Integrating Memory Compression and Decompression with Coherence Protocols in Distributed Shared Memory Multiprocessors

Lakshmana Rao Vittanala
Intel Technology India Pvt. Ltd.
136 Airport Road
Bangalore 560017
INDIA
lakshman.vittanala@intel.com

Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

Abstract

Ever-increasing memory footprint of applications and increasing mainstream popularity of shared memory parallel computing motivate us to explore memory compression potential in distributed shared memory (DSM) multiprocessors. This paper for the first time integrates on-the-fly cache block compression/decompression algorithms in the cache coherence protocols by leveraging the directory structure already present in these scalable machines. Our proposal is unique in the sense that instead of employing custom compression/decompression hardware, we use a simple on-die protocol processing core in dual-core nodes for running our directory-based coherence protocol suitably extended with compression/decompression algorithms. We design a low-overhead compression scheme based on frequent patterns and zero runs present in the evicted dirty L2 cache blocks. Our compression algorithm examines the first eight bytes of an evicted dirty L2 block arriving at the home memory controller and speculates which compression scheme to invoke for the rest of the block. Our customized algorithm for handling completely zero cache blocks helps hide a significant amount of memory access latency. Our simulation-based experiments on a 16-node DSM multiprocessor with seven scientific computing applications show that our best design achieves, on average, 16% to 73% storage saving per evicted dirty L2 cache block for four out of the seven applications at the expense of at most 15% increased parallel execution time.

1. Introduction

The memory footprint of high-end data-intensive applications is ever-increasing. While parallel computing makes inroad to mainstream applications, efficient management of physical memory will be an important issue affecting the end-performance of these applications as well as the energy consumption of the system. In this paper we explore the performance of memory compression in a medium-scale distributed shared memory (DSM) multiprocessor executing some popular scientific computing parallel workloads. We integrate the compression/decompression algorithms in the underlying directory-based cache coherence protocol. To the best of our knowledge, this is the first attempt to look at memory compression as a coherence protocol extension. When a dirty cache block is evicted from the last level of cache hierarchy (in this paper, we consider a two-level hierarchy), it is sent to the home memory controller for appropriate coherence book-keeping and updation of main memory. Our compression algorithm is invoked at this point and a compressed cache block is written back to main memory, thereby reducing the pressure on the memory. When a cache block is requested by a node (on an L2 cache miss), the request is forwarded to the home memory controller and at this point our decompression algorithm is invoked. Therefore, the main processor always receives uncompressed cache blocks and its cache hierarchy does not require any modification at all. All the modifications come in the form of a new coherence protocol which integrates the compression and the decompression algorithms. We leverage the directory structure and a small header to store compression information on a per cache block basis (each L2 cache block has its own directory entry, as in the usual protocols).

We exploit the flexibility of a simple protocol core per node to design compression-aware coherence protocols without any extra hardware in the memory controller. Such protocol processors are used in an array of DSM multiprocessors [4, 21, 24, 27, 28] to increase the flexibility in the choice of the protocol and in some cases to simplify the protocol verification process. Although these designs did not consider using a spare core sharing the same die with the main core for executing the coherence protocol, in this work we envision an architecture that uses simple and less powerful spare cores for running the directory-based coherence protocol in software. With the trend toward larger number of on-die cores, such an architecture presents a cost-effective design point for the next-generation scalable DSM multiprocessors. In our case we can execute a compression-enabled protocol or a vanilla baseline bitvector protocol without requiring any hardware changes. Essentially, the integrated protocol core executes a chosen coherence protocol in software. We present two simple compression algorithms and combine them in the coherence protocol. Our execution-driven simulations on a 16-node DSM multiprocessor running seven explicitly parallel scientific computing workloads show that our best configuration saves up to 73% storage of an L2 cache block on average while experiencing at most 15% increase in execution time. The major challenge in the design is to maintain a good compression ratio while keeping the performance overhead at an acceptable level.

In the following, we briefly mention prior research in the area of cache and memory compression. Section 2 presents the details of our simulation environment. We discuss the extensions to the baseline bitvector coherence protocol in Section 3. The compression and decompression algorithms are discussed in Section 4 while the detailed simulation results for these algorithms are presented in Section 5. We conclude in Section 6.

1.1. Related Work

A significant amount of research has been done in the area of memory and cache compression for single-threaded systems. Historically, dictionary-base memory compression has received wide attention. IBM's MXT technology [1, 13, 31] implements a parallel block referential compression algorithm in hardware and employs a large 32 MB tertiary cache to hide the memory decompression overhead as much as possible. The IBM MXT algorithm is recently extended to compress on-chip cache [16] using indirection-based full associativity [17]. X-Match algorithm [20] carries out compression in the presence of full or partial match with a dictionary entry. Principle of locality is used in [8] to compress the address and data streams on the system bus. Profile-driven and differential compression/decompression algorithms for memory blocks on the path between cache and memory are explored in [5, 6]. The profile-driven scheme builds a CAM dictionary from profile information and maintains a RAM for decompression, while the online differential scheme relies on the fact that several bits in different words of a cache block may be common (e.g., the higher order bits). X-RL algorithm [23] compresses L2 cache and memory blocks with X-Match algorithm enhanced with a special treatment for zero runs. Since the dictionary-based compression techniques work best for data granularity larger than cache blocks, they have been used to compress physical page frames, thereby accommodating a larger working set in memory [10, 11]. Compiler transformations to exploit common prefix and narrow data for compressing dynamic data structures (32-bit address pointers as well as integer fields) have been explored in [34].

Our work is most closely related to the computationally less demanding compression/decompression schemes presented in [2, 3, 12]. While frequent pattern-based compression schemes are implemented for on-chip cache blocks in [2, 3] to increase the effective cache area, an in-depth study of zero-aware compression schemes for memory blocks (off-chip compression) is presented in [12]. Our compression algorithms are influenced by the findings of these studies. However, none of these studies explore the effectiveness of compression algorithms in a scalable DSM multiprocessor environment. We present an effective way of combining frequent pattern and zero-aware compression schemes. Our design does not require any extra hardware, as we integrate the compression/decompression algorithms in the software cache coherence protocols executing on protocol cores of a hardware DSM multiprocessor.

2. Simulation Environment

This section presents our simulated DSM multiprocessor environment and the benchmark applications we use for evaluating our compression algorithms. We simulate a 16-node system where each node contains a modern out-of-order eight-wide core, an on-die integrated memory controller [9, 18, 19, 29, 30] clocked at half the frequency of the main core, an on-die in-order static dual-issue protocol core clocked at the same frequency as the memory controller, an on-die integrated e-cube router, and off-chip local SDRAM banks. Figure 1 shows a high-level block diagram of one node. The nodes are interconnected by a scalable hyper-

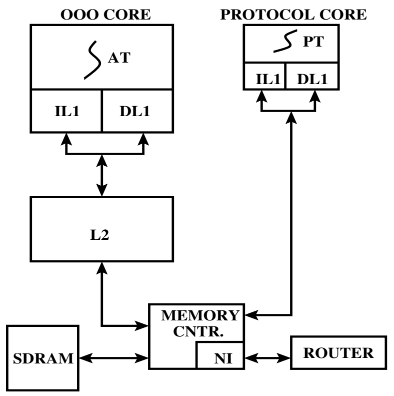

Figure 1. Simulated node architecture. AT denotes the application thread which runs on the out-of-order (OOO) main core. PT denotes the directory protocol thread which runs on the protocol core. Each core has its own L1 instruction (IL1) and data (DL1) caches while only the main core has an L2 cache. The network interface (NI), which talks to the router, is integrated into the memory controller. Everything except the SDRAM is on the same die.

cube network. The in-order static dual-issue protocol core executes a directory-based coherence protocol. The protocol core design is derived from the Memory and General Interconnect Controller (MAGIC) of the Stanford FLASH multiprocessor [15, 21]. However, we have extensively modified the design of MAGIC to closely resemble the hub of the SGI Origin 2000 [22]. The directory-based coherence protocol is similar to the write-invalidate full-map bitvector protocol implemented in the SGI Origin for small and medium-scale servers. We simulate a 64-bit directory entry (to match the datapath width of the protocol core), of which 16 bits are used to maintain sharer and owner information and four bits are used for coherence states (dirty, local, and two flavors of busy states). As discussed in the next section, compressed memory block's metadata is maintained in 38 out of the remaining 44 bits. The protocol core has a MIPS-like ISA and enjoys its own single-level of instruction and data caches backed by local SDRAM. Both the caches are sized for single-cycle access at 65 nm. In this study, one or two protocol cores may be present per node. We also explore the impact of both in-order and out-of-order network interface (NI) message scheduling in the memory controller. Table 1 summarizes the salient parameters. In the rest of this article we will interchangeably use the terms protocol core and protocol processor (PP). We evaluate our compression algorithms on seven explicitly parallel applications, six of which are selected from the SPLASH-2 suite [32]. The details are in Table 2.

Table 1. Simulated DSM architecture

Parameter	Value
Number of nodes	16
Main core	
Frequency	3.2 GHz
Pipe stages	22
Front-end/Commit width	8/8
BTB	256 sets, 4-way
Branch predictor	Tournament (Alpha 21264)
RAS	32 entries
Branch mispred. penalty	18 cycles (minimum)
ROB size	192 entries
Integer/FP Register	224/224
Integer/FP/LS issue queue	48/48/64 entries
ALU/FPU	8 (two for addr. calc.)/3
Integer mult./div. latency	6/35 cycles
FP mult./div. latency	2/12 cycles
Data cache ports	2
ITLB, DTLB	128/fully assoc./LRU
Page size	4 KB
L1 Icache	32 KB/64B/2-way/LRU
L1 Dcache	32 KB/32B/2-way/LRU
Unified L2 cache	512 KB/128B/8-way/LRU
L1 cache hit latency	3 cycles
L2 cache hit latency	11 cycles (round trip)
System	
System bus width	64 bits
System bus frequency	1.6 GHz
Memory controller freq.	1.6 GHz
SDRAM frequency	DDR2 400 MHz
SDRAM page hit latency	40 ns
SDRAM page miss latency	80 ns
SDRAM bandwidth	6.4 GB/s
PP frequency	1.6 GHz
PP Icache	32 KB/128B/direct map
PP Dcache	128 KB/128B/direct map
Router ports	6 (SGI Spider)
Network topology	2-way bristled hypercube
Hop time, link bandwidth	10 ns, 3.2 GB/s

Table 2. Simulated applications

Application	Problem size
Barnes-Hut [32]	8192 particles, three time steps
FFT [32]	1M complex double points, tiled for DTLB
FFTW [14]	8192×16×16 complex double points, 32×32 tiles
LU [32]	512×512 matrix, 16×16 tiles
Ocean [32]	514×514 grid, 1e-5 tolerance
Radix-Sort [32]	2M integer keys, radix 32
Water [32]	1024 molecules, three time steps

3. Directory Protocol Extensions

Before we proceed to discuss the compression/decompression algorithms in the next section, in the following we point out the directory protocol extensions. Essentially, this involves figuring out the places where our compression or decompression algorithm needs to be invoked. In general, the compression extensions are needed in all places where the directory protocol initiates a memory write. Note that this can happen only in the home node because a cache block can be written to the home memory only. In the following discussion, we assume a simple MSI coherence protocol. However, extension to other coherence protocols (e.g., MESI, MOESI, or MOSI) can be handled in the same fashion. We will denote a processor requesting a cache block by RP, the protocol processor in the requesting node by RPP, the protocol processor at the home node of a requested cache block by HPP, the local main processor at the home node by HP, and a processor holding a cache block in dirty state by DP. We will also denote a read request by GET, a read-exclusive request by GETX, a writeback by WB, a writeback acknowledgment by WB_ACK, a read reply by PUT, and a read-exclusive reply by PUTX.

There are two categories of situations where a cache block compression would be invoked. The first category pertains to the arrival of a writeback message at the home node. The two cases in this category are shown in Figure 2. In the first case the writeback is originated from a non-home node while in the second case the writeback comes from the home's local processor. In both cases the home protocol processor compresses the evicted cache block before writing it back to memory.

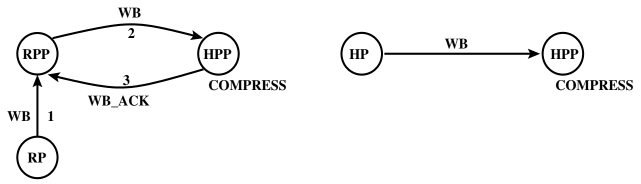

Figure 2. Compression cases for writeback

The second category of situations where a cache block compression is invoked pertains to the arrival of a read (GET) intervention reply at the home node. A typical example is shown in Figure 3 (a) where RP sends a cache block read request to the home node and the message is handled by the HPP at the home node. On consulting the directory entry, HPP finds that the requested cache block is dirty in processor DP meaning that the most up-to-date copy of the block is in the cache of DP. Therefore, the read request is forwarded to DP. DP sends a put reply to RP and also a sharing writeback (SWB) message to HPP. Both the messages carry the cache block. The SWB message is needed because now the directory state will be demoted from dirty to shared with both RP and DP marked as sharers. Subsequent requests to this cache block must be satisfied by the home memory itself. Therefore, the home memory must be updated with the most up-to-date copy of the block. On arrival of the SWB message, the HPP compresses the cache block and writes it back to memory. Two slight variations of this case are shown in Figures 3 (b) and 3 (c). In Figure 3 (b) the dirty processor is the local processor of the home node itself with the cache block residing in the local processor's cache in the dirty state. On the other hand, in Figure 3 (c) the requesting processor is the local processor of the home node itself.

Note that a read-exclusive (GETX) intervention does not update the home memory because it involves only an ownership hand-over and there is no demotion of state in the directory entry. Therefore, the GETX interventions do not invoke compression. There are few other cases where a cache block compression may be needed. But these are tied to some of the lower level details of how intervention races are handled in the underlying coherence protocol and in favor of brevity we omit the discussion of these.

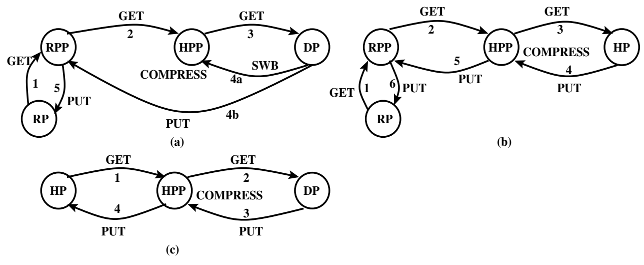

Figure 3. Compression cases for GET intervention

The decompression cases are relatively easier to identify. In general, when a cache block request (either GET or GETX) arrives at the home node, the HPP must decompress the memory block and send the decompressed block with the reply message (PUT or PUTX, respectively). The two cases are shown in Figure 4. In one case the requester is a non-home node while in the other the local processor of the home is the requester. In case of an upgrade request, the directory state may not indicate the requester as a sharer. Such a situation arises due to certain protocol races. In this case also the home protocol processor may have to decompress the memory block depending on the directory state and send it to the requester as a PUTX reply. Having found out the places where the compression/decompression algorithms should be invoked, now we are ready to discuss the design of these algorithms.

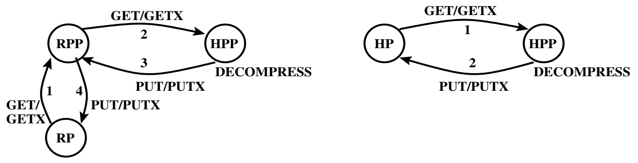

Figure 4. Decompression cases

4. Compression/Decompression Algorithms

We experimented with various potential bit patterns present in the evicted L2 cache blocks of the applications for a range of machine sizes and settled with three compression algorithms. All the algorithms consider 64 bits of a 128-byte cache block at a time. Thus, each of the 16 double words in a cache block may potentially undergo different compression algorithms. One of the compression schemes is very specific to the IEEE 754 double-precision floating-point format. Extensive use of double-precision floating-point numbers in the chosen applications (only Radix-Sort is a purely integer application) motivated us to experiment with this algorithm. Each of the algorithms can use one of four subschemes to compress the 64 bits under consideration. We start with the floating-point algorithm. The four subschemes are presented below.

- Encoding 00: 52-bit mantissa is zero. Only the sign bit and 11-bit exponent are stored.
- Encoding 01: 11-bit exponent is zero. Only the sign bit and 52-bit mantissa are stored.
- Encoding 10: 11-bit exponent is 1023 (this corresponds to exponent value zero). Only the sign bit and 52-bit mantissa are stored.
- Encoding 11: uncompressed. Full 64 bits are stored.

Although this algorithm produced good compression ratio, we dropped it from our final design due to extremely high decompression time.

The four subschemes of the second compression algorithm are presented below. This algorithm exploits zero runs as well as repeated word patterns within 64-bit double words.

- Encoding 00: 64-bit zero run. No data is stored.
- Encoding 01: the most significant 32 bits are zero. The least significant 32 bits are stored.
- Encoding 10: the most and the least significant 32 bits are equal. One of them is stored.
- Encoding 11: uncompressed. Full 64 bits are stored.

The four subschemes of the third algorithm are presented below. This algorithm exploits only zero runs.

- Encoding 00: 64-bit zero run. No data is stored.
- Encoding 01: the most significant 32 bits are zero. The least significant 32 bits are stored.
- Encoding 10: the least significant 32 bits are zero. The most significant 32 bits are stored.

- Encoding 11: uncompressed. Full 64 bits are stored.

As the evicted dirty cache blocks (clean evictions are dropped silently by the L2 cache controller) arrive at the home memory controller, it invokes the appropriate coherence protocol handler. Since the compression algorithms are integrated in the protocol handlers, the compression activity also gets carried out at this point. However, we found it impossible to run all the three algorithms one after another on each 64-bit chunk and store the best compressed block in memory. This would impose extremely high compression overhead. So our compression scheme unifies the aforementioned last two algorithms based on speculation (we drop the floating-point specific algorithm from further discussion due to high decompression overhead). We inspect the first 64 bits of a 128-byte cache block and decide which of the two algorithms should be invoked. If within the first 64 bits, the most and the least significant 32 bits match, we decide to use the first algorithm to compress all the 16 double words in the cache block. On the other hand, if at least one of the two 32-bit words is zero within the first 64 bits, we decide to compress all the 16 double words in the cache block using the second algorithm. If none of these tests on the first 64 bits pass, the cache block is stored uncompressed. While compressing a cache block, the running size of the compressed block is compared against a reconfigurable parameter, *maxCsz*. Only if the compressed size at the end is within this limit, is the compressed cache block written back to memory. As soon as the running size exceeds *maxCsz*, the compression is aborted and the block is stored in memory uncompressed. This parameter presents a flexible trade-off between the performance overhead and the compression ratio. We found that careful choice of this parameter is critical for performance.

Next, we turn to the decompression process. The main processors (specifically the TLBs) are not made aware of the compressed blocks. So they keep on generating uncompressed cache block addresses. When such an address arrives at the home memory controller (resulting from an L2 cache miss), the directory entry for the block is looked up first. In the following, we systematically derive the information required to generate the decompressed cache block. While decompressing a memory block, we need to know which algorithm was used to compress it. Since we have three choices (two algorithms and uncompressed), two bits out of the free 44 bits in the directory entry are sufficient. We call these two bits the compression state. Next, we need to locate the compressed cache block in the main memory. We use 32 bits out of the left-over 42 bits in the directory entry to store the address of a compressed cache block. It is clear from the algorithms that a compressed cache block's size will always be a multiple of four bytes. However, to minimize relocation we always make it a multiple of eight bytes potentially providing a cushion for use later, if needed. So, a compressed cache block's address will have the last three bits zero and there is no need to store these bits. Therefore, with 32 bits, we can access a total of 32 GB of compressed memory. Finally, we need to know within each algorithm which of the four subschemes is used for each 64-bit chunk of a 128-byte cache block. This is achieved by storing a 32-bit header with each compressed block (two bits are needed for each of the 16 double words to remember the subscheme used for that double word). Note that a fully uncompressed memory block does not store this header.

As an optimization, we decide to store the size of the compressed block in the directory entry also. This requires four bits, since a 128-byte cache block can contain at most 16 eight-byte chunks (compressed block size is always a multiple of eight bytes). This information helps us in quickly triggering a relocation, if the newly compressed size exceeds the last compressed size of a cache block. Finally, we take special care of completely zero cache blocks. In these cases, we do not store anything in memory (not even the 32-bit header). Instead, we use the fourth unused state in the directory entry's two compression state bits to identify such cache blocks. When such a cache block is identified from the directory entry of the requested address, the home memory controller immediately replies to the requester node with a zero cache block obviating the need to access the main memory. A careful reader may notice that we leave six bits in the 64-bit directory entry unused. These bits can be used later to further expand the compressed memory size or to store information about a more complex compression algorithm. In this paper, 70 bits (38 bits in directory entry and 32 bits in header) are used to store compression information per compressed block.

4.1. Implementation Issues

The protocol processor occupancy is the most important determinant of performance in the DSM multiprocessors [7]. In spite of delivering excellent compression ratio, the floating-point compression algorithm had to be dropped due to high occupancy. We found that one big bottleneck to compression performance is accessing the memory-mapped uncached data buffer containing the evicted dirty cache block to be compressed. This buffer is filled with the data payload coming through the network or processor interface of the home memory controller. Given a 64-bit datapath, sixteen uncached loads are needed to compress 128 bytes of data. Uncached accesses to these buffers are extremely slow. Therefore, in this paper we explore the possibility of caching the data buffer storage in the protocol processor's data cache. However, this leads to a coherence problem when the same data buffer is used next time. To solve it, the protocol processor, at the end of the decompression or compression, flushes the cache block containing the data buffer using cache index invalidate or cache index writeback instruction available in the MIPS ISA [26]. However, we found that draining a dirty block containing a modified buffer from the protocol processor's data cache hurts performance significantly. Therefore, our best design implements only cached loads to the data buffer space and continues to use uncached stores. Our algorithms running on the protocol core are tuned in such a way that they never require simultaneous loads and stores to the same data buffer. This allows us to implement independent policies for loads and stores to the data buffer space (e.g., cached loads and uncached stores).

To further reduce the protocol processor's occupancy, we explore the use of dual protocol cores per node. However, the L1 instruction and data caches are not replicated and are shared among the cores. Therefore, the designs with two protocol cores do not have a high area overhead. We modify the memory controller's request scheduler to guarantee mutual exclusion among addresses being handled by the two protocol processors simultaneously. The critical sections in the protocol handlers containing the compression and decompression algorithms are appropriately guarded by high-throughput test-and-set locks. To further improve the concurrency in protocol processing, we explore out-of-order network message scheduling in the memory controller. Usually, the network interface (NI) queues are FIFO and it may happen that in a cycle none of the requests at the heads of these queues can be scheduled due to address conflicts with the requests currently being handled by the protocol processors on a node. Out-of-order scheduling has the potential to improve the utilization of the protocol processors by considering all the requests (as opposed to just the heads) in the queues.

Finally, before closing this discussion, we would like to point out two major implementation issues that are left open and need to be addressed in future efforts. Enabling memory compression requires significant changes in the operating system. The page replacement and migration algorithms need to be modified so that these become compression-aware. Second, relocation of compressed memory blocks leads to fragmentation in the physical frames. Efficient memory compaction algorithms would be needed to fully utilize the potential of compressed main memory.

5. Simulation Results

This section presents our simulation results. We show results only for the best *maxCsz* of 48 bytes decided through simulation. In all the results we show the L2 cache block compression achieved by each of the seven applications along with the experienced slowdown in execution time. For a particular L2 cache block, the achieved compression is computed as saved bytes divided by the size of an L2 cache block i.e. 128 bytes (presented as a percentage). First, this is computed on average for each unique written back L2 cache block. Note that the same L2 cache block can be written back multiple times and each time a different compression ratio may be achieved. So an average is needed. Finally, this number is averaged over all unique written back L2 cache blocks. In summary, this is precisely the percent saved memory space (we include the overhead of storing the 32-bit header). The slowdown is computed as the ratio of the execution time with compression enabled to the baseline execution time on 16 nodes.

Figure 5 shows the achieved storage saving and execution slowdown for all the seven applications running on a 16-node DSM multiprocessor. Barnes and Water achieve excellent compression (77% and 70% savings) while suffering from at most 10% increase in execution time. Barnes enjoys a significant amount of memory access reduction due to the presence of a large number of completely zero cache blocks. This helps us nullify a sizable proportion of compression overhead. LU and Ocean achieve mediocre compression (18% and 26% savings). FFT, FFTW, and Radix-Sort are not compressible at all. Radix-Sort suffers from a 54% increase in execution time due to lost cycles in futile compression activity. Initialization with random data values is the main reason why these three applications fail to deliver any compression at all.

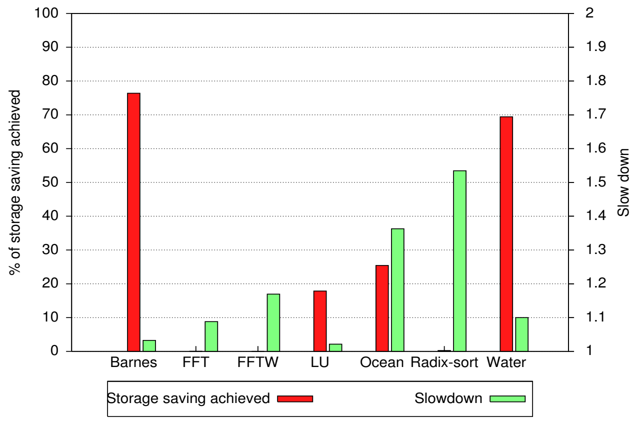

Figure 5. Performance with one PP per node

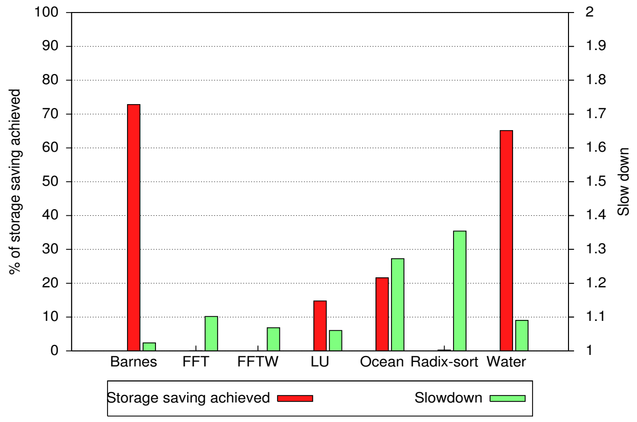

Figure 6. Performance with two PPs per node

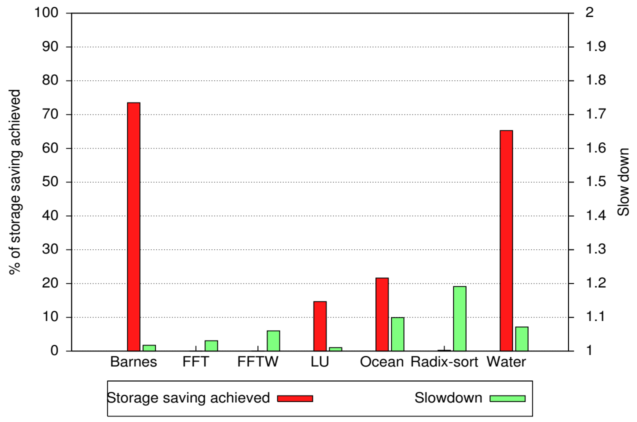

Figure 7. Performance with two PPs and OOO NI scheduling

Figure 6 shows the results after incorporating a second protocol core per node. While the achieved compression remains largely unchanged, the slowdown factors of FFTW, Ocean, and Radix-Sort have gone down dramatically. Radix-Sort now experiences a 36% increase in execution time. Note that the baseline system (without compression enabled) also employs two protocol processors per node.

Figure 7 shows dramatic improvement in performance across the board after employing out-of-order NI message scheduling. Compared to the baseline system Barnes, FFT, and LU suffer from negligible slowdown, while FFTW, Ocean, and Water suffer from at most 10% increase in execution time. Radix-Sort experiences only a 20% increase in execution time.

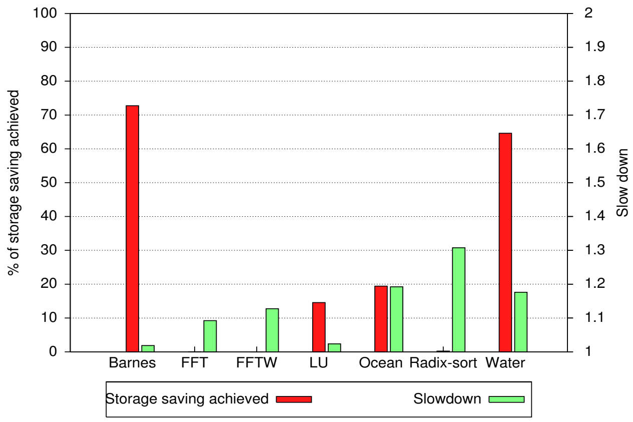

Figure 8. Performance with two PPs, OOO NI scheduling, and cached load/store to compression/decompression buffer space

Figure 8 shows the results after implementing cached load/store to the compression/decompression buffer space. However, performance of this optimization is worse compared to the design without it. Only Barnes and LU remain unaffected, while Radix-Sort now experiences a 31% increase in execution time compared to the baseline system. As already pointed out in the last section, flushing of dirty buffers from protocol core's L1 data cache keeps the datapath occupied leading to slowdown.

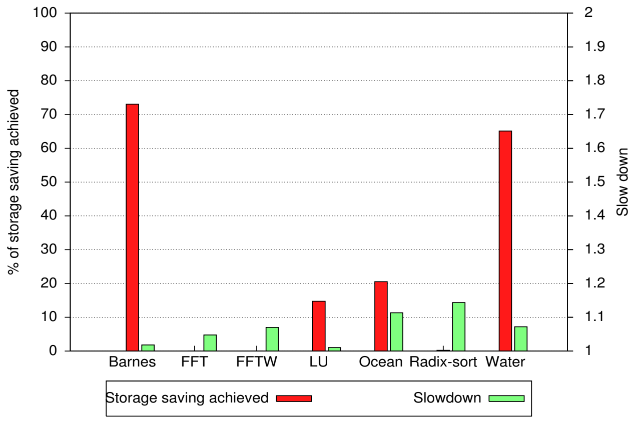

Figure 9. Performance with two PPs, OOO NI scheduling, and cached loads to compression/decompression buffer space

Figure 9 presents the results of our best design which incorporates two protocol cores per node, out-of-order NI message scheduling, and cached loads to the compression/decompression buffer area. Interestingly, FFT, FFTW, and Ocean suffer from a slight performance degradation when compared to the design without cached loads to the buffer area (compare Figure 7 and Figure 9). We found that for these three applications, the majority of the decisions to store a cache block uncompressed are based on the speculation of the first 64 bits of the block. While reading the first 64 bits from a buffer through a single uncached access requires only two cycles, loading the entire 128-byte buffer into cache before the first load can be executed requires 16 cycles. Therefore, with cached loads, a load to the first 64 bits sees this 16-cycle latency resulting in degraded performance for applications that carry out successful speculations based on the first 64 bits. On the other hand, Radix-Sort suffers from a large number of aborted compressions due to the chosen value of *maxCsz* (this value was chosen to offer good average performance across the board) and therefore, shortening the latency of these futile compression attempts by executing pipelined cached loads to the buffer area helps improve performance (note that once the buffer is brought into the cache, every cycle a load can be issued, unlike the uncached loads). Overall, this design achieves a storage saving of 73%, 16%, 21%, and 66% respectively for Barnes, LU, Ocean, and Water. The increases in execution times of these applications are 2%, 1%, 11%, and 8%, respectively. FFT, FFTW, and Radix-Sort do not benefit from any compression, but suffer from 5%, 7%, and 15% increase in execution time, respectively. Overall, while the maximum performance loss in a 16-node DSM multiprocessor employing our compressed memory is only 15%, four of the seven applications achieve reasonable compression. The remaining three applications that fail to achieve any compression initialize the data points with random values, thereby increasing the entropy of the cache block contents dramatically. It is unlikely that any compression algorithm would be able to achieve good compression ratio for these three applications.

5.1 Analyzing the Protocol Processing Overhead

In this section, we try to further understand the additional overhead of protocol processing when the baseline directory protocol is extended with compression/decompression algorithms. We start by analyzing the memory stall cycles experienced by the applications. These cycles are counted at the commit stage of the main core pipeline whenever the re-order buffer (ROB) is blocked with an outstanding load/store instruction at the head. Note that following the design of the MIPS R10000 [33] our main core implements sequential consistency and therefore, outstanding stores remain in the ROB until completed. Figure 10 shows the memory stall cycles of the compression-enabled architectures relative to the corresponding baseline architectures without compression. For each application we show a set of five bars representing the five design options discussed in the last section. "CachedRW" denotes the design with cached load/store of the compression/decompression buffer enabled, while "CachedR" denotes the design with only cached load of the compression/decompression buffer space enabled. We observe that the design with either out-of-order NI scheduling or cached loads to compression/decompression buffer space achieves the lowest extra memory stall cycles compared to the baseline architecture. For the best design this overhead ranges from 5% (Barnes and LU) to as large as 28% (Radix-Sort). In all applications, except FFT, LU, and Water, introduction of a second protocol core ("Two PPs") reduces the memory stall overhead significantly. In FFT, LU, and Water we found that the extra synchronization instruction overhead in the concurrent directory protocol running on the "Two PPs" configuration outweighs the benefits. This point is further explained below. Nonetheless, enabling both cached loads and stores to the compression/decompression buffer space significantly increases the memory stall cycles across the board. Finally, as expected, we observe that the trends presented in these results closely track the performance trends discussed in the last section.

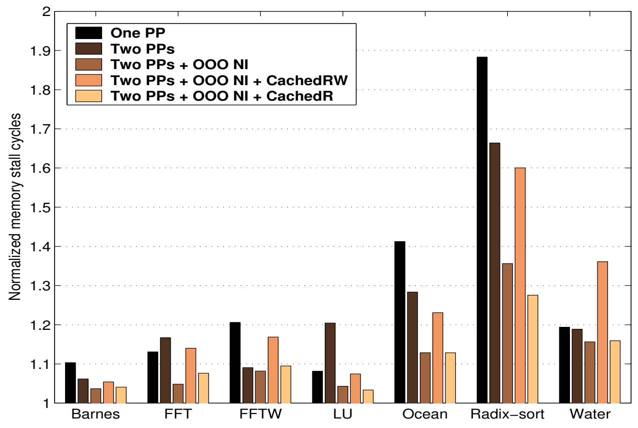

Figure 10. Memory stall cycles of the compression-aware architectures relative to the baseline architectures

To further understand the origin of the memory stall cycle overhead, we next focus on the average busy cycle count of the protocol core. This number is computed by taking an average of the busy cycles of all the 16 or 32 protocol cores (one or two per node). Figure 11 shows the average protocol core busy cycles for all the five design options relative to the corresponding baseline architectures. As expected, we find that the trends in memory stall cycle overhead are closely reflected in the trends of the protocol core busy cycle overhead. The best compression-aware design introduces 40% (Water) to as large as three times (Radix-Sort) extra protocol core busy cycles. We observe that even though for some of the applications this overhead is extremely high, the corresponding memory stall cycle overhead is not that significant. For example, Barnes suffers from a 2.8 times protocol core busy cycle overhead in the best design, while the corresponding memory stall cycle overhead is only 5%. The main reason for this seemingly anomalous result is that in some of the applications even though the protocol core busy cycle overhead is high, the protocol core is busy for an extremely small percentage of the total execution time. Barnes and LU are the best examples of such applications.

In Table 3 we present the root cause of such a high protocol core busy cycle overhead when compression is enabled. This table compares the dynamic instruction count (M denotes million) of the compression-aware protocol with that of the baseline protocol. This count is the aggregate of all the instructions executed by all the 16 protocol cores (one per node). Note that these counts do not include the extra synchronization instruction overhead that would be needed when executing on the "Two PPs" configuration. Further, out-of-order NI scheduling and caching buffer spaces do not alter this count much because these are largely hardware enhancements. This table clearly brings out the high overhead of the compression algorithm. In Barnes the dynamic instruction count increases by almost seven times, but many of these do not affect the critical path of execution. On the other hand, in Radix-Sort although the protocol instruction count increases by roughly a factor of 3.5, almost all of these fall in the critical path, thereby affecting the memory stall cycles significantly. Ocean has a large number of L2 cache misses and they translate into a large protocol instruction count.

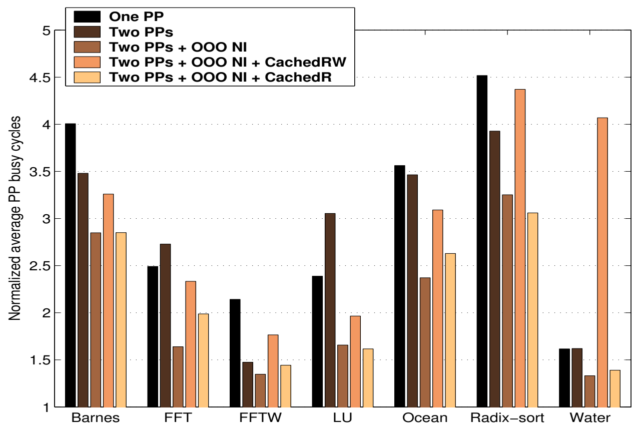

Figure 11. Average protocol processor busy cycles of the compression-aware architectures relative to the baseline architectures

Finally, to guage how much of the protocol processing overhead can be hidden under the memory access latency, in Table 4 we present the average occupancy of a protocol handler with and without compression. We also include the occupancy numbers when the protocol is augmented with proper synchronization instructions (test-and-set) to be used with two protocol processors. We observe that with compression enabled, the average handler occupancy increases significantly for all applications except Water. Radix-Sort has the maximum handler occupancy of 36.9 ns. However, the most interesting observation is that when compared to the fastest memory access latency of 40 ns (in case of a page hit as shown in Table 1), all these occupancy numbers are less than that. Even though we can hide the handler occupancy under the memory access latency, we need to remember that a bigger handler occupancy will in any case introduce contention in the interface queues connecting the protocol core and the memory controller, thereby lengthening the waiting time of the outstanding requests. This effect quickly builds up especially when the protocol requests are bursty leading to an increase in memory stall cycles seen by the applications. Radix-Sort is known to have bursty writes in the histogram permutation phase and that clearly

Table 3. Dynamic instructions executed by protocol processor

Config.	Barnes	FFT	FFTW	LU	Ocean	Radix-Sort	Water
Without compression	29.1 M	82.7 M	177.8 M	11.4 M	376.6 M	24.7 M	62.4 M
With compression	215.5 M	185.6 M	417.6 M	29.2 M	1553.5 M	87.0 M	137.3 M

Table 4. Average protocol handler occupancy

Config.	Barnes	FFT	FFTW	LU	Ocean	Radix-Sort	Water
Without compression one PP	7.5 ns	6.7 ns	10.5 ns	6.3 ns	6.7 ns	8.1 ns	5.5 ns
With compression one PP	31.9 ns	16.7 ns	22.7 ns	14.8 ns	24.1 ns	36.9 ns	8.8 ns
Without compression two PPs	8.4 ns	7.7 ns	13.3 ns	6.6 ns	7.0 ns	8.3 ns	5.6 ns
With compression two PPs	31.9 ns	26.7 ns	20.9 ns	24.4 ns	33.1 ns	45.6 ns	9.8 ns

explains why it is the most affected application. Nonetheless, the handler occupancy numbers are quite encouraging for most of the applications and they will improve compared to the memory access latency as the gap between processing speed and memory speed widens in future. Finally, we observe that the average occupancy of the handlers running on two PP configuration is higher than that in the one PP protocol in most of the applications. The synchronization overhead is the main reason for this. We found that in FFT and LU, this overhead outweighs all benefits of concurrent handler execution and increases the memory stall cycles, as already shown in Figure 10.

5.2 Discussion

One needs to be careful when evaluating a memory compression technique. The trade-off here is usually between the execution time overhead and the compressed memory size. Complex algorithms achieve better compression ratio while running the risk of degrading the performance. On the other hand, energy consumption is increasingly becoming a first class citizen in system design. In this section we develop a useful relationship involving the energy-delay product as the metric for evaluating compression techniques. Let the execution time without compression be τ and with compression be $(1+\alpha)\tau$ for some non-negative α. Before compression is enabled let the average power consumption be P and therefore, the total energy is $P\tau$. After compression is enabled extra energy is consumed due to increased execution time and increased activity of the protocol core. Assuming that the average energy per instruction of the protocol core is EPI_{PP} and that it executes extra N instructions to achieve compression, the total energy overhead coming from the protocol core is $EPI_{PP}N$. The energy impact of memory compression can be evaluated in two ways. One effect could be that due to smaller amount of memory needed, dynamic energy per memory access is reduced (it is natural to assume that dynamic energy per memory access is proportional to the size of the memory per module [25]). The second effect could be that due to bigger amount of effective memory available, less frequent accesses are made to the next level of memory (e.g., hard disk) leading to lower energy consumption. Nonetheless, in the following analysis we assume that the energy saved due to compression is $\gamma\mathcal{E}$ where a fraction γ of the original memory space is saved and $\mathcal{E}$ is the energy consumption by the memory modules without compression. Therefore, total energy consumption after compression is $(1+\alpha)P\tau + EPI_{PP}N - \gamma\mathcal{E}$ and hence, the energy-delay product is $(1+\alpha)^2P\tau^2 + (1+\alpha)\tau EPI_{PP}N - (1+\alpha)\tau\gamma\mathcal{E}$. Without compression, energy-delay product is $P\tau^2$. Therefore, memory compression is beneficial if $(1+\alpha)^2P\tau^2 + (1+\alpha)\tau EPI_{PP}N - (1+\alpha)\tau\gamma\mathcal{E} < P\tau^2$. On simplification, we get $\gamma > (1+\alpha-\frac{1}{1+\alpha})\frac{P\tau}{\mathcal{E}} + \frac{EPI_{PP}}{\mathcal{E}}N$.

This inequality is extremely useful in determining the minimum required memory saving due to compression so that the resulting design is energy-delay effective. For complex compression algorithms, both α and N will increase leaving the other quantities unchanged. This will raise the required minimum compression ratio, as expected. We observe that the fraction $\frac{EPI_{PP}}{\mathcal{E}}$ is going to be extremely small, given the simplicity of the protocol core. From our experience of preliminary power simulations at 65 nm we expect EPI_{PP} to be tens of pico Joules. Therefore, $\frac{EPI_{PP}}{\mathcal{E}}$ is expected to be of the order of 10^{-9} or even less depending on SDRAM power consumption and activity factors. Even though this fraction is multiplied by the extra dynamic instruction count of the protocol core, we do not expect the overall product to be large. On the other hand $\frac{P\tau}{\mathcal{E}}$ is going to be bigger than one and therefore, the first term of the inequality is expected to have more influence on the minimum required γ. However, it would be wrong to conclude that we can design complex compression algorithms and still end up not affecting the minimum required γ because the protocol core related term in the inequality does not have much influence. We need to note that the two terms in the inequality are not independent. The value of α is expected to be a monotonically increasing function of N, although the shape of the function may be quite complex and may vary from application to application. We recommend that the designers of memory compression techniques use this inequality to pick the appropriate design. Currently, we do not have a good DDR2 SDRAM power estimator integrated with our simulator. In future we will use this inequality to evaluate the five design options presented in this paper in terms of the energy-delay product.

6. Conclusions

In this paper, we have explored the performance potential of a simple hybrid of frequent pattern and zero run-based memory block compression/decompression scheme integrated with the directory-based cache coherence protocol in DSM multiprocessors. Our design exploits the flexibility of on-die protocol cores, thereby obviating the need for any extra hardware while enabling on-the-fly cache block compression and decompression in the home memory controller. Our results highlight the importance of coherence throughput in the end-performance of such systems. Even when the protocol processor is clocked at half the frequency of the main processor we find that two protocol processors per node with out-of-order NI message scheduling are needed to keep the performance overhead acceptable. Caching of compression/decompression buffers also turns out to be critical for performance. Although cached loads to the buffers are helpful, cached stores hurt performance due to subsequent expensive flushes from the protocol core's L1 data cache. Overall, our best design suffers from at most 15% increased execution time while delivering 16% to 73% memory block storage saving for four out of seven scientific computing workloads.

References

[1] B. Abali et al. Performance of Hardware Compressed Main Memory. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 73–81, January 2001.

[2] A. R. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223, June 2004.

[3] A. R. Alameldeen and D. A. Wood. Frequent Pattern Compression: A Significance-based Compression Scheme for L2 Caches. *Technical Report 1500*, Department of Computer Science, University of Wisconsin-Madison, April 2004.

[4] L. A. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282–293, June 2000.

[5] L. Benini et al. An Adaptive Data Compression Scheme for Memory Traffic Minimization in Processor-based Systems. In *IEEE International Symposium on Circuits and Systems*, May 2002.

[6] L. Benini et al. Hardware-assisted Data Compression for Energy Minimization in Systems with Embedded Processors. In *Proceedings of the Design, Automation, and Test in Europe*, pages 449–455, March 2002.

[7] M. Chaudhuri et al. Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation. In *IEEE Transactions on Computers*, **52**(7): 862–880, July 2003.

[8] D. Citron and L. Rudolph. Creating a Wider Bus using Caching Techniques. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 90–99, January 1995.

[9] Z. Cvetanovic. Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 218–228, June 2003.

[10] R. S. deCastro, A. P. doLango, and D. daSilva. Adaptive Compressed Caching: Design and Implementation. In *Proceedings of the 15th Symposium on Computer Architecture and High-Performance Computing*, pages 10–18, November 2003.

[11] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proceedings of the Winter USENIX Conference*, pages 519–529, January 1993.

[12] M. Ekman and P. Stenstrom. A Robust Main Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, June 2005.

[13] P. A. Franaszek, J. T. Robinson, and J. Thomas. Parallel Compression with Cooperative Dictionary Construction. In *Proceedings of the 6th Data Compression Conference*, pages 200–209, March/April 1996.

[14] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, May 1998.

[15] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[16] E. G. Hallnor and S. K. Reinhardt. A Unified Compressed Memory Hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201–212, February 2005.

[17] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[18] R. Kalla, B. Sinharoy, J. M. Tendler. IBM Power5 Chip: A Dual-core Multithreaded Processor. In *IEEE Micro*, **24**(2): 40–47, March-April 2004.

[19] C. N. Keltcher et al. The AMD Opteron Processor for Multiprocessor Servers. In *IEEE Micro* **23**(2):66–76, March-April 2003.

[20] M. Kjelsø, M. Gooch, and S. Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd Euromicro Conference*, pages 423–430, September 1996.

[21] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[22] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[23] J-S. Lee, W-K. Hong, and S-D. Kim. Design and Evaluation of a Selective Compressed Memory System. In *Proceedings of the International Conference on Computer Design*, pages 184–191, October 1999.

[24] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–317, May 1996.

[25] Micron Technical Note. Calculating Memory System Power for DDR2. *Micron TN-4704*, available at http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf.

[26] MIPS/SGI. R10000 Microprocessor User's Manual.

[27] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, Vol. 1, pages 1–10, August 1995.

[28] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34–43, May 1996.

[29] Sun Microsystems. An Overview of UltraSPARC III Cu. White Paper, September 2003. Available at http://www.sun.com/processors/whitepapers/USIIICuoverview.pdf.

[30] Sun Microsystems. UltraSPARC IV Processor Architecture Overview. White Paper, February 2004. Available at http://www.sun.com/processors/whitepapers/ us4_whitepaper.pdf.

[31] R. B. Tremaine et al. IBM Memory Expansion Technology (MXT). In *IBM Journal of Research and Development*, **45**(2): 271–285, March 2001.

[32] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[33] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, **16**(2):28–40, April 1996.

[34] Y. Zhang and R. Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction*, pages 14–28, April 2002.