A Dynamically Reconfigurable Cache for Multithreaded Processors

Alex Settle [a,*] Dan Connors [a] Enric Gibert [b]
Antonio Gonzalez [b]

[a] University of Colorado at Boulder
Department of Electrical and Computer Engineering
425 UCB, Boulder, Colorado 80309

[b]Department of Computer Architecture
Politecnic University of Catalunya
C/. Jordi Girona, 1-3
Modulo C6 (Campus Nord) E-08034 Barcelona Spain

*Corresponding author. Tel.: +1 303 786 0560; Fax +1 303 492 2758 ;
E-mail: settle@colorado.edu

Abstract

Chip multi-processors (CMP) are rapidly emerging as an important design paradigm for both high performance and embedded processors. These machines provide an important performance alternative to increasing the clock frequency. In spite of the increase in potential performance, several issues related to resource sharing on the chip can negatively impact the performance of embedded applications. In particular, the shared on-chip caches make each job's memory access times dependent on the behavior of the other jobs sharing the cache. If not adequately managed, this can lead to problems in meeting hard real-time scheduling constraints. This work explores adaptable caching strategies which balance the resource demands of each application and in turn lead to improvements in throughput for the collective workload. Experimental results demonstrate speedups of up to 1.47X for workloads of two co-scheduled applications compared against a fully-shared two-level cache hierarchy. Additionally, the adaptable caching scheme is shown to achieve an average speedup of 1.10X over the leading cache partitioning model. By dynamically managing cache storage for multiple application threads at runtime, sizable performance levels are achieved, which provides chip designers the opportunity to maintain high performance as cache size and power budgets become a concern in the CMP design space.

Keywords: Embedded, multithreading, cache sharing, memory, fairness

1 Introduction

For years, the steadily growing clock speed of new uniprocessors has been relied upon to consistently deliver increasing performance for a wide range of applications. However, uniprocessor frequency scaling has stalled, driving microprocessor companies to add value by producing chips that incorporate multiple processors. In only a few short years, every performance-oriented and high-performance embedded system manufactured will contain a chip-multiprocessor (CMP). The current CMP generation explores multithreading support in the context of establishing inexpensive throughput computing on a single core. Simultaneous Multithreaded (SMT) [15][27][26] processors share the resources (ALUs, branch target buffers, etc.) of one physical processor between multiple "virtual processors" that simultaneously execute each cycle. The most commonly available SMT processor is the Intel Pentium-4 processor with Hyper-Threading [8] support. Relatedly, IBM released the PowerPC RS64-IV [3] which is a commercial implementation of a coarse-grain multithreading processor. Next generation processors such as the IBM Power 5 and the Intel Itanium Montecito provide multithreading support on each of the on-chip cores. As the power budget is limiting the ability of uniprocessor performance to increase considerably from one process generation to the next, chip designers are increasingly turning to increased parallelism to drive performance.

Advances in computing performance have enabled a myriad of advances in science and technology. Faster processors have enabled detailed simulation of the global climate and sophisticated biomedical modeling of the human genome. Likewise, revolutions in digital media distribution have been supported by the advances in computing performance that allow portable devices to decode compressed media. Consequently, continued breakthroughs in these areas are now intimately tied to continual improvements in processor performance. Increasing across each of the CMP generations will be the need for greater hardware support to manage system efficiency. As the amount of

on-chip concurrency increases so too will the penalties associated with resource sharing unless dynamic resource management systems are employed. In general, advances can only be realized if application resource requests with differing characteristics are mapped to the system resources that best execute a task given certain real-time computing constraints.

Recent studies [4, 9] have investigated the effects of resource sharing in the processor core of SMT systems and have shown that both the instruction fetch and issue policies are critical to achieving optimal performance. In addition to the shared processor data path resources, the memory hierarchy is also shared by multiple threads and is a critical area for improving co-scheduled execution. Interference occurs in the cache system when data belonging to one thread is evicted by a cache line from another thread. Thus, in a processor that supports two or more thread contexts, important data belonging to one thread is often replaced by less important data from other threads. Additionally, the rate of access to the cache by each thread determines the degree of interference that can result. Furthermore, this interference affects the overall storage available to each thread which in turn influences the rate of cache requests made by each of the threads. Thus, measuring the impact of cache interference on performance is a multi variable problem that is considerably challenging to perform at runtime. Thus, to effectively control the performance penalties that arise in shared caches, the system must accurately manage the interference to resource allocation problem.

Current SMT memory systems are designed for uniprocessors and therefore lack the ability to detect and manage the interference that occurs between active jobs. Moreover the effect of cache interference on a given job is difficult to measure since it depends on a number of factors: other co-scheduled jobs, characteristics of each job's current execution phase [5], and the re-use potential of evicted cache lines. Furthermore, the difficulty of measuring the anticipated penalty of interferences increases with the number of thread contexts that are supported by the processor. The problems that arise from having a shared cache depend upon the overall storage demands of the work-

ing set for each job and the spacial and temporal locality that they exhibit. If two jobs share a cache and their accesses are concentrated in small but different groups of cache sets, then they should experience relatively few interference misses. However, if one of the jobs has a large working set that is accessed frequently, it will likely cause a large number of interference misses with the other thread. Multithreaded memory systems therefore must detect the different resource requirements of the jobs and assign the resources accordingly.

By explicitly managing cache interference in multithreaded processors, this work makes the following contributions. First, it provides a dynamic hardware cache management system for various workloads of two independent applications. The proposed system specifically addresses the balance between both inter and intra-thread cache conflict misses and addresses both synchronous and asynchronous feedback directed cache optimizations. Additionally, the cache models are tested in both a cycle accurate simulator, and on workloads run on an Intel Xeon processor manged by the PIN dynamic instrumentation tool. The remainder of this paper is organized as follows. Section 2 describes the motivation and high level details of dynamic partitioning. The impact of inter-thread cache interference is discussed in Section 2.1 followed by a discussion of the experimental workloads in Section 4 and the details of the partition mechanisms in Section 3. The results are presented in Section 5 followed by a discussion of the related literature in Section 6. Finally, the future work and conclusions are discussed in Section 7.

2 Motivation

Due to the fact that multithreaded and multicore processors are emerging in the market place as the dominant architectures for high performance computing, they are rapidly gaining importance in embedded computing. Given that the running applications share certain processor resources at run time, effective resource management is an increas-

ingly more critical problem for these architectures. As such, the organization of cache memory in these systems is becoming an important design consideration. The most recent multi-core processors are able to simplify resource sharing issues by providing separate cache hierarchies for each of the cores. The dual core Intel Itanium Montecito for example provides 12MB of L3 cache to each processor core [20]. Although cache hierarchies of this size are likely more than adequate for most common workloads, it is unlikely that the size of the cache will be able to continue increasing at this rate as more cores are integrated onto the chip. As power constraints continue to regulate chip designs, cache sizes may soon have to be reduced in order to build the next generation multi core machines.

For a given cache size budget, a trade-off must be made when considering using fully private or fully shared caches. In the case of private caches, internal fragmentation can be an issue. Consider the case where the working set size of a given thread is less than the total combined cache storage but considerably more than the individual cache. If this benchmark is run in parallel with a workload that has a considerably smaller cache footprint, then one workload will suffer an excessive number of conflict misses while the other only touches a fraction of its available storage. In a properly managed shared cache organization, however, both applications could access sufficient storage leading to greater processor performance. The problem with shared caching however, is that one benchmark with a large working set can severely limit the performance of one with a small working set. In the shared cache model, the temporal and spatial locality of the combined job mix determine the size and location of cache storage allocated to each thread. This means that jobs with high relative cache access rates can claim ownership of the majority of the cache resources, making it increasingly difficult for the other applications to reuse their cached data. Thus, for a given cache size, the working set sizes [1] of the workloads sharing the cache determine the expected performance loss due to sharing.

Figure 1 is an example of the cache allocation from a co-scheduling interval run-

ning the benchmarks 164.gzip and 181.mcf. Modern SMT aware operating system job schedulers such as [19, 21] would avoid co-scheduling these jobs because the capacity misses experienced by 181.mcf considerably limit the performance of 164.gzip. From the figure, the LRU replacement policy assigns 181.mcf roughly 80% of the available cache storage leaving 164.gzip with the remaining 20%. In this case, the additional storage available to 181.mcf contributes very little to improving its overall IPC. As shown in Figure 2, the IPC of 181.mcf running on a 2MB L2 is almost identical to the IPC when the same application is run on a 512KB cache. Thus, it is clear that 181.mcf's individual performance will be changed very little as its cache storage is reduced. Similarly, the figure shows that 164.gzip performs equally well on a 512KB cache as it does on the larger sizes. Thus, if given access to more of the cache, it is likely that 164.gzip will approach its maximum single thread IPC.

Figure 3 shows the cache allocation that results with one of the partitioning mechanisms introduced in this paper. In this case, the storage available to each application is nearly the opposite of the default case. In this scheme 164.gzip owns roughly 65% of the storage while 181.mcf has the remaining 35%. Under this allocation, the overall processor utilization is improved considerably without causing significant slowdown to 181.mcf. The benchmark 164.gzip in this case is able to maintain a greater set of its cached data and as a result execute considerably more instructions than in the default cache organization. One of the primary goals of cache partitioning therefore, is to identify these types of workload imbalances and correct them dynamically by throttling the cache allocation mechanism. Additionally, enabling the hardware to adapt to these types of problematic workloads will provide greater flexibility for the job scheduler in a multiprogrammed environment.

In operating system scheduling approaches to handling negative interference due to multithreading, the goal is to identify groups of jobs that exhibit little performance degradation due to sharing [19, 21] when co-scheduled. The hardware techniques presented in this paper however, aim to improve the performance of jobs mixes that suffer

from considerable performance degradation when sharing the L2 cache. Thus, by optimizing the workloads that a job scheduler would normally try to avoid co-scheduling, this technique allows for greater flexibility in the scheduling algorithm.

When considering the effects of shared caching on individual applications, their working set sizes should be considered. Figure 2 shows the IPC of each of the benchmarks while running on L2 caches ranging from 512KB to 4MB in size. This graph will help in predicting the expected interference that results while sharing the cache between any two of the workloads. The benchmarks 179.art, 188.ammp, 181.mcf, and 300.twolf are all capacity limited for smaller cache sizes, while the others perform comparatively well on the 512KB cache. The 512KB cache size therefore provides an interesting challenge for co-scheduling jobs.

2.1 Detecting Inter-thread Interference

In a single-threaded system, cache misses are classified as either cold, capacity, or conflict misses [12]. In a shared cache, however, misses can be attributed to conflicts with other threads. Inter-thread interference occurs when data from one thread is evicted by a cache line belonging to another thread, and it can cause a miss that would not have happened in a private cache. The challenge with understanding inter-thread interference however, is that it varies depending upon other jobs in the system and the timing of accesses to each cache line. Thus, while it is possible to measure the number of accesses to a cache line by each thread during a given time interval, it is difficult to estimate the number of misses that would occur due to inter-thread interferences. Additionally, cache interferences and SMT fetch policies such as *icount* [25] are closely coupled. This means that changes in cache miss patterns can cause a change in the sequence of issued instructions.

Figure 4 shows an example of the challenges involved in accurately measuring the interferences between threads. The figure is a directed graph whose nodes represent each thread and whose edges point to the thread that caused an eviction for the source

thread. Each time a cache eviction occurs, a counter for the requesting thread can measure whether or not the evicted cache line belongs to the same thread. In the figure, data requests from thread A have caused 100 cache lines from thread C to be evicted, 45 from thread B, and 30 self evictions. Here, in this example, each thread interferes with all the others. If any one of the nodes is removed from the graph, then some of the interferences will be removed, while others will be redistributed among the remaining nodes. Consider removing thread A from the graph as shown in Figure 4. Initially, it appears that the total interference count will be reduced by the number of interferences between A and C plus A and B. However, once A is removed, it changes the sequence in which B and C access the cache. This means that the number of misses between B and C will likely increase. Additionally, misses that were caused by thread A could have simply been masking evictions that would have occurred between B and C anyway. Thus, while the counters required to measure the miss counts shown in the graph are not difficult to implement, they do not provide sufficient information to accurately determine the actual cause of the inter-thread evictions.

Various modern processors implement different on-chip performance monitoring units (PMUs) to capture detailed execution information. For example, the Pentium-4 [22], Itanium [13], PowerPC processors [7], and others provide facilities for sampling a limited number of architecture events. While these conventional performance counters provide coarse-grain information, more fine-grained counters are necessary to steer cache partitioning decisions. In this paper, robust interference analysis was performed to help with the design of cost effective hardware inter-thread interference techniques. Two techniques are presented, the first being a synchronous allocation throttling mechanism and the second an asynchronous cache line allocation controller. The discussion of cache partitioning in the remainder of the paper, shows the details of each model, in addition to a performance comparison between them and the best partitioning technique available in the literature [11].

3 Cache Partitioning Description

Hardware cache partitioning is a method for controlling the allocation of cache storage resources in a shared cache. While the name implies a physical barrier around specific regions of the cache, this is not always the case. Virtual partitions can be controlled by the LRU replacement algorithm in order to modulate the distribution of resources between independent access sequences. There are two distinct components involved in the implementation of a dynamically partitioned cache: the mechanism used to make the partitions, and the runtime partition management hardware. The partitioning mechanism determines the properties of the management system. Two different partitioning mechanisms have been proposed in the literature [24, 14, 4]. The first is to make the LRU replacement algorithm responsible for controlling the total number of cache lines allocated to each thread. While the second physically divides the cache into regions aligned with the cache's natural way boundaries. In this model, data belonging to given application is confined to one or more of these regions. Additionally, the partition control mechanism is driven by feedback from the running application. Partitions can be managed either synchronously or asynchronously. Synchronous partition control consists of the system periodically sampling performance monitoring registers that are used to drive partition changes. Similarly, asynchronous partition controllers react to changes in the cache state based on the occurrence of one or more events. This paper introduces partition mechanisms and controllers from each of the above categories.

Instead of relying on the LRU replacement algorithm to control a virtual partitioning mechanism, a cache can be partitioned along physical boundaries that exist naturally in hardware. Since each *way* in a modern set-associative cache is essentially implemented as a small direct mapped cache [2], the ways can serve as the partition boundaries. This partitioning mechanism is shown in Figure 5. In the figure, each thread has a bit vector that indicates the set of ways that the thread belongs to. When a cache line replacement occurs, the LRU algorithm first consults the vector correspond-

ing to the thread which made the cache request. Then it selects the LRU line from one of these ways and replaces it as it normally would. On a cache lookup however, all of the cache ways are searched, obviating the need to flush a thread's data from a way after the way is removed from its bit vector. This technique is referred to as column caching [6] and can be implemented with different levels of granularity. In this work, threads are allowed to both share cache ways and to hold them privately. The partition control algorithm uses information based on the threads' interactions to determine whether or not it is appropriate to share a given cache way.

The column cache partition is managed by a synchronous feedback system. In this model, performance counter registers are sampled every 1 million processor cycles in order to determine the cache partition for the next interval. When two or more threads are active in the cache, each thread's individual cache behavior can influence the behavior of the other. When running under the normal LRU cache replacement policy, the cache storage allocated to an application is proportional to the rate of cache requests. Thus, the application that makes requests most often will have the most cache storage. More importantly, there is no requirement that the application uses this storage efficiently. This can lead to cases where an application that normally exhibits high levels of temporal locality can be penalized by a program with very poor temporal locality. This observation led to the idea of monitoring global cache reuse behavior to control the cache storage available to each thread. By assigning programs that exhibit high degrees of global data reuse a larger percentage of the available cache storage, the chances of increasing processor utilization increases.

For the first partition control mechanism, the reuse count of each evicted cache line is recorded upon eviction and summed to get the average reuse value for the sampling interval. Along with this value, the percentage of cache storage already allocated to the thread is recorded. The reuse weight is then recorded by multiplying the average reuse by the percentage of cache lines not already allocated to the thread. By factoring in the cache allocation, threads that have low reuse due to lack of cache storage can

compete with threads that have high reuse due simply to owning the majority of the cache lines. After computing the reuse weight, the thread with the highest weight is selected as the target. The cache partition is considered for modification based upon the resource requirements of the target thread. The target thread can gain access to more cache resources in one of two ways: it can be granted access to one more cache way, or a thread that currently shares a way with the target thread can be removed from that way. Each of these two operations has an associated cost and benefit which are based upon changes in both inter and intra thread interference.

A score is computed based on the expected changes to the cache hit performance that results from the new cache map. The score due to removing a thread from a way that is shared with the target thread is computed by subtracting the performance loss from the expected gain. The loss refers to the increase in misses that will result when the thread is removed from a way. This loss will be equivalent to the number of hits that this thread currently makes in the LRU cache way. Thus, each thread requires a special register that counts the hits that it makes to the LRU cache line. The expected gain is computed by first recording the access count per way that this thread makes. When the thread is removed from one cache way, it can be approximated that the number of accesses to the way will be reduced by the access per way count of the thread. Finally, the expected drop in inter-thread cache evictions to this way is computed by taking the per way inter-thread interference counter and dividing by the number of threads that will occupy the way. Algorithm 1 shows this gain to loss formula in more detail.

Next, the potential benefit due to adding *target* thread to a new way is computed. In this case, the expected gain is the number accesses made by *target* thread distributed across all of its ways. This gain results in *target* thread having more cache storage available to it. Thus, the thread access count is divided by the total number of ways that it owns. The loss term is approximated by the number of accesses that the other threads in the system currently make to the proposed way. Here an access counter for each cache way is used to collect this information. The loss term represents the

increase in inter-thread interference that will occur when *target* thread is introduced to this way. The overall benefit for adding *test* thread to the way is the difference between the gain and the loss.

3.1 Reuse

The second cache partition control algorithm exploits asynchronous feedback from the cache system and regulates cache allocation with the LRU replacement policy. In this case, the LRU algorithm is modified to use reuse information of the other threads in the system, before selecting an LRU candidate. When the thread id of the cache request differs from that of the normal LRU candidate, the cache controller checks the reuse of the candidate line to determine its potential for harming the system performance. The reuse is simply the cache access frequency counter that is used in least frequently used cache replacement policies. If the reuse rank of the candidate thread relative to the other cache lines in the set is higher than a threshold value, the line is not considered for eviction. Instead the LRU - 1 line is evaluated and the process repeated. If there are no options available under this scheme, the algorithm reverts back to the normal LRU candidate. By consulting the reuse information of each potential victim cache line, this algorithm helps increase the time that data from another thread stays in the cache. Thus, in the case where one thread has a very high cache access frequency, this technique will make it less likely for the high frequency thread to evict important data belonging to another thread that accesses the cache much less often. In the context of a real time job scheduler, the cache allocation for high priority threads can be weighted to favor these threads over the others.

4 Experimental

The benchmarks were compiled with the IMPACT compiler using a set of aggressive profile-guided optimizations, including in-lining and super-block formation. Each of the benchmarks was fast forwarded past the initialization code, then the processor state

was warmed up for 10 million clock cycles before performance statistics were collected. The simulation time was set to 200 million cycles multiplied by the number of threads in the system, similar to the experimental technique used in [28]. For these experiments, the set of benchmarks in each workload was unchanged over the lifetime of the simulation. A cycle accurate simulator based on the IMPACT Lsim simulator was used to conduct the experiments. The simulator supports both multi core and multi threading execution models. It is an in-order processor core that runs aggressively optimized and scheduled code from the IMPACT compiler. The details of the simulator are shown in Table 1.

The results are presented against both a baseline fully shared L2 cache and the marginal gains algorithm introduced in [24]. Two cache allocation models are studied in this work. The first uses a synchronous feedback mechanism that samples hardware registers every 1 million cycles in order to determine whether a change in the cache partition is warranted. The second introduces an asynchronous feedback mechanism that adjusts cache allocation based on the reuse behavior of the individual cache lines. In this approach, the cache allocation policy can adapt more quickly to the local changes in behavior of each program.

In addition to conducting experiments on a microarchitecture simulator, the cache models were tested on applications under the control of the PIN binary instrumentation tool. This was done in order to analyze a set of modern scientific applications that prove difficult to execute on a simulator. Additionally, this experimental framework allows for analysis of the cache behavior related to programs running on the native hardware. In this environment, the memory model consumes address traces from any number of applications running under PIN on the host architecture. The cache model enforces the access timing delays that occur at each cache level in addition to recording hit and miss rates. The internal timing in the cache model regulates the rate at which the PIN tool emits addresses from each application. Thus, it can emulate the effects that cache interferences have on the issue rates of each individual application in a multithreaded

environment. Since this model does not simulate the core processor microarchitecture, it is limited to measuring changes in hit and miss rates for a given workload. The host machine in these experiments is an Intel Xeon 2.40GHz processor. The benchmark applications were all compiled with gcc at optimization level O3.

4.1 Workloads

Figure 2 lists the set of SPEC 2000 benchmarks that were used in the experiments. The benchmarks were selected so as to produce a mix of applications whose working sets stress the cache system, along with others that perform well at the 512KB L2 cache size. Additionally, three new applications(sonic, vorpal, blast), which are representative of modern scientific workloads were introduced. The purpose of these applications was to provide insight into the behavior of programs that perform tasks which will drive the direction of future processor development. The benchmark sonic [18] is a speech recognition program which has complicated control flow and uses pointer intensive data structures such as decision trees. This application is typical of enterprise programs that would likely be run on a cluster of servers in order to perform tasks such as automated business phone services. Next, the benchmark BLAST (Basic Local Alignment Search Tool) [16] is a sequence analysis program used to characterize protein structures. The sequences found in a given protein are compared to those found in sequence data bases in order to determine the statistical significance of a given match. Lastly, the benchmark vorpal is a plasma physics simulator. It provides for the visualization and simulation of charged fluids and particles in order to model concepts critical to high energy physics research [17]. These applications combined represent some of the software demands that will be driving the performance requirements of both embedded and high performance systems.

5 Results

Workloads of 2 benchmarks were run under each of the cache partition models and were examined on a 512KB L2 cache with a 300 cycle l2 miss penalty and 12 cycle L2 access time. The speedup shown in Figure 6 is measured relative to a fully shared cache hierarchy. Both of the partition control mechanisms are compared against the marginal gains cache partitioning mechanism introduced by Suh in [24]. In the figure legend, the dynamic partition heading refers to the column cache configuration discussed in the previous section and the modified LRU is the asynchronous allocation control algorithm. From the graph, the dynamic partition model reaches speedups of up to 1.46X while maintaining an average improvement of 10% over the marginal gains approach. Although there are some workloads (art_equake and art_mcf) where marginal gains outperforms the dynamic partition scheme, the dynamic model has a much less erratic performance than the others. Similarly, in most cases where the dynamic approach performs well, the modified LRU approach also makes considerable performance gains. Most notably the workload of art_gzip sees a 20% improvement. In this graph, the workloads that get the greatest performance are ones that have a large difference in the working set sizes of the individual applications. In the case of mcf_gzip for example, performance is gained by restricting the cache resources available to mcf in favor of gzip, the application with high cache reuse.

The fairness of the partitioning algorithms was evaluated next. Figure 7 shows the harmonic mean of each thread's IPC when compared to its IPC running alone on a uniprocessor. The harmonic mean values of each of the partitioning models are compared against the base configuration. Harmonic mean values that remain greater than or equal to the base value show that the IPC of both threads was improved to get a net benefit for the workload. Cases where the harmonic mean drops below the base case indicate either a net performance loss, or that workload performance came at the cost of a penalty to one of the threads in the workload. In this graph the legend entry reuse

refers to the modified LRU algorithm and the entry named part is the column cache model. In this example, both models exhibit good overall fairness for the workloads where the optimizations lead to speedup. The notable exception is the workload of ammp gzip where performance for the modified LRU case comes at the drop in individual performance of the ammp benchmark. In this experiment, the goal is to boost the performance of at least one thread while keeping the other thread as close as possible to the base case. In a real-time operating system however, priority can be given to a specific thread so as to provide enough cache storage to keep its IPC both high and predictable.

In addition to running experiments on the cycle accurate simulator, the PIN based model was used to measure the change in hit count that each cache partition model made on the host processor. For each workload the gain in hits is normalized with respect to the base processor configuration. In this graph, the column cache model outperforms the others by reaching L2 hit count improvements of up to 20% over the base case and an average of 4%. These results are consistant with the results from the simulation. In nearly all of the cases where performance improves, the benchmark art is co-scheduled in the workload. This suggests that improvement comes from limiting the cache storage for art in order to increase storage for the other thread with higher IPC and L2 cache reuse.

Lastly, the cache performance of the partitioning techniques were tested on a CMP system. In this experiment, the PIN based cache model was configured to provide private data caches for each of the jobs and a shared L2 cache. By having private data caches, address requests do not experience interference until reaching the L2cache. Figure 9 shows the hit count normalized against the base cache configuration, for each of the partitioning models. The results are similar to those presented in Figure 8 however, in this configuration the marginal gains algorithm does slightly better than in the SMT configuration. In this case where the data caches are private, improvements to the L2cache hit rate are not likely to drive improvements to the data cache due to the

reduction of cache interference. On average, the column cache model outperforms the others with an increase in L2 hits of 4%.

6 Related

While there exists a large body of literature concerning cache performance for uniprocessors, there is comparatively little on the subject of caching for multithreaded processors. Cache partitioning was first introduced as a way to improve the performance of workloads running in in a time-shared uniprocessor environment [23, 6]. The concept of column caching was introduced by Chiou in [6] as a way to make the timing of cache accesses for real-time embedded systems applications more predictable.

Recently, cache partitioning has been used in the domain of multithreaded processors to address the memory latency problems introduced by a cache between. In particular Suh et al. [24] introduced the idea of hardware cache partitioning to explicitly manage the allocation of cache storage for multithreaded workloads. This work introduced the marginal gains counters which were used to determine the best cache partition for the coming sampling interval. This work focused primarily on the storage capacity of each job and did explicitly address the issue of inter-thread interference.

Kim et al [14] examined the cache partitioning problem from the point of view of thread fairness. They based the partitioning mechanism on the LRU managed scheme used in [24] and improved it by balancing the partition decisions with a set of fairness metrics. The principle draw back with this technique however, is that it requires profile information from each of the jobs in order that their isolated MPI rates are known at runtime. Additionally, this work focused on a system with only 2 threads.

The impact of inter-thread interferences between workloads of 2 jobs was studied by Chandra et al in [5]. In this work, they focused on predicting the number of cache evictions that would be introduced to a given thread if it shared the second level cache with another job running on a CMT processor. While this work reveals important

insight into the nature of interferences between jobs it does so only for 2 jobs at a time and it does not specifically look at hardware techniques to exploit this information.

Other researches have since used column caching for different problems. Carzola et al [4] used it for partitioning the L2 cache in order to solve a QoS problem in SMT processors. Here the goal was to partition the cache so that one high priority application would receive very little interference from the other jobs in the system. While this technique does implore column caching, the goal of the work is not system performance and therefor requires a fundamentally different partition controller. Garcia et al examined the impact of cache sharing on multithreaded systems from the point of view of the data cache [10]. They examined static partitioning models but concluded that dynamic cache partitioning was needed to significantly improve the performance of caches for multithreaded processors. While there has been recent studies that propose various schemes for cache partitioning, the literature still lacks a body of work that explicitly addresses interferences between threads in a system of more than 2 threads, and applies a dynamic hardware solution that does not rely on prior knowledge of each job's execution characteristics.

7 Conclusion

This study has shown that intelligent management of cache systems in multi core processors can considerably reduce the interference between jobs competing for limited cache resources. This in turn improves the system performance by reducing the overall memory latency experienced by the workload. Given that the performance of CMP processors depends heavily on the set of running jobs, it is important that they can adapt to the different workload characteristics. By providing a dynamic feedback directed cache optimization, this work demonstrates that efficient management of existing cache resources can lead to significant improvements in the execution time of the workload. Specifically, speedups of up to 1.46X are achieved with a fully dynamic

hardware based cache controller, and up to 1.22X with a thread sensitive LRU allocation algorithm. By making more efficient use of the existing cache storage, this technique puts less pressure on processor designers to increase the cache size with each new processor generation. Since embedded computing platforms typically run workloads that have more software contexts than hardware contexts, this work will be extended to include cooperation between the operating system and the dynamically adaptable cache.

Acknowledgements

This work was made possible by a Fulbright Grant for the 2004-2005 academic year. Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research, and a grant from the IBM Shared University Research (SUR) program.

References

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *International Symposium on Microarchitecture*, pages 248–, 1999.

[3] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[4] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in smt processors. *Computing Frontiers*, pages 433–443, April 2004.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 340–351. IEEE Computer Society, 2005.

[6] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Design Automation Conference*, pages 416–419, 2000.

[7] I. Corporation. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*. IBM, 1999.

[8] I. Corporation. Special issue on intel hyperthreading in pentium-4 processors. *Intel Technology Journal*, 1(1), January 2002.

[9] A. Falcón, A. Ramírez, and M. Valero. A low-complexity, high-performance fetch unit for simultaneous multithreading processors. In *HPCA*, pages 244–253, 2004.

[10] M. Garcia, J. Gonzalez, and A. Gonzalez. Data caches for multithreaded processors, January 2000.

[11] S. G.E.1, R. L., and D. S. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, April 2004.

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Francisco, California, 1996.

[13] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*, May 2004.

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

[15] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[16] S. McGinnis and T. Madden. Blast: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Res.*, 32:Web Server Issue, July 2004.

[17] C. Nieter and J. R. Cary. Vorpal: a versatile plasma simulation code. *Journal of Computational Physics.*, 196(2):448–473, 2004.

[18] B. Pellom. Sonic: The university of colorado continuous speech recognizer. Technical Report TR-CSLR-2001-01, University of Colorado, Boulder, Colorado, March 2001.

[19] A. Settle, J. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced smt job scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[20] A. Shilov. Intels montecito to offer breakthrough performance, July 2005. http://www.xbitlabs.com/news/cpu/display/20050708133307.html.

[21] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[22] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro 22(4)*, pages 72–82, 2002.

[23] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[24] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, pages 117–, 2002.

[25] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual International Symposium on Computer Architecture*, pages 191–202, 1996.

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

[27] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 54–58, 2000.

[28] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *MICRO*, pages 183–194, 2004.

Tables

Module	Size	Latency
SMT Contexts	2	-
Issue Width	8	-
Branch Predictor	-	2-level gshare
Fetch Policy	icount	-
Data Cache	16KB 4wy	1 cycle hit 12 cycle miss
Instruction Cache	16 KB 4wy	12 cycle miss
L2 Cache	512 8wy	12 cycle hit 300 cycle miss
Register File	64 I/F	-
Sampling Interval	-	100,000 cycles

Table 1:

.

Table Titles

Microarchitecture Configuration

Figure Captions

Algorithm 1: Partition Controller Heuristic

Figure 1: Gzip_Mcf Allocation

Figure 3: Gzip_Mcf Desired Allocation

Figure 2: Working Set Sizes

Figure 4: Inter-thread Interference Graph

Figure 5: Partitioned Cache

Figure 6: L2 Cache Allocation Model Speedup

Figure 7: L2 Cache Fairness

Figure 8: L2 Cache Hit Count

Figure 9: CMP L2 Cache Hit Count

Figures

Algorithm 1

for all Threads **do**
  $score$=$ave_reuse * 1 - pct_lines$
end for
$target_thread$ = Thread with highest $score$
for all Ways not owned by pri_thread **do**
  $gain$ = $target_th_per_way_access$ - way_access_cnt
  $bank_inc_score$ = $gain$ - $loss$
  max_inc_score = MAX $bank_inc_score$
end for
for all Ways owned by Thread **do**
  $loss$ = lru_hits
  $gain$ = $th_per_way_access$ + $per_way_inter_th_miss$
  $bank_dec_score$ = $gain$ - $loss$
  max_dec_score = MAX $bank_dec_score$
end for
if $max_dec_score > max_inc_score$ **then**
  Remove thread from highest scoring way
else
  Add test thread to highest scoring way
end if

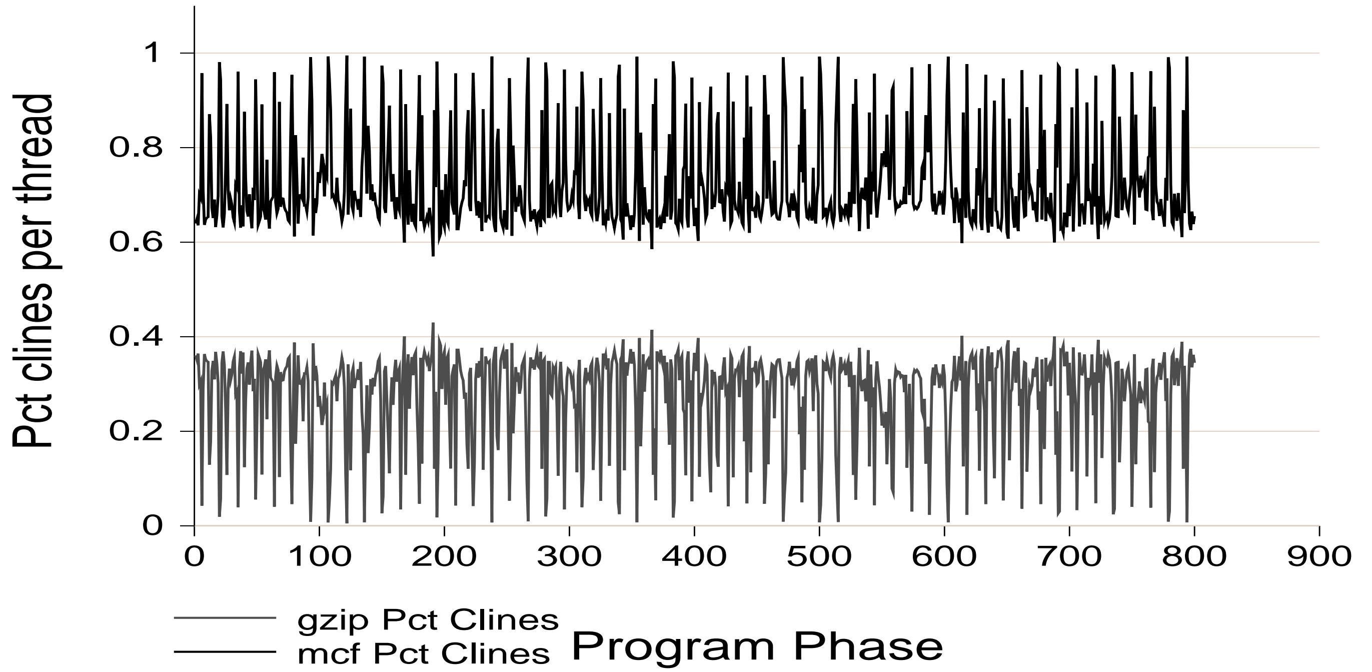

Figure 1:

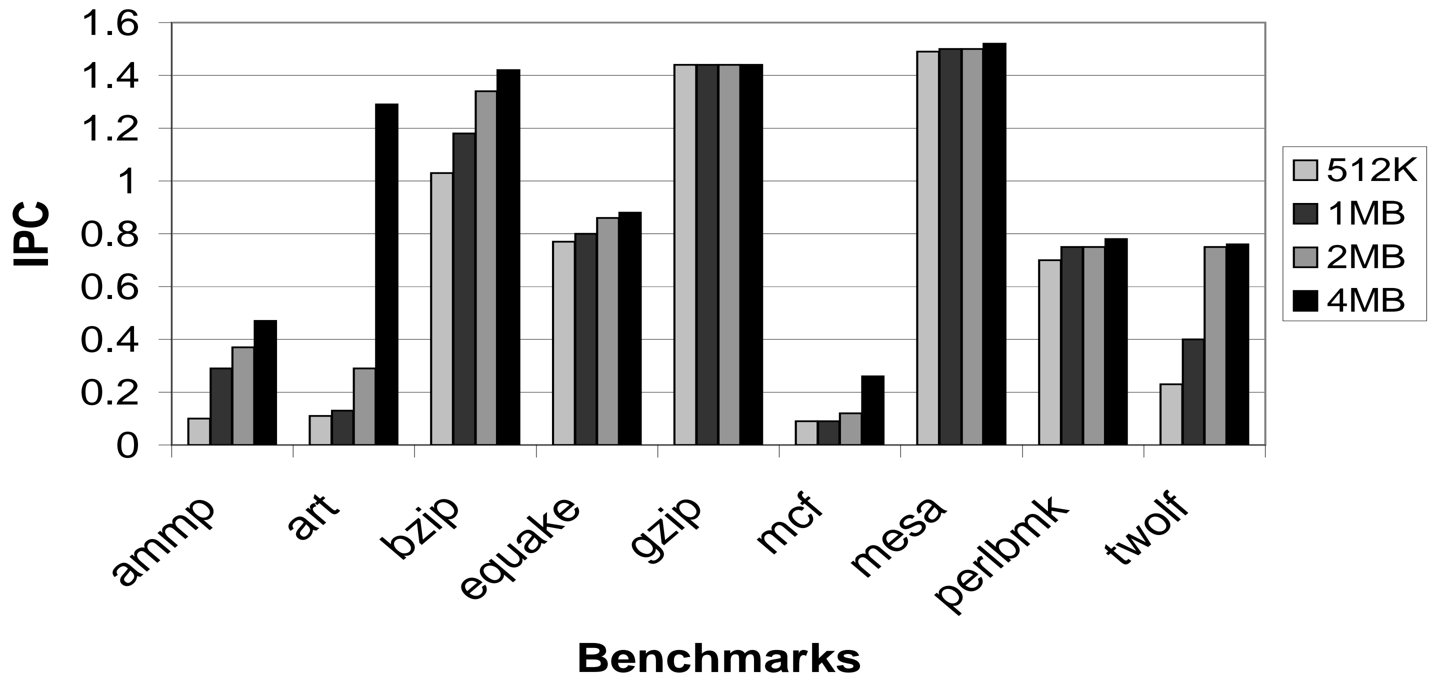

Figure 2:

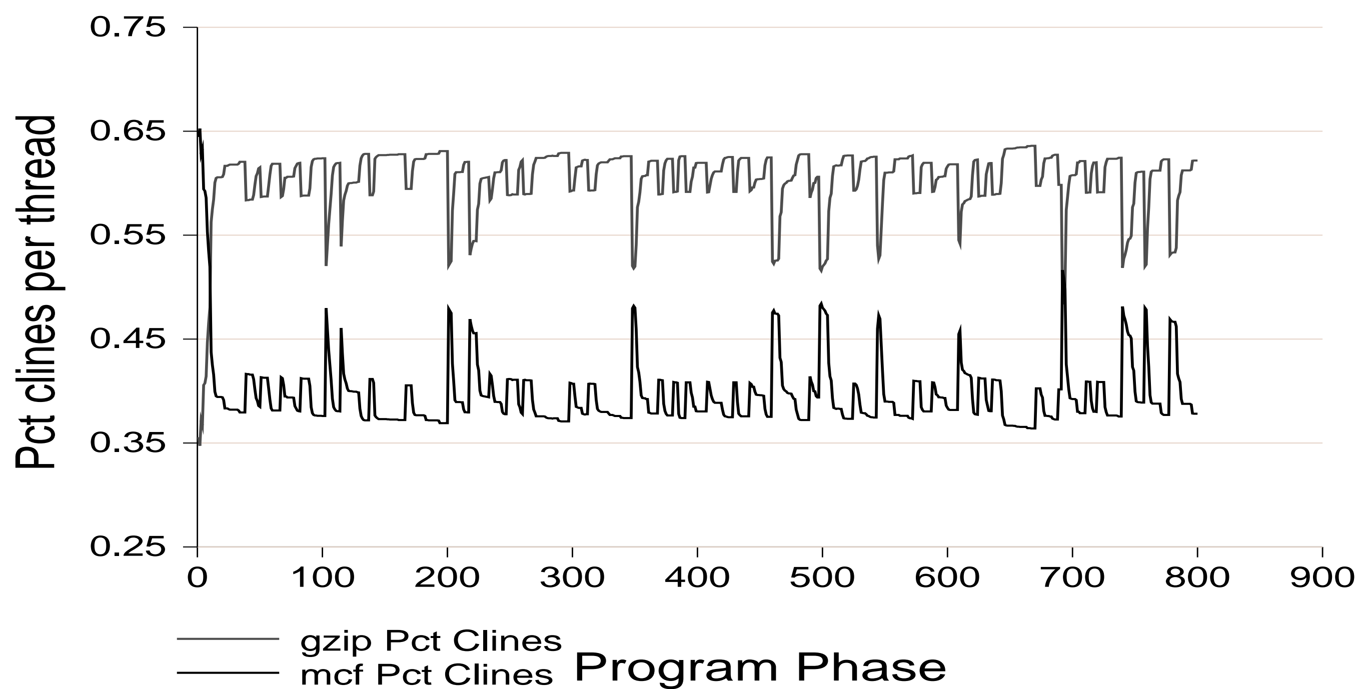

Figure 3:

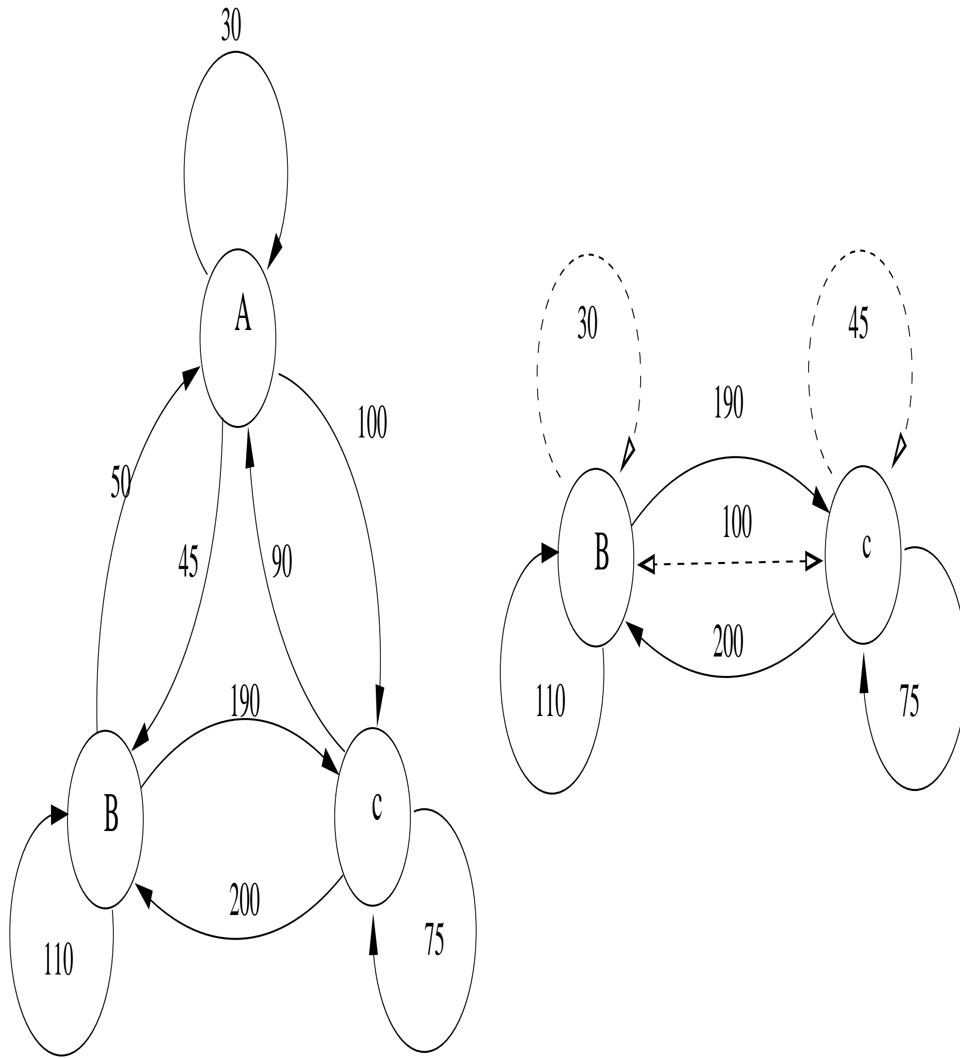

Figure 4:

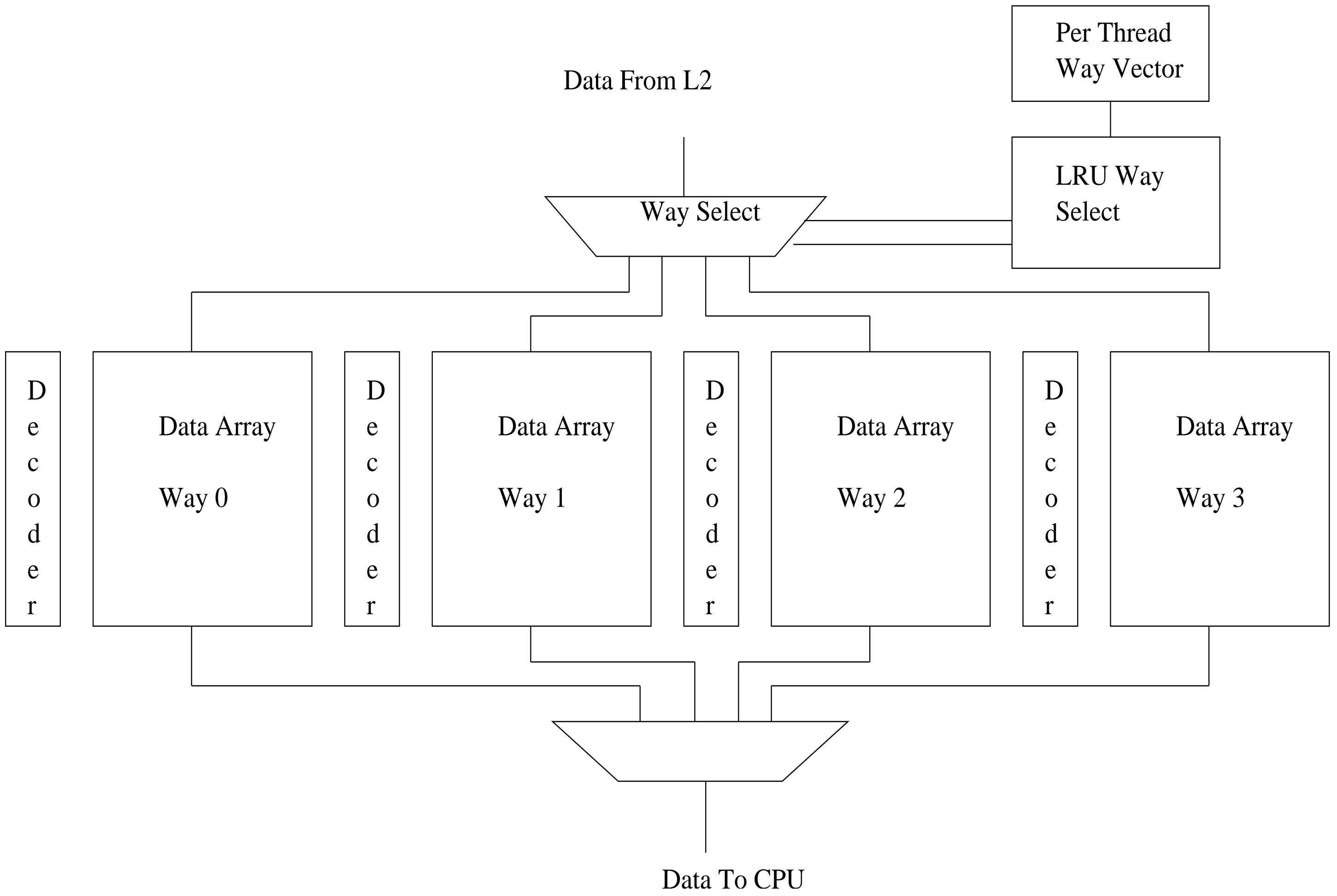

Figure 5:

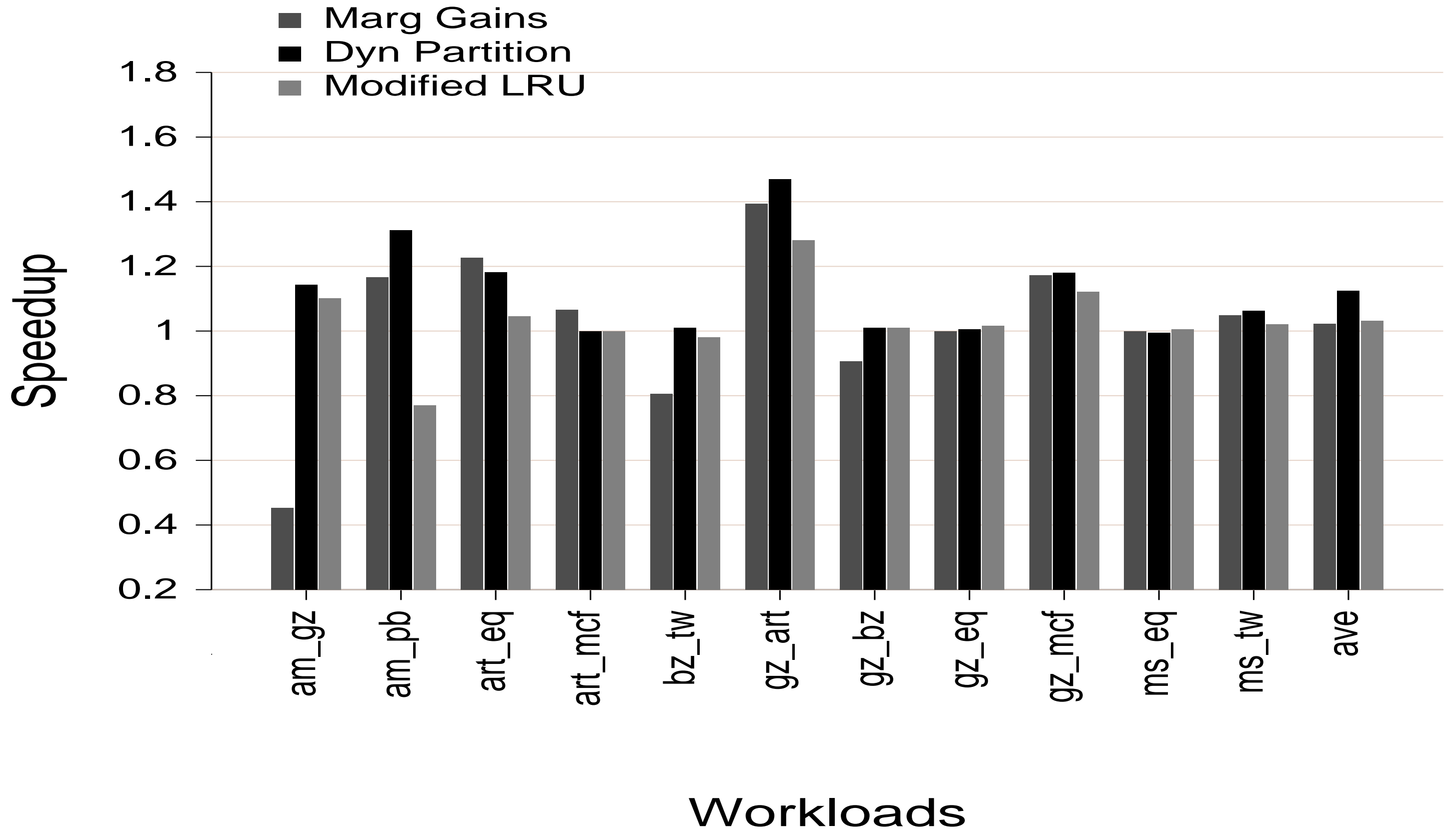

Figure 6:

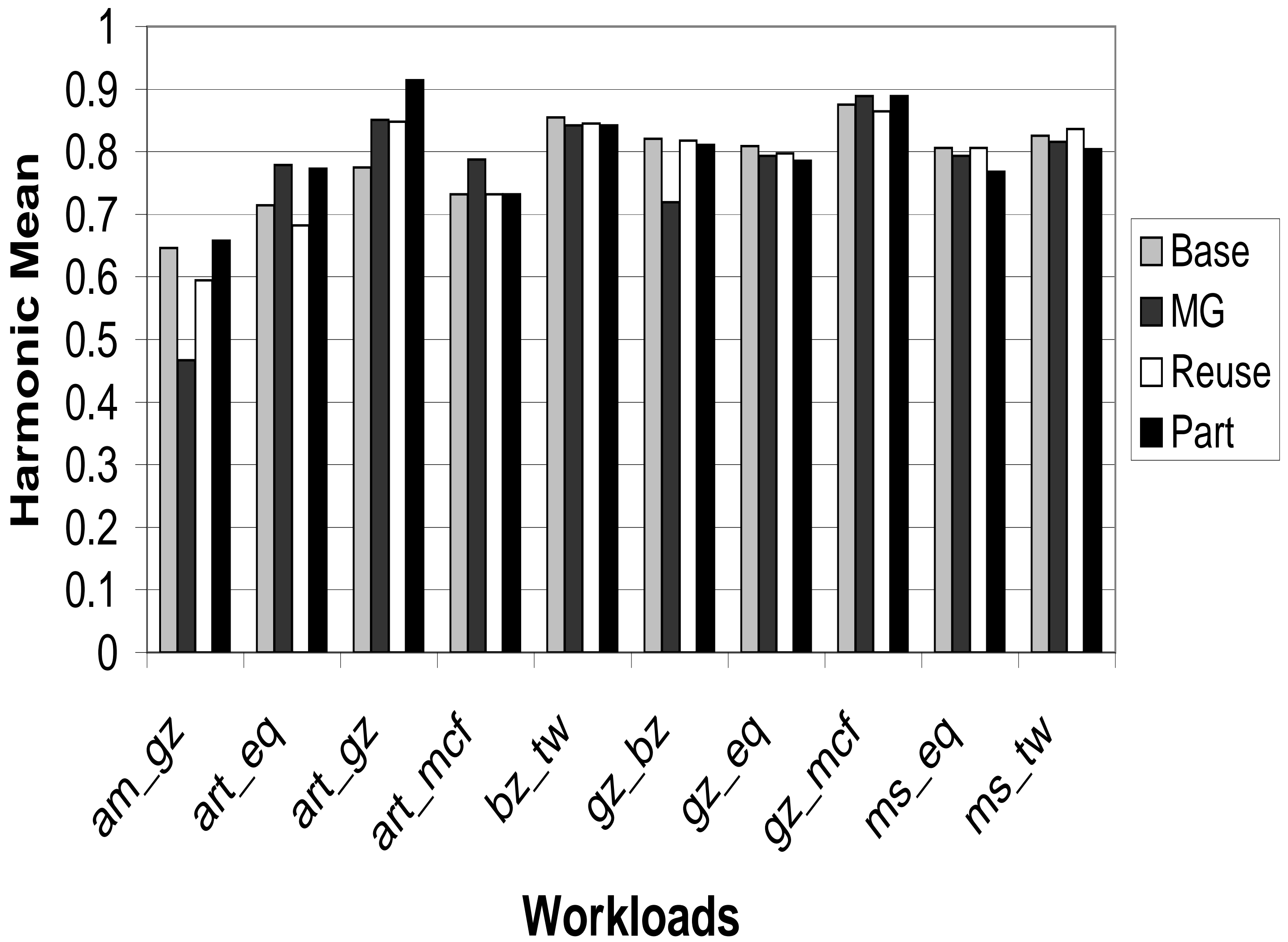

Figure 7:

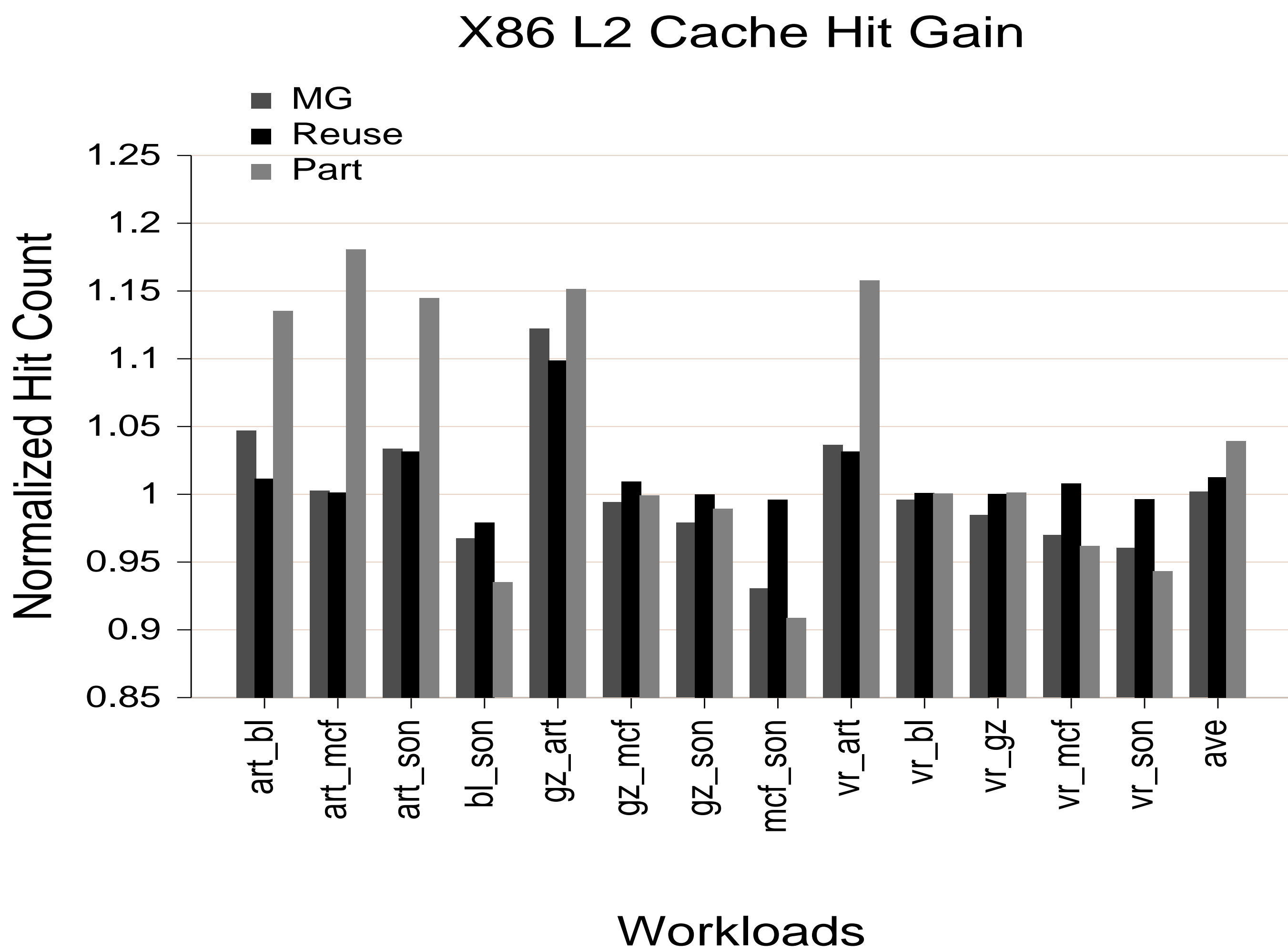

Figure 8:

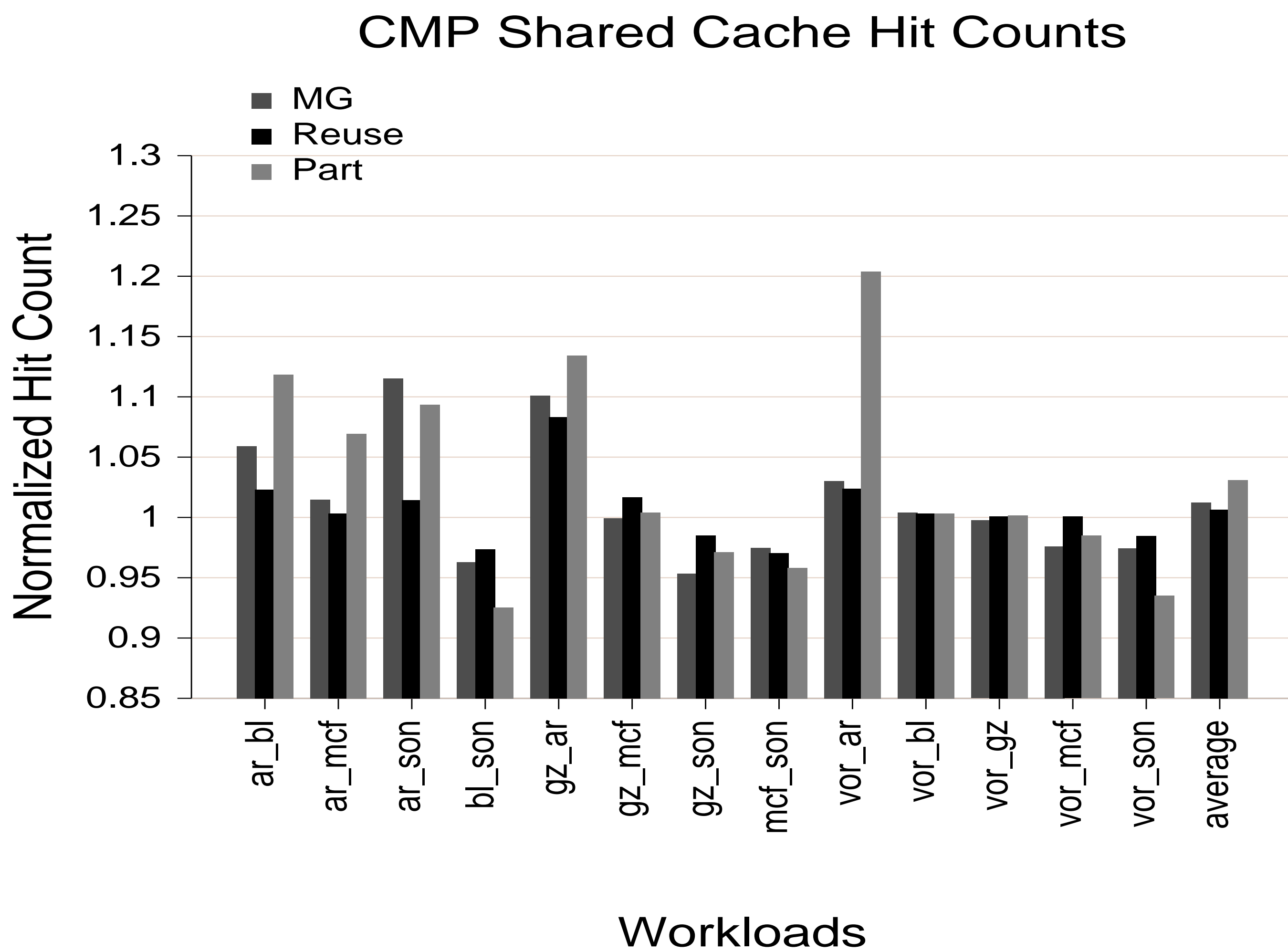

Figure 9: