A Unified Compressed Memory Hierarchy

Erik G. Hallnor and Steven K. Reinhardt
Advanced Computer Architecture Laboratory
EECS Dept., University of Michigan
{ehallnor, stever} @eecs.umich.edu

Abstract

The memory system's large and growing contribution to system performance motivates more aggressive approaches to improving its efficiency. We propose and analyze a memory hierarchy that uses a unified compression scheme encompassing the last-level on-chip cache, the off-chip memory channel, and off-chip main memory. This scheme simultaneously increases the effective on-chip cache capacity, off-chip bandwidth, and main memory size, while avoiding compression and decompression overheads between levels.

Simulations of the SPEC CPU2000 benchmarks using a 1MB cache and 128-byte blocks show an average speedup of 19%, while degrading performance by no more than 5%. The combined scheme achieves a peak improvement of 292%, compared to 165% and 83% for cache or bus compression alone. The compressed system generally provides even better performance as the block size is increased to 512 bytes.

1. Introduction

A computer system's memory hierarchy has a large impact on its overall performance. With each technology generation, increases in microprocessor execution rates outpace improvements in latency and bandwidth of both DRAM and disks, further increasing the importance of the memory hierarchy.

The latency, bandwidth, and capacity of each level in a memory hierarchy are key determinants of its performance. Levels closest to the CPU focus on providing data with very low latency, constraining their physical size. Moving further from the CPU, the latency cost of missing in a particular level of a hierarchy grows rapidly: from tens of processor cycles for a first level cache miss, to hundreds of cycles for a main memory access, to millions of cycles for a disk access. Levels further from the CPU thus emphasize high capacity, to minimize the number of expensive accesses to more distant levels. The capacities of the last level of on-chip cache and main memory are constrained primarily by economic factors, i.e., the amount customers are willing to pay. These cost factors manifest themselves as die area constraints for on-chip cache capacity and chip count for DRAM capacity. Bandwidth is an important factor at all levels of the hierarchy, particularly because many of the techniques employed by processor designers to deal with memory latency—e.g., prefetching, multithreaded CPUs, and chip multiprocessors—result in increased bandwidth demands. The primary constraint on off-chip bandwidth is also economic, in this case manifested as pin count (package cost).

Data compression is a relatively low-cost avenue for increasing both capacity and bandwidth within a given hardware budget. Removing redundancy allows information to be stored in fewer physical bits and to be transmitted in fewer cycles across a fixed number of wires. This paper investigates the performance benefit of applying a unified compression scheme across multiple levels of the memory hierarchy, spanning both the last level of on-chip cache and off-chip DRAM main memory.[1] Our approach combines the advantages of using compression at multiple points in the hierarchy; specifically, it:

1. increases effective on-chip cache capacity, reducing the frequency of off-chip memory accesses;
2. increases effective off-chip bandwidth, accelerating transfers for the remaining off-chip accesses; and
3. increases effective main memory capacity, reducing the frequency of disk accesses.

While compression has been proposed for each of these components in isolation [1, 2, 6, 15], our unified approach achieves these benefits without the expense of compression and decompression at every stage. We use a single compression algorithm and block size so that data can be stored in and transferred between the last-level cache and main memory in compressed form without additional compression or decompression overhead. These compression and decompression operations take place only between the last level of on-chip cache and

[1] Compression often yields power benefits as well [4][20][22]; however, these are beyond the scope of this paper.

Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)
1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

caches closer to the CPU, and between main memory and I/O devices.

To manage the variable block storage required for compression, our proposed cache design uses an indirect cache structure similar to the Indirect Index Cache (IIC) [10]. We modify the original IIC to associate multiple data subblock pointers with each tag. Although each tag has enough subblock pointers to address a full uncompressed block, a particular cache block is assigned only as many subblock frames as are needed to store the block in compressed form. Thanks to the indirection-based full associativity of the IIC, the subblock frames saved by compressing one cache block can be used to store data for any other block. We call our design the Indirect Index Cache with Compression, or IIC-C.

We find that for a few applications, the additional cache capacity afforded by compression does not compensate for the decompression latency [2]. We extend the generational replacement algorithm originally proposed for the IIC to keep the most recently accessed cache blocks uncompressed, partially mitigating the negative impact of compression in these cases.

Our compressed main-memory organization and our compression algorithm are taken from IBM's Memory Expansion Technology (MXT) system [1]. However, because our main memory system delivers data in compressed form to the on-chip cache, we avoid the large uncompressed off-chip cache needed by MXT.

We evaluate our design using the SPEC CPU2000 benchmark suite. Our baseline system uses a 1MB last-level cache to compensate for the limited data set sizes of these benchmarks. Because of the difficulty in simulating realistic workloads that stress the disk subsystem, we do not consider the additional cost/performance advantage our system derives from compressed main memory [1, 3]. Focusing solely on the benefit of compressing the last-level cache and off-chip memory channel, we find that:

- Our compressed memory system outperforms a traditional memory hierarchy by an average of 19% using 128-byte blocks. Hierarchies that compress only the last-level cache or off-chip memory bus provide improvements of 13% and 7%, respectively. Only a few benchmarks see performance degradation, of at most 5%.
- Unlike the traditional hierarchy, the performance of our compressed hierarchy improves as the block size increases beyond 128 bytes, peaking at 512-byte blocks. At this block size, our 1 MB compressed cache has an average effective capacity of 1.35 MB, and outperforms the uncompressed 128B-block hierarchy by 20% on average, with 27% less bus traffic. Twelve of the twenty-six benchmarks show significantly higher performance (up to 176%), while only three show noticeable degradation due to the larger block size (15% in the worst case).
- As memory latencies increase, compression improves performance by an even greater margin.
- Cache compression provides significant benefits only to the extent a program's working set fits in the cache when compressed but not when uncompressed. While, unsurprisingly, increasing the cache size eliminates the benefit of compression on some benchmarks, it also increases the benefit on others. There are individual CPU2000 benchmarks that benefit dramatically from compression at 1 MB, 2 MB, and 4 MB cache sizes.

Overall, this paper shows considerable potential for a unified approach to compression across the on-chip and off-chip levels of the memory hierarchy for a single-threaded uniprocessor on a limited set of benchmarks. We believe that the combination of larger datasets (e.g., from commercial or multimedia applications) and increased cache capacity and memory bandwidth contention from multithreaded CPUs and chip multiprocessors will make these techniques even more beneficial in the future.

The rest of this paper is organized as follows. Section 2 presents previous work in compressed memory systems. Our system design is presented in Section 3 and evaluated in Section 4. We conclude in Section 5.

2. Previous Work

In this section we present on data compression—compressed caches, compressed bus transfers, and compressed main memory systems—in turn. Compression has also been applied to instructions in the embedded world for reducing memory requirements [13, 16, 17]. This application differs significantly from data compression in that no on-line compression is required; code is compressed typically at compile time and is read-only during execution.

2.1. Cache compression

Recently, a number of designs for cache data compression have been proposed. Most of these schemes have applied compression solely for power/energy savings rather than performance. This emphasis allows the use of more conventional cache structures, where unused storage cells and wires provide a benefit simply by not consuming power.

The Frequent Value Cache (FVC) [23, 22] replaces the top N frequently used 32-bit values with $\log_2 N$ bits. When built as a separate structure [23], the FVC can increase cache size if an entire cache block is made up of frequent values. However, the probability of this situation occurring decreases with larger cache blocks, making this scheme most effective at the L1 level. The FVC can also be utilized to decrease cache energy [22]. The encoded frequent values are stored in the first bits of the cache line. If the value is marked as compressed, only these bits

IEEE COMPUTER SOCIETY

are read, saving the energy of reading the entire line. This power-saving variation does not increase the effective size of the cache.

Another scheme to reduce cache energy is Dynamic Zero Compression (DZC) [20]. In this scheme, each zero-valued byte is represented by a single bit. This encoding provides an 8 to 1 reduction in the number of bit lines that need to be driven, thus saving energy.

Kim et al. [12] exploit small sign-extended values by compressing the upper portion of a word to a single bit if it is all 1s or all 0s. These compressed sign bits, along with the lower portions of the words, are stored in one cache bank. This bank is accessed first; the second bank containing the uncompressed high-order bits is accessed only if necessary, again saving bit-line reads. They also propose adding a second tag per line to allow storage of a separate block in the second bank when possible.

Fewer researchers have investigated the use of compression to increase effective cache size. Alameldeen and Wood [2] also take advantage of small values to create a compression algorithm called frequent pattern compression (FPC). However, as in our work, they target increased capacity and performance rather than only power savings. Their cache uses indirection between tags and data, like the IIC, so that a single set can store either 4 uncompressed blocks, or up to 8 compressed blocks. This allows them to increase the effective size of the cache with only the extra tag storage. However, their design has the constraint that the data needs to be stored in the same order as the tags appear. When any block in the set (but the last) changes size, the entire set's data need to be reordered. This overhead is potentially expensive. Also, any bytes freed by compression can only be used in the associated set, leading to wasted storage. They also observed that the decompression latency can negatively impact some applications and developed an adaptive scheme to selectively store blocks uncompressed.

The Selective Compressed Memory System (SCMS) [15] is closest to ours in spirit, combining a compressed L2 cache with a compressed memory and using a general compression algorithm. Cache lines are compressed in pairs (where the line address is the same except for the low-order bit). If both lines compress by 50% or more, they are stored in a single cache line, freeing a cache line in an adjacent set. However, it becomes necessary to check two sets for a potential hit on every access. SCMS also fails to take advantage of lines that compress by less than 50%, and provides at most a 2 to 1 advantage even when lines compress by more than 50%. Our design allows blocks to be compressed to 75%, 50%, or 25% of their original size, and our results show a significant benefit from this flexibility (see Section 4.3). We also provide an execution-driven evaluation of compression performance on the entire SPEC CPU2000 suite, while the SCMS work presented trace-driven evaluation of a subset of the SPEC95 benchmarks.

2.2. Bus Compression

Bus compression has been proposed in the past to support wider data on narrower busses [6, 8]. These schemes use the property that addresses and data often use the same high order bits repeatedly. A small cache of recently used high order bits is kept on each end of the bus, so the bus often only needs to carry the low order bits and an index into this cache. More recently, a similar scheme [4] has been proposed for reducing the energy consumed by the bus.

2.3. Memory Compression

There have been several designs that use compression in main memory to increase effective DRAM capacity, reducing the number of disk accesses. Most schemes [14, 18, 21] set up a portion of physical memory to store compressed pages after they are evicted from the uncompressed portion. Virtual memory support is used to manage this partition flexibly. Compression and decompression may be done in software on the CPU or with a dedicated hardware assist.

IBM's Memory Expansion Technology (MXT) [1] differs in that all main-memory data is stored in compressed form. A hardware engine built into the memory controller manages compression and decompression transparently to software. To reduce decompression latency for misses in the on-chip caches, the MXT memory controller includes a large off-chip uncompressed cache.

MXT maps the "real" addresses generated by the processor to the physical addresses of the compressed memory using a sector translation table (STT). Each entry in the STT maps a 1KB real address area. An entry consists of 4 physical address that each point to a 256B sector. A block is typically stored in one to four of the 256B sectors depending on its compressibility. If the 1KB block compresses to less than 120 bits, termed a "trivial" value, it is stored in the STT entry itself. While some space is lost due to internal fragmentation in the 256B sectors, most of the space saved by compression can be accessed by the STT. We adopt this design when creating our compressed cache.

3. A Compressed Memory Hierarchy Design

Our compressed memory hierarchy consists of two parts: a new cache design, the Indirect Index Cache with Compression (IIC-C), and a compressed main memory, similar to IBM's Memory Expansion Technology (MXT). We describe our design below.

IEEE COMPUTER SOCIETY

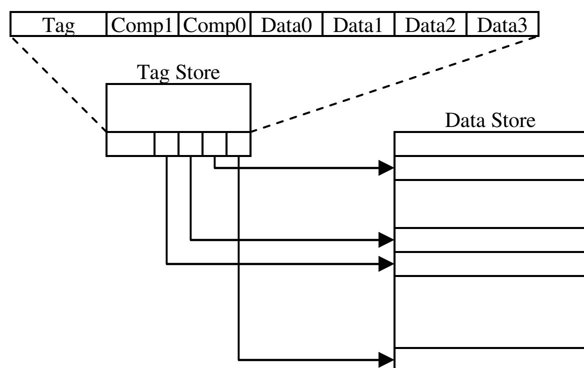

Figure 1. The Indirect Index Cache w/Compression

3.1. The IIC-C Hardware

The IIC-C, shown in Figure 1, is a fully-associative cache (based on the IIC [10]) that consists of a data array, which contains the cache blocks, and a tag store, which contains the tags for these blocks. Each tag entry holds pointers to its associated data blocks. If the cache block is compressible, some of these subblock pointers will not be used. The associated data array entries will not be allocated, leaving additional storage for other blocks. Because each subblock pointer can index anywhere in the data array, these unallocated entries can be addressed by any other tag entry. In addition to the standard status bits, each tag entry contains bits relating to the compression state of the block. These bits state the number of subblocks currently used; they read 0 if the compressed value is stored completely in the subblock pointers.

To compensate for the added latency of decompressing blocks, both MXT and SCMS use intermediate structures to cache decompressed data. Our design leverages the trend in current high-performance processors to have three levels of cache on-chip [11]. With the L3 storing data in a compressed form, the L2 serves as a buffer of decompressed cache blocks, reducing the impact of the decompression latency. In addition to the buffering of uncompressed data provided by the L2, we developed an adaptive compression policy in the L3 to reduce the decompression penalty even further.

The adaptive scheme builds on the IIC replacement algorithm, called Generational Replacement (Gen) [10]. Gen maintains prioritized FIFOs, or pools, of the blocks in the cache. Each block stays resident in a given pool for a set number of misses, when it will either move to a higher or lower priority depending on whether it has been referenced. In our adaptive scheme, a compressed block is stored back into the cache in decompressed form when it is accessed. When the block is moved out of its current Gen pool it is recompressed. This policy has the effect of balancing the number of misses against the number of compressed blocks. If the cache has more misses, blocks will move between pools more frequently and hence would be compressed more frequently, which should reduce the number of misses. Under this policy, if an application's working set fits entirely in the cache, i.e. there are no misses, then no blocks will move between pools and the data would remain uncompressed, removing the cost of decompression.

3.2. IIC-C Hardware Overhead

This design does introduce some hardware overheads. The compression/decompression engine takes up some die area, which increases linearly with block size, but this area is minimal when compared to a megabyte cache [3]. In addition to the basic IIC overheads [10], the IIC-C adds extra subblock pointers to the tag, increasing the size by 6-8 bytes per tag entry for 4 subblocks. Extra tag entries are also needed to index the extra space made available by compression; however, the base IIC already has more tags than cache lines to improve tag lookup performance [10]. We assume the IIC-C has twice the minimum number of tags needed, unchanged from the original IIC design. The total of these overheads comes to 124KB for a 1MB IIC-C with 128-byte blocks and 32-byte subblocks, and falls to 27KB at a 512-byte block size with 128-byte subblocks. Our results in Section 4.4 show that, on average, the benefits of compression significantly outweigh the benefits of increasing the capacity of a 1MB conventional cache by 10%.

3.3. Choosing a Compression Algorithm

Our choice of compression algorithm is driven by a number of factors. First, the algorithm needs to have an efficient hardware implementation. Second, the algorithm should have reasonable compression and decompression latencies. Third, the algorithm should be scalable to work effectively with both a large cache and main memory.

LZSS [9] meets all of the criteria. LZSS is a parallel version of LZ77 which can be implemented in hardware [3]. The speed of LZSS is dependent on the number of parallel compressors. Thus designers can increase the speed of compression/decompression by adding more hardware. The hardware requirement scales linearly with the degree of parallelism. LZSS also has the property that decompression is much faster than compression—by a factor of 4 in the MXT implementation [3]. This property is desirable with a write-back cache hierarchy because reads will outnumber writes to the L3 cache. Each block carries its own internal dictionary, simplifying communication between the IIC-C and main memory. Of course, other algorithms could be used, such as X-RL [15] or FVC [2], but we opted for LZSS since MXT provides both good performance and a real-world example of its implementation [3].

IEEE COMPUTER SOCIETY

Table 1. Simulation Parameters

Structure	Configuration
CPU	2 Ghz, 8-wide OOO, 256 entry IQ, 512 entry ROB
L1 Icache/ Dcache	Both: 16KB, 4-way assoc., 64B block size, 1 cycle latency Instr: 8 MSHRs, Data: 32 MSHRs
L2 Unified Cache	256KB, 8-way assoc., 128B block size, 12 cycle hit latency, 40 MSHRs
L3 Unified Cache	1MB, 128B to 1024B block size, 26 cycle hit latency, 40 MSHRs
Memory Bus	6.4 GB/sec, 400Mhz, 16B data path
Main Memory	150 cycle latency

3.4. Integrating the IIC-C with a Compressed Main Memory

To improve effective pin bandwidth and reduce compression/decompression overhead, we couple our compressed cache with a compressed main memory. We assume an MXT-like scheme, where we match the compression block size of main memory to the cache line size. This equivalence allows us to pass compressed data blocks directly across the bus. However, this feature has the side effect of making it impossible to send the critical word first. This penalty can be partially offset by overlapping decompression with transmission [15]; we do not include that optimization in this paper. We assume compression/decompression logic on the memory controller for decompressing and compressing DMA transfers to and from I/O devices.

4. Evaluation

We evaluate the unified compressed hierarchy using detailed execution-driven simulation. This section presents our simulation methodology and our results.

4.1. Methodology

We used the M5 simulator [5] to evaluate our design. We configure M5 to simulate an 8-wide out-of-order speculative Alpha processor with a detailed timing memory hierarchy. We based this cache hierarchy on the McKinley Itanium2 [11], but we increased the cycle delays to match a 2 GHz processor. For most of our runs, we limit the L3 cache size to 1 MB to compensate for the limited data set sizes of our benchmarks. (Section 4.3.3 reports the effects of increasing the L3 cache size.) Table 1 summarizes our baseline simulation parameters.

To keep the tag overhead manageable, we set the number of IIC-C subblocks within a cache line to four. To determine cache block compressibility, we run the actual program data block contents through the sequential LZ77 algorithm. For the range of block sizes we are looking at (128B and up), the sequential algorithm has similar compression performance to the parallel LZSS [9]. We assume hardware similar to the MXT implementation, where decompression is four times faster than compression. Our experiments use a compression latency of 32 cycles with the corresponding decompression latency of 8 cycles.

Our simulations use the 26 benchmarks from the SPEC CPU2000 benchmark suite. We run each simulation starting at the “early single” SimPoint [19] and run for 300M instructions.

We begin by analyzing the effects of compression on our baseline memory hierarchy at various block sizes. We then analyze the sensitivity of these results to other parameters such as cache size and memory latency. This section concludes with a broader perspective on the impact of a compressed hierarchy.

4.2. Compression Results

We first compare the average behavior of all 26 SPEC benchmarks on the baseline uncompressed system (an 8-way set-associative LRU L3 cache) to the compressed memory hierarchy. We varied the L3 block size (which is also the compression block size) from 128B to 1024B blocks. We assume the compression/decompression hardware scales with the block size to keep latencies constant. Figure 2 presents the average effective size of the L3 cache, the number of bytes transmitted across the memory bus, the number of misses, and the harmonic mean of the benchmark IPCs, all normalized to the uncompressed system with 128B blocks. With 128-byte blocks, cache compression increases effective capacity by 30%. The number of misses is reduced on average by 32%. Bus compression reduces the remaining traffic by another 40%, resulting in a traffic reduction of 52%. Consequently, mean IPC increases by 22%.

Increasing the block size brings in more unrequested data on each miss, increasing bus traffic and bus queuing delays. The cache also holds fewer unique blocks, increasing miss rates. Performance may increase or decrease, depending on whether the larger blocks prefetch useful data. For the uncompressed system, average performance is nearly flat out to 512B, but drops by 16% at 1KB blocks. Meanwhile, traffic increases by nearly a factor of two. On the other hand, larger blocks also provide more context for the compression algorithm, typically enabling higher compression ratios. As a result, the effective capacity of the compressed cache increases with block size, reaching 1.39 MB with 1KB blocks. The combination of greater capacity and more effective bus compression keeps bus traffic increases manageable as well. Though the compressed system has 35% more bus traffic at 1KB than at 128B, it is still 45% lower than the uncompressed system at 128B. As a result, the compressed system sees performance increases with larger block sizes out to 512B. At this point, average

IEEE COMPUTER SOCIETY

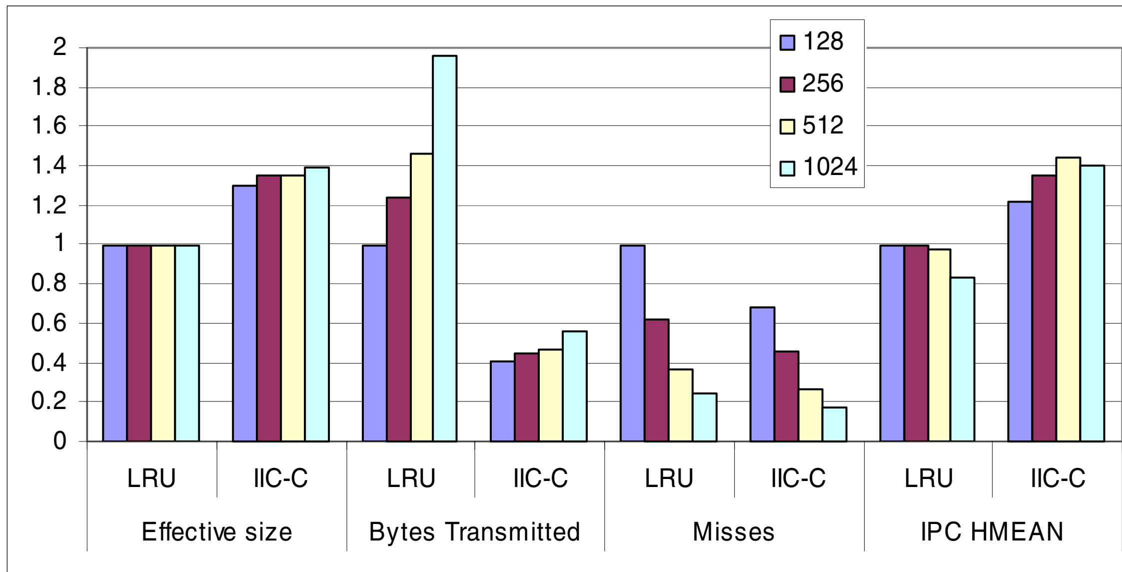

Figure 2. Performance of an uncompressed 8-way LRU cache and a compressed IIC-C hierarchy, normalized to 128B-block uncompressed LRU case.

performance is 44% higher than the best uncompressed system at 128B.

Table 2 presents detailed results for each benchmark at 128B and 512B block sizes (the best performing block size for the uncompressed and compressed systems, respectively). To further isolate the performance impact of each portion of our design, we tested two additional configurations. The first, denoted C--, uses compression only in the L3 cache (i.e., a stand-alone IIC-C with uncompressed bus and memory). In the second, -BM, we store and transmit data in a compressed form in main memory and across the bus, but store data uncompressed in a conventional L3 cache. We use CBM to denote the unified compressed hierarchy. We also simulated an uncompressed IIC to isolate the performance impact of the IIC's higher associativity and different replacement algorithm. The results in Table 2 indicate the absolute IPCs for the uncompressed LRU caches (shown in italics) and relative IPCs for the other configurations (relative to the LRU cache of the same block size).

The top 12 benchmarks in Table 2 (above the bold line) are those that see a significant impact (positive or negative) from compression. For space reasons, further results will report individual results only for these benchmarks. *However, in all cases the averages presented are over all 26 SPEC CPU2000 benchmarks, not just the twelve presented.* Table 3 presents per-benchmark results for misses, bus traffic, and effective cache size for the top 12 benchmarks in a format similar to Table 2.

4.2.1. Cache compression (C- -). As shown in Table 2, simply compressing the L3 cache improves performance at both block sizes. Cache compression achieves the largest gain (165%) on art at 128B blocks while the worst loss is 5% on facerec. On average, performance improves by an average of 13% over a traditional LRU cache, while increasing the average effective cache size by 30-35% (see Table 3).

Art shows large gains at 128B blocks because the IIC-C is able to eliminate two thirds of its 23M misses. It shows less relative improvement at 512B blocks, despite doubling the effective cache size to 2MB, due to the higher bandwidth demands of transmitting larger blocks across the uncompressed bus. In addition, the prefetching effect of the larger block size eliminates some of the misses in the uncompressed case that compression removed at smaller block size. Most other affected benchmarks see increasing benefit from compression as the block size increases. Mgrid sees a slight performance decrease with increasing block size, due mainly to pollution effects in the IIC.

4.2.2. Bus compression (-BM). In this configuration, we ship compressed data across the bus but store it uncompressed in the L3 cache. As can be seen in Table 2, bus compression alone is also a net performance win, though generally less so than cache compression. Art and lucas see large benefits at both block sizes due to their high memory bandwidth requirements. Galgel and mcf also show gains from bus compression when the larger 512-byte blocks make bandwidth more critical. Performance decreases in a few cases (at most 6%) due to the added latency of decompression and the inability of the compressed memory system decreases in a few cases (at most 6%) due to the added latency of decompression and the inability of the compressed memory system to send the critical word first.

IEEE COMPUTER SOCIETY

Table 2: IPCs of compressed memory hierarchy relative to uncompressed LRU cache of the same block size (numbers in italics are absolute IPCs). The top 12 benchmarks have the largest positive and negative changes.

	128-byte blocks					512-byte blocks				
			IIC-C					IIC-C		
	LRU	IIC	C--	-BM	CBM	LRU	IIC	C--	-BM	CBM
art	*0.27*	1.08	2.65	1.83	3.92	*0.21*	1.03	1.71	2.47	3.57
lcs	*1.44*	1.00	1.23	1.26	1.33	*0.85*	0.76	1.24	1.34	2.90
gal	*2.43*	1.01	1.60	1.10	1.60	*1.79*	1.21	1.71	1.37	1.79
mcf	*0.13*	1.00	1.11	1.03	1.14	*0.15*	1.03	1.28	1.40	1.69
twf	*0.87*	0.99	1.17	0.95	1.14	*0.83*	0.99	1.39	0.95	1.35
amp	*2.44*	1.53	1.53	1.52	1.53	*1.65*	1.22	1.36	1.23	1.35
bzp	*1.95*	1.01	1.02	1.02	1.03	*1.35*	1.00	1.17	1.06	1.23
prl	*0.64*	1.20	1.19	1.19	1.19	*0.62*	1.21	1.22	1.19	1.20
vpr	*1.39*	0.99	0.98	0.96	0.95	*1.05*	0.97	1.11	1.01	1.12
fac	*1.54*	0.98	0.95	0.99	0.97	*1.73*	1.00	1.05	1.01	1.05
mgd	*2.24*	0.97	0.99	0.97	0.99	*2.80*	0.95	0.96	0.97	0.96
gcc	*1.84*	0.99	0.96	0.97	0.95	*1.95*	0.99	0.98	0.98	0.97
apl	*1.36*	1.00	1.00	1.00	1.01	*1.33*	1.00	1.03	1.01	1.05
six	*4.78*	1.06	1.06	1.06	1.06	*4.88*	1.04	1.04	1.04	1.04
swm	*1.24*	0.98	0.98	1.02	1.01	*1.32*	0.98	0.98	1.03	1.04
aps	*2.15*	1.00	1.03	1.03	1.03	*2.17*	1.00	1.03	1.03	1.03
mes	*3.49*	1.00	1.01	1.00	1.01	*4.04*	1.00	1.01	1.02	1.02
vor	*1.76*	1.01	1.01	1.01	1.01	*1.74*	1.01	1.02	1.01	1.02
par	*1.35*	0.99	1.02	0.94	0.98	*1.63*	1.00	1.04	0.98	1.02
cft	*1.64*	1.00	1.01	1.00	1.01	*1.63*	1.00	1.01	1.00	1.02
gap	*3.09*	1.00	1.00	0.98	0.98	*3.60*	1.00	1.00	0.99	0.99
wup	*3.18*	1.00	1.01	1.00	1.01	*3.78*	1.00	0.99	1.02	0.99
fma	*2.60*	1.00	1.00	1.00	1.00	*2.60*	1.00	1.00	1.00	1.00
enc	*2.93*	1.00	1.00	1.00	1.00	*2.93*	1.00	1.00	1.00	1.00
gzr	*2.76*	1.00	1.00	1.00	1.00	*2.77*	1.00	1.00	1.00	1.00
eqk	*4.85*	1.00	1.00	1.00	1.00	*4.85*	1.00	1.00	1.00	1.00
avg		1.03	1.13	1.07	1.19		1.02	1.13	1.12	1.28

4.2.3. Combining cache and bus compression (CBM).

Combining the compressed cache with a similarly compressed main memory and transferring data in a compressed form further increases performance beyond either scheme alone. The maximum improvement of 292% is achieved by art at 128B blocks, due to a reduction of almost 15 million misses and bus bandwidth savings of 86% (67% from the reduced misses and 19% from bus compression). The average speedup over an uncompressed system with the same block size is 19% at 128-byte blocks, and increases to 28% with 512-byte blocks. In some cases, small losses due to compression overheads in the cache or bus only are compensated by gains in the other category. Unfortunately, the few benchmarks that lose performance with both C-- and -BM have these losses compounded with the combined scheme. However, the losses are modest (no more than 5%), and disappear for two of the four benchmarks at the larger block size.

Table 3. Number of misses, bytes transmitted, and effective cache size of compressed memory scheme (CBM) relative to a 1MB traditional LRU cache. Numbers in italics are absolute).

	128					512				
	LRU (abs.)		IIC-C (rel.)			LRU (abs.)		IIC-C (rel.)		
	bus bytes (millions)	misses (thousands)	eff. size	bus bytes	misses	bus bytes (millions)	misses (thousands)	eff. size	bus bytes	misses
art	*3,242.04*	*25,328.44*	1.78	0.33	0.14	*4,285.14*	*8,369.41*	1.99	0.59	0.16
lcs	*352.16*	*2,751.22*	1.99	1.00	0.19	*792.62*	*1,548.09*	1.99	1.00	0.08
gal	*178.44*	*1,394.03*	1.09	0.06	0.03	*333.38*	*651.14*	1.14	0.29	0.19
mcf	*2,753.07*	*21,508.35*	1.42	0.94	0.52	*3,580.17*	*6,992.52*	2.00	0.88	0.31
twf	*137.00*	*1,070.31*	1.30	0.47	0.31	*430.89*	*841.58*	1.48	0.18	0.08
amp	*110.81*	*865.72*	1.02	0.08	0.08	*258.92*	*505.70*	1.02	0.57	0.50
bzp	*74.56*	*582.52*	1.06	0.93	0.82	*292.11*	*570.52*	1.09	0.81	0.70
prl	*86.13*	*672.88*	1.20	0.28	0.19	*335.72*	*655.71*	1.28	0.32	0.21
vpr	*177.47*	*1,386.49*	1.14	1.03	0.78	*564.83*	*1,103.19*	1.31	0.90	0.56
fac	*158.45*	*1,237.92*	1.07	1.03	0.98	*183.95*	*359.28*	1.06	1.02	0.98
mgd	*144.00*	*1,124.97*	1.17	1.05	0.89	*166.97*	*326.12*	1.15	1.07	0.92
gcc	*12.49*	*97.58*	1.10	1.01	0.70	*14.79*	*28.89*	1.30	1.01	0.50
avg	*330.10*	*2,578.87*	1.30	0.78	0.48	*480.91*	*939.28*	1.35	0.78	0.43

4.3. Sensitivity Analysis

To characterize the impact our design decisions and future trends have on our results, we ran several sensitivity experiments. For these experiments, we use 512B blocks, since this size achieves the highest performance. We used an IIC with the same block size for our baseline to factor out block size and replacement effects. Again, we present results only for the twelve benchmarks listed above the bold line in Table 2, but the averages displayed are for all 26 SPEC benchmarks.

4.3.1. Adaptive Compression. Table 4 shows the performance of our adaptive compression algorithm (see Section 3.1) compared to a similar system that always compresses L3 cache data. The primary purpose of the adaptive scheme is to limit the negative performance impact of decompression latency for working sets that fit uncompressed in the L3 cache: the adaptive algorithm suffers a maximum 1% performance loss compared to a 12% performance hit on the non-adaptive cache. On average the adaptive scheme performs 2% better than the non-adaptive scheme.

4.3.2. Cache Size. Table 5 shows the relative performance of the IIC-C as we increase the cache size from 1MB to 4MB. As expected, for several benchmarks, the usefulness of compression decreases when the cache size grows larger than the working set size. For art, the 2MB IIC-C continues to greatly out perform the IIC, but at 4MB we show no improvement whatsoever. Twolf similarly ceases to show improvement at 2MB, while vpr goes from a performance gain to small loss. On the other hand, some benchmarks—mcf and mgrid in particular—see increasing benefit from compression as the cache becomes large enough to hold their entire working set in compressed form, though not in uncompressed form. Thus, though the average speedup for SPEC CPU2000 decreases as the cache size is increased past 1 MB, a significant benefit remains. Clearly the utility of compression at any particular cache size is a strong function of the workload of interest.

4.3.3. Number of Subblocks. Increasing the number of subblocks per block allows the IIC-C to take advantage of compression at a finer granularity, but increases the storage overhead in the tag structure. Figure 3 shows the impact of varying the number of subblocks on effective cache size. Using four subblocks as we do strikes a compromise between effective compression ratio and tag overhead.

4.3.4. Compression/Decompression Latency. Table 6 presents the performance of the IIC-C as we vary the block decompression latency from 0 cycles to 32 cycles. (Our default configuration takes 8 cycles.) Compression latency is four times the decompression latency in each case. The latency is determined by the amount of hardware used, so larger latencies can be traded for reduced hardware overhead. As expected, as we increase the penalty of decompression, the performance improvement decreases. However, most benchmarks are relatively tolerant of these latency increases, and

IEEE COMPUTER SOCIETY

Table 4. Performance of non-adaptive and adaptive (default) compression schemes, relative to an uncompressed IIC.

	Always Compress	Adaptively Compress
amp	7%	11%
art	224%	245%
bzp	15%	22%
fac	4%	5%
gal	36%	48%
gcc	-12%	-1%
lcs	283%	283%
mcf	56%	64%
mgd	3%	1%
prl	0%	0%
twf	49%	37%
vpr	22%	16%
avg	27%	29%

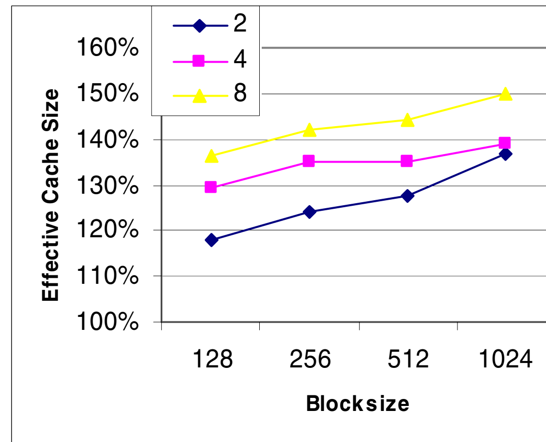

Figure 3. Average effective size of the IIC-C with varying numbers of subblocks per block.

Table 5. Performance of compressed system, relative to an IIC of the same size, for varying cache size.

	WS	1MB	2MB	4MB
amp	2MB	11%	-1%	0%
art	4MB	245%	114%	0%
bzp	8MB	22%	3%	3%
fac	>16MB	5%	12%	8%
gal	8MB	48%	2%	2%
gcc	4M	-1%	4%	5%
lcs	>16MB	283%	71%	70%
mcf	>16MB	64%	131%	213%
mgd	>16MB	1%	21%	20%
prl	8MB	0%	-1%	0%
twf	2MB	37%	0%	0%
vpr	>16MB	16%	-6%	1%
avg		29%	17%	16%

Table 6. Performance of compressed system, relative to an IIC, for varying decompression (and compression) latency.

	0	8	16	32
amp	11%	11%	10%	10%
art	249%	245%	239%	227%
bzp	22%	22%	22%	22%
fac	5%	5%	5%	4%
gal	48%	48%	46%	45%
gcc	-1%	-1%	-2%	-4%
lcs	290%	283%	276%	261%
mcf	68%	64%	62%	53%
mgd	2%	1%	1%	0%
prl	0%	0%	-2%	-3%
twf	43%	37%	32%	22%
vpr	18%	16%	14%	11%
avg	30%	29%	28%	26%

compression remains an overall win. On average, we see the performance improvement of compression reduce from 30% with no penalty to 26% at 32 cycles per block.

4.3.5. Memory Latency. As we head into the future, memory latencies are likely to increase in terms of processor cycles. To evaluate how compression will perform with these longer latencies, we varied the memory latency from 150 cycles to 1200 cycles. The bus bandwidth is fixed at 6.4 GB/sec in all cases.

Table 7 presents the performance of the compressed cache relative to an uncompressed IIC for four memory latencies. As the memory latency increases, off-chip misses becomes more expensive, decreasing the impact of the decompression latency while increasing the importance of a higher hit rate in the cache. These changes mean that compression becomes more and more desirable. As can be seen, a compressed hierarchy looks increasingly attractive as memory latencies increase; the average performance increase goes from 29% to 67%.

IEEE COMPUTER SOCIETY

Table 7. Performance improvement of compressed system relative to IIC for varying memory latency.

	150	300	600	1200
amp	11%	18%	24%	23%
art	245%	291%	316%	322%
bzp	22%	32%	40%	44%
fac	5%	7%	5%	6%
gal	48%	80%	119%	146%
gcc	-1%	-1%	1%	3%
lcs	283%	425%	509%	549%
mcf	64%	94%	109%	118%
mgd	1%	5%	8%	10%
prl	0%	0%	1%	4%
twf	37%	87%	168%	268%
vpr	16%	28%	37%	43%
avg	29%	43%	56%	67%

4.4. Overall Performance

Overall, we see that the compressed hierarchy favors larger blocks due to their better compressibility and the ability of compression to mitigate their key drawback, increased bandwidth consumption. To investigate this phenomenon in more detail, Figure 4 compares the performance of the compressed system at its best block size (512B) relative to the best performing uncompressed configuration (128B blocks) for the 20 SPEC benchmarks where compression has an impact of more than 1%. (As in the previous section, averages continue to be taken over all 26 benchmarks.)

We also present the relative performance of four other configurations. To show the impact of simply increasing block size on uncompressed caches, we show both an 8-way associative uncompressed LRU cache and an uncompressed IIC with 512B blocks. A 1.1 MB 9-way associative uncompressed LRU cache indicates the effectiveness of using the area overhead consumed by the IIC-C tags to simply increase the size of an uncompressed cache. (Note that the additional 128KB of space reflects the overhead of an IIC-C with 128B blocks; the IIC-C area overhead with 512B blocks is only 27KB.) Finally, a 2 MB conventional cache represents an upper bound on the capacity of the compressed 1 MB cache; due to the number of tag entries in the IIC-C, 2 MB is its maximum possible effective cache size.

Focusing on the IIC-C bars, we see that seventeen of the 26 SPEC benchmarks show a speedup with larger blocks and a compressed hierarchy—fifteen of them by more than 15%, ranging up to 176%—while only three exhibit a slowdowns of at most 15% (due to the larger block size rather than compression). Interestingly, these benefits come from a variety of sources. Art, with the largest gain, benefits significantly from both cache and bus compression (see Table 3). However, the next two largest improvements (mcf and lucas) come on benchmarks where doubling the cache has very little effect. Mcf's gains come from a combination of the prefetching effect of the larger block size and increased bandwidth due to bus compression. For lucas, simply increasing the block size degrades performance by 41%, but the added bandwidth of bus compression more than compensates, resulting in a 71% boost. Galgel and twolf benefit primarily due to cache compression; in these cases, the compressed hierarchy provides more than half the performance gain of doubling the cache. Galgel shows a very large gain from the small increase in overall size. The 1.1MB LRU cache outperforms the IIC-C at 512B blocks due to block size effects, at 128B blocks the IIC-C performs almost as well as the 2MB cache.

Many of the other benchmarks' gains can be attributed to the increased block size. Although prefetching would achieve some of this gain at the smaller block size, prefetching would increase rather than decrease bus bandwidth consumption—recall from Figure 2 that the 512B compressed system consumes 45% less bandwidth than the 128B uncompressed system—and would not improve capacity-bound benchmarks either.

For three benchmarks—ammp, bzip2, and vpr—the 512B-block compressed hierarchy reduces performance by up to 15%. This degradation is due to the larger block size; in these cases, compression reduces the performance loss due to larger blocks by almost half.

Overall, we see an average improvement per benchmark of 20%. Because some of the largest speedups occur on low-IPC memory-bound benchmarks, the impact on harmonic mean IPC is even larger, at 44%. In contrast, the 2 MB cache provides a 21% average speedup and only a 19% increase in mean IPC, despite having an area overhead of 100% compared to 3% for the 512B-block IIC-C. Although several benchmarks in this comparison gain primarily from increasing the block size, doing so without the bandwidth benefit of bus compression results in dramatic slowdowns in others (from 23% up to 41%), resulting in a slight performance degradation overall. Thus we see a synergy between cache compression, bus compression, and increased block sizes, providing much more overall benefit than any one of these techniques in isolation.

IEEE COMPUTER SOCIETY

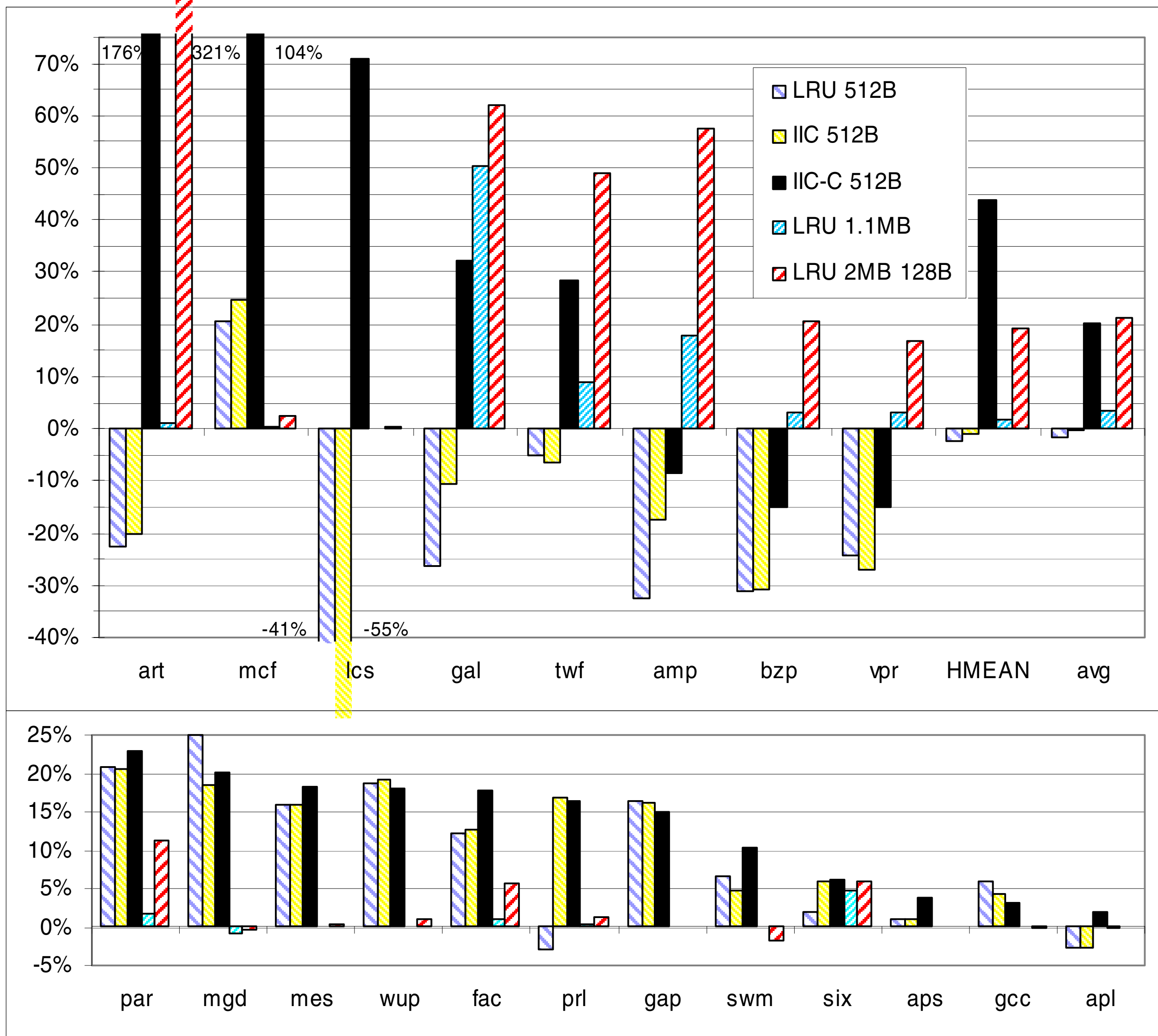

Figure 4. Performance relative to 128B-block uncompressed LRU.

5. Conclusion

In this paper we evaluate the use of a unified compressed memory hierarchy to improve processor performance. Our system consists of a compressed on-chip cache design—the Indirect Index Cache with Compression (IIC-C)—and a compressed main memory modeled after IBM's MXT system. This configuration not only increases the effective size of the cache and main memory, eliminating costly off-chip misses and disk accesses, but also allows transmission of compressed data across the bus, increasing effective bus bandwidth.

The IIC-C is a novel design for an on-chip compressible cache. The IIC-C uses indirection to translate cache block addresses into the physical indexes in a data store where these blocks are held. This indirection, combined with subblocking, allows blocks to be stored in compressed form while freeing the saved space to store other blocks, allowing the cache to effectively increase its size when storing compressed data. We also present an adaptive scheme to reduce the impact of decompression latency on processor performance.

For a 1 MB L3 cache with 128B blocks, the compressed memory system gives an average speedup of 19% across the full SPEC CPU2000 benchmark suite, increasing mean IPC by 22%. Both cache and bus compression contribute significantly to this speedup. Moreover, the compressed memory system provides even

IEEE COMPUTER SOCIETY

better absolute performance with larger block sizes; using 512B blocks in the compressed system increases the mean IPC over the baseline to 44%. These results will be further improved by reduced disk accesses due to the larger effective main memory size; quantifying this benefit is outside the scope of this paper.

Overall, our compressed memory hierarchy achieves greater than 50% of the performance gain of doubling the on-chip cache, for one tenth the area. As memory latencies increases, the benefits of compression increase as well. Coupled with greater bandwidth demands due to SMT and CMP processors, these results indicate great potential for a unified approach to memory compression in future processors.

Future work that must be addressed along the way includes dealing with the performance impact of false sharing for large blocks in large-scale (multi-chip) multiprocessor systems and better adaptive compression algorithms to further reduce performance degradation due to compression latencies on those few benchmarks that are sensitive to it. These two issues would benefit from an adaptive block size scheme [7], if one could be added to the compressed hierarchy.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCR-0105503. This work was also supported by gifts from Intel and IBM and a Sloan Research Fellowship.

References

[1] B. Abali, H. Franke, S. Xiaowei, et.al., "Performance of Hardware Compressed Main Memory", Proc. 7th Int'l Symp. on High-Performance Computer Architecture, 2001, pp. 73-81.

[2] A. Alameldeen and D. Wood, "Adaptive Cache Compression for High-Performance Processors", Proc. 31st Annual Int'l Symp. on Computer Architecture, 2004.

[3] S. Arramreddy, D. Har, K. Mak, et al., "IBM X-Press Memory Compression Technology Debuts in a ServerWorks NorthBridge", Hot Chips 12, 2000.

[4] K. Basu, A. Choudhary, J. Pisharath, and M. Kandemir, "Power Protocol: Reducing Power Dissipation on off-chip Data Buses", Proc. 35th Annual Int'l Symp. on Microarchitecture, 2003.

[5] N. Binkert, E. Hallnor, and S. Reinhardt, "Network-Oriented Full-System Simulation using M5", Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2003.

[6] D. Citron and L. Rudolph, "Creating a Wider Bus Using Caching Techniques", Proc. First Annual Symp. on High-Performance Computer Architecture, Feb. 1995, pages 90–99.

[7] C. Dubnicki and T. LeBlanc, "Adjustable Block Size Coherent Caches", Proc. 19th Int'l Symp. on Computer Archtecture, May 1992.

[8] M. Farrens and A. Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width", Proc. 18th Annual Int'l Symp. on Computer Architecture, May 1991, pages 128–137.

[9] P. Franaszek, J. Robinson, and J. Thomas, "Parallel Compression with Cooperative Dictionary Construction", Proc. Data Compression Conf., 1996, pp. 200–209.

[10] E. Hallnor and S. Reinhardt, "A Fully Associative Software-Managed Cache Design", Proc. 27th Annual Int'l Symp. on Computer Architecture, pages 107-116, June 2000.

[11] G. Hammond and S. Naffziger, "Next Generation Itanium Process Overview", Intel Developers Forum, 2001.

[12] N. Kim, T. Austin, and T. Mudge, "Low-Energy Data Cache using Sign Compression and Cache Line Bisection", 2nd Annual Workshop on Memory Performance Issues, May 2002.

[13] D. Kirovski, J. Kin, and W.H. Mangione-Smith, "Procedure Based Program Compression", Proc. 30th Int'l Symp. on Microarchitecture, 1997, pp. 204–213.

[14] M. Kjelso, M. Gooch, and S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor", Proc. 22nd EUROMICRO Conf., Beyond 2000: Hardware and Software Design Strategies, 1995, pp. 423–430.

[15] J.S. Lee, W.K. Hong, and S. D. Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity," Journal of Systems Architecture, vol. 46, Dec. 2000, pp. 1365-1382.

[16] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a High Performance Code Compression Method", Proc. 32nd Annual Int'l Symp. on Microarchitecture, Dec. 1999, pp. 93–102.

[17] H.Lekatsas and W. Wolf, "Code Compression for Embedded Systems", Proc. 35th Design Automation Conf., 1998.

[18] S. Roy, R. Kumar, and M. Prvulovic, "Improving System Performance with Compressed Memory", Proc. 15th Int'l Parallel and Distributed Processing Symp., Apr 2001, pp. 630–636.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior," Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, October 2002. San Jose, California. See also http://www.cs.ucsd.edu/~calder/simpoint/single-sim-pionts.htm.

[20] L. Villa, M. Zhang and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction", Proc. 33rd Int'l Symp. on Microarchitecture, Dec 2000.

[21] P. R. Wilson, S. F. Kaplan and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems", Proceedings of USENIX 1999.

[22] J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design", Proc. 35th Annual Int'l Symp. on Microarchitecture, Dec. 2002.

[23] Y. Zhang, J. Yang and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design", Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

IEEE COMPUTER SOCIETY