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 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%.

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 capability 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.

2. BACKGROUND AND MOTIVATION

2.1 Motivation

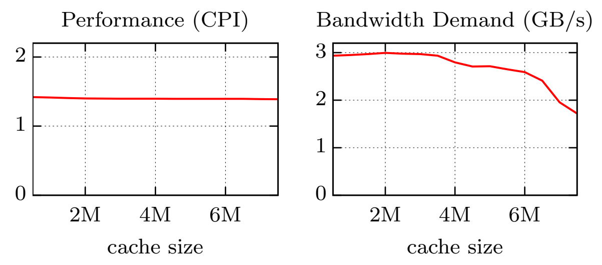


Figure 1: Sample application performance and bandwidth demand as a function of cache size.

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.

[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].

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.

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.

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

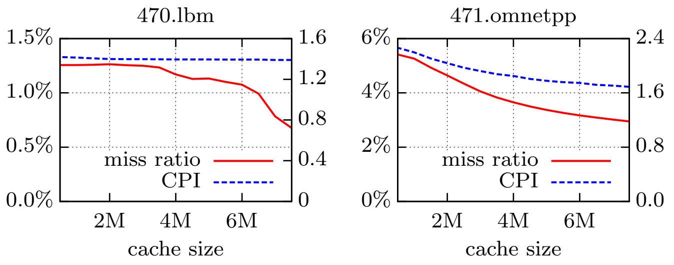


Figure 2: Miss ratios (left axis) and CPIs (right axis) for two applications as a function of cache size.

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.

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 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.

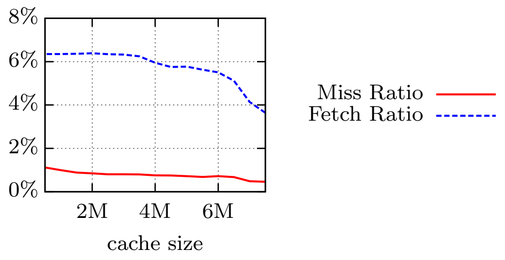


Figure 3: Miss ratio and fetch ratio for the same application.

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.

3. CACHE PIRATING

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.

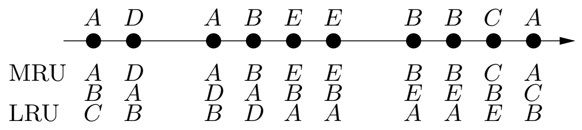
3.2 The Pirate

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.

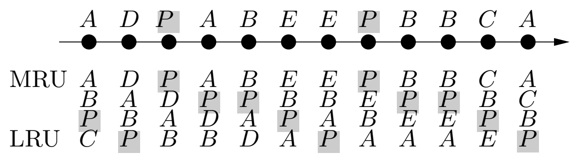
3.2.1 Stealing LRU Cache

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



(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.

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.

3.2.2 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.

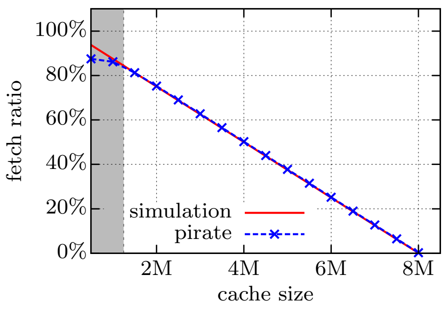
Figure 5(b) shows the MRCs for a micro benchmark that accesses an array of 8MB with a sequential access pattern. To our surprise, the Cache Pirating curve is far from the pseudo-LRU reference for cache sizes between 4MB and 8MB, even though the fetch ratio of the Pirate is virtually zero. Even more disturbingly, 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.

3.2.3 Nehalem L3 Replacement Policy

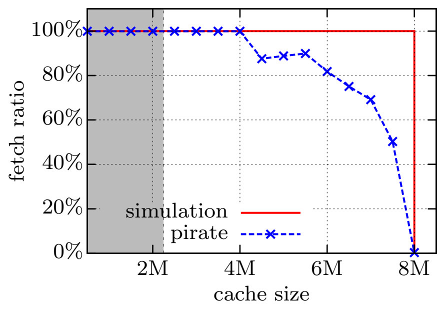
The replacement policy implemented in the L3 cache of Nehalem works as follows: For every cache-line the cache maintains an accessed bit. When a cache-line is accessed, the accessed bit is set. On eviction, the accessed bits are searched, and the first cache-line found with an unset access bit is evicted.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

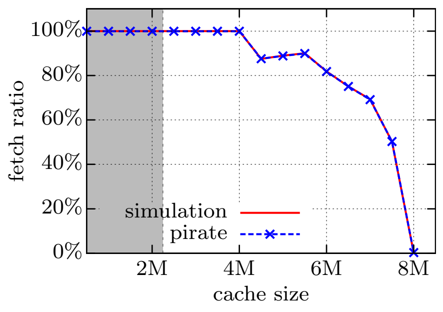
However, the Nehalem cache is sufficiently different from an ideal LRU cache that the space remaining for the Target application does not behave quite the same as a "true" Nehalem cache of that size would. This deviation occurs when the access bits are cleared by the Pirate when access-



(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.

ing 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 4.2.

3.3 Enhancements

3.3.1 *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 rerun 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 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

[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.

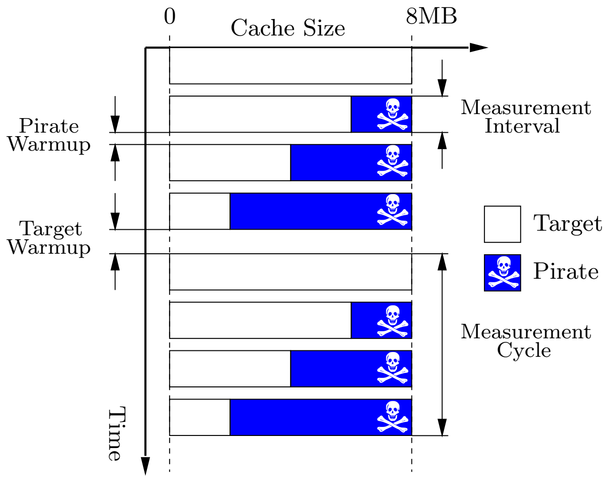


Figure 6: Schedule for dynamic working set adjustment of the Pirate while the Target application executes. During the time between each measurement interval, the application (Target or Pirate) whose working set size increases is allowed to execute alone to warm up its new cache space while the other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

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.

3.3.2 *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 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.

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.

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 3.3.2). 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.

4.1 Experimental Setup

We have implemented Cache Pirating with both dynamic working set adjustment (Section 3.3.1) and a multithreaded Pirate (Section 3.3.2). 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.

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 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.

4.2.1 *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 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.

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

Table 1: Nehalem Cache Hierarchy

4.2.2 *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 9.1, Figure 12.) 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 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.

[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.

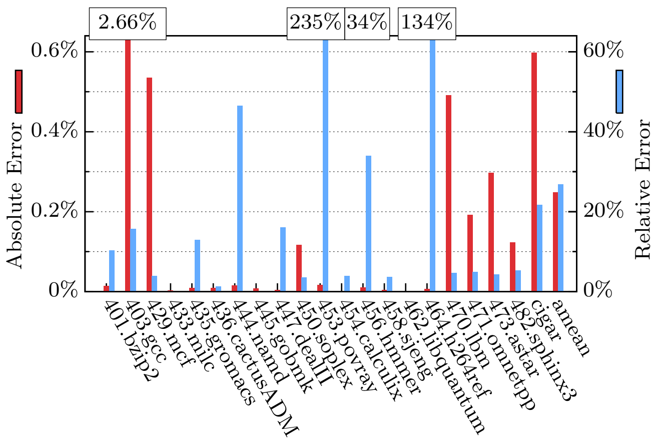


Figure 8: Absolute and relative fetch ratio errors.

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.

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 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

[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.

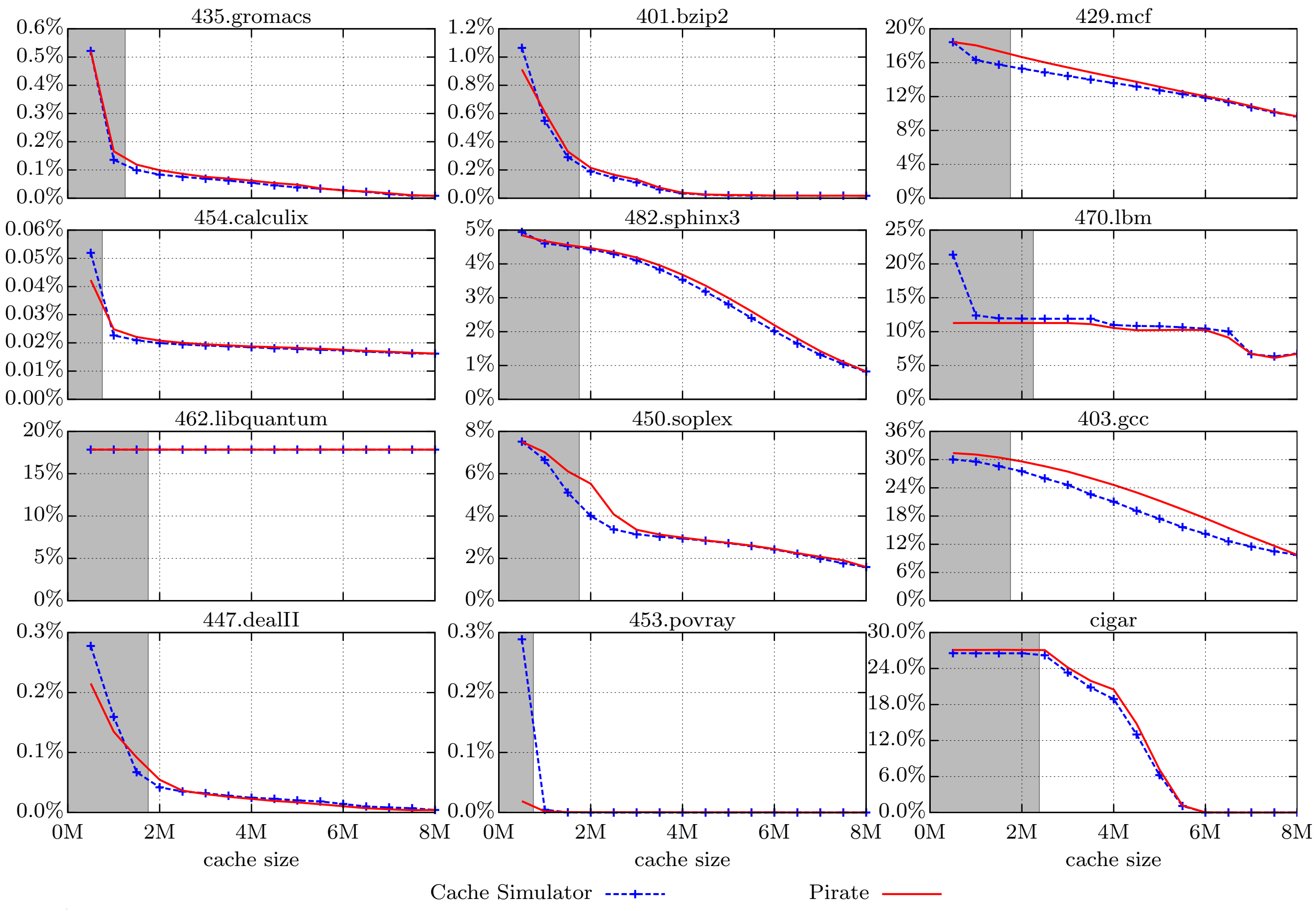


Figure 7: Cache Pirating and reference fetch ratio curves for the benchmarks with the smallest (left), median (middle) and largest (right) errors. The gray regions shows where the Pirate's fetch ratio reached above a threshold (of 1%) which indicates that the Pirate was unable to steal the desired amount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

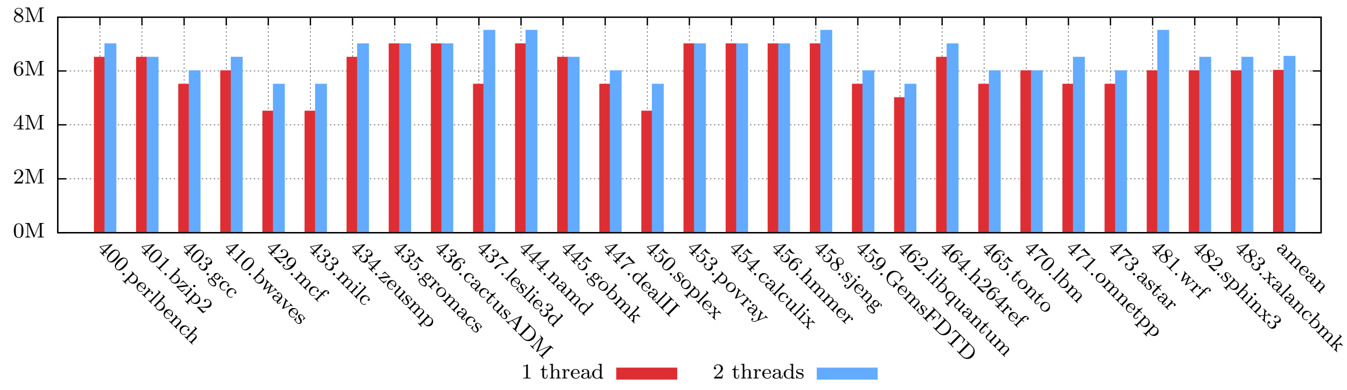


Figure 9: Maximum cache space the Pirate could steal for each application, ignoring the performance impact on the Target of running multiple Pirate threads.

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.

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.

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.

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 4.4.

		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

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

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.

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 9.2, Figure 13 and 14.) 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 compen-

[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.

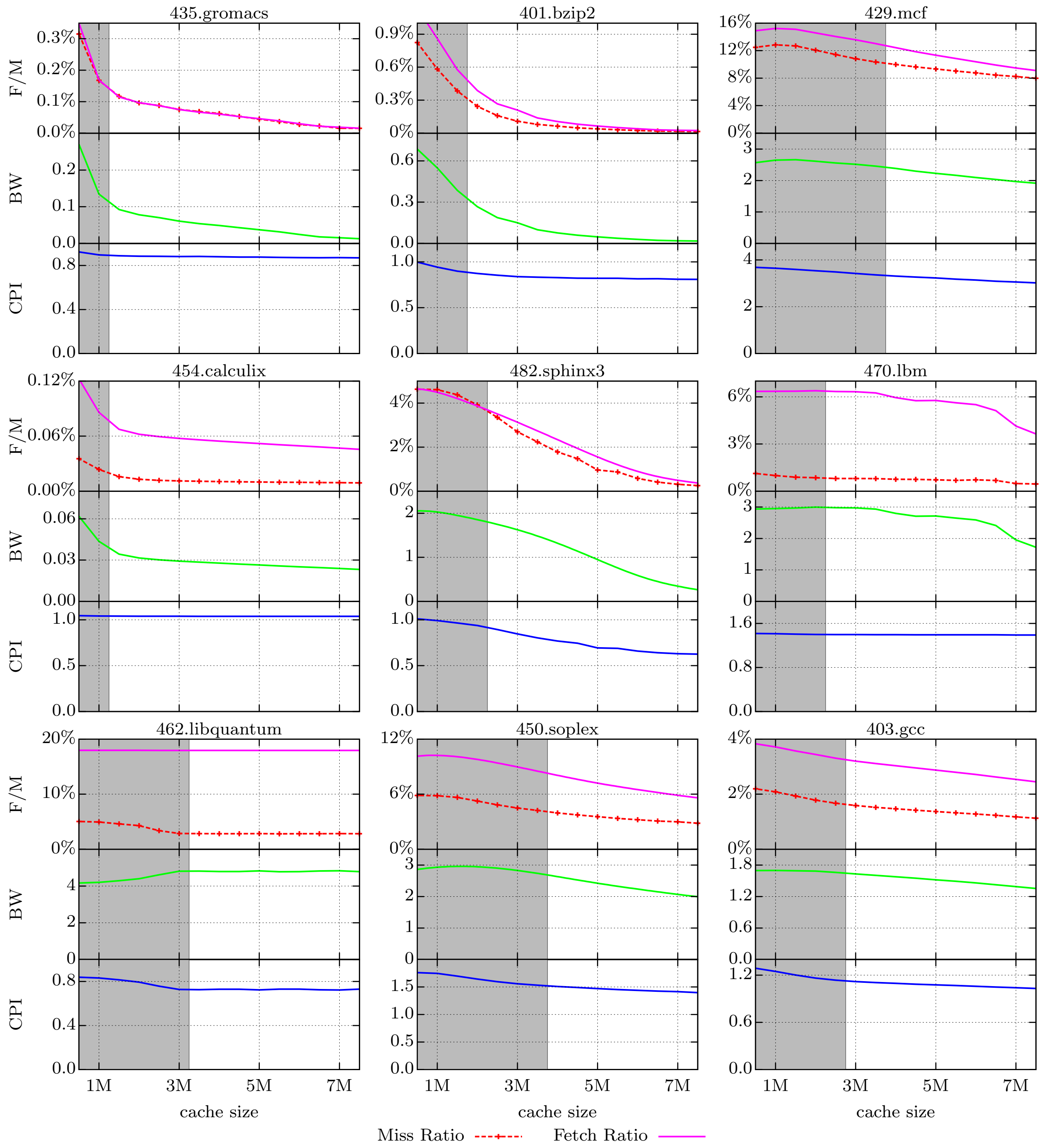


Figure 10: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.

sate. 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%.

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 prefetch memory access for every demand access. 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 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.

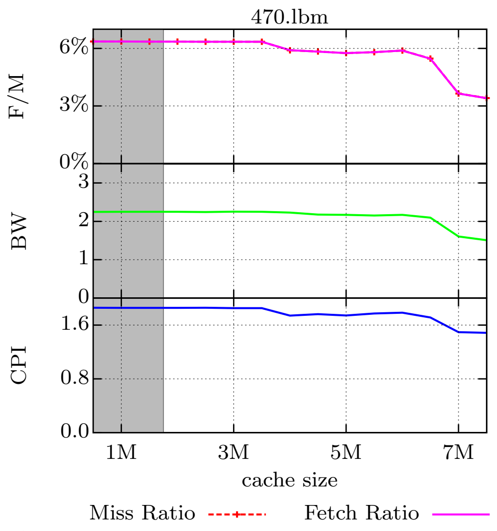


Figure 11: Performance data for 470.lbm with hardware prefetching disabled. (Fetch ratio and miss ratio are identical.)

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 bandwidth 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.

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 3.2.2. 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.

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.

8. 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.

9. APPENDIX

9.1 Simulation Results

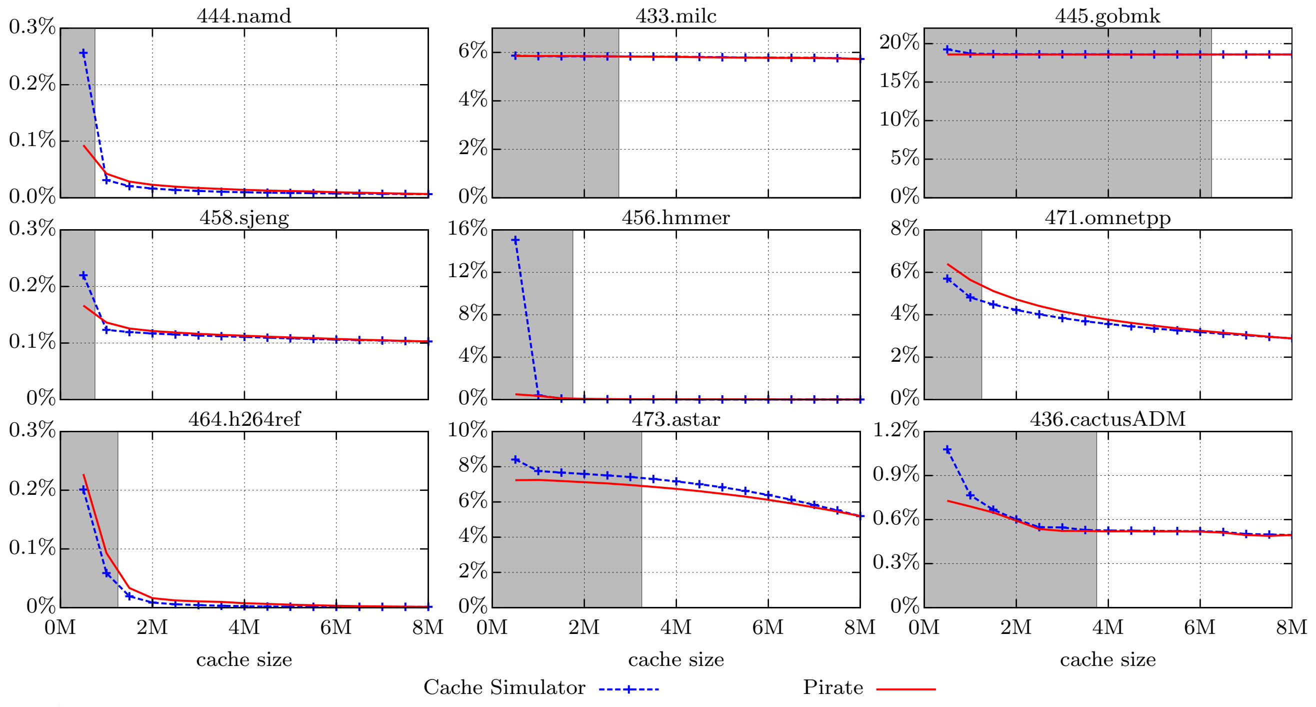


Figure 12: Cache Pirating and reference fetch ratio curves for the remaining benchmarks. See Figure 7 for further discussion.

9.2 Pirate results

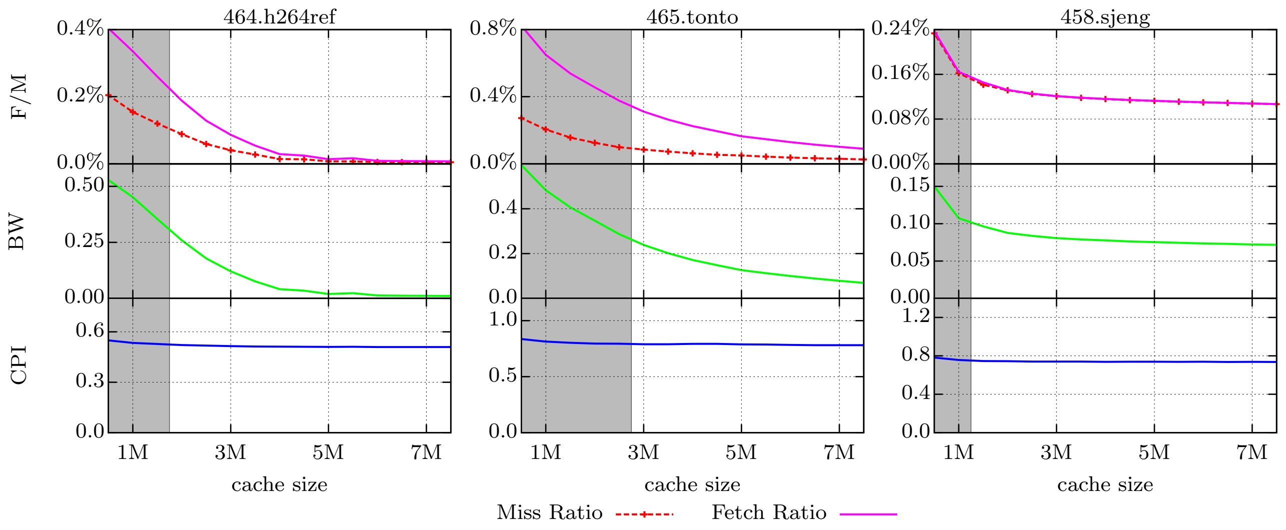


Figure 13: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.

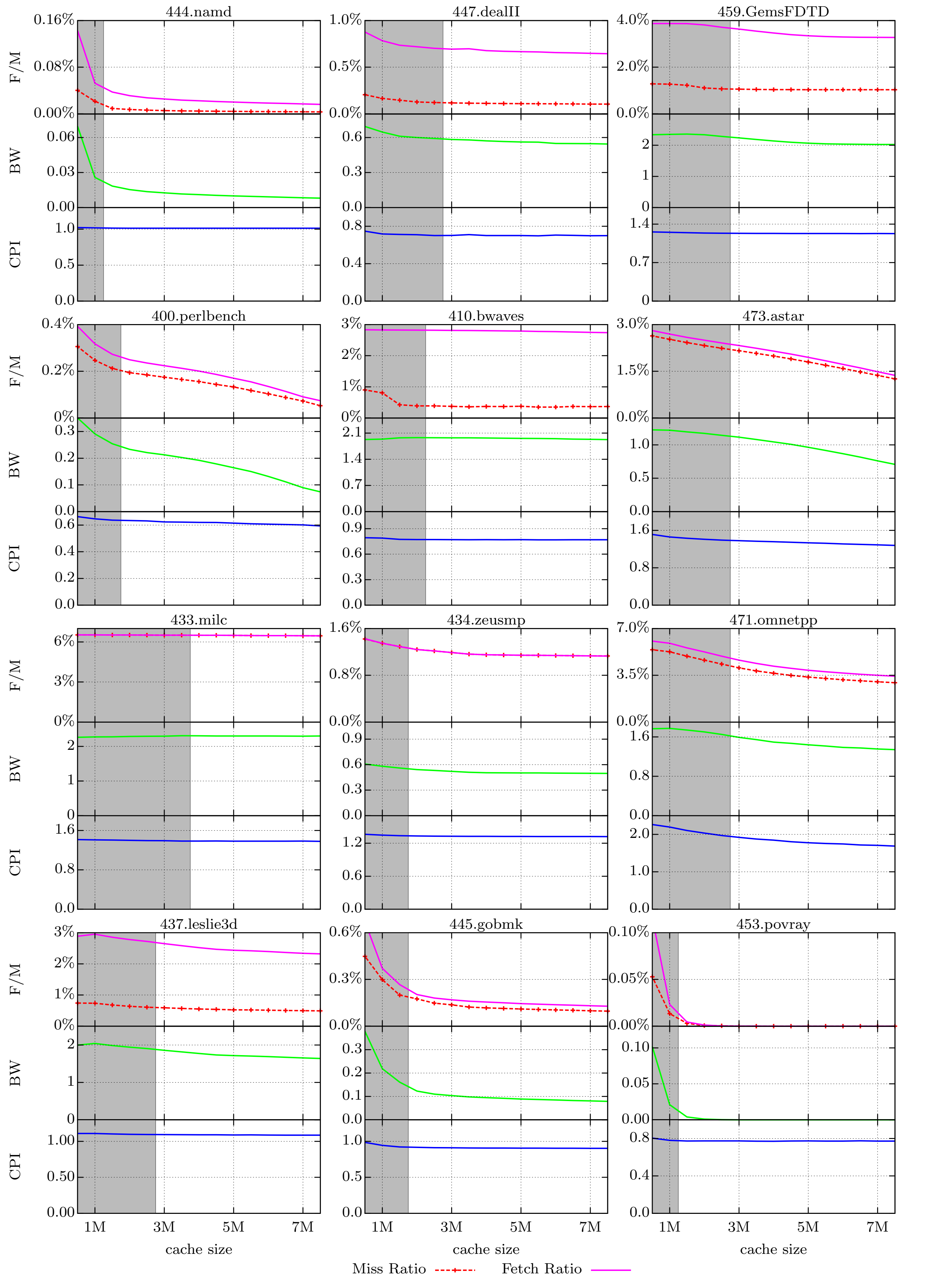


Figure 14: Performance (CPI), bandwidth requirements (BW) in GB/s, and fetch/miss ratios (F/M) for several benchmarks. This data was collected with hardware prefetching enabled.