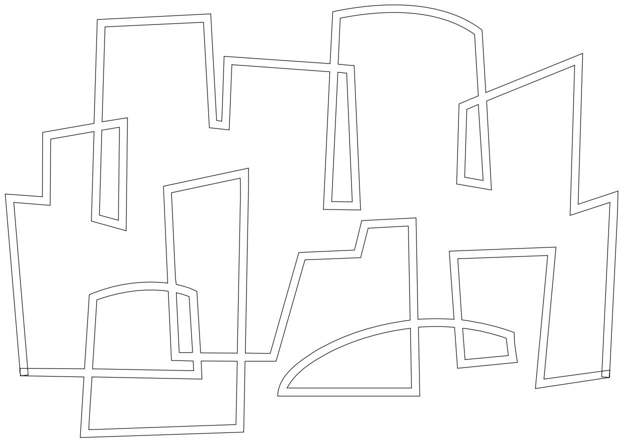
Dynamic Cache Partioning for Simultaneous Multithreading Systems

Ed Suh, Larry Rudolph, Srinivas Devadas

2001, March

Computation Structures Group
Memo 438



Dynamic Cache Partitioning for Simultaneous Multithreading Systems

Computation Structures Group Memo 438
March 2001

G. Edward Suh, Larry Rudolph and Srinivas Devadas
`email: {suh,rudolph,devadas}@mit.edu`

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Defense Advanced Research Projects Agency under the Air Force Research Lab contract F30602-99-2-0511.

Dynamic Cache Partitioning for Simultaneous Multithreading Systems

G. Edward Suh, Larry Rudolph and Srinivas Devadas
Laboratory for Computer Science
MIT
Cambridge, MA 02139
email: {suh,rudolph,devadas}@mit.edu

Abstract

This paper proposes a dynamic cache partitioning method for simultaneous multithreading systems. Unlike previous works, our method works for set-associative caches and at any partition granularity. Further, threads can have overlapping partitions in our method. The method collects the miss-rate characteristics of threads that are executing simultaneously at run-time, and dynamically partitions the cache among those threads. Trace-driven simulation results show a relative improvement in the L2 hit-rate up to 40.5% over those generated by the standard least recently used replacement policy. The partitioning also improves IPC up to 17%. Our results show that smart cache management and scheduling is essential for SMT systems to achieve high performance.

1 Introduction

Modern computer systems exploit two forms of parallelism, instruction-level (ILP) and thread-level (TLP) [10], however, performance is limitted by competition for processing resources. Simultaneous multithreading (SMT) [19, 13, 6] has been proposed as a way to better utilize resources. SMT converts TLP to ILP by executing multiple threads simultaneously and since it dynamically allocates processing resources to both forms of parallelism, it tends to achieve higher performance than conventional multiprocessors. Since multiple threads simultaneously execute, the working sets of those threads may interfere in the cache and the increased processor throughput places greater demands on the memory system. We have found that to achieve the best utilization of the functional resources, it is crucial to properly manage the memory system

It is easy to address this problem by having L2 and L3 caches that are large enough to hold all the working sets of executing threads. Studies have shown that a 256-KB L2 cache, which is reasonable size for modern microprocessors [8, 5, 14], is large enough for a particular set of workloads [13]. Unfortunately, we believe that workloads have become much larger and it is impractical to have a cache large enough to hold the entire working sets. Multimedia programs such as video or audio processing software often consume hundreds of MB. Even many SPEC CPU2000 benchmarks now have memory footprints larger than 100 MB [11]. Therefore, to achieve high performance for modern workloads, SMT systems should optimize the cache performance by proper allocation of cache space and careful selection of executing threads.

This paper presents a dynamic cache partitioning algorithm for SMT systems that minimizes the total cache misses. The algorithm estimates the miss-rate of each process on-line. Based on the information, a cache is partitioned to each thread dynamically. We develop two different approaches to cache partitioning. First, the standard LRU replacement unit is modified to consider the number of cache blocks for each thread deciding the block to be replaced. Second, column caching [3] which allows the threads to be assigned to overlapping partitions is used to partition the cache. Simulation experiments shows that the partitioning algorithm can significantly improve both the miss-rate and the instructions per cycle (IPC) of overall workload.

This paper is organized as follows. In Section 2, we describe related work. In Section 3, we study the optimal cache partitioning for the ideal case, and extend the study to real set-associative caches. Section 4 evaluates the partitioning method by simulations. Finally, Section 5 concludes the paper.

2 Related Work

Stone, Turek and Wolf [15] theoretically investigated the optimal allocation of cache memory between two competing processes that minimizes the overall miss-rate of a cache. Their study focuses on the partitioning of instruction and data streams, which can be thought of as multitasking with a very short time quantum. Their model for this case shows that the optimal allocation occurs at a point where the miss-rate derivatives of the competing processes are equal. The LRU replacement policy appears to produce cache allocations very close to optimal for their examples.

Suh [16] proposed an analytical cache model for multitasking, and also studied the cache partitioning problem for time-shared systems based on the model. That work is applicable to any length of time quantum rather than just short time quantum, and shows that the cache performance can be improved by partitioning a cache into dedicated areas for each process and a shared area. However, the partitioning was done by collecting the miss-rate information of each process off-line. The study did not investigate how to partition the cache memory at run-time.

Thiébaut, Stone and Wolf applied their theoretical partitioning study [15] to improve disk cache hit-ratios [17]. The model for tightly interleaved streams is extended to be applicable for more than two processes. They also describe the problems in applying the model in practice, such as approximating the miss-rate derivative, non-monotonic miss-rate derivatives, and updating the partition. Trace-driven simulations for 32-MB disk caches show that the partitioning improves the relative hit-ratios in the range of 1% to 2% over the LRU policy.

Our partition work differs from previous efforts that tend to focus on some ideal cases. First, our partition method works for set-associative caches with multiple threads, whereas Thiébaut, Stone and Wolf [17] only focused on disk caches that are fully-associative. Moreover, our work also covers cases when partitioning is only possible in coarse granularity.

Previous works only considered partitioning in cache block granularity. Finally, our work discusses an on-line method to partition the cache, whereas the previous work only covered partitioning based on off-line profiling [16].

3 Partitioning Algorithm

This section discusses the cache partitioning algorithm and its implementation for set-associative caches. Section 3.1 discusses the optimal cache partitioning for the ideal case. This is mostly a review of work suitable for disk caching where the cache is fully-associative and miss-rate information is easy to collect. The ideal algorithm is extended to real set-associative processor caches in the following three sections. First, the method to estimate each thread's miss-rate (or the number of misses) curve is presented. The search algorithm for convex miss-rate curves is also extended to non-convex curves. Then, coarse granularity partitioning is discussed focusing on evaluating the effect of overlapping partitions. Finally, the mechanisms to actually allocate cache blocks to each thread are presented.

3.1 Optimal Cache Partitioning

In conventional time-shared systems, where each thread executes for a while and then the context switches, cache partitioning depends not only on the memory reference pattern of each thread but also the thread that is active at the moment. On the other hand, SMT systems execute multiple threads at the same time, and as a result, interleave memory references from each thread very tightly. In this case, all threads can be considered active at any given moment. Therefore, cache partitioning in SMT systems depends only on the memory reference characteristics of executing threads.

Consider the case when N excuting threads simultaneously sharing C blocks. If the cache partition can be controlled in a cache block granularity, the problem can be thought of as a resource allocation problem that allocates each cache block to one of the executing threads to minimized the overall miss-rate. In our study, the miss-rate of each thread is assumed to be a function of partition size (the number of cache blocks). Note that the replacement policy tries to keep the same data no matter which physical cache blocks are available. Therefore, a cache partition is specified by the number of cache blocks allocated to each thread. We use the notation c_n to represent the number of cache blocks allocated to the n^{th} thread.

Since it is unreasonable to repartition the cache every memory reference, we define a time period during which the partition remains fixed. The time period should be long enought to amortize the cost of repartitioning. Assuming that the number of misses for the n^{th} thread's over a period is given by a function of partition size $(m_n(x))$, the optimal partition for the period is the set of integer values $\{c_1, c_2, ..., c_N\}$ that minimizes the following expression.

$$\text{misses} = \sum_{i=1}^{N} m_i(c_i) \tag{1}$$

where $\sum_{i=1}^{N} c_i = C$. This partition is optimal in a sense that it minimizes the total number of misses.

For the cases when the number of misses for each thread is a strict convex function of cache space, Stone, Turek and Wolf [15] pointed out that finding the optimal partition, $\{c_1, c_2, ..., c_N\}$, falls into the category of separable convex resource allocation problems. The problem has been heavily studied, and the following simple greedy algorithm can solve it [15, 7].

1. Compute the marginal gain $g_n(x) = m_n(x-1) - m_n(x)$. This function represents the additional hits for the n^{th} thread, when the allocated cache blocks increases from $x-1$ to x.

2. Initialize $c_1 = c_2 = ... = c_N = 0$.

3. Assign a cache block to the thread that has the maximum marginal gain. For each thread, compare marginal gain $g_n(c_n + 1)$ and find the thread that has the maximum marginal gain n_{max}. Increase the allocation for the thread $c_{n_{max}}$ by one.

4. Repeat step 3 until all cache blocks are assigned (C times).

3.2 Extension to Set-Associative Caches

To perform dynamic cache partitioning, the marginal gains of having one more cache block ($g_n(x)$) should be estimated on-line. As discussed in the previous section, $g_n(x)$ is the number of additional hits that the n^{th} thread can obtain by having $x+1$ cache blocks compared to the case when it has x blocks. Assuming the LRU replacement policy is used, $g_n(0)$ represents the number of hits on the most recently used cache block of the n^{th} thread, $g_n(1)$ represents the number of hits on the second most recently used cache block of the n^{th} thread, and so on.

Ideally, we should know the LRU ordering of all blocks in the cache to estimate the exact values of $g_n(x)$. Unfortunately, standard set-associative caches only maintain the LRU ordering of cache blocks within a set, and it is very expensive to maintain the global LRU ordering. Therefore, we approximate $g_n(x)$ based on the LRU ordering within a set.

For each thread, a cache has counters, one for each associativity (way) of the cache. On every cache hit, the corresponding counter is increased. That is, if the hit is on the most recently used cache block of the thread, the first counter is increased by one, and so on. Now the k^{th} counter value represents the number of additional hits for the thread by having k^{th} way. If we ignore the degradation due to low associativity, the k^{th} counter value can also be thought of as the number of additional hits for a cache with $k \cdot S$ blocks compared to a cache with $(k-1) \cdot S$ blocks, where S is the number of cache sets. Therefore, $g_n(x)$ satisfies the following equation.

$$\sum_{x=(k-1)\cdot S}^{k\cdot S-1} g_n(x) = count_n(k) \tag{2}$$

where $count_n(k)$ represents the k^{th} counter value of the n^{th} thread.

To estimate marginal gains from Equation 2, a certain form of $g_n(x)$ should be assumed. The simplest way is to assume that $g_n(x)$ is a straight line for x between $k \cdot S$ and $(k+1) \cdot S - 1$. This approximation is very simple to calculate and yet shows reasonable performance in

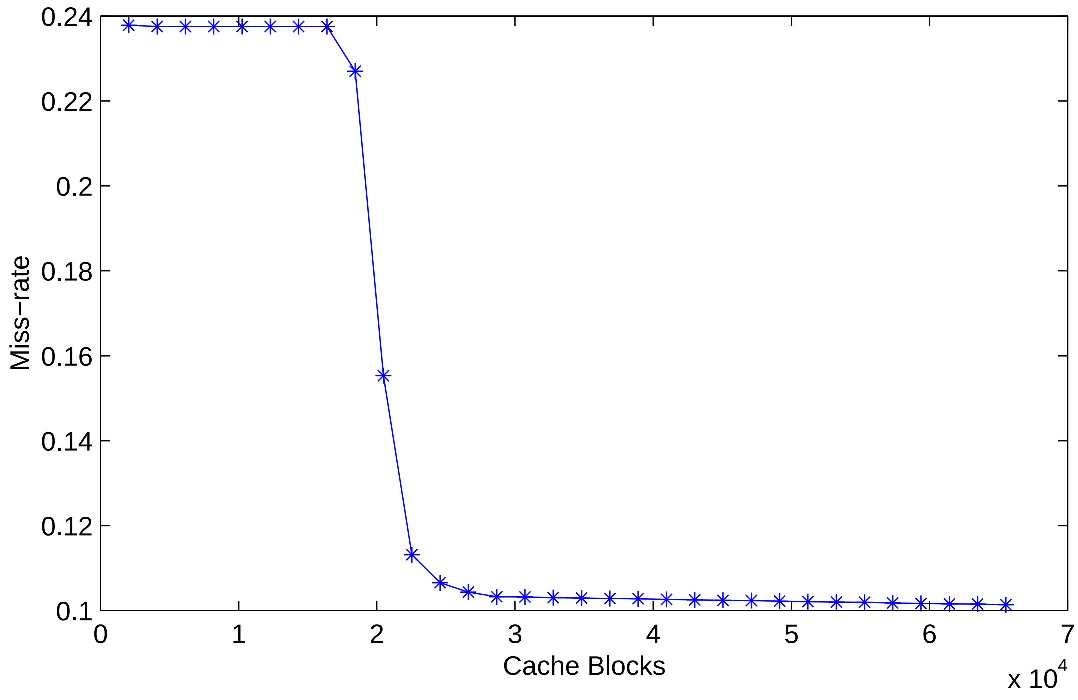


Figure 1: The miss-rate of `art` as a function of cache blocks.

partitioning. Especially, large L2 (level 2) caches only see memory references that are filtered by L1 (level 1) caches, and often show the miss-rate that is proportional to cache size. To be more accurate, $g_n(x)$ can be assumed to be a form of an power function$(a \cdot x^b)$. Empirical studies showed that the power function often well estimates the miss-rate [4].

Since characteristics of threads change dynamically, the estimation of $g_n(x)$ should reflect the changes. This is achieved by giving more weight to the counter value measured in more recent time periods. After every T memory references, we multiply each counter by δ, which is between 0 and 1. As a result, the effect of hits in previous time periods exponentially decays.

Once the marginal gains for each thread are estimated, the optimal partition should be decided based on the information. The previous work presented the algorithm for strictly convex miss-rate curves. Unfortunately, the number of misses for a real application is often not strictly convex as illustrated in Figure 1. The figure shows the miss-rate curve of `art` from the SPEC CPU2000 benchmark suite [11] for a 32-way 1-MB cache. In theory, every possible partition should be compared to obtain the optimal partition for non-convex miss-rate curves. However, non-convex curves can be approximated by a combination of a few convex curves. For example, the miss-rate of `art` can be approximated by two convex curves, one before the steep slope and one after that. Once a curve only has a few non-convex points, the convex resource allocation algorithm can be used to guarantee the optimal solution for non-convex cases.

1. Compute the marginal gain $g_n(x) = m_n(x-1) - m_n(x)$, and remember the non-convex points $\{p_{n,1}, p_{n,2}, ..., p_{n,P_n}\}$ for each thread, $g_n(p_{n,i}) < g_n(p_{n,i}+1)$.

2. Perform the convex algorithm starting c_n initialized with 0 or $p_{n,i}$.
3. Repeat step 2 for all possible initialization, and choose the partition that results in the maximum $\sum_{n=1}^{N} m_n(c_n)$.

3.3 Coarse Granularity Partitioning

Previous sections discussed cache partitioning assuming that it is possible to partition the cache in cache block granularity. Although this assumption simplifies the discussion, most practical partitioning mechanisms does not have control over each cache block. Since it is rather expensive to control each cache block, existing mechanisms often have a chunk of cache blocks that should be allocated together. We call this chunk as a partition block, and define D as the number of cache blocks in a partition block. Since a partition block consists of multiple cache blocks, partitioning mechanisms often allow the allocation of one partition block to multiple threads and let the replacement policy decide the cache block level partition.

First, let us consider the case without sharing a partition block among threads. Each partition block is allocated to only one thread. In this case, the algorithm for cache block granularity partitioning can be directly applied. In the algorithm, the marginal gain is defined as $g_n(x) = m_n((x-1)\cdot D) - m_n(x \cdot D)$. With a new marginal gain, the greedy algorithm can be used to assign one partition block at a time. This algorithm results in the optimal partition without sharing. However, sharing a partition block is essential to achieve high performance with coarse granularity partitioning. For example, imagine the case when there are many more threads than partition blocks. It is obvious that threads must share partition blocks in order to use the cache.

Knowing the number of misses for each thread as a function of cache space, the effect of sharing partition blocks can be evaluated once the allocation of the shared blocks by the LRU replacement policy is known. Consider the case when N_{share} threads share B_{share} partition blocks. Since each partition block consists of D cache blocks, the case can be thought of as N_{share} threads sharing $B_{share} \cdot D$ cache blocks. Since SMT systems tightly interleave memory references of the threads, the replacement policy can be thought of as random.

Define $B_{dedicate,n}$ as the number of partition blocks that are allocated to the n^{th} thread exclusively, and x_n as the number of cache blocks that belongs to the n^{th} thread. Since the replacement can be considered as random, the number of replacements for a certain cache region is proportional to the size of the region. The number of misses that replaces the cache block in the shared space $m_{share,n}(x)$ can be estimated as follows.

$$m_{share,n}(x) = \frac{B_{share}}{B_{dedicate,n} + B_{share}} \cdot m_n(x). \tag{3}$$

Under the random replacement, the number of cache blocks belongs to each process for the shared area is proportional to the number of cache blocks that each process brings into the shared area. Therefore, x_n can be written by

$$x_n = B_{dedicate,n} \cdot S + \frac{m_{share,n}(x_n)}{\sum_{i=1}^{N} m_{share,i}(x_i)} \cdot (B_{share} \cdot S). \tag{4}$$

Since there are x_n in both left and right sides of Equation 4, an iterative method can be used to esimate x_n starting with a initial value of x_n that is between $B_{dedicate,n} \cdot S$ and $(B_{dedicate,n} + B_{share}) \cdot S$.

3.4 Partitioning Mechanisms

For set-associative caches, various partitioning mechanisms can be used to actually allocate cache space to each thread. One way to partition the cache is to modify the LRU replacement policy. This approach has an advantage of controlling the partition at cache block granularity, but could be complicated. On the other hand, there are mechanisms that operate at coarse granularity. Page coloring [1] can restrict virtual address to physical address mapping, and as a result restrict cache sets that each thread uses. Column Caching [3] can partition the cache space by restricting cache columns (ways) that each thread can replace. However, it is relatively expensive to change the partition in these mechanisms, and the mechanisms support only several partition blocks. In this section, we describe the modified LRU mechanism and column caching to be used in our experiments.

3.4.1 Modified LRU Replacement

In addition to LRU information, the replacement decision depends on the number of cache blocks that belongs to each thread (b_i). On a miss, the LRU cache block of the thread (i) that caused the miss is chosen to be replaced if its allocation ($x_{i,k}$) is larger than its current use ($x_{i,k} \geq b_i$). Otherwise, the LRU cache block of another over-allocated thread is chosen. For set-associative caches, there may be no cache block of the desired thread in the set. In this case, the LRU cache block of a randomly chosen thread is replaced.

For set-associative caches, this modified replacement policy may result in replacing recently used data to keep useless data. Imagine the case when a thread starts to heavily access two or more addresses that happens to be mapped to the same set. If the thread already has many cache blocks in other sets, our partitioning will allocate only a few cache blocks in the accessed set for the thread, causing lots of conflict misses. To solve this problem, we can use (i) better mapping functions [18, 9] or (ii) a victim cache [12].

3.4.2 Column Caching

Column caching is a mechanism to allow partitioning of a cache at cache column granularity, where each column is one "way" or bank of the n-way set-associative cache. A standard cache considers all cache blocks in a set as candidates for replacement. As a result, a process' data can occupy any cache block. Column caching, on the other hand, restricts the replacement to a sub-set of cache blocks, which is essentially partitioning the cache.

Column caching specifies replacement candidacy using a bit vector in which a bit indicates if the corresponding column is a candidate for replacement. A LRU replacement unit is modified so that it replaces the LRU cache block from the candidates specified by a bit vector. Each partitionable unit has a bit vector. Since lookup is precisely the same as for a standard cache, column caching incurs no performance penalty during lookup.

Name	Thread	Benchmark Suite	Description
Mix-1	`art`	SPEC CPU2000	Image Recognition/Neural Network
	`mcf`	SPEC CPU2000	Combinatorial Optimization
Mix-2	`vpr`	SPEC CPU2000	FPGA Circuit Placement and Routing
	`bzip2`	SPEC CPU2000	Compression
	`iu`	DIS Benchmark Suite	Image Understanding
Mix-3	`art1`	SPEC CPU2000	Image Recognition/Neural Network
	`art2`		
	`mcf1`	SPEC CPU2000	Combinatorial Optimization
	`mcf2`		

Table 1: The benchmark sets simulated.

4 Experimental Results

Simulation is a good way to understand the quantitative effects of cache allocation. This section presents the results of a trace-driven simulation system. An 8-way 32-KB L1 cache is used to filter the memory references. The simulation system concentrates on the affects of an 8-way set-associative L2 caches with 32-Byte blocks by varying the size of the L2 cache over a range of 256 KB to 4 MB. We focus on L2 caches because the partitioning is more likely to be applicable to L2 caches due to large space and long latency. However, the algorithm itself works as well as or better on L1 caches than on L2 caches.

Three different sets of benchmarks are simulated, see Table 1. The first set (Mix-1) has two threads, `art` and `mcf` both from SPEC CPU2000. The second set (Mix-2) has three threads, `vpr`, `bzip2` and `iu`. Finally, the third set (Mix-3) has four threads, two copies of `art` and two copies of `mcf`, each with a different phase of the benchmark.

4.1 Hit-rate Comparison

The simulations compare the overall hit-rate of a standard LRU replacement policy and the overall hit-rate of a cache managed by our partitioning algorithm. The partition is updated every two hundred thousand memory references ($T = 200000$), and the weighting factor is set as $\delta = 0.5$. These values have been arbitarily selected; more carefully selected values of T and δ are likely to give better results. The hit-rates are averaged over fifty million memory references and shown for various cache sizes (see Table 2).

The simulation results show that the partitioning can improve the L2 cache hit-rate significantly: for cache sizes between 1 MB to 2 MB, partitioning improved the hit-rate up to 40% relative to the hit-rate from the standard LRU replacement policy. For small caches, such as 256-KB and 512-KB caches, partitioning does not seem to help. We conjecture that the size of the total workloads is too large compared to the cache size. At the other extreme, partitioning cannot improve the cache performance if the cache is large enough to hold all the workloads. Thus, the optimal cache size depends on both the number of simultaneous threads and the characteristics of the threads. Considering that SMT systems usually support eight simultaneous threads, cache partitioning can improve the performance

Cache Size (MB)	LRU L1 Hit-Rate(%)	LRU L2 Hit-Rate(%)	Partition L2 Hit-Rate(%)	Absolute Improvement(%)	Relative Improvement(%)
`art` + `mcf`					
0.25		15.61	15.37	-0.24	-1.54
0.5		17.29	16.48	-0.81	-4.68
1	71.99	26.27	36.90	10.63	40.46
2		50.01	51.13	1.12	2.24
4		76.73	75.04	-1.69	-2.20
`vpr` + `bzip2` + `iu`					
0.25		22.99	22.16	-0.83	-3.61
0.5		27.58	28.27	0.69	2.50
1	95.47	33.52	35.87	2.35	7.01
2		59.69	66.38	6.69	11.21
4		81.31	81.52	0.21	0.26
`art1` + `mcf1` + `art2` + `mcf2`					
0.25		12.04	12.68	0.64	5.31
0.5		14.22	14.33	0.11	0.77
1	71.50	16.95	19.07	2.12	12.51
2		26.67	34.94	8.27	31.01
4		50.59	51.35	0.76	1.50

Table 2: Hit-rate Comparison between the standard LRU and the partitioned LRU.

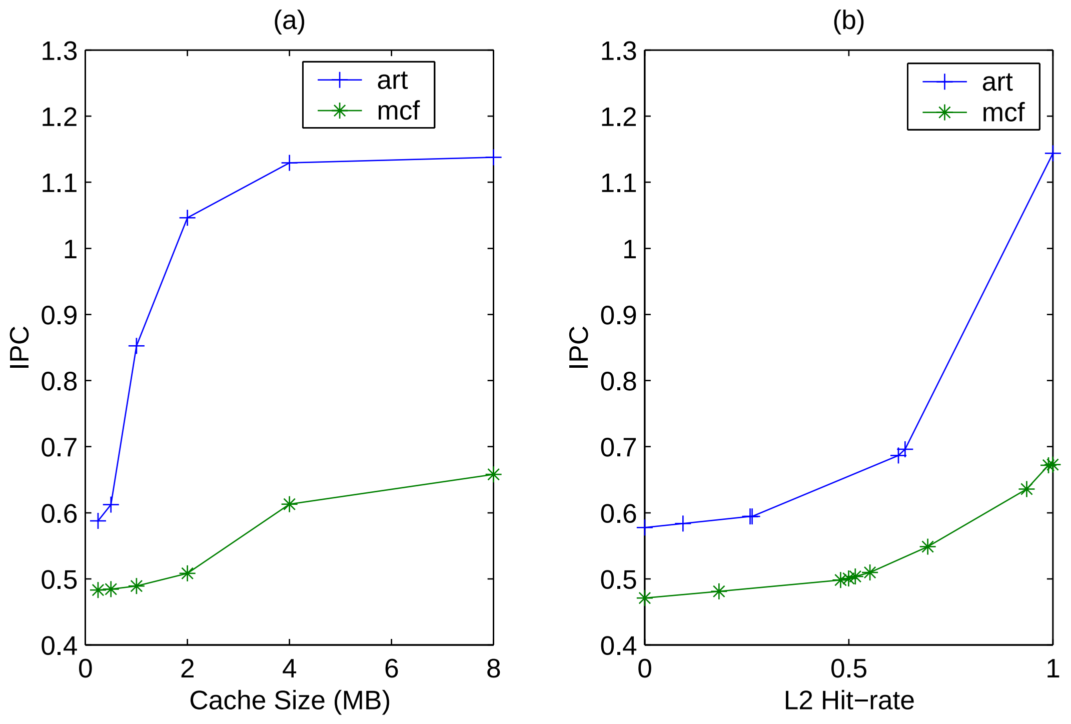


Figure 2: IPC of `art` and `mcf` under 32-KB 8way L1 caches and various size 8-way L2 caches. (a) IPC as a function of cache size. (b) IPC as a function of L2 hit-rate.

of caches in the range of up to tens of MB.

The results also demonstrate that the benchmark sets have large footprints. For all benchmark sets, the hit-rate improves 10% to 20% as the cache size doubles. This implies that these benchmarks need a large cache, and therefore executing benchmarks simultaneous can degrade the memory system performance significantly.

4.2 Effect of Partitioning on IPC

Although improving the hit-rate of the cache also improves the performance of the system, modern superscalar processors can hide memory latency by executing other instructions that are not dependent on missed memory references. Therefore, the effect of cache partitioning on the system performance, and in particular on IPC (Instructions Per Cycle), is evaluated based on entire system simulations.

The simulation results in this section are produced by SimpleScalar tool set [2]. SimpleScalar is a cycle-accurate processor simulator that supports out-of-order issue and execution. Our processor model for the simulations can fetch and commit 4 instructions at a time, and has 4 ALUs and 1 multiplier for integers and floating points respectively. To be consistent with the trace-driven simulations, 32-KB 8-way L1 caches with various size of 8-way L2 caches are simulated. L2 access latency is 6 cycles and main memory latency is 16 cycles.

Figure 2 (a) shows the IPC of two benchmarks (`art` and `mcf`) as a function of L2 cache

Cache Size (MB)	LRU Hit-rate(%) `art`	LRU Hit-rate(%) `mcf`	LRU IPC	Partition Hit-rate(%) `art`	Partition Hit-rate(%) `mcf`	Partition IPC	Abs. Improv. (%)	Rel. Improv. (%)
`art` + `mcf`								
0.25	8.8	20.4	1.064	8.0	20.5	1.065	0.001	0.09
0.5	10.3	22.2	1.067	14.5	17.8	1.070	0.003	0.28
1	25.7	26.6	1.080	61.6	19.5	1.167	0.087	8.06
2	63.7	40.3	1.189	76.8	33.1	1.347	0.158	13.29
`art1` + `mcf1` + `art2` + `mcf2`								
0.25	6.4/6.7	16.4/15.2	2.123	6.5/3.5	29.8/11.3	2.126	0.003	0.14
0.5	7.3/7.6	19.5/18.2	2.128	7.7/4.6	30.7/15.2	2.131	0.003	0.14
1	9.3/10.1	22.1/21.4	2.134	9.1/32.4	31.1/13.5	2.161	0.027	1.27
2	25.1/25.5	28.1/25.1	2.160	57.2/73.2	32.0/16.0	2.456	0.307	14.21
4	63.9/63.6	41.7/41.2	2.382	73.9/86.7	49.5/26.6	2.786	0.404	16.96

Table 3: IPC Comparison between the standard LRU and the partitioned LRU for the case of executing `art` and `mcf` simultaneously (Some results are missing due to the extensively long simulation time, and results for all benchmark sets will be included in the final version).

size. Each benchmark is simulated separately and is allocated all system resources including all the L2 cache. L1 caches are assumed to be 32-KB 8-way for all cases. For various L2 cache sizes, IPC is estimated as a function of the L2 hit-rate (Figre 2 (b)).

The figures illustrate two things. First, the IPC of `art` is very sensitive to the cache size. The IPC almost doubles if the L2 cache is increased from 1 MB to 4 MB. Second, the IPCs of these two benchmarks are relatively low considering there are 10 functional units (5 for integer, 5 for floating point instructions). Since the utilization of the functional units are so low, executing these two benchmarks simultaneous will not cause many conflict in functional resources.

When executing the threads simultaneously the IPC values are approximated from Figure 2 (b) and the hit-rates are estimated from the trace-driven simulations (of the previous subsection). For example, the hit-rates of `art` and `mcf` are 25.79% and 26.63% respectively if two threads execute simultaneously with a 32-KB 8-way L1 cache and a 1-MB 8-way L2 cache from the trace-driven simulation. From Figre 2 (b) the IPC of each thread for the given hit-rates can be estimated as 0.594 and 0.486. If we assume that there is no resource conflicts, the IPC with SMT can be approximated as the sum, 1.08. This approximation tells you the maximum IPC that can be achieved by SMT.

Table 3 summarizes the approximated IPC for SMT with a L2 cache managed by the standard LRU replacement policy and the one with a L2 cache managed by our partitioning algorithm. The absolute improvement in the table is the IPC of the partitioned case subtracted by the IPC of the standard LRU case. The relative improvement is the improvement relative to the IPC of the standard LRU, and calculated by dividing the absolute improvement by the IPC of the standard LRU. The table shows that the partitioning algorithm improves IPC for all cache sizes up to 17%.

The table also clearly shows that SMT should manage caches carefully for threads that are sensitive to the performance of the memory system. In the case of a 2-MB L2 cache, SMT can only achieve IPC of 2.160 whereas executing `art` alone can achieve IPC of 1.04. Although SMT can still achieve a higher IPC than the case when only one thread is executing, the IPC of `art` is only 0.594. Moreover, the performance degradation by cache interference will become even more severe as the latency to the main memory increases. This problem can be solved by smart partitioning of cache memory for some cases. If the cache is too small, the thread scheduling should be changed.

5 Conclusion

Low IPC can be attributed to two factors, data dependency and memory latency. SMT exploits the first factor but not the second. We have discovered that SMT only exacerbates the problem when the executing threads require large caches. That is, when multiple executing threads interfere in the cache, even SMT cannot utilize the idle functional units.

We have studied one method to reduce cache interference among simultaneously executing threads. Our on-line cache partitioning algorithm estimates the miss-rate characteristics of each thread at run-time, and dynamically partition the cache to the threads that are executing simultaneously. The algorithm estimates the marginal gains as a function of cache size and finds the partition that minimizes the total number of misses. To apply theory to practice, some problems have been solved. First, the search algorithm has been modified to deal with non-convexity. Second, partitioning period T and aging factor δ have been decided. Finally, the effect of sharing cache space has been studied for partitioning mechanisms with coarse granularity.

The partitioning algorithm has been implemented in a trace-driven cache simulator. The simulation results show that partitioning can improve the cache performance noticeably over the standard LRU replacement policy for a certain range of cache size for given threads. Using a full-system simulator, the effect of partitioning on the instructions per cycle (IPC) has also been studied. The preliminary results show that we can also expect IPC improvement using the partitioning algorithm.

The simulation results have shown that our partitioning algorithm can solve the problem of thread interference in caches for a range of cache sizes. However, partitioning alone cannot improve the performance if caches are too small for the workloads. Therefore, threads that execute simultaneously should be selected carefully considering their memory reference behavior. The cache-aware job scheduling remains to be done.

Even without SMT, one can view an application as multiple threads executing simultaneously where each thread has memory references to a particular data structure. Therefore, the result of this investigation can also be exploited by compilers for a processor with multiple functional units and some cache partitioning control.

References

[1] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS VI*, 1994.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[3] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching.* PhD thesis, Massachusetts Institute of Technology, 1999.

[4] C. K. Chow. Determining the optimum capacity of a cache memory. IBM Tech. Disclosure Bull., 1975.

[5] Compaq. Compaq alphastation family.

[6] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), 1997.

[7] B. Fox. Discrete optimization via marginal analysis. *Management Science*, 13, 1966.

[8] C. Freeburn. The hewlett packard PA-RISC 8500 processor. Technical report, Hewlett Packard Laboratories, Oct. 1998.

[9] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *the 1997 international conference on Supercomputing*, 1997.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann, 1996.

[11] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennim. *IEEE Computer*, July 2000.

[12] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *the 17th Annual International Symposium on Computer Architecture*, 1990.

[13] J. L. Lo, J. S. Emer, D. M. T. Henry M. Levy, Rebecca L. Stamm, and S. J. Eggers. Converting thread-level parallelism to insruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.

[14] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual*, 1996.

[15] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), Sept. 1992.

[16] G. E. Suh. Analytical cache models with applications to cache partitioning in time-shared systems. Master's thesis, Massachusetts Institute of Technology, 2001.

[17] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.

[18] N. Topham and A. González. Randomized cache placement for eleminating conflicts. *IEEE Transactions on Computers*, 48(2), Feb. 1999.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.