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. We present a general partitioning scheme that can be applied to set-associative caches at any partition granularity. Furthermore, in our scheme threads can have overlapping partitions, which provides more degrees of freedom when partitioning caches with low associativity.

Since memory reference characteristics of threads can change very quickly, our method collects the miss-rate characteristics of simultaneously executing threads at runtime, and partitions the cache among the executing threads. Partition sizes are varied dynamically to improve hit rates. Trace-driven simulation results show a relative improvement in the L2 hit-rate of up to 40.5% over those generated by the standard least recently used replacement policy, and IPC improvements of up to 17%. Our results show that smart cache management and scheduling is important for SMT systems to achieve high performance.

KEY WORDS
Memory System, Simultaneous Multithreading, Cache Partitioning

1. Introduction

Microprocessors with multiple functional units have low IPC (Instructions per Cycle) rates either because of a lack of parallelism, or because of a high incidence of data dependencies. Simultaneous Multi-Threading, [1, 2, 3] (SMT), helps in the former case but, in the latter case, it only exacerbates the stress on the memory subsystem, especially since the standard LRU replacement scheme treats all references in the same way. Thus, a single thread can easily "pollute" the cache with its data, causing higher miss rates for other threads, and resulting in low overall performance.

This paper presents a dynamic cache partitioning algorithm that minimizes the overall cache miss rate for simultaneous multithreading systems. Rather than relying on the standard least recently used (LRU) cache replacement policy, our algorithm dynamically allocates parts of the cache to the most needy threads using on-line estimates of individual thread miss rates. The cache is assumed to be large enough to support multiple contexts, but not large enough to hold all of the working sets of the simultaneously executing threads. Although a 1997 study has shown that a 256-KB L2 cache, which is reasonable size for modern microprocessors [4, 5, 6], is large enough for a particular set of workloads [2], we believe that workloads have become much larger and diverse; multimedia programs such as video or audio processing software often consume hundreds of MB and many SPEC CPU2000 benchmarks now have memory footprints larger than 100 MB [7].

We propose a novel cache partitioning scheme wherein a cache miss will only allocate a new cache block to a thread if its current allocation is below its limit. To implement this scheme, we require counters in the cache that provide on-line estimates of individual thread miss rates. Based on these counters, we can augment LRU replacement to better allocate cache resources to threads, or we can use Column caching [8], which allows threads to be assigned to overlapping partitions, to partition the cache. Simulation shows that the partitioning algorithm can significantly improve both the miss-rate and the instructions per cycle (IPC) of the overall workload.

In conventional time-shared systems, cache partitioning depends not only on the active thread, but also on the memory reference pattern of inactive threads which have run in the past, and will run again in the near future. On the other hand, in SMT systems, multiple threads are active at the same time, collectively stressing the memory system. Since these threads very quickly use up cache resources once they start running, partitioning depends only on the memory reference characteristics of the set of active threads. This differs from traditional time sharing systems where one must also consider the length of the time quantum and the characteristics of the ready, but not executing threads. Since the memory references from each thread are interleaved very tightly, one can consider an SMT system to be a traditional time-sharing system with a context switch at each memory reference.[1]

This paper is organized as follows. In Section 2, we

[1] In many systems, each page fault or disk access causes a context switch and so disk cache partitioning schemes are somewhat relevant to SMT cache partitioning.

describe related work. In Section 3, we first study the optimal cache partitioning problem for the ideal case of fully associative caches that are partitionable on a cache-block basis. We then extend our method to the more realistic set-associative cache case. Section 4 evaluates the partitioning method by simulations. Finally, Section 5 concludes the paper.

2. Related Work

Stone, Turek and Wolf [9] 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, and 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.

In previous work [10] we 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 performed by collecting the miss-rate information of each process off-line. The work of [10] did not investigate how to partition the cache memory at run-time.

Thiébaut, Stone and Wolf applied their theoretical partitioning study [9] to improve disk cache hit-ratios [11]. 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. It works for set-associative caches with multiple threads and a coarse-grained partition, whereas Thiébaut et al. [11] only focused on disk caches that are fully-associative with cache block granularity. Finally, this work discusses an on-line method to partition the cache, whereas our previous only covered partitioning based on off-line profiling [10].

3. Partitioning Algorithm

This section presents our cache partitioning algorithm. We lead up to a general partitioning method in several steps. First, given a fully-associative cache that can be partitioned on a cache-block basis and knowing the miss-rate for each task as a function of partition size, we show how an optimal partition is obtained by iteratively increasing the partition size for the thread that will benefit the most. Next, we show that it is possible to compute the miss rate functions on-line using many hardware counters for a fully associative cache, and that it is possible to approximate the miss-rate function using fewer counters in the case of a set-associative cache. These results are then combined and applied to the more practical case of coarse grained partitioning. Finally, the algorithm to actually allocate cache blocks to each thread is developed.

3.1 Optimal Cache Partitioning

Given N executing threads sharing a cache of C blocks with partitioning on a cache block granularity, the problem is to partition the cache into N disjoint subsets of cache blocks so as to minimize the overall miss-rate. For each thread, the miss-rate as a function of partition size (the number of cache blocks), is known. Let c_i represent the number of cache blocks allocated to the i^{th} thread. A cache partition is specified by the number of cache blocks allocated to each thread, i.e., $\{c_1, c_2, ..., c_N\}$. Since it is unreasonable to repartition the cache every memory reference, the partition remains fixed over a time period, π, that is long enough to amortize the repartitioning cost.

The number of cache misses for the i^{th} thread over π is given by a function of partition size ($m_i(x)$). The optimal partition for the period is the set of integer values $\{c_1, c_2, ..., c_N\}$, that minimizes the following expression:

$$\text{total misses over time period } \pi = \sum_{i=1}^{N} m_i(c_i) \qquad (1)$$

under the constraint that $\sum_{i=1}^{N} c_i = C$. C is the total number of blocks in the cache.

For the case where the number of misses for each thread is a strict convex function of cache space, Stone, Turek and Wolf [9] noted that finding the optimal partition, $\{c_1, c_2, ..., c_N\}$, falls into the category of separable convex resource allocation problems. The following, well-known, simple greedy algorithm yields an optimal partition [9, 12]:

1. Let the marginal gain, $g_j(x)$, be the number of additional hits for the j^{th} thread, when the allocated cache blocks increases from x to $x + 1$.

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

3. *Increase by one the number of cache blocks assigned to the thread that has the maximum marginal gain given the current allocation.*
   Increase c_j by one, where j is the index for which $g_j(c_j)$ is largest.

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

3.2 Computing the Marginal Gain

The computation of the marginal gain, $g_i(x)$, depends on the the miss rate for task i as a function of the cache partition size, $m_i(x)$, over a time period, π. For a fully associative LRU cache, it is possible to compute $m_i(x)$ on-line using C counters. When a task references a data item in the cache that is the k^{th} most recently referenced item, then counter k for task i is increased. At the end of the time period, these counters form the miss rate function for each task, as described below. The description below applies to the general case of a set-associative cache.

To perform dynamic cache partitioning, the marginal gains of having one more cache block can be estimated on-line. As discussed in the previous section, $g_i(x)$ is the number of additional hits that the i^{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, $g_i(0)$ represents the number of hits on the most recently used cache block of the i^{th} thread, $g_i(1)$ represents the number of hits on the second most recently used cache block of the i^{th} thread, and so on.

For each thread, a set of counters, one for each associativity (way) of the cache, is maintained. 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. The k^{th} counter value represents the number of additional hits for the thread by having the 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, $g_i(x)$ satisfies the following equation.

$$\sum_{x=(k-1)\cdot S}^{k\cdot S-1} g_i(x) = count_i(k) \qquad (2)$$

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

To estimate marginal gains from Equation 2, assume that $g_i(x)$ 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 partitioning. This is especially true in the case of large L2 (level 2) caches, which 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, $g_i(x)$ can be assumed to be a form of an power function, e.g., $a \cdot x^b$. Empirical studies showed that the power function often accurately estimates the miss-rate [13].

Since characteristics of threads change dynamically, the estimation of $g_i(x)$ 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.

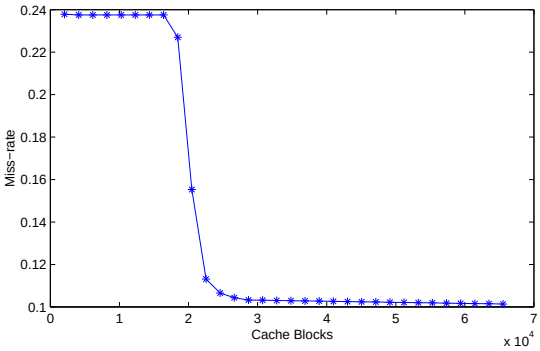

Figure 1. The miss-rate of `art` as a function of cache blocks.

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 [7] for a 32-way 1-MB cache. As long as the miss-rate curve is convex, the marginal gain function decreases, and at the non-convex points, the marginal gain function will increase. 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. For each thread, i, compute the ρ_i non-convex points of its miss-rate curve: $\{p_{i,1}, p_{i,2}, ..., p_{i,\rho_i}\}$, $g_i(p_{i,j}) < g_i(p_{i,j}+1)$.

2. Execute the convex algorithm with c_i initialized to 0 or $p_{i,j}$, $\forall j$.

3. Repeat step 2 for all possible initializations, and choose the partition that results in the maximum $\sum_{i=1}^{N} m_i(c_i)$.

3.3 Coarse Granularity Partitioning

Since it is rather expensive to control each cache block, practical partitioning mechanisms perform allocation of chunks of cache blocks, referred to as a *partition block*. We will use D to refer to the number of cache blocks in a partition block. We allow the allocation of one partition block to multiple threads and let the replacement policy decide the cache block level partition.

First, consider the no sharing case where each partition block is allocated to only one thread. The algorithm for cache block granularity partitioning can be directly applied. Define the partition marginal gain as $g_i(x) =$

$m_i(x \cdot D) - m_i((x+1) \cdot D)$ and use the greedy algorithm to assign one partition block at a time, resulting in an optimal partition without sharing. However, sharing a partition block is essential to achieve high performance with coarse granularity partitioning. For example, 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 $B_{dedicate,i}$ as the number of partition blocks that are allocated to the i^{th} thread exclusively, and x_i as the number of cache blocks that belongs to the i^{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 $m_{share,i}(x)$ can be estimated as follows.

$$m_{share,i}(x) = \frac{B_{share}}{B_{dedicate,i} + B_{share}} \cdot m_i(x). \quad (3)$$

Under the random replacement, the number of cache blocks belonging 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_i can be written by

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

Since x_i is on both the left and right sides of Equation 4, an iterative method can be used to estimate x_i starting with a initial value that is between $B_{dedicate,i} \cdot S$ and $(B_{dedicate,i} + B_{share}) \cdot S$.

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 which has the advantage of controlling the partition at cache block granularity, but LRU implementations can be expensive for high-associativity caches.

On the other hand, there are mechanisms that operate at coarse granularity. Page coloring [14] can restrict virtual address to physical address mapping, and as a result restrict cache sets that each thread uses. Column Caching [8] 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 a limited number of 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 actually allocation (b_i) is larger than the desired one ($x_i \leq b_i$). 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, so the LRU cache block of a randomly chosen thread is replaced.

3.4.2 Column Caching

Column caching is a mechanism that allows partitioning of a cache at column or "way" granularity. 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 essentially partitions 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.

4. Experimental Results

This section presents the results of a trace-driven simulation system in order to understand the quantitative effects of our cache allocation scheme. The simulations concentrate on an 8-way set-associative L2 cache with 32-Byte blocks and vary the size of the cache over a range of 256 KB to 4 MB. Due to large space and long latency, our scheme is more likely to be useful for an L2 cache, and so that is the focus of our simulations. We note in passing, that we believe our approach will work on L1 caches as well.

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.

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

Table 1. The benchmark sets simulated. All but the Image Understanding benchmark are from SPEC CPU-2000.

Size (MB)	L1 %Hits	L2 %Hits	Part. L2 %Hits	Abs. %Imprv.	Rel. %Imprv.
art + mcf					
0.2		15.6	15.3	-0.2	-1.5
0.5		17.2	16.4	-0.8	-4.6
1	71.9	26.2	36.9	10.6	40.4
2		50.0	51.1	1.1	2.2
4		76.7	75.0	-1.6	-2.2
vpr + bzip2 + iu					
0.2		22.9	22.1	-0.8	-3.6
0.5		27.5	28.2	0.6	2.5
1	95.4	33.5	35.8	2.3	7.0
2		59.6	66.3	6.6	11.2
4		81.3	81.5	0.2	0.2
art1 + mcf1 + art2 + mcf2					
0.2		12.0	12.6	0.6	5.3
0.5		14.2	14.3	0.1	0.7
1	71.5	16.9	19.0	2.1	12.5
2		26.6	34.9	8.2	31.0
4		50.5	51.3	0.7	1.5

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

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. The range of cache sizes for which partitioning can improve performance 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 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 by 10% to 20% as the cache size doubles. This implies that these benchmarks need a large cache, and therefore executing benchmarks simultaneously 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 [15]. 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 sizes 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 size. Each benchmark is simulated separately and is allocated all system resources including all of 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 (Figure 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 size 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 utilizations of the functional units are so low, executing these two benchmarks simultaneously will not cause many conflicts 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

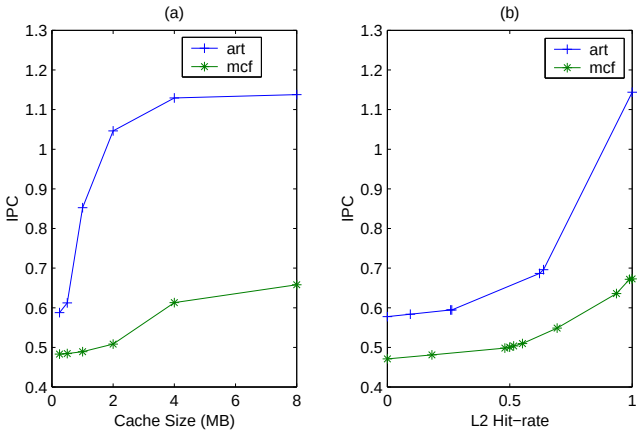

Figure 2. IPC of `art` and `mcf` under 32-KB 8way L1 caches and various size 8-way L2 caches. (a) IPC as a function of cache size. (b) IPC as a function of L2 hit-rate.

`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 trace-driven simulation. From Figure 2 (b) the IPC of each thread for the given hit-rates can be estimated as 0.594 and 0.486. Assuming no resource conflicts, the IPC with SMT can be approximated as the sum, 1.08. This approximation bounds 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 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 is 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 experiment results also show that SMT should manage caches carefully. In the case of four threads with a 2-MB cache, SMT can achieve the overall IPC of 2.160 from Table 3. However, if you only consider one thread (`art1`), its IPC is only 0.594 whereas it can achieve an IPC of 1.04 alone (Figure 2). The performance of a single thread is significantly degraded by sharing caches. 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, we believe that the thread scheduling should be changed.

5. Conclusion

Low IPC can be attributed to two factors, data dependency and memory latency. SMT mitigates 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 all the functional units because not all required data is present in the memory.

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 partitions the cache among the threads that are executing simultaneously. The algorithm estimates the marginal gains as a function of cache size and uses a search algorithm to find the partition that minimizes the total number of misses.

The hardware overhead for the modifications proposed in this paper are minimal. A small number of additional counters is required. The counters are updated on cache hits, however, they are not on the critical path and so a small buffer can absorb any burstiness. To actually partition the cache, we can modify the LRU replacement hardware in a simple way to take the values of the counters into account. Or, we can use column caching which requires a small number of additional bits in the TLB entries, and a small amount of off-critical-path circuitry that is invoked only during a cache miss.

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. While we have not used a full SMT simulator to generate IPC numbers, the large improvements obtained in hit rates lead us to believe that significant IPC improvements will be obtained using a full SMT simulator, or on real hardware.

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. Cache-aware job scheduling is a subject of our ongoing work.

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.

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 strategy for the case of executing art and mcf simultaneously.

Acknowledgements

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, titled "Malleable Caches for Data-Intensive Computing". Thanks also to Enoch Peserico, Derek Chiou, David Chen and Vinson Lee for their comments.

References

[1] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[2] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.

[3] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), 1997.

[4] Catherine Freeburn. The hewlett packard PA-RISC 8500 processor. Technical report, Hewlett Packard Laboratories, October 1998.

[5] Compaq. Compaq alphastation family.

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

[7] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

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

[9] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), September 1992.

[10] G. Edward Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Application to Cache Partitioning. In *the* 15^{th} *international conference on Supercomputing*, 2001.

[11] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.

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

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

[14] Brian K. Bershad, Bradley J. Chen, Denis Lee, and Theodore H. Romer. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS VI*, 1994.

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