NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-08-02
April 26, 2008

Chip Multiprocessor Cooperative Cache Compression and Migration

Lei Yang[†], Xi Chen[†], Robert P. Dick[†], Li Shang[‡], and Haris Lekastas[§]
[†]l-yang, dickrp, xi-chen-0@northwestern.edu, [‡]li.shang@colorado.edu, [§]lekatsas@vorras.com

Abstract

CMPs are now in common use. Increasing core counts implies increasing demands for instruction and data. However, processor-memory bus throughput has not kept pace. As a result, cache misses are becoming increasingly expensive. Yet the performance benefits of using more cores limits the cache per core, and cache per process, increasing the importance of efficiently using cache.

To alleviate the increasing scarcity of on-chip cache, we propose a cooperative and adaptive data compression and migration technique and validate it with detailed hardware synthesis and full-system simulation. This is the first work that completes optimized microarchitectural design and synthesis of the hardware to support adaptive cache compression and migration. Full-system simulation on multi-programmed and multithreaded workloads indicate that, for cache-sensitive applications, the maximum CMP throughput improvement with the proposed technique ranges from 4.7%–160% (on average 34.3%), relative to a conventional private L2 cache architecture. No performance penalty is imposed for cache-insensitive applications. Furthermore, we demonstrate that our cooperative techniques may influence the optimal core cache area ratio for maximum CMP throughput, and has the potential to reduce on-chip cache requirement, and thereby increase the number of cores.

This work was supported in part by AFOSR award FA9550-06-1-0152 and in part by NSF awards ITR-CCR-0325207 and CNS-0347941.

Keywords: CMP, memory hierarchy, compression, migration

Chip Multiprocessor Cooperative Cache Compression and Migration

Abstract

CMPs are now in common use. Increasing core counts implies increasing demands for instruction and data. However, processor-memory bus throughput has not kept pace. As a result, cache misses are becoming increasingly expensive. Yet the performance benefits of using more cores limits the cache per core, and cache per process, increasing the importance of efficiently using cache.

To alleviate the increasing scarcity of on-chip cache, we propose a cooperative and adaptive data compression and migration technique and validate it with detailed hardware synthesis and full-system simulation. This is the first work that completes optimized microarchitectural design and synthesis of the hardware to support adaptive cache compression and migration. Full-system simulation on multi-programmed and multithreaded workloads indicate that, for cache-sensitive applications, the maximum CMP throughput improvement with the proposed technique ranges from 4.7%–160% (on average 34.3%), relative to a conventional private L2 cache architecture. No performance penalty is imposed for cache-insensitive applications. Furthermore, we demonstrate that our cooperative techniques may influence the optimal core cache area ratio for maximum CMP throughput, and has the potential to reduce on-chip cache requirement, and thereby increase the number of cores.

1. INTRODUCTION AND MOTIVATION

Increases in VLSI integration density and the increasing importance of power consumption are leading to the use of chip-level multiprocessor (CMP) architectures. The move to CMPs substantially increases capacity pressure on the on-chip memory hierarchy. The 2006 ITRS Roadmap [1] predicts that transistor speed will continue to grow faster than DRAM speed and pin bandwidth. This means that cache miss costs will continue to increase. However, increases in on-chip cache size may block the addition of processor cores, thereby resulting in suboptimal CMP throughput. Our full-system simulation results indicate that under area constraints, the allocation of core-cache area that achieves maximum CMP throughput use less cache per core than is common for existing processors. However, such allocation increases capacity misses, and thereby reduces the performance of individual cores. As a result, the marginal improvement in performance as a result of increasing usable cache size increases. In other words, *the move to CMPs increases the importance of carefully using available cache area*.

In this paper, we address the problem of optimizing the use of on-chip cache to reduce off-chip memory access, and thereby improve CMP throughput. We develop and evaluate our techniques for private on-chip L2 cache hierarchy, because in contrast to a shared L2 cache, the performance and design styles of private L2 caches remain consistent and predictable when the number of processor cores increases. The techniques proposed in this paper address the weakness of private caches by increasing their capacity without significantly increasing their access latency.

When data are evicted from a private L2 cache, they are generally discarded (if not modified) or transferred to the next level of the memory hierarchy, i.e., off-chip memory. Writing to off-chip memory can cause contention with other accesses to the processor-memory bus and result in significant latency when later reading back the same data for reuse. In CMPs, lower-overhead alternatives should be considered, such as compressing data and storing them locally, migrating data to the private L2 caches of other processor cores, or a combination of these techniques. One can view compression and migration as introducing a new, virtual layer in the memory hierarchy. This new layer permits an increase in usable L2 cache without increasing physical L2 cache size, at a cost of higher access latency.

We propose a method of alleviating the increasing scarcity of on-chip memory via cooperative and adaptive cache compression and migration. Both of these techniques introduce a number of design problems, and their cooperative use compounds these problems. We propose a unifying adaptive policy that considers the time-varying merits of the workloads. Defining this compression–migration policy and analyzing its hardware implications are the main foci of our work.

2. Contributions

This work is the first to cooperatively use cache compression and migration for CMPs. We propose an adaptive control policy integrating the two techniques and permitting run-time adaptation to workload. This control policy is based on a new optimization metric: processor marginal performance gain. Compared to static techniques, our adaptive techniques allow on average 42.3% greater performance improvement for cache-sensitive applications and prevent performance degradation for cache-insensitive applications. We heavily modified the *MSI_MOSI_CMP_directory* cache coherence protocol provided by the GEMS infrastructure [2] to support cache compression and migration.

This work is the fist to present optimized microarchitectural design and synthesis results for the hardware required for control, compression/decompression, and migration. We propose a new scheme to organize compressed cache lines, namely pair-matching, that is more efficient than commonly accepted segmentation based schemes. We also present the first hardware implementation of the PBPM [3] compression/decompression algorithm. Compared to prevailing cache compression/decompression algorithms, our implementation provides the best overall performance, in terms of compression ratio, maximum frequency, decompression latency, power consumption, and area cost.

The proposed techniques are evaluated using full-system (application and OS) simulation of numerous multiprogrammed SPEC CPU2000 benchmarks and Data-Intensive Systems (DIS) stressmarks [4] and multithreaded SPEC OMP and Nasa Parallel Benchmarks. We found that they can improve the throughput of CMPs by up to 160% for cache-sensitive applications. Consequently, given a fixed core count, our techniques can help reduce chip area by allowing cache size reduction. Alternatively, given a fixed area, our techniques allow more processor cores to be used while maintaining good performance for each core.

The rest of the paper is organized as follows. Section 3 describes the proposed cooperative cache compression and migration techniques. Section 4 presents our implementation and evaluation of necessary hardware to support cache compression and migration. Section 5 describes our simulation environment and workloads, and presents and analyzes the simulation results. Section 6 discusses related work. We conclude in Section 7.

3. Cooperative Cache Compression and Migration

This section describes the proposed cooperative cache compression and migration techniques for optimizing on-chip cache utilization. We first describe the problem definition for optimizing on-chip cache utilization, then give an overview of the proposed architecture. Finally, we present the details of our technique to control adaptive compression and migration.

3.A. Optimizing On-Chip Cache Utilization

Our objective is to improve the overall throughput of CMPs by optimizing the utilization of on-chip cache resources. To that end, we consider two methods: (1) compressing data stored in its local cache or (2) making use of the caches of other cores.

Designing the architectural extensions required to support either compression or migration is challenging, and using the two techniques cooperatively complicates the problem. It is important to select the right lines to compress/migrate, determine the right moment to compress/migrate, and decide the locations of cache lines after compression/migration. In addition, both compression and migration involve overhead and should only be used when the CMP throughput can be improved. Furthermore, migration increases on-chip network traffic and may result in contention if used improperly: only "useful" data should be migrated. Therefore, the overall technique must be able to determine the time-varying cache requirement of the workload during execution and must adaptively control compression and migration.

TABLE I
Comparison of eviction, compression, and migration

Scheme	$\Delta cycles$
Eviction	$P \cdot Miss\_Penalty_{L2}$
Compression	$(P + P') \cdot Decompress\_Penalty$
Migration	$P \cdot Remote\_Latency_{L2} + P'' \cdot Miss\_Penalty_{L2}$

To further understand the costs associated with compression and migration, Table I compares the first-order effect on total CPU cycle count when a L2 cache line is (1) evicted off-chip, (2) compressed locally, and (3) migrated to a remote L2 cache. For compression, in order to have the same effect as eviction, two lines must be compressed and placed in one cache line. We call that compressed line the *compression victim line*. For migration, in order to accept the migrated block, a local cache block must be evicted, which we call the *migration victim line*. P, P', and P'' represent the probability of the cache line, the compression victim line, and the migration victim line being accessed again, respectively. To simplify illustration, we assume $P' \approx P''$. Therefore, as long as $Decompress\_Penalty < Remote\_Latency_{L2}$, the penalty of compression is always smaller than migration. In modern CMP cache hierarchies, the $Remote\_Latency_{L2}$ is usually greater than 15 cycles, while the $Decompress\_Penalty$ of our proposed hardware is only 8 cycles. Therefore we conclude that it is always more beneficial to compress than to migrate.

3.B. Overview of Proposed Solution

We propose a unified solution that uses compression and migration to cooperatively and adaptively manage on-chip cache resource in CMPs. In our technique, when the running process can benefit from a larger cache size, the least recently used (LRU) lines in local L2 cache are compressed. Depending on the sensitivity of the application to cache size, the number of cache lines being compressed may continue to grow until all lines are compressed. After all the lines in the local cache are compressed, if the running process would benefit from a further increased cache size more than the processes on other cores, then the least recently used compressed lines are migrated to remote on-chip caches. This allows the aggregate cache resources to adapt to the dynamic demands of different applications. Migrated lines are transferred in compressed format and stay compressed once they arrive at their targets. Based on the analysis in Section 3.A, local compression is used as a first defense to cache shortage and migration is used as a last resort before evicting data off chip. We evaluate the marginal performance gain of an application to decide its sensitivity to cache size, and organize compressed lines using our pair-matching scheme.

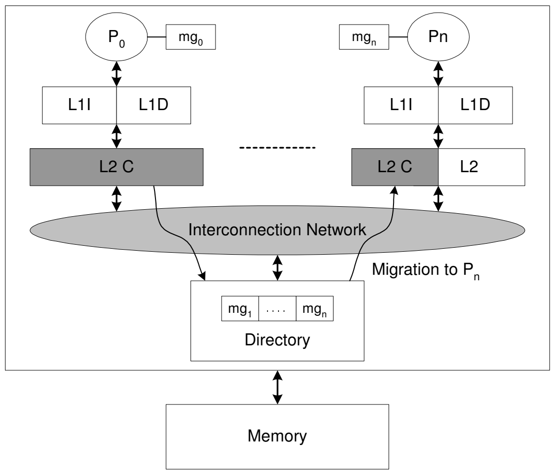
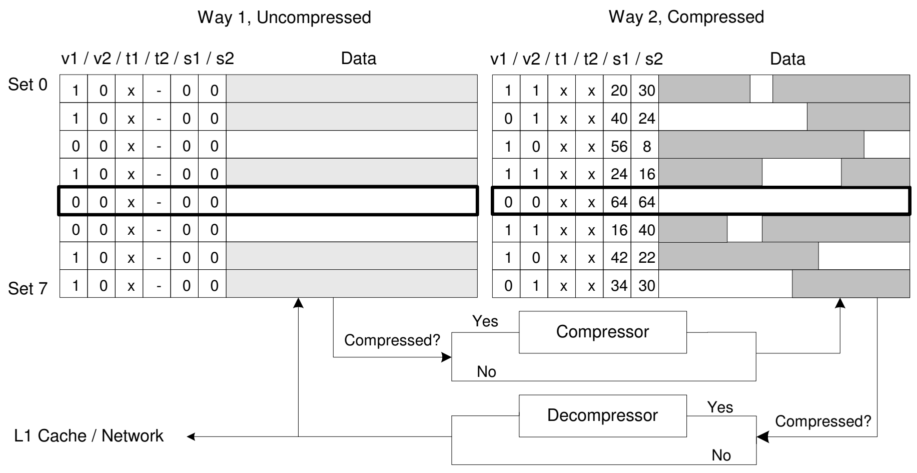


Figure 1. Technique overview.

Figure 1 gives an overview of the proposed techniques. Each processor has private L1 and L2 caches. For processor P_0, the whole L2 cache is compressed; for processor P_n, half of the L2 cache is compressed (L2C in the figure), which results in a four-level cache hierarchy: L1, uncompressed L2, compressed L2, and remote L2. Note that uncompressed L2 and compressed L2 share the same resources but have different access latencies. After being accessed, data are placed in L1 cache and are demoted through the levels of the memory hierarchy until dropped or sent to off-chip memory.

3.C. Effective System-Wide Compression Ratio and Pair-Matching Compressed Cache Organization

This section describes the organization of non-uniform sized compressed lines. Organizing a compressed cache is challenging because the compression ratios of different cache lines may vary dramatically. In the past, researchers have proposed segmentation techniques [5, 6] to handle this problem. The main idea is to divide the compressed lines into fixed-size segments, e.g., 8 bytes, and use indirect indexing to locate all segments for a compressed line. This approach has significant overhead due to the latency and hardware cost required to address all segments. Consequently, the



v1	v2	t1	t2	s1	s2
1	0	x	-	0	0
1	0	x	-	0	0
0	0	x	-	0	0
1	0	x	-	0	0
0	0	x	-	0	0
0	0	x	-	0	0
1	0	x	-	0	0
1	0	x	-	0	0

v1	v2	t1	t2	s1	s2
1	1	x	x	20	30
0	1	x	x	40	24
1	0	x	x	56	8
1	1	x	x	24	16
0	0	x	x	64	64
1	1	x	x	16	40
1	0	x	x	42	22
0	1	x	x	34	30

Figure 2. Compressed cache organization.

size segments, e.g., 8 bytes, and use indirect indexing to locate all segments for a compressed line. This approach has significant overhead due to the latency and hardware cost required to address all segments. Consequently, the number of segments per line is tightly constrained, resulting in wasted space.

In contrast, we propose a *pair-matching* scheme for the compressed line organization, i.e., each line of the original length is used to store two lines after compression. In a pair-matching based cache, the location of a newly compressed line depends on not only its own compression ratio but also the compression ratio of its “partner”. More specifically, the compressed line first tries to locate the partner line with sufficient unused space for the compressed line without replacing any existing compressed lines. The candidates for partner line are the other cache lines in the same cache set, e.g., seven candidates if the set-associativity is eight. If no such line exists, one or two compressed lines are evicted to store the new line. Note that successful placement of a line does not require that it has a compression ratio smaller than 50%, but requires that the sum of the compression ratios of the line and its partner is less than 100%. This scheme greatly simplifies designing hardware to manage the locations of compressed lines compared to allowing arbitrary positions in segmentation-based techniques.

For both segmentation based and pair-matching based techniques, their effect on increasing cache capacity depends on the number of compressed lines actually stored, which has a complex relationship between the compression ratios of individual lines. We propose using *system-wide compression ratio* to evaluate such effectiveness. For pair matching, a newly compressed line has an effective compression ratio of 100% when it takes up a whole cache line, and an effective compression ratio of 50% when it is placed with its partner within one cache line. For line segmentation, if a cache line is divided into 4 fixed-length segments, a compressed line has an effective compression ratio of 25% when it takes up one segment, 50% for two segments, and so on. Varying raw compression ratio between 25% and 50% has little impact on the effective cache capacity. We collected real cache trace data from full-system simulation, and evaluated the effective system-wide compression ratio for both schemes. Pair matching generally achieves a better effective system-wide compression ratio (58%) than line segmentation with four segments per line (62%) and the same compression ratio as line segmentation with eight segments, which would impose substantial hardware overhead for indirect indexing.

The overhead of pair-matching is low relative to the effective cache size improvement resulting from compression. Since any cache line may be used to store two compressed lines, each line has two *valid* bits and *tag* fields to indicate status and indexing. When compressed, two lines share a common *data* field. There are two additional *size* fields to indicate the compressed sizes of the two lines. Whether a line is compressed or not is indicated by its *size* field. As shown in Figure 2 a *size* of zero is used to indicate uncompressed lines. For compressed lines, *size* is set to the line size for an empty line, and the actual compressed size for a valid line. For a 64-byte line in a 32-bit architecture the tag is no longer than 32 bits, hence the worst-case overhead is less than 32 (tag) + 1 (valid) + 2 $\times$ 7 (size) bits, i.e., 6 bytes.

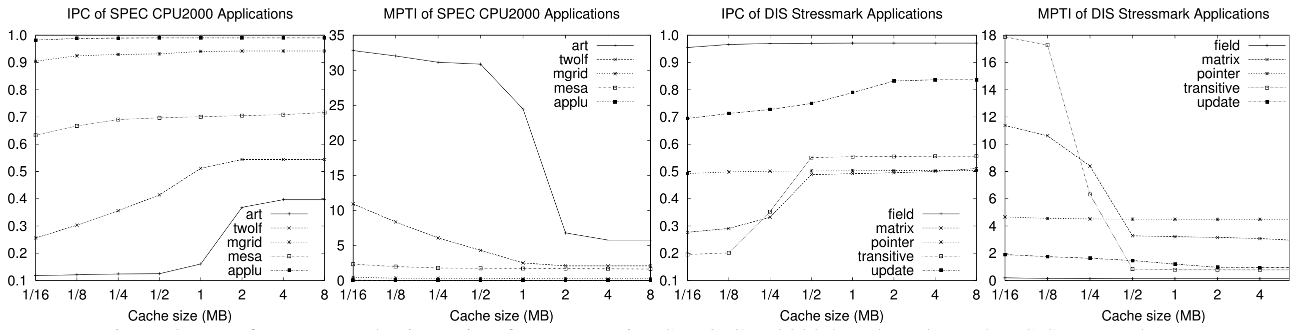


Figure 3. Performance and miss ratio of representative SPEC CPU2000 benchmarks and DIS Stressmarks.

3.D. Marginal Performance Gain

As previously mentioned, compression and migration techniques should be adaptive because (1) the cache requirements of different processors vary and the total on-chip cache resource should be allocated to optimize overall performance, and (2) applications that can fit in the original cache can be hurt by the decompression overhead and the migration overhead. This indicates that these techniques should only be used when the running process may benefit from a larger cache size. In this section, we define precisely what we mean by "running process may benefit from a larger cache size".

We use *marginal performance gain* as a measure of the usefulness of cache space for an application, or more specifically, the reduction in cycles per instruction (CPI) as a result of an incremental change to cache size. The benefit of an increase in the effective cache size depends largely upon the current running application and the current cache size. Figure 3 illustrates the impact of L2 cache size on performance as well as miss rate for ten representative applications from the SPEC CPU2000 benchmark suite and the Data-Intensive System (DIS) stressmark suite [4].

Within a time period T, the marginal performance gain at a given cache size represents the number of cycles saved executing the same number of instructions, if one more cache block were available. Thus, it indicates the benefit of increasing cache size from c to $c+1$ for an application, or approximately the loss from decreasing cache size from c to $c-1$ (considering that the marginal performance gain curve is smooth). Whenever the least recently used (LRU) line of processor i is accessed, if the number of cache lines of this processor is reduced by one, this access will be a miss. Thus, the number of cycles necessary to finish this instruction will be increased by the miss penalty, MP_i. Therefore, the performance gain of process i can be estimated as:

$$g_i(c) = \frac{d}{dc}CPI_i(c) = \frac{CycleCount_i(c-1) - CycleCount_i(c)}{InstructionCount_i(c)} = \frac{LRURefCount_i(c) \cdot MP_i}{InstructionCount_i(c)} \qquad (1)$$

In reality, the MP_i for a certain process at a certain cache allocation is non-uniform; it depends on the location of the required data. If the required line is in off-chip memory, MP_i is the memory access latency; if it is in one of the on-chip caches, MP_i is the communication latency between two processor nodes. The communication latencies between every two nodes may also differ depending on their physical locations. We use the same MP_i for all processes in order to reduce hardware implementation complexity. We therefore define the marginal performance gain of process i as:

$$MPG_i(c) = \frac{LRURefCount_i(c)}{InstructionCount_i(c)} \tag{2}$$

3.E. Adaptive Compression and Migration

We use marginal performance gain to compress and migrate lines only when the running application may benefit from a larger cache size. The procedure of adaptive compression and migration operates as follows. Three marginal performance gain thresholds are used to control adaption: MPG_{T1}, MPG_{T2}, and MPG_{T3}. In Figure 2, we give a simple example of 2-way associative cache. A cache with higher associativity would be organized into two multi-way hierarchies.

1. When the marginal performance gain, mpg, of the running application is lower than the first threshold MPG_{T1}, neither cache way is compressed. However, a newly-loaded line is always placed in way-1 instead of way-2. Evicted lines from way-1 are placed in way-2 instead of off-chip memory. Evicted lines from way-2 are evicted off-chip. One can view way-2 as another level of cache hierarchy with the same access time as a regular L2 cache.

2. When mpg exceeds the first threshold MPG_{T1}, the compression of way-2 starts. Note that way-2 stores older data than way-1. Whenever a new line is sent to way-2, it is compressed, replacing a stale uncompressed line. The additional space left in the data block will later be used to store another compressed line. At this point, way-2 is split into two compressed ways. When mpg further exceeds the second threshold MPG_{T2}, way-1 is also compressed, following the same procedure.

3. When mpg decreases below MPG_{T2}, the decompression of way-1 starts. Whenever any compressed line is accessed, it is decompressed, evicting its partner compressed line, if presents. When mpg falls below MPG_{T1}, decompression of way-2 begins, following the same procedure.

4. When a compressed line in way-2 is to be evicted, if the current mpg exceeds MPG_{T3}, depending on the cache coherence protocol, data may be sent to the directory node. The directory node decides whether to drop the line, migrate it to a remote on-chip cache, or send it to off-chip memory.

In practice, MPG_{T1}, MPG_{T2}, and MPG_{T3} can be decided via experiments on typical applications. Fixed empirical values can then be used for all applications. Our experimental results validate this approach[1]. We now discuss several important issues for compression and migration.

- Whenever a line is compressed, the technique first tries to find the best-fit compressed partner that has space available in the data block. If that fails, the LRU line in the compressed ways is evicted to accommodate this line. If necessary, the partner line of the LRU line is also evicted. Note that no processor stalls or additional buffers are necessary to handle compression requests. Compression does not block the issue of later instructions because data awaiting compression are placed in the input buffer of the hardware compressor. Furthermore, since compression

[1]The OS might potentially dynamically adjust threshold values, but fixed, empirically-determined values were sufficient for good results.

is only triggered by L2 cache misses, another compression request will not be generated before the first missed line is sent to the processor, either from off-chip memory or from a remote L2 cache. In both cases the latency is longer than compression plus arbitrating the location of a compressed line.

- Whenever a compressed line is accessed, it is decompressed and sent to the L1 cache or the on-chip network and then forwarded to the requesting processor. The line is also sent back to the uncompressed ways of the same set in the L2 cache. If there is no available space, the LRU line in the uncompressed ways is sent to the compressed ways. Note that decompression is on the performance critical path and therefore its latency must be minimized.

- The LRU compressed lines in local L2 cache may be migrated to remote caches to remain on-chip. Cache migration can improve the performance of the local processor, but may also hurt the performance of the remote processor. Therefore, migration requests must be sent to the right target, i.e., a processor where the running application has smaller marginal performance gain at its current cache size. Our goal is to minimize the total CPI for simultaneous processes:

$$\sum_{i=1}^{N} CPI_i(c_i) = \sum_{i=1}^{N} \frac{CycleCount_i(c_i)}{InstructionCount_i(c_i)} \tag{3}$$

In this equation, $\sum_{i=1}^{N} c_i = C$, where C is the total number of cache lines in all on-chip L2 caches. To minimize the total CPI, the net benefit of incrementing the number of cache lines for one processor and the loss of decrementing the number of cache lines for another processor must be positive. Therefore, migration should only be allowed from processors with higher marginal performance gains to processors with lower marginal performance gains.

4. Hardware Support for Control, Compression, and Migration

In this section, we present the necessary hardware support to implement marginal performance gain, the adaptive control logic, and compression/decompression algorithm.

4.A. Hardware Requirement for Marginal Performance Gain

We now describe the necessary hardware assistance to obtain runtime marginal performance gain. Over a time period T, an approximation of $MPG(c)$ can be obtained using two counters. The first counts the references to the LRU line and the second counts the number of instructions executed. For set-associative caches, LRU ordering is kept within each set. Therefore, we use set LRU to approximate global LRU, similar to Suh et al. [7]. Efficient hardware implementations of approximated LRU for cache blocks are described in the literature [8]. When an LRU line in any set is accessed, the LRU counter is incremented. During each time period T, each processor calculates its marginal performance gain $MPG(c)$ by dividing the value of the two counters, and stores the result in a special register. The counters are then reset.

The marginal gain values of processors are also updated in the on-chip directory node. When the directory node receives a cache migration request from a processor, it selects the processor with lowest marginal performance gain as the target processor and forwards the migration request to it. If the marginal performance gain of the requestor is lower than those of the remaining processors, the migration request is declined and the line is sent to off-chip memory, if modified. Once a processor receives a migration request from the directory node, it must place the compressed line in its own cache, even if this forces the eviction of one of its own cache lines.

We now describe how to decide update period T. A small T increases the accuracy of marginal performance gain estimation but has higher computational overhead. A large T reduces the computational overhead but may cause inaccurate estimation due to application phase changes being missed. Moreover, in a multitasking OS, T must be smaller than the context switching period, which is typically within the range of 100 ms and 200 ms. When developing their shared cache partitioning technique, Suh et al. [7] also encountered the problem of selecting an appropriate repartitioning period. They experimented with different repartitioning periods and found that both 5 million cycles and 10 million cycles permitted accurate identification of program phase changes with low overhead. We therefore used an update period of 10 million cycles. Note that at a CPU frequency of 1 GHz, 10 million cycles (10 ms) is significantly shorter than the context switch period. In order to control the recalculation of marginal performance gain, an OS timer may be used to interrupt the processor and switch to kernel mode every time period T. The additional overhead of the context switch is usually less than 10 μs for modern processors, which is negligible compared to the recalculation period.

4.B. Hardware Implementation for Compression and Decompression

Most existing work on cache compression assumes, or uses high-level evaluation to argue that, it is possible to build control, compression, and decompression hardware with adequate compression ratio, performance, area, and power consumption. In fact it is not obviously possible to build efficient hardware with a sufficient compression ratio (approximately 50%). Detailed design is necessary to demonstrate feasibility, and may lead to changes in higher-level architectural design decisions.

We developed a high-performance hardware compressor and decompressor for cache compression, based on the PBPM algorithm [3], which was initially designed for on-line compression of data in main memory. Past work only describes and evaluates a software implementation of the algorithm. In this section, we describe how PBPM can be efficiently implemented in hardware and used for cache data. We also present the synthesis results and report latency, area, and power consumption estimates.

The PBPM algorithm is based on the observations that frequently-encountered data patterns can be encoded to save space and that there is similarity among words stored near each other in memory. Scanning through the input data a word (32 bits) at a time, PBPM exploits patterns within each word by comparing against predefined data patterns, and identifies similarities among words by searching for complete and partial matches with entries stored in a small look-up table, i.e., a dictionary. The inherent parallelism of PBPM, e.g., parallel dictionary matching and pattern matching, makes it a good candidate for hardware implementation.

TABLE II
DESIGN CHOICES FOR DIFFERENT PARAMETERS

Parameters	Candidates	Selected Candidate
Dictionary replacement policy	(1) First-in first out (FIFO) (2) Least recently used (LRU) (3) Using two FIFO queues to simulate LRU (4) FIFO combined with run-length encoding (RLE)	FIFO - least HW complexity with only 1.74% higher CR than best case
Coding scheme	(1) Huffman coding (2) Two/Three-level coding	Two-level coding due to only up to 0.5% increase in CR with best HW complexity
Dictionary size	Ranging from 16 B to 512 B	32 B - optimal CR for FIFO and low HW cost

4.B.1) Design Trade-offs and Decisions: There are several important trade-offs during the design and implementation of our hardware compression and decompression algorithm. We list them below.

- **Dictionary design and pattern coding**: To optimize compression ratio (CR), we evaluated the impact of different parameters of the compression/decompression algorithm, including dictionary replacement policy, dictionary size, and coding scheme. Our input data are cache traces collected from a full-system simulation running various workloads, e.g., media applications and SPEC CPU2000 benchmarks. An 8-way set associative cache with a 64 B line size is used. The candidates for different parameters and the final selected values are shown in Table II. Note that two/three level coding scheme in Table II refers to one in which the code length is fixed within the same level, but differs from level to level. For example, a two-level code can contain 2-bit and 4-bit codes only. With our selection of parameters, the average compression ratio for a 64 byte cache line is 52.3% on the test data.
- **Trade-off between compression ratio and hardware complexity**: In order to determine whether the mean and variance of the compression ratio achieved by our hardware implementation of PBPM is sufficient for most lines to find partners, we simulated a pair-matching based cache and computed the probability of two cache lines fitting within one uncompressed cache line, using our cache trace data. We evaluated the "best fit + best fit" policy: for a given compressed cache line, we first try to find the cache line with minimal but sufficient unused space. If the attempt fails, the compressed line replaces one or two compressed lines. With this scheme, we are penalized only when two lines are evicted to store the new line. Our simulation results indicate that the probability of fitting a compressed line into the cache without additional penalty is at least 99.45%. We implemented and synthesized this line replacement policy in hardware. The delay and area cost are reported in Table IV.
- **Trade-off between area and decompression latency**: Decompression latency is a critically-important metric. During decompression, the processor may stall, waiting for data. Therefore, a high decompression latency can undermine performance improvement. We use parallelism to increase the throughput of the compression hardware. For example, our hardware decompresses the second word in parallel by combining bytes from the input and the dictionary. This is challenging because the locations of the output bytes depend on the codes of the first word and second word. Given each code can have 6 possible values (Table III), there are 36 possible combinations of two codes, and 1,296 possible combinations for four codes. To achieve a balance between area and throughput, we

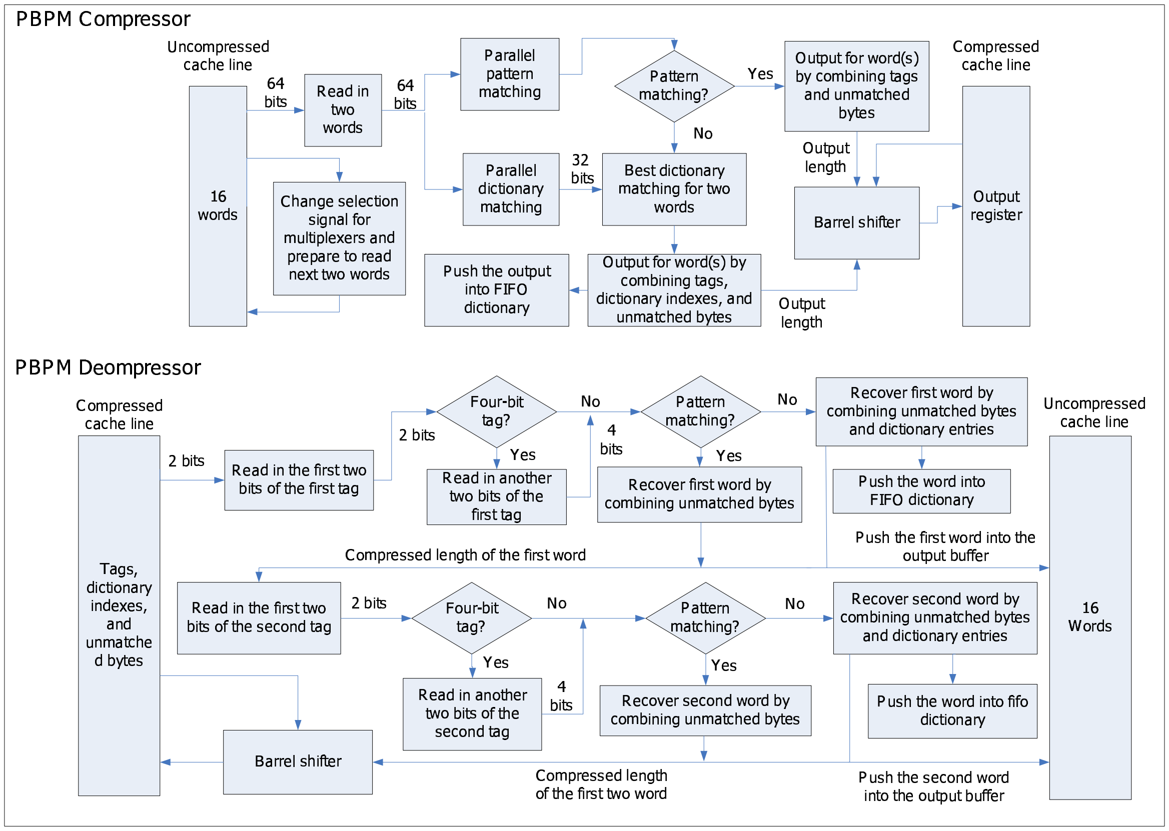

PBPM Compressor
Uncompressed cache line
16 words
64 bits
Read in two words
64 bits
Parallel pattern matching
Pattern matching?
Yes
No
Output for word(s) by combining tags and unmatched bytes
Compressed cache line
Change selection signal for multiplexers and prepare to read next two words
Parallel dictionary matching
32 bits
Best dictionary matching for two words
Output length
Barrel shifter
Output register
Push the output into FIFO dictionary
Output for word(s) by combining tags, dictionary indexes, and unmatched bytes
Output length
PBPM Deompressor
Compressed cache line
Tags, dictionary indexes, and unmatched bytes
2 bits
Read in the first two bits of the first tag
2 bits
Four-bit tag?
Yes
No
Read in another two bits of the first tag
4 bits
Pattern matching?
Yes
No
Recover first word by combining unmatched bytes
Recover first word by combining unmatched bytes and dictionary entries
Push the word into FIFO dictionary
Uncompressed cache line
16 Words
Push the first word into the output buffer
Compressed length of the first word
Read in the first two bits of the second tag
2 bits
Four-bit tag?
Yes
No
Read in another two bits of the second tag
4 bits
Pattern matching?
Yes
No
Recover second word by combining unmatched bytes
Recover second word by combining unmatched bytes and dictionary entries
Push the word into fifo dictionary
Barrel shifter
Compressed length of the first two word
Push the second word into the output buffer

Figure 4. Hardware compressor and decompressor.

decided to compress/decompress two words per cycle.

4.B.2) Description of the Hardware Compressor and Decompressor: Figure 4 illustrates the hardware compression and decompression process. Compression is decomposed into three pipeline stages to improve performance. The first pipeline stage performs pattern matching and dictionary matching on two uncompressed words in parallel. It then determines whether to push the two words into the dictionary depending on the pattern matching results. The second pipeline stage uses the dictionary matching results to compute the maximum number of matching bytes and the corresponding dictionary index, and to compute the total length of the compressed words. The last stage determines the final output for the two words by combining codes, matched bytes, and unmatched bytes from the previous stages. The output register is then shifted to store the compressed words.

TABLE III
PATTERN ENCODING IN PBPM

Code	Pattern	Output	Size (bits)	Frequency
00	`zzzz`	`(00)`	2	39.7%
01	`xxxx`	`(01)BBBB`	34	32.1%
10	`mmmm`	`(10)bbb`	5	7.6%
1100	`mmxx`	`(1100)bbbBB`	23	6.1%
1101	`zzzx`	`(1100)B`	12	7.3%
1110	`mmmx`	`(1110)bbbB`	15	7.2%

The decompressor reads in the code for the first word (see Table III). The first two bits of its code indicate its length. The data are then decoded to obtain a pattern of this word. If there is a match, the original word is recovered by combining zeroes and unmatched bytes. Otherwise, the original word is recovered by combining bytes from input and the corresponding dictionary entry; the word is then inserted into the dictionary. Note that unmatched bytes and dictionary indices are determined from successive input bits. The word is finally pushed into the output buffer. The same procedure is used for the second compressed word in the same cycle. Afterwards, the input line is shifted using a barrel shifter by the length of the first two compressed words, and the next two words are decompressed.

4.C. Synthesis Results of Control, Compression, and Decompression Hardware

Designing hardware of sufficient efficiency and performance is not obviously possible. Therefore, we verified feasibility by doing a detailed mostly-structural design of the compression and decompression hardware for PBPM and synthesized our design in 180 nm, 90 nm, and 65 nm libraries, using Synopsys Design Compiler. Table IV presents the resulting performance, area, and power consumption at maximum internal frequency. "Loc" refers to the compressed line locater/arbitrator described in Section 3.C. The total power consumption of the compressor, decompressor, and compressed line arbitrator at 1 GHz is about 69.7 mW in 65 nm technology. The total area cost is 0.109 mm^2.

4.D. Comparison with Literature

TABLE IV
SYNOPSYS DESIGN COMPILER SYNTHESIS RESULTS

Parameters	180 nm			90 nm			65 nm		
	Comp.	Decomp.	Loc.	Comp.	Decomp.	Loc.	Comp.	Decomp.	Loc.
Worst case delay (cycles)	11	8	2	11	8	2	11	8	2
Max. frequency (GHz)	0.46	0.38	0.48	1.25	1	1.72	1.43	1.25	1.89
Area (mm^2)	0.867	0.661	0.080	0.099	0.075	0.017	0.058	0.043	0.008
Power consumption (mW)	116.78	91.23	191.95	59.98	76.37	19.79	26.05	37.61	6.04

In this section, we compare our hardware implementation of PBPM to three existing hardware compression designs that have been considered for cache compression, namely X-Match [9], FPC [10], and MXT [11]. To evaluate the compression ratio of all algorithms, we used the same set of cache trace data, set the block size and dictionary size to 64 B and 32 B, and simulated a pair-matching compressed cache. We used Lempel-Ziv implementation (LZSS) to approximate compression results for an MXT implementation. Table V compares the average raw compression ratio (average compression ratio on individual cache lines), system-wide compression ratio (as defined in Section 3.C), and hardware performance (decompression latency, peak frequency, and area) of the candidates. Power consumption comparison is excluded because none of the other algorithms havw reported the power consumption of the hardware. Decompression latency is defined as the number of cycles to decompress a 64 B cache line, at the maximum claimed frequency of the hardware.

TABLE V
COMPARISON OF CACHE COMPRESSION HARDWARE

Candidates	Raw CR	System-wide CR	Impl.	Peak Frequency	Deco. Latency	Area
MXT	71.68%	76.51%	ASIC	133 MHz (0.25 μm) 511 MHz (65 nm)	16 cycles	n.a.
X-Match	49.50%	58.47%	FPGA	50 MHz (0.25 μm)	16 cycles	n.a.
FPC	63.39%	64.28%	n.a.	n.a.	n.a.	$>$ 0.31 mm^2
PBPM	51.76%	58.15%	ASIC	1 GHz (65 nm)	8 cycles	0.106 mm^2

Table V shows that PBPM has the best effective system-wide compression ratio and MXT has the worst. The poor compression ratio produced by MXT is mainly due to the limited dictionary size, which indicates MXT is not suitable for cache data compression. Because the raw compression ratios of both X-Match and PBPM are close to 50%, they achieve similar effective system-wide compression ratio.

Regarding the hardware performance comparison in Table V, MXT was implemented on a memory controller chip operating at 133MHz using 0.25 μm CMOS ASIC technology. The decompression latency is 8 B/cycle with 4 decompression engines. We scale the frequency up to 511 MHz by a factor of $(250/65)$, i.e., using constant electrical field scaling, to reflect the move to 65 nm technology. FPC has not been implemented on a hardware platform; no area, peak frequency, or power consumption numbers are reported. The authors claimed that assuming one processor cycle requires 12 FO4 gate delays, the decompression latency is limited to five processor cycles. However, they did not provide the synthesis results of their maximum frequency, and therefore we cannot compare performance with FPC. To estimate the area cost of FPC, we observe that the FPC compressor and decompressor are decomposed into multiple pipeline stages. Each of these stages imposes significant area overhead. For example, assuming each 2-to-1 multiplexer takes 5 gates, the fourth stage of the FPC decompression pipeline takes approximately 290 K gates or 0.31 mm^2 in 65 nm technology, more than the total area of our compressor and decompressor.

5. Full-System Simulation Results

This section describes full-system simulation results of the proposed cooperative compression and migration techniques. We first describe our simulation environment and workloads, and then present and analyze the results.

5.A. Simulation Environment

TABLE VI
Parameters in Simulations

Parameter	Value
Processor	In-order, 1 GHz
L1 I/D cache	32 KB, 2-way, 64 B line size, 2 cycles
L2 cache	8-way, 64 B line, 10 cycles private, 30 cycles shared
Off-chip memory	80 cycles, 10 bytes per cycle
On-chip network	Point-to-point, 5 cycles/hop, 100 bytes/cycle

We used the Simics [12] full-system simulator and the Ruby module from the GEMS infrastructure [2] to simulate the detailed memory system and interconnect network. We heavily modified the *MSI_MOSI_CMP_directory* cache coherence protocol provided by GEMS to support cache compression and migration. Our simulator is based on the Simics/Aurora SPARC Linux, which models the SPARC V9 architecture in sufficient detail to run a Linux 2.6.13 SMP kernel. We used the in-order processor model provided by Simics mainly to reduce simulation time. For all our simulations, each processor has private L1 instruction and data caches, and a private L2 instruction/data cache. Inclusion is maintained between L1 and L2 caches. The architectural parameters used in the simulations are listed in Table VI.

We used CACTI 5.0 beta [13] to estimate L2 cache area. We held line size (64 Byte) and associativity (8-way) constant. All cache area estimates are based on 65 nm process. For CPU area estimation, we assume a fixed-area processor model [14], in which each processor core and its associated L1 cache is have the same area as 1 MB of L2 cache, i.e., approximately 20 mm^2.

5.B. Workloads

To evaluate the cache compression and migration techniques, we used eight multiprogrammed workloads and four multithreaded workloads. Our multiprogrammed workloads are combinations of ten heterogeneous SPEC CPU2000 benchmarks and Data-Intensive Systems (DIS) stressmarks [4]. Our multithreaded workloads include two benchmarks (*ammp* and *art*) from the SPEC OMP v3.2 and two benchmarks (*CG* and *EP*) from the Nasa

Parallel Benchmarks [15] (NPB) v3.2. Our benchmark configurations are listed in Table VII. Below we describe the set up of the multiprogrammed and multithreaded benchmarks, respectively.

TABLE VII
BENCHMARKS

Multiprogrammed SPEC Benchmarks			Multiprogrammed DIS Benchmarks		
Mix 1	art, twolf	T1 and T1	Mix 5	matrix, field	T1 and T2
Mix 2	applu, mesa	T2 and T2	Mix 6	transitive, update	T1 and T1
Mix 3	art, mgrid	T1 and T2	Mix 7	matrix, transitive	T1 and T1
Mix 4	twolf, applu	T1 and T2	Mix 8	field, pointer	T2 and T2

For multiprogrammed workloads, we studied via simulation the memory characteristics of the SPEC CPU2000 benchmarks and the DIS stressmarks and divided them into two categories: performance sensitive to L2 cache size (*T1*) and performance-insensitive to L2 cache size (*T2*). We then used benchmark mixes that represent the following three types of application combinations: (1) mix of two cache-sensitive applications, (2) mix of one cache-sensitive application and one cache-insensitive application, and (3) mix of two cache-insensitive applications. For each mixed workload of two applications, we ran the simulation twice, each time starting at the "early single" SimPoint [16] of a different application, and stopping simulation when this application has executed 100 million instructions. We then averaged the results of the two runs to approximate the performance of this benchmark mix.

For multithreaded workloads, we set the number of threads to the number of processors. For the two multithreaded SPEC OMP benchmarks, *ammp* and *art*, we use the medium versions and the reference inputs. The central part of these two benchmarks is a major loop whose body contains code executed by multiple threads in parallel. We first fast forward to the major loop of the benchmarks, then warm up the cache with 100 million instructions, and finally measure performance in the following 100 million instructions. For the two multithreaded NPB benchmarks, *CG* and *EP*, we use the problem class S, warm up the cache with 100 million instructions, and then run to completion and measure the performance.

5.C. Performance Evaluation on Multiprogrammed Workloads

For multiprogrammed workloads, we performed two sets of experiments to evaluate the impact of cooperative and adaptive cache compression and migration on the overall system throughput and the performance of each application. First, we fixed the number of cores to two and varied the L2 cache size. We show that our techniques reduce cache requirements and thereby reduce total chip area. Second, we identify the impact of our techniques on optimal core-cache area ratio given a fixed on-chip area.

For each set of experiments, we present the results of multiprogrammed benchmarks listed in Table VII under four system settings: (1) basic private L2 cache, (2) compressed private L2 cache, (3) private L2 cache with migration under the control of marginal performance gain (using MPG_{T1} to control when migration should start), and (4) private L2 cache with adaptive and cooperative compression and migration. We use a fixed cache line decompression latency of eight cycles as indicated by our hardware synthesis results.

In order to decide the appropriate threshold values MPG_{T1}, MPG_{T2}, and MPG_{T3}, we measured the marginal performance gain values of all evaluated applications at the cache size range of 64 KB to 8 MB. Three threshold

TABLE VIII
THROUGHPUT IMPROVEMENTS FOR EVALUATED TECHNIQUES AS FUNCTIONS OF L2 CACHE SIZE

	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.
L2 size	art, mgrid			applu, mesa			art, twolf			applu, twolf		
64 KB	0%	0.5%	0.7%	0.6%	0.5%	0.6%	6.5%	8.8%	9.9%	2.0%	3.9%	3.9%
128 KB	0%	-0.5%	0%	0.5%	0%	0.5%	4.6%	6.3%	6.8%	1.8%	2.7%	2.8%
256 KB	0%	-0.4%	0%	0%	-1.8%	0%	3.4%	5.5%	5.7%	1.8%	2.8%	2.9%
512 KB	0%	2.0%	8.4%	0%	-2.1%	0%	3.1%	10.7%	12.7%	2.6%	4.7%	4.7%
1 MB	6.9%	11.3%	12.6%	0%	-2.0%	0%	1.1%	12.4%	17.0%	0%	-1.9%	0%
2 MB	0.4%	-1.7%	0.5%	0%	-2.1%	0%	0%	-10.7%	0%	0%	-4.1%	0%
L2 size	matrix, field			matrix, transitive			transitive, update			field, pointer		
64 KB	0.6%	1.0%	0.8%	2.1%	4.8%	2.7%	0.6%	1.1%	0.6%	0%	-4.2%	0%
128 KB	0%	1.0%	1.1%	2.3%	14.6%	23.2%	0.2%	9.8%	15.2%	0%	-4.6%	0%
256 KB	3.9%	3.3%	4.7%	18.8%	34.9%	37.5%	14.8%	16.9%	17.5%	0%	-4.7%	0%
512 KB	0%	-1.6%	0%	0%	-8.3%	0%	0%	-3.2%	1.8%	0%	-4.8%	0%
1 MB	0%	-1.8%	0%	0%	-9.3%	0%	1.7%	3.0%	2.1%	0%	-4.8%	0%
2 MB	0%	-1.7%	0%	0%	-9.2%	0%	0%	-5.8%	0%	0%	-4.9%	0%

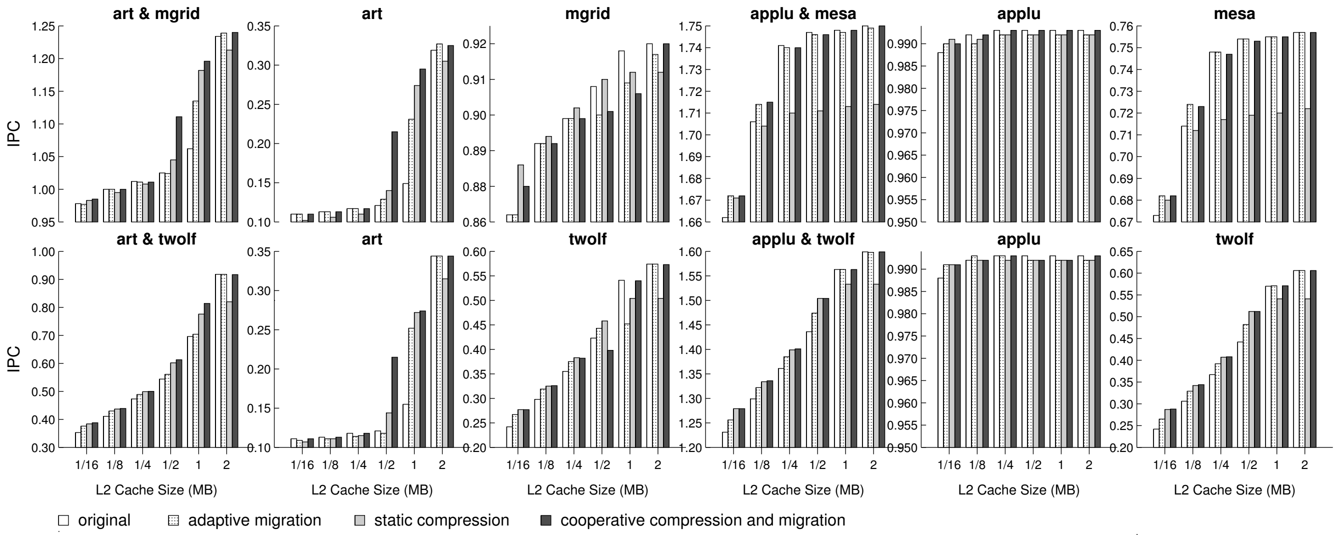


Figure 5. Performance of multiprogrammed SPEC benchmarks as a function of cache size.

values spanning the range of variation in performance for cache-sensitive applications were empirically determined (MPG_{T1} = 0.0000375, MPG_{T2} = 0.0000625, and MPG_{T3} = 0.0001). A single set of thresholds were used: it was not necessary to tune them to particular application.

5.C.1) Evaluation at Fixed Core Count: In this section, we show the evaluation of the cooperative compression and migration techniques in a two-core CMP with varied private L2 cache sizes. The overall throughput and individual performance of evaluated multiprogrammed benchmarks are shown in Figure 5 and Figure 6. Table VIII summarizes the relative throughput improvements of the evaluated techniques: adaptive migration (mig.), static compression (comp.), and cooperative and adaptive compression and migration (coop.).

As shown in Figure 5 and Figure 6, the cooperative cache compression and migration techniques provide the maximum throughput improvement over all system settings whenever improvement is possible by increasing L2 cache size. When such improvement is not possible, the cooperative technique maintains the lowest performance degradation due to its adaptive nature. We find that for the three *T1* + *T2* mixes (i.e., *art* and *mgrid*, *applu* and *twolf*, as well as *matrix* and *field*) the proposed techniques result in significant throughput improvements. The maximum improvements for these applications over different L2 cache sizes range from 4.7% to 12.6%. For the three *T1*

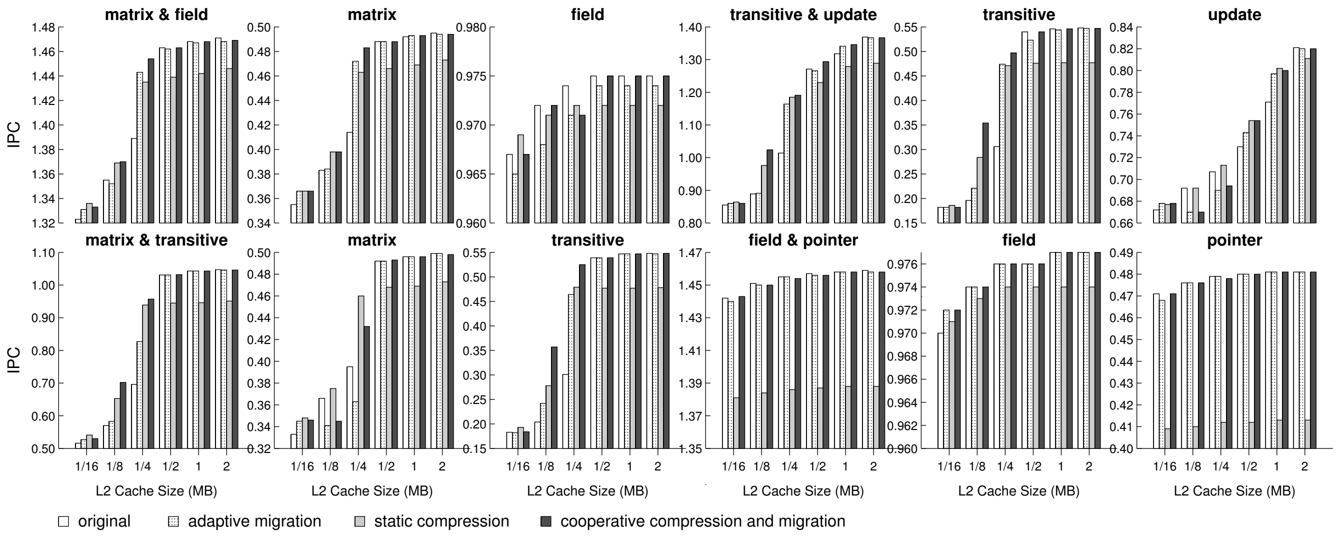


Figure 6. Performance of multiprogrammed DIS benchmarks as a function of cache size.

+ *T1* mixes (i.e., *art* and *twolf*, *matrix* and *transitive*, as well as *transitive* and *update*) the maximum throughput improvements range from 17% to 37.5%. The other two *T2* + *T2* mixes (i.e., *applu* and *mesa*, as well as *field* and pointer) are not affected by our techniques because their performance cannot be improved with increased cache size. We make the following observations on the evaluated techniques.:

1. The cooperative technique combine the advantages of both static compression and adaptive migration: in all cases, it enables the maximum performance improvement in sensitive cache size ranges, and results in minimum performance penalty in insensitive cache size ranges. Therefore, if compression and/or migration are used, they should be adaptively guided using a metric such as the proposed marginal performance gain.

2. In their sensitive cache size ranges, *T1* applications usually benefit more from static compression than adaptive migration. This is because locally compressed caches have lower access latency than remote caches. However, in cache size ranges for which performance is not strongly dependent on cache size, static compression usually results in a performance penalty, due to the increased cache hit latency. In contrast, adaptive migration seldom imposes any performance penalty because lines are not migrated in this case.

3. The cooperative technique has the greatest potential to reduce on-chip cache requirement. For example, for the *matrix* and *field* mix, with the cooperative techniques, the throughput at the cache size of 256 KB is 99.4% of the throughput at the cache size of 512 KB; the individual performance of *matrix* and *field* are also almost identical. This implies that the actual cache area requirement can be reduced to half, and the total chip area requirement can be reduced by 17%. For the mix of *art* and *mgrid*, to achieve similar performance of a 2 MB private cache (4 MB in total for two cores), only half the cache area is required with the proposed technique, thereby reducing the total chip area by 33%.

TABLE X
THROUGHPUT IMPROVEMENTS FOR EVALUATED TECHNIQUES FOR DIFFERENT CORE–CACHE AREA USAGES

	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.
Cores		art, mgrid			applu, mesa			art, twolf			applu, twolf	
2	0%	-5.2%	0%	0%	-2.0%	0%	0%	-13.9%	0%	0%	-4.1%	0%
4	5.6%	5.6%	9.9%	0%	-1.2%	0%	4.9%	4.1%	7.9%	0%	-3.0%	0%
6	0%	7.7%	8.1%	0%	-2.1%	0%	3.5%	18.0%	18.0%	1.4%	1.4%	1.6%
8	0%	-0.5%	0%	0%	-1.1%	0%	3.2%	5.8%	7.6%	1.0%	1.8%	2.5%
Cores		matrix, field			matrix, transitive			transitive, update			field, pointer	
2	0%	-1.6%	0%	0%	-9.8%	0%	0%	-6.5%	0%	0%	-4.9%	0%
4	0%	-1.7%	0%	0%	-9.9%	0.1%	0.7%	-4.3%	1.1%	0%	-4.7%	0.1%
6	0%	-1.5%	0.1%	0%	-4.6%	0%	1.0%	-1.2%	2.8%	0%	-4.8%	0%
8	3.7%	3.2%	4.3%	19.8%	34.0%	36.0%	11.0%	16.8%	17.6%	0%	-4.4%	0%

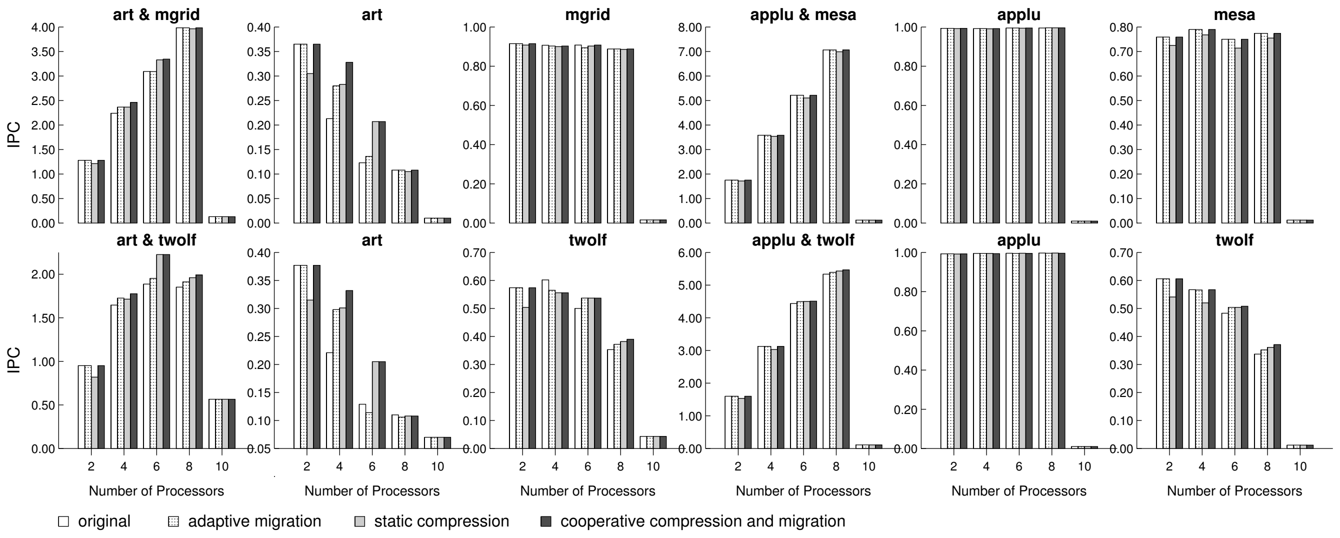


Figure 7. Performance of multiprogrammed SPEC benchmarks for different cache–core area tradeoffs.

TABLE IX
EVALUATED CORE-CACHE AREA RATIOS

Core/L2 area ratio	Number of cores	Core area (mm^2)	L2 size (MB)	Cache area (mm^2)	Total area (mm^2)
0.28	2	40	8	143.1	183.1
0.72	4	80	6	109.9	189.9
1.70	6	120	4	70.6	190.6
4.37	8	160	2	36.6	196.6
∞	10	200	0	0	200

5.C.2) Evaluation at Various Core-Cache Area Ratio: This section evaluates the impact of the proposed techniques on CMPs with different core-cache area ratios given a fixed chip area constraint. The total chip area is approximately the sum of the cache area and the CPU area multiplied by number of CPUs. We assume the L1 caches are integrated into the CPU cores, and use the CACTI 5.0 Beta model [13] to estimate the area of L2 cache given its size (with a 65 nm technology). Finally, we set the total chip area constraint to 200 mm^2: the approximate area of a 10 MB L2 cache.

To demonstrate the effect of migration, we simulate configurations at two-processor increments to balance the mixes of application types. Each core executes one thread. For example, in a six-core system, for benchmark mix 1, three copies of *art* and three copies of *mgrid* are executed, and each copy is assigned to a single processor core. We evaluate the core-cache area ratios listed in Table IX, approximating the area constraint of 200 mm^2 as closely as possible. Figures 7 and 8 report the IPC of each individual core and the total throughput. Table X summarizes the relative throughput improvements of the three evaluated techniques. Note that the 10-core configuration has no L2 cache and therefore consistently suffered large performance penalties. We plot the results of 10-core CMP

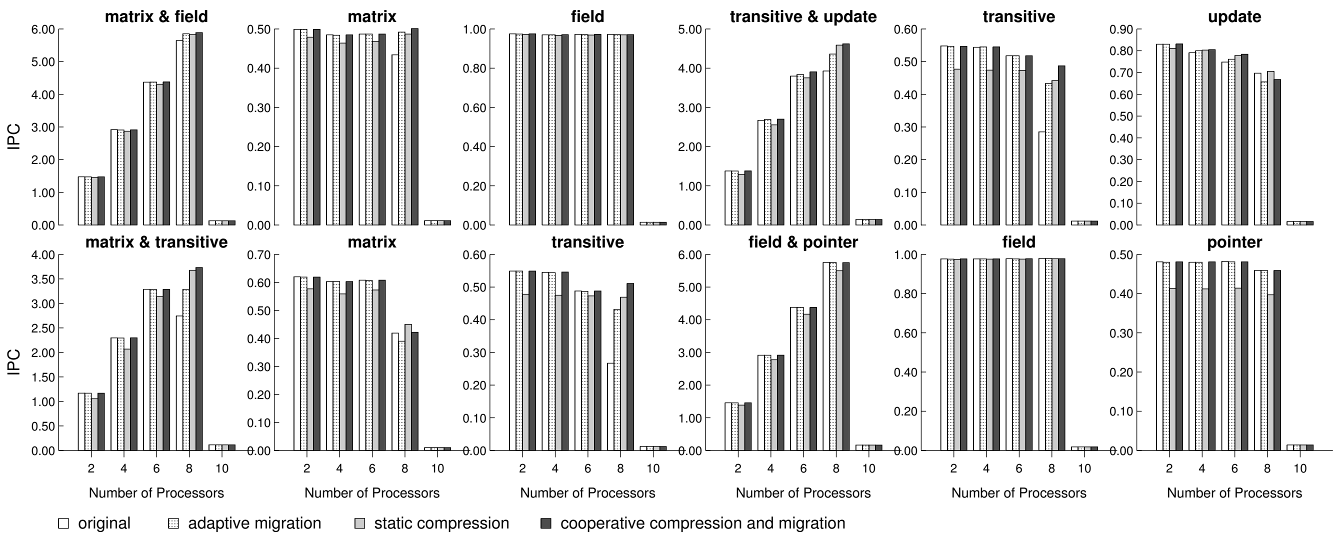


Figure 8. Performance of multiprogrammed DIS benchmarks for different cache-core area tradeoffs.

in the figures to illustrate an extreme case: with no on-chip L2 cache, none of the evaluated techniques can help improve throughput.

The behavior of the evaluated technique for different benchmark mixes is similar to that shown in Section 5.C.1. Recall that our goal is to optimize the overall throughput, i.e., total IPC over all cores. We make the following observations on the evaluated techniques.

1. The area-constrained, performance-optimal solution uses less cache per core than is common for existing processors. In current dual-core and quad-core processors, a significant portion of the chip area is devoted to the cache. The Intel Core 2 Duo processor has a die size of 143 mm^2, about 50% of which appears to be used by the 4 MB L2 cache. The AMD Athlon 64 FX-62 processor also has 50% of its die area used as on-chip cache. However, our experimental results indicate that adding processor cores often achieves better throughput than increasing L2 cache size. We feel that the current design trend of CMPs has been influenced by the lack of on-line techniques to adjust cache use to running applications. For example, for the mix of *art* and *mgrid*, when the core count is increased from four to six, although the throughput is improved by 38%, the performance of *art* reduces by 42.3%. The performance of *mgrid* is not affected by this change. However, with the proposed cooperative and adaptive techniques, the individual performance of *art* at the core count of six is 97% of the original performance without our techniques at the core count of four. This indicates that, for this benchmark mix, with the proposed adaptive techniques, the 6-core design not only outperforms the 4-core design, providing 36% higher throughput, but also guarantees same good performance for individual cores.

2. We found that the cooperative techniques may influence optimal core-cache area ratio for maximum throughput. For the mix of *art* and *twolf*, without the cooperative techniques, the overall performances of 6-core CMP and 8-core CMP are approximately the same. However, with the 18% performance improvement brought by cooperative compression and migration, it is clear that the optimal core count is six instead of eight. On the contrary, for the

mix of *matrix* and *transitive*, without the cooperative techniques, the 6-core CMP has the maximum throughput. However, with cooperative cache compression and migration, a 36% improvement makes the throughput of 8-core 13.5% higher than that of the 6-core design, thereby becoming the new optimal core-cache ratio. In summary, the optimal core-cache ratio depends on the application mix for which the CMP is optimized and is influenced by the use of adaptive cache compression and data migration.

5.D. Performance Evaluation on Multithreaded Workloads

In this section, we present the evaluation results of the cooperative cache compression and migration technique on four multithreaded workloads. We compare the overall system throughput of our technique with that of a private L2 cache and a shared L2 cache on a four-core CMP. We set the number of OpenMP threads to the number of processors. We evaluated four L2 cache sizes; in each case, the aggregate cache size is the same for each comparison, i.e., 1 MB private cache vs. 4 MB shared cache. The results are presented in Table XI.

TABLE XI
EVALUATION ON MULTITHREADED BENCHMARKS AS A FUNCTION OF AGGREGATE L2 CACHE SIZE

Total L2	Pivate L2	Shared L2	Coop.	+% private	+% shared
		ammp			
1MB	1.37	2.98	1.37	0%	-54%
2MB	1.39	3.00	3.61	160%	21%
4MB	4.23	3.00	4.23	0%	41%
8MB	4.25	3.01	4.25	0%	41%
		art			
1MB	3.01	3.13	3.01	0%	-4%
2MB	3.08	3.23	3.14	2%	-3%
4MB	3.26	3.34	3.26	0%	-2%
8MB	3.30	3.35	3.31	0%	-1%
		CG			
1MB	2.85	2.55	3.96	39%	55%
2MB	4.36	3.40	4.34	0%	28%
4MB	4.46	5.38	5.42	22%	1%
8MB	5.52	5.39	5.53	0%	3%
		EP			
1MB	5.41	5.37	5.41	0%	1%
2MB	5.46	5.40	5.89	8%	9%
4MB	6.18	6.31	6.88	11%	9%
8MB	7.06	6.74	7.05	0%	5%

The OpenMP implementations of the SPEC OMP and NPB benchmarks achieve parallelization by searching for loops with fully independent iterations and then annotating these loops with `OMP PARALLEL DO` directives. Therefore, the threads usually perform identical tasks and seldom communicate or share data. As a result, the marginal performance gains of the threads are very similar and we observed little migration for these benchmarks. The benefits of our technique mainly comes from adaptive cache compression. Table XI illustrates that our technique outperforms private cache in all cases and outperforms shared cache in most cases. In specific, for benchmark *ammp*, the maximum throughput improvement over private cache is 160%, and the maximum throughput improvement over shared cache is 41%. The only case where our technique has a lower performance than a shared cache is when the cache size per core is 256 KB. This is because at that cache size, the marginal performance gain of *ammp* is not large enough to trigger compression. (Doubling cache size does not improve throughput, but quadrupling it does.) The only case when shared L2 cache performs the best is for benchmark *art*. The cooperative technique does not result in obvious throughput improvement for *art* because its marginal performance gain is small over all evaluated cache sizes.

Our evaluation of cache-sensitive multithreaded benchmarks indicates that if the threads of the application execute almost identical tasks, the difference between their marginal performance gain is too small to trigger migration. Therefore, only compression may help improve the throughput. However, note that our migration technique is orthogonal to replication techniques. Therefore, other replication techniques may also be deployed if the applications exhibit good data sharing characteristics.

5.E. Sensitivity to Decompression Latency

To determine the sensitivity of cooperative cache compression and migration techniques to decompression latency, we performed the following experiments. We used the experimental setup in Section 5.C.1 with a fixed core count of two and evaluated the benchmark mixes that benefit from cooperative techniques at their largest performance improvement cache size. We varied the decompression latency from 2 to 12 cycles. Figure 9 illustrates the improvements to overall throughput under different decompression penalties. Note that it illustrates the importance of decompression speed, not variation in PBPM decompression speed. We found that the performance improvements for all benchmark mixes are strongly affected by increased decompression latency. In fact, the percentage improvement is directly linear in decompression latency. The results clearly demonstrate the importance of efficient compression/decompression hardware design.

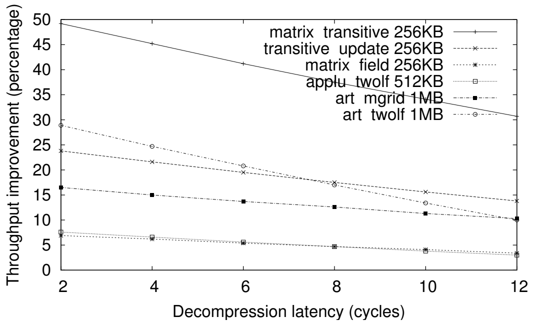


Figure 9. Sensitivity to decompression latency.

6. Related Work

When exploring the design space for future CMPs, Huh, Burger, and Keckler [17] observed that off-chip bandwidth will likely limit the number of cores per die because transistor count is increasing much faster than the number of signaling pins. Li et al. [14] studied the tradeoff between number of cores and cache size under area constraints. Their results showed the challenges of accommodating both CPU-bound and memory-bound workloads in the same design. The proposed techniques provide the most benefit when used with such applications.

A number of cache compression [5, 6] and memory compression [11, 18] systems have been developed for improving single-processor memory system performance. However, few researchers have studied reducing off-chip communication cost using compression for CMPs. We are aware of only two recent articles on this topic. Ozturk et al. [19] proposed to compress data blocks for which reuse intervals are large using compile time optimization based on an integer-linear programming formulation. This approach was evaluated using access patterns extracted from embedded applications. However, it has not been implemented or evaluated within a multiprocessor simulator: the hardware complexity and performance impact are not yet well understood. Alameldeen and Wood investigated L2 cache and link compression in CMPs [20]. Their CMP design assumed private L1 caches for each core and a shared L2 cache for all cores. However, we believe compression is more important for private L2 caches because they face more severe capacity issue than shared caches. Moreover, decompression adds overhead to the shared cache access latency, which is already much higher than private caches. Therefore, their technique is more useful for larger commercial benchmarks with huge cache requirements.

Suh et al. [7] proposed a dynamic technique to partition shared on-chip cache among processes. Their technique collects the cache miss characteristics of processes at run-time and uses this information to partition cache among cores. They defined their control metric to be the derivative of the cache miss curve. However, this definition has shortcomings and may produce suboptimal performance. Because the same reduction in miss rate may have different performance implications for different applications. For example, as shown in Figure 3, for application *art*, doubling the L2 cache size at 512 KB would result in six fewer misses per thousand instructions, and 0.05 absolute IPC improvement. Meanwhile, for application *twolf*, doubling the cache size at 512 KB would result in two fewer misses per thousand instructions, but a 0.1 absolute IPC improvement. Defining marginal performance gain as the reduction in cache misses implies that the benefit of increasing cache size for *art* is larger than *twolf*, which is not correct. If one's goal is to improve performance rather than reduce miss, our definition of marginal performance gain should be used instead.

Quresi and Patt [21] improved upon Suh's dynamic cache partitioning scheme by separating the utility monitoring unit from the shared cache. However, they used reduction in misses to make cache partitioning decisions, the same metric as in Suh's work. Chang and Sohi [22] proposed cooperative caching to manage the distributed on-chip caches for CMPs. Their techniques include cache-to-cache transfers of clean data, replication-aware data replacement, and global replacement of inactive data. Zhang and Asanovic proposed victim replication [23] for private L2 caches, which attempts to keep copies of local primary cache victims within the local L2 cache. They later proposed victim migration [24] that improves on victim replication. Beckmann et al. proposed the adaptive selective replication (ASR) scheme [25], a more advanced replication scheme for private caches. Our migration technique is orthogonal to these replication techniques, because they make decisions on whether to duplicate a cache line fetched from a remote cache and we make decisions on what we should do with a useful cache line upon eviction. In fact, our adaptive compression and migration technique can be used together with any replication techniques, and is likely to improve their performance by replicating and transmitting compressed data.

7. Conclusions

The move to CMPs increases the importance of carefully using available cache area. We proposed a cooperative L2 cache compression and data migration technique to permit improvement in CMP throughput without increasing area, or to reduce area without degrading throughput. We evaluated these techniques using full-system simulation running various multiprogrammed and multithreaded workloads. We show that for cache-sensitive applications, the maximum CMP throughput improvement with the proposed technique range from 4.7%–160% (on average 34.3%), relative to a conventional private L2 cache architecture. No performance penalty is imposed for cache-insensitive applications. The results indicate that proactively distributing cache resources among processors based on their relative performance impacts permits the best overall performance on a wide range of applications. Using a cooperative combination of compression and migration guided by marginal performance gain permitted better performance than either migration or compression, alone. The feasibility, area, and performance overheads of the required control, compression, and decompression hardware were evaluated via detailed design and synthesis. The performance impact was explicitly modelled and the area overhead was found to be negligible compared to processor and cache area. We therefore conclude that cooperative compression and migration appears to be a viable technique for improving area and/or performance for workloads containing cache-limited applications.

References

[1] "International Technology Roadmap for Semiconductors," 2006, http://public.itrs.net.

[2] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "GEMS: Multifacet's general execution-driven multiprocessor simulator," in *Proc. Int. Symp. Computer Architecture*, June 2005.

[3] L. Yang, H. Lekatsas, and R. P. Dick, "High-Performance Operating System Controlled Memory Compression," in *Proc. Design Automation Conf.*, July 2006, pp. 701–704.

[4] H. Du, "Analysis of memory behavior of DIS stressmark suite and optimization," University of California, Irvine, Tech. Rep., Dec. 2000, http://www.ics.uci.edu/~amrm/hdu/DIS_Stressmark/DIS_stressmark.html.

[5] E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in *Proc. 3rd workshop on Memory performance issues*, 2004.

[6] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. Int. Symp. Computer Architecture*, June 2004.

[7] G. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[8] H. Ghasemzadeh, S. S. Mazrouee, and M. R. Kakoee, "Modified pseudo LRU replacement algorithm," in *Proc. Int. Symp. Engineering of Computer Based Systems*, Mar. 2006.

[9] J. N and S. Jones, "The X-MatchPRO 100 Mbytes/second FPGA-based lossless data compressor," in *Proc. Design, Automation and Test in Europe*, Mar. 2000, pp. 139–142.

[10] A. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," Dept. of Computer Sciences, University of Wisconsin-Madison, Tech. Rep., Apr. 2004.

[11] B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. Wazlowski, and P. M. Bland, "IBM memory expansion technology," *IBM J. of Research and Development*, vol. 45, no. 2, pp. 271–285, Mar. 2001.

[12] "Simics," http://www.virtutech.com.

[13] "CACTI: An integrated cache access time, cycle time, area, leakage, and dynamic power model," http://quid.hpl.hp.com:9082/cacti/.

[14] Y. Li, B. Leez, D. Brooks, Z. Huyy, and K. Skadron, "CMP design space exploration subject to physical constraints," in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2006.

[15] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," Tech. Rep., 1999, nAS Technical Report: NAS-99-011.

[16] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2003.

[17] J. Huh, D. Burger, and S. W. Keckler, "Exploring the design space of future CMPs," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2001.

[18] L. Benini, D. Bruni, A. Macii, and E. Macii, "Hardware-assisted data compression for energy minimization in systems with embedded processors," in *Proc. Design, Automation & Test in Europe Conf.*, Mar. 2002.

[19] O. Ozturk, M. Kandemir, and M. J. Irwin, "Increasing on-chip memory space utilization for embedded chip multiprocessors through data compression," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Sept. 2005, pp. 87–92.

[20] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2007.

[21] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. Int. Symp. Microarchitecture*, Dec. 2006.

[22] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *Proc. Int. Symp. Computer Architecture*, June 2006, pp. 264–276.

[23] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled CMPs," in *Proc. Int. Symp. Computer Architecture*, June 2005.

[24] M. Zhang and K. Asanovic, "Victim migration: Dynamically adapting between private and shared CMP caches," Massachusetts Institute of Technology, Tech. Rep., Oct. 2006.

[25] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive selective replication for cmp caches," in *Proc. Int. Symp. Microarchitecture*, Dec. 2006.