The Journal of Supercomputing, 28, 7–26, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Dynamic Partitioning of Shared Cache Memory

G. E. SUH suh@mit.edu
Massachusetts Institute of Technology

L. RUDOLPH rudolph@mit.edu
Massachusetts Institute of Technology

S. DEVADAS devadas@mit.edu
Massachusetts Institute of Technology

Abstract. This paper proposes dynamic cache partitioning amongst simultaneously executing processes/threads. We present a general partitioning scheme that can be applied to set-associative caches.

Since memory reference characteristics of processes/threads can change over time, our method collects the cache miss characteristics of processes/threads at run-time. Also, the workload is determined at run-time by the operating system scheduler. Our scheme combines the information, and partitions the cache amongst the executing processes/threads. Partition sizes are varied dynamically to reduce the total number of misses.

The partitioning scheme has been evaluated using a processor simulator modeling a two-processor CMP system. The results show that the scheme can improve the total IPC significantly over the standard least recently used (LRU) replacement policy. In a certain case, partitioning doubles the total IPC over standard LRU. Our results show that smart cache management and scheduling is essential to achieve high performance with shared cache memory.

Keywords: cache partitioning, shared caches, CMP and SMT

1. Introduction

In the near future, microprocessors will be able to execute multiple processes/threads simultaneously and exploit process/thread-level parallelism.[1] Multiple processors may be on a single chip (CMP) [5], or simultaneous multithreading (SMT) may be used [6, 11, 20]. In these systems, processes or threads share parts of the memory system often including L1 and L2 caches. Therefore, executing multiple contexts simultaneously exacerbates the stress on the memory subsystem, especially since the standard least recently used (LRU) replacement scheme treats all references in the same way. In the LRU scheme, a single process can easily "pollute" the cache with its data, causing higher miss-rates for other threads, and resulting in low overall performance.

Managing cache space amongst multiple processes is particularly important when the cache is large enough to support multiple contexts, but not large enough to hold all of the working sets of the simultaneously executing processes. In fact, an early study of the SMT architecture demonstrated significant improvement in IPC for a set of workloads that fit into a 256-KB L2 cache, where cache contention is not a problem [11]. However, we believe that workloads have become much larger and more diverse; multimedia programs such as video or audio processing software often consume hundreds of MB and many SPEC CPU2000 benchmarks benefit by using several MB caches [9].

This paper presents a dynamic cache partitioning scheme that explicitly allocates cache space amongst simultaneously executing processes and minimize the overall cache misses. Using a set of on-line counters, our scheme dynamically estimates each process' gain or loss in different cache allocations in terms of the number of cache misses. Then, the allocation is dynamically changed so that more needy processes can use more cache space. For example, consider the case when a streaming process runs simultaneously with a process with high temporal locality. Our partitioning scheme detects that more cache space does not improve a streaming process, and allocates most of cache space to the other process.

Our scheme only considers partitioning amongst simultaneous processes. In conventional time-shared systems, cache partitioning depends not only on the active process, but also on the memory reference pattern of inactive processes which have run in the past, and will run again in the near future. On the other hand, in CMP/SMT systems, multiple processes are active at the same time, collectively stressing the memory system. Since these processes very quickly use up cache resources once they start running, partitioning depends only on the memory reference characteristics of the set of active processes.

The cache can be partitioned by either augmenting the standard LRU replacement policy or using column caching [4]. In the augmented LRU policy, the replacement unit keeps track of the number of cache blocks belong to each active process, and allocates a new cache block to a process only if its current allocation is below its limit. Column caching partitions the cache at cache column or "way" granularity (A *D*-way associative cache has *D* columns). The simulation experiments presented here are based on the augmented LRU policy.

Simulation results demonstrate that the cache partitioning can significantly improve the instructions per cycle (IPC) metric of the overall workload. Partitioning the cache amongst simultaneous processes is especially effective when the cache is not large enough to hold the entire working set, but not too small so that it can hold some critical portion of the working set.

This paper is organized as follows. In Section 2, we 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, and discuss implementation details in Section 4. Section 5 evaluates the partitioning method by simulations. Finally, Section 6 concludes the paper.

2. Related work

Stone et al. [13] 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.

In previous work [15] we proposed an analytical cache model for multitasking, and also studied the cache partitioning problem for time-shared systems based on the model. The technique is applicable to any length of time quanta 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, and we did not describe techniques to partition the cache memory at run-time.

Thiébaut et al. [18] applied their theoretical partitioning study [13] to improve disk cache hit-ratios. 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. However, they only focused on disk caches that are fully-associative with cache block granularity whereas the scheme in this paper works for set-associative caches.

Recently, people have studied re-sizing caches to save energy [1, 12, 22]. Although they change the size of the cache, their purpose is to reduce the energy consumption. These works focus on one process and re-size the cache by turning off a part of the cache. Unsal proposed to partition the cache to optimize the energy consumption for multimedia processing [21]. His work isolates a stream in a media application, which pollutes the cache. Ranganathan introduced a way of partitioning the cache exploiting the associativity, and showed an application to media processing. Although these two works studied partitioning of cache memory, they only rely on static compiler analysis for media processing. Our work is significantly different from these in a sense that we focus on optimizing performance considering multiple processes and that we partition the cache dynamically using hardware counters.

3. Partitioning algorithm

This section presents an analytical analysis of cache partitioning. First, we define the optimal cache partition that minimizes the total number of misses for simultaneous processes. To develop a partitioning algorithm, marginal gains are introduced as a way of determining the usefulness of cache space for a process. Finally, we discuss how to find the optimal partition when the marginal gains for each process are given. A search algorithm for the case when the marginal gains monotonically decrease as cache space increases is summarized from previous work [13], and extended to handle non-convexity.

3.1. Optimal cache partitioning

Given N executing processes 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 misses. Since it is unreasonable to re-partition the cache every memory reference, the partition remains fixed over a time period, T, that is long enough to amortize the re-partitioning cost. Let c_i represent the number of cache blocks allocated to the i-th process over the time period. A cache partition is specified by the number of cache blocks allocated to each process, i.e., $\{c_1, c_2, \ldots, c_N\}$.

Define $m_i(c)$ as the number of cache misses for the i-th process over a time period T as a function of partition size (c). Then the optimal partition for the period is the set of integer values $\{c_1, c_2, \ldots, c_N\}$, that minimizes the following expression:

$$\text{Total misses over time period } T = \sum_{i=1}^{N} m_i(c_i) \tag{1}$$

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

3.2. Partitioning using marginal gains

To find the optimal partition (or at least a good partition), we make use of marginal gains of each competing process. The marginal gain of a process, namely $g_i(c)$, is defined as the derivative of the miss curve $(m_i(c))$ at a given cache space c;

$$g_i(c) = m_i(c) - m(c+1). \tag{2}$$

The marginal gain at a given allocation represents the number of cache misses that is reduced by having the last cache block. Thus, it indicates the benefit of increasing the cache allocation from c to $c+1$ blocks for a process.

For the case where the marginal gain for each process is a monotonically decreasing function of cache space, Stone et al. [13] noted that finding the optimal partition, $\{c_1, c_2, \ldots, c_N\}$, falls into the category of separable convex resource allocation problems. The following, well-known, simple greedy algorithm is guaranteed to result in an optimal partition [7, 13]:

1. Initialize $c_1 = c_2 = \cdots = c_N = 0$.
2. Increase by one the number of cache blocks assigned to the process that has the maximum marginal gain given the current allocation. Increase c_k by one, where k is the index for which $g_k(c_k)$ is largest.
3. Repeat step 3 until all cache blocks are assigned (i.e., C times).

3.3. Handling non-convexity

The number of misses for a real application is often not strictly convex as illustrated in Figure 1. Thus, the marginal gain may not be a monotonically decreasing function. The figure shows the miss-rate curve of art from the SPEC CPU2000 benchmark suite [9] 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 process, i, compute the ρ_i non-convex points of its miss-rate curve: $\{p_{i,1}, p_{i,2}, \ldots, p_{i,\rho_i}\}, g_i(p_{i,j}) < g_i(p_{i,j}+1)$.
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 $\sum_{i=1}^{N} m_i(c_i)$.

This algorithm is very effective in finding the optimal point when the marginal gains are not monotonically decreasing functions. However, it may be too expensive

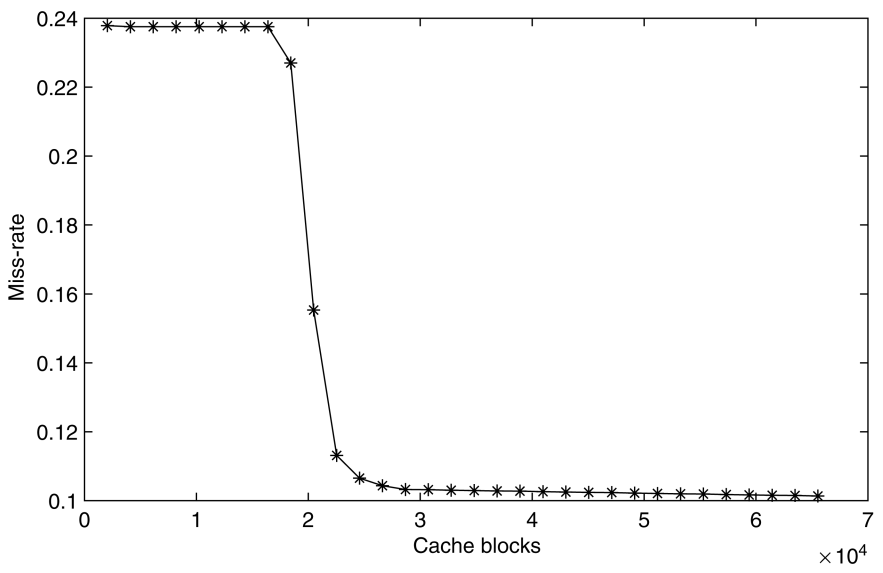


Figure 1. The miss-rate of `art` as a function of cache blocks.

to actually implement this algorithm in cache partitioning. In practice, we use a simplified version discussed in the next section.

4. Implementation

The previous section discussed partitioning the cache based on marginal gains of competing processes. Now, we consider how to implement cache partitioning in set-associative caches. Our partitioning scheme consists of three parts: marginal gain counters, a partitioning mechanism, and an OS controller. First, we use a set of counters to estimate the marginal gains of executing processes. Second, we need a mechanism in the cache that can actually control the allocation to each process. Finally, the operating system determines the best partition based on the information from counters and sets the cache allocation.

4.1. Marginal-gain counters

To perform dynamic cache partitioning, the marginal gains of having one more cache block should be estimated on-line. For a running process, we want to obtain marginal gains for various cache sizes without actually changing the cache configuration. In cache simulations, it has been shown that different cache sizes can be simulated in a single pass [14]. We emulate this technique in hardware to obtain multiple marginal gains while executing a process with a fixed cache configuration.

We use a set of counters to collect the marginal gains of each process for the past time periods, and assume that the past marginal gain is a good prediction for the future. Our mechanism assumes that the cache uses the standard LRU replacement policy.

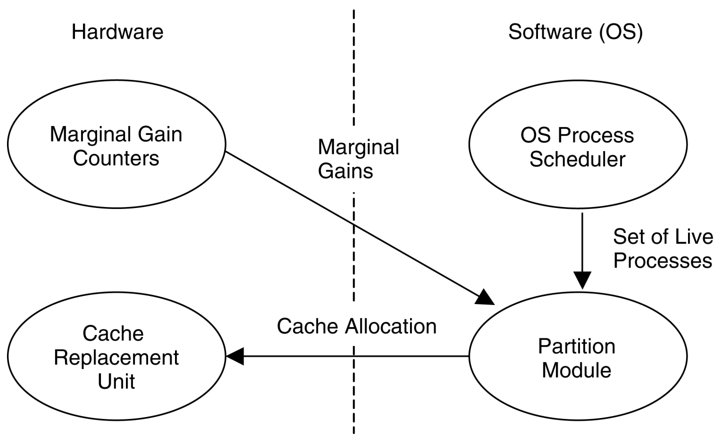


Figure 2. The implementation of overall partitioning scheme.

For a fully-associative cache with C blocks, it is possible to compute $g(c)$ over a time period T on-line using C counters. Computing the marginal gain simply follows from the following set of counters:

Counters for a fully associative cache: There is one counter for each block in the cache; counter(0) records the number of hits in the most recently used block, and counter(1) is the number of hits in the second most recently used block, etc. When there is a reference to the i-th most recently used block, then counter$(i-1)$ is incremented. Note that the item referenced then becomes the most recently used block, so that a subsequent reference to that item is likely to increment a different counter.

The marginal gain $g(c)$ is obtained directly by counting the number of hits in the $(c+1)$-th most recently used block (counter(c)).

In set-associative caches, LRU ordering is kept only within each set. A set of counters, one for each associativity (way) of the cache rather than each cache block, is maintained per process. On every cache hit, the corresponding counter is increased. Although we can only estimate marginal gains of having each way, not each cache block, this is often good enough for partitioning if the cache has reasonably high associativity.

Way-counters for a set-associative cache: There is one counter for each way of the cache. A hit in the cache to the MRU block of some set updates counter(0). A hit in the cache to the LRU block of some set updates counter$(D-1)$, assuming D-way associativity.

Figure 3 illustrates the implementation of these hardware counters for two-way associative caches.

Each way in a set-associative cache includes S cache blocks, where S is the number of sets. Assuming that cache accesses are well distributed over sets, we can approximate the marginal gain of having additional S blocks for a fully-associative cache from the way-counters as follows:

$$\text{counter}(k) = \sum_{c=k \cdot S}^{(k+1) \cdot S-1} g(c), \tag{3}$$

where S is the number of sets.

With a minimum monitoring granularity of a way, high-associativity is essential for obtaining enough information for performance optimization; our experiments show that eight-way associative caches can provide enough information for partitioning. Content-addressable-memory (CAM) tags are attractive for low-power processors [23] and they have higher associativity; the SA-1100 StrongARM processor [10] contains a 32-way associative cache. When a cache is low-associative,

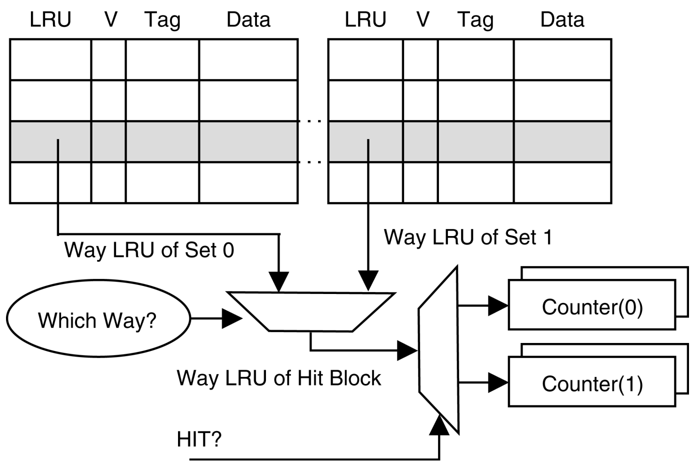


Figure 3. The implementation of memory monitors for two-way associative caches. On a cache access, the LRU information is read for the accessed set. Then the counter is incremented based on this LRU information if the access hits on the cache.

we need to add additional information about temporal ordering of sets to obtain detailed marginal gains [16].

4.2. Partitioning mechanisms

For set-associative caches, various partitioning mechanisms can be used to actually allocate cache space to each process. One way to partition the cache is to modify the LRU replacement policy. This 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 [2] can restrict virtual address to physical address mapping, and as a result restrict cache sets that each process uses. Column caching [4] can partition the cache space by restricting cache columns (ways) that each process 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.

4.2.1. Modified LRU replacement. In addition to LRU information, the replacement decision depends on the number of cache blocks that belong to each process (b_i). On a miss, the LRU cache block of the process (i) that caused the miss is chosen to be replaced if its actual allocation (b_i) is larger than the desired one ($c_i \leq b_i$). Otherwise, the LRU cache block of another over-allocated process is chosen. For set-associative caches, there may be no cache block of an over-allocated process in the set. In this case, the standard LRU replacement policy is used and evicts the LRU cache block in the set.

To implement this augmented LRU scheme in hardware, a processor ID is added to each cache block. A L2 cache that is shared by two processors requires one bit processor IDs. The space overhead of these IDs is negligible since they are only a few bits per cache block. We also need one cache block counter (b_i) per processor, which keeps the number of cache blocks belonging to the processor. The counter is decremented whenever a block is evicted from the cache, and incremented when a new cache block is brought into the cache. Note that the counters are only updated on a cache miss, and are not on the critical path of accessing the cache.

4.2.2. Column caching. Column caching is a mechanism that enables partitioning a cache at column or "way" granularity [4]. 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 subset 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 look-up is precisely the same as for a standard cache, column caching incurs no performance penalty during look-up.

4.3. Partition controller

The previous two subsections discussed two hardware mechanisms to enable cache partitioning: marginal gain counters and partition mechanisms. In our scheme, the operating system controls these mechanisms to properly partition caches. A partition module in the operating system has two major roles. First, it determines a desired cache allocation based on the marginal gain counters. Second, it properly updates the counters to reflect dynamic changes in program behavior.

Every T cycles, the operating system interrupts a running process and starts the partition module. The partition module first reads marginal gain counters to update its data structure for marginal gains. Based on the new marginal gains, it decides a proper partition for each process and changes the allocation at cache partition unit (modified replacement unit). Finally, the module clears the counters and restarts a process. To reduce the overhead, the partition period T should be long enough. We discuss this issue in more detail in the experiments section (Section 5).

4.3.1. Cache allocation. The algorithms in the previous section assume that we can control cache allocation at a cache block granularity, and we know marginal gains also at a block granularity. However, with way-counters for set-associative caches, we cannot accurately estimate marginal gains at a cache block granularity.

3B2 Version Number 7.51a/W (May 2 2001) {Kluwer}Supe/SUPE 28_1/02Suh.3d Date: 6/1/04 Time 13:19pm Page 16 of 26

Also, it is very difficult to control the cache allocation at a block granularity. Therefore, we allocate chunks of cache blocks at a time, referred to as a partition block.

Using way counters, we obtain marginal gains at a cache way granularity. Thus, we use a partition block that is the same size with one way of the cache (S blocks). The cache allocation to each process can only be multiples of S blocks. For this purpose, we define marginal gains of a partition chunk $g_{\text{way}}(k)$:

$$g_{\text{way}}(k) = \sum_{c=k \cdot S}^{(k+1) \cdot S-1} g(c). \tag{4}$$

To handle non-convexity, we randomly choose an initial allocation and use a greedy algorithm to decide a partition. After computing a new partition, we compare it with the previous partition and pick the better one to be a partition for the next partition period. The complete algorithm is as follows.

1. Initialize $\{c_1, c_2, \ldots, c_N\}$ randomly.
2. Find the process that will get the most benefit by having one more partition block (index i for which $g_{\text{way},i}(c_i)$ is largest), and the process that will lose the least by giving up one partition block (index $j(\neq i)$ for which $g_{\text{way},j}(c_j - 1)$ is smallest).
3. If $g_{\text{way},i}(c_i) > g_{\text{way},j}(c_j - 1)$, increase c_i and decrease c_j.
4. Repeat steps 3 and 4 until $g_{\text{way},i}(c_i) \leq g_{\text{way},j}(c_j - 1)$ (maximum D times).
5. Compare the new partition with the previous one, and choose the better one.

Considering that N (the number of simultaneous processes) and D (the number of partition blocks = associativity) are small, the overhead of computing the new partition is rather small. For the case when we have two processors sharing a eight-way L2 cache, the number of instructions for the computation should be well below 10,000. If we have a partition period of five million cycles, which we show is reasonable in the experimental section, the overhead of computing a new partition is less than 0.2%.

4.3.2. ***Counter update.***

Since characteristics of processes change dynamically, the estimation of $g_{\text{way}}(x)$ should reflect the changes. But we also wish to maintain some history of the memory reference characteristics of a process, so we can use it to make decisions. We can achieve both objectives, by giving more weight to the counter value measured in more recent time periods.

When a process begins running for the first time, all marginal gains are set to zero. The partition module updates the marginal gains ($g_{\text{way}}(k)$) every partition period by giving more weighting to new counter values:

$$g_{\text{way}}(k) = \delta \cdot g_{\text{way}}(k) + \text{counter}(k). \tag{5}$$

As a result, the effect of hits in the previous period exponentially decays, but we maintain some history.

5. Experimental results

This section presents the simulation results in order to understand the quantitative effects of our cache partitioning scheme. First, we describe our simulation framework based on the SimpleScalar tool set [3]. Then, we evaluate our partitioning scheme by running different sets of benchmarks on a CMP system with two processors sharing a L2 cache. Finally, we discuss how long the partition period should be.

5.1. Simulation framework

Our simulation framework is based on the SimpleScalar 3.0 tool set [3], which models speculative out-of-order execution. The original SimpleScalar is modified to have multiple processors sharing a L2 cache. A processor has its own L1 instruction and data caches, but all processors share one unified L2 cache. To model contention among multiple processors, the L2 cache bus and the memory bus were implemented in SimpleScalar.

Our partitioning scheme is implemented for the shared L2 cache. 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. For an eight-way associative L2 cache, eight counters per processor are added, which estimate the marginal gains. Based on the marginal gains, SimpleScalar decides a new partition every *T* cycles and updates counters. Finally, we implemented the augmented LRU replacement policy to actually control the number of block in the cache. To be conservative, we add 10,000 cycles overhead of computing a partition for every partition period.

The architectural parameters used in the simulations are shown in Table 1. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance. The size of the L2 cache is varied

Table 1. Architectural parameters used in simulations

Architectural parameters	Specifications
Processors	2
Clock frequency	1 GHz
L1 I-caches	64-KB, two-way, 32-B cache lines
L1 D-caches	64-KB, two-way, 32-B cache lines
L2 caches	Unified, eight-way, 64-B cache lines
L1 latency	2
L2 latency	10
Memory latency (the first chunk)	80
Instruction/data TLBs	Four-way, 128-entries
Memory bus	200 MHz, eight-B wide (1.6 GB/s)
Fetch/decode/issue/commit width	4/4/4/4
Load/store queue size	64
Register update unit size	128

from 1 to 32 MB. Unless indicated otherwise, the partition period is five million cycles, which is 5 ms in 1 GHz clock speed.

5.2. *Benchmarks*

To evaluate the cache partitioning scheme, we use several benchmarks from SPEC CPU2000 benchmark suite [9]: `art`, `mcf`, `vpr`, `swim`, `applu`, `vortex`, and `gcc`. Benchmarks that show different characteristics for the L2 cache are selected to represent various types of applications. Some benchmarks require a large L2 cache to achieve high performance, while less than 1 MB is enough for others. Some benchmarks get significant benefit by having more cache space, while others do not.

Figure 4 illustrates the characteristics of the benchmarks that we use. The first figure shows the IPC of each benchmark as a function of L2 cache size between 1 MB and 32 MB. The second figure shows the number of L2 cache misses per million processor cycles for various L2 cache sizes. To exclude the effect of an initial setup phase, the benchmarks are simulated after skipping the first three billion instructions.

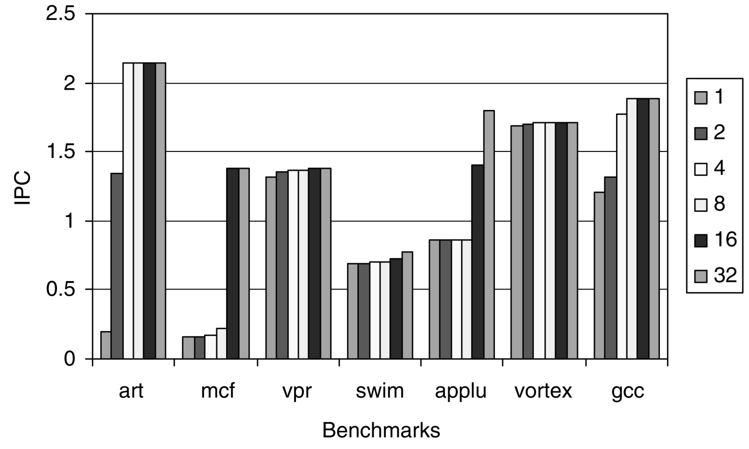
Figure 4(a) demonstrates that the performance of some benchmarks is very sensitive to cache size. The IPC of `art`, `mcf`, `applu`, and `gcc` can be significantly improved by having a larger L2 cache. This means that the performance of these benchmarks can be significantly degraded by other competing processes when the L2 cache is shared. Therefore, for these benchmarks, it is particularly important to manage cache space carefully to achieve a high IPC.

The figures also indicate that the standard LRU replace policy may not manage the cache space properly. The LRU policy tends to allocate more space to processes that generate more cache misses. However, processes with many misses do not necessarily benefit from having large cache space. For example, `swim` experiences many L2 cache misses as shown in Figure 4(b). But the IPC improvement of `swim` as we increase the L2 cache size is minimal compared to other benchmarks such as `art`, `mcf`, `applu`. Therefore, the standard LRU policy will not manage the cache properly when `swim` executes with `art`, `mcf`, or `applu`.

Finally, the figures illustrate that the characteristics of a benchmark is heavily dependent on the L2 cache size in the system. For example, `mcf` will look like a streaming application with little temporal locality if a system has a cache smaller than 8 MB. However, if the cache is larger than 16 MB, `mcf` shows high temporal locality. Thus, static analysis alone is not enough to determine proper cache allocation; compilers cannot tell whether an application shows high temporal locality or not. The characteristics of an application should be determined dynamically at run-time.

5.3. *Partition results*

Now we study the advantage and the weakness of our cache partitioning scheme. For the evaluation, we simulate four mixes of SPEC CPU2000 benchmarks. The first



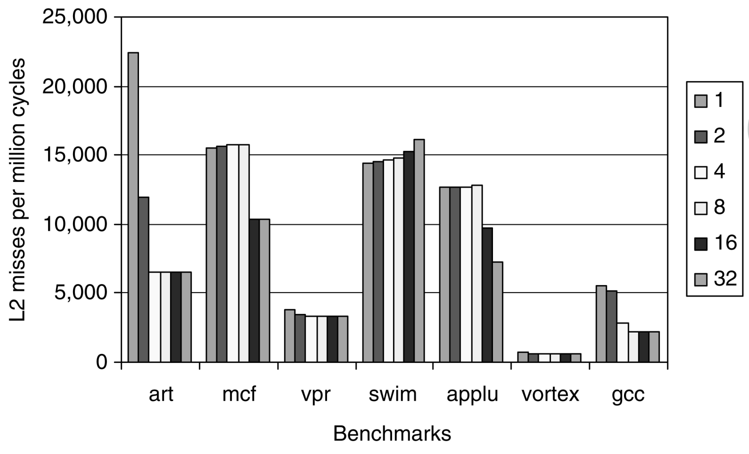


Figure 4. The L2 cache characteristics of the benchmarks used in simulations. (a) IPC as a function of L2 size. (b) The number of L2 misses per million processor cycles as a function of L2 size.

mix runs `art` and `mcf`, where each program has large marginal gain at different cache size. The second mix runs `mcf` and `swim`. In this case, both programs need a 16 MB or larger L2 cache to achieve high performance. The third mix runs `vpr` and `swim`, where both benchmarks only get minimal improvement by having more cache space. Finally, we run `vortex` and `gcc`. In this case, only `gcc` gets benefit by having a large cache (see Figure 4).

The simulations compare the IPC of the standard LRU replacement policy and the IPC of our partitioning scheme. In each case, two processes are run sharing the L2 cache. Each simulation was run for two hundred million cycles after skipping first three billion instructions of each process. The partition period T is five million cycles, and the weighting factor is set as $\delta = 0.5$. There is 10,000 cycles overhead modeling the operating system computing a partition every partition period.

Figure 5 illustrates the speed-up of our partitioning scheme over the standard

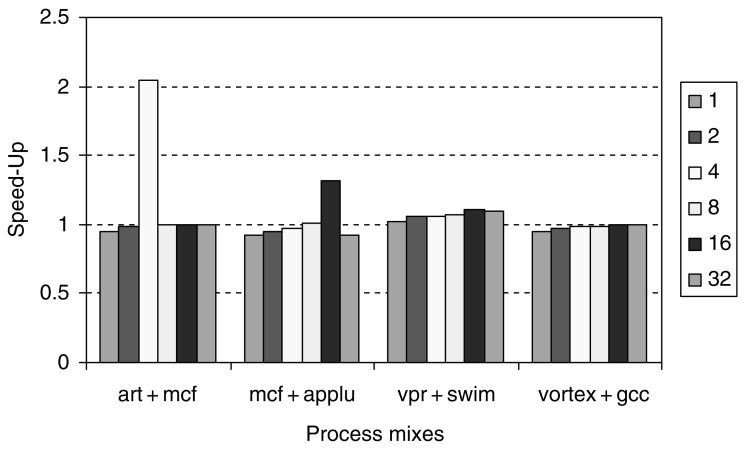


Figure 5. The speed-up of the partitioning scheme over the standard LRU replacement policy. $T = 5{,}000{,}000$, $\delta = 0.5$.

LRU replacement policy. The results are shown for various L2 cache sizes, which range from 1 MB to 32 MB.

The simulation results demonstrate that cache partitioning can improve the total IPC significantly; for a 4-MB L2 cache running art and mcf, partitioning doubles the IPC over the LRU policy. Partitioning also improves the IPC up to 32% for the second mix (mcf and applu). Our partitioning scheme allocates more cache space to the process with a larger marginal gain, whereas the LRU policy blindly allocates more space to the one with more cache misses. More interestingly, the process with a larger marginal gain changes depending on the process mixes. mcf is treated as a streaming process in the first mix, but the same mcf gets most of cache space in the second mix.

The figure also illustrates the relationship between the cache size and the effectiveness of partitioning. For small caches, partitioning does not seem to help since the size of the total workloads is too large compared to the cache size. In this case, changing the process schedule should be considered. 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 processes and the characteristics of the processes. In our experiments, cache partitioning improves the performance in the range of up to tens of MB.

Table 2 summarizes more detailed IPC information for the same set of simulations. The IPC of each process and the total IPC are shown for the LRU replacement policy and the partitioning scheme. The total IPC is just a sum of two IPCs.

The results in the table provide more insight into sharing caches among multiple processors and partitioning the cache. First, the experimental results demonstrate that performance degradation due to sharing the cache can really be significant. For example, consider the case when running art and mcf with a 4-MB L2 cache. If we

Table 2. Detailed comparison of IPCs between the standard LRU and the partitioned LRU strategy

	LRU IPC			Partition IPC			
Cache (MB)	`art`	`mcf`	Tot.	`art`	`mcf`	Tot.	Speed-up
1	0.1507	0.0441	0.1948	0.1428	0.0423	0.1852	0.95
2	0.5553	0.1001	0.6554	0.5468	0.0952	0.6420	0.98
4	0.9698	0.1263	1.0962	2.0761	0.1622	2.2382	2.04
8	1.9490	0.1803	2.1294	1.9490	0.1803	2.1294	1.00
16	2.1727	0.7476	2.9204	2.1727	0.7476	2.9204	1.00
32	2.1932	1.3682	3.5614	2.1932	1.3682	3.5614	1.00
	`mcf`	`applu`	Tot.	`mcf`	`applu`	Tot.	
1	0.0922	0.4821	0.5744	0.0879	0.4414	0.5292	0.92
2	0.0900	0.4757	0.5656	0.0859	0.4503	0.5362	0.95
4	0.0926	0.4804	0.5730	0.0909	0.4672	0.5582	0.97
8	0.1073	0.4958	0.6030	0.1133	0.4921	0.6054	1.01
16	0.5958	0.7804	1.3762	0.9921	0.8269	1.8190	1.32
32	1.1373	1.1640	2.3012	1.1280	1.0037	2.1316	0.93
	`vpr`	`swim`	Tot.	`vpr`	`swim`	Tot.	
1	0.7304	0.5987	1.3290	0.7972	0.5610	1.3582	1.02
2	0.7981	0.6115	1.4096	0.8818	0.6039	1.4858	1.05
4	0.8817	0.6289	1.5106	0.9693	0.6370	1.6064	1.06
8	0.9404	0.6427	1.5830	1.0396	0.6624	1.7020	1.08
16	0.9949	0.6605	1.6554	1.1461	0.6900	1.8360	1.11
32	1.0513	0.6879	1.7392	1.1892	0.7090	1.8982	1.09
	`vortex`	`gcc`	Tot.	`vortex`	`gcc`	Tot.	
1	1.6420	1.5122	3.1542	1.5882	1.4882	3.0764	0.98
2	1.6896	1.5662	3.2558	1.6691	1.5303	3.1994	0.98
4	1.7058	1.6910	3.3968	1.6993	1.6323	3.3316	0.98
8	1.7168	1.7624	3.4792	1.7165	1.7603	3.4768	1.00
16	1.7224	1.7682	3.4906	1.7224	1.7682	3.4906	1.00
32	1.7227	1.7683	3.4910	1.7227	1.7683	3.4910	1.00

execute only one process at a time, the IPC of `art` and `mcf` will be 2.1489 and 0.1719, respectively from Figure 4. If we run them simultaneously on two separate processors without cache partitioning, the total IPC is only 1.0962. In this case, it is better to just idle the second processor than use it unless we partition the cache.

Cache partitioning also improves the performance by reducing the memory bandwidth. In the table, consider the case of running `art` and `mcf` on a 4-MB L2 cache again. In this case, our partitioning scheme allocates less space to `mcf` than the LRU scheme would allocate. Thus, the miss-rate of `mcf` is higher with cache partitioning. However, the IPC of `mcf` is still higher than in the case of the LRU policy (from 0.1263 to 0.1622). This happens because we reduce the memory latency by reducing the total bandwidth usage. The effect is also shown in the results of the second and the third mixes.

Although our partitioning scheme significantly improves the performance when the LRU is wrong, the experimental results also show some cases where the partitioning degrades the performance by a few percent. There are two weaknesses of our partitioning scheme that cause a problem. First, we can change the partition only every partition period *T* and our partition granularity is *S* blocks, whereas the standard LRU policy can change the allocation every cycle on a cache block basis. Due to this limitation, the partitioning can result in slightly worse performance than the LRU policy if the LRU policy happens to result in a near-optimal allocation. This is why the partitioning does not help for the last mix (`vortex` and `gcc`). The LRU policy does the right thing for this mix by allocating more space to `gcc`. However, the performance degradation due to this reason is very small compared to the significant benefit of our scheme for the other cases.

The other weakness of our partitioning scheme is the fact that we can only estimate marginal gains using cache ways. This mechanism is based on the assumption that cache blocks are randomly distributed over different sets. Unfortunately, some programs only use a part of cache sets heavily. In this case, our algorithm is likely to allocate an unnecessarily large amount of cache space to the program. Figure 6 illustrates this problem for the second mix (`mcf` and `applu`). The figure shows the number of cache blocks allocated to `mcf` over time. The black line represents the number of blocks that are actually in the cache, the gray line represents the desired number of cache blocks allocated by our partitioning scheme. As shown in the figure, the partitioning scheme over-allocates cache space to `mcf`. The allocated space cannot be fully used by `mcf` since `mcf` uses only a part of the cache sets heavily and therefore it is likely to replace its own block in a heavily accessed set. The figure shows that the number of cache blocks of `mcf` does not increase in the middle of an execution. This allocation limits the

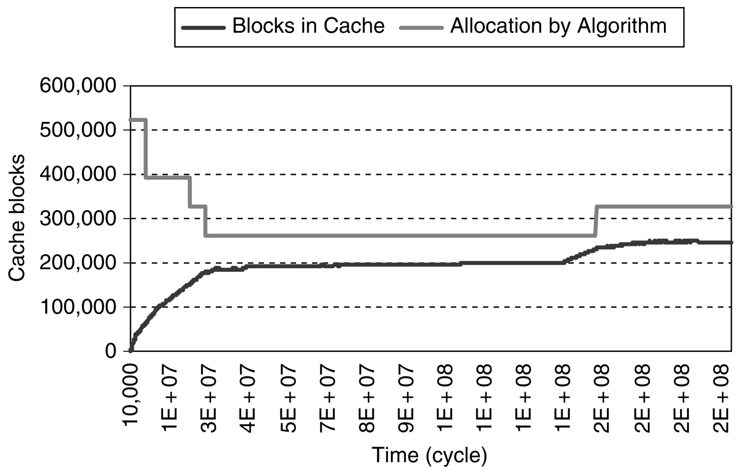


Figure 6. The cache allocation for `mcf` when running with `applu` using a 32-MB cache. The black line represents the actually number of cache blocks in the cache. The gray line represents the desired allocation decided by the partition scheme.

3B2 Version Number 7.51a/W (May 2 2001) {Kluwer}Supe/SUPE 28_1/02Suh.3d Date: 6/1/04 Time 13:19pm Page 23 of 26

performance of `applu` since `applu` will get fewer cache blocks. This problem should disappear as we use better mapping functions that maps an address to a cache set [8, 19].

5.4. Partition period

Thus far, we set the partition period T to be five million cycles for all our experiments without any discussion. In this subsection, we study how long the partition period should be. Figure 7 illustrates the effects of the partition period on the speed-up. Each curve represents a process mix with the particular L2 size indicated in the legend.

As shown in the figure, the performance degrades when the partition period is either too short or too long. Short partition periods hurt the performance due to two reasons. First, there is an overhead of computing a new partition every partition period. Second, short partition periods hurt the estimation of marginal gains. We age our marginal gains every partition period by multiplying them by δ. If the partition period is too short, the marginal gains quickly lose the past history.

On the other hand, if the partition period is too long, a partition cannot track dynamic changes in the program behavior quickly enough, which results in poor performance. However, program behavior does not change very quickly in our experiments. As a result, any partition period between 500,000 cycles and 10,000,000 showed the best possible performance. Therefore, we can make the partition period the same as the time slice, and handle re-partitioning as a part of the context switching process.

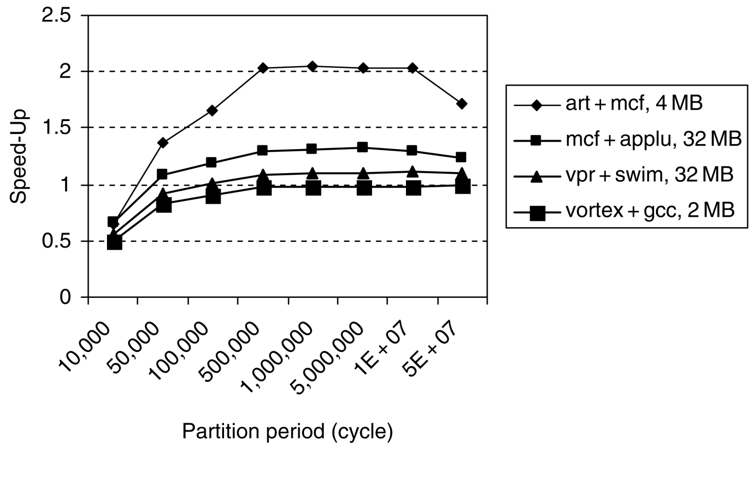


Figure 7. The speed-up as a function of the partition period. Each curve represents a process mix with the particular L2 size indicated in the legend.

M11146 Kluwer Academic Publishers The Journal of Supercomputing (SUPE) *Tradespools, Frome, Somerset*

6. Conclusion

Low IPC can be attributed to two factors, data dependency and memory latency. Executing multiple processes simultaneously such as in CMP/SMT systems mitigates the first factor but not the second. We have discovered that simultaneous execution of multiple processes may exacerbate the problem when the executing processes require large caches. That is, when multiple executing processes interfere in the cache, even multiple processing units cannot be well utilized because not all required data is present in the memory.

We have studied one method to reduce cache interference among simultaneously executing processes. Our on-line cache partitioning algorithm estimates the miss characteristics of each process at run-time, and dynamically partitions the cache amongst the processes 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 the shared L2 cache of a CMP simulator based on the SimpleScalar tool set. 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 processes. Additionally, our partitioning algorithm can solve the problem of process 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, processes that execute simultaneously should be selected carefully considering their memory reference behavior. Cache-aware job scheduling is a subject of our ongoing work [16, 17].

Even without CMP/SMT, one can view an application as multiple processes executing simultaneously where each process 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.

Acknowledgments

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.

Notes

1. Hereafter, we use a term “process” to represent both process and thread rather than explicitly using “process/thread”.

References

1. D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *The 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, 1999.
2. 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.
3. D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.
4. D. T. Chiou. Extending the reach of microprocessors: Column and curious caching. Ph.D. Thesis, Massachusetts Institute of Technology, 1999.
5. W. J. Dally, S. Keckler, N. Carter, A. Chang, M. Filo, and W. S. Lee. M-Machine Architecture v1.0. Technical Report Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, 1994.
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. 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.
9. J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 2000.
10. Intel. Intel StrongARM SA-1100 microprocessor, 1999.
11. J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.
12. M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction. In *The 34th Annual IEEE/ACM International Symposium on Microarchitecture*, 2001.
13. H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), 1992.
14. R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 1995.
15. G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *The 15th International Conference on Supercomputing*, 2001a.
16. G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *The 8th High-Performance Computer Architecture*, 2002.
17. G. E. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *7th International Workshop on Job Scheduling Strategies for Parallel Processing (in LNCS 2221)*, pp. 116–132, 2001b.
18. D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), 1992.
19. N. Topham and A. González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2), 1999.
20. D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multi-threading: Maximizing on-chip parallelism. In *The 22nd Annual International Symposium on Computer Architecture*, 1995.
21. O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. The minimax cache: An energy-efficient framework for media processors. In *The 8th High-Performance Computer Architecture*, 2002.

22. S. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *The 8th High-Performance Computer Architecture*, 2002.
23. M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop in 33rd International Symposium on Microarchitecture*, 2000.