A Compressed Memory Hierarchy using an Indirect Index Cache

Erik G. Hallnor and Steven K. Reinhardt

Advanced Computer Architecture Laboratory
EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
{ehallnor, stever}@eecs.umich.edu

Abstract

The large and growing impact of memory hierarchies on overall system performance compels designers to investigate innovative techniques to improve memory-system efficiency. We propose and analyze a memory hierarchy that increases both the effective capacity of memory structures and the effective bandwidth of interconnects by storing and transmitting data in compressed form.

Caches play a key role in hiding memory latencies. However, cache sizes are constrained by die area and cost. A cache's effective size can be increased by storing compressed data, if the storage unused by a compressed block can be allocated to other blocks. We use a modified Indirect Index Cache to allocate variable amounts of storage to different blocks, depending on their compressibility.

By coupling our compressed cache design with a similarly compressed main memory, we can easily transfer data between these structures in a compressed state, increasing the effective memory bus bandwidth. This optimization further improves performance when bus bandwidth is critical.

Our simulation results, using the SPEC CPU2000 benchmarks, show that our design increases performance by up to 107% on some benchmarks while degrading performance by no more than 2% on others. Compressed bus transfers alone account for up to 59% of this improvement, with the remainder coming from increased effective cache capacity. As memory latencies increase, our design becomes even more beneficial.

1 Introduction

Memory latencies have long been a performance bottleneck in modern computers. In fact, with each technology generation, microprocessor execution rates outpace improvements in main memory latencies. Memory latencies are thus having increasing impact on overall processor performance.

The rift between processor and memory speeds is alleviated primarily by using caches. Today's microprocessors have on-chip cache hierarchies incorporating multiple megabytes of storage [6]. Increasing the size of on-chip caches can greatly increase processor performance; for example, increasing the size of the on-chip tertiary cache from 1MB to 4MB results in a 427% performance increase for the art benchmark on our simulated configuration. However, the amount of on-chip storage cannot be increased without bound. Caches already consume most of the die area in high-performance microprocessors. Practical cache sizes are constrained by the increased fabrication costs of larger die, and ultimately limited by semiconductor manufacturing technology.

Memory bandwidth is also a scarce resource in high-performance systems. Although recent years have seen significant advances in signaling and DRAM interface technologies, chip I/O bandwidth is limited by the cost of packaging technology. Furthermore, many of the techniques being employed by processor designers to deal with memory latency— e.g., prefetching, multithreaded CPUs, and chip multiprocessors—result in increased bandwidth demands.

Data compression has long been used to alleviate capacity and bandwidth constraints in several domains. This paper explores the use of data compression in the processor cache hierarchy. We examine two applications. First, we store compressed data in the last-level on-chip cache to increase its effective capacity. Second, by adding a compressed main memory system, such as IBM's Memory Expansion Technology (MXT) [1], data can be transmitted across the bus in compressed form, increasing the effective per-pin bandwidth. Transferring compressed data from memory also removes the need to compress data before loading it into the cache.

Data compression also tends to reduce power and energy consumption. Fewer bits are required to store and to transmit a given amount of information, reducing static and dynamic power consumption, respectively. Of course, these benefits must be weighed against the power cost of the compression/decompression hardware. We leave analysis of the power/energy impact of our compressed memory hierarchy, along with potential integration of energy-specific optimizations, for future work.

A key challenge in the design of a compressed data store is the management of variable-sized data blocks. For example, compressing a 128-byte block to 72 bytes does not increase effective cache capacity if none of the 56 bytes saved can be used for other data. A conventional cache structure, in which each address tag is associated statically with a single fixed-size data block, is thus inadequate for storing compressed data.

We address this problem using a variation of the Indirect Index Cache (IIC) [5]. The IIC does not associate a tag with a specific data block; instead, each tag contains a pointer into a data array which contains the blocks. The original intent of this indirection is to provide a fully-associative cache, amenable to software management. In this paper, we use this indirection in combination with compression to associate a single tag with a variable number of data blocks, while allowing unused blocks to associate with other tags. We call our design the Indirect Index Cache with Compression, or IIC-C.

We evaluate our design using the SPEC CPU2000 benchmark suite. We find that our compressed cache outperforms a traditional LRU cache by an average 16% at a 1024-byte block size. We also show that as memory latencies increase, our design improves performance by an even greater margin.

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

2 Previous Work

In this section we present previous work on compression in the memory hierarchy. First we discuss previous work in compressed caches, and then present proposed compressed main memory designs that we can use to supply compressed data to our cache.

2.1 Cache compression

Compression has long been used in the embedded world for reducing code size [9, 12, 13]. 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.

Recently, a number of designs for cache data compression have been proposed. Most of these schemes have applied compression 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) [19, 18] replaces the top N frequently used 32-bit values with log (N) bits. When built as a separate structure [19], 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 [18]. The encoded frequent values are stored in the first bits of the cache line. If the value is marked as compressed, only these bits 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) [16]. 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. [8] utilize the knowledge that most of the bits of values stored in a L1 data cache are merely sign bits. Their scheme compresses 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. They find that with an overall increase in storage of 50%, their cache approaches the performance of a 100% larger conventional cache.

Alameldeen and Wood [2] also use the leverage that small values are stored in larger memory blocks to create a compression algorithm called frequent pattern compression (FPC). Their cache uses indirection, much like the IIC, between tags and data 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 for only the cost of the extra tag storage.

The Selective Compressed Memory System (SCMS) [11] 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%. We also evaluate the performance potential of our compression scheme on the entire SPEC CPU2000 suite using execution-driven simulation, while the SCMS work presented trace-driven evaluation of a subset of the SPEC95 benchmarks.

2.2 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 [10, 14, 17] set up a portion of physical memory

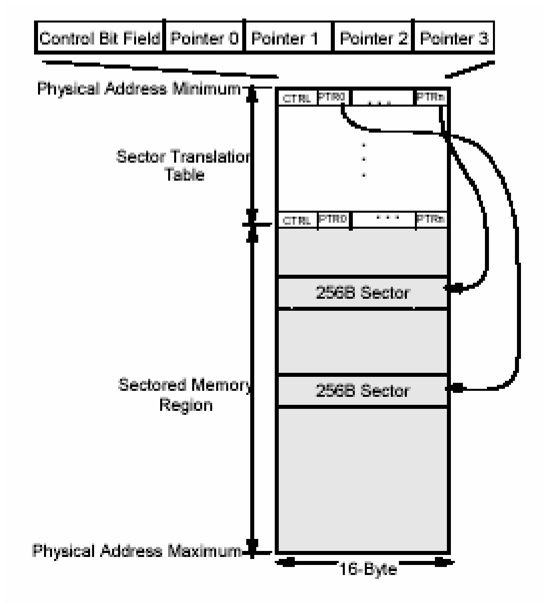

Figure 1: IBM's Memory eXpansion Technology (MXT) (from [1]).

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 cache for uncompressed data.

MXT maps the "real" addresses generated by the processor to the physical addresses of the compressed memory using a sector translation table (STT) (see Figure 1). 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 leverage this design when creating our compressed cache.

3 Design of the IIC-C

3.1 The Indirect Index Cache (IIC)

The Indirect Index Cache with Compression (IIC-C) is based on the Indirect Index Cache (IIC) [5]. The basic IIC, shown in Figure 2, consists of a data array containing the cache blocks and a tag store which contains the tags for these blocks. Each IIC tag entry holds a pointer to the data block with which it is currently associated. This indirection provides the ability to easily manage a fully associative cache.

Replacements in the IIC are managed by a software algorithm running on an embedded controller or as a thread on the main CPU. Our primary algorithm, called Generational Replacement (GEN) [5], maintains prioritized pools (queues) of blocks; periodically, referenced blocks are moved to higher priority pools and unreferenced blocks are moved to lower priority pools. Replacements are selected from the unreferenced blocks in the lowest priority pool. To reach the highest priority pool, a block must be referenced regularly over

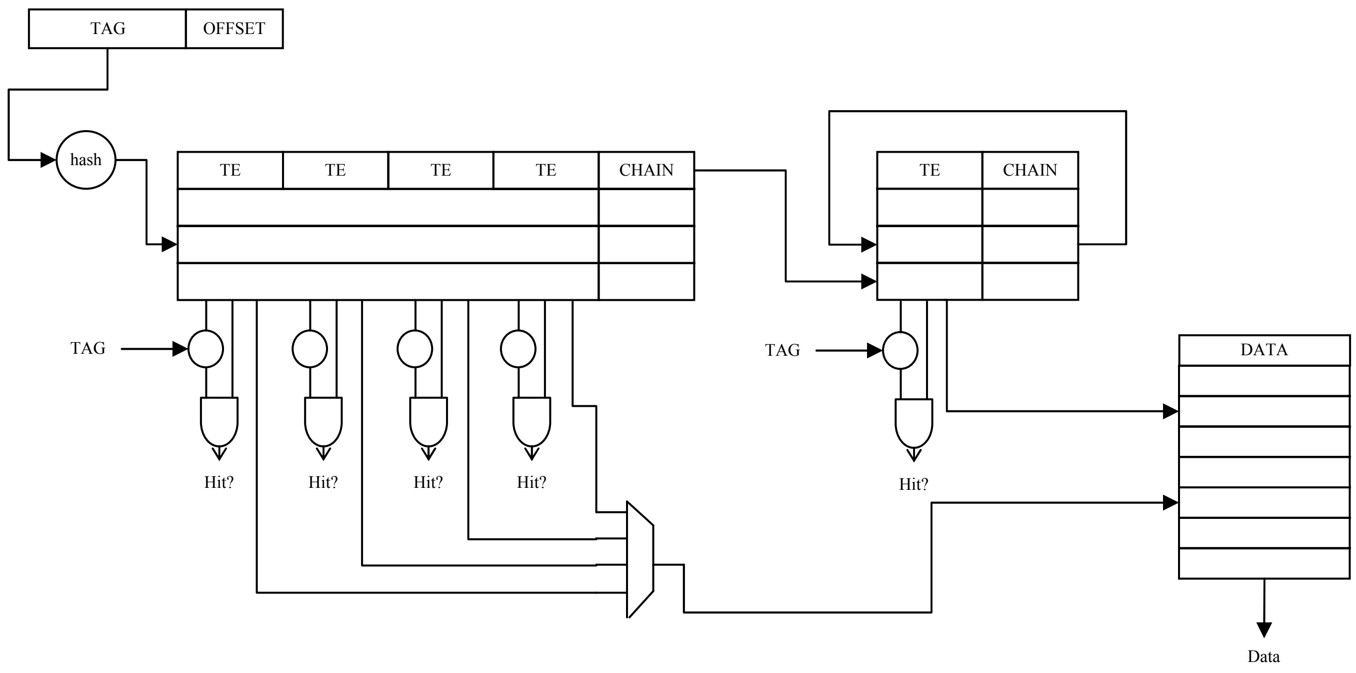

Figure 2: The Indirect Index Cache

an extended period of time; once there, it must remain unreferenced for a similarly long period to return to the lowest priority pool. This algorithm thus combines reference recency and frequency information with a hysteresis effect, while relying only on block reference bits and periodic data-structure updates.

To ensure that adequate replacement candidates are available to deal with bursts of misses, the algorithm can identify multiple candidates per invocation and maintain a small pool of replacement blocks. These blocks are used by hardware to handle incoming responses, while GEN works in the background to keep this pool at a predetermined level.

The IIC was originally designed to utilize large on-chip caches to achieve better performance over traditional LRU caches. With a few minor changes the IIC can also be used to increase the effective size of these caches by incorporating compression.

Tag	Comp?	Triv?	D0	D1	D2	D3

Figure 3: An IIC-C Tag

3.2 Extending the IIC to Incorporate Compression

We can leverage the indirection of the IIC to create an STT-like structure for the on-chip cache. We make a few simple changes to the IIC tag entry to support compressed data. First, we modify the IIC tag to contain multiple pointers to smaller data blocks to represent a single cache block, just as an MXT SST entry contains multiple sector pointers. If the 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. We also add two status bits to the tag: one to denote whether the block is compressed or not, and another to indicate that the compressed value is “trivial” and stored in the subblock pointers themselves. Figure 3 represents an IIC-C tag entry.

We also needed to modify GEN to keep a pool of available sub-blocks instead of just full data blocks. We also need to expand the algorithm to check the available blocks on writes as well as misses. This is because there is the possibility that a write to a compressed block will decrease the compressibility and need another sub-block to store the result. This is a minor change and doesn't affect the algorithm's performance noticeably.

To improve effective pin bandwidth and avoid compressing incoming blocks, our system uses a compressed main memory as well. 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 [11]; we do not include that optimization in this paper. Compressed bus transfers could be used without a compressed main memory by compressing/decompressing blocks in the memory controller on each access. However, once the compression logic is moved to the memory controller, it seems worthwhile to keep main memory in a compressed state as well. Otherwise, both the compression and decompression latencies will be added to each cache miss.

We chose to use LZSS [7] as our compression algorithm since it is also used in MXT. LZSS is a parallel version of LZ77 which can be implemented in hardware [3]. The speed of LZSS is dependent on the number of simultaneous compressions. 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 our compressed cache. Of course, the choice of LZSS is arbitrary; we can use any compression algorithm, such as the X-RL used in SCMS [11], as long as it can be efficiently realized in hardware.

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 [6]. With the L3 storing data in a compressed form, the L2 will serve as a buffer of decompressed cache blocks. The L2 thus reduces the impact of the decompression latency by reducing the number of accesses to the L3. The IIC-C itself could serve the same function, storing recently accessed blocks uncompressed and others in compressed form; we leave exploration of that design space for future work.

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 multi-megabyte L3 [3]. In addition to the basic IIC overheads [5], the IIC-C adds extra subblock pointers to the tag, increasing the size by about 6 bytes per tag entry. 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 [5]. 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 only 134K for a 1MB IIC-C with 128-byte blocks and 64-byte subblocks, and falls to 14K at a 1024-byte block size with 256-byte subblocks.

The next section presents the evaluation of our compressed memory hierarchy.

4 Evaluation

4.1 Methodology

We used the M5 simulator [4] to evaluate our design. M5 simulates an out-of-order speculative Alpha processor with a detailed timing memory hierarchy. The simulation parameters are presented in Table 1. We based the cache hierarchy on the McKinley Itanium2 [6], but we increased the cycle delays to match a 2 GHz processor. Although future L3 caches will doubtless be larger than 1MB, the limitations of our SPEC CPU2000 workload and simulation runtime constraints made it difficult to explore larger cache sizes.

Table 1: Simulation Parameters

Parameter	Value
Frequency	2 GHz
Front-end pipeline	10 cycles fetch-to-decode 5 cycles decode-to-dispatch
Fetch bandwidth	Up to 8 instructions per cycle, Max 3 branches per cycle
Branch predictor	Hybrid local/global (ala 21264) Global: 13-bit history, 8K-entry PHT Local: 2K 11-bit history regs, 2K-entry PHT Choice: 13-bit global history, 8K-entry PHT
BTB	4K entries, 4-way set associative
Instruction Queue	Unified int/fp, 256 entries
Reorder buffer	512 entries
Execution BW	Up to 8 insts per cycle
Function Units	8 int alu, 4 int mul, 4 fp add/sub, 4 fp mul/div/sqrt, 4 data-cache rd/wr port
Latencies	Integer: mul 3, div 20, all others 1 fp: add/sub 2, mul 4, div 12, sqrt 24 all ops fully pipelined exc. div and sqrt
L1 Icache / Dcache	Both: 16KB, 4-way set assoc., 64B block size, 1 cycle latency Instr: up to 8 outstanding misses Data: up to 32 outstanding misses
L2 Unified Cache	256KB, 8-way set assoc., 128B block size, 12 cycle hit latency, up to 40 outstanding misses
L3 Unified Cache	1MB, 8-way set assoc., 128B block size, 26 cycle hit latency, up to 40 outstanding misses
Memory Bus	500 MHz, 32 byte data path
Main Memory	150 cycle latency

To keep the tag overhead manageable, we set the number of IIC-C subblocks within a cache line to four. To estimate the compressibility of cache blocks, 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 [7]. We also assume that 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 [15] and run for 300M instructions.

4.2 Benchmark Classification

We begin by measuring the basic memory behavior of the SPEC CPU2000 benchmarks. We simulated each benchmark as described above using fully associative LRU instruction and data caches with 128-byte blocks and sizes varying from 128K to 16M. From these runs we were able to estimate the working set size of our sample of each benchmark. To estimate the memory bandwidth demand and compressibility of each benchmark we also ran with a 1MB, 8-way set-associative LRU cache with 128-byte blocks and measured the number of bytes requested from the memory as well as the compressibility of the data requested by the cache. The results are presented in Table 2. All the benchmarks had instruction working sets smaller than 128K, so that column is not shown. As can be seen, five of the benchmarks (apsi, eon, equake, fma, and sixtrack) have data working set sizes of 1MB or less, and so will not show any benefit from increasing the cache size further. Since the IIC-C we model has only twice the minimum number of tag entries, it can at best double the effective size of the cache. Thus only those benchmarks that show a significant decrease in the number of misses when going from a 1MB to 2MB cache can show improvements from using our 1MB IIC-C. There are nine benchmarks that show a reduction of 100K or more misses in this situation: ammp, art, bzip2, facerec, galgel, mcf, parser, twolf, and vpr. When we present an average, it will be over all 26-benchmarks, however we will focus on these nine benchmarks to cope with presentation space and simulation time constraints

Table 2: SPEC CPU2000 Memory Behavior

Benchmark	Data WS (bytes)	Mem BW (bytes/inst)	Data Compressibility	Miss Rate	# of misses eliminated by incr from 1M to 2M	% of misses eliminated
ammp	2M	0.40	12.12%	20.94%	117,194	78.36%
applu	4M	1.42	3.08%	76.48%	17,522	0.32%
apsi	1M	0.07	95.59%	45.44%	0	0%
art	4M	9.79	57.49%	72.44%	33,841,778	87.93%
bzip2	8M	0.15	3.96%	27.17%	393,950	69.88%
crafty	2M	0.01	66.75%	9.22%	24,059	61.31%
eon	128K	0.00	40.91%	100%	0	0%
equake	128K	0.00	92.43%	100%	0	0%
facerec	>16M	0.59	4.55%	34.31%	176,216	7.58%
fma	128K	0.00	28.50%	100%	0	0%
galgel	8M	0.42	30.55%	49.71%	3,282,158	97.45%
gap	>16M	0.06	56.78%	49.37%	2	0%
gcc	4M	0.02	31.37%	0.46%	46	0.06%
gzip	2M	0.01	6.30%	25.75%	83	0.16%
lucas	>16M	1.15	81.32%	65.62%	7,317	0.16%
mcf	>16M	7.89	44.68%	57.27%	691,126	2.24%
mesa	8M	0.04	88.98%	43.45%	2,041	1.25%
mgrid	>16M	0.53	15.16%	49.40%	20,025	0.98%
parser	>16M	0.17	37.01%	30.42%	277,233	41.04%
perl	8M	0.26	60.45%	77.74%	34,931	13.37%
sixtrack	1M	0.01	35.24%	4.36%	0	0%
swim	16M	1.58	11.95%	73.38%	860	0.01%
twolf	2M	0.39	35.25%	20.24%	1,497,484	99.16%
vortex	4M	0.03	77.50%	14.96%	18,949	19.49%
vpr	>16M	0.36	22.86%	38.56%	688,982	49.97%
wupwise	8M	0.16	46.71%	73.00%	8,974	1.41%

4.3 Results

Because the IIC-C design builds on the IIC, we establish our baseline by comparing the performance of a plain 1MB IIC to an 8-way set-associative LRU cache of the same capacity. Figure 4 presents the relative performance for the nine benchmarks identified above, as well as the average over all the benchmarks. Except for ammp, the IIC results are similar to the 8-way LRU results. Ammp suffers from a number of conflict misses in the LRU case which the IIC removes. To account for the additional area overhead of the IIC-C, we also include the performance of a 9-way conventional LRU cache (roughly a 13% area increase).

To factor out the impact on performance of varying block sizes, we will present our IIC-C results below relative to the performance of an uncompressed IIC using the same block size. To understand the impact of block size on absolute performance, we show the IPC for the base IIC configuration at various block sizes, relative to a 128-byte block size, in Figure 5. The effect of increasing block size is quite varied: ammp, art, bzip2, and galgel suffer from pollution at large block sizes, while mcf and twolf benefit from prefetching effects. Since the memory bandwidth remains fixed the large blocks take longer to transmit. At 1024 byte blocks this latency slightly diminishes the benefit from prefetching in MCF.

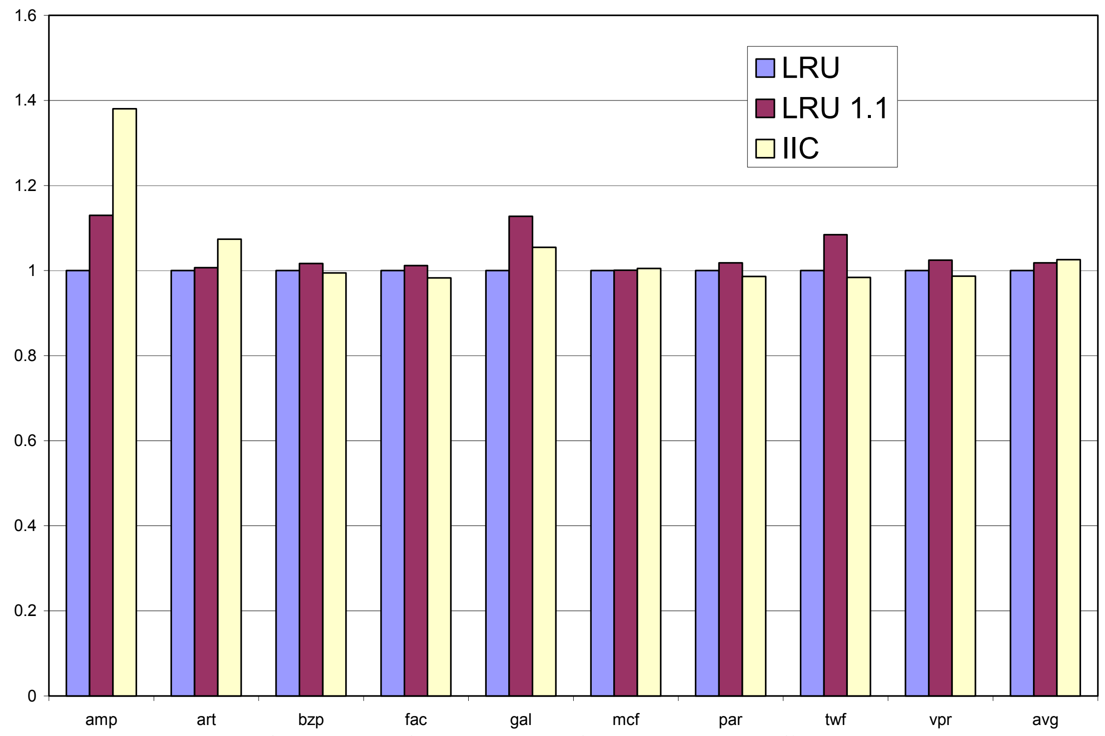

Figure 4: Performance Relative to a 1M LRU Cache

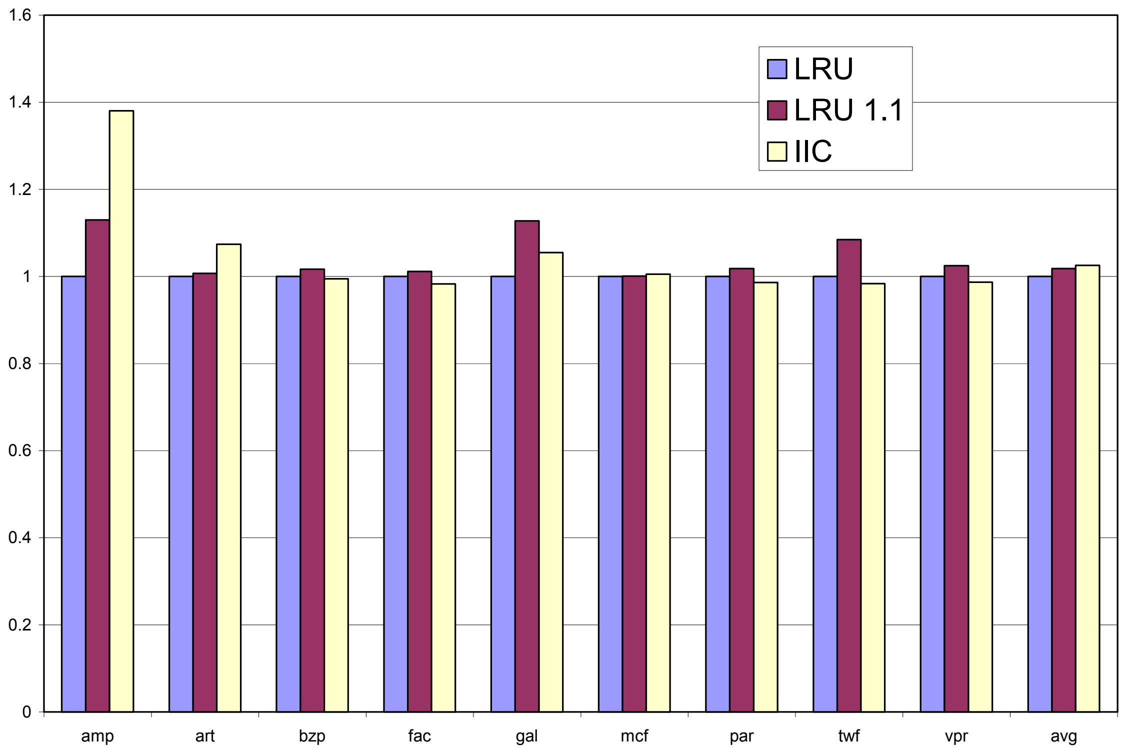

Figure 5: Performance of the IIC relative to the 128-byte block

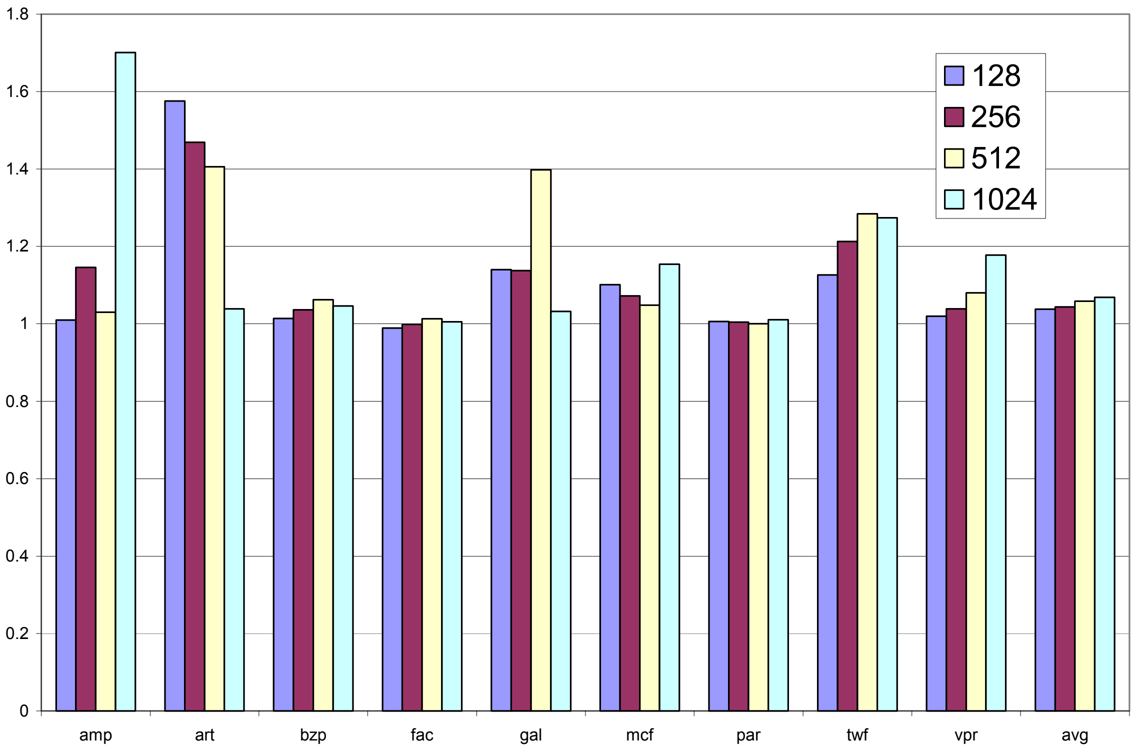

Figure 6: Performance of the stand alone IIC-C relative to the IIC with the same block size

4.3.1 Cache compression

Figures 6 and 7 present the performance and miss rate of the IIC-C relative to the IIC in a stand-alone environment (neither the bus nor main memory use compression). We again vary the L3 block size from 128B to 1024B to explore the effect of larger blocks on performance. The compression block size is set equal to the L3 cache line size and varied from 128B to 1024B.[1] Because larger blocks provide more context for the compression algorithm, they typically enable higher compression ratios. Recall that we normalize our IIC-C results to an IIC with matching block size to factor out other block-size effects. As can be seen, on average our compressed cache outperforms the traditional IIC for all block sizes. For 128B blocks, we get our largest gain (57%) on art, while our worst loss is only 1%. Averaged across all the benchmarks, the compressed cache improves performance over by 4-7% depending on the block size, while increasing the average effective cache size by 47%.

Of the nine benchmarks we identified in Section 4.2 with potential savings, only 6, showed significant improvement at any block size. Art shows huge wins because the IIC-C is able to eliminate up to 50% of its 33M misses. It suffers at 1024B blocks, despite doubling the effective cache size to 2M, because the prefetching effect of the large block eliminates most of the misses that compression removed at lower sizes. Mcf shows modest performance improvement and modest miss rate reduction, however it also doubles the effective cache size at 512B and 1024B blocks. Galgel, and twolf all show massive reductions in miss rates, up to 96%, but these reductions do not translate to as large gains in performance. Ammp shows improvement

[1] With the same compression hardware, the compression latency will increase with the block size. We assume that compression hardware scales to keep the same compression latency across all block sizes. With LZSS, this scaling is linear as discussed in Section 3.2.

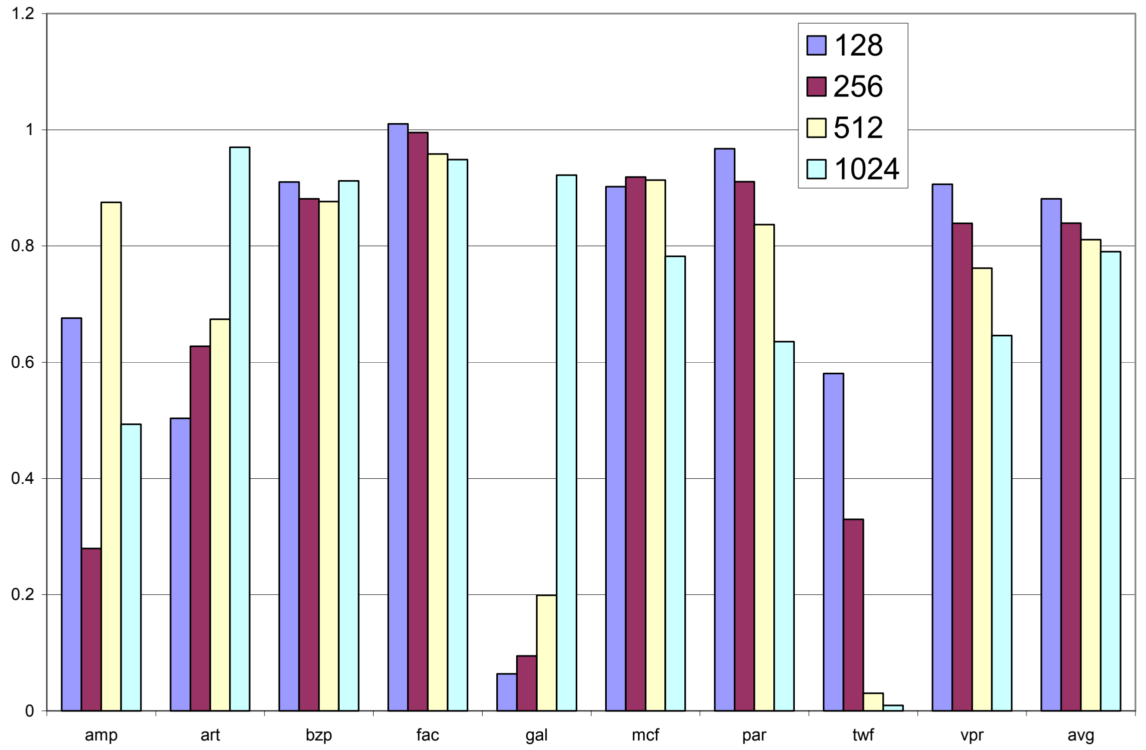

Figure 7: Miss rates of the stand alone IIC-C relative to those of an IIC with the same block size

because the larger block sizes are more compressible allowing more to be stored, eliminating its pollution problem. This same factor is seen in a smaller magnitude for vpr at a 1024B block size.

The other benchmarks fail to show significant performance improvements for a number of reasons. Bzip2 and facerec both increase the effective size of the cache by less than 10% at the lower block sizes, which leads to lower gains. Even when facerec increases the effective size by 47% at 1024B blocks the large improvement there is not an appreciable performance gain, but we do see a 5% improvement in miss rate. Parser sees large miss rate reductions, but these are not translated into a similar performance gain.

4.3.2 Bus compression

To isolate the performance impact of transmitting compressed data across the bus, we simulated a system with a compressed main memory and compressed bus transfers, but with no on-chip compressed cache. Data is thus decompressed on-chip immediately after being received from main memory. Figure 8 presents the performance of this scheme for each block size relative to an uncompressed IIC with the same block size. Art sees large benefits at every block size due to its high memory bandwidth requirements (see Table 2). The benefit increases with block size since we use a fixed bus bandwidth; larger block sizes incur longer delays, resulting in more potential savings when blocks are compressed. These results show that compression can be used to utilize the existing bus bandwidth more effectively, especially for long-latency transmissions. However, compressed bus transfers do decrease performance by up to 4% on twolf at 1024 byte blocks. This decrease comes both from the added latency of decompression and from the inability to send the critical word first in compressed data. For some benchmarks, these losses can be covered partially by reduced miss rates when storing the data in a compressed form.

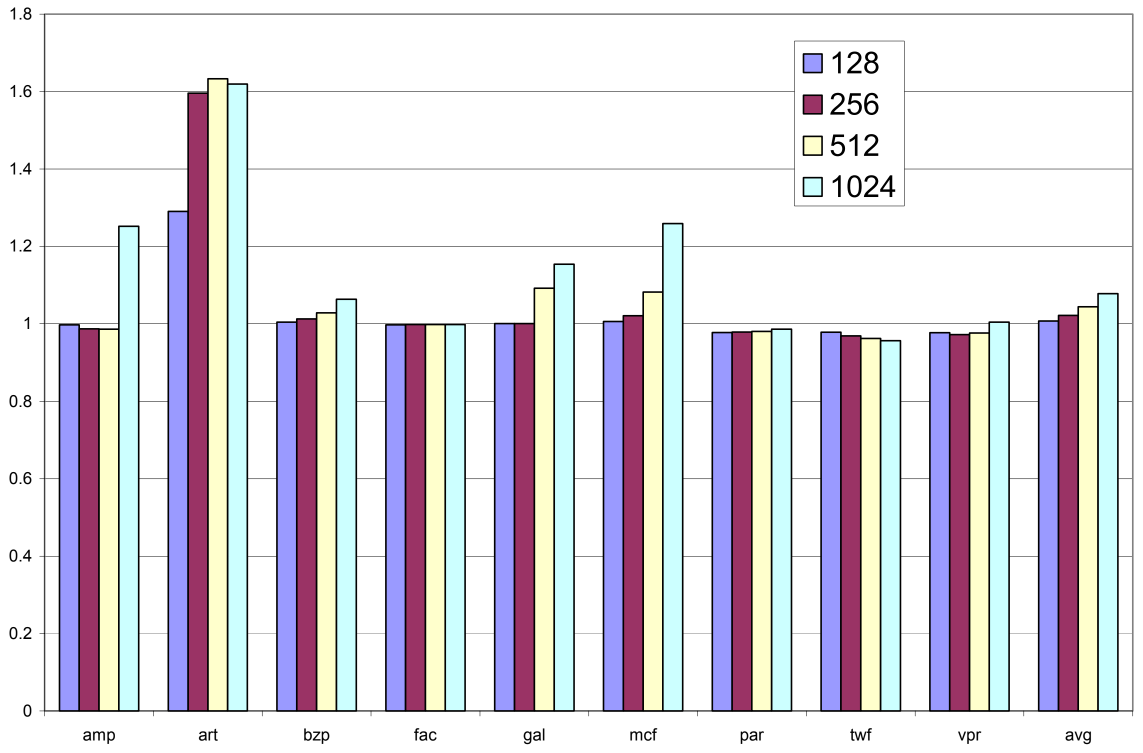

Figure 8: Performance of sending compressed blocks over the bus relative to an IIC with the same block size

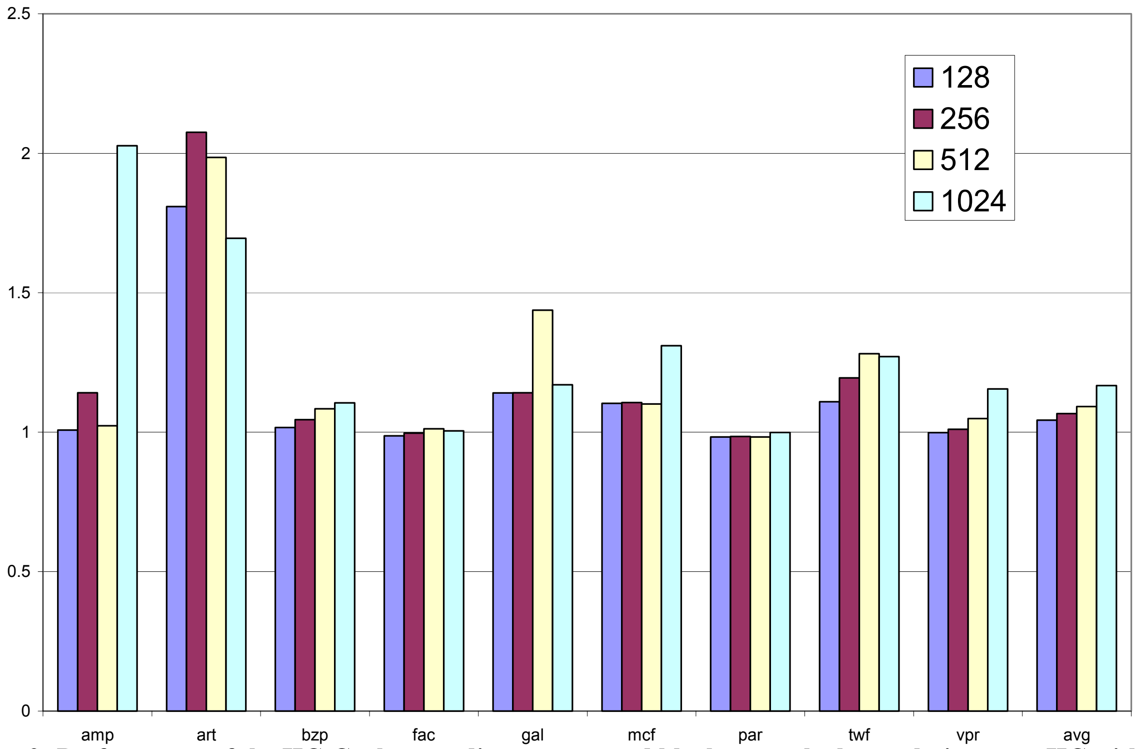

Figure 9: Performance of the IIC-C when sending compressed blocks over the bus, relative to an IIC with the same block size

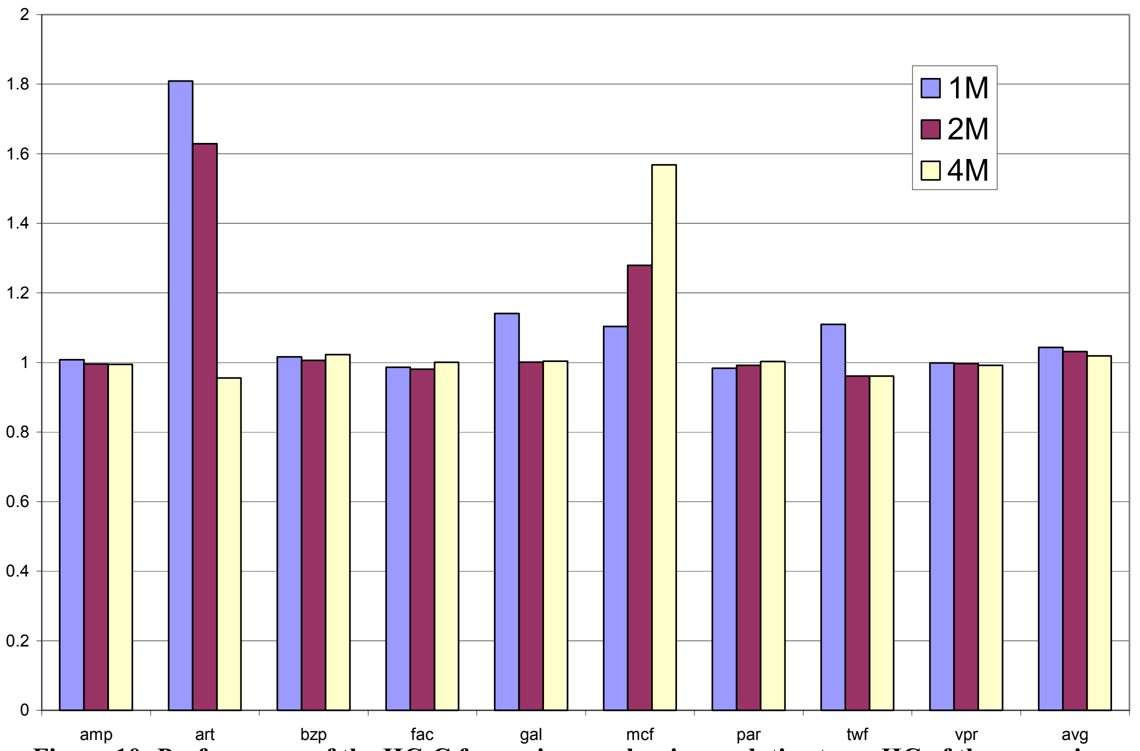

Figure 10: Performance of the IIC-C for various cache sizes, relative to an IIC of the same size

4.3.3 Combining cache and bus compression

We can further improve system performance by combining the IIC-C with a similarly compressed main memory and transfer data in a compressed form. Figure 9 presents our results for this system. As can be seen, combining the two solutions improves performance beyond either design alone. The maximum improvement of 107% is achieved by art at 256 byte blocks. All 26 benchmarks have an average improvement of 4% at 128 byte blocks, and 16% at 1024 byte blocks.

Figure 10 shows the effect of increasing the cache size from 1M to 4M. As expected, as the cache size grows greater than the working set size the usefulness of compression decreases. For art, the 2M IIC-C performs as well as a 4M IIC, but the compression penalties decrease performance by 5% at 4M. Twolf shows similar behavior at 2M. Mcf, on the other hand, shows increasing benefit from compression as more and more misses are removed by increasing the effective cache size. Galgel is interesting because while its working set is 8M, it too drops off at a 2M cache size. Referring back to Table 2, this is expected since 97% or the misses are removed by going to a 2M cache.

There is an inherent tradeoff on the number of subblocks (better compression resolution) and hardware overhead (more bits in the tags). To save space we tried reducing the number of subblocks from 4 to 2, effectively halving the overhead. Figure 11 presents the performance of a 2 subblock IIC-C relative to the 4 subblock IIC-C. Since a block needs to be compressed by 50% or more to be stored compressed with 2 subblocks we would expect the performance to decrease. Most of the benchmarks show a decrease in performance up to 24%. Facerec, mcf, nd parser actually show a slight increase in performance, for some block sizes. This is due to the fact that data that is stored compressed in the 4 subblock case is now stored uncompressed, removing the decompression latency from any access to that data. Increasing the number of subblocks to 8 or more should actually improve performance, at the expense of doubling or more the overhead. We considered the increased cost to be to great to explore this option.

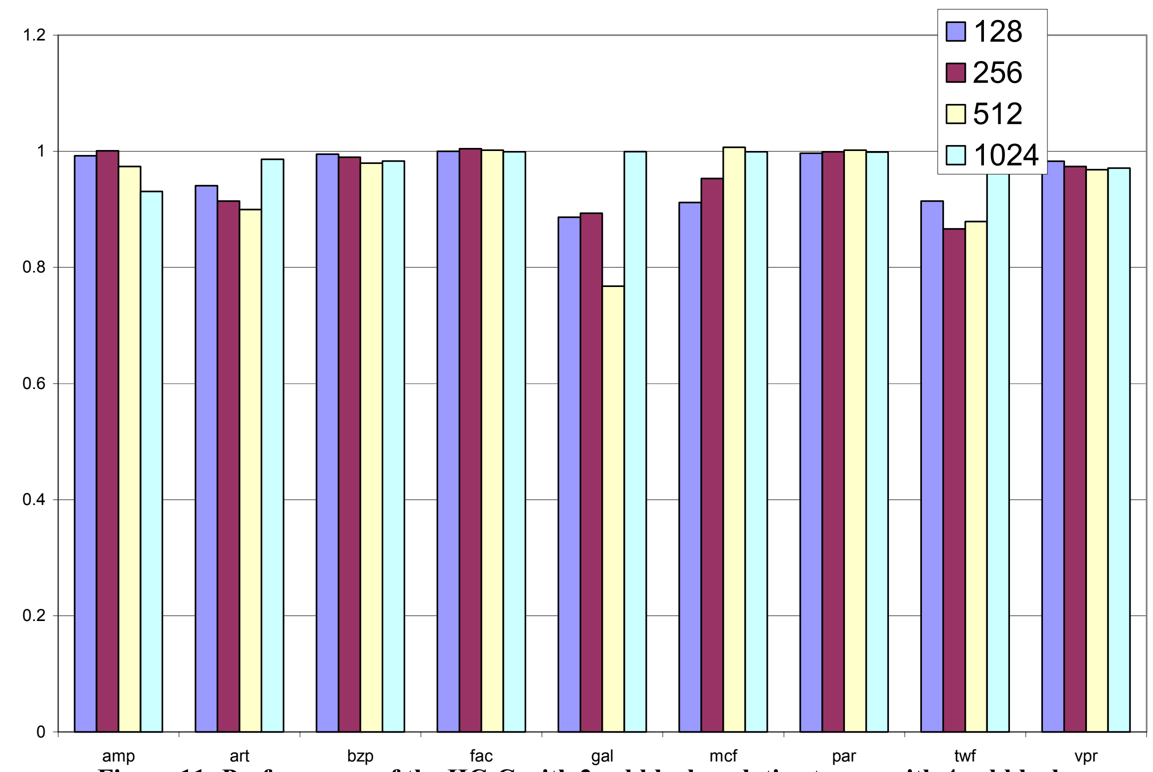

Figure 11: Performance of the IIC-C with 2 subblocks relative to one with 4 subblocks

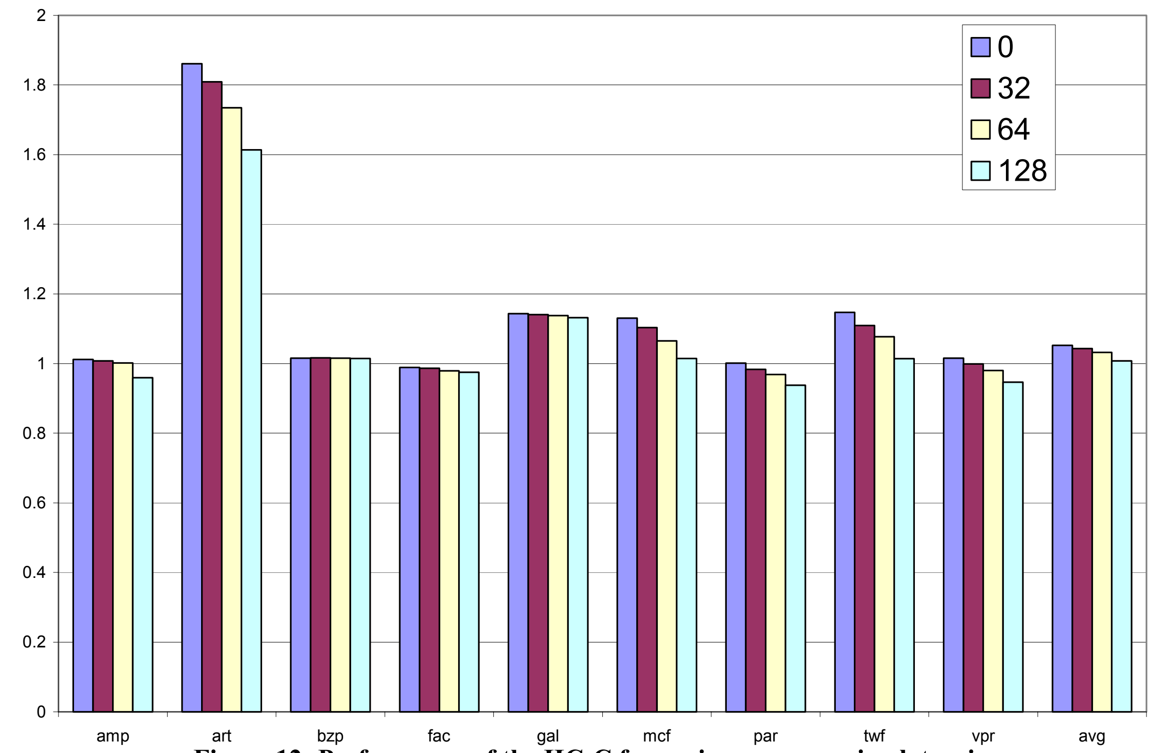

Figure 12: Performance of the IIC-C for various compression latencies

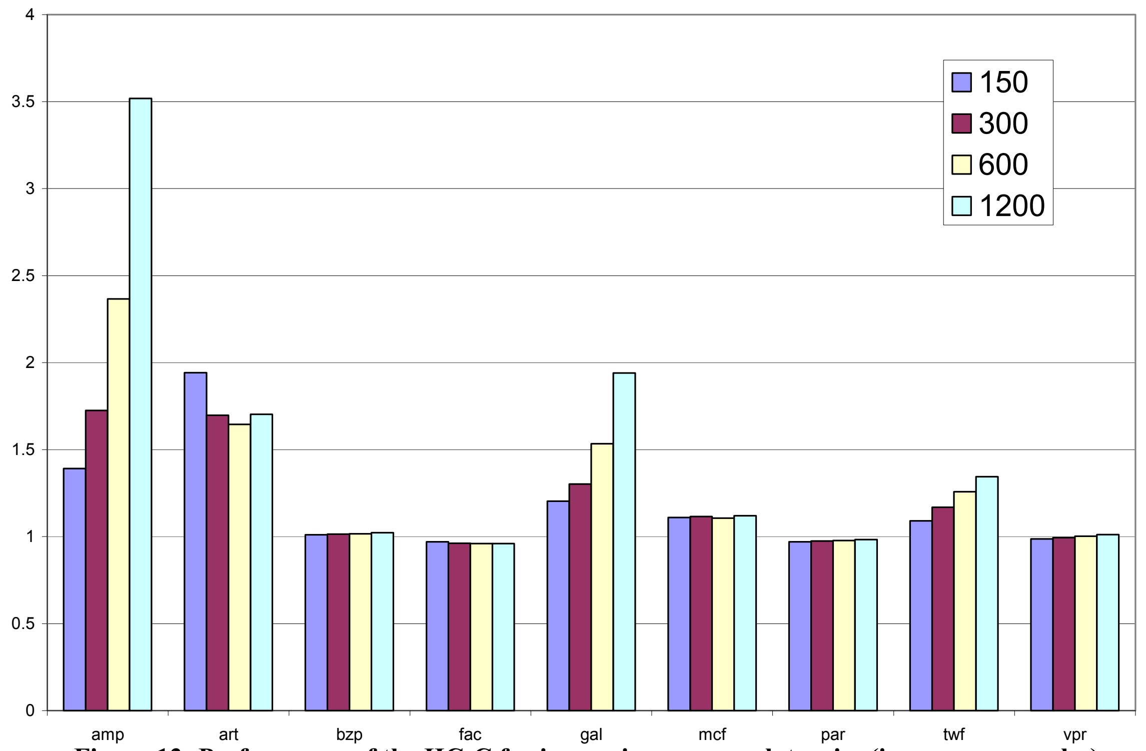

Figure 13: Performance of the IIC-C for increasing memory latencies (in processor cycles)

To test the sensitivity of our results to the compression/decompression latency, we ran experiments where we varied the compression latency from 0 to 128 cycles with a block size of 128B, keeping the decompression latency at ¼ of the compression latency. As can be seen in Figure 12, these experiments showed the expected trend that performance decreases with higher compression latencies. Bzip actually showed a very slight increase in performance as the latency increases due to the fact that it is thrashing in the L1 data cache; this effect is reduced at longer L2 miss latencies. It should be noted that, even with extremely long latencies, our scheme still shows a very slight improvement, with a maximum 7% decrease in performance.

As we head into the future, memory latencies will 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 with a 128B block size. While the cache will undoubtedly grow in future processors, we kept the on-chip cache latencies and sizes fixed, due to the limitations of our SPEC CPU2000 benchmark suite.

Figure 13 presents the performance of the compressed cache relative to a traditional LRU cache for the four memory latencies. As the memory latency increases the cost of an off-chip miss becomes more expensive. This decreases the impact of the compression latency while increasing the importance of a higher hit rate in the cache. These changes mean that compression becomes more and more desirable. Ammp, galgel, and twolf illustrate this as their performance increases from an average 23% to 127% as the latency increases. Mcf shows only a very slight increase in performance since the miss rate is reduced only by 6%, so the remaining misses drive the performance. Art is an anomaly. While compression still improves performance at all latencies, its performance falls off at a faster rate than the baseline. Further study is needed to unravel this anomaly.

5 Conclusion

In this paper we presented IIC-C, a feasible design for an on-chip compressible cache using the IIC. The IIC uses indirection to translate cache block addresses into the physical indexes in a data store where these blocks are held. Using this indirection, combined with sub blocking the cache, allows us to store these blocks in

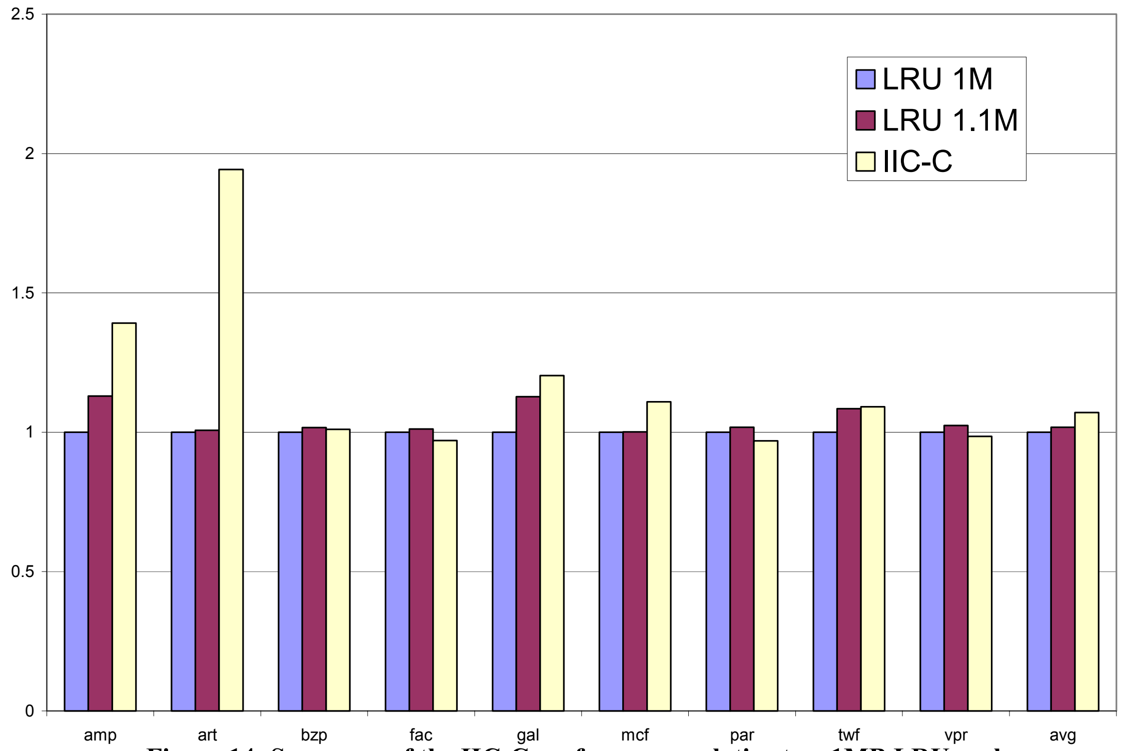

Figure 14: Summary of the IIC-C performance relative to a 1MB LRU cache

compressed form while using the saved space to store other blocks. This allows the cache to effectively increase its size when storing compressed data.

We evaluated the IIC-C using the SPEC2000cpu benchmark suite. Figure 14 presents a summary of our results for 128 byte cache blocks. The performance is normalized to a traditional LRU cache. We also compare to a 1.1M LRU cache to compare with the IIC-C area overhead. The IIC-C improves performance over a traditional LRU cache by up to 94% with an average improvement of 7%. We also find that using a compressed memory in concert with the IIC-C can help to alleviate bus contention by shipping compressed data. The IIC-C continues to increase its performance gains as the latency of memory increases, making it even more desirable in future processors.

References

[1] B. Abali, H. Franke, S. Xiaowei, et.al., “Performance of Hardware Compressed Main Memory”, The Seventh International Symposium on High-Performance Computer Architecture, 2001, (HPCA 2001), pp. 73-81.

[2] A. Alameldeen, D. Wood, “Adaptive Cache Compression for High-Performance Processors”, To appear in the 31st Annual International Symposium on Computer Architecture, June 2004.

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

[4] N. Binkert, E. Hallnor, S. Reinhardt, ”Network-Oriented Full-System Simulation using M5”, Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), February 2003

[5] E. Hallnor, S. Reinhardt, “A Fully Associative Software-Managed Cache Design”, In the proceedings of the 27th Annual International Symposium on Computer Architecture, pages 107--116, June 2000.

[6] G. Hammond, S. Naffziger, “Next Generation Itanium Process Overview”, Intel Developers Forum, 2001

[7] P. Franaszek, J. Robinson, J. Thomas, “Parallel Compression with Cooperative Dictionary Construction”, In the Proceedings of the Data Compression Conference, 1996, pp. 200 – 209.

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

[9] D. Kirovski, J. Kin, W.H. Mangione-Smith, “Procedure Based Program Compression”, In the proceedings of the 30th annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 204 – 213.

[10] M. Kjelso, M. Gooch, S. Jones, “Design and Performance of a Main Memory Hardware Data Compressor”, In the proceedings of the 22nd EUROMICRO Conference, Beyond 2000: Hardware and Software Design Strategies, 1995, pp. 423 – 430.

[11] 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.

[12] C. Lefurgy. E. Piccininni, T. Mudge, “Evaluation of a High Performance Code Compression Method”, In the proceedings of the 32nd Annual International Symposium on Microarchitecture, 1999, (MICRO-32), pp. 93 – 102.

[13] H.Lekatsas, W. Wolf, “Code Compression for Embedded Systems”, In the proceedings of the 35th Design Automation Conference, 1998.

[14] S. Roy, R. Kumar, M. Prvulovic, “Improving System Performance with Compressed Memory”, In the proceedings of the 15th International Parallel and Distributed Processing Symposium, Apr 2001, pp. 630 – 636.

[15] T. Sherwood, E. Perelman, G. Hamerly, B. Calder. “Automatically Characterizing Large Scale Program Behavior,” In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), October 2002. San Jose, California.

[16] L. Villa, M. Zhang, K. Asanovic, “Dynamic Zero Compression for Cache Energy Reduction”, In the proceedings of the 33rd International Symposium on Microarchitecture, Dec 2000.

[17] P. R. Wilson, S. F. Kaplan, Y. Smaragdakis, “The Case for Compressed Caching in Virtual Memory Systems”, In the proceedings of USENIX 1999.

[18] J. Yang, R. Gupta, “Energy Efficient Frequent Value Data Cache Design”, In the proceedings of the 35th Annual International Symposium on Microarchitecture, 2002, (MICRO-35)

[19] Y. Zhang, J. Yang, R. Gupta, “Frequent Value Locality and Value-Centric Data Cache Design”, In the proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, Nov. 2000