Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture [*]

Seongbeom Kim, Dhruba Chandra and **Yan Solihin**
Dept. of Electrical and Computer Engineering
North Carolina State University
{skim16,dchandr,solihin}@ncsu.edu

Abstract

This paper presents a detailed study of fairness in cache sharing between threads in a chip multiprocessor (CMP) architecture. Prior work in CMP architectures has only studied throughput optimization techniques for a shared cache. The issue of fairness in cache sharing, and its relation to throughput, has not been studied. Fairness is a critical issue because the Operating System (OS) thread scheduler's effectiveness depends on the hardware to provide fair cache sharing to co-scheduled threads. Without such hardware, serious problems, such as thread starvation and priority inversion, can arise and render the OS scheduler ineffective.

This paper makes several contributions. First, it proposes and evaluates five cache fairness metrics that measure the degree of fairness in cache sharing, and shows that two of them correlate very strongly with the execution-time fairness. Execution-time fairness is defined as how uniform the execution times of co-scheduled threads are changed, where each change is relative to the execution time of the same thread running alone. Secondly, using the metrics, the paper proposes static and dynamic L2 cache partitioning algorithms that optimize fairness. The dynamic partitioning algorithm is easy to implement, requires little or no profiling, has low overhead, and does not restrict the cache replacement algorithm to LRU. The static algorithm, although requiring the cache to maintain LRU stack information, can help the OS thread scheduler to avoid cache thrashing. Finally, this paper studies the relationship between fairness and throughput in detail. We found that optimizing fairness usually increases throughput, while maximizing throughput does not necessarily improve fairness. Using a set of co-scheduled pairs of benchmarks, on average our algorithms improve fairness by a factor of 4×, while increasing the throughput by 15%, compared to a non-partitioned shared cache.

1. Introduction

In a Chip Multi-Processor (CMP) architecture, the L2 cache and its lower memory hierarchy components are typically shared by multiple processors to maximize resource utilization and avoid costly resource duplication [9]. Unfortunately, the cache contention due to cache sharing between multiple threads that are co-scheduled on different CMP processors can adversely impact throughput and fairness. Throughput measures the combined progress rate of all the co-scheduled threads, whereas fairness measures how uniformly the threads are slowed down due to cache sharing.

Prior work in CMP architectures has ignored fairness and focused on studying throughput and its optimization techniques on a shared L2 cache [19, 10]. In Simultaneous Multi-Threaded (SMT) architectures, where typically the entire cache hierarchy and many processor resources are shared, it has been observed that throughput-optimizing policies tend to favor threads that naturally have high IPC [15], hence sacrificing fairness. Some studies have proposed metrics that, if optimized, balances between throughput and fairness [12, 15].

This paper studies fairness in L2 cache sharing in a CMP architecture and its relation to throughput. In contrast to prior work in CMP and SMT, we pursue fairness as the main (and separate) optimization goal. Fairness is a critical aspect to optimize because the Operating System (OS) thread scheduler's effectiveness depends on the hardware to provide fairness to all co-scheduled threads. An OS enforces thread priorities by assigning timeslices, i.e., more timeslices to higher priority threads. However, it assumes that in a given timeslice, the resource sharing uniformly impacts the rates of progress of all the co-scheduled threads. Unfortunately, we found that the assumption is often unmet because a thread's ability to compete for cache space is determined by its temporal reuse behavior, which is often very different compared to that of other threads which are co-scheduled with it.

When the OS' assumption of fair hardware is not met, there are at least three problems that can render the OS scheduler ineffective. The first problem is *thread starvation*, which happens when one thread fails in competing for sufficient cache space necessary to make satisfactory forward progress. The second problem is *priority inversion*, where a higher priority thread achieves a slower forward progress than a lower priority thread, despite the attempt by the OS to provide more timeslices to the higher priority thread. This happens when the higher priority thread loses

[*]This work is supported in part by the National Science Foundation through Faculty Early Career Development Award (CCF-0347425), and North Carolina State University.

to the lower priority thread (or other threads) in competing for cache space. To make things worse, the operating system is not aware of this problem, and hence cannot correct this situation (by assigning more timeslices to the higher priority thread). The third problem is that the forward progress rate of a thread is highly dependent on the thread mix in a co-schedule. This makes the forward progress rate difficult to characterize or predict, making the system behavior unpredictable. Unfortunately, despite these problems, cache implementations today are thread-blind, producing unfair cache sharing in many cases.

To avoid these problems, ideally the hardware should provide *fair caching*, i.e. a scheme that guarantees that the impact of cache sharing is uniform for all the co-scheduled threads. With fair caching, the OS can mostly abstract away the impact of cache sharing, and expect priority-based timeslice assignment to work as effectively as in a time-shared single processor system.

This paper makes several contributions. First, it proposes and evaluates five cache fairness metrics that measure the degree of fairness in cache sharing, and shows that two of them correlate very strongly with the execution-time fairness. Execution-time fairness is defined as how uniform the execution times of co-scheduled threads are changed, where each change is relative to the execution time of the same thread running alone. Secondly, using the metrics, the paper proposes static and dynamic L2 cache partitioning algorithms that optimize fairness. The dynamic partitioning algorithm is easy to implement, requires little or no profiling, has low overhead, and does not restrict the cache replacement algorithm to LRU. The static algorithm, although requiring the cache to maintain LRU stack information, can help the OS thread scheduler to avoid cache thrashing. Finally, this paper studies the relationship between fairness and throughput in detail. We found that optimizing fairness usually increases throughput, while maximizing throughput does not necessarily improve fairness. This is because throughput may improve at the expense of fairness, for example by favoring some threads whose throughput is easy to improve, over others. Using a set of co-scheduled pairs of benchmarks, our fair caching algorithms improve fairness by 4×, while increasing the throughput (combined instructions per cycle) by 15%, compared to a non-partitioned shared cache. The throughput improvement is slightly better than a scheme that minimizes the total number of cache misses as its main objective.

The rest of the paper is organized as follows. Section 2 discusses fairness in cache sharing in greater details and presents our metrics to measure fairness. Section 3 presents our fair caching algorithms. Section 4 details the evaluation setup, while Section 5 presents and discusses the evaluation results. Section 6 describes related work. Finally, Section 7 summarizes the findings.

2. Fairness in Cache Sharing

Cache implementation today is thread-blind, i.e. it applies the same placement and replacement decisions to any cache line, regardless of which thread the line belongs to. To exploit temporal locality in a program, most caches use a Least Recently Used (LRU) replacement algorithm or its approximation. Since temporal locality behavior is application-specific, when two threads from different applications share a cache, the LRU algorithm tends to favor one application over the other. This section discusses the impact of unfair cache sharing (Section 2.1), the conditions in which unfair cache sharing may occur (Section 2.2), and formally defines fairness and proposes metrics to measure it (Section 2.3).

2.1. Impact of Unfair Cache Sharing

To illustrate the impact of cache sharing, Figure 1 shows *gzip*'s number of cache misses per instruction and instruction per cycle (IPC), when it runs alone compared to when it is co-scheduled with different threads, such as *applu*, *apsi*, *art*, and *swim*. All the bars are normalized to the case where *gzip* is running alone. The figure shows that *gzip*'s number of cache misses per instruction increases significantly compared to when it runs alone. Furthermore, the increase is very dependent on the application that is co-scheduled with it. For example, while *gzip*'s cache miss per instruction increases by only 3× when it runs with *apsi*, it increases by 9.5× when it runs with *art* and 7.3× when it runs with *swim*. As a result, the IPC is affected differently. It is reduced by 35% when *gzip* runs with *apsi*, but reduced by 63% when *gzip* runs with *art*. Although not shown in the figure, *art*, *apsi*, *applu*, and *swim*'s cache miss per instruction increases less than 15% when each of them runs with *gzip*. This creates a very unfair cache sharing.

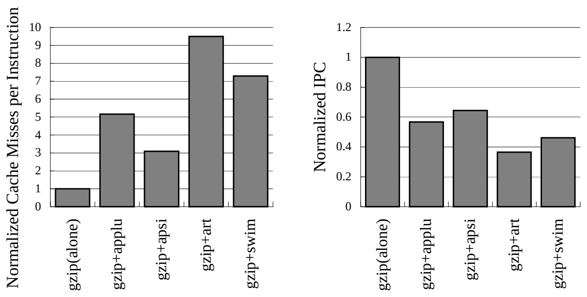


Figure 1: *gzip*'s number of cache misses per instruction and instruction per cycle (IPC), when it runs alone compared to when it is co-scheduled with another thread on a 2-processor CMP, sharing an 8-way associative 512-KB L2 cache.

There are several important things to note. In terms of fairness, *gzip*'s significant slow down can easily result in priority inversion. For example, if *gzip* has a higher priority than *art*, for *gzip* to achieve a higher progress rate, it has to be assigned more than three times the number of time slices compared to that assigned to *art*. Otherwise, to the end users, *gzip* may appear to be starved. In terms of throughput, *gzip*'s significant slow down reduces the overall throughput because the utilization of the processor where *gzip* runs on is also significantly reduced. Therefore, improving fairness in cache sharing is likely to also improve the overall throughput of the system.

Finally, fair caching alone cannot always prevent cache thrashing. Fair caching merely equalizes the impact of cache sharing to all the threads. However, it is still possible that the co-scheduled threads' working sets severely overflow the cache and create a thrashing condition. Therefore, although the hardware should provide fair caching, the OS thread scheduler still needs a tool that can help it to judiciously avoid co-schedules that cause cache thrashing.

2.2. Conditions for Unfair Cache Sharing

To illustrate why some threads such as *gzip* are prone to suffer from a large increase in the number of cache misses, it is important to analyze its temporal reuse behavior, obtained by stack distance profiling [1] [13, 2, 19, 11].

Stack Distance Profiling. For an A-way associative cache with LRU replacement algorithm, there are $A + 1$ counters: $C_1, C_2, \ldots, C_A, C_{>A}$. On each cache access, one of the counters is incremented. If it is a cache access to a line in the i^{th} position in the LRU stack of the set, C_i is incremented. Note that our first line in the stack is the most recently used line in the set, and the last line in the stack is the least recently used line. If it is a cache miss, the line is not found in the LRU stack, resulting in incrementing the miss counter $C_{>A}$. Stack distance profile can easily be obtained statically by the compiler [2], by simulation, or by running the thread alone in the system [19]. It is well known that the number of cache misses for a smaller cache can be easily computed using the stack distance profile. For example, for a smaller cache that has A' associativity, where $A' < A$, the new number of misses can be computed as:

$$miss = C_{>A} + \sum_{i=A'+1}^{A} C_i \tag{1}$$

For our purpose, since we need to compare stack distance profiles from different applications, it is useful to take the counter's frequency by dividing each of the counter by the number of processor cycles in which the profile is collected (i.e., $Cf_i = \frac{C_i}{CPUcycle}$). Furthermore, we call $Cf_{>A}$ as the *miss frequency*, denoting the frequency of cache misses in CPU cycles. We also call the sum of all other counters, i.e. $\sum_{i=1}^{A} Cf_i$ as *reuse frequency*.

[1] Also referred to as the marginal gain in [19].

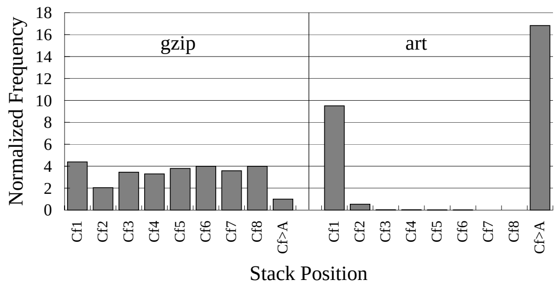


Figure 2: Stack distance frequency profile of *gzip* and *art* on an 8-way associative 512-KB cache, collected by running *gzip* and *art* alone. The bars are normalized to *gzip*'s cache miss frequency ($Cf_{>A}$ bar).

Figure 2 shows the stack distance frequency profiles of *gzip* and *art*. The profiles show that the temporal reuse behavior of *gzip* is such that it reuses many lines that it has accessed in the past, whereas *art* only reuses a few most recently used lines. Therefore, they suffer differently from L2 cache sharing. For example, if the cache space for *art* is reduced by a half, its number of misses will increase by only 0.1%, whereas *gzip*'s number of misses will increase by $16\times$! In fact, even a slight reduction in *gzip*'s cache space increases its cache misses significantly.

2.3. Defining and Measuring Fairness

To measure fairness, let us first define it. Let $Tded_i$ denote the execution time of thread i when it runs alone with a dedicated cache, and $Tshr_i$ denote its execution time when it shares the cache with other threads. When there are n threads sharing the cache and assuming that the threads are always co-scheduled for their lifetime, an ideal fairness is achieved when:

$$\frac{Tshr_1}{Tded_1} = \frac{Tshr_2}{Tded_2} = \ldots = \frac{Tshr_n}{Tded_n} \tag{2}$$

which we refer to as the *execution time fairness* criteria. The criteria can be met if, for any pair of threads i and j that are co-scheduled, the following metric (M_0^{ij}) is minimized:

$$M_0^{ij} = |X_i - X_j|, \text{ where } X_i = \frac{Tshr_i}{Tded_i} \tag{3}$$

Obviously, it is difficult to measure $Tshr_i$ in reality because of the lack of reference points in the execution where the execution time of the shared and dedicated cache cases can be collected and compared. To enforce fairness, a metric that is easier to measure, and one that highly correlates with M_0, is needed. Since we deal with an L2 cache sharing policy, the metric should only be impacted by and dependent on the L2 cache sharing policy, so that it can be reliably used as an input to the fair cache sharing algorithms. For example, instructions per cycle (IPC) is not a good metric for measuring L2 cache sharing fairness because it is highly dependent on many external factors, such as the available ILP, different types of processor hazards, branch mispredictions, L1 data and instruction caches, TLBs, etc. How responsive the IPC is to the L2 cache sharing policy is very application-specific, can have a large range of values, and is therefore unreliable. In addition, IPC is prone to change in response to power saving techniques, such as frequency and voltage scaling, fetch gating, etc.

We propose five L2 cache fairness metrics that are directly related to the L2 cache performance and are insensitive to external factors, while at the same time easy to measure. Let $Miss$ and $Missr$ denote the number of misses and miss rates, respectively. For any pair of co-scheduled threads i and j, the following metrics measure the degree of fairness between a thread pair:

$$\begin{aligned}
M_1^{ij} &= |X_i - X_j|, \text{ where } X_i = \frac{Miss\_shr_i}{Miss\_ded_i} && (4)\\
M_2^{ij} &= |X_i - X_j|, \text{ where } X_i = Miss\_shr_i && (5)\\
M_3^{ij} &= |X_i - X_j|, \text{ where } X_i = \frac{Missr\_shr_i}{Missr\_ded_i} && (6)\\
M_4^{ij} &= |X_i - X_j|, \text{ where } X_i = Missr\_shr_i && (7)\\
M_5^{ij} &= |X_i - X_j|, \text{ where}\\
& \quad X_i = Missr\_shr_i - Missr\_ded_i && (8)
\end{aligned}$$

By minimizing M_x^{ij}, a fair caching algorithm seeks to improve fairness. When there are more than two threads, the fairness metrics can be summed or averaged over all possible pairs, and the resulting metric, e.g. $M_x = \sum_i \sum_j M_x^{ij}$, becomes the target for minimization. Metric M_1 tries to equalize the ratio of miss increase of each thread, while M_2 tries to equalize the number of misses. Similarly, metric M_3 tries to equalize the ratio of miss rate increase of each thread, while M_4 tries to equalize the miss rates. Finally, metric M_5 tries to equalize the increase in the miss rates of each thread.

Metrics M_2 and M_4 enforce fairness in absolute terms, that is, the number of misses, or the miss rates under sharing are equalized. They may over-penalize applications with few misses or low miss rates. In M_1 and M_3, the number of misses and miss rates are normalized by those from a dedicated L2 cache, so that the increase of misses or miss rates is proportional to ones from the dedicated cache case. The normalization makes sense because, when an application suffers many misses (or have high miss rates), it is likely that many of these misses are overlapped with each other. Therefore, increasing them has a smaller impact on the execution time of the application, compared to applications with few misses or low miss rates. Finally, the metric M_5 adjusts the miss rates by subtracting them with the miss rates on the dedicated cache. At this point, it seems that M_1, M_3, and M_5 are better metrics because they enforce fairness in relative terms with respect to the dedicated cache case.

The remaining issue for the metrics is how well they correlate with the execution time fairness metric M_0 (Equation 3). To find out, we compute the statistical correlation [5] between M_0 and M_i, where $i = 1, 2, \ldots, 5$:

$$\begin{aligned}
Corr(M_i, M_0) &= \frac{Cov(M_i, M_0)}{\sigma(M_i)\,\sigma(M_0)}, \text{where} && (9)\\
Cov(M_i, M_0) &= E(M_i\, M_0) - E(M_i)\, E(M_0)
\end{aligned}$$

where $\sigma(M_i)$ and $E(M_i)$ denote the standard deviation and expected value of M_i. Since we are interested in how the L2 cache sharing policy impacts both the execution time fairness metric (M_0) and other fairness metrics (M_i), we can obtain one data point for each L2 sharing policy. For example, by partitioning the L2 cache at n different partition sizes, we obtain n data points, that can be used to compute $Corr(M_i, M_0)$. The value of $Corr(M_i, M_0)$ ranges from -1 to 1, where 1 indicates a perfect correlation, 0 indicates no correlation, and -1 indicates negative correlation. A perfect correlation between M_i and M_0 indicates that M_i can be used in place of M_0 to guide a fair caching policy.

3. Cache Partitioning

This section discusses the hardware support for partitionable caches based on existing techniques (Section 3.1). Then it presents the proposed static algorithms (Section 3.2) and dynamic algorithms (Section 3.3) for finding the partitions that optimize fairness.

In Section 3.1, we adapt the *modifi ed LRU* hardware support for partitionable caches from Suh, et al. [19]. Although we use similar hardware support, our partitioning algorithms (Section 3.2 and 3.3) are different in that while they optimize throughput, we optimize fairness. In addition, while their dynamic partitioning algorithms rely on the cache to implement true LRU replacement algorithm, our dynamic partitioning algorithms can work with any replacement algorithms. This is important because the L2 cache often implements pseudo-LRU replacement algorithms (such as in [3]), partly due to the complexity of implementing LRU replacement for highly associative caches, and partly due to a small performance difference between LRU and random replacement in large and highly associative caches [7]. Therefore, to be useful, a cache partitioning algorithm should work with pseudo-LRU replacement algorithms.

3.1. Hardware Support for Partitionable Caches

In general, hardware support for partitionable caches can be categorized into two approaches. The first approach relies on modifying the cache placement algorithm by restricting where data can be placed in the cache [14, 10].

This approach relies on configurable cache hardware or programmable partition registers. The drawbacks of this approach are that it modifies the underlying cache hardware, may increase the cache access time due to having to locate the correct partition, and makes the cache unavailable during reconfiguration.

An alternative approach is to modify the cache replacement algorithm [19]. In this approach, partitioning the cache is incremental: on each cache miss, we can reallocate a cache line from another thread to the thread that suffers the cache miss by selecting the line for replacement. Because a replacement only occurs on cache misses, this approach does not add to cache access time. In addition, selecting a line to be replaced can be overlapped with the cache miss latency. Finally, due to the incremental repartitioning nature, the repartitioning does not make the cache unavailable. It involves writing to a counter the target number of lines and tracking the current number of lines already allocated to a thread. In this paper, we apply the second approach.

In [19], the cache is augmented with two types of counters: one that keeps track of the current number of lines allocated to each processor or thread j ($curAlloc_j$), and the target number of lines for the thread ($targetAlloc_j$). Initially, or after each repartitioning, $targetAlloc_j$ may differ from $curAlloc_j$. If $curAlloc_j > targetAlloc_j$, j has too many lines allocated to it (over-allocation). If $curAlloc_j < targetAlloc_j$, j has too few lines allocated to it (under-allocation). To achieve the target allocation, when a thread j suffers a cache miss, we check whether j is over- or under-allocated. If j has the right allocation, or is over-allocated, one of j's line in the same set as the missed address is selected for replacement. If, however, j is under-allocated, we need to add more lines to it. To achieve that, a line in the same set that belongs to another thread that is over-allocated (say, thread k) is selected for replacement. Then, the current allocations are updated: for thread k it is decremented, and for thread j it is incremented. If a line that belongs to another thread cannot be found, a line that belongs to j is selected for replacement. With this algorithm, each set may be partitioned differently, although the sum of partitions over all sets for each thread is the same as the thread's target allocation. Programming a new partition is accomplished by writing to the target allocation register of each thread.

One parameter of the cache is the partition granularity, the smallest unit of partition that can be reallocated. Ideally, taking into account the number of cores, it should be as coarse-grain as possible without sacrificing the ability to enforce fairness. For 2-way CMP, we found that a partition granularity equal to the total cache space divided by its associativity, works well. For example, for an 8-way 512 KB L2 cache, the granularity would be 64 Kbytes.

3.2. Static Fair Caching

Our static fair caching algorithm partitions the cache based on the stack distance profile of an application, collected through a profiling run. The stack distance profile can only be collected when the cache uses an LRU replacement policy, or at least records the LRU stack information. Note that we only need to perform one profiling run per application.

For static partitioning, we first run each application and collect its global stack distance frequency profile described in Section 2.2. Let $P =< p_1, p_2, \ldots, p_n >$, $\sum p_i = cacheSize$, denote a partition. Then, for each thread i and partition P, we apply Equation 1, using $A' = \frac{p_i}{numSet \times lineSize}$ to obtain the expected new misses ($Miss\_shr_i$) and miss rate ($Missr\_shr_i$) under cache sharing. After that, for each possible partition, we choose the partition that minimizes the chosen metric M_x, and apply the partition to the cache for the duration of the co-schedule. The partition needs to be re-evaluated after a context switch because the thread mix may have changed.

The benefit of static partitioning is that it can be used as an input to the OS thread scheduler to avoid co-scheduling threads that will cause cache thrashing. As discussed in 2.1, cache partitioning alone cannot prevent all cases of cache thrashing. The drawback is that static partitioning relies on the cache to implement LRU replacement algorithm and is unable to adapt to applications' dynamic behavior.

3.3. Dynamic Fair Caching

The dynamic partitioning algorithm consists of three parts: initialization, rollback, and repartitioning. *Initialization* is performed only once when a co-schedule is formed. Then, at the end of each time interval *t*, the *rollback step*, immediately followed by the *repartitioning step*, are invoked. The goal of the rollback step is to reverse a repartitioning decision that has not been beneficial, i.e. when the cache miss rates of the thread that was given a larger partition in the prior time interval has not improved. In the first time interval, since no repartitioning has been performed, the rollback step performs nothing. All threads that are not rolled back are considered for repartitioning in the repartitioning step. The goal of this step is to increase or decrease a thread's partition such that a better fairness can be achieved in the next time interval. At the end of the steps, both the rollback and repartitioning decisions are programmed into the cache and applied to the next time interval $t + 1$. The time interval period needs to be large enough so that partition changes can take full effect in a single time interval, and that the cost of invoking the algorithm is relatively small. However, since the cache never becomes unavailable when the algorithm is invoked, the algorithm latency is not much of a critical issue.

Initialization.
Initialize partition equally, $p_i = \frac{cacheSize}{n}$, where n is the number of cores or threads sharing the cache. Apply that for the first interval period.

In the rollback step, the *considered set* is initialized to contain all the co-scheduled threads, indicating that by default all of them will be considered for repartitioning (Step 1). Then each thread i is checked whether it has received a larger partition in the time interval that just completed (t), indicated by a repartitioning assignment A_{ij} being a member of the repartition assignment set AS, which was recorded in the prior time interval by the repartitioning step. For such a thread, if its new miss rate has not decreased by at least $T_{rollback}$, the prior repartitioning assignment is deemed ineffective, and therefore it is rolled back to what it was before that (Step 2). In addition, both threads i and j are removed from the considered set, so that they will no longer be considered by the repartitioning step.

```
Rollback Step. At the end of an interval period $t$:
  1. Initialize Considered Set $CS = \{1, 2, \ldots, n\}$.
  2. For each thread $i$, if a repartition assignment $A_{ij} \in AS$, where $AS$ is the repartition assignment set, and $Missr\_shr_i^{t-1} - Missr\_shr_i^t \leq T_{rollback}$, roll back the repartitioning assignment:
       • $p_i^{t+1} = p_i^{t-1}$
       • $p_j^{t+1} = p_j^{t-1}$
       • $CS = CS - \{i, j\}$
```

In the repartitioning step, the repartition assignment set AS is emptied (Step 1). Then, for each thread in the considered set, we compute the statistics (X_i) that corresponds to the metric being optimized (Step 2). Since an ideal fairness is achieved when all X_i's have equal values, the next step tries to equalize these values by changing the partition sizes. Two threads in the considered set which have the maximum and minimum X values are repartitioned, if the difference in their X values exceeds a repartition threshold ($T_{repartition}$) (Step 3 and 4). The thread with a larger X value obviously is impacted too much by the partition, and is therefore assigned a larger partition, where the additional partition ($gran$) is taken from the thread with a smaller X value. The repartition assignment is then recorded in the repartition assignment set AS, in case it needs to be rolled back at the end of the next time interval. Both threads are then removed from the considered set (Step 5). Steps 2 to 5 are repeated until all threads in the considered set have been considered for repartitioning (Step 6). Finally, the new repartition assignments (including rolled back repartitions) are applied in the next time interval (Step 7).

```
Repartitioning Step. At the end of an interval period $t$, and right after the rollback step:
  1. Initialize repartition assignment $AS = \{\}$.
  2. For each thread $i \in CS$, compute the statistics related to the fairness metric being optimized. For example, for metric $M_1$, we compute $X_i^t = \frac{Miss\_shr_i}{Miss\_ded_i}$, whereas for $M_3$, we compute $X_i^t = \frac{Missr\_shr_i}{Missr\_ded_i}$.
  3. Find $i_{max} \in CS$ and $i_{min} \in CS$ such that $X_{i_{max}}$ and $X_{i_{min}}$ have the maximum and minimum values, respectively.
  4. If $X_{i_{max}} - X_{i_{min}} > T_{repartition} \geq 0$ then reassign the partition:
       • $p_{i_{max}}^{t+1} = p_{i_{max}}^t + gran$
       • $p_{i_{min}}^{t+1} = p_{i_{min}}^t - gran$
       • $AS = AS \bigcup \{A_{i_{max} i_{min}}\}$
  5. $CS = CS - \{i_{max}, i_{min}\}$.
  6. Repeat step 2 to 5 until $CS = \{\}$.
  7. Apply the partition in the next interval period $t + 1$.
```

The parameters for the algorithms are the partition granularity ($gran$), rollback threshold ($T_{rollback}$), repartitioning threshold ($T_{repartition}$), and the time interval period.

3.4. Dynamic Algorithm Overhead

There are three types of overheads in our dynamic partitioning schemes: profiling overhead, storage overhead, and fair caching algorithm overhead. A single static profiling run per thread is needed to obtain the base miss per cycle (representing miss count) and miss rate, except when the algorithm optimizes the M_2 or M_4 metrics, in which case it does not need any profiling. Once the base miss frequency and miss rate are obtained, they are used by the dynamic partitioning algorithms as constants. In terms of storage overhead, we need a few registers per thread to keep track of the miss count or miss rate of the current and prior time intervals. Finally, we assume that the fair caching algorithm is implemented in hardware, and therefore does not incur any algorithm overhead. However, we have also measured the overhead of a software implementation of the algorithm for a 2-processor CMP and found that the algorithm takes less than 100 cycles to run per invocation, making a software implementation a feasible alternative. Furthermore, since the algorithm invocation does not make the cache unavailable and can be overlapped with the application execution, its latency can be hidden. Even when compared to the smallest time interval, the overhead is less than 0.01%.

4. Evaluation Setup

Applications. To evaluate the benefit of the cache partitioning schemes, we choose a set of mostly memory-intensive benchmarks: *apsi, art, applu, bzip2, equake, gzip, mcf,* *perlbmk*, *swim*, *twolf*, and *vpr* from the SPEC2K benchmark suite [18]; *mst* from Olden benchmark, and *tree* [1]. We used test input sets for the Spec benchmarks, 1024 nodes for mst, and 2048 bodies for tree. Most benchmarks are simulated from start to completion, with an average of 786 millions instructions simulated.

Miss/Reuse	Concentrated	Flat
High/High		mcf(22%), applu(66%)
High/Low	art(99.8%), swim(77%), equake(85%)	
Low/High	vpr(0.05%), apsi(29%)	tree(7%), gzip(5%), bzip2(18%)
Low/Low	perlbmk(60%), twolf(3%)	mst(67%)

Table 1: The applications used in our evaluation.

Table 1 categorizes the benchmarks based on their miss frequency, reuse frequency, and the shape of their stack distance (Concentrated vs. Flat). The L2 miss rates of the applications are shown in parenthesis. A benchmark is categorized into the high miss frequency (or high reuse frequency) categories if the number of misses (or accesses) per 1000 CPU cycles are more than 2. It is categorized into the "Flat" category if the standard deviation of its global stack distance frequency counters is smaller than 0.1. Otherwise, it is categorized into "Concentrated". Finally, all data is collected for the benchmark's entire execution time, which may differ from when they are co-scheduled. The table shows that we have a wide range of benchmark behavior. These benchmarks are paired and co-scheduled. Eighteen benchmark pairs are co-scheduled to run on separate CMP cores that share the L2 cache. To observe the impact of L2 cache sharing, each benchmark pair is run from the start and is terminated once one of the benchmarks completes execution.

Simulation Environment. The evaluation is performed using a cycle-accurate, execution-driven multiprocessor simulator. The CMP cores are out-of-order superscalar processors with private L1 instruction and data caches, and shared L2 cache and all lower levels of memory hierarchy. Table 2 shows the parameters used for each component of the architecture. The L2 cache replacement algorithm is either LRU or pseudo-LRU. For the pseudo-LRU replacement, we use a random number to select a non-MRU line for replacement.

Algorithm Parameters. The parameters for the fair caching algorithm discussed in Section 3.3 are shown in Table 3. The time interval determines how many L2 cache accesses must occur before the algorithm is invoked. We also vary the rollback threshold ($T_{rollback}$), while the repartition threshold ($T_{repartition}$) is set to 0. Although the partition granularity can be any multiples of a cache line size, we choose 64KB granularity to balance the speed of the algorithm in achieving the optimal partition, and the ability of the algorithm to approximate an ideal fairness.

CMP
2 cores, each 4-issue out-of-order, 3.2 GHz
Int, fp, ld/st FUs: 2, 2, 2
Max ld, st: 64, 48. Branch penalty: 17 cycles
Re-order buffer size: 192
MEMORY
L1 Inst (private): WB, 32 KB, 4 way, 64-B line, RT: 3 cycles
L1 data (private): WB, 32 KB, 4 way, 64-B line, RT: 3 cycles
L2 data (shared): WB, 512 KB, 8 way, 64-B line, RT: 14 cycles
L2 replacement: LRU or pseudo-LRU
RT memory latency: 407 cycles
Memory bus: split-transaction, 8 B, 800 MHz, 6.4 GB/sec peak

Table 2: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*.

Parameter	Values
$gran$	64 KB
Time interval	10K, 20K, 40K, 80K L2 accesses
$T_{rollback}$	0%, 5%, 10%, 15%, 20%, 25%, 30%
$T_{repartition}$	0

Table 3: Dynamic partitioning algorithm parameters.

Dedicated Cache Profiling. Benchmark pairs are run in a co-schedule until a thread that is shorter completes. At that point, the simulation is stopped to make sure that the statistics collected reflect the impact of L2 cache sharing. To obtain accurate dedicated mode profiles, the profile duration should correspond to the duration of the co-schedules. For the shorter thread, the profile is collected for its entire execution in a dedicated cache mode. But for the longer thread, the profile is collected until the number of instructions executed reaches the number of instructions executed in the co-schedule.

5. Evaluation

In this section, we present and discuss several sets of evaluation results. Section 5.1 presents the correlation of the various fairness metrics. Section 5.2 discusses the results for the static partitioning algorithm, while Section 5.3 discusses the results for the dynamic partitioning algorithm. Finally, Section 5.4 discusses how sensitive the dynamic partitioning algorithm is to its parameters.

5.1. Metric Correlation

Figure 3 presents the correlation of L2 cache fairness metrics ($M_1, M_2, \ldots, M_5$) with the execution time fairness met-

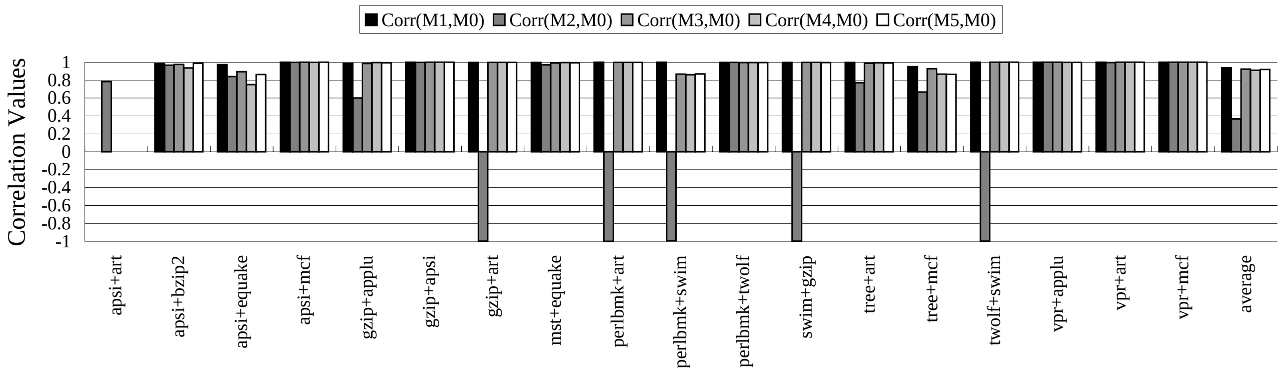


Figure 3: Correlation of five L2 cache fairness metrics (M_1, M_2, . . . , M_5) with the execution time fairness metric (M_0).

ric from Equation 3 (M_0). We collect one data point of fairness metric values for each possible partition. Since the partitioning granularity is 64KB (Section 4), there are seven possible partitions. Once the seven data points are collected, the correlation is computed by applying Equation 9.

The figure shows that on average, M_1, M_3, M_4 and M_5 produce good correlation (94%, 92%, 91%, and 92%, respectively), whereas M_2 produces poor correlation (37%). It is clear that M_1 is the best fairness metric, not only on average, but also across all the benchmark pairs. The second best metric is M_3 where it consistently has a high correlation on all benchmark pairs. Therefore, for the rest of the evaluation, we only consider M_1 and M_3 for our fairness metrics.

M_4 and M_5 occasionally do not correlate as well as M_3, such as for benchmark pairs *apsi+equake* and *tree+mcf*. Zero correlation appears in *apsi+art* for most metrics, showing an anomaly caused by constant values of the metrics regardless of the partition sizes, versus the M_0 metric which produces a very slight change due to external factors such as bus contention. In five benchmark pairs, M_2 produces negative correlation, indicating that it is not a good metric to use for measuring fairness. This is because equalizing the number of misses over-penalizes threads that originally have few misses. Therefore, enforcing M_2 can aggravate unfair cache sharing.

Note that most of the benchmarks tested are L2 cache intensive, in that they access the L2 cache quite frequently. It is possible that benchmarks that do not access the L2 cache much may not obtain a high correlation between the L2 fairness metrics and the execution time fairness. However, arguably, such benchmarks do not need high correlation values because they do not need fair caching. Even when their number of L2 cache misses increases significantly, their execution time will not be affected much.

5.2. Static Fair Caching Results

Figure 4 shows the throughput (combined IPC) on the top chart and M_1 fairness metric value on the bottom chart, for all benchmark pairs and their average, using our static partitioning algorithm. Each benchmark pair shows the results for four schemes. The first bar (*LRU*) is a non-partitioned shared L2 cache with LRU replacement policy. The second bar (*MinMiss*) is a L2 cache partitioning scheme that minimizes the total number of misses of the benchmark pair. *MinMiss* is similar to the scheme in [19], except that *MinMiss* finds such partition statically. The third bar (*FairM1*) is our L2 cache partitioning algorithm that enforces fair caching by minimizing the M_1 metric for each benchmark pair. The last bar (*Opt*) represents the best possible (most fair) partition. *Opt* is obtained by running the benchmark pairs once for every possible partition. The partition that produces the best fairness is chosen for *Opt*. All bars are normalized to *LRU*, except for *apsi+art*, in which case all schemes, including *LRU*, achieve an ideal fairness. For static partitioning, to obtain precise target partitions, the same partition sizes are kept uniform across all the sets.

On average, all schemes, including *MinMiss*, improve the fairness and throughput compared to *LRU*. This indicates that *LRU* produces a very unfair L2 cache sharing, resulting in poor throughput because in many cases one of the co-scheduled threads is significantly slowed down compared to when it runs alone. *MinMiss* increases the combined IPC by 13%. However, it fails to improve the fairness much compared to *LRU*, indicating that maximizing throughput does not necessarily improve fairness.

FairM1 achieves much better fairness compared to both *LRU* and *MinMiss*, reducing the M_1 metric by 47% and 43% compared to *LRU* and *MinMiss*, respectively. Interestingly, *FairM1* achieves a significant increase in throughput (14%) compared to *LRU*, which is even slightly better compared to that of *MinMiss*. This result points that, in most cases, optimizing fairness also increases throughput. This is because

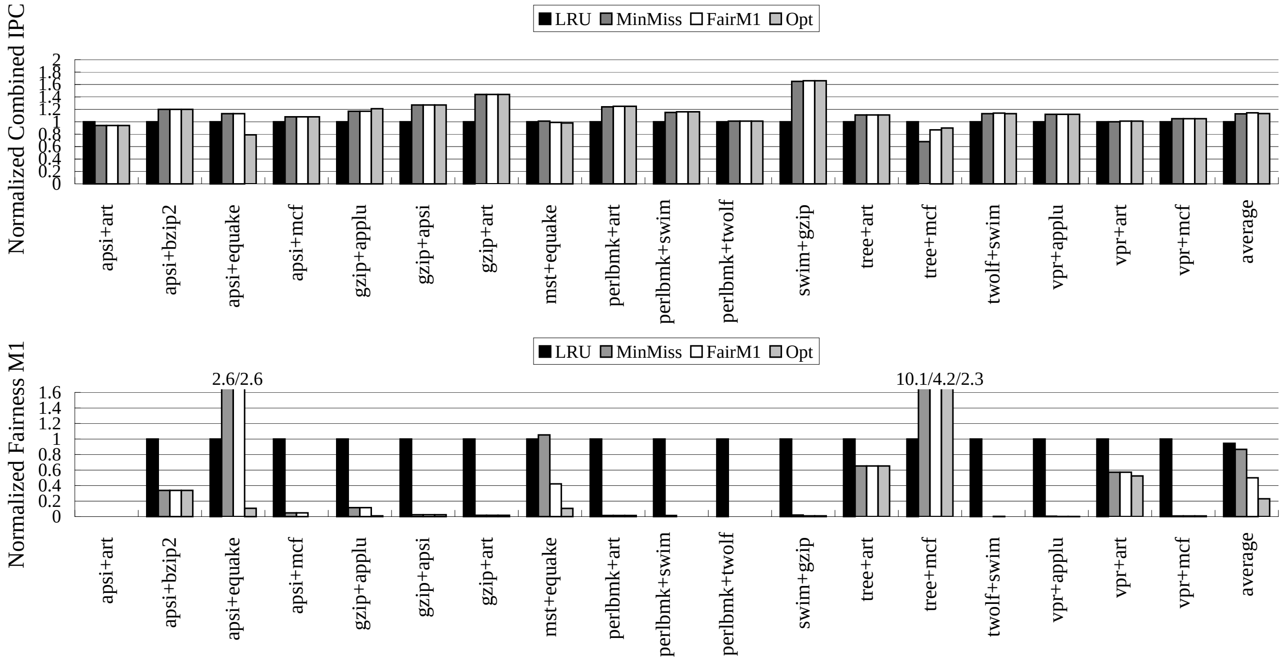


Figure 4: Throughput (top chart) and fairness metric M_1 (bottom chart) of static partitioning algorithms. Lower M_1 values indicate better fairness.

by equalizing the impact of cache sharing on all threads, it avoids the pathological throughput where one thread is starved and its IPC is significantly penalized. The exceptional cases are *apsi+art*, *apsi+equake*, and *tree+mcf*, where even *Opt* yields a lower throughput compared to *LRU*. Out of these three cases, the throughput reduction in *apsi+equake* and *tree+mcf* are due to the limitation of a static partition to adapt to the changing dynamic behavior of the benchmark pairs. They are much improved by the dynamic fair caching algorithms.

5.3. Dynamic Fair Caching Results

Figure 5 shows the throughput (top chart) and fairness results (bottom chart) of our dynamic partitioning algorithm for the benchmark pairs and their average. Similar to Figure 4, throughput and fairness are represented as the combined Instructions Per Cycle (IPC) and metric M_1, respectively. Each benchmark pair shows the result for four schemes. The first bar (*LRU*) is a non-partitioned shared L2 cache with LRU replacement policy. The second bar (*PLRU*) is a non-partitioned shared L2 cache with pseudo-LRU replacement policy described in Section 4. The last three bars (*FairM1Dyn*, *FairM3Dyn*, and *FairM4Dyn*) are our L2 algorithms from Section 3.3 which minimize M_1, M_3, and M_4 metrics, respectively. All the bars are normalized to *PLRU*, which is selected as the base case because it is a more realistic L2 cache implementation, and that unlike the static algorithm, our dynamic algorithms can work with it. *LRU* is the same as in Figure 4 but looks different in the chart because it is normalized to *PLRU*. The results in the figure are obtained using 10K L2 accesses for the time interval period, and 20% for the rollback threshold.

The figure shows that *PLRU* and *LRU* achieve roughly comparable throughput and fairness. *FairM1Dyn* and *FairM3Dyn* improve fairness over *PLRU* significantly, reducing the M_1 metric by a factor of 4 on average (or 75% and 76%, respectively) compared to *PLRU*. This improvement is consistent over all benchmark pairs, except for *FairM3Dyn* on *tree+mcf*. In *tree+mcf*, *PLRU* already achieves almost ideal fairness, and therefore it is difficult to improve much over this.

The figure also confirms an earlier observation where fairness strongly impacts throughput. By achieving better fairness, both algorithms achieve a significant increase in throughput (15%). The throughput improvement is consistent for almost all benchmark pairs. Nine out of eighteen cases show throughput improvement of more than 10%. In *gzip+art*, the throughput increases by almost two times (87%). The only noticeable throughput decrease is in *apsi+art* and *tree+mcf*, where *FairM1Dyn* reduces the throughput by 11% and 4%, respectively. In those cases, the fairness is improved (M_1 is reduced by 89% and 33%, respectively). This implies that, in some occasions, optimizing fairness may reduce throughput.

Since the height of the *LRU* bar in the figure is almost the same as in Figure 4 (0.94 vs. 1.07), we can approximately compare how the dynamic partitioning algorithms perform with respect to the static partitioning algorithms. Comparing the two figures, it is clear that the dynamic partitioning

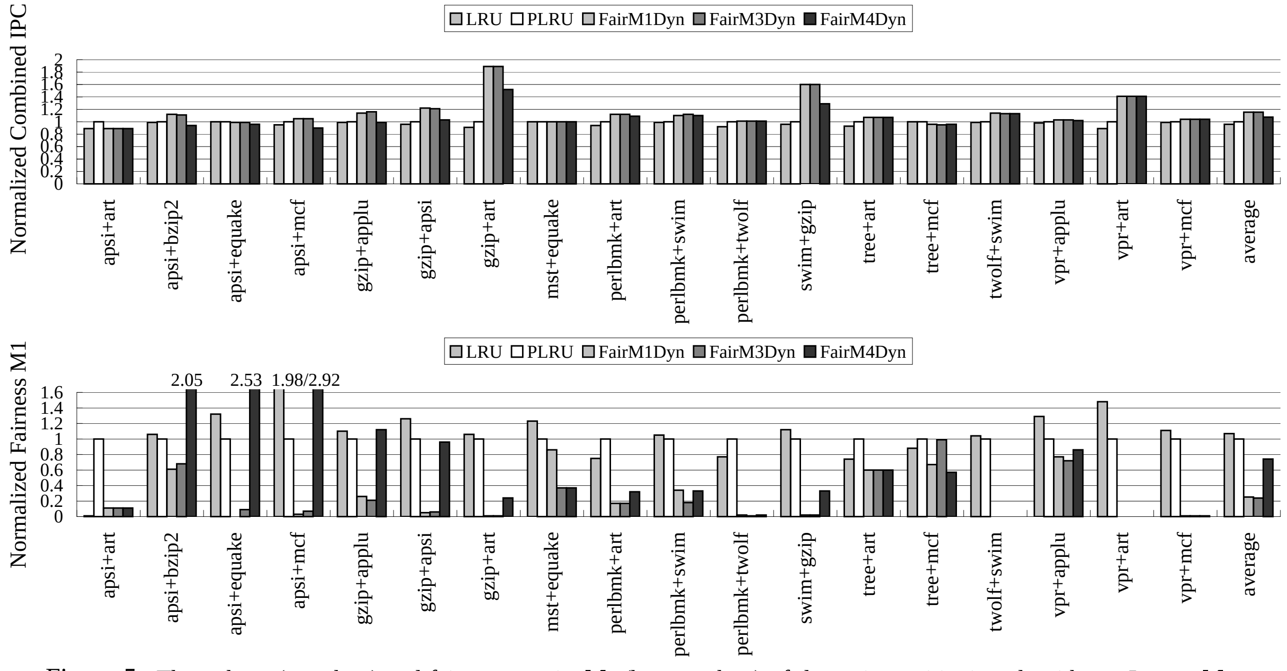


Figure 5: Throughput (top chart) and fairness metric M_1 (bottom chart) of dynamic partitioning algorithms. Lower M_1 values indicate better fairness.

algorithms (*FairM1Dyn* and *FairM3Dyn*) achieve better fairness compared to *FairM1* (0.25 and 0.24 vs. 0.5) and even achieve a slightly better average throughput. This is nice because compared to *FairM1*, the dynamic partitioning algorithms do not require stack distance profiling or rely on an LRU replacement algorithm.

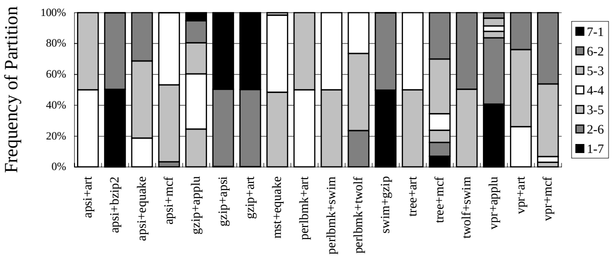


Figure 6: The distribution of partitions for *FairM1Dyn*.

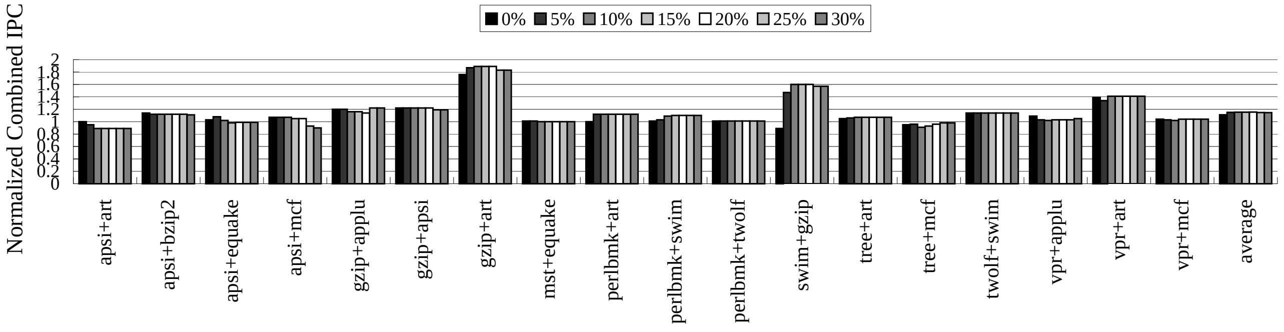
Finally, *FairM4Dyn* gives moderate throughput and fairness improvement. It improves throughput by 8% and reduce M_1 metric value by 26%. Although it is worse than *FairM1Dyn* and *FairM3Dyn*, it may be attractive due to not requiring any profiling information. Also note that *FairM4Dyn*'s pathological cases seem to be isolated to only a few benchmark pairs in which one of the threads is *apsi*, which suffers from a very high L2 miss rate. Therefore, *FairM4Dyn* is a promising algorithm that needs to be investigated more thoroughly.

Figure 6 shows the fraction of all time intervals where each different partition is applied, for each benchmark pair using the *FairM1Dyn* algorithm. An 'x-y' partition means that the first thread is assigned $\frac{x}{8} \times 100\%$ of the cache, while the second thread is assigned $\frac{y}{8} \times 100\%$ of the cache. To collect the data for the figure, at the end of every time interval, we increment the partition count reflecting the partition size that was applied in the time interval just completed.

The figure shows that the resulting partitions are often non-symmetric, with larger partition for one thread and smaller for another. In one benchmark pair (*gzip+applu*), the partitions seem to be quite symmetric, where the brightest section representing 4-4 partition is in the middle of the bar. The figure shows that in most cases, there are only two partitions that are used much more often than all other partitions. This indicates that *FairM1Dyn* quickly converges to two partitions, and oscillates between the two partitions as a result of repartitioning and rolling back from it at alternating time intervals. This highlights the role of the rollback step that allows the partitioning algorithm oscillate around the optimal partition.

5.4. Parameter Sensitivity

Impact of Rollback Threshold. In the previous section, Figure 6 highlights the importance of the rollback mechanism in the dynamic partitioning algorithms in rolling back from bad repartitioning decision. The rollback threshold is used by the dynamic algorithms to undo the prior repartition assignment if it fails to reduce the miss rate of a thread that was given a larger partition. A higher rollback threshold makes the algorithm more conservative, in that it is harder to assign a new partition (without rolling it back) unless the



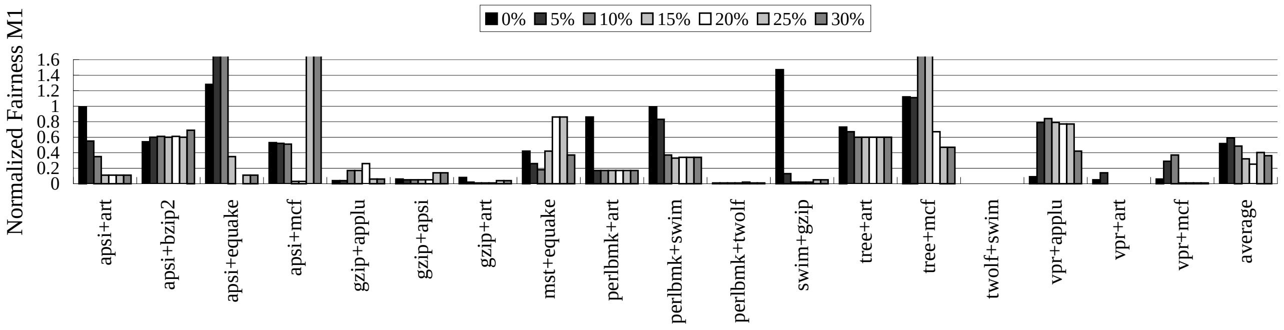


Figure 7: The impact of rollback threshold ($T_{rollback}$) on throughput (top chart) and fairness metric M_1 (bottom chart). Lower M_1 values indicate better fairness.

miss rate improvement is more substantial.

Figure 7 shows the throughput and fairness produced by different rollback threshold values ranging from 0% to 30%. All the bars are normalized to the *PLRU* case. The figure shows that the rollback threshold affects the throughput and fairness in some benchmark pairs in a very benchmark pair-specific manner. However, although the M_1 metric value changes with the threshold, the throughput improvement over *PLRU* does not vary much. Finally, on average, 20% rollback threshold achieves the best fairness.

Impact of Time Interval. Due to space constraint, we cannot show the detailed results of how different time intervals affect throughput and fairness. To summarize, we found that 10K L2 cache accesses to be slightly better than other time intervals. Larger time intervals affect the ability of the algorithms to converge quickly to the optimal partition, while smaller time intervals do not allow sufficient time to collect reliable miss count and miss rate statistics.

6. Related Work

Prior work in CMP architectures has ignored fairness issue and focused on throughput and its optimization techniques on a shared L2 cache [19, 10].

In Simultaneous Multi-Threaded (SMT) architectures, where typically the entire cache hierarchy and many processor resources are shared, metrics that mix throughput and fairness have been studied. The need to consider both aspects is intuitive given the high number of resources that are shared by SMT threads. Even in SMT architectures, however, the studies have only focused on either improving throughput, or improving throughput without sacrificing fairness too much [8, 17, 15, 12]. Fairness has not been studied as the main or separate optimization goal. For example, a weighted speedup, which incorporates fairness to some extent, has been proposed by Snavely, et al. [15]. Luo, et al. proposed harmonic mean of each thread's individual speedups to encapsulate both throughput and fairness [12]. Typically, to obtain the optimization goal, fetch policies, such as ICOUNT [6], can be used to control the number of instructions fetched from different SMT threads.

Our study differs in all the previous work in that we pursue fairness as the main and separate optimization goal on a CMP architecture. We also show that in most cases, optimizing fairness is sufficient because throughput is also improved. However, optimizing throughput does not always result in an optimal fairness due to the tendency to favor threads whose IPCs can be easily improved. An important benefit of our fair caching scheme is that it greatly simplifies OS design because to a large extent, it can abstract away the impact of cache sharing, and treat a CMP system the same way as a single time-shared processor. Without hardware that enforces fairness, the hardware and the OS have to be designed with extra complexity to detect pathological performance cases such as thread starvation and priority inversion, and implement a policy that correct the problems.

To deal with processor resource contention, an approach taken by Snavely, et al. [16], and Dorai and Yeung [4] is to expose the OS thread priority information to the SMT hardware. Dorai and Yeung [4] categorize SMT threads into foreground and background threads. Resources in SMT architecture are allocated in such a way to preserve the performance of the foreground thread, at the expense of fairness of the background (transparent) threads. Since transparent threads are not intrusive performance-wise, they can be used to run background tasks, such as software prefetching and performance monitoring. While transparent threads could be useful for some background tasks, in the general case, the OS already has a timeslice mechanism to make sure that background tasks are not intrusive to other tasks. Rather than exposing the OS structures to the hardware, our fair caching approach hides the complexity of the hardware from the OS, allowing simpler OS design.

7. Conclusions

This paper has shown that a thread-blind LRU replacement algorithm often produces unfair cache sharing, where in a co-schedule, some threads are slowed down much more than others, creating suboptimal throughput, as well as causing pathological performance problems that the OS has to tackle, such as thread starvation, priority inversion, and cache thrashing. To avoid that, we proposed the concept of *fair caching*, where the hardware guarantees that the impact of cache sharing is uniform for all the co-scheduled threads. With fair caching, the OS can mostly abstract away the impact of cache sharing, and expect priority-based timeslice assignment to work as effectively as in a time-shared single processor system.

This paper also proposed and evaluated five cache fairness metrics that measure the degree of fairness in cache sharing, and can be used to guide static and dynamic L2 cache partitioning algorithms in optimizing fairness. The dynamic partitioning algorithm is easy to implement, requires little or no profiling, has low overhead, and does not restrict the cache replacement algorithm to LRU. The static algorithm, although requiring the cache to maintain LRU stack information, can help the OS thread scheduler to avoid cache thrashing. Finally, this paper has studied the relationship between fairness and throughput in detail. We found that optimizing fairness usually increases throughput, while maximizing throughput does not necessarily improve fairness. This is because throughput may improve at the expense of fairness, for example by favoring some threads whose throughput is easy to improve, over others. Using a set of co-scheduled pairs of benchmarks, on average our algorithms improve fairness by a factor of $4\times$, while increasing the throughput by 15%, compared to a non-partitioned shared cache. The throughput improvement is slightly better than a scheme that minimizes the total number of cache misses as its main objective.

8. Acknowledgement

We would like to thank Milos Prvulovic for his valuable comment on an early version of this paper, and anonymous reviewers for their suggestions to improve the paper.

References

[1] J. E. Barnes. Treecode. Institute for Astronomy, Univ. of Hawaii. *ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode*, 1994.

[2] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Twelfth Intl. Workshop on Languages and Compilers for Parallel Computing*, 1999.

[3] Z. Cvetanovic. Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor. In *30th Intl. Symp. on Computer Architecture*, 2003.

[4] G. K. Dorai and D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 30–41, 2002.

[5] E. Dudewicz and S. Mishra. *Modern Mathematical Statistics*. John Wiley, 1988.

[6] J. Emer. EV8: The Post Ultimate Alpha. *PACT Keynote Presentation*, 2001.

[7] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan-Kaufmann Publishers, Inc., 2nd edition, 1996.

[8] S. Hily and A. Seznec. Contention on the 2nd level cache may limit the effectiveness of simultaneous multithreading. *IRISA Tech. Rep. 1086*, 1997.

[9] IBM. *IBM Power4 System Architecture White Paper*, 2002.

[10] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-Assisted Cache Replacement Mechanisms for Embedded Systems. In *Intl. Conf. on Computer-Aided Design*, 2001.

[11] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code on an Intelligent Memory Architecture. In *7th Intl. Symp. on High Performance Computer Architecture*, 2001.

[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, 2001.

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

[14] P. Ranganathan, S. Adve, and N. Jouppi. Reconfi gurable Caches and their Application to Media Processing. In *27th Intl. Symp. on Computer Architecture*, June 2000.

[15] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[16] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRIC*, 2002.

[17] A. Snavely, et al. Explorations in symbiosis on two multithreaded architectures. In *Workshop on Multithreaded Execution, Architecture, and Compilation*, 1999.

[18] Standard Performance Evaluation Corporation. Spec benchmarks. *http://www.spec.org*, 2000.

[19] G. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *High Performance Computer Architecture*, 2002.