MLP-Aware Dynamic Cache Partitioning

Miquel Moreto[1], Francisco J. Cazorla[2], Alex Ramirez[1,2], and Mateo Valero[1,2]

[1] Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors, Barcelona, Spain
HiPEAC European Network of Excellence
{mmoreto,aramirez,mateo}@ac.upc.edu,
[2] Barcelona Supercomputing Center – Centro Nacional de Supercomputación, Spain
francisco.cazorla@bsc.es

Abstract. Dynamic partitioning of shared caches has been proposed to improve performance of traditional eviction policies in modern multithreaded architectures. All existing cache partitioning algorithms work on the number of misses caused by each thread and treat all misses equally. However, it has been shown that cache misses cause different impact in performance depending on the distribution of the Memory Level Parallelism (MLP) of the application L2 misses: clustered misses share their miss penalty as they can be served in parallel, while isolated misses have a greater impact as the memory latency is not shared with other misses.

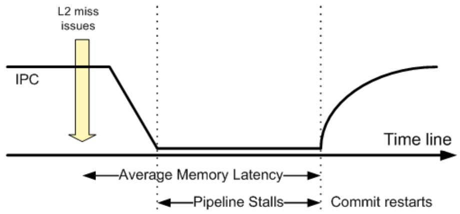
We take this fact into account and propose a new dynamic cache partitioning algorithm that considers misses differently depending on their influence in throughput. Our proposal obtains improvements over traditional traditional eviction policies up to 63.9% (10.6% on average) and it also outperforms previous dynamic partitioning proposals by up to 15.4% (4.1% on average) in a four-core architecture. Finally, we have used a sampling technique to propose a practical implementation with a hardware cost under 1% of the total L2 cache size.

1 Introduction

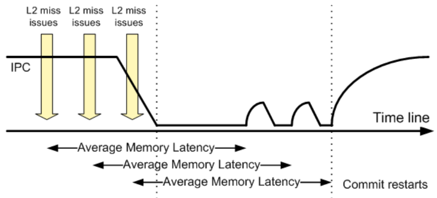
The limitation imposed by instruction-level parallelism (ILP) has motivated the use of thread-level parallelism (TLP) as a common strategy for improving processor performance. TLP paradigms such as simultaneous multithreading (SMT) [15, 20], chip multiprocessor (CMP) [4] and combinations of both offer the opportunity to obtain higher throughputs. However, they also have to face the challenge of sharing resources of the architecture. Simply avoiding any resource control can lead to undesired situations where one thread is monopolizing all the resources and harming the other threads. Some studies deal with the resource sharing problem in SMTs at core level resources like issue queues, registers, etc. [1]. In CMPs, resource sharing is lower than in SMT, focusing in the cache hierarchy.

Some applications present low reuse of their data and pollute caches with data streams, such as multimedia, communications or streaming applications, or have many compulsory misses that cannot be solved by assigning more cache space to the application. Traditional eviction policies such as Least Recently Used (LRU), pseudo LRU or random are demand-driven, that is, they tend to give more space to the application that has more accesses to the cache hierarchy. As a consequence, some threads can suffer a severe degradation in performance. Previous work has tried to solve this problem by using static and dynamic partitioning algorithms that monitor the L2 cache accesses and decide a partition for a fixed amount of cycles in order to maximize throughput [3,13,16,19] or fairness [7]. Basically, these proposals predict the number of misses per application for each possible cache partition. Then, they use the cache partition that leads to the minimum number of misses for the next interval.

A common characteristic of these proposals is that they treat all L2 misses equally. However, it has been shown that L2 misses affect performance differently depending on how clustered they are. An isolated L2 miss has approximately the same miss penalty than a cluster of L2 misses, as they can be served in parallel if they all fit in the reorder buffer (ROB) [6]. In Figure 1 we can see this behavior. We have represented an *ideal* IPC curve that is constant until an L2 miss occurs. After some cycles, commit stops. When the cache line comes from main memory, commit ramps up to its steady state value. As a consequence, an isolated L2 miss has a higher impact on performance than a miss in a burst of misses as the memory latency is shared by all clustered misses.



(a) Isolated L2 miss.



(b) Clustered L2 misses.

Fig. 1. Isolated and clustered L2 misses

Based on this fact, we propose a new dynamic cache partitioning algorithm that gives a cost to each L2 access according to its impact in final performance. We detect isolated and clustered misses and assign a higher cost to isolated misses. Then, our algorithm determines the partition that minimizes the total cost for all threads, which is used in the next interval. Our results show that differentiating between clustered and isolated L2 misses leads to dynamic cache partitions with higher performance than previous proposals. The main contributions of this work are the following.

1) A runtime mechanism to dynamically partition shared L2 caches in a CMP scenario that takes into account the MLP of each L2 access to improve throughput. We obtain improvements over LRU up to 63.9% (10.6% on average) and over previous proposals up to 15.4% (4.1% on average) in a four-core architecture.

2) We have extended previous classifications of workloads for CMP architectures with more than two cores. Results can be better analyzed in every workload group.

3) We give a sampling technique that reduces the hardware cost in terms of storage under 1% of the total L2 cache size with an average throughput degradation of 0.76% (compared to the throughput obtained without sampling).

The rest of this paper is structured as follows. In Section 2 we introduce the methods that have been previously proposed to decide L2 cache partitions and related work. Next, in Section 3 we explain our MLP-aware dynamic cache partitioning algorithm. In Section 4 we describe the experimetal environment and in Section 5 we discuss simulation results. Finally, we conclude with Section 6.

2 Prior work in Dynamic Cache Partitioning

In this Section, we first introduce the main method to predict the number of misses in a partitioned L2 cache: Stack Distance Histograms (SDHs). Then, we summarize previous work related with dynamic and static partitioning of shared L2 caches.

Stack Distance Histogram. Mattson et al. introduce the concept of stack distance to study the behavior of storage hierarchies [10]. Common eviction policies such as LRU have the *stack property*. Thus, each set in a cache can be seen as an LRU stack, where lines are sorted by their last access cycle. In that way, the first line of the LRU stack is the Most Recently Used (MRU) line while the last line is the LRU line. The position that a line has in the LRU stack when it is accessed again is defined as the *stack distance* of the access. As an example, we can see in Table 1(a) a stream of accesses to the same set with their corresponding stack distances.

For a K-way associative cache with LRU replacement algorithm, we need $K+1$ counters to build SDHs, denoted $C_1, C_2, \ldots, C_K, C_{>K}$. 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. If

Table 1. Stack Distance Histogram

(a) Stream of accesses to a given cache set.

# Reference	1	2	3	4	5	6	7	8
Cache Line	A	B	C	C	A	D	B	D
Stack Distance	-	-	-	1	3	-	4	2

(b) SDH example.

Stack Distance	1	2	3	4	>4
# Accesses	60	20	10	5	5

it is a cache miss, the line is not found in the LRU stack and, as a result, we increment the miss counter $C_{>K}$. SDH can be obtained during execution by running the thread alone in the system [3] or by adding some hardware counters that profile this information [13,19]. A characteristic of these histograms is that the number of cache misses for a smaller cache with the same number of sets can be easily computed. For example, for a K'-way associative cache, where $K' < K$, the new number of misses can be computed as:

$$misses = C_{>K} + \sum_{i=K'+1}^{K} C_i \tag{1}$$

As an example, in Table 1(b) we show a SDH for a set with 4 ways. In this example, 5 accesses have missed in the cache. However, if we had reduced the number of ways to 2 (keeping the number of sets constant), we would have experienced 20 misses (5 + 5 + 10).

Minimizing Total Misses. Using the SDHs of N applications, we can derive the L2 cache partition that would minimize the total number of misses: this last number corresponds to the sum of the number of misses of each thread with the assigned number of ways. The optimal partition in the last period of time is assumed to be a suitable candidate to become the future optimal partition. Partitions are decided periodically after a fixed amount of cycles. In this scenario, partitions are decided at a *way granularity*. This mechanism is used in order to minimize the total number of misses and try to maximize throughput. A first approach proposed a static partitioning of the L2 cache using profiling information [3]. Then, a dynamic approach estimated SDHs with information inside the cache [19]. Finally, Qureshi et al. presented a suitable and scalable circuit to measure SDHs using sampling and obtained performance gains with just 0.2% extra space in the L2 cache [13]. Throughout this paper, we will call this last policy *MinMisses*.

Fair Partitioning. In some situations, *MinMisses* can lead to unfair partitions that assign nearly all the resources to one thread while harming the others [7]. For that reason, the authors propose considering fairness when deciding new partitions. In that way, instead of minimizing the total number of misses, they try to equalize the statistic $X_i = \frac{misses_{shared_i}}{misses_{alone_i}}$ of each thread i. They desire to force all threads to have the same increase in percentage of misses. Partitions are decided periodically using an iterative method. The thread with largest X_i receives a way from the thread with smallest X_i until all threads have a similar value of X_i. Throughout this paper, we will call this policy *Fair*.

Table 2. Different Partitioning Proposals

Paper	Partitioning	Objective	Decision	Algorithm	Eviction Policy
[3]	Static	Minimize Misses	Programmer	–	Column Caching
[19]	Dynamic	Minimize Misses	Architecture	Marginal Gain	Augmented LRU
[13]	Dynamic	Maximize Utility	Architecture	Lookahead	Augmented LRU
[7]	Dynamic	Fairness	Architecture	Equalize X_1^i	Augmented LRU
[16]	Dynamic	Maximize reuse	Architecture	Reuse	Column Caching
[14]	Dynamic/Static	Configurable	Operating System	Configurable	Augmented LRU

Other Related Work. Several papers have the objective of dynamically adjusting the cache size assigned to each thread in a multithreaded scenario. In Table 2 we summarize these proposals with their most significant characteristics. Settle et al. introduce a dynamic scheme similar to

MinMisses that decides partitions depending on the average data reuse of each application [16]. Rafique et al. propose to manage shared caches with a hardware cache quota enforcement mechanism and an interface between the architecture and the OS to let the latter decide quotas [14]. We have to note that this mechanism is completely orthogonal to our proposal and, in fact, they are compatible as we can let the OS decide quotas according to our scheme. Hsu et al. evaluate different cache policies in a CMP scenario [5]. They show that none of them is optimal among all benchmarks and that the best cache policy varies depending on the performance metric being used. Thus, they propose to use a thread-aware cache resource allocation. In fact, their results reinforce the motivation of our paper: if we do not consider the impact of each L2 miss in performance, we can decide suboptimal L2 partitions in terms of throughput.

Cache partitions at a way granularity can be implemented with *column caching* [3], which uses a mask that marks the ways (or columns) reserved for each thread, or by augmenting the LRU policy with counters that keep track of the number of lines that a thread has in a set [19]. This second option assigns a quota of owned lines per set. The evicted line will be the LRU line among its owned lines or other threads lines depending on wether it reaches its quota or not.

In [12] a new eviction policy for *private* caches was proposed in single threaded processors. This policy gives a weight to each L2 miss according to its MLP when the block was filled from main memory. The evicted line is decided using the LRU counters and this weight. This idea was proposed for a different scenario as it focus on single threaded architectures.

3 MLP-Aware Dynamic Cache Partitioning

3.1 Algorithm overview

The algorithm steps to decide dynamic cache partitions according to the MLP of each L2 access can be seen in Algorithm 1.

Step 1: Establish an initial even partition for each core ;
Step 2: Run threads and collect data for the MLP-aware SDHs ;
Step 3: Decide new partition ;
Step 4: Update MLP-aware SDHs ;
Step 5: Go back to Step 2 ;

Algorithm 1: MLP-Aware dynamic cache partitioning algorithm.

When we start executing different applications in our CMP architecture, we have to decide an initial partition of the L2 cache. As we have no prior knowledge of the applications, we choose to assign the same amount of L2 cache space to each core. Thus, we assign $\frac{Associativity}{Number\ of\ Cores}$ ways to each core.

Afterwards, we begin a period of measuring the total MLP cost of each application. We denote MLP-aware SDH the histogram of each thread containing the total MLP cost for each possible partition. The length of this measuring interval is a parameter of the partitioning algorithm. For small values of this period, the partitioning algorithm reacts quicker to phase changes. However, the overhead of this method also increases. Once again, small performance variations are obtained for different periods ranging from 10^5 to 10^8 cycles. We observe that for longer periods throughput tends to decrease. The peak performance is obtained with a period of 5 million cycles.

When this interval ends, MLP-aware SDHs are analyzed and a new partition is decided for the next interval. We assume that we will have a similar pattern of L2 accesses in the next measuring period. Thus, the optimal partition for the last period will be chosen for the following period. Evaluating all possible combinations gives the optimal partition. However, this algorithm does not scale adequately when associativity and the number of cores is raised. If we have a K-way associativity L2 cache shared by N cores, the number of possible partitions without considering the order is $\binom{N+K-1}{K}$. For example, for 8 cores and 16 ways, we have 245157 possible combinations.

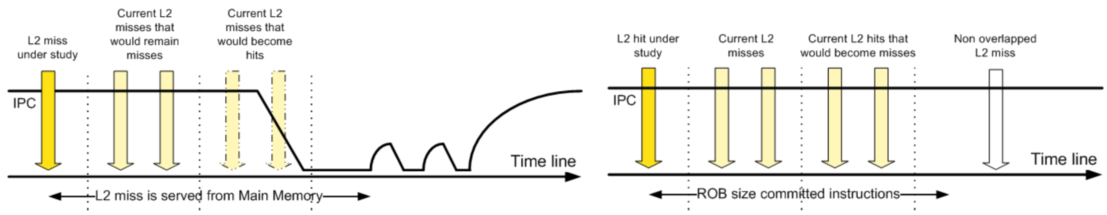
Several heuristics have been proposed in previous work to reduce the number of cycles required to decide the new partition [13, 19], which can be used in our situation. These proposals bounded the decision period by 10000 cycles. If we have a partition period of 5 million cycles, this overhead becomes very low (under 0.2%).

Since characteristics of applications dynamically change, the estimation of the MLP-aware SDHs should reflect these changes. However, we also wish to maintain some history of the past MLP-aware SDHs to make new decisions. Thus, after a new partition is decided, we multiply all the values of the MLP-aware SDHs times $\rho \in [0, 1]$. Large values of ρ have larger reaction times to phase changes, while small values of ρ quickly adapt to phase changes but tend to forget the forget the behavior of the application. Small variations are obtained in average throughput for different values of ρ ranging from 0 to 1, with a peak for $\rho = 0.5$. Furthermore, this value is very convenient as we can use a shifter to update histograms. Next, a new period of measuring MLP-aware SDHs begins. The key contribution of this paper is the method to obtain MLP-aware SDHs that we explain in the following Subsection.

3.2 MLP-Aware Stack Distance Histogram

As previously stated, *MinMisses* assumes that all L2 accesses are equally important in terms of performance. However, this is not always true. Cache misses affect differently the performance of applications, even inside the same application. Thus, L2 misses are not equally important for final performance. As was said in [6], an isolated L2 data miss has a penalty cost that can be approximated by the average memory latency. In the case of a burst of L2 data misses that fit in the ROB, the penalty cost is shared among misses as L2 misses can be served in parallel. In case of L2 instruction misses, they are serialized as fetch stops. Thus, L2 instruction misses have a constant miss penalty and MLP.

We want to assign a cost to each L2 access according to its effect on performance. In [12] a similar idea was used to modify LRU eviction policy for single core and single threaded architectures. In our situation, we have a CMP scenario where the shared L2 cache has a number of reserved ways for each core. At the end of a measuring period, we can decide to continue with the same partition or change it. If we decide to modify the partition, a core i that had w_i reserved ways will receive $w'_i \neq w_i$. If $w_i < w'_i$, the thread receives more ways and, as a consequence, some misses in the old configuration will become hits. Conversely, if $w_i > w'_i$, the thread receives less ways and some hits in the old configuration will become misses. Thus, we want to have an estimation of the performance effects when misses are converted into hits and vice versa. Throughout this paper, we will call this impact on performance MLP_cost. All accesses are treated as if they were in the correct path until the branch prediction is checked. All misses on the wrong path are not considered as accesses in flight.



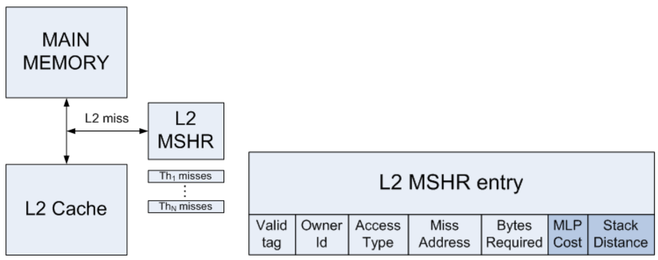
(a) MLP_cost of an L2 miss.

(b) Estimated MLP_cost when an L2 hit becomes a miss.

Fig. 2. MLP_cost of L2 accesses.

MLP_cost of L2 misses. In order to compute the MLP_cost of an L2 miss with stack distance d_i, we consider the situation shown in Figure 2(a). If we forced an L2 configuration that assigned exactly $w'_i = d_i$ ways to the thread i with $w'_i > w_i$, some of the L2 misses of this thread will become hits, while other will remain as misses, depending on their stack distance.

In order to track the stack distance and *MLP_cost* of each L2 miss, we have modified the L2 Miss Status Holding Registers (MSHR) [8]. This structure is similar to an L2 miss buffer and is used to hold information about any load that has missed in the L2 cache. The modified L2 MSHR has one extra field that contains the *MLP_cost* of the miss as can be seen in Figure 3(b). It is also necessary to store the stack distance of each access in the MSHR. In Figure 3(a) we show the MSHR in the cache hierarchy.



(a) MSHR. (b) MSHR fields.

Fig. 3. Miss Status Holding Register.

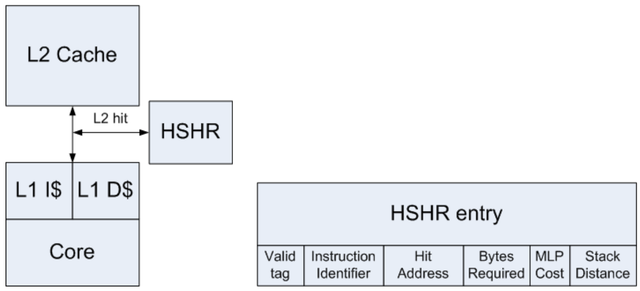
When the L2 cache is accessed and an L2 miss is determined, we assign an MSHR entry to the miss and wait until the data comes from Main Memory. We initialize the *MLP_cost* field to zero when the entry is assigned. We store the access stack distance together with the identificator of the owner core. Every cycle, we obtain N, the number of L2 accesses with stack distance greater or equal to d_i. We have a hardware counter that tracks this number for each possible number of d_i, which means a total of *Associativity* counters. If we have N L2 misses that are being served in parallel, the miss penalty is shared. Thus, we assign an equal share of $\frac{1}{N}$ to each miss. The value of the *MLP_cost* is updated until the data comes from Main Memory and fills the L2. At this moment we can free the MSHR entry.

MLP_cost of L2 hits. Next, we want to estimate the *MLP_cost* of an L2 hit with stack distance d_i when it becomes a miss. If we forced an L2 configuration that assigned exactly $w'_i = d_i$ ways to the thread i with $w'_i < w_i$, some of the L2 hits of this thread would become misses, while L2 misses would remain as misses (see Figure 2(b)). The hits that would become misses are the ones with stack distance greater or equal to d_i. Thus, we count the total number of accesses with stack distance greater or equal to d_i (including L2 hits and misses) to estimate the length of the cluster of L2 misses in this configuration.

Deciding the moment to free the entry used by an L2 hit is more complex than in the case of the MSHR. As it was said in [6], in a balanced architecture, L2 data misses can be served in parallel if they all fit in the ROB. Equivalently, we say that L2 data misses can be served in parallel if they are at ROB distance smaller than the ROB size. Thus, we should free the entry if the number of committed instructions since the access has reached the ROB size or if the number of cycles since the hit has reached the average latency to memory. The first condition is clear as we have said that L2 misses can overlap if their ROB distance is less than the ROB size. The second condition is also necessary as it can occur that no L2 access is done for a period of time. To obtain the average latency to memory, we add a specific hardware that counts and averages the number of cycles that a given entry is in the MSHR.

We use new hardware to obtain the *MLP_cost* of L2 hits. We denote this hardware Hit Status Holding Registers (HSHR) as it is similar to the MSHR. However, the HSHR is private for each core. In each entry, the HSHR needs an identificator of the ROB entry of the access, the address accessed by the L2 hit, the stack distance value and a field with the corresponding *MLP_cost* as can be seen in Figure 4(b). In Figure 4(a) we show the HSHR in the cache hierarchy.

When the L2 cache is accessed and an L2 hit is determined, we assign an HSHR entry to the L2 hit. We init the fields of the entry as in the case of the MSHR. We have a stack distance d_i



(a) HSHR. (b) HSHR fields.

Fig. 4. Hit Status Holding Register.

and we want to update the MLP_cost field in every cycle. With this objective, we need to know the number of active entries with stack distance greater or equal to d_i in the HSHR, which can be tracked with one hardware counter per core. We also need a ROB entry identificator for each L2 access. Every cycle, we obtain N, the number of L2 accesses with stack distance greater or equal to d_i as in the L2 MSHR case. We have a hardware counter that tracks this number for each possible number of d_i, which means a total of *Associativity* counters.

In order to avoid array conflicts, we need as many entries in the HSHR as possible L2 accesses in flight. This number is equal to the L1 MSHR size. In our scenario, we have an L1 MSHR of 32 entries, which means a maximum of 32 in flight L2 accesses per core. However, we have checked that we have enough with 24 entries to ensure that we have an available slot 95% of the time in an architecture with a ROB of 256 entries. If there are no available slots, we simply assign the minimum weight to the L2 access as there are many L2 accesses in flight.

Quantification of MLP_cost. Dealing with values of MLP_cost between 0 and the memory latency (or even greater) can represent a significant hardware cost. Instead, we decide to quantify this MLP_cost with an integer value between 0 and 7 as was done in [12]. For a memory latency of 300 cycles, we can see in Table 3 how to quantify the MLP_cost. We have splitted the interval $[0; 300]$ with 7 intervals of equal length.

Table 3. MLP_cost quantification

MLP_cost	Quantified MLP_cost
From 0 to 42 cycles	0
From 43 to 85 cycles	1
From 86 to 128 cycles	2
From 129 to 170 cycles	3
From 171 to 213 cycles	4
From 214 to 246 cycles	5
From 247 to 300 cycles	6
300 or more cycles	7

Finally, when we have to update the corresponding MLP-aware SDH, we add the quantified value of MLP_cost. In contrast, *MinMisses* always adds one to its histograms. Thus, isolated L2 misses will have a weight of 7, while two overlapped L2 misses will have a weight of 3 in the MLP-aware SDH.

3.3 Obtaining Stack Distance Histograms

Normally, L2 caches have two separate parts that store data and address tags to know if the access is a hit. Basically, our prediction mechanism needs to track every L2 access and store a separated copy of the L2 tags information in an *Auxiliary Tag Directory* (ATD), together with the LRU counters [13]. We need an ATD for each core that keeps track of the L2 accesses for any possible cache configuration. Independently of the number of ways assigned to each core, we store the tags and LRU counters of the last K accesses of the thread, where K is the L2 associativity. As we have explained in Section 2, an access with stack distance d_i corresponds to a cache miss in any configuration that assigns less than d_i ways to the thread. Thus, with this ATD we can determine whether an L2 access would be a miss or a hit in all possible cache configurations.

3.4 Putting all together

In Figure 5 we can see a sketch of the hardware implementation of our proposal. When we have an L2 access, the ATD is used to determine its stack distance d_i. Depending on whether it is a miss or a hit, either the MSHR or the HSHR is used to compute the MLP_cost of the access. Using the quantification process we obtain the final MLP_cost. This number estimates how performance is affected when the applications has exactly $w_i' = d_i$ assigned ways. If $w_i' > w_i$, we are estimating the performance benefit of converting this L2 miss into a hit. In case $w_i' < w_i$, we are estimating the performance degradation of converting this L2 hit into a miss. Finally, using the stack distance, the MLP_cost and the core identifier, we can update the corresponding MLP-aware SDH.

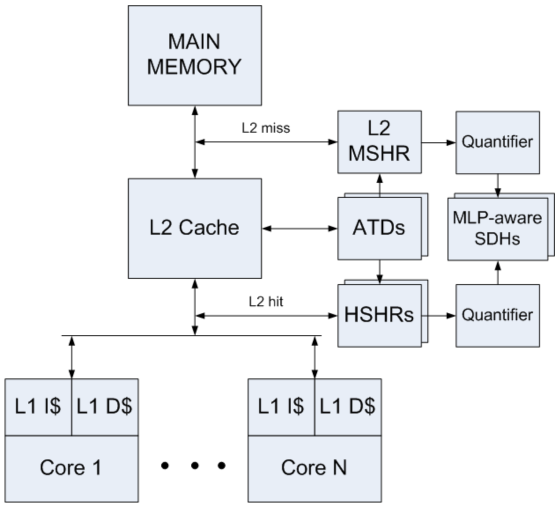


Fig. 5. Hardware implementation.

We have used two different partitionig algorithms. The first one, that we denote *MLP-DCP* (standing for MLP-aware Dynamic Cache Partitioning), decides the optimal partition according to the MLP_cost of each way. We define the total MLP_cost of a thread i that uses w_i ways as $TMLP(i, w_i) = MLP\_SDH_{i,>K} + \sum_{j=w_i}^{K} MLP\_SDH_{i,j}$. We denote the total MLP_cost of all accesses of thread i with stack distance j as $MLP\_SDH_{i,j}$. Thus, we have to minimize the sum of total MLP_costs for all cores:

$$\sum_{i=1}^{N} TMLP(i, w_i), \text{ where } \sum_{i=1}^{N} w_i = \text{Associativity}.$$

The second proposal consists in modifying the weight of each total MLP_cost using the IPC of the application running in core i, IPC_i. In this situation, we are giving priority to applications with higher IPC. This point will give better results in throughput at the cost of being less fair. IPC_i is measured at runtime with an extra hardware counter per core. We denote this proposal

MLPIPC-DCP and consists in minimizing the following expression:

$$\sum_{i=1}^{N} IPC_i \cdot TMLP(i, w_i), \text{ where } \sum_{i=1}^{N} w_i = \text{Associativity}.$$

Both proposals have two important paramaters. First, we have the length of the period between repartition decisions. In our experiments we have used a period of 5 Million cycles. The second one is the multiplicative factor $\rho \in [0,1]$ that updates MLP-aware SDH after each repartition. We have chosen $\rho = 0.5$.

3.5 Case Study

We have seen that SDHs can give the optimal partition in terms of total L2 misses. However, total number of L2 misses is not the goal of dynamic partitioning algorithms. Throughput is the objective of these policies. The underlying idea of *MinMisses* is that while minimizing total L2 misses, we are also increasing throughput. This idea is intuitive as performance is clearly related to L2 missrate. However, this heuristic can lead to inadequate partitions in terms of throughput as can be seen in the next case study.

In Figure 6, we can see the IPC curves of benchmarks `galgel` and `gzip` as we increase L2 cache size in a way granularity (each way has a 64KB size). We also show throughput for all possible 15 partitions. In this curve, we assign x ways to `gzip` and $16 - x$ to `galgel`. The optimal partition consists in assigning 6 to `gzip` and 10 ways to `galgel`, obtaining a total throughput of 3.091 instructions per cycle. However, if we use MinMisses algorithm to determine the new partition, we will choose 4 to `gzip` and 12 ways to `galgel` according to the SDHs values. In Figure 6 we can also see the total number of misses for each cache partition as well as the per thread number of misses.

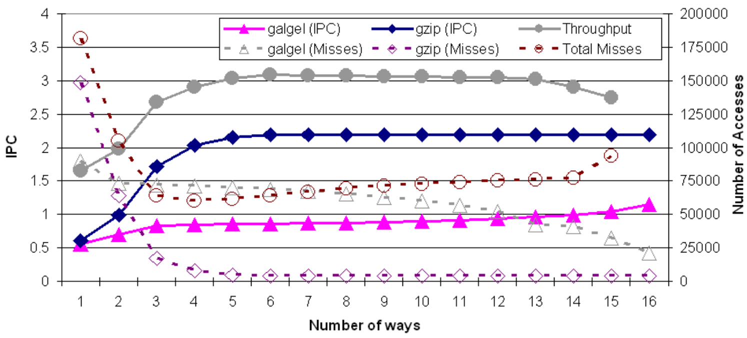


Fig. 6. Misses and IPC curves for `galgel` and `gzip`.

In this situation, misses in `gzip` are more important in terms of performance than misses in `galgel`. Furthermore, `gzip` IPC is larger than `galgel` IPC. As a consequence MinMisses obtains a non optimal partition in terms of IPC and its throughput is 2.897, which is a 6.3% smaller than the optimal one.

In fact, `galgel` clusters of L2 misses are, in average, longer than the ones from `gzip`. In that way, *MLP-DCP* assigns one extra way to `gzip` and increases performance by 3%. If we use *MLPIPC-DCP*, we are giving more importance to `gzip` as it has a higher IPC and, as a consequence, we end up assigning another extra way to `gzip`, reaching the optimal partition and increasing throughput an extra 3%.

4 Experimental Environment

We have targeted this study to the case of a CMP with two and four cores with their respective own data and instruction L1 caches and a unified L2 cache shared among threads as in previous studies [7,13,19]. In this situation, the effects of the partitioning algorithm can be analyzed easier as there is no collision with effects concerning other shared resources as in the case of an SMT. Each core is single threaded and can fetch up to 8 instructions each cycle. It has 6 integer (I), 3 floating point (FP), and 4 load/store functional units and 32-entry I, load/store, and FP instruction queues. Each thread has its own 256-entry ROB and 256 physical registers. We use a two-level cache hierarchy with 64B lines with separate 16KB, 4-way associative data and instruction caches, and a unified L2 cache that is shared among all cores. We have used two different L2 caches, one of size 1MB and 16-way associativity, and the second one of size 2MB and 32-way associativity. Latency from L1 to L2 is 15 cycles, and from L2 to memory 300 cycles. We use a 32B width bus to access L2 and a multibanked L2 of 16 banks with 3 cycles of access time.

We extended the SMTSim simulator [20] to make it CMP. We collected traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology [17]. We use the FAME simulation methodology [21] with a Maximum Allowable IPC Variance of 5%. This evaluation methodology measures the performance of multithreaded processors by re-executing all threads in a multithreaded workload until all of them are fairly represented in the final IPC taken from the workload. As performance metrics we have used the IPC throughput, which corresponds to the sum of individual IPCs. Average IPCs are computed using the harmonic mean of IPC throughputs.

5 Evaluation Results

In this Section, we first characterize the workloads that we use and, then, evaluate the performance of our MLP-aware dynamic cache partitioning algorithms for two- and four-core architectures.

5.1 Workload classification

In [11] two metrics are used to model the performance of a partitioning algorithm like *MinMisses* for pairings of benchmarks in the SPEC CPU 2000 benchmark suite. Here, we extend this classification for architectures with more cores.

Metric 1. The $w_{P\%}(B)$ metric measures the number of ways needed by a benchmark B to obtain at least a given percentage $P\%$ of its maximum IPC (when it uses all L2 ways).

The intuition behind this metric is to classify benchmarks depending on their cache utilization. Using $P = 90\%$ we can classify benchmarks into three groups: *Low utility* (L), *Small Working Set* or *Saturated utility* (S) and *High utility* (H). L benchmarks have $1 \leq w_{90\%} \leq \frac{K}{8}$ where K is the L2 associativity. L benchmarks are not affected by L2 cache space because nearly all L2 accesses are misses. S benchmarks have $\frac{K}{8} < w_{90\%} \leq \frac{K}{2}$ and just need some ways to have maximum throughput as they fit in the L2 cache. Finally, H benchmarks have $w_{90\%} > \frac{K}{2}$ and always improve their performance as the number of ways given to them is increased. Clear representatives of these three groups are `applu` (L), `gzip` (S) and `ammp` (H) in Figure 7.

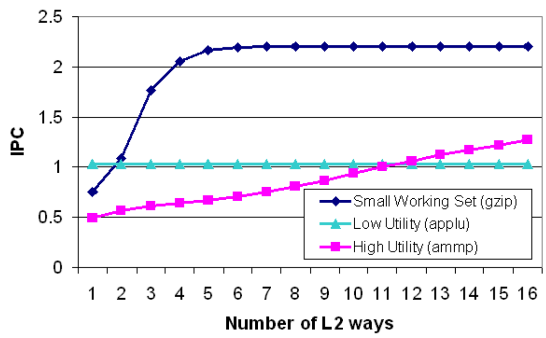


Fig. 7. Throughput as we vary the number of assigned ways of a 1MB 16-way associativity L2 cache.

In Table 4 we can see $w_{90\%}$ for all SPEC CPU 2000 benchmarks. Next, we list the benchmarks that belong to each group.

– *High Utility*: ammp, apsi, art, facerec, fma3d, galgel, mgrid, parser, twolf and vpr.
– *Small Working Set or Saturated Utility*: crafty, eon, gcc, gzip, perl and vortex.
– *Low Utility*: applu, bzip2, equake, gap, lucas, mcf, mesa, sixtrac, swim and wupwise.

Table 4. Benchmark characterization

Bench	$w_{90\%}$	APTC	IPC	Bench	$w_{90\%}$	APTC	IPC
ammp	14	23.63	1.27	applu	1	16.83	1.03
apsi	10	21.14	2.17	art	10	46.04	0.52
bzip2	1	1.18	2.62	crafty	4	7.66	1.71
eon	3	7.09	2.31	equake	1	18.6	0.27
facerec	11	10.96	1.16	fma3d	9	15.1	0.11
galgel	15	18.9	1.14	gap	1	2.68	0.96
gcc	3	6.97	1.64	gzip	4	21.5	2.20
lucas	1	7.60	0.35	mcf	1	9.12	0.06
mesa	2	3.98	3.04	mgrid	11	9.52	0.71
parser	11	9.09	0.89	perl	5	3.82	2.68
sixtrack	1	1.34	2.02	swim	1	28.0	0.40
twolf	15	12.0	0.81	vortex	7	9.65	1.35
vpr	14	11.9	0.97	wupw	1	5.99	1.32

We have measured the average miss penalty of an L2 miss for the whole SPEC CPU 2000 benchmark suite. Results are shown in Figure 8. We note that this average miss penalty varies a lot, even inside each group of benchmarks, ranging from 30 to 294 cycles. This Figure reinforces the main motivation of the paper, as it proves that the clustering level of L2 misses changes for different applications.

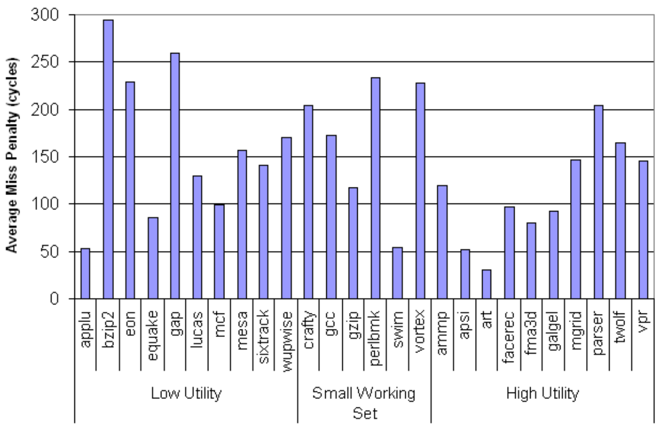


Fig. 8. Average miss penalty of an L2 miss for SPEC CPU 2000 benchmarks with a 1MB 16-way L2 cache.

Metric 2. The $w_{LRU}(th_i)$ metric measures the number of ways given by LRU to each thread th_i when N benchmarks run together. This can be done simulating all benchmarks alone and using the total number of L2 accesses in a fixed period of cycles for each benchmark [2]. We denote the number of L2 Accesses in a Period of one Thousand Cycles for thread i as $APTC_i$. In Table 4 we list these values for each benchmark.

$$w_{LRU}(th_i) = Associativity \cdot \frac{APTC_i}{\sum_{j=1}^{N} APTC_j}$$

In [11], the authors use these two metrics to classify pairings of benchmarks into three groups and model the behavior of pairings when a partitioning policy like *MinMisses* is used. Next, we extend this classifications to workloads with more than two benchmarks.

Case 1. When $w_{90\%}(th_i) \leq w_{LRU}(th_i)$ for all threads. In this situation LRU attains 90% of each benchmark performance. Thus, it is intuitive that in this situation there is very little room for improvement.

Case 2. When there exists two threads A and B such that $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$. In this situation, LRU is harming the performance of thread A, because it gives more ways than necessary to thread B. Thus, in this situation LRU is assigning some shared resources to a thread that does not need them, while the other thread could benefit from these resources. In [11] it was observed that the higher the difference $w_{90\%}(th_A) - w_{LRU}(th_A)$, the higher performance benefits for architectures with two cores.

Case 3. Finally, the third case is obtained when $w_{90\%}(th_i) > w_{LRU}(th_i)$ for all threads. In this situation, our L2 cache configuration is not big enough to assure that all benchmarks will have at least a 90% of their peak performance. In [11] it was observed that pairings belonging to this group showed worse results when the value of $|w_{90\%}(th_1) - w_{90\%}(th_2)|$ grows. In this case, we have a thread that requires much less L2 cache space than the other to attain 90% of its peak IPC. LRU treats threads equally and manages to satisfy the less demanding thread necessities. In case of *MinMisses*, it assumes that all misses are equally important for throughput and tends to give more space to the thread with higher L2 cache necessity, while harming the less demanding thread. This is a problem due to *MinMisses* algorithm. We will show in next Subsections that MLP-aware partitioning policies are available to overcome this situation.

Table 5. Workloads belonging to each case for a 16-way 1MB and a 32-way 2MB shared L2 caches.

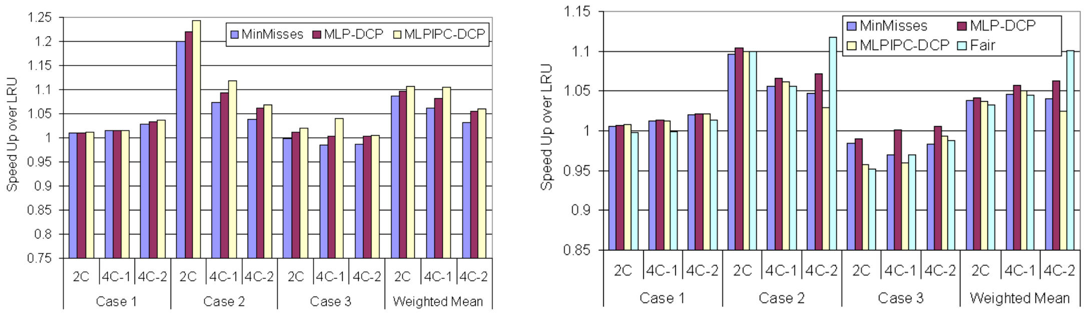
	1MB 16-way			2MB 32-way		
# Cores	Case 1	Case 2	Case 3	Case 1	Case 2	Case 3
2	155 (48%)	135 (41%)	35 (11%)	159 (49%)	146 (45%)	20 (6.2%)
4	624 (4%)	12785 (86%)	1541 (10%)	286 (1.9%)	12914 (86%)	1750 (12%)
6	306 (0.1%)	219790 (95%)	10134 (4.5%)	57 (0.02%)	212384 (92%)	17789 (7.7%)
8	19 (0%)	1538538 (98%)	23718 (2%)	1 (0%)	1496215 (96%)	66059 (4.2%)

In Table 5 we can see the total number of workloads that belong to each case as the number of cores increases. We have generated all possible combinations without repeating benchmarks. The order of benchmarks is not important. These numbers are obtained considering a constant baseline cache hierarchy with a shared L2 of 1MB and associativity of 16. When we increase the size of the L2 cache while keeping constant associativity, some benchmarks reduce their value of $w_{90\%}$ (the size of one way has also increased), while others increase the value of $w_{90\%}$ (the working set now fits in the L2 cache). As a result, with an L2 cache of 2MB and 16-way associativity, Case 2 remains as the dominant case. The same trend is observed for L2 caches with larger associativity. In Table 5 we can also see the total number of workloads that belong to each case as the number of cores increases for a 32-way 2MB L2 cache. Note that with different L2 cache configurations, the value of $w_{90\%}$ and $APTC_i$ will change for each benchmark. An important conclusion from Table 5 is that as we increase the number of cores, there are more combinations that belong to the second case, which is the one with more improvement possibilities.

To evaluate performance results, we randomly generate 16 workloads belonging to each group for three different configurations. We use a 16-way 1MB L2 cache for a two- and four-core architecture and a 32-way 2MB L2 cache for a four-core architecture. We denote these configurations *2C* (2 cores and 1MB L2), *4C-1* (4 cores and 1MB L2) and *4C-2* (4 cores and 2MB L2). We have also used a 32-way 2MB L2 cache as future CMP architectures will continue scaling L2 size and associativity. For example, the IBM Power5 [18] has a 10-way 1.875MB L2 cache and the Niagara 2 has a 16-way 4MB L2. In that way, we composed a total of 48 workloads for the different configurations and reported average speed ups over LRU in the three cases. Average improvements of each configuration do consider the distribution of workloads among the three groups. We denote this mean *weighted mean*, as we assign a weight to the speed up of each case depending on the distribution of workloads from Table 5. For example, for the *2C* configuration, we compute the weighted mean improvement as $0.48 \cdot x_1 + 0.41 \cdot x_2 + 0.11 \cdot x_3$, where x_i is the average improvement in Case i.

5.2 Performance Results

Throughput. The first experiment consists in comparing the throughput for different dynamic partitioning algorithms. We have used as baseline the results of the traditional LRU policy. We have simulated *MinMisses* and our two proposals with the 48 workloads that were selected in the previous Subsection. We can see in Figure 9(a) the average speed up over LRU for these mechanisms. We can see that *MLPIPC-DCP* systematically obtains the best average results, nearly doubling the performance benefits of *MinMisses* over LRU in the four-core configurations. In configuration *4C-1*, *MLPIPC-DCP* outperforms *MinMisses* by a 4.1%. *MLP-DCP* always improves *MinMisses* but obtains worse results than *MLPIPC-DCP*.



(a) Average throughput speed up over LRU. (b) Average fairness speed up over LRU.

Fig. 9. Performance speed ups over LRU.

All algorithms have similar results in Case 1. This is intuitive as in this situation there is little room for improvement. In Case 2, *MinMisses* obtains a relevant improvement over LRU in configuration *2C*. *MLP-DCP* and *MLPIPC-DCP* achieve an extra 2.5% and 5% improvement, respectively. In the other configurations, *MLP-DCP* and *MLPIPC-DCP* still outperform MinMisses by a 2.1% and 3.6%. In Case 3, *MinMisses* presents larger performance degradation as the asymmetry between the necessities of the two cores increases. As a consequence, it has worse average throughput than LRU. Assigning an appropiate weight to each L2 access gives the possibility to obtain better results than LRU using *MLP-DCP* and *MLPIPC-DCP*.

Fairness. In order to measure fairness, we have used the harmonic mean of relative IPCs [9]. The relative IPC is computed as $\frac{IPC_{shared}}{IPC_{alone}}$. In Figure 9(b) we show the average speed up over LRU of the harmonic mean of relative IPCs. *Fair* stands for the policy explained in Section 2. We can see that in all situations, *MLP-DCP* always improves over both *MinMisses* and LRU (except in Case 3 for two cores). It even obtains better results than *Fair* in configurations *2C* and *4C-1*. *MLPIPC-DCP* is a variant of the *MLP-DCP* algorithm optimized for throughput. As a consequence, it obtains worse results in fairness than *MLP-DCP*.

5.3 Hardware Cost

We have used the hardware implementation of Figure 5 to estimate the hardware cost of our proposal. In this Subsection, we focus our attention on the configuration *2C*. We suppose a 40-bit physical address space. Each entry in the ATD needs 29 bits (1 valid bit + 24-bit tag + 4-bit for

LRU counter). Each set has 16 ways, so we have an overhead of 58 Bytes (B) for each set. As we have 1024 sets, we have a total cost of 58KB per core.

The hardware cost that corresponds to the extra fields of each entry in the L2 MSHR is 5 bits for the stack distance and 2B for the MLP_cost field. As we have 32 entries, we have a total of 84B. HSHR entries need 1 valid bit, 8 bits to identify the ROB entry, 34 bits for the address, 5 bits for the stack distance and 2B for the MLP_cost field. In total we need 64 bits per entry. As we have 24 entries in each HSHR, we have a total of 192B per core. Finally, we need 17 counters of 4B for each MLP-Aware SDH, which supposes a total of 68B per core. In addition to the storage bits, we also need an adder for incrementing MLP-aware SDHs and a shifter to halve the hit counters after each partitioning interval.

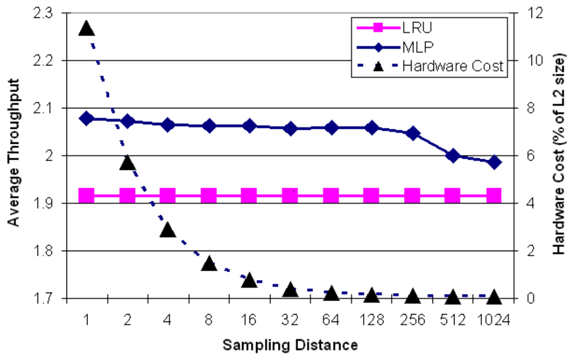


Fig. 10. Average IPC and hardware cost depending on the sampling distance in a two-core CMP

Sampled ATD. It is clear that our main contribution to hardware cost corresponds to the ATD, for which we propose to use a sampled version of the ATD to reduce its cost. Instead of monitoring every cache set, we can decide to track accesses from a reduced number of sets. This idea was also used in [13] in a CMP environment to determine the cache partition that minimizes the total number of misses using a sampled number of sets. Instead, we use it in a different situation, say to estimate MLP-aware SDHs with a sampled number of sets. We define a *sampling distance* d_s that gives the distance between sampled sets. For example, if $d_s = 1$, we are tracking all the sets. If $d_s = 2$, we track half of the sets, and so on. Sampling reduces the size of the ATD at the expense of less accuracy in MLP-aware SDHs predictions. In this situation, some accesses are not tracked and, as a consequence, the information in the MLP-aware SDHs is always less than real values.

Figure 10 shows throughput degradation in a 2 cores scenario as the sampling distance increases. This curve is measured on the left y-axis. We also show the storage overhead in percentage of the total L2 cache size, measured on the right y-axis. Thanks to the sampling technique, storage overhead drastically decreases with sampling distance. Thus, with a sampling distance of 16 we obtain average throughput degradations of 0.76% and a storage overhead of 0.77% of the L2 cache size. We think that this is an interesting point of design.

6 Conclusions

In this paper we propose a new dynamic cache partitioning algorithm that gives a cost to each L2 access according to its impact in final performance: isolated misses receive higher costs than clustered misses. Next, our algorithm decides the L2 cache partition that minimizes the total cost for all running threads. Furthermore, we have classified workloads for multiple cores into three groups and shown that the dominant situation is precisely the one that offers room for improvement.

We have evaluated our proposal and shown that it reaches high throughput for two- and four-core architectures. In the three configurations that we have simulated, *MLP-DCP* and *MLPIPC-DCP* systematically outperform both LRU and *MinMisses*, reaching a speed up of 63.9% (10.6%

on average) and 15.4% (4.1% on average) over LRU and *MinMisses*, respectively. Finally, we have used a sampling technique to propose a practical implementation with a hardware cost in terms of storage under 1% of the total L2 cache size with nearly no performance degradation.

Acknowledgments

This work has been supported by the Ministry of Education and Science of Spain under contracts TIN2004-07739-C02-01, TIN2007-60625, and grant AP-2005-3318. The authors would like to thank the support of the SARC European Project (Contract number 27648). They also thank Carmelo Acosta, Ayose Falcon, Daniel Ortega, Jeroen Vermoulen and Oliverio J. Santana for their work in the simulation tool and the reviewers for their helpful comments..

References

1. F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *MICRO*, 2004.
2. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.
3. D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.
4. L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.
5. L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, 2006.
6. T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.
7. S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.
8. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, 1981.
9. K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.
10. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
11. M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Explaining dynamic cache partitioning speed ups. *IEEE CAL*, 2007.
12. M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, 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*, 2006.
14. N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.
15. M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.
16. A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 1(3-4), 2005.
17. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.
18. B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.
19. G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.
20. D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA*, 1995.
21. J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *PACT*, 2007.