Cache Sharing Management for Performance Fairness in Chip Multiprocessors

Xing Zhou Wenguang Chen Weimin Zheng
Dept. of Computer Science and Technology
Tsinghua University, Beijing, China
zhoux07@mails.tsinghua.edu.cn, {cwg, zwm-dcs}@tsinghua.edu.cn

Abstract

Resource sharing can cause unfair and unpredictable performance of concurrently executing applications in Chip-Multiprocessors (CMP). The shared last-level cache is one of the most important shared resources because off-chip request latency may take a significant part of total execution cycles for data intensive applications. Instead of enforcing ideal performance fairness directly, prior work addressing fairness issue of cache sharing mainly focuses on the fairness metrics of cache miss numbers or miss rates. However, because of the variation of cache miss penalty, fairness on cache miss cannot guarantee ideal fairness. Cache sharing management which directly addresses ideal performance fairness is needed for CMP systems.

This paper introduces a model to analyze the performance impact of cache sharing, and proposes a mechanism of cache sharing management to provide performance fairness for concurrently executing applications. The proposed mechanism monitors the actual penalty of all cache misses and uses Auxiliary Tag Directory (ATD) to dynamically estimate the cache misses with dedicated cache when the applications are actually running with shared cache. The estimated relative slowdown for each core from dedicated environment to shared environment is used to guide cache sharing in order to guarantee performance fairness. The experiment results show that the proposed mechanism always improves the performance fairness metric, and can provide no worse throughput than the scenario without any management mechanism.

1. Introduction

Data intensive applications usually spend a large proportion of total execution cycles on memory accessing because of the long latency of off-chip requests. The on-chip last level cache (usually L2 or L3 cache), which is the key component to hide off-chip request latency, is typically shared by multiple cores in a Chip-Multiprocessors(CMP) architecture. As a result, the sharing of last level cache has significant impacts on the performance of concurrently executing applications on different cores. The technologies of server consolidation and virtual machine are calling for the demand of scheduling heterogeneous workloads together. The different memory accessing characteristics of heterogeneous workloads will lead to unfair cache sharing, which breaks the hardware fairness assumption of the scheduler and may bring thread starvation, priority inversion and other problems to operating system's process scheduler [6].

Prior work has noticed the problem of unfair cache sharing and the consequential result of unfair performance. [6] proposed a cache partitioning mechanism to enhance fairness of cache sharing. Although intuitively the ideal fairness of cache sharing should be equal slowdowns relative to running with a dedicated cache for all co-scheduled applications, [6] uses the equal increment of cache miss numbers or miss rates as fairness metrics, because [6] points out that ideal fairness is hard to measure and cache miss fairness are usually highly correlates with the ideal fairness.

The mechanism proposed in [6] is an improvement over unmanaged caches. However, because the cache miss fairness is not identical to ideal fairness, the improvement on cache miss fairness can not guarantee the same degree of ideal fairness improvement, and enforcing fairness on cache miss does not necessarily lead to the situation of ideal fairness. To explain the above results, we should note that the correlation between cache miss fairness and ideal fairness (performance) actually depends on two factors: (1) The performance sensitivity of each application to cache misses varies, as applications spend differing fractions of execution time stalled on cache misses. The performance of those applications with a large fraction of execution cycles stalled on misses will be more sensitive in cache miss varies. This fraction are diverse; for example, data centric applications will spend much more cycles on memory access (stalled by cache misses) than computation oriented applications. (2) The stalls arising from each miss vary as a function of Memory Level Parallelism (MLP) [13][2] and ILP. Clustered cache misses' latency cycles overlapped with recent

misses, and the average penalty of each cache miss can be reduced. So the actual penalty of each cache miss is smaller than memory access latency and may vary according to different memory access characteristics. Besides, computation operation cycles and memory access latency cycles may overlap, which can also reduce the actual penalty of cache misses.

Thus, in order to enforcing ideal fairness on cache sharing, we must firstly build an analytic model to account for the performance impact of cache sharing. The analytic model should be able to help to divide applications' whole execution cycles into "private part", which is not related to cache sharing, and "vulnerable part", which is susceptible to additional cache misses caused by cache sharing. Then, according to the analytic performance model we can develop a mechanism to provide a reference point for ideal fairness metric by measuring or estimating the application's performance in a dedicated cache when it is actually running in a shared cache.

This paper makes two contributions: (1) we proposes a model to analyze the performance impact of cache sharing for CMP, considering not only additional cache misses caused by cache interference of concurrent workloads, but also the variation of the actual penalty for each cache miss. (2) To the best of our knowledge, this paper proposes the first mechanism that partitions shared cache and enforces fairness for the goal of ideal performance fairness. This mechanism is dynamic and adaptive, no static profile needed. The proposed mechanism always improves the performance fairness metric, and can provide no worse throughput than the cache without any management mechanism.

The rest of the paper is organized as follows. Section 2 gives a definition of ideal fairness as well as cache miss fairness which is used in prior work. Section 3 introduces an accurate model, which connects cache miss rate and overall performance, to estimated the performance impact of cache sharing. In Section 4, the hardware mechanism of enforcing fairness on shared cache in a typical CMP architecture is described in detail. Section 5 describes the experiment methodology and discusses experiment results. Section 6 introduces related work briefly and Section 7 gives a conclusion.

2. Defining Performance Fairness

We define *performance fairness*, which is the ideal fairness discussed above, as identical slowdown for each concurrent workload running with shared cache compared to running with separate, dedicated caches, which is called *execution time fairness* in [6]. Let T_i^{ded} denotes the execution time (count of execution cycles) of workload i with dedicated cache and T_i^{shr} for the execution time with shared cache, then *performance fairness* is achieved when:

$$\frac{T_i^{shr}}{T_i^{ded}} = \frac{T_j^{shr}}{T_j^{ded}} \tag{1}$$

for every pair of concurrent workloads i and j. T_i^{ded} and T_i^{shr} can be replaced by CPI_i^{ded} and CPI_i^{shr} for a certain slice of running instructions, and then Equation 1 can be transformed to:

$$\frac{CPI_i^{shr}}{CPI_i^{ded}} = \frac{CPI_j^{shr}}{CPI_j^{ded}} \tag{2}$$

Actually $\frac{CPI_i^{shr}}{CPI_i^{ded}}$ is the slowdown of workload i running with shared cache compared to dedicated cache. And we define performance fairness metric of a pair of concurrently running workloads i and j under a certain cache partition as M_{perf}:

$$M_{perf} = \sum_i \sum_j \left| \frac{CPI_i^{shr}}{CPI_i^{ded}} - \frac{CPI_j^{shr}}{CPI_j^{ded}} \right| \tag{3}$$

Intuitively, M_{perf} is the sum of the slowdown difference between every two co-scheduled workloads. The smaller M_{perf} is, the smaller the slowdown difference among all co-scheduled workloads, thus the better the performance fairness. If M_{perf} equals zero, perfect performance fairness is achieved.

For the purpose of comparison, we also define that *cache miss fairness* is achieved when:

$$\frac{MPKI_i^{shr}}{MPKI_i^{ded}} = \frac{MPKI_j^{shr}}{MPKI_j^{ded}} \tag{4}$$

for every pair of concurrent workloads i and j, in which $MPKI_i^{shr}$ and $MPKI_i^{ded}$ denote the miss count per thousand instructions when workload i running with dedicated cache and shared cache. Cache miss fairness metric is defined as:

$$M_{miss} = \sum_i \sum_j \left| \frac{MPKI_i^{shr}}{MPKI_i^{ded}} - \frac{MPKI_j^{shr}}{MPKI_j^{ded}} \right| \tag{5}$$

Similar to M_{perf}, the smaller M_{miss} is, the better the cache miss fairness, and perfect cache miss fairness is achieved when M_{miss} equals zero.

M_{miss} is the same as M_3 in [6] and $FM3$ in [7]. [6] proposed five fairness metrics, but M_2 has been shown to have a poor correlation with performance fairness; M_1 and M_3 contribute the most highly correlation with performance fairness for most cases. If the workload runs the same count of instructions with dedicated cache and shared cache, M_{miss} is the same as M_1, too. So M_{miss} is representative enough.

3. Modeling Performance Impact of Cache Sharing

We use a typical CMP architecture configuration for following analysis: N cores on chip; each core has private L1 instruction cache and data cache; unified on-chip L2 cache shared by all on-chip cores is the last level cache, and a miss in L2 cache will issue an off-chip request. All caches are set-associated.

To model performance impact of cache sharing, the cycles consumed during application's running time can be categorized into two classes: *private operation cycles* (T_{pri}) and *vulnerable operation cycles* (T_{vul}). Intuitively, private operation cycles are the part of execution cycles which only depends on the characteristics of the workload. Vulnerable operation cycles are sensitive to different co-schedulers on other cores and may varies diversely because of resource sharing. Private operation cycles are consumed by those operations which only need private resources of the core, such as computation time, the latency of fetching instruction and data from private L1 cache. A L2 request is a hybrid operation. The tag looking up latency is accounted in private operation cycles for it is constant no matter whether this request misses or not. However, if the L2 cache misses, the cycles of off-chip request latency are vulnerable operation cycles because this cache miss may be caused by cache sharing and the off-chip request is extra cycles.

However, the whole execution cycles is not simply equal to the sum of T_{pri} and T_{vul} because there are overlap cycles of private operations and vulnerable operations. For example, in an out-of-order processor, even if a instruction is stalled by a L2 cache miss, other instructions in the schedule windows still can enter pipeline if there is no data dependence. As a result, the *overlap cycles* (T_{ovl}) must be introduced, then:

$$T = T_{pri} + T_{vul} - T_{ovl} \tag{6}$$

The example in Figure 1 illustrates Equation 6.

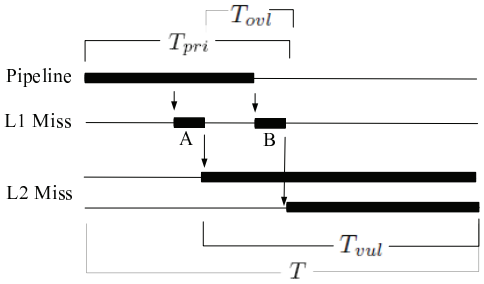


Figure 1. Illustration of Equation 6. The horizontal axis represents time (cycles) elapsed.

In this example, during the execution time of T, two L2 cache misses occur. After the first miss, the instruction window still has instructions with no data dependence on this miss, so the pipeline is not stalled yet until the second miss occurs. The computation time of pipeline is certainly part of T_{pri}, and the off-chip request latency of L2 misses is T_{vul}. Note that when L1 misses occur, the L1 request latency (tag looking up latency in L2 cache), such as slice A and slice B in the figure, is part of T_{pri}; however, it ends up as an L2 miss, the latency of off-chip request does not belong to T_{pri} any more. T_{ovl} is formed by those cycles when T_{pri} and T_{vul} overlap.

Because the vulnerable operation cycles are mainly the cycles of off-chip request latency, T_{vul} can be approximated by the total penalty of off-chip requests. Average MLP is needed to consider to estimate the average penalty of each cache miss. We derive the average MLP definition in [2] as *the average number of useful outstanding off-chip requests when there is at least one outstanding off-chip requests*, denoted by MLP_{avg}. We have the following equation:

$$\begin{aligned} T_{vul} &= \sum_{i=1}^{N_{miss}} Miss\_Penalty_i \\ &= N_{miss} * \frac{Miss\_Latency}{MLP_{avg}} \end{aligned} \tag{7}$$

Then Equation 6 can be:

$$T = T_{pri} - T_{ovl} + N_{miss} * \frac{Miss\_Latency}{MLP_{avg}} \tag{8}$$

In Equation 8 the latency of off-chip request ($Miss\_Latency$) can be treated as a constant (ignoring other factors such as bus congestion). Equation 6 and 8 shows that the total execution time of an application including three parts: T_{pri} is the inherit part that does not change for cache sharing. T_{ovl} and T_{vul} are related to cache sharing; they are the cause of unfair performance and the part of execution time that cache partition mechanisms want to adjust. According Equation 8, if we can get the parameters of T_{ovl}, N_{miss} and MLP_{avg} through hardware profiler, the total execution time can be estimated dynamically.

4 Hardware Mechanism

The necessary hardware support includes two interdependent parts: cache partition mechanism and hardware profiler. Figure 2 shows how it works. The hardware profiler gather the statistics of each core's pipeline, each private L1 instruction cache and data cache, and unified L2 cache during last period. At the end of period, cache partition decision is made and the cache partition mechanism applies the decided partition to cache.

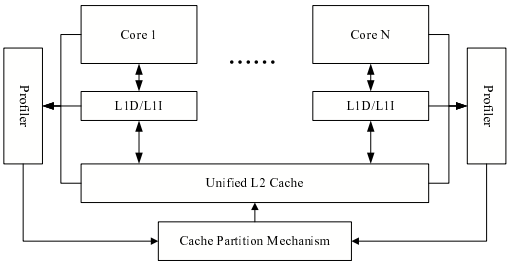


Figure 2. Hardware Framework

4.1 Cache Partition Mechanism

The cache partition mechanism is the simpler part. Assuming there are N cores on chip sharing L2 cache, we add $\log_2 N$ bits for each cache block to mark which core this cache block belongs to. And each core has a bit of $overAlloc$ flag to indicate whether this core has been allocated too much cache space or not. When a cache miss from core i occurred, firstly found the correct cache set; if the $overAlloc$ flag of core i shows that core i is not over allocated, use original cache replacement policy to find a victim cache block, and change the mark bits of this cache block as belonging to core i; if the $overAlloc$ flag shows that core i has been allocated too much cache space, randomly select one cache block with mark of core i as victim cache block. This mechanism guarantees over allocated cores can not gain more cache lines, and the additional cache block will be gradually taken by other under allocated cores, and the partition target set by $overAlloc$ flag of each cores. A special situation is that though the register of core i shows over allocated, there is no cache block marked as belonging to core i, we still allocate a cache line for core i.

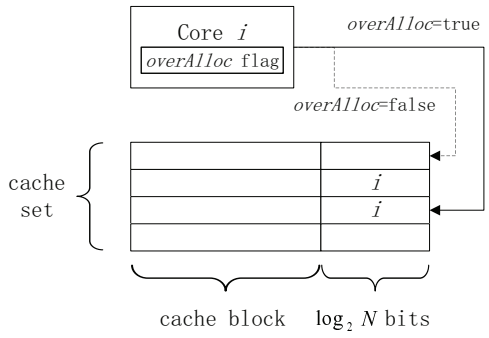


Figure 3. Cache Partition Mechanism

4.2 Hardware Profiler

The hardware profiler is more complex. To achieve performance fairness for a shared cache CMP, identical slowdowns compared to running with dedicated cache is the ideal goal. The key problem is to estimate the cycles needed by the workload with dedicated cache when it is actually running with a shared cache. T_i^{shr} and T_i^{ded} or CPI_i^{shr} and CPI_i^{ded} for each core i are need to evaluate M_{perf} defined in Equation 3. The job of hardware profiler is to profile necessary runtime or statistical parameters when the applications are executing concurrently, and use Equation 6 and Equation 8 to dynamically estimate the situation if running with dedicated cache.

According to Equation 6 and Equation 8:

$$T^{shr} = T^{shr}_{pri} - T^{shr}_{ovl} + T^{shr}_{vul} \tag{9}$$

$$T^{ded} = T^{ded}_{pri} - T^{ded}_{ovl} + N^{ded}_{miss} * \frac{Miss\_Latency}{MLP^{ded}_{avg}} \tag{10}$$

According to the definition of T_{pri} we can get $T^{shr}_{pri} = T^{ded}_{pri} = T_{pri}$. So in order to enforce performance fairness, T^{shr}, T^{shr}_{ovl} and T^{shr}_{vul} should be profiled in shared mode, and then T_{pri} can be calculated; at the same time, T^{ded}_{ovl}, N^{ded}_{miss} and MLP^{ded}_{avg} need to be estimated, and finally we gain T^{ded} and be able to evaluate M_{perf}. The data flow for analysis is showed in Figure 4.

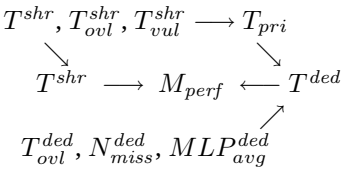


Figure 4. Data Flow for Analysis

If we only want to enforce cache miss fairness defined by Equation 4, things are much simpler: only N^{shr}_{miss} and N^{ded}_{miss} are needed, then M_{miss} defined by Equation 5 can be evaluated.

4.2.1 Profiling parameters for shared cache

The profiler calculate statistics in every PT cycles (profiling period), then $T^{shr} = PT$. To get T^{shr}_{vul}, we add a $log_2 N$ bits flag to each entry of L2 MSHR (Miss Status Holding Register) to identify which core causes this miss; then monitor the count of entries with the specified flag to see if there is request of this core in MSHR. At the beginning of profiling period, T^{shr}_{vul} is initialized to zero. If MSHR has at least one entry for this core, T^{shr}_{vul} should be increased because there is at least one off-chip request on this cycle.

To decide whether a cycle of the whole execution time belongs to T^{shr}_{ovl}, three conditions should be considered: (1) whether the pipeline is stalled in this cycle; (2) whether there is at least an L1 request in this cycle; (3) whether there

Algorithm 1: Calculating T_{vul}^{shr}

```
/* at the beginning of each profiler period */
  T_vul^shr = 0;

  /* for each cycle*/
if L2 MSHR has entries for this core then
      T_vul^shr ++;
```

Algorithm 2: Calculating T_{ovl}^{shr}

```
/* at the beginning of each profiler period */
  T_ovl^shr = 0;

/* for each cycle; for L1I and L1D of this core*/
 if L1 miss occurs at this cycle
      then timer_of_this_L1=L2_LAT;
       else timer_of_this_L1--;

  /* for each cycle*/
 if pipeline is not stalled              /*(1)*/
      then pipelineNotStall=true;
      else pipelineNotStall=false;
if L2 MSHR has entries for this core /*(3)*/
       then hasL2Request=tru
      else hasL2Request=false;
 if L1I or L1D timer is not zero       /*(2)*/
       then hasL1Request=true;
      else hasL1Request=false;
if ( pipelineNotStall or hasL1Request)
    and hasL2Request then T_ovl^shr ++;
```

is at least an L2 request for this core in this cycle. Condition (2) only means the on-chip request; if the L1 request ends up to be an L2 miss, the off-chip latency cycles is not condition (2) (referring the example showed in Figure 1). Condition (1) can be easily got by monitoring the status of pipeline and condition (3) by monitoring the whether the L2 MSHR has entries for this core. To decide condition (2), we add a timer for each L1 cache, including instruction cache and data cache. Let $L2\_LAT$ denotes the lookup latency of L2 ($L2\_LAT$ is always a fixed value). If this L1 cache misses and issues a request to L2, the timer is set to $L2\_LAT$. For each cycle, the timer is decreased by 1. The condition (2) can be decided by checking whether the timer is zero or not. Algorithm 2 described the logic in detail.

4.2.2 Profiling and estimating parameters for dedicated cache

The three parameters of T_{ovl}^{ded}, N_{miss}^{ded} and MLP_{avg}^{ded} shown in Figure 4 are responsible to the application running with a dedicated cache. However, because the workload is actually running with a shared cache, we need special hardware to help to estimate the situation when it is running with a dedicated cache. To achieve this, we uses the technology of Auxiliary Tag Directory (ATD) [13] to attach a virtual "private" L2 cache to each core. An ATD has the same associativity as the main tag directory of the shared L2 cache and uses the same replacement policy. However, an ATD only contains the tag and other functional bits of each cache block but do not keep data. We also add an auxiliary MSHR to each ATD, then an ATD can act just as a private L2 cache. Each entry in the auxiliary MSHR has a timer to simulate memory accessing request. When a memory request is issued to an auxiliary MSHR, the timer of the inserted entry is set to the round trip cycles of memory accessing latency. For every cycle, all entries' timers are automatically decreased by 1, A timer's value equals 0 means this memory request has returned and the request block is ready for ATD.

Because there is an ATD for each core, the total hardware cost is substantial. We employ set sampling technology [13] to reduce the storage cost. The ATD with set sampling only selects cache set samples in a specified interval, and the behavior of the whole can be approximated by sampled cache sets. An ATD with large sampling interval requires less hardware overhead, but gives less accurate statistics. So, there is a tradeoff between hardware cost and accuracy in choosing sampling interval.

With an ATD and an auxiliary MSHR, it is possible to simulate a private L2 cache for each core. When a request from upper level L1 cache is coming, it is forwarded to the really L2 cache as well as the attached ATD of the core which the request sender (L1 cache) belongs to. Then the ATD respond to this request just as L2 cache does, including touching MRU bits, replacement, counting miss count and issue requests to the auxiliary MSHR. So N_{miss}^{ded} and T_{ovl}^{ded} can be gained by the ATD and auxiliary MSHR attached to each core using similar mechanism which is used to get N_{miss}^{shr} and T_{ovl}^{shr}.

MLP_{avg}^{ded} can also be gained with the help of auxiliary MSHR. MLP_{avg} is defined as the average number of useful outstanding off-chip requests when there is at least one outstanding off-chip requests. Two counter MLP_sum and MLP_cycles are used to store the accumulated off-chip latency and the count of cycles when there is at least one outstanding off-chip request. The algorithm is described as follows:

Note that T_{ovl}^{ded} and MLP_{avg}^{ded} are both estimated values while we can treat N_{miss}^{ded} as accurate value if set sampling is not employed. Because L2 cache misses have impact on the pipeline, the relative order of L2 cache misses may be

Algorithm 3: Calculating MLP_{avg}^{ded}

```
/* at the beginning of each profiler period */
  MLP_sum = 0;
  MLP_cycles = 0;

  /* for each cycle*/
if auxiliary MSHR is not empty then
    MLP_cycles++;
  MLP_sum+ =count of auxiliary MSHR entries;

/* at the end of each profiler period */
MLP_avg^ded = MLP_sum / MLP_cycles;
```

different between running with shared cache and with dedicated cache, which makes the mechanism for calculating T_{ovl}^{ded} and MLP_{avg}^{ded} inaccurate compared to really running the application with a dedicated cache. However, if the application is actually running with dedicated cache, the same algorithm can used to obtain the accurate MLP by replacing auxiliary MSHR with main L2 MSHR.

Now the hardware profiler can provide all the parameters needed to calculate T_i^{shr} and T_i^{ded} for each core i. The slowdown for core i can be calculated at the end of profiling period by:

$$Slowdown_i = \frac{T_i^{shr}}{T_i^{ded}} \tag{11}$$

The core which has the least slowdown is selected as being allocated too much cache space by setting the this core's *overAlloc* flag. Through the cache partition mechanism, the slowdown of each core will be closer and performance fairness will be improved.

4.3 Hardware cost

For each cache line, $\log_2 N$ bits are added to indicate which core this cache line belongs to, in which N is the number of cores sharing the cache. In addition, there should be some monitor circuits in L2 MSHR, L1 cache and pipeline. For each core, there are 2 additional L1 timers and 1 auxiliary MSHR, typically 32-entry.

ATDs are the major part of cost. Set sampling can significantly reduce the storage of ATD. For the configuration of 2-way CMP with 1M, 64 bytes line size, 8-way associative L2 cache, and sampling for each 16 cache sets, the hardware cost is:

- For each ATD entry: (24 bits tag)+(4 bits LRU)+(1bit valid)+(2 bits in-addition)=31 bits
- Number of ATD entries: (1MB L2)/(64 Bytes block)/(16 cache sets sampling)=1024

Total cost for 1 ATD: 1024*31bit<4KB. The original L2 cost is: 1MB+(24+4+1)*(1M/64B)=1.45MB. So the additional cost for 2 ATDs of the 2 cores is less than 1%:

$$(2 * 4KB)/1.45MB = 0.005 < 1\%$$

5 Experimental Methodology

5.1 Configuration

The evaluation is performad using Simics [8], which is a whole system simulator supporting CMPs. The memory timing model is derived from GEMS [9], which enables detailed cycle-accurate simulation of multiprocessor systems for Simics.

Table 1 shows the basic parameters of the simulated architecture. The simulated CMP cores are out-of-order superscalar processors with private L1 instruction and data caches, sharing unified L2 cache and all lower levels of memory hierarchy. All caches are set-associated using Pseudo LRU policy for replacement decision.

CMP	2 cores on chip, share on-chip L2 cache
Processor core	4-issue out-of-order processors instruction window size: 64 Re-order buffer size: 128
L1 cache	private Icache and Dcache for each core Icache: 32KB, 64B line-size, 4-way PLRU Dcache: 32KB, 64B line-size, 4-way PLRU
Unified shared L2 cache	1MB, 64B line-size, 8way PLRU 12-cycle hit, 32-entry MSHR shared by all cores on chip
Memory	400-cycle round trip latency

Table 1. Basic configuration of the simulated architecture

5.2 Metrics

To evaluate the fairness improvement of different schemes at the baseline of uncontrolled L2 cache sharing (original Pseudo LRU policy), we define:

$$F_{perf}^{scheme} = \frac{M_{perf}^{scheme}}{M_{perf}^{PLRU}} \tag{12}$$

$$F_{miss}^{scheme} = \frac{M_{miss}^{scheme}}{M_{miss}^{PLRU}} \tag{13}$$

M_{perf}^{scheme} and M_{perf}^{PLRU} are performance fairness metrics define by Equation 3; M_{miss}^{scheme} and M_{miss}^{PLRU} are cache

miss fairness metrics define by Equation 5. Obviously $F_{perf}^{PLRU} = 1$ and $F_{miss}^{PLRU} = 1$. Those metrics are normalized fairness metrics. The smaller the value, the better the fairness of the scheme. Zero means perfect fairness. If the baseline already achieves perfect fairness, F_{perf}^{scheme} and F_{miss}^{scheme} would be infinite unless the cache partition scheme also achieves perfect fairness.

5.3 Benchmarks

We select a set of most memory-intensive benchmarks from SPEC CPU 2000 benchmark suite, and run them concurrently in pairs in the simulated dual-core CMP system to see the benefit of different cache partition schemes, including uncontrolled L2 cache sharing using original Pseudo LRU policy, enforcing cache miss fairness on cache partitioning and enforcing performance fairness on cache partitioning. We compare the normalized cache miss fairness metric (F_{miss}^{scheme}) and normalized performance fairness metric (F_{perf}^{scheme}) for all cache partition schemes at the baseline of uncontrolled L2 cache sharing. For each selected benchmark, a representative slice of instructions is obtained using a tool SimPoint [10] for simulation.

To categorize the selected benchmarks, we consider two features of each benchmark that can affect the correlation between cache miss fairness and performance fairness: (1) the portion which vulnerable time took in the whole execution time, which is T_{vul}/T according to Equation 6; (2) the average MLP during execution of the benchmark. Workloads with high value of T_{vul}/T are more sensitive to interleaving, while workloads with high average MLP are more tolerant for last level cache misses. These two factors must be combined in analysis.

	High T_{vul}/T	Low T_{vul}/T
High MLP	mcf(55.2%, 1.53) art(56.5%, 1.82)	ammp (29.2%, 1.47)
Low MLP	applu(72.3%, 1,21) swim(63.3%, 1.10) equake(47.6%,1.19) sixtrack(39.2%,1.09)	gzip(24.1%, 1.26) apsi(28.5%, 1.08) vpr(28.3%, 1.28)

Table 2. Benchmark classification

Table 2 shows the classification of the ten selected benchmarks based on the values of T_{vul}/T and average MLP: *mcf, ammp, art, applu, gzip, swim, apsi, equake, vpr* and *sixtrack*. The pair of numbers shown in parenthesis denotes the benchmark's values of T_{vul}/T and average MLP. These data are collected by running a single benchmark with dedicated cache of original Pseudo-LRU replacement policy. The eight benchmarks are categorized into four classes: high-vulnerability/high-MLP, high-vulnerability/low-MLP, low-vulnerability/high-MLP and low-vulnerability/low-MLP.

We pair the benchmarks presented above and running them concurrently in our simulated system with shared cache to evaluate fairness metrics of different schemes. Table 3 shows the classification parameters when the benchmarks are running concurrently with shared cache of original replacement policy. We can see that although the two parameters may vary diversely (especially the portion of cache miss latency), the relationship of parameter values are kept. That is, if a benchmark from the class of high/high is running concurrently with another benchmark from the class of low/low, the first benchmark usually still has a larger portion of cache miss latency and a larger MLP. So is other combinations of benchmarks from the rest classes.

	T_{vul}/T App1	T_{vul}/T App2	MLP App1	MLP App2
gzpi+mcf	32.0%	82.7%	1.28	1.36
gzip+art	26.7%	62.2%	1.16	1.83
apsi+art	36.1%	61.9%	1.19	1.78
swim+apsi	87.1%	31.1%	1.07	1.10
vpr+applu	31.2%	78.1%	1.31	1.04
ammp+applu	44.0%	86.7%	1.45	1.17
mcf+swim	83.8%	67.8%	1.30	1.02
swim+art	85.0%	73.6%	1.19	1.87
equake+mcf	55.0%	68.8%	1.16	1.57
ammp+sixtrack	40.0%	41.6%	1.50	1.09
equake+sixtrack	64.6%	48.2%	1.19	1.07

Table 3. Benchmark pairs and parameters when running concurrently.

5.4 Results and Analysis

5.4.1 Model Accuracy Verification

Before evaluating fairness metrics, we need to verify how accurate the model described in above sections could be. In this verification experiment, we ran benchmarks for two passes. For the first pass, we used the hardware configuration in Table 1, but only applied profiler component of the hardware mechanism to the shared L2 cache; the cache replacement policy is not modified. We ran the benchmark pairs in Table 3 in the share cache configuration and get necessary statistics for estimating the benchmarks' performance (IPC') when running with dedicated cache. In the second pass, we ran these benchmarks again in identical hardware except using dedicated L2 cache for each core (the dedicated cache has the same configurations as the shared

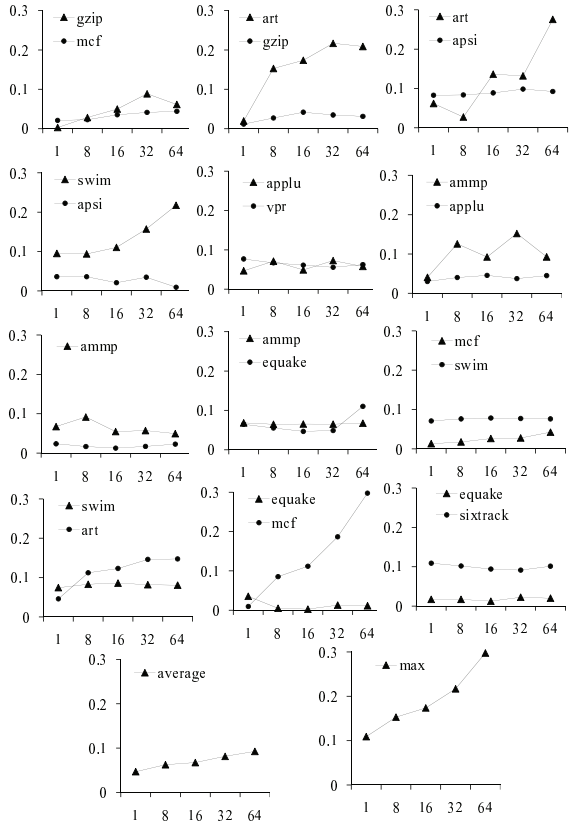


Figure 5. Verifying accuracy of the model. Y axis represents the relative error (E defined in Equation 14). X axis represents the interval of the set sampling in ATD; interval 1 means do not use set sampling (sample every tag set in ATD).

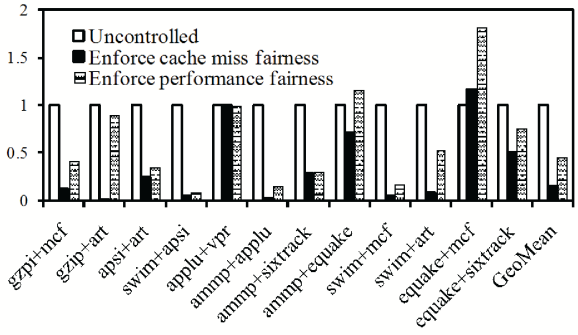
cache). In this pass of running, we can get the the benchmarks' actual (IPC) performance when running with dedicated cache. Then we can compare IPC' and IPC, and get the *relative error*:

$$E = \left| 1 - \frac{IPC'}{IPC} \right| \tag{14}$$

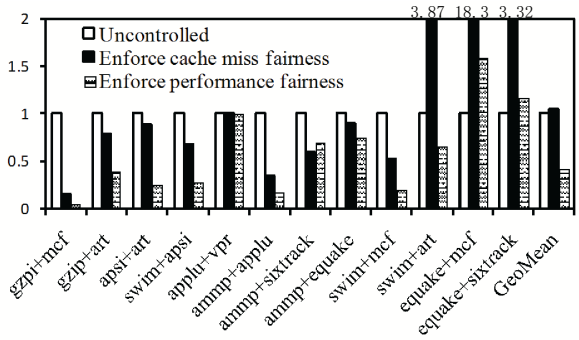
To review the effect of employing set sampling technology on ATD, we compared the estimating error of different sampling interval. From Figure 5 we can see that, if do not use set sampling in ATD (sampling interval 1), the average E is 4.7% and max E is 10.9% (*equake* running with *sixtrack*). As the sampling interval increasing, E increases as well. When sampling ATD set with interval 16, the average E is 6.7% and max E is 13.3% (*art* running with *gzip*), which is still accurate enough. So the following experiments used interval 16 for set sampling in ATD.

5.4.2 Evaluation of fairness metrics

We use F_{perf}^{scheme} and F_{miss}^{scheme} described in Section 4.2 to measure the fairness improvement for two cache partition schemes: the scheme that enforces cache miss fairness (SECF) and the scheme that enforces performance fairness (SEPF). The baseline is uncontrolled cache sharing using originally Pseudo LRU policy, which is widely used in production processors.



(a) Cache miss fairness metric (F_{miss}^{scheme})



(b) Performance fairness metric (F_{perf}^{scheme})

Figure 6. Comparing fairness metrics of selected benchmark pairs with uncontrolled Pseudo LRU and the two cache partition schemes. Note that shorter bar means better fairness

Figure 6 shows the experiment results of cache miss fairness metric (F_{miss}^{scheme}) and performance fairness metric (F_{perf}^{scheme}). For each benchmark pair, the two benchmarks are executed concurrently for three passes: in the first pass, shared L2 cache uses original, uncontrolled Pseudo LRU replacement policy; in the second and third pass, cache partition schemes of enforcing cache miss fairness (SECF) and enforcing performance fairness (SEPF) are used. The two fairness metrics of (F_{miss}^{scheme}) and (F_{perf}^{scheme}) are measured for each pass of execution.

In Figure 6 (a), we can see that when using SECF to en-

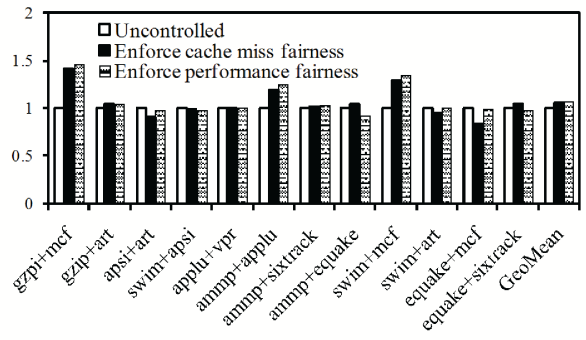


Figure 7. Comparing throughput of selected benchmark pairs with uncontrolled Pseudo LRU and the two cache partition schemes. Note that taller bar means higher throughput.

force cache miss fairness on shared cache, cache miss fairness metric (F_{miss}^{scheme}) is improved significantly compared to uncontrolled Pseudo LRU replacement policy (note that shorter bar means better fairness). However, when using SEPF to enforce performance miss fairness on shared cache, though there is still cache miss fairness metric (F_{miss}^{scheme}) improvement compared to uncontrolled cache for most benchmarks, F_{miss}^{scheme} is not improved as much as SECF. Even there are benchmark pairs which show worse cache miss fairness than uncontrolled Pseudo LRU cache when using SEPF(*ammp+equake* and equake+mcf). On average, SECF gains a cache miss fairness improvement of F_{miss}^{scheme} = 0.14, while SEPF gains an improvement of F_{miss}^{scheme} = 0.45.

Figure 6 (b) shows the performance fairness metric (F_{perf}^{scheme}) when using different cache partition schemes. The result is different from the result of cache miss fairness metric. Although SECF always shows better F_{miss}^{scheme} than the SEPF, it fails to gain a better performance fairness metric (F_{perf}^{scheme}) than SEPF. Whats more, there are three benchmark pairs suffer great performance fairness degradation when using SECF to enforce cache miss fairness: swim+art (3.87), equake+mcf (18.3) and equake+sixtrack(3.32), which means that SECF may make the problem of unfair performance of concurrent workloads even worse in some cases. In contrast, although SEPF got poor F_{miss}^{scheme} for ammp+equake and equake+mcf, it ends up that SEPF gains better performance fairness in these benchmark pairs compared to SECF. On average, SECF gains a performance fairness improvement of F_{perf}^{scheme} = 1.04, while SEPF gains an improvement of F_{perf}^{scheme} = 0.41.

From the comparison of the results showed in Figure 6 (a) and Figure 6 (b), we can get a deeper insight into the correlation between cache miss fairness and performance fairness. Merely enforcing cache miss fairness on concurrent workloads can improve performance fairness in most cases, but can not guarantee ideal performance fairness and sometimes may suffer performance fairness instead of improving it, especially when the co-scheduled workloads have different features. For example, when benchmarks of type 4 (low-vulnerability/low-MLP) are running with benchmarks with type 1 (high-vulnerability/high-MLP) such as benchmark pairs of *gzip+mcf*, *gzip+art* and *apsi+art*, SECF gains a much poorer performance fairness than SEPF; when benchmarks of type 2 (high-vulnerability/low-MLP) are running with benchmarks with type 1(high-vulnerability/high-MLP) such as benchmark pairs of *swim+art*, *equake+mcf* and *equake+sixtrack*, SECF even suffers performance fairness a lot. The reason is that when a low-MLP workload is running concurrently with another high-MLP workload, more shared cache resource should be allocated to low-MLP workload to guarantee the whole performance fairness because low-MLP workload has a relatively larger cache miss penalty and enforcing cache miss fairness can not guarantee performance fairness. And if both of the concurrently running workloads are highly sensitive to cache interference (type 1 benchmarks running with type 2 benchmarks), the degradation of performance will be more obvious when using SECF to enforcing cache miss fairness. By taking more factors into account, SEPF can achieve better performance fairness in most cases.

Figure 7 shows the throughput of selected benchmark pairs with uncontrolled Pseudo LRU and the two cache partition schemes. We use fair speedup defined in Equation 14 to measure the throughput of concurrent running benchmark pairs. Note that taller bar in the figure means higher throughput. We can see that SEPF gains a competitive throughput for most benchmark pairs. And for *gzip+mcf*, *ammp+applu and swim+mcf*, SEPF can provider higher throughput than original Pseudo LRU and SECF. On average, SECF gains a fair speedup of 1.04 while SEPF gains a fair speedup of 1.07 on the base line of original Pseudo LRU replacement policy.

6 Related Work

Prior work has noticed the impact of cache sharing for concurrently running threads. [15] proposed to use hardware counters to estimate the cache miss-rate as a function of cache size, which can be used to optimize cache partition to minimum overall miss rate. [13] designed a runtime mechanism that partitions a shared cache according to the cache utility of concurrent running multiple applications. [15] and [13] both focused on optimization of overall miss rate. [6] pointed out the necessity of enforce fairness for co-scheduled workloads. [6] defined a set of fairness metric for cache sharing and proposed both static strategy and

dynamic mechanism to improve fairness. [4] proposed a framework to provide QoS for resources including shared caches. [14] designed architectural support for OS to manage shared caches.

Some researches indicates that decreasing in cache miss rate does not necessarily lead to performance improvement. [12] indicated that not every cache miss has an equal penalty because of the existing of MLP. By taking MLP related cost of each cache miss into account, [12] modified the standard LRU replacement policy and higher performance guaranteed. [2] analyzed the microarchitecture impact on MLP and developed a detailed model to relating MLP to overall performance.

There are researches of performance models for other architectures. [3] proposed a cycle accounting architecture for SMT processors to estimate the performance of each co-scheduled thread had they ran alone. [5] proposed a performance model for superscalar processors.

7 Conclusion

Uncontrolled sharing usually leads to unfair performance of concurrent workloads. That is, some workloads suffer a much more significant slowdown than other workloads. This phenomenon brings more problems such as priority inversion and thread starvation to operating system's process scheduler. Instead of enforcing ideal performance fairness directly, prior work addressing fairness issue of cache sharing mainly focuses on the fairness metrics of cache miss numbers or miss rates. However, because of the variation of cache miss penalty, fairness on cache miss cannot guarantee ideal fairness. Cache sharing management which directly addresses ideal performance fairness is needed for CMP systems.

This paper proposed the concept of performance fairness metric. We built a detailed model to analyze the performance impact of cache sharing. Guided by this model, we designed a hardware mechanism to enforcing performance fairness on shared cache. The hardware mechanism proposed in this paper is adaptive and hardware efficient. For comparison, the concept of cache miss fairness metric and a hardware mechanism to enforcing cache miss fairness are also introduced. We implemented these two cache partition schemes in a simulator. The experiment results showed that the proposed mechanism always improves the performance fairness metric, and can provide no worse throughput than the scenario without any management mechanism.

References

[1] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[2] Y. Chou, B. Fahs, and S. G. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, pages 76–89, 2004.

[3] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in smt processors. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 133–144, New York, NY, USA, 2009. ACM.

[4] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, pages 343–355, 2007.

[5] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 338, Washington, DC, USA, 2004. IEEE Computer Society.

[6] S. Kim, D. Ch, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *IEEE PACT*, pages 111–122, 2004.

[7] J. Lin, Q. Lu, X. Ding, Z. Zhang, and X. Zhang. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In *In HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[9] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacets general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

[10] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, pages 318–319, 2003.

[11] M. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 45–54, Feb. 2009.

[12] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ISCA-33*, pages 167–178, 2006.

[13] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO-39*, pages 423–432, 2006.

[14] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

[15] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2002.