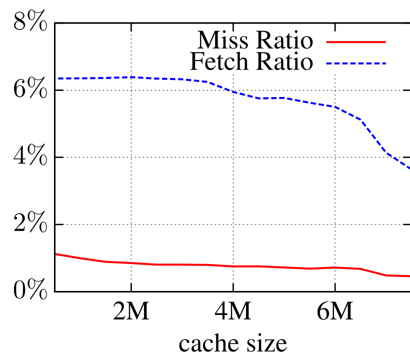


Figure 3: Miss ratio and fetch ratio for the same application.

accessed while live in the cache, the total number of fetches is increased while the number of misses stays the same. This distinction between fetches and misses is important, since it is the number of misses that dictate how many memory-related stalls the CPU suffers, while fetches determine the off-chip bandwidth consumption.

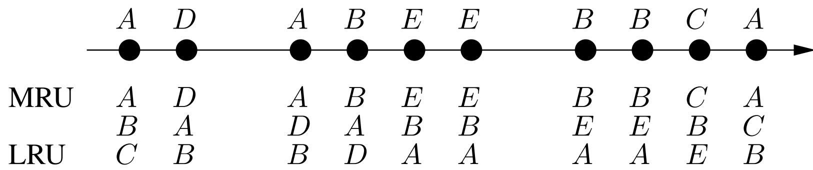
In addition to distinguishing between *misses* and *fetches*, we also distinguish between *ratios* and *rates*. For example, *miss ratio* is the number of misses per executed memory access instruction, and *fetch rate* is the number of fetches per cycle. This distinction is important because *ratios* are a property of the memory access stream, and do not incorporate specific information about the execution *rate* of the hardware, which is needed for performance analysis.

III.3 Cache Pirating

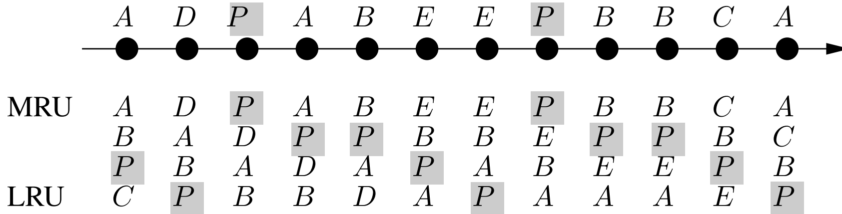
III.3.1 Overview

Cache Pirating is a method that allows us to accurately measure any performance metric available through hardware performance counters as a function of how much shared cache space is available to the application. We do this while the applications is running on real hardware, and therefore account for all effects of the memory hierarchy, such as non standard replacement policies and hardware prefetching. The basic idea of Cache Pirating is to control exactly how much shared cache space is available to the application under measurement (the Target) by co-running it with a cache-"stealing" application (the Pirate). The Pirate steals cache from the Target by ensuring that its entire working set is always resident in the shared cache. This effectively reduces the cache space available to the Target. Indeed, as long as the Pirate keeps its entire working set in the cache, we know that the Target has exactly the remainder of the cache space available.

To retain its working set in the cache, the Pirate must actively compete with the Target for cache space. How successful the Pirate is at stealing cache depends on the Target and how much it fights back. However, using performance counters we can readily determine if the Pirate is successful in stealing the requested amount of cache. *When the fetch ratio of the Pirate is zero, we can be sure its entire working set is resident in the cache, since none of its data is fetched from main memory.* To ensure this, we monitor the fetch ratio of the Pirate while measuring the performance of the Target. If the Pirate's fetch ratio is above zero, we know that the Pirate can not maintain its entire working set in the cache, and we discard the measurement. This feedback enables us to ensure the accuracy of our measurements because we can detect the point at which the Pirate is unable to steal the requested amount of cache. As alluded to above, for some target applications, it is hard for the Pirate to steal large amounts of cache, limiting the range of cache sizes that we can measure. But due to the feedback we receive from monitoring the Pirate's fetch ratio, we can detect the point at which this happens.



(a) True 3-way associative cache.



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

Figure 4: Evolution of the LRU stack of a true 3-way associative cache vs. a 4-way associative cache with the Pirate (P) stealing one way. The contents and relative ordering of the remaining 3 ways in the 4-way cache are not affected by the Pirate, resulting in a cache that behaves as the desired true 3-way cache.

III.3.2 The Pirate

For a Pirate application to be successful it must meet the following three objectives: *1) The cache space available to the Target must behave like a real cache of the intended size.* This means that the cache-lines used by the Target have to be evicted in the same order (determined by the replacement policy) that they would if the Target was running on a system with a cache of the intended size. *2) The Pirate has to be capable of keeping large working sets resident in the shared cache.* The larger the working set it can keep resident in the cache, the more cache it can steal from the Target. *3) The Pirate can not make significant use of any shared resource other then the shared cache its stealing.* Doing so could unintentionally impact the performance of the Target, which can distort the performance measurements. A side benefit of keeping the Pirates entire working set in the cache is that it will not consume shared off-chip bandwidth.

Stealing LRU Cache

We begin our discussion of the Pirate application in the context of (true) LRU caches. For LRU caches, we can conceptually think of each cache set as being organized as a finite sized stack, with the most recently used (MRU) cache-line at the top, and the least recently used (LRU) cache-line at the bottom. On a cache miss, the LRU cache-line is evicted to make room for the newly fetched cache-line which is pushed on the top of the stack. On a cache hit, the accessed cache-line is moved to the top. Figure 4(a) shows from left-to-right how the stack of one set in

a 3-way associative cache evolves over time with the access pattern shown at the top. The stack maintains an age ordering of its cache-lines, with the more recently used cache-lines at the top of the stack. Importantly, it also maintains the relative age ordering among the cache-lines.

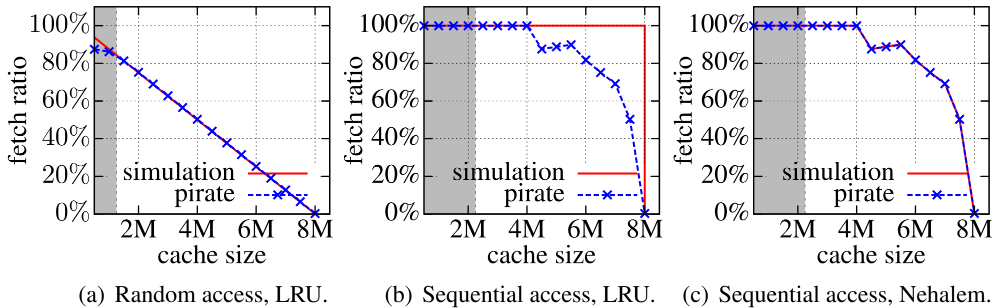
Figure 4(b) shows how the stack evolves for a 4-way associative cache when the Target is co-running with the Pirate. In this example, the Pirate is configured to steal one cache-line per cache-set, thereby leaving three cache-lines per set to be used by the Target. To avoid having its cache-line evicted,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

Now, the question is: In this 4-way associative cache with the Pirate stealing one cache-line, do the remaining three cache-lines available to the Target behave as a the equivalent 3-way associative LRU cache would? Comparing Figure 4(a) and Figure 4(b), we see that this is indeed the case. The stack content and the order of the cache-lines belonging to the Target are exactly the same in the two figures. This observation holds true in general, and satisfies our requirement that the remaining cache space available to the Target behaves as expected.

In summary, the most effective access pattern to achieve the objectives listed above is a simple linear access pattern. As long as the Pirate's access rate is high enough it will retain its entire working set in the cache. However, the more cache-lines the Pirate tries to steal, the higher its access rate needs to be to avoid having its cache-lines evicted by the Target. Fortunately, the linear access pattern can easily be implemented to maximally exploit features such as out of order execution and hardware prefetchers, allowing the Pirate to achieve the highest possible L3 access rate.

Stealing Cache in a Nehalem-Based System

Figure 5 shows the MRCs for two micro benchmarks generated using Cache Pirating on our Nehalem-based evaluation system, and reference MRCs generated using a trace driven LRU cache simulator. On the left (Figure 5(a)) is the MRC for a micro benchmark that randomly accesses a working set of 8MB. The MRC generated by Cache Pirating perfectly matches the reference curve down to a cache size of 1MB. Below 1MB, our measured fetch rate for the Pirate has increased sufficiently that we determine that the Pirate is no longer able to maintain its working set in the cache, and we can no longer take accurate measurements. To indicate this we have shaded that region of the graph in gray.



(a) Random access, LRU. (b) Sequential access, LRU. (c) Sequential access, Nehalem.

Figure 5: MRCs for two micro benchmarks that access data in random and sequential patterns for our Nehalem system. The gray regions show where the Pirate experienced elevated fetch ratios, indicating that we can not trust its data. Figures 5(a) and 5(b) use reference curves from an LRU cache simulator while 5(c) uses a Nehalem-specific cache simulator. (Simulating a pseudo-LRU policy did not improve the results.) For the random access benchmark the LRU and Nehalem simulators generate identical results, but for the case of sequential accesses, the a Nehalem-specific simulator must be used to reproduce the hardware performance.

Figure 5(b) shows the MRCs for a micro benchmark that accesses an array of 8MB with a sequential access pattern. To our surprise, the Cache Pirating curve is far from the pseudo-LRU reference for cache sizes between 4MB and 8MB, even though the fetch ratio of the Pirate is virtually zero. Even more disturbingly, the Cache Pirate data shows an *increasing* fetch ratio as the cache size *increases* from 4MB and 6MB. Assuming we can trust the hardware performance counters, we know that the data from the Cache Pirate is accurate, and because the Pirate had a zero fetch rate we know its entire working set was in the cache for those measurements. This suggests that the Pirate's data is correct, and that the discrepancy is due to some un-modeled behavior in the real hardware. Indeed, we were unable to explain these results until we contacted the manufacturer [19] and learned that the Nehalem L3 cache implements a replacement policy similar the clock algorithm used to approximate LRU for paging in operating systems [20]. When we generated a reference simulation MRC using this exact algorithm (Figure 5(c)) we found that it matched the Cache Pirate data perfectly, including the increase in fetch ratio between 4MB and 8MB. This exemplifies the challenges of modeling real hardware with simulators, and shows both the size and qualitatively misleading results one can obtain when the wrong simulation parameters are used.

Nehalem L3 Replacement Policy

The replacement policy implemented in the L3 cache of Nehalem works as follows: For every cache-line the cache maintains an accessed bit. When a cache-line is accessed, the accessed bit is set. On eviction, the accessed bits are searched, and the first cache-line found with an unset access bit is evicted. Eventually, the

accessed bit for all but one of the cache-lines will be set. When this last cache-line is accessed all accessed bits are cleared except for the one corresponding to the last cache-line accessed. Note that this replacement policy has two invariants: The accessed bits are never all set or all unset at the same time.

When the Pirate is co-running with the Target on Nehalem system, the Pirate needs to have a high enough access frequency to its working set so that its accessed bits are always set when one of its cache-lines are considered for eviction. As long as this is the case, the replacement policy will never consider the Pirate's cache-lines for eviction. This suggests that the linear access pattern described for LRU caches is also optimal for the Nehalem cache.

However, the Nehalem cache is sufficiently different from an ideal LRU cache that the space remaining for the Target application does not behave quite the same as a "true" Nehalem cache of that size would. This deviation occurs when the access bits are cleared by the Pirate when accessing one of its cache-lines[2]. In this state, the portion of the cache owned by the Target application has no access bits set, because the only accessed bit that is set is in the Pirate's portion of the cache. As noted above, such a situation could not occur in a "true" Nehalem cache, which means that the replacement policy seen by the Target when running with the Pirate and that of a "true" cache of the corresponding size are different. We evaluated this with a "worst-case" Nehalem cache simulator that always clears all accessed bits when the last cache-line with a unset accessed bit is accessed, and found that the "worst-case" results differed very little from the "true" results across our benchmark applications. The negligible impact of this effect can be seen by examining the accuracy with which the Cache Pirate results match the "true" simulation results in Section III.4.2.

III.3.3 Enhancements

Dynamic Working Set Adjustment

The simplest way to implement the Pirate is to steal a fixed amount of cache for each execution. Therefore, to generate a performance curve for 15 cache sizes, one would re-run the Target together with the Pirate once for each cache size. This would result in an overhead of at least 15 times the execution time of the Target alone. However, the actual overhead will be larger as most of the application's runs will be with smaller amounts of available cache, and hence most applications will execute more slowly. Ideally we would like to capture data for all possible cache sizes from only one execution of the Target.

Capturing data for multiple cache sizes from a single Target run can be achieved by varying how much the Pirate steals as the Target executes. Figure 6 schemat-

[2]The state when all the Target's accessed bits are set does not present a problem. In this state, when the Target suffers a cache miss, one of the Pirate's cache-lines will be evicted. As we reject measurements taken when the Pirate's fetch ratio is above a threshold (close to zero), we effectively limit the impact of this.

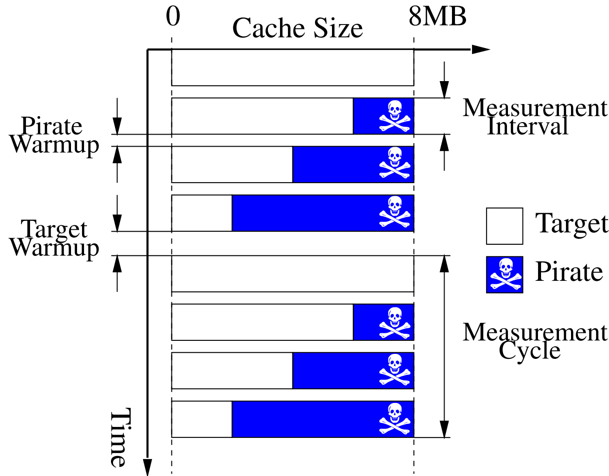


Figure 6: Schedule for dynamic working set adjustment of the Pirate while the Target application executes. During the time between each measurement interval, the application (Target or Pirate) whose working set size increases is allowed to execute alone to warm up its new cache space while the other application is suspended.

ically shows how this can be achieved. For each *measurement interval* the Pirate steals a constant amount of cache and measures the performance of the Target for that size. At the end of the measurement interval the results are recorded, and the process is repeated for the next size, with the Pirate cycling through the full range of cache sizes to be evaluated in each *measurement cycle*. For this approach to work correctly, the full measurement cycle must be evaluated in each significant program phase.

To obtain accurate measurements when dynamically adjusting the Pirate's working set size, it is important to warm up any new cache space after each change that increases the effective cache size. Therefore, we have to allow the Target to warmup its cache before entering each measurement cycle. If we did not do this, we would introduce and measure artificial cold misses for the Target. Similarly, we have to let the Pirate warm up its cache space each time it increases its working set. These cache warm ups are achieved by halting the Pirate/Target to allow the Target/Pirate to bring its working set size into the cache without having to compete for cache space.

Multithreaded Pirate

The amount of cache the Pirate can steal is limited by its ability to access its working set fast enough to keep it in the cache while the Target's attempts to use the cache at the same time. Therefore, if we can increase the Pirate's access rate we can increase the amount of cache it can steal. To do so we parallelized the Pirate by simply using multiple threads to access disjoint parts of the the Pirate's working

set simultaneously. (These threads must of course be pinned to a set of cores such that they never run on the same core as the Target.)

Multithreading the Pirate has the potential to increase its accesses rate linearly with the number of threads. However, we must make sure that the Pirate does not saturate the shared last-level cache bandwidth, as doing so can impact the execution rate of the Target. This is essential for timing dependent metrics (rates), such as IPC, but for timing independent metrics (ratios), such as fetch ratio, this is typically not an issue. Importantly, we can determine whether increasing the number of Pirate threads will impact the Target by examining its execution rate for small Pirate sizes. (See Section III.4.4.) This allows us to dynamically determine the number of threads we can use to maximize the Pirate's ability to steal cache without impacting the Target's execution rate on a per-Target basis.

III.3.4 Summary

For both the general LRU/pseudo-LRU and hardware-specific Nehalem replacement policies, the most effective access pattern for keeping the Pirates working set in the cache is a simple strided pattern. With this pattern, as long as the Pirate's access rate is high enough it will keep its entire working set in the cache. This allows us to easily compute the cache space available to the Target. But most importantly, we have shown the following: 1) When the fetch ratio (measured with performance counters) is zero, the Pirate's entire working set is resident in the cache; and 2) The cache space made available to the Target behaves like a cache with the intended replacement policy; and 3) By adjusting the number of Pirate threads we can adjust the amount of shared-cache bandwidth the Pirate consumes. Combined, these give us a tool with which we can measure any application performance metric(s) available through performance counters as a function of cache space.

III.4 Evaluation

To evaluate the Cache Pirating method we must look at the following: *1) Does the cache available to the Target behaves sufficiently similarly to a real cache of that size?* For this, we use a trace-driven cache simulator to generate fetch ratio curves, and compare them to the fetch ratio curves captured using Cache Pirating. The similarity of these curves indicates that the cache space available to the Target behaves as a real cache of the intended size. *2) How much cache the Pirate can steal?* For this, we rely on the observation that when the Pirate's fetch ratio is zero, its entire working set must be resident in the cache. This allows us to detect how much cache the Pirate can steal with different numbers of threads. *3) How many threads can the Pirate use before its L3 bandwidth impacts execution rate the Target?* To increase the Pirate's accesses rate we use a multithreaded Pirate (Section III.3.3). However, doing so risks saturating the L3 bandwidth, which can adversely affect

the performance of the Target. To evaluate this we measure how much the Target's CPI increases when the number of Pirate threads is increased. *4) Can we generate results for multiple cache sizes from one run by varying the Pirate's working set size as the Target executes?* In order to evaluate this, we first run the Target to completion with the Pirate stealing a fixed amount of cache for each Target execution. We can then compare this to the results obtained from varying the Pirate's working size while the Target executes.

III.4.1 Experimental Setup

We have implemented Cache Pirating with both dynamic working set adjustment (Section III.3.3) and a multithreaded Pirate (Section III.3.3). We have added an additional feature that allows us to attach to a running Target process and start and stop the Pirate at specific Target instruction addresses. This latter feature is used to collect data for reference simulation comparison. For benchmark applications, we use all 28 SPEC CPU2006 applications (except for 416.gamess that we could not run on our system), unless noted otherwise. We also examined the Cigar [1] application as it has a distinctive jump in its fetch ratio curve at 6MB.

We run all experiments on a quad-core Intel Nehalem E5520 running Linux 2.6.32 configured with large pages. Our kernel is patched with the perfctr-2.6.41 patch [15] to expose the OFF_CORE_RSP_0 performance counter that we need to count per-core L3 events, such as misses and fetches. We need per-core events to measure L3 events for the Target and the Pirate threads individually. This approach can be easily adopted to the Perfevents used in recent main-line Linux kernels as soon as support for the per-core L3 events is exposed.

III.4.2 Does the Cache Behave as Expected?

Assuming we can trust the hardware performance counters, we know that the data we collect for the Target are accurate measurements of the application's behavior with the Pirate running. However, to show that this data correctly reflects the behavior of the system with a cache of the size we are trying to evaluate, we need to investigate whether the cache available to the Target behaves as expected. To do so, we compare the shared cache *fetch ratio*, as captured using the Pirate, to that generated from an address trace-driven cache hierarchy simulator. The shared cache fetch ratio is a good metric for such a comparison because it directly reflects the behavior of the cache, including replacement policies and capacity, while being less sensitive to the hardware prefetchers than miss ratio (see Section III.2.4). We can therefore use these reference results from the simulated cache to reliably assess whether the real cache available to Target application is behaving as expected.

Reference Cache Simulator

To generate our reference fetch ratio curves, we first capture addresses traces using the Pin [3] dynamic instrumentation framework, and then run them through a cache simulator that models the Nehalem cache hierarchy to the best of our knowledge (see Table 1). To speed up the reference generation we analyze the time profiles of the applications using Gprof [13] and identify the code responsible for the largest fraction of the applications' execution times. We then configure our simulator to start tracing when the applications enter their hot code segments, and capture traces of approximately one billion memory accesses. Contrary to the standard approach of fast forwarding a fixed number of instructions for all applications, this approach ensures that our traces capture relevant parts of the applications' executions. When capturing Cache Pirate data, we make sure to attach and detach the Pirate at the exact same instructions at which we started and stopped tracing to ensure a fair comparison.[3]

Cache Pirating captures data on real hardware. To make a fair comparison, our reference cache simulator therefore has to model the exact behavior of the hardware. As the manufacturer of our evaluation system has not disclosed all the details of its hardware prefetchers, we can not accurately model them in our cache simulator. Instead, we disabled as much hardware prefetching as we could on our evaluation system when capturing Cache Pirating data for this experiment. We then calibrated our cache simulator using performance counters (with no cache stealing) to measure the baseline fetch ratio of our benchmark applications. This provided us with a reference fetch ratio, which was used to offset the the fetch ratio curves generated by the cache simulator to match the reference point. This corrects for cold start effects introduced by our simulation methodology and for the prefetchers that we were unable to disable.

Table 1: *Nehalem Cache Hierarchy*

L1 Cache	**32K, 8–way set associative, private**, pseudo-LRU, write allocate, writeback
L2 Cache	**256K, 8–way set associative, private**, pseudo-LRU, write allocate, writeback, non-inclusive
L3 Cache	**8M, 16–way set associative, shared,** Nehalem replacement policy, write allocate, writeback, inclusive

[3]We were unable to instrument the 6 Fortran only SPEC benchmarks to enable the address starting and stoping required for our reference simulation, and therefore do not include them in our reference comparison.

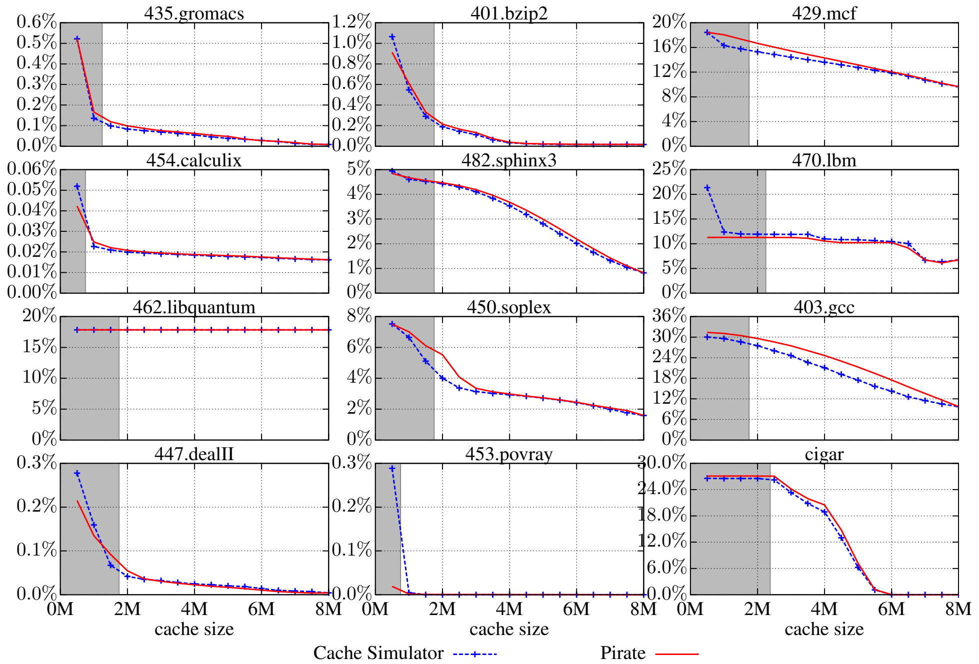


Figure 7: Cache Pirating and reference fetch ratio curves for the benchmarks with the smallest (left), median (middle) and largest (right) errors. The gray regions shows where the Pirate's fetch ratio reached above a threshold (of 1%) which indicates that the Pirate was unable to steal the desired amount of cache. As can be seen, the chosen threshold is quite conservative for many of the benchmarks.

Results

Figure 7 shows the reference and captured fetch ratio curves for the most interesting 12 of the 20 simulated benchmarks. (The remaining benchmarks either show similar behavior or have very low miss ratios to begin with. They are shown in Appendix III.A.1, Figure A.1.) The graphs are arranged from smallest error (left column) to largest error (right column). The shaded regions indicate the cache sizes for which the Pirate's fetch ratio was greater than 1%. At this point the Pirate can no longer retain its working set in the cache, and we can not trust the measured data. The fetch ratio threshold of 1% was chosen empirically and is further discussed in Section III.4.3. Across all 20 benchmarks the average and maximum absolute fetch ratio errors were 0.24% and 2.66%, respectively. The data from Cigar (lower-right) clearly displays the expected shape and indicates the 6MB working size. Indeed, in all cases, the Cache Pirate data correctly reflects the behavior of the application with the intended cache size, and for most the accuracy is excellent. Even in the case of the worst benchmark, 403.gcc, the Pirate data shows the correct trend across the full range. This demonstrates that the cache available to the Target

does indeed behave like a cache of the intended size and that the Cache Pirating method accurately captures the fetch ratio curves of the benchmark applications.

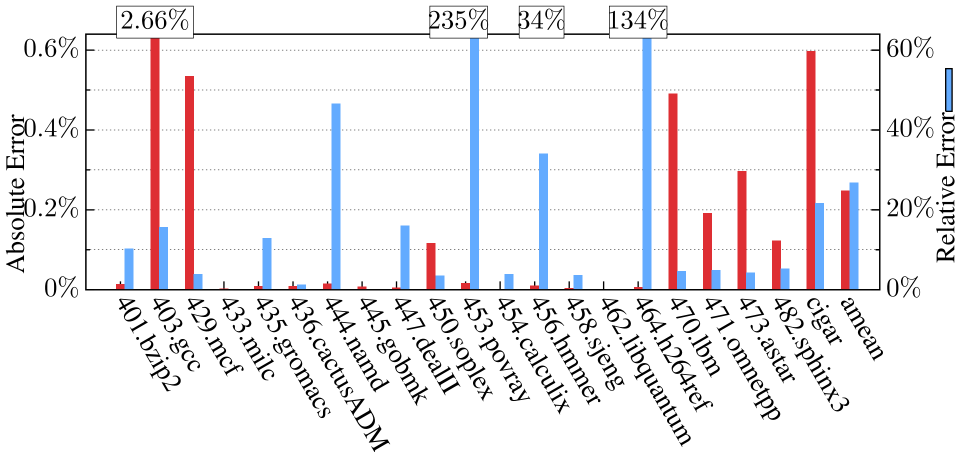


Figure 8: Absolute and relative fetch ratio errors.

To evaluate the errors more carefully, we present both *absolute* and *relative* fetch ratio errors in Figure 8. These errors are computed as the average absolute/relative difference between the Pirate and simulator fetch ratio curves across all cache sizes for which the Pirate has a $< 1.0\%$ fetch ratio. It has been previously argued [7] that relative errors in fetch ratios can be misleading for applications with low overall fetch ratios, and this data bears that out. The benchmark 453.povray has the largest relative error of 235% despite having an absolute error of only 0.01%. This is due to it having an overall fetch ratio of essentially zero (see Figure 7), which causes the relative error calculation to blow up.

III.4.3 How Much Cache Can We Steal?

We can determine when the Pirate can no longer steal the amount of cache requested by measuring when its fetch ratio increases above zero. At this point the Pirate is unable to keep its entire working set in the cache and we therefore know that the Target is not seeing the desired effective cache size. However, if we are less strict, we can actually use the Pirate's fetch ratio to put bounds on the amount of cache being stolen. For example, if the Pirate's fetch ratio is 5%, then in the worst case the Pirate has 95% of its working set resident in the cache, and in the best case 100%. This allows us to bound the error in effective cache size as a function of the Pirate's measured fetch ratio. In practice we use a threshold of 1% to

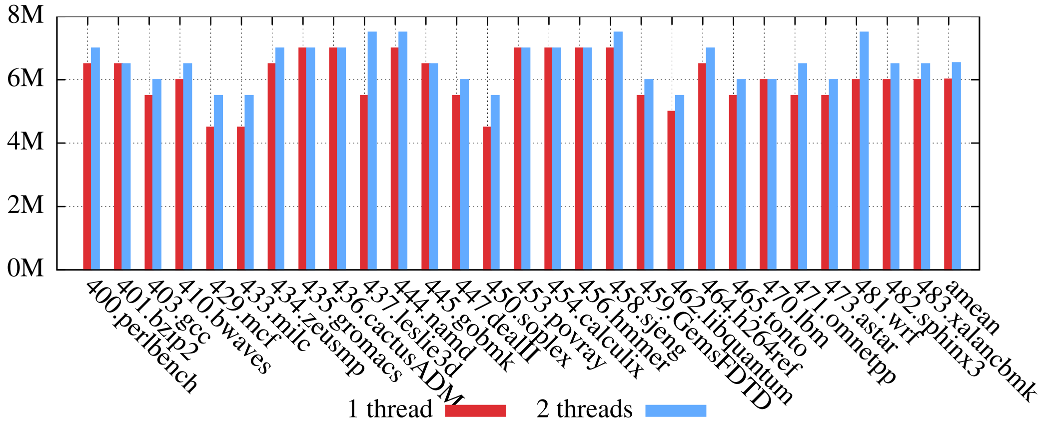


Figure 9: Maximum cache space the Pirate could steal for each application, ignoring the performance impact on the Target of running multiple Pirate threads.

determine if the Pirate is unable to steal a given amount of cache, which puts our results between 99% and 100% of the reported cache size.[4]

For the experiments presented here, we empirically chose a threshold of a 1% fetch ratio to consider the Pirate unable to maintain its working set in the cache. This number was chosen by examining a variety of benchmarks and noting that shortly after passing 1% the fetch ratio for the Pirate often went up very sharply. On average, with a fetch ratio threshold of 1%, the Pirate can steal 6.4MB of cache with one thread, and 6.9MB with two, or 80% and 86% of the 8MB of total cache, respectively. (See Figure 9.) (Note that these results were collected with the Pirate not varying its size dynamically, and are therefore not necessarily the same as the gray zones shown Figure 7.) One of the most difficult application for the Pirate to steal cache from is 462.libquantum due to its streaming access pattern. This pattern is both prefetcher-friendly, which increases its fetch rate, and contains little reuse, which thrashes the cache. Yet despite this, the Pirate able to steal 69% of the cache and obtain accurate performance measurements.

III.4.4 How Many Pirate Threads?

Multithreading the Pirate allows us to increase the Pirate's access rate to the shared L3 cache, and thereby potentially increase the amount of cache it can steal. However, doing so also increases the Pirate's use of the shared L3 bandwidth, which can adversely impact the execution rate of the Target. We therefore need a method to determine how many Pirate threads we can execute without impacting the Target's execution rate.

[4] The maximum off-chip bandwidth consumption of the Pirate with a 1% fetch ratio is 0.3GB/s, which we consider small enough to not impact the Target performance.

To determine the maximum number of Pirate threads, we first have the Pirate steal 0.5MB of cache with one thread. If it is able to successfully steal this much cache, we increase the number of Pirate threads while we measure the Target's CPI. We then examine the Target's CPI to see if it increased as we increased the number of Pirate threads. If it did, then we clearly know that we can not increase the number of Pirate threads without impacting the Target's execution rate. However, if the Target's CPI did not increase as we increased the number of Pirate threads, then we can safely run that many Pirate threads at all smaller Target cache sizes.

The reason for this is that as the Pirate steals more cache, the Target gets less cache space, its execution rate decreases (or stays the same), and its L3 bandwidth consumption therefore decreases (or stays the same). Furthermore, the L3 bandwidth consumed by the Pirate is the same no matter how much cache it steals. Therefore, if the bandwidth demand of the Target can be satisfied when the Pirate steals a small amount of cache it must also be satisfied when the Pirate steals more cache, as the Target's bandwidth demand is lower. This allows us to use the method described above to dynamically determine the maximum number of Pirate threads we can run on a per-target basis.

However, for this method to work we first need to verify that the Pirate does not impact the Target's execution rate with one thread. To investigate this we identified 10 SPEC benchmarks whose fetch ratio does not increase when their cache space is reduced by a small amount (in this case 0.5MB). For these benchmarks, we expect the CPI stay unchanged when their cache space is reduced. We then used the Pirate to steal 0.5MB of cache from these applications and compared the measured CPI to that when the applications run alone (using the whole L3 cache). The average and maximum relative difference were 0.2% and 0.6% respectively. This indicates that the Pirate can indeed use at least one thread without impacting the execution rate of these applications.[5]

Using these criteria we were able to run 14 of the 28 SPEC benchmarks with two threads without impacting their execution rate. Considering how many threads the Pirate can use for each application the average amount of cache it can steal is 6.14MB, or 77% of the total cache capacity. The results presented in this paper were collected using the maximum number of Pirate threads we could run without impacting Target performance.

[5]We also evaluated the maximum number of Pirate threads we could execute by measuring the Pirate's L3 bandwidth. We found that when running 1, 2, 3, and 4 Pirate threads, each thread was able to achieve an L3 bandwidth of 28.7, 28.4, 22.8, and 17.0GB/s, respectively. This demonstrates that two Pirate threads do not saturate the L3 bandwidth, while three do. Since three Pirate threads and no Target saturate the L3 bandwidth, we can not use more than two Pirate threads without affecting the Target's performance.

III.4.5 Dynamically Varying the Pirate Size

Dynamically varying how much cache the Pirate steals while the Target is running allows us to capture data for the full range of cache sizes from a single execution of the Target. To evaluate the effectiveness of this approach, we collected reference data by running the Pirate and Target to completion once for each cache size, and compared it to measurements taken while dynamically adjusting the Pirate's working set size. All measurement were done with the Pirate running the maximum number of threads determined using the algorithm described in Section III.4.4.

Table 2: *Average and maximum execution time overhead and relative CPI error.*

		With Gcc	Without Gcc
Measurement Interval Size	Avg./Max Overhead (%)	Avg./Max. Error (%)	Avg./Max. Error (%)
10M	6.6 / 18	0.7 / 2.4	0.6 / 1.6
100M	5.5 / 17	0.5 / 3.1	0.3 / 1.0
1B	5.1 / 13	1.9 / 23	0.8 / 3.5

To evaluate the tradeoff between accuracy and overhead we evaluated measurement intervals of size 10M, 100M, and 1B executed Target instructions (see Figure 6). In all cases data was collected for 15 different cache sizes, ranging from 8MB to 0.5MB in 0.5MB increments The resulting execution time overheads and errors are presented in Table 2. For these three measurement intervals, the average increase in execution time over running the Target alone was 6.6%, 5.5% and 5.1%, respectively. This overhead is clearly low enough to analyze the complete executions of real applications.

The accuracy varied with measurement interval size, with 100M executed Target instructions giving the most accurate measurements. Across all benchmarks, the average relative CPI error was 0.5%, with a maximum error of 3.1%. Across all interval sizes, 403.gcc had the largest errors of 2.4%, 3.1% and 23%, respectively. The reason for 403.gcc's large error is that its many small phases are not accurately captured with large measurement intervals. Decreasing the interval size to 10M decreases 403.gcc's error to 2.4%, for an average error across all benchmarks of 0.7%, and an average overhead of 6.6%. This demonstrates that dynamically varying the Pirate's size can reduce the overhead from 1500% for 15 cache sizes to 5.5% with only a 0.5% relative increase in CPI error.

A further reduction in overhead could be accomplished by running the Pirate in a sampling mode where it waits for a random amount of time between measurement cycles. As long as the the sampling covers all phases of the application fairly, this would allow accurate data collection with a further reduction of the overhead. Such approaches have been used to speed up simulation [16] and stack distance collection [6]. However, we have not implemented this approach.

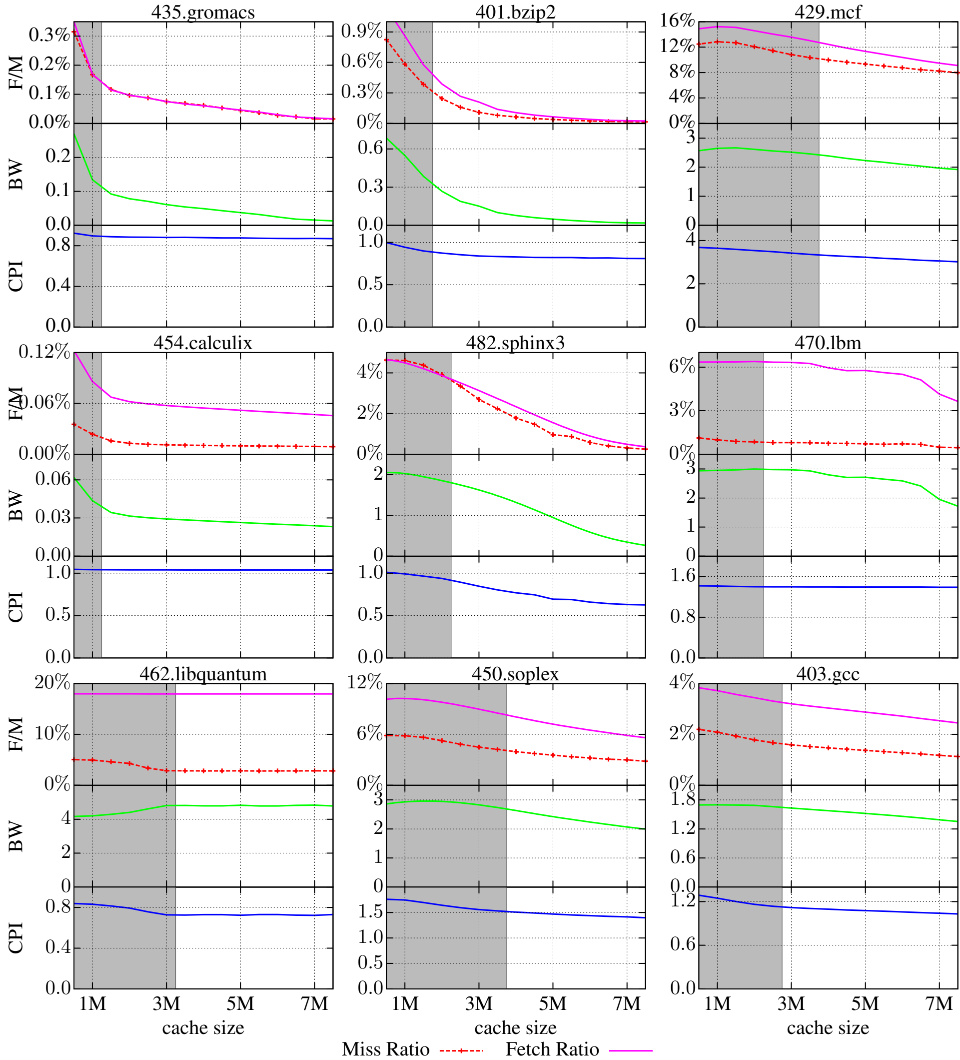


Figure 10: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.

III.5 Results

The results of applying the Cache Pirating method to collect performance (CPI), bandwidth (GB/s), miss ratios and fetch ratios for several benchmarks are shown in Figure 10. (Graphs for more benchmarks are shown in Appendix III.A.2, Figure A.2 and A.3.) As can be seen from the data, these applications span a wide range, with off-chip bandwidth varying from 5.0GB/s (462.libquantum) to

0.01GB/s (401.bzip), CPI from 3.5 (429.mcf) to 0.7 (462.libquantum), and miss ratios from 10% (429.mcf) to 0.009% (454.calculix).

For most of the benchmarks, the CPI curves are relatively flat as the cache size is decreased, despite noticeable increases in miss ratio. The reason for this can be clearly seen by examining the off-chip bandwidth consumption. As the cache size is reduced, the bandwidth increases to compensate. How successful an application is in compensating for decreased cache size with increased bandwidth depends on its sensitivity to long-latency memory operations and how effectively it can utilize the hardware prefetchers.

For example, 435.gromacs, has a constant CPI down to 1MB of cache, but its miss ratio and bandwidth increase by a factor of nearly $10\times$. However, its fetch ratio and miss ratio are nearly identical, indicating no prefetching. This suggests that the application is relatively insensitive to the increased memory latency it sees when its miss ratio increases from 0.01% to 0.1%.

482.sphinx3 behaves quite differently from 435.gromacs. As its cache size is decreased its CPI increases by 50%, while its miss ratio and bandwidth increase by a factor of $20\times$. The fetch ratio and miss ratio curves are slightly different indicating that there is a small amount of prefetching. However, the significant performance decrease despite the increased bandwidth indicates that the benchmark is more sensitive to increased memory latency.

470.lbm shows an $8\times$ difference between its fetch and miss ratios, indicating 8 prefetch memory access for every demand access. However, the relative increase in miss ratio is still roughly $2\times$. This indicates that 470.lbm is also relatively insensitive to the increased latency. The data in Figure 11 show the performance of 470.lbm with hardware prefetching disabled. Disabling hardware prefetching reduces bandwidth by a third and increases CPI at all cache sizes. Furthermore, the CPI is now no longer constant with varying cache size, clearly showing that prefetching was helping to compensate for the reduced cache space. This demonstrates that 470.lbm not only heavily leverages hardware prefetching, but also heavily benefits from it.

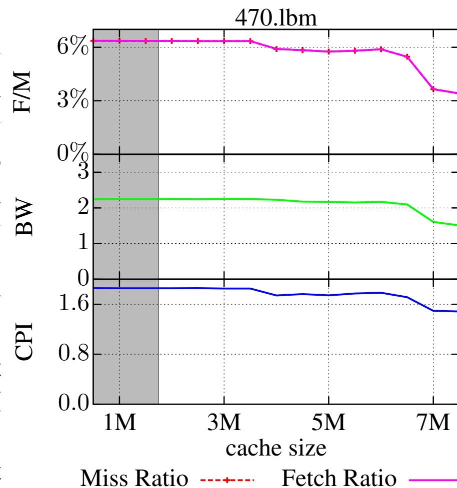


Figure 11: Performance data for 470.lbm with hardware prefetching disabled. (Fetch ratio and miss ratio are identical.)

This data shows that on real systems most applications are not overly sensitive to decreasing cache capacity because they can compensate with increased bandwidth. However, this requires that the memory system provides sufficient aggregate band-

width for all cores sharing the cache. With the bandwidth data we can collect, we can estimate the aggregate bandwidth demand and predict when it will reach the system's bandwidth limit. This relationship between available cache size and off-chip bandwidth makes it clear that it is essential to take into account the impact on bandwidth when examining the effect of sharing cache capacity.

III.6 Related Work

The most closely related work to Cache Pirating is that of Xu et al. [4, 24]. As in Cache Pirating, they use a stress application (called Stressmark) to steal cache from a co-run target application. But unlike our approach of controlling the stress application to ensure that it keeps its working set resident in the cache, they infer the how much cache their stress application from an analysis of its known MRC. This approach incurs two measurement problems avoided by Cache Pirating. First, by allowing the stress application to experience a fetch ratio they consume off-chip bandwidth, which can adversely affect the performance of the target application. Second, the miss ratio of the stress application is determined by its interaction with the target application's cache footprint. As this behavior varies during execution, they are only able to determine the average cache utilization for a given execution. This presents a problem for applications where varying phases have distinct cache behavior as the minimum and maximum cache used by the target application could vary significantly from the average. Unfortunately they do not present sufficient evaluation of their approach to determine if this is significant. Cache Pirating avoids both of these issues by carefully controlling and monitoring how much of the cache is taken by the pirate application to ensure that we do not consume off-chip bandwidth and that we accurately know how much cache is available to the target application.

To test the impact of allowing Xu's Stressmark application to consume off-chip memory bandwidth, we implemented it exactly as described in the paper. We set the Stressmark application to steal 4MB of cache from the sequential access micro benchmark used in Section III.3.2. The Stressmark saturated the off-chip bandwidth, and thereby increased the measured CPI of the target 37% over that measured with Cache Pirating. This impact on the Target's CPI due to the Stressmark's off-chip bandwidth usage indicates that it is not possible to reliably measure execution rate dependent metrics (such as CPI) on our Nehalem system using this approach.

Doucette and Fedorova [5] co-run a set of micro benchmarks, called base vectors, with a Target application to measure how the Target reacts to the base vectors, and how the base vectors react to the Target. They evaluate their method on a UltraSPARC T1, with base vector application's stealing the following resources: L1 data cache capacity, L1 instruction cache capacity, L2 cache capacity and FPU cycles. In the context of this paper, the most interesting base vector is the L2 base

vector. This base vector sequentially access a fixed size working set whose size to the L2 cache size. (The L1 instruction and data base vectors are similar.) This is different from Cache Pirating that sweeps a range of working set sizes. Furthermore, they do not relate the working set size of the base vector to the cache capacity available to the Target, instead they interpret the slow down of the Target, as a single cache sensitivity measure, and the slowdown of the base vector as an single intensity measure.

Cakarevic et al. [23] use a similar set of micro benchmarks as Doucette and Fedorova to characterize the shared hardware resource in UltraSPARC T2. They co-run their micro benchmarks, stressing different shared resources, and investigate how the micro benchmarks impact each other. This allows them to identify and characterize the critical shared hardware resources. Contrary to Cache Pirating, they do not characterize the behavior of applications.

III.7 Conclusion

We have show that the Cache Pirating technique allows us to accurately measure the Target application's performance and bandwidth demand as a function of its available shared cache space. By multi-threading the Pirate and dynamically varying its working set size as the Target runs, we are able to steal on average 6.1MB of the shared cache with an average overhead of 16%, without impacting the Target's measured performance. All of this is done without requiring special hardware or modifications to the Target application or operating system. This demonstrates that the Cache Pirating technique is a viable and accurate method for measuring any combination of hardware performance counter statistics for the Target application as a function of its available shared cache space.

Cache Pirating has enabled us collect performance data for real applications running on real hardware. The results show that as the available cache size is decreased most application's bandwidth increases to compensate, resulting in relatively flat performance curves. To better understand this bandwidth increase, we can examine the fetch and miss ratio curves we to determine how much of it is due to hardware prefetchers. The ability to collect and visualize this data is an important first step towards enabling us to analyze the performance scaling effects of shared resources in the memory system. Future work is to evaluate the accuracy of this approach for scalability analysis and extend the method to measure performance in the presence of limited off-chip bandwidth. Beyond scaling analysis, we are excited to see what can be done with the myriad of other performance counters available on modern systems, and hope to make this tool available to others shortly.

1. References

[1] Cigar. http://www.cse.unr.edu/~sushil/class/gas/code/.

[2] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proc. of the Intl. Symposium on Computer Architecture (ISCA)*, Austin, TX, USA, June 2009.

[3] C.-K. Luk and R. Muth and R. Cohn and H. Patil and A. Klauser and S. Wallace G. Lowney and V. J. Reddi and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of Pogramming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005.

[4] C. Xu, X. Chen, R. P. Dick and Z. Morley Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Proc. of the Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, White Plains, NY, USA, Mar. 2010.

[5] D. Doucette and A. Fedorova. Base Vectors: A Potential Technique for Microarchitectural Classification of Applications. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34*, San Diego, CA, USA, June 2007.

[6] D. Eklov and E. Hagersten. StatStack: Efficient Modeling of LRU caches. In *Proc. of the Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, White Plains, NY, USA, Mar. 2010.

[7] D. Eklov, D. Black-Schaffer and E. Hagersten. Fast Modeling of Shared Caches in Multicore Systems. In *Proc. of the Intl. Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, Jan. 2011.

[8] D. K. Tam and R. Azimi and L. B. Soares and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proc. of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 2009.

[9] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proc. of ACM SIGMETRICS 2005*, Banff, Canada, June 2005.

[10] E. Z. Zhang and Y. Jiang and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 2010.

[11] F. Liu, X. Jiang and Y. Solihin. Understanding how Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *Proc. of the Intl. Symposium on High Performance Computer Architecture (HPCA)*, Bangalore, India, Jan. 2010.

[12] G. Hamerly and E. Perelman and J. Lau and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[13] J. Fenlason and R. Stallman. GNU Gprof. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`.

[14] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the Intl. Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 2006.

[15] M. Pettersson. Perfctr. `http://user.it.uu.se/~mikpe/linux/perfctr/`.

[16] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Intl. Symposium on Computer Architecture (ISCA)*, 2003.

[17] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept. 2007.

[18] S. Eyerman and L. Eeckhout and T. Karkhanis and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Trans. Comput. Syst.*, 27:1–37, May 2009.

[19] S. Singhal, Intel. personal communication, Sept. 2010.

[20] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[21] T. F. Wenisch and R. E. Wunderlich and M. Ferdman and B. Falsafi and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26:18–31, July 2006.

[22] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *Proc. of Intl. Symposium on Computer Architecture (ISCA)*, 2004.

[23] V. Cakarev, P. Radojkovi, J. Verdu, A. Pajuelo, F. J. Cazorla, M. Nemirovsky and M. Valero. Characterizing the Resource-Sharing Levels in the UltraSPARC T2 Processor. In *Intl. Symposium on Microarchitecture (MICRO)*, New York, NY, USA, Dec. 2009.

[24] X. Chen, C. Xu, R. P. Dick, Z. Morley Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proc. of the Design Automation Conference (DAC)*, Anaheim, CA, USA, June 2010.

III.A Appendix

III.A.1 Simulation Results

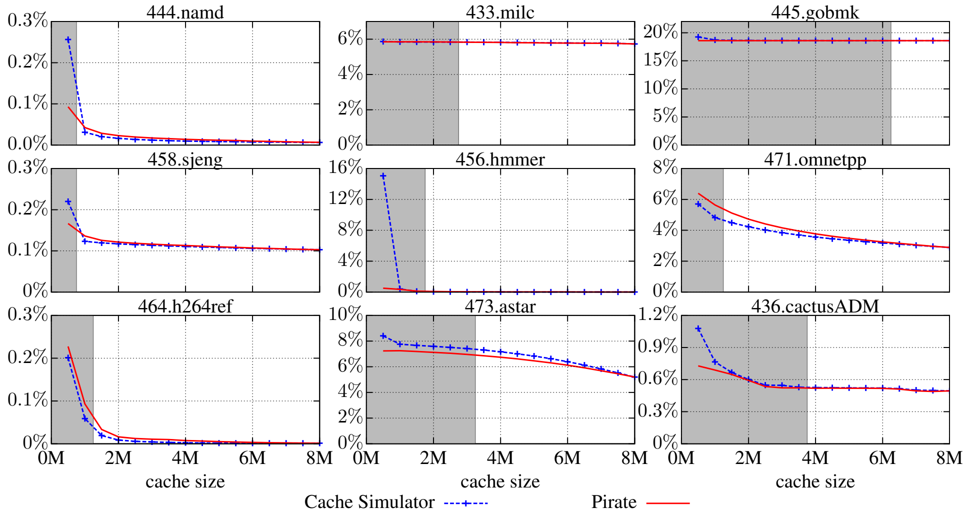


Figure A.1: Cache Pirating and reference fetch ratio curves for the remaining benchmarks. See Figure 7 for further discussion.

III.A.2 Pirate results

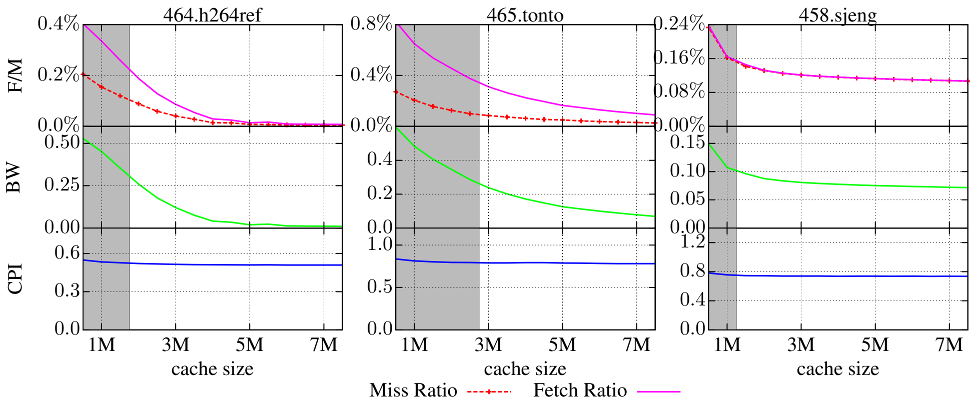


Figure A.2: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.

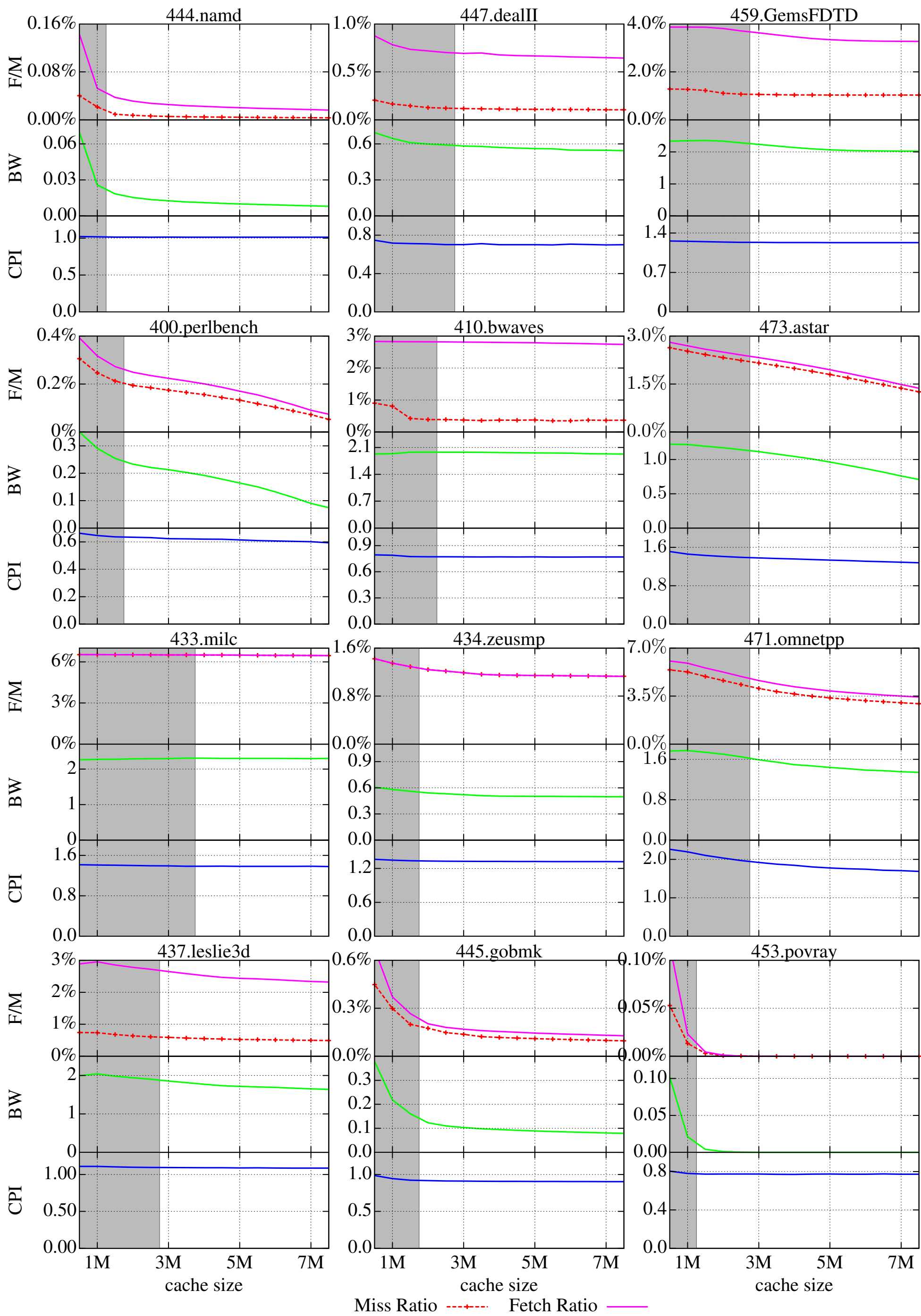


Figure A.3: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.