2011 International Conference on Parallel Processing

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
P.O. Box 337, SE-751 05 Uppsala, Sweden
{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 available shared cache capacity. The method is implemented 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 bias the Target's performance. As a result, we are able to accurately measure the Target's performance while stealing up to an average of 6.8MB of the 8MB of cache on our Nehalem based test system with an average measurement overhead of only 5.5%.

I. Introduction

The increasing core count of modern processors has not been met with a corresponding increase in off-chip bandwidth [2], [20]. Instead, modern CMPs have come to rely on large on-chip caches to reduce off-chip bandwidth demand. Since both caches and off-chip bandwidth 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], [7], [13], understanding performance as a function of the available shared memory system resources is increasingly important for performance analysis.

This paper presents *Cache Pirating*, a general, low-overhead technique for measuring any application performance metrics available through hardware performance counters, such as performance (CPI) and off-chip bandwidth (GB/s), as a function of the shared cache capacity available to the target application. Our approach is simply to measure the performance of the target application (the Target) while it is co-run with a cache-"stealing" application (the Pirate) that intentionally "steals" space in the shared cache. 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 the desired amount of cache. With this capability we can assure the accuracy of our measurements regardless of the Target application's behavior. Importantly, the Pirate is designed to steal cache without making excessive use of other shared resources, such as off-chip bandwidth, that can adversely impact the performance of the Target. As a result we can accurately measure the Target's performance as a function of the shared cache capacity available to it by varying the amount of cache the Pirate steals.

The standard approach to measuring application performance over a range of cache sizes is to use simulators (e.g. Simics [11]) or performance models (e.g. interval simulation [18]) and sweep over a range of cache sizes. For performance analysis, these methods have two limitations: 1) To accurately predict the performance of the application on real hardware the simulator has to accurately simulate the performance critical details of the target machine. This poses two challenges. First, to trust the simulation results, the simulator has to be validated against the target hardware. Second, the implementation details of performance critical features are not always disclosed by the manufacturer. 2) Even when all implementation details are known, implemented and validated, faithfully simulating them takes orders of magnitudes longer than native execution. As a result it is typically impractical to simulate full application executions with realistic input sets.

Cache Pirating overcomes both of these limitations. By using performance counters to measure the Target's performance while executing on the target machine (co-running with the Pirate), it implicitly includes all performance critical features of the target machine. This also allows the Target to execute at close to native speed, enabling analysis using realistic full-scale input sets. Furthermore, the Cache Pirating technique is easy to implement and requires no operating system modifications, thereby lowering the bar for adoption.

A. Motivating Example

To motivate Cache Pirating, we will present a specific example of how its ability to capture performance and bandwidth data on real hardware enables us to analyze throughput scaling on a commercial multicore system. For this example we will consider co-executing multiple instances of the SPEC2006 benchmark OMNeT++ or LBM. As running multiple instances of the same application together imposes no synchronization overhead (e.g., the applications instances are completely independent), one would naively expect the throughput to scale perfectly with the number of cores used. However, due

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.15

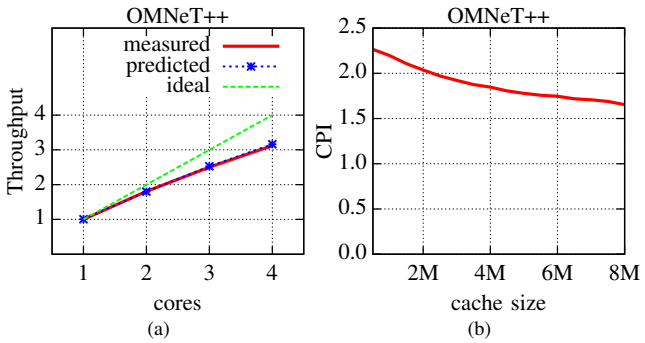


Fig. 1. Figure 1(a) shows the throughput when co-running up to four instances of OMNeT++. This graph shows the *measured* throughput scaling, *ideal* (linear) scaling, and the *predicted* scaling based on the CPI curve captured by the Cache Pirate, shown in Figure 1(b). The predicted scaling precisely matches the measured data, indicating that the CPI curve can explain the observed throughput.

to shared hardware resources, such as cache and off-chip bandwidth, this is not always the case. In this example we will show how the data captured by Cache Pirating can be used to understand the resulting throughput scaling.

Figure 1(a) shows the total throughput when running up to four instances of OMNetT++ on a 4-core Intel Nehalem processor with an 8MB shared last level cache. As the figure shows, the throughput does not scale perfectly. For example, when running four instances, we observed a total throughput of only three times that of a single instance. When running multiple instances of the same application, all instances typically receive equal portions of the shared resources. Since the application's shared cache size is reduced, the throughput scaling is affected by the application's sensitivity to its shared cache allocation.

To understand why OMNeT does not achieve perfect scaling in this setting, we examine its performance (CPI) as a function of its available shared cache space. Figure 1(b) presents this data as captured using Cache Pirating. The data clearly shows that OMNeT's performance decreases when it receives less space in the shared cache. For example, when running four instances of OMNeT, each instance will receive approximately 2MB of the shared cache, and therefore run at a CPI of 2.0, or 20% slower than with the full shared cache. From the CPI data we can accurately predict the observed throughput scaling (Figure 1(a)). As a result, we can conclude that OMNeT's limited throughput scaling is due to its increased CPI as its shared cache space is reduced.

The second application, LBM, is more interesting. Figure 2(b) shows a very flat CPI curve for LBM, indicating that its performance does not decrease when its shared cache allocation is reduced. This implies that its performance is unaffected by cache sharing. Based on this information alone, we would expect LBM to scale perfectly. However, as shown in Figure 2(a), this is not the case.

To understand this, we look at LBM's off-chip memory bandwidth curve (Figure 2(c)), also captured using Cache Pirating. This data shows the off-chip bandwidth required by

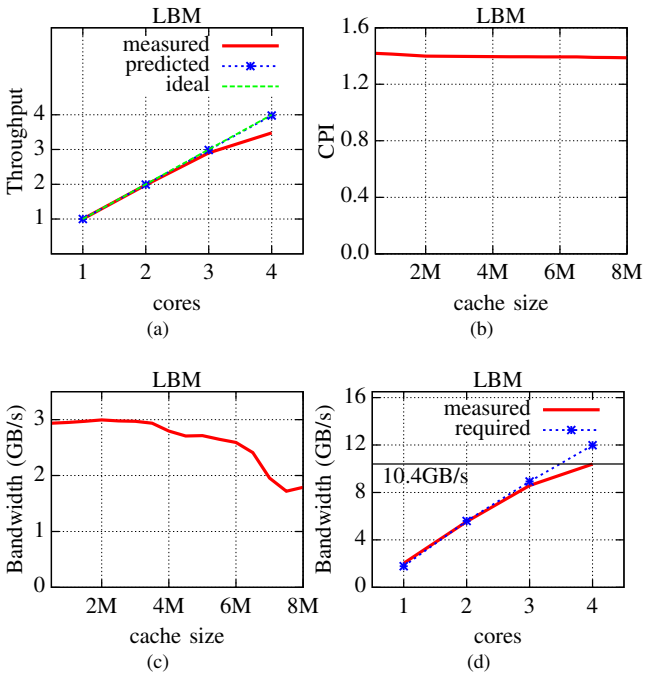


Fig. 2. Figures 2(a) and 2(b) show the throughput and CPI curves for LBM. Figure 2(c) shows the bandwidth required by LBM to achieve the CPI in Figure 2(b). The aggregate *measured* and *required* bandwidth of the co-running instances are shown in Figure 2(d), indicating that the application is bandwidth limited on this system.

LBM to achieve the CPI in Figure 2(b). We see that when LBM's available shared cache space is reduced it requires more bandwidth. Figure 2(d) shows the required and measured off-chip bandwidth for running up to four instances of LBM. As Figure 2(c) shows, when running three instances, each instance require 3GB/s, for a total of 9GB/s, which is less than the system's maximum bandwidth of 10.4GB/s. However, the required bandwidth to run four instances at the CPI shown in Figure 2(b) is 12GB/s, which is more than the system's maximum. As this bandwidth requirement is constant throughout their execution, the application instances will be memory bound, and their execution rate will be limited by the rate at which they can access memory. Since the memory system can only provide 87% (10.4/12) of the required bandwidth, we expect to achieve 87% of the performance indicated in Figure 2(b). This works out to a throughput of 3.5, which is precisely what the measured data in Figure 2(a) shows.

This example shows that the data captured by the Cache Pirating technique can enable the understanding of non-trivial scaling characteristics of real applications on a modern multicore processor. However, this is just one example of what can be done with such data. The remainder of this paper will focus on the general Cache Pirating technique, describe a low-overhead implementation, and demonstrate its accuracy and performance.

B. Terminology: Fetches vs. Misses

In this paper we distinguish between *fetches* and *misses* and *fetch ratio* and *miss ratio*. Fetches is the number of cache-lines fetched from main memory, including hardware prefetching, while misses is the number of actual cache misses. With these definitions, the number of fetches and misses are the same when hardware prefetching is disabled.

We further distinguish between *ratios* and *rates*. For example, fetch ratio is defined as the number of fetches per memory access, while fetch rate is the number of fetches per time unit and is directly proportional to off-chip bandwidth consumption.

C. Results and Contributions

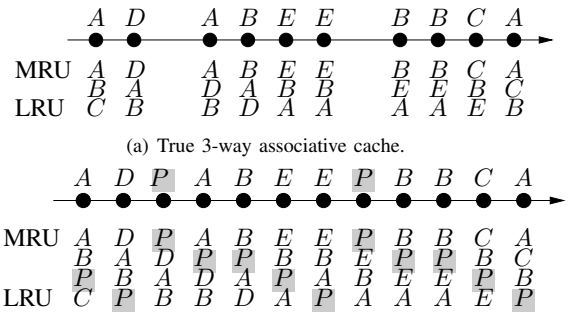
We have designed and implemented Cache Pirating, a general method that enables accurate and efficient capturing of application performance data as a function of available shared cache. The technique measures the application while running on real hardware, and therefore account for all effects of the memory hierarchy, such as hardware prefetching and memory-level parallelism. In this paper we demonstrate that Cache Pirating can steal on average up to 6.8MB of the 8MB shared cache on a Nehalem-based system, while accurately capturing the target application's fetch ratio curve with average and maximum absolute errors of 0.2% and 2.7%, respectively. This is done without adversely impacting the target application's execution rate, and with an average measurement overhead of only 5.5%. This implies that Cache Pirating can efficiently and accurately capture any application performance metric available through performance counter as a function shared cache size while the target is running on real hardware.

II. Cache Pirating

A. Overview

The basic idea of Cache Pirating is to control how much shared cache space is available to an application (the Target) by co-running it with a cache-"stealing" application (the Pirate). The Pirate steals cache from the Target by ensuring that its working set is always resident in the shared cache. This reduces the cache space available to the Target. Indeed, as long as the Pirate keeps its working set in the cache, we know that the Target has only 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.* Therefore, we monitor the fetch ratio of the Pirate while measuring the performance of the Target. If the Pirate's fetch ratio is too large, we know that the Pirate cannot maintain its working set in the cache. Therefore, any Target measurements will not accurately represent the behavior of the Target with the intended amount of cache space. This feedback enables us to assess the accuracy of how much cache we steal because we can detect the point at which the Pirate is unable to steal the requested amount of cache.



(a) True 3-way associative cache.

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

Fig. 3. Evolution of the LRU stack of a 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 3-way cache.

In general there are two ways in which the size of a cache can be reduced: by reducing the number of cache-ways or by reducing the number of cache-sets. (We do not consider reducing the cache line size.) However, when multiple applications/threads share a cache they share the cache-sets and therefore contend for cache-ways. This effectively reduces the associativity of the shared cache available to each process or thread. The Pirate application is therefore designed to "steal" cache-ways. To achieve this it is important that the Pirate is successful in stealing cache across all cache-sets. If the Target has hot cache-sets for which the Pirate cannot steal enough ways, the resulting evicted Pirate data will be observed as an increase in the Pirate's fetch ratio, which will indicate that the Pirate was unable to successfully steal that much cache.

B. The Pirate

For the Pirate application to be successful it must meet the following three objectives: *1) The cache space available to the Target must behave like a cache of the intended size.* This means that the Target should exhibit the same miss ratio as it would if it ran 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 than the shared cache it is stealing.* Doing so could unintentionally impact the performance of the Target, which would 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.

1) Stealing LRU Cache: 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 3(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.

Figure 3(b) shows how the stack evolves for a 4-way associative cache when 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. From the Target's point of view the cache behaves exactly like a 3-way cache, as the relative order of its cache-lines are the same as in Figure 3(a). This holds true in general. For example, if the Pirate had stolen two cache-lines the Target would have seen a 2-way cache. This shows that the cache space available to the Target behaves as a cache of the intended size.

The most effective way for the Pirate to keep its working set in the cache is to always access the "oldest" cache-line at the highest possible rate. Such an access pattern is easily constructed by accessing a contiguous chunk of memory with a stride equal to the cache-line size. This implies that the Pirate will steal the same number of cache-lines in every set, effectively reducing the associativity of the shared cache.

2) Stealing Cache in a Nehalem Processor: Our Nehalem based evaluation system (Section III-A) implements an approximation of LRU in its shared last-level cache that works as follows [15]: For every cache-line in a set the cache maintains an accessed bit. When a cache-line is accessed, the accessed bit is set. On an 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 its access bit is set and all other accessed bits are cleared.

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 is considered for eviction. This suggests that the same linear access pattern described for LRU caches is also appropriate for the Nehalem cache.

However, the Nehalem cache replacement policy is sufficiently different from standard LRU policies that it must be modeled accurately when generating simulation results for evaluation. (See Figure 4 for an example of the impact of these differences even with very simple benchmarks.) Further, while the linear Pirate access pattern for LRU caches is appropriate for the Nehalem cache, the space available to the Target application does not behave quite the same as if it were a "true" Nehalem cache of that size. This deviation can occur if the access bits are cleared by the Pirate when accessing one of its own cache-lines[1]. 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. In Section III-B we show that this situation does not have a significant impact and that the same linear access pattern used for LRU caches can be used to accurately steal cache on Nehalem processors.

[1]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, and this will be measured in the Pirate's fetch ratio.

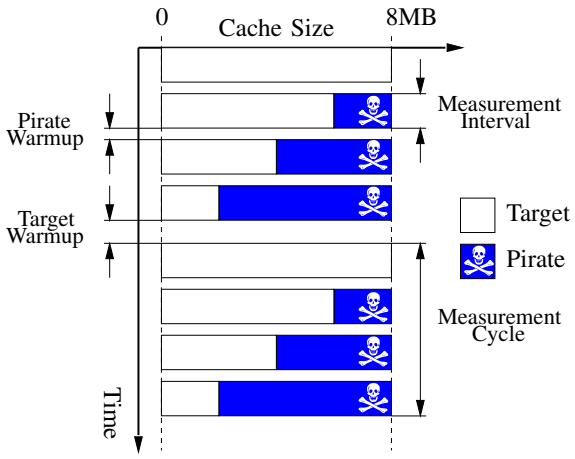


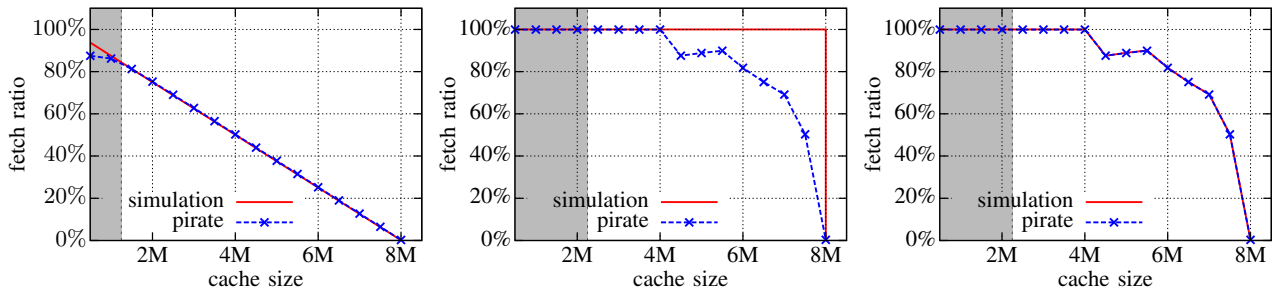
Fig. 5. Schedule for dynamic working set adjustment of the Pirate while the Target application executes. During the time between each measurement interval, the application (Target or Pirate) whose working set size increases is allowed to execute alone to warm up its new cache space while the other application is suspended.

C. Enhancements

1) 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.

To reduce this overhead, we can capture data for multiple cache sizes from a single execution of the Target. This is achieved by varying how much the Pirate steals as the Target executes. Figure 5 schematically 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.

When dynamically adjusting the Pirate's 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.



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

Fig. 4. Simulated and measured fetch ratio curves for two micro benchmarks that access data in random (4(a)) and sequential (4(b) and 4(c)) patterns for our Nehalem system. The gray regions denote where the Pirate experienced elevated fetch ratios, indicating that we can not trust its data. Figures 4(a) and 4(b) use reference curves from an LRU cache simulator while 4(c) uses the Nehalem-specific cache simulator we developed for generating references. (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, a Nehalem-specific simulator must be used to reproduce the hardware performance. This exemplifies the challenges of modeling real hardware, and shows both the size and qualitatively misleading results one can obtain when the wrong parameters are used.

2) Multithreaded Pirate: The amount of cache the Pirate can steal is limited by the rate at which it accesses its working set. Therefore, we use multiple threads accessing disjoint parts of the Pirate's working set to increase the amount of cache it can steal. (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 cache bandwidth, as doing so can adversely impact the execution rate of the Target. This is essential for timing dependent metrics (rates), such as IPC. To avoid this, we dynamically determine whether increasing the number of Pirate threads will impact the Target (See Section III-C).

III. Evaluation

To evaluate the Cache Pirating method we look at the following: *1) Does the cache available to the Target behave as expected?* 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. *2) How much cache can the Pirate 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 the Target's execution rate?* To increase the Pirate's accesses rate we use a multithreaded Pirate, which risks saturating the L3 bandwidth and adversely affecting 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 Target execution by dynamically varying the Pirate's working set size?* 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 dynamically varying the Pirate's working size while the Target executes.

A. Experimental Setup

We have implemented Cache Pirating with both dynamic working set adjustment (Section II-C1) and a multithreaded Pirate (Section II-C2). The Pirate implements the simple linear access pattern discussed in Section II-B1, which results in optimal use of the hardware prefetchers and a negligible code footprint.

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 otherwise noted. We also examined the Cigar [1] application as it has a distinctive jump in its fetch ratio curve at 6MB. Most of the benchmarks actively use the shared L3 cache as can be seen from their increasing fetch ratios as the available cache size is decreased in Figure 6.

We ran all experiments on a quad-core Intel Nehalem E5520 under Linux 2.6.32 configured with large pages. The Target application was pinned to one core and the Pirate to other core(s) to ensure that the Pirate did not impact the Target's execution performance. Our kernel was patched with the perfctr-2.6.41 patch [10] to expose the OFF_CORE_RSP_0 performance counter that we need to count per-core L3 events, such as misses and fetches, for the Target and Pirate threads separately. More recent kernels have built-in support for this performance counter through the Perfevents framework, thereby avoiding the need for a patch.

B. 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, we need to investigate if this data correctly reflects the behavior of the Target when running on a system with the corresponding cache size. 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

TABLE I
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

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 I-B). We can therefore use these reference results from the simulated cache to reliably assess whether the cache available to Target application is behaving as expected.

1) 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 I). To speed up the reference generation we analyze the time profiles of the applications using Gprof [9] 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.[2]

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.

2) Results: Figure 6 shows the reference and pirate fetch ratio curves for 12 of the 20 simulated benchmarks, chosen

[2]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.

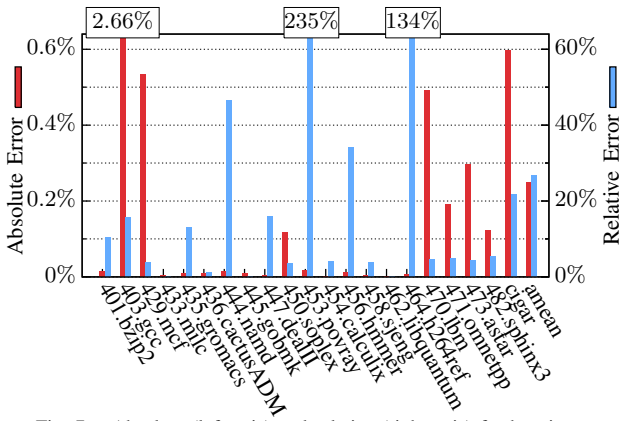


Fig. 7. Absolute (left axis) and relative (right axis) fetch ratio errors.

to show examples of the smallest errors (left column) through largest errors (right column). The shaded regions indicate the cache sizes for which the Pirate's fetch ratio was greater than 3%, at which point the Pirate can no longer retain its working set in the cache, and we cannot trust the measured data. The fetch ratio threshold of 3% was chosen empirically. (See Section III-C.)

Across all 20 benchmarks the average and maximum absolute fetch ratio errors were 0.2% and 2.7%, respectively. The data from Cigar (lower-right) clearly displays the expected shape and indicates the 6MB working size. 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 for 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 as expected and that the Cache Pirating method accurately captures the fetch ratio curves of the benchmark applications[3].

To evaluate the errors more carefully, we present both *absolute* and *relative* fetch ratio errors in Figure 7. 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 less than 3.0% fetch ratio. It has been previously argued [6] that relative errors in fetch ratios can be misleading for applications with low overall fetch ratios, and this data bears that out. While the average relative fetch ratio error is 27%, this is primarily due to two applications whose very low absolute miss rates lead to high relative errors. The benchmarks 453.povray and 464.h264ref have the largest relative errors of 235% and 134%, respectively, despite both having absolute errors of only 0.01%. This is due to both having overall fetch ratios of essentially zero (see Figure 6 for 453.povray; 464.h264ref is similar), which cause the relative error calculation to blow

[3]For completeness, we generated reference results while keeping the associativity constant across all cache sizes. The results showed that for more than four ways there was no difference between the two approaches for all applications except LBM.

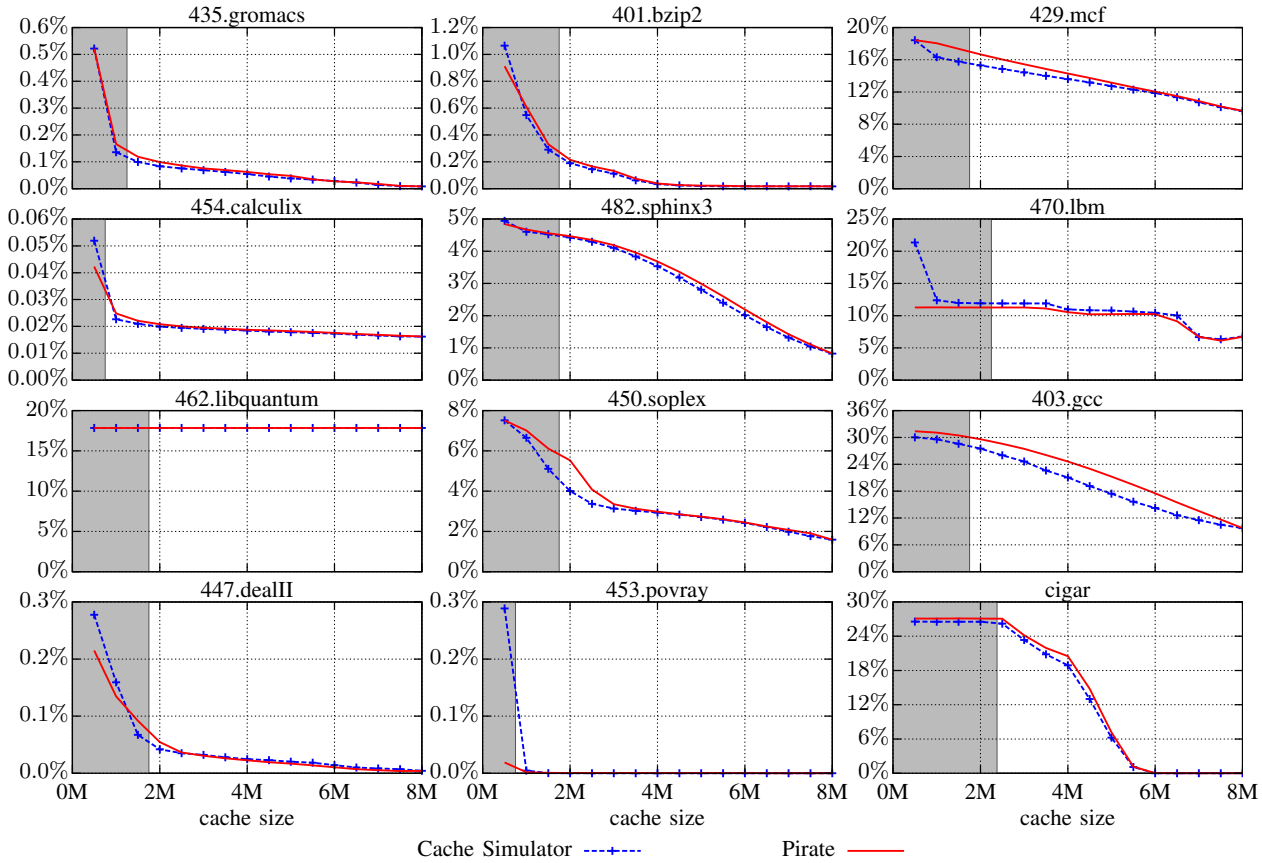


Fig. 6. Cache Pirating and reference fetch ratio curves for the benchmarks with the smallest (left), median (middle), and largest (right) errors. The gray regions shows where the Pirate's fetch ratio rose above a 3% threshold, indicating that the Pirate was unable to steal the desired amount of cache. As can be seen, the chosen threshold is quite conservative for many of the benchmarks.

up. Across all benchmarks, the low average absolute fetch ratio error of 0.2% quantitatively demonstrates that the cache space available to the Target application behaves as desired.

C. How Much Cache Can the Pirate Steal?

The amount of cache that the Pirate needs to be able to steal depends on what the captured data is to be use for. For example, to perform the analysis in Section I-A, the Pirate has to steal 6MB of cache. This leaves the Target with 2MB, corresponding to the amount of cache received by each of the four co-running instances of the studied applications. In this section we show that the Pirate can steal an average of 6.8MB of cache, and at least 6MB for all our benchmarks except 462.libquantum.

We can determine when the Pirate can no longer steal the requested amount of cache by monitoring its fetch ratio. When the Pirate's fetch ratio is above zero, the Pirate is unable to keep its entire working set in the cache and we know that the Target is not seeing the desired cache size. If we are less strict, we can use the Pirate's fetch ratio to bound the amount of cache it steals. For example, if the Pirate's fetch ratio is 3%, then it has between 97% and 100% of its working set resident in the cache. For the experiments presented here, we use a fetch ratio threshold of 3%. This limits the Pirate's off-chip bandwidth consumption to 0.9GB/s of the 10.4GB/s available, which has virtually no impact on the Targets' performance. (See Section IV for a discussion of the Target applications' bandwidth requirements.) With a 3% fetch ratio threshold a single-threaded Pirate can steal on average 6.6MB of cache.

To further increase the amount of cache the Pirate can steal, we can run the Pirate with two threads, effectively doubling the access rate of the Pirate, and enabling us to steal an average of 6.9MB. However, doubling the access rate of the Pirate also doubles its L3 bandwidth consumption. With two threads the Pirate uses 56GB/s of the 68GB/s total L3 bandwidth, leaving 12GB/s to the Target applications. For applications that require more than 12GB/s of L3 bandwidth, a dual-threaded Pirate can adversely impact the performance of the Target and distort our measurements.

Fortunately, we can easily determine how many threads the Pirate can safely use. The Pirate can always use at least one thread as highest achievable L3 bandwidth consumption for two cores is 56GB/s, which is less than the total L3 bandwidth of 68GB/s. Therefore we can always run one Pirate thread with the Target without exceeding the system's L3 bandwidth. To determine if the Pirate can use two threads, we set it to steal a small amount of cache (0.5MB), first with one, and

TABLE II
CACHE CAPACITY STOLEN VS. TARGET SLOWDOWN FOR THE HARDEST-TO-STEAL-FROM APPLICATIONS

Benchmark	1 Thread MB Stolen	2 Threads MB Stolen	$\frac{cpi_2 - cpi_1}{cpi_1}$
429.mcf	5.5	6.5	5%
433.milc	5.5	6.0	3%
450.soplex	5.5	6.0	5%
462.libquantum	5.0	5.0	6%

then with two threads, and measure the Target's CPIs (cpi_1 and cpi_2). If the Target did not slowdown, we know that it is safe to steal any amount of cache with two threads. This is because when the Pirate steals more cache the Target receives less cache space, and its execution rate slows down (or stays unchanged). Its L3 bandwidth demand therefore decreases (or stays unchanged). At the same time the Pirate's L3 bandwidth demand is unchanged, resulting in a lower overall demand for L3 bandwidth.

For multithreaded Targets it is important to consider the aggregate bandwidth of the Target threads when deciding how many Pirate threads to run. While we believe this is a straightforward extension, we have not investigated it for this work.

We used the above method to determine which applications can run with two Pirate threads. Our baseline threshold requires less than a 1% slowdown ($\frac{cpi_2 - cpi_1}{cpi_1}$) to enable the second Pirate thread. With a 1% threshold, we are able to steal an average of 6.7MB, and more than 6MB of cache for all but four applications (433.milc, 429.mcf, 450.soplex and 462.libquantum). If we relax this requirement, we can tradeoff accuracy for the amount of cache we can steal. As a result, we can steal an average of 6.8MB, and at least 6MB for all but 462.libquantum, for which we can not steal more than 5MB even with two threads. Table II shows how much cache the Pirate can steal with one and two threads, and the resulting Target slowdown, for these four benchmarks.[4]

D. 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 accuracy 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 by dynamically adjusting the Pirate's working set size during a single Target execution. All measurement were done with the Pirate running the maximum number of threads determined as described in Section III-C.

[4]It is worth noting that the applications for which it is hardest to steal cache are the ones for which it is generally least critical to do so. This is simply because these applications fight the hardest to retain cache space, thereby making it unlikely that another application will steal more of the cache than the Pirate. Indeed, as the Pirate is designed to be the most effective cache-stealing application possible, one would expect that an application for which the Pirate has a hard time stealing cache will not experience more cache being stolen by other applications.

TABLE III
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 5) and 15 different cache sizes, ranging from 8MB to 0.5MB in 0.5MB increments. (See Table Table III.) For these 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 23% error for 403.gcc at the largest interval size indicates that its phases are too small to be accurately captured at that granularity. With the smallest interval size, this error decreases to 2.4%, for an average error across all benchmarks of 0.7%, and an average overhead of 6.6%. This evaluation demonstrates that dynamically varying the Pirate's size can reduce the overhead for measuring 15 cache sizes from 1500% to 5.5%, with only a 0.5% relative increase in CPI error over running with a fixed Pirate size.

IV. RESULTS

Cache Pirate collected performance (CPI), bandwidth (GB/s), miss ratios and fetch ratios for several benchmarks are shown in Figure 8. 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 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×. 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%.

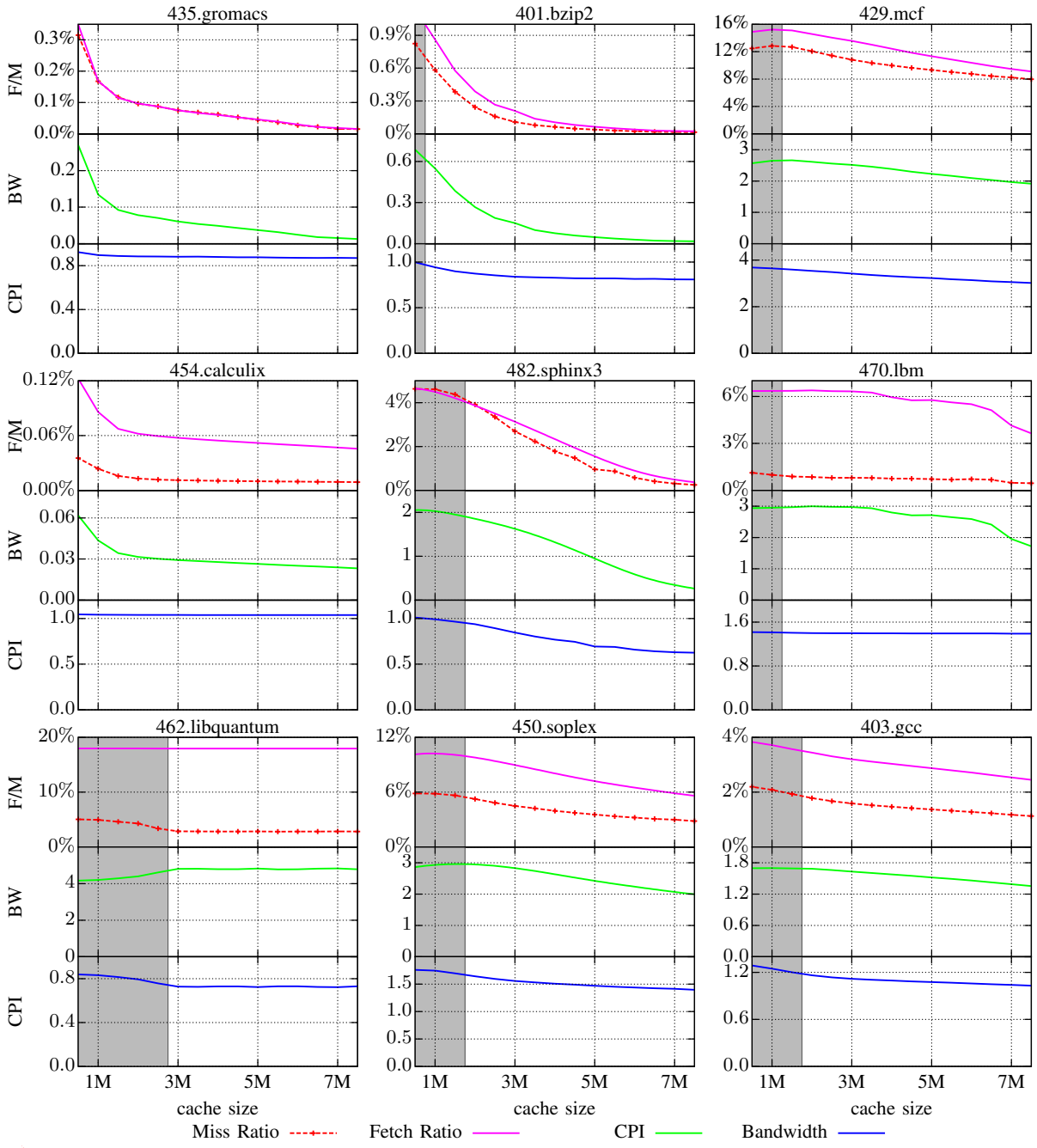


Fig. 8. Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.

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×. 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× difference between its fetch and miss ratios, indicating 8 prefetched memory accesses for each miss. However, the relative increase in miss ratio is still roughly 2×. This indicates that 470.lbm is also relatively insensitive to the increased latency. The data in Figure 9 show the performance of 470.lbm with hardware prefetching disabled. This reduces bandwidth by a third and increases CPI at all cache sizes. Furthermore, the CPI is now no longer constant with varying

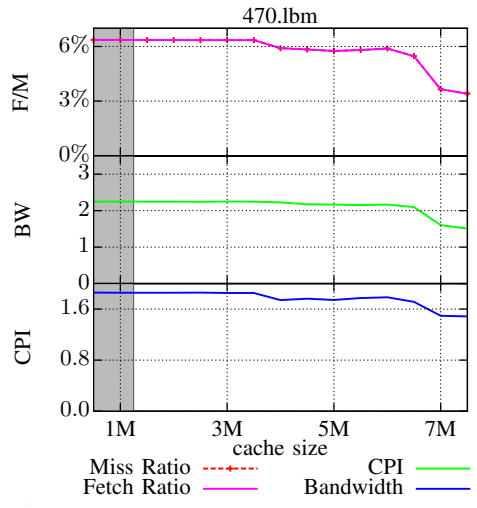


Fig. 9. Performance data for 470.lbm with hardware prefetching disabled. (Fetch ratio and miss ratio are identical.)

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.

V. Related Work

The standard approach to measuring performance as a function of cache size is to use architectural simulators (e.g. [11], [16]) or performance models (e.g. [14], [18]). For performance analysis the main limitation of simulation is the long execution time, which is typically orders of magnitude slower than native execution. To address this, a substantial amount of work has been done to improve simulation performance [8], [12], [17], in particular by trading off accuracy and detail for improved speed through the use of analytical models [14], [18]. However, despite these advances, the overhead of simulation and modeling is still a limitation for performance analysis of real application with large data sets.

Xu et al. [4] presented a model for predicting the performance degradation of co-running applications contending for shared cache. As part of their work they co-run a stress application with each application to measure its performance as a function of miss ratio. For our purpose of measuring performance as a function of cache size, their approach has two limitations: 1) Instead of ensuring that their stress application steals a fixed amount of cache, they let it freely contend for space with the Target, and then estimate the *average* amount of cache stolen after the fact. Such an average is hard to correlate to one cache size and is sensitive to program phases. 2) As their micro benchmark freely contends for cache with the Target, they cannot limit how much off-chip bandwidth it consumes, which can distort performance measurements[5].

[5]We implemented and used their method to measure the CPI of the simple sequential micro benchmark in Figure 4. When we tried to steal 4MB of cache, their stress application consumed sufficient off-chip bandwidth to increase the measured CPI by 37%.

Doucette and Fedorova [5] also co-run a set of micro benchmarks, called base vectors, with a Target application to determine the impact on the target. Their base vector for measuring shared cache contention has a similar sequential access pattern to that of the Cache Pirate, but has its working set size fixed to that of the shared cache. This provides a single cache "sensitivity" measure for the Target application and does not relate the performance of the Target to the amount of cache available, nor to specific performance measurements such as CPI or off-chip bandwidth.

Cakarevic et al. [19] use a similar set of micro benchmarks as Doucette and Fedorova, but their work focuses on hardware characterization. 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, but not the behavior of other applications.

VI. Conclusion

We have presented a general technique for quickly and accurately measuring application performance characteristics as a function of the available shared cache space. By leveraging a co-executing Pirate application that steals shared cache space and hardware performance counters, this approach works on standard hardware, accurately reflects the idiosyncrasies of the system, has a low overhead, and requires no modifications to either the operating system or target applications

To evaluate our technique we identified and investigated four key issues: 1) *Does the cache available to the Target behave as expected?* 2) *How much cache can the Pirate steal?* 3) *How many Pirate threads can we run before we impact the Target's performance?* and, 4) *Can we minimize the overhead by dynamically varying the Pirate's size?* For the first question we used a trace-driven cache simulator to show that the cache space available to the Target application behaved as expected with average and maximum absolute fetch ratio errors of 0.2% and 2.7%, respectively. We introduced a method to determine when the Pirate can execute two threads to increase its fetch rate without impacting the Target's performance and demonstrated that we were able to steal an average of 6.7MB of the 8MB cache, and up to an average of 6.8MB with a small decrease in Target execution rate. And, finally, by dynamically adjusting the amount of cache the Pirate steals during execution, we were able to reduce the total overhead to 5.5%, with an average CPI error of only 0.5%

While we motivated this general performance characterization approach with an example of how it can be used to explain throughput scaling, we believe that the data captured by Cache Pirating is fundamentally important for the understanding and optimization of applications executing in shared resource environments. Future work includes extending this approach to collect performance data against other shared resources and exploring the range of information available from the ever increasing variety of performance counters.

ACKNOWLEDGMENT

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.

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 Programming 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, 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

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