Interactions Between Compression and Prefetching in Chip Multiprocessors

Alaa R. Alameldeen*
Oregon Microarchitecture Lab
Intel Corporation
alaa.r.alameldeen@intel.com

David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
david@cs.wisc.edu

Abstract

In chip multiprocessors (CMPs), multiple cores compete for shared resources such as on-chip caches and off-chip pin bandwidth. Stride-based hardware prefetching increases demand for these resources, causing contention that can degrade performance (up to 35% for one of our benchmarks).

In this paper, we first show that cache and link (off-chip interconnect) compression can increase the effective cache capacity (thereby reducing off-chip misses) and increase the effective off-chip bandwidth (reducing contention). On an 8-processor CMP with no prefetching, compression improves performance by up to 18% for commercial workloads. Second, we propose a simple adaptive prefetching mechanism that uses cache compression's extra tags to detect useless and harmful prefetches. Furthermore, in the central result of this paper, we show that compression and prefetching interact in a strong positive way, resulting in combined performance improvement of 10-51% for seven of our eight workloads.

1 Introduction

Chip multiprocessors (CMPs) have emerged as a dominant system design paradigm. However, a good CMP design must balance cores, caches and communication to prevent one resource from becoming the only bottleneck. This presents a challenge, since the 2004 ITRS Roadmap [22] predicts that transistor performance will continue to improve faster than DRAM latency and pin bandwidth (26%, 10%, and 11% per year, respectively). To offset such trends, CMP designs are likely to dedicate significant area to on-chip caches and use latency hiding techniques like hardware prefetching [12, 27, 37, 38, 48]. Many current CMP designs (e.g., IBM's Power5 [39]) implement hardware stride-based prefetching, which eliminates some off-chip misses and overlaps the latencies of others.

Unfortunately, hardware prefetching significantly increases demand on off-chip pin bandwidth. It also increases a workload's working set size, possibly increasing cache pollution. CMPs exacerbate both these problems since more processors share cache and pin bandwidth resources. Figure 1 shows that for a uniprocessor, hardware stride-based prefetching achieves a 74% performance improvement for zeus (described in Section 4). But for a 16-processor CMP with the same cache size and pin bandwidth, stride-based prefetching degrades performance by 8%. For most of our benchmarks, the benefit of stride-based prefetching decreases as the number of processor cores grows, eventually degrading performance.

Compression can mitigate the main factors that degrade prefetching's performance. Cache compression (previously proposed for uniprocessors [4, 10, 23, 32, 34, 45, 47]) stores compressed lines in the cache, thereby increasing the effective cache size for a fixed area. This helps eliminate some cache misses, reducing both average memory latency and contention for off-chip pin bandwidth. Link compression [9, 13, 14, 19, 30] compresses data sent on the off-chip interconnect, increasing the effective pin bandwidth. However, compression introduces latency overheads to compress and decompress data, partially reducing its performance benefits.

This paper examines two complementary designs to alleviate demand on cache size and pin bandwidth. First, we study a CMP system design that supports both cache and link compression. We show that cache compression improves CMP performance by increasing the effective cache size, thereby reducing miss rates and decreasing pin bandwidth demand. Link compression improves performance by increasing the effective pin bandwidth. For zeus (Figure 1), cache and link compression alone increase performance by 6-12% over the base system (without prefetching or compression). Second, we propose an adaptive prefetching mechanism that uses compression's extra cache tags and simple heuristics to throttle prefetching when it replaces more useful lines than it brings in. Figure 1 shows that adaptation changes prefetching from an 8% degradation to a 16% performance improvement on 16 processors.

This paper also shows that compression and hardware prefetching interact positively. First, link compression reduces the contention for off-chip bandwidth caused by prefetching. Second, L1 prefetching hides part of the decompression penalty incurred by cache compression. Third, cache compression increases the effective cache capacity, which partially offsets prefetching's larger

* This work was largely done while a Ph.D. student at Wisconsin.

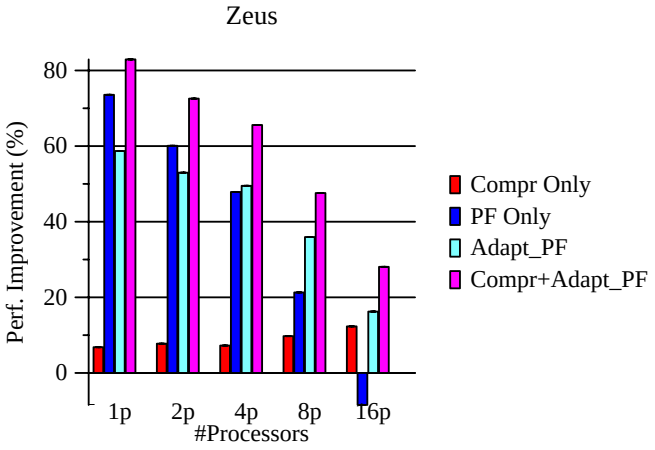

Figure 1. Benefits of Compression and Prefetching

working set size. Cache compression and prefetching also have a small negative interaction factor, since both techniques eliminate some of the same misses. The positive interactions generally outweigh the negative interactions, resulting in a combined performance improvement greater than the product of the individual improvements. Figure 1 shows that the combination of prefetching and compression improves the performance of zeus by 28% for a 16-processor CMP. This reflects a positive interaction of 24% (i.e., the speedup of prefetching and compression together exceeds the product of the speedup of prefetching alone and the speedup of compression alone by 24%).

Contributions. This paper makes four main contributions:

- It presents the first quantitative evaluation of cache and link compression on shared CMP caches. On an 8-processor CMP, this paper shows that cache compression alone increases the effective shared cache capacity by 36-80%, reduces misses by up to 23%, and improves performance by up to 18% for commercial benchmarks. It demonstrates that link compression alone reduces pin bandwidth demand by 34-41% for commercial benchmarks and up to 23% for scientific benchmarks.
- It presents the first quantitative evidence that stride-based prefetching improves the performance of CMPs far less than it does for uniprocessors, even degrading performance in some cases.
- It proposes a simple adaptive prefetching scheme that uses cache compression's extra address tags to detect useless and harmful prefetches. Compared to non-adaptive prefetching, the adaptive mechanism improves performance by 12-34% for commercial workloads and by 0-2% for SPEComp workloads on an 8-processor CMP.
- It shows that compression and prefetching interact in strongly positive ways, resulting in a combined performance improvement of 10-51% for seven of our eight workloads on an 8-processor CMP.

The remainder of the paper describes a CMP system using hardware stride-based prefetching and cache and link compression (Section 2) and our proposed adaptive prefetching scheme (Section 3). Full-system simulation of an 8-processor CMP running commercial and scientific workloads provide a quantitative evaluation for compression and prefetching (Section 4), and their interactions (Section 5). We finally discuss related work (Section 6) and conclude (Section 7).

2 A CMP with Prefetching and Compression

Base Design. The base system design is an eight-processor CMP with single-threaded cores, illustrated in Figure 2. Each core has private L1 instruction and data caches. All cores share an eight-banked L2 cache interleaved using the least significant block address bits.

An MSI protocol maintains coherence between the L1's and the shared L2. The L2 cache maintains inclusion and has full knowledge of on-chip L1 sharers via individual bits in its cache tag. L1 caches are write-back and only communicate with memory through the shared L2 cache. An off-chip memory controller is accessed via the memory interface.

Cache Compression. To increase effective cache capacity, and thus reduce off-chip misses, we use the decoupled variable-segment cache to pack more compressed cache lines into each L2 set [4]. We use the Frequent Pattern Compression scheme [1] to compress cache lines. The L1 caches hold uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. Accessing compressed L2 lines incurs a five cycle decompression penalty prior to insertion into the L1 cache, while uncompressed L2 lines may bypass the decompression pipeline. Both L1 and L2 caches use 64-byte lines.

In our implementation, each set in the L2 cache contains data space for 4 uncompressed lines but has 8 address tags. Thus compression can at most double the capacity and can increase the associativity from 4-way to 8-way. Each set divides its data space into 64 8-byte segments. Uncompressed blocks use eight segments; compressed blocks use between one and seven segments. We extend each address tag with a compression tag that indicates how many segments are allocated for the corresponding line. The additional state adds little storage overhead; roughly 7% more than a 4-way uncompressed cache [4].

We also implement an adaptive compression algorithm that dynamically compresses lines only when the benefit of compression (i.e., reduced misses) outweighs the cost (i.e., increased L2 hit latency due to decompression) [4]. However, for the workloads used in this study, the policy always adapted to compress all compressible cache lines.

Link Compression. To increase effective pin bandwidth, we compress cache blocks prior to sending/receiving them to/from the off-chip memory controller. Both the on-chip memory interface and the off-chip memory controller must support variable-length compressed message formats. We use the same segmentation scheme and compression algorithm as in the L2 cache. Each data message that originally included

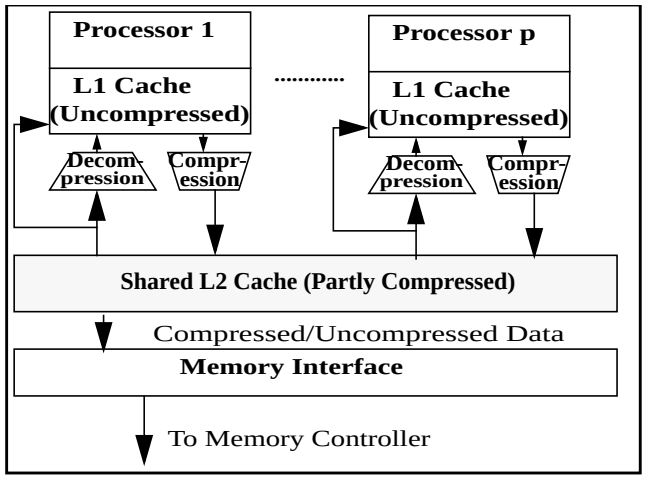

Figure 2. A Single-Chip CMP with compression support.

a complete cache line is transferred in 1-8 sub-messages (flits), each containing an 8-byte segment. The message header contains a length field indicating the number of segments in the line. Link compression increases effective pin bandwidth, reducing contention for this shared resource.

Memory Interface. Memory compression has previously been proposed to increase the effective memory capacity and reduce overall system cost [42]. However, since this study focuses on the cache hierarchy, we use a simpler memory interface that does not increase the effective memory capacity. Each 64-byte cache line is stored in memory using the form—uncompressed or compressed—that the processor sends across the memory interface, with a bit encoded in the ECC to indicate this meta information. An alternative scheme proposed by Ekman and Stenstrom [18] also uses the same frequent pattern compression algorithm to store cache lines in memory. Both schemes have the advantage of being transparent to software. Cache and link compression can also be used with IBM's MXT scheme [42] with additional complexity due to the differences in compression algorithms and granularities used in caches and memory.

Stride-Based Prefetching. We use L1 and L2 prefetchers based on the IBM Power 4 implementation [41]. Each processor has three associated prefetchers for the L1I, L1D and L2 caches. Each prefetcher contains three separate filter tables: positive unit stride, negative unit stride, and non-unit stride. Once a filter table entry detects a miss stream, the prefetcher allocates a stream table entry and initiates a number of startup prefetches. Each prefetcher issues prefetches for both loads and stores because our target system uses an L1 write-allocate protocol supporting sequential consistency (unlike Power 4). We also model separate L2 prefetchers per processor rather than a single shared prefetcher to reduce stream interference [7] and we allow L1 prefetches to trigger L2 prefetches.

3 Adaptive Prefetching

Hardware stride-based prefetching can improve performance of many workloads by eliminating many cache misses or hiding part of their latency. However, inaccurate or early prefetches may replace useful lines and/or waste pin bandwidth. Srinivasan, et al.'s prefetching taxonomy identifies nine cases based on the outcome of a prefetched block and its victim [40]. Only two of the nine cases actually eliminate misses. The others increase traffic (e.g., a useful prefetch replaces a live line) or increase both misses and traffic (e.g., a useless prefetch replaces a live line). In this paper, we use cache compression's extra tags to estimate when prefetches increase misses and/or traffic, and we use this information to throttle the basic stride-based prefetcher.

Our adaptive prefetching scheme uses a single saturating counter per cache (i.e., a counter for each L1 cache and a counter for the shared L2 cache) that is used to increment or decrement the number of startup prefetches per prefetching stream. When prefetching helps performance, these counters saturate at their maximum value and the prefetchers behave in the normal way. Useless and harmful prefetches decrement the counter, which disables prefetching completely (for the corresponding cache) when it reaches zero. All counters begin at their maximum values and are incremented/decremented by one.

To detect useless prefetches, we add a single "prefetch" bit per cache tag. This bit is set when a prefetched line enters the cache and is reset on the first access to the line. A prefetch is considered useless if it is evicted while the prefetch bit is still set. To detect harmful prefetches, we use the extra tags provided for cache compression. These tags record the addresses of replaced blocks, allowing us to detect when a prefetch may have evicted a useful line. Counters are updated on cache accesses as follows:

Cache hit. If the accessed line's prefetch bit is set, then a useful prefetch occurred and we increment the counter.

Cache replacement. If the allocated line (demand miss or prefetch) replaces a line whose prefetch bit is set (i.e., has not been accessed), then the earlier prefetch was useless and we decrement the counter.

Cache miss. We examine all the invalid tags in the cache set in LRU stack order. If the tag matches, then the line was replaced by one of the currently cached lines. If the prefetch bits of any valid lines are set, we make the conservative assumption that the line was victimized by a harmful prefetch, and we decrement the counter.

While additional state could more accurately distinguish the different cases, we show in the next sections that this simple mechanism greatly reduces the number of useless and harmful prefetches.

Table 1. Simulation Parameters

Processor Cores	Eight processors, each a single-threaded core with private L1 caches
Private L1 Caches	Split I & D, each 64 KB 4-way set associative with LRU replacement, 64-byte lines, 3-cycle access time, 320 GB/sec. total on-chip bandwidth (from/to L1's)
Shared L2 Cache	Unified 4 MB, composed of eight 512 KB banks, 8-way set associative (uncompressed) or 4-8 way set associative (compressed) with LRU replacement, 64-byte lines, 15 cycle uncompressed hit latency (includes bank access latency), 20 cycles compressed hit latency (15 + 5 decompression cycles)
Memory Configuration	4 GB of DRAM, 400 cycles access time with 20 GB/sec. chip-to-memory bandwidth. Each processor can have up to 16 outstanding memory requests
Processor Model	Each processor is an out-of-order superscalar processor with a 5 GHz clock frequency.
Processor Pipeline	4-wide fetch and issue pipeline with 11 stages (or more): fetch (3), decode (4), schedule (1), execute (1 or more), retire (2)
IW/ROB	64-entry instruction window, 128-entry reorder buffer
Branch Prediction	4 KB YAGS direct branch predictor [17], a 64-entry cascaded indirect branch predictor [16], and a 64-entry return address stack predictor [29]
Stride-based Prefetching	Each processor has three associated prefetchers for the L1I, L1D and L2 caches. Each prefetcher contains three separate 32-entry filter tables: positive unit stride, negative unit stride, and non-unit stride. Filter table entries allocate a miss stream into an 8-entry stream table when it recognizes 4 fixed-stride misses. Upon allocation, the L1I or L1D prefetcher launches 6 consecutive prefetches (at most for the adaptive scheme) along the stream to compensate for the L1 to L2 latency, while the L2 prefetcher launches 25 (at most) consecutive prefetches to memory.

4 Evaluation

In this section, we describe our evaluation methodology and evaluate the individual impact of compression and the base and adaptive stride-based prefetching schemes on the performance of our baseline CMP.

4.1 Methodology

Base System Configuration. We evaluate the performance of the compression and prefetching schemes on an 8-processor CMP with a 5 GHz clock and SPARC V9 processors. We used the Simics full-system simulator [35], extended with GEMS [36] (a detailed memory system timing and out-of-order processor simulator). Table 1 presents our basic simulation parameters.

Workloads. To evaluate alternative compression and prefetching schemes, we use four multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [2] and four benchmarks from the SPEComp2001 suite [5] (Table 2). The SPEComp benchmarks are compiled using C and Fortran compilers which implement software prefetching using the SPARC prefetch instructions. The GEMS simulator implements these prefetch instructions as non-blocking loads, so misses caused by them are largely indistinquishable from those caused by load and store instructions. These workloads cover a wide range of compressibility properties, miss rates and working set sizes. For each data point, we present the average and the 95% confidence interval of multiple simulations to account for space variability [3].

4.2 Cache and Link Compression

To compare the relative benefits of cache and link compression, we monitor their separate and combined effects on miss rates, off-chip bandwidth and performance for our base 8-processor, 4 MB L2 configuration. We evaluate compression in the absence of hardware prefetching in this section.

Table 2. Workload Descriptions

OLTP. Based on the TPC-C v3.0 benchmark, oltp uses IBM's DB2 v7.2 EEE DBMS with a 5 GB database (25,000 scaled warehouses). We simulate 16 users per processor, warm up for 100,000 transactions, and measure 100 transactions.
SPECjbb. SPECjbb2000 runs on Sun's HotSpot 1.4.0 Server JVM using 1.5 threads and 1.5 warehouses per processor and ~44 MB of data. We warm up for 200,000 transactions and measure 2,000 transactions.
Apache. Apache 2.0.43 serves 20,000 files (~500 MB) and SURGE [6] simulates 400 clients per processor, each with 25 ms think time. We warm up for ~2 million requests and measure 500 requests.
Zeus. Zeus is an event-driven static web server using the same client and data configuration as Apache.
SPEComp. The four SPEComp2001 benchmarks use the reference input set and are fast-forwarded to the beginning of the main loop. We warm up caches for approximately 2 billion instructions and measure until the end of the loop iteration.

Workload Compressibility. Our eight workloads show a wide range of compression ratios (Table 3). We obtained these ratios by periodically measuring the average effective cache size for each simulation and comparing it to an uncompressed 4 MB cache. The compression ratios for the commercial benchmarks (on the left) were relatively high, up to 1.8 which translates to an effective cache size of approximately 7.2 MB. However, the SPEComp benchmarks showed smaller gains, with average compression ratios ranging from

Table 3. Compression Ratios for a 4MB cache for commercial and SPEComp benchmarks

Benchmark	apache	zeus	oltp	jbb	art	apsi	fma3d	mgrid
Compr. Ratio	1.74	1.80	1.36	1.48	1.15	1.01	1.19	1.02

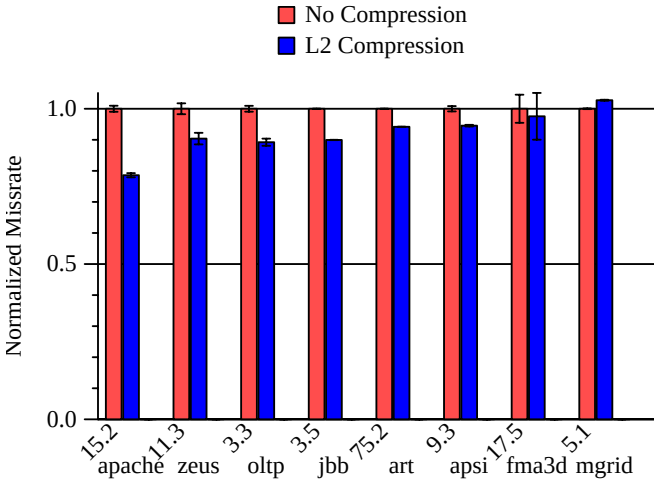

Figure 3. Normalized L2 cache miss rate (Misses per 1000 instructions shown below for no compression).

1.01 to 1.19. Lossless compression of floating-point data remains a hard problem even for more complex software schemes [44]. Most of the benefit for floating-point applications comes from compressing zeros [1].

Reduction in Cache Misses. Compression significantly increases the effective cache size for six of the eight benchmarks, leading to miss rate reductions. The commercial benchmarks reduce their miss rates by 10% to 23% (Figure 3). The miss rate reductions for the SPEComp benchmarks are substantially less, in keeping with their lower compression ratios. However, the relationship between miss rate and effective cache size is less clear: the apsi miss rate improves by 5%, despite only a 1% increase in effective cache size; the fma3d miss rate doesn't improve, despite a 19% increase in effective cache size. It is well known that a workload's miss rate can differ dramatically depending upon whether or not a critical working set fits in the cache. Our earlier uniprocessor cache compression study observed this phenomenon as well [4].

Bandwidth Reduction. By reducing cache misses, cache compression also reduces *pin bandwidth demand*, defined as the bandwidth utilization on a system with infinite available pin bandwidth. Link compression reduces pin bandwidth demand by transferring a cache block using fewer flits. Figure 4 presents the bandwidth demand for our benchmarks with no compression, only cache compression, only link compression, and both types of compression. Without compression, the average bandwidth demand of the commercial workloads ranges from 5.0 GB/sec. for oltp to 8.8 GB/sec. for apache. For SPEComp benchmarks, the bandwidth demands trend higher, ranging from 7.6 GB/sec. for art to 27.7 GB/sec. for fma3d. Although six of the eight workloads have average bandwidth demand less than the 20 GB/sec. available on the baseline system, they may still benefit from bandwidth reduction since cache misses typically occur in bursts.

Cache compression reduces bandwidth 5% to 10% for the commercial workloads and 0% to 10% for the SPEComp benchmarks. Link compression reduces bandwidth 34-41% for the commercial benchmarks and up to 23% for the SPEComp benchmarks. Only apsi, whose compression ratio is 1.01, fails to achieve at least a 17% reduction. Combining cache and link compression performs slightly better than link compression alone, reducing bandwidth 35-45% for the commercial benchmarks and up to 23% for SPEComp benchmarks.

Note that the bandwidth reduction due to cache compression is generally smaller than the miss ratio reduction. This is because cache compression affects two terms in the bandwidth equation:

$$BandwidthDemand(bytes/second) = (bytes/miss) \times (misses/instruction) \times (instructions/cycle) \times (cycles/second) \quad \text{(EQ 1)}$$

Cache compression reduces the number of *misses/instruction*, which reduces bandwidth demand. Conversely, reducing

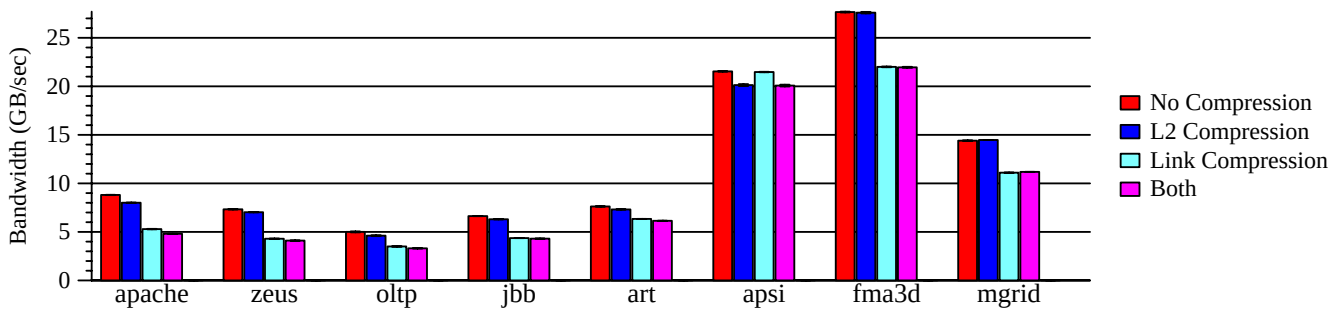

Figure 4. Pin bandwidth demand (GB/sec.).

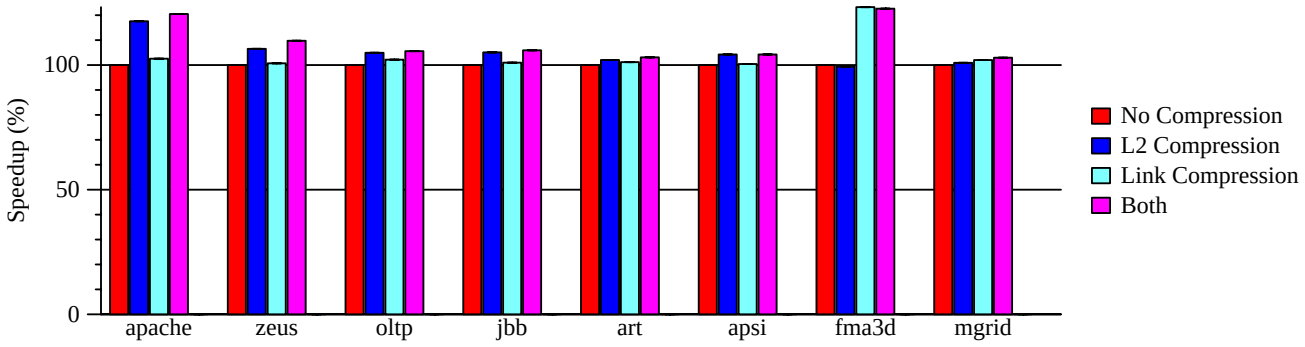

Figure 5. Speedup for compression alternatives.

misses usually improves *instructions/cycle*, which increases the bandwidth demand. The exact relationship between *misses/instruction* and *instructions/cycle* depends upon many factors, including the processor's ability to tolerate cache miss latency. However, we observe that these two effects largely offset each other for SPEComp, and our results show that cache compression has little impact on bandwidth reduction for these benchmarks, except apsi. For the commercial benchmarks, decreasing *misses/instruction* exceeds the impact of increasing *instructions/cycle* (which is reduced by decompression penalties), leading to a net decrease in bandwidth demand. Link compression reduces the *bytes/miss* term and has less direct impact on the other terms except for systems with high contention. Therefore, link compression always leads to a reduction in bandwidth demand for compressible benchmarks.

Performance. For the workloads where cache compression significantly reduces misses, it also significantly improves performance. Figure 5 plots the speedup of the compression schemes relative to no compression. Cache compression alone improves performance by 5-18% for the commercial workloads. Conversely, the less-compressible SPEComp benchmarks improve by 0-4%. With the base CMP configuration's relatively generous 20 GB/sec. pin bandwidth, link compression significantly improves performance only for fma3d, the benchmark with the highest bandwidth demand, achieving a 23% speedup. The combined impact of cache and link compression is slightly higher than that of cache compression alone (except for fma3d).

4.3 Prefetching

To characterize the benefit of hardware stride-based prefetching, we evaluate the coverage, accuracy and prefetching rate of the L1I, L1D, and L2 prefetchers. We then evaluate their impact on performance in the absence of compression.

Prefetching Characteristics. We analyze the characteristics of the hardware prefetchers using the following metrics:

$$\text{PrefetchRate} = \text{Prefetches /1000 inst.} = \frac{TotalPrefetches \times 1000}{TotalInstructions} \quad \text{(EQ 2)}$$

$$Coverage(\%) = \frac{PrefetchHits}{PrefetchHits + DemandMisses} \times 100\% \quad \text{(EQ 3)}$$

Where a prefetch hit is defined as the first reference to a prefetched block, excluding partial hits where a block is still in flight.

$$\text{Accuracy}(\%) = \frac{PrefetchHits}{TotalPrefetches} \times 100\% \quad \text{(EQ 4)}$$

Table 4 shows the prefetching characteristics for an 8-processor CMP[1]. We note several differences between the commercial and SPEComp benchmarks. Beginning with the L1

Table 4. Prefetching Properties for Commercial and SPEComp Benchmarks

Benchmark	L1 I Cache			L1D Cache			L2 Cache		
	Pf rate	Coverage	Accuracy	Pf rate	Coverage	Accuracy	Pf rate	Coverage	Accuracy
apache	4.9	16.4%	42.0%	6.1	8.8%	55.5%	10.5	37.7%	57.9%
zeus	7.1	14.5%	38.9%	5.5	17.7%	79.2%	8.2	44.4%	56.0%
oltp	13.5	20.9%	44.8%	2.0	6.6%	58.0	2.4	26.4%	41.5%
jbb	1.8	24.6%	49.6%	4.2	23.1%	60.3%	5.5	34.2%	32.4%
art	0.05	9.4%	24.1%	56.3	30.9%	81.3%	49.7	56.0%	85.0%
apsi	0.04	15.7%	30.7%	8.5	25.5%	96.9%	4.6	95.8%	97.6%
fma3d	0.06	7.5%	14.4%	7.3	27.5%	80.9%	8.8	44.6%	73.5%
mgrid	0.06	15.5%	26.6%	8.4	80.2%	94.2%	6.2	89.9%	81.9%

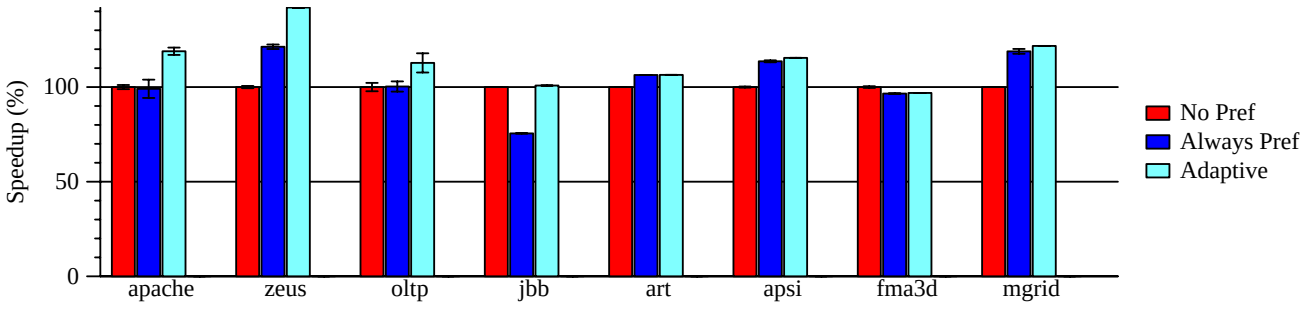

Figure 6. Prefetching speedup (%) (relative to no prefetching).

instruction prefetcher, the commercial workloads issue many more prefetches (up to 13.5 per 1000 instructions for oltp) than the SPEComp benchmarks (at most 0.06 per 1000 instructions). Despite the higher prefetch rate, both the coverage and accuracy are generally higher for the commercial workloads. However, because the prefetch streams are initiated only after recognizing four fixed-stride accesses, even the best case, jbb, only achieves 25% coverage and 50% accuracy. In contrast, the SPEComp benchmarks have generally higher prefetch rates, accuracy, and coverage for the L1 data and L2 prefetchers. In particular, the L2 prefetcher covers 45-92% of the misses with 74-98% accuracy for SPEComp, but only covers 26-45% of the misses with 32-58% accuracy for the commercial workloads. This confirms the conventional wisdom that scientific benchmarks have more regular access patterns than commercial workloads.

Performance. Hardware prefetching can improve performance by reducing cache misses and hiding miss latencies. Conversely, prefetching can degrade performance by replacing useful cache blocks and causing contention. Figure 6 shows the speedup due to hardware prefetching for our eight benchmarks. Prefetching improves performance for half of our benchmarks, with speedups of 21% for zeus and 19% for mgrid. On the other hand, prefetching degrades fma3d's performance by 3% and jbb's performance by 25%. Fma3d is bandwidth limited and the useless prefetches generated by its L2 prefetcher simply aggravates this bottleneck. For jbb, the low accuracy of the L2 prefetcher (32%) causes the replacement of many useful L2 cache lines.

The adaptive prefetcher eliminates many useless and harmful L2 prefetches, increasing accuracy across all the benchmarks. Speedup over no prefetching increases significantly for commercial benchmarks (jbb's 25% slowdown becomes a 0.8% speedup, apache's 0.9% slowdown becomes a 19% speedup, zeus's 21% speedup becomes 42%, and oltp's no speedup turns into a 12% speedup). For SPEComp applications, the performance improvement is limited since non-adaptive stride-based prefetching is generally highly accurate. Compared to non-adaptive prefetching, the adaptive mechanism improves performance by 12-34% for commercial workloads and by 0-2% for SPEComp workloads.

1. Uniprocessors have higher L1D and L2 coverage for commercial benchmarks due to having less thread contention. Other uniprocessor prefetching properties do not differ significantly from those of Table 4.

5 Interactions of Prefetching and Compression

In this section, we study the interactions between compression and prefetching. We use the following terminology derived from Fields, et al.'s interaction cost definition [21]. For an architectural enhancement A (e.g., L2 compression), we define its speedup for a certain workload, *Speedup(A)*, as the workload's runtime on a base system (without A) divided by the workload's runtime on the same system with enhancement A. For two architectural enhancements A and B (e.g., compression and prefetching), we define the combined speedup of the base system with both enhancements as:

$$Speedup(A, B) = Speedup(A) \times Speedup(B) \times (1 + Interaction(A, B)) \quad \text{(EQ 5)}$$

When *Interaction(A, B)* is positive, the speedup of the two enhancements together exceeds the product of individual speedups. We call this case a *positive interaction* between A and B. When *Interaction(A, B)* is negative, the speedup of the combined system is less than that of the product of individual speedups. We call this case a *negative interaction* between A and B. We next describe different factors that affect the interaction between prefetching and compression.

5.1 Pin Bandwidth Demand

Figure 7 shows the pin bandwidth demand of prefetching and compression combinations, normalized to the case of no compression or hardware prefetching. Stride-based prefetching alone increases off-chip bandwidth demand for our benchmarks by 23-206% (on a system with infinite pin bandwidth). Since our base system configuration has limited pin bandwidth, the increased demand may cause performance slowdowns due to queuing delays. Combining prefetching with cache and link compression reduces off-chip pin bandwidth demand for all benchmarks, a *positive interaction* between the two techniques. For example, L2 prefetching alone increases zeus's bandwidth demand by 98%, but when combined with cache and link compression bandwidth demand increases by only 14%. Similarly, art's 23% band-

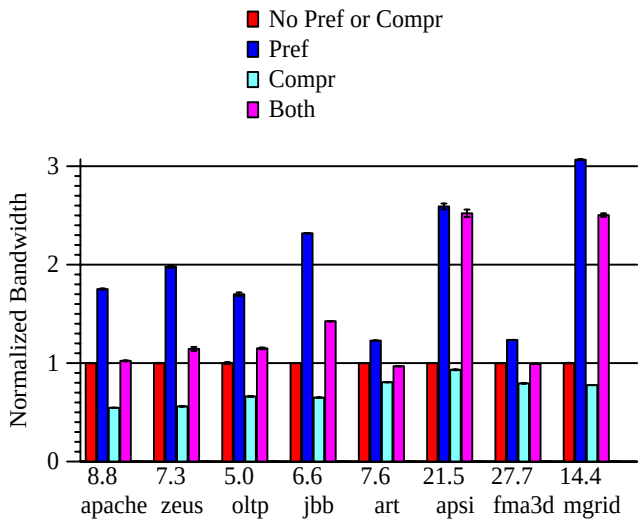

Figure 7. Bandwidth demand for prefetching and compression.

width demand increase due to prefetching turns into a 4% reduction when combined with cache and link compression.

By avoiding useless prefetches, our adaptive prefetching scheme significantly reduces bandwidth demand for commercial applications (with little effect on the mostly accurate prefetches of SPEComp). While the non-adaptive prefetcher increases commercial benchmarks' bandwidth demand by 70-132%, the adaptive prefetcher limits the bandwidth demand increase to 19-52% (not shown).

5.2 Classification of L2 Misses

Figure 8 classifies L2 misses according to whether prefetching or compression can avoid them. There are six classes of accesses (from bottom up): demand misses that cannot be avoided, demand misses avoided only by L2 compression, demand misses avoided only by L2 prefetching, demand misses avoided by either L2 compression or prefetching, prefetches not avoided by L2 compression, and prefetches avoided by L2 compression. The 100% line represents the total demand misses in the absence of compression or hardware prefetching. The figure presents approximate data estimated by comparing cache miss profiles across simulations of different configurations and using set theory and the theory of inclusion and exclusion to obtain cardinalities of different sets of accesses.

Figure 8 shows that L2 prefetching succeeds in avoiding many misses in SPEComp benchmarks, while L2 compression is not as successful. For commercial benchmarks, both prefetching and compression avoid significant numbers of L2 misses. We also note two sources of interaction:

Negative Interaction: Misses Avoided by Both. Figure 8 illustrates the intersection between the sets of misses avoided by L2 compression and L2 prefetching (i.e., misses that can be avoided by either technique). This intersection represents a negative interaction factor, since they can only be eliminated once in a system using both techniques. However, the

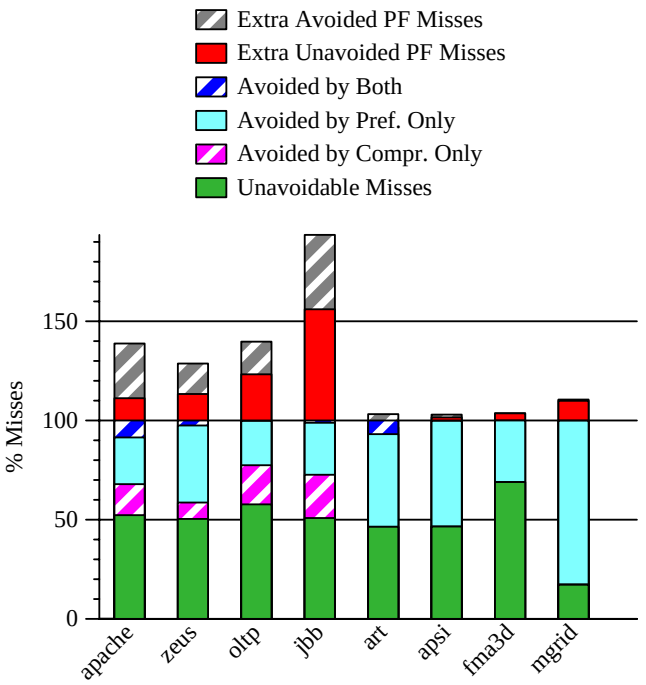

Figure 8. Breakdown of L2 cache misses and prefetches.

intersection represents a small fraction of all misses (8% for apache, 7% for art, and 3% or less for all other benchmarks). We attribute this small intersection to the fact that L2 compression and L2 prefetching target different sets of misses. L2 compression targets those conflict and capacity misses eliminated by going at most twice as far down the LRU stack. L2 prefetching targets capacity misses that follow a strided pattern, which may represent blocks much further down the LRU stack. The two sets of misses, while partially overlapping, are largely orthogonal.

Positive Interaction: Prefetching Misses Avoided by Compression. Figure 8 also shows that the extra capacity provided by cache compression helps avoids many of the additional misses caused by prefetching in the commercial workloads. Prefetching, in effect, increases a workload's working set size (or cache footprint) and compression helps by increasing the effective cache size to tolerate that increase in working set size.

5.3 L2 Hit Latency

Cache compression increases the average L2 hit latency for compressible benchmarks by 1.2-3.7 cycles, since hits to compressed lines suffer from an additional 5-cycle decompression overhead. L1 prefetching mitigates some of that impact by prefetching compressed lines into the L1 cache, a positive interaction between the two techniques. However, the percentage of penalized hits avoided by L1 prefetching is small (less than 12%) for most benchmarks due to the low L1 prefetching accuracy and coverage for commercial workloads, and the poor compressibility of SPEComp benchmarks (Table 4).

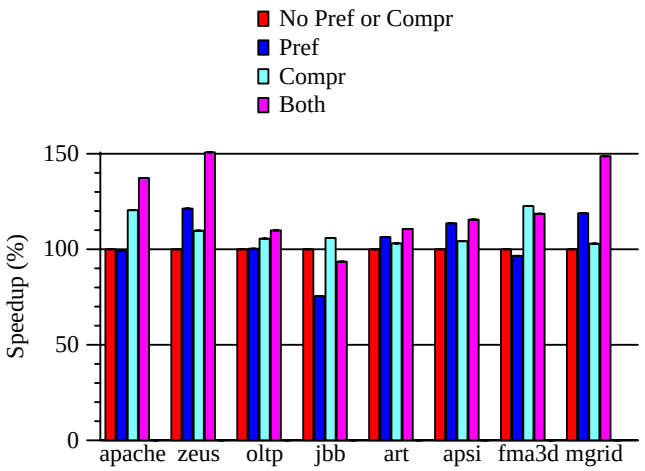

Figure 9. Speedup of prefetching and compression.

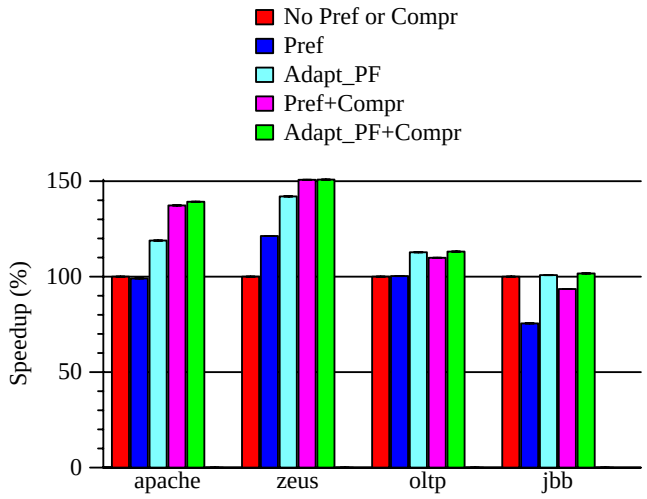

Figure 10. Speedup of prefetching and adaptive prefetching.

5.4 Performance

Figure 9 shows speedups for different combinations of prefetching and compression, relative to the base case of no prefetching or compression. Table 5 presents the speedups (shown as percentage improvement) and interaction coefficients for different combinations. Stride-based prefetching alone speeds up half the benchmarks, with zeus and mgrid improving roughly 20% each. Conversely, jbb and fma3d slow down 25% and 3%, respectively. Cache and link compression speed up all benchmarks, with apache and fma3d improving by more than 20%. Combining prefetching with compression achieves speedups of 10-51% across all benchmarks, except jbb. The two schemes interact positively for all benchmarks, except apsi (last row of Table 5). The interaction coefficients are as high as 22% (mgrid) and 17% (jbb) indicating that the additional cache capacity and pin bandwidth provided by compression significantly mitigates the impact of useless and harmful prefetches.

Turning to adaptive prefetching, Figure 10 compares the speedup of the base strided prefetcher to the adaptive policy, focusing on the commercial workloads where adaptation helps. Compared to prefetching alone, the adaptive policy dramatically improves performance. Zeus improves an additional 21% and apache and oltp (where the base prefetcher did not help) improve 20% and 12%, respectively. Most strikingly, jbb improves from a 25% performance degradation to a 1% gain.

For prefetching combined with compression, Figure 10 shows that adaptation has much less impact, with improvements ranging from 0.1% to 8%. Two factors contribute to this. First, as shown in Figure 8, compression eliminates many of the strided prefetches, significantly reducing the potential benefit. Second, the adaptive prefetcher uses unused cache compression tags to detect harmful prefetches. When cache compression is disabled, the adaptive algorithm has four extra tags per set. But with compression enabled, this drops dramatically for the highly-compressible commercial workloads (roughly one tag for apache and zeus and two for oltp and jbb). Increasing the number of tags would help address this second factor.

5.5 Sensitivity to Available Pin Bandwidth

Since prefetching increases bandwidth demand, we expect compression to have the strongest interaction for systems with limited bandwidth. Figure 11 presents the interaction terms as the available pin bandwidth varies from 10 to 80 GB/sec. For commercial benchmarks, the interaction term is large for the 10 and 20 GB/sec. configurations, up to 29% and 17%, respectively. The interaction drops dramatically for the 40 and 80 GB/sec. configurations since the available bandwidth significantly exceeds demand, even with prefetching

Table 5. Speedups and Interactions between Prefetching and Compression

Benchmark	apache	zeus	oltp	jbb	art	apsi	fma3d	mgrid
Speedup (Pref.)	-0.9%	21.3%	0.3%	-24.5%	6.4%	13.6%	-3.4%	18.9%
Speedup (Compr.)	20.5%	9.7%	5.6%	5.9%	3.1%	4.2%	22.6%	2.9%
Speedup (Pref., Compr.)	37.3%	50.7%	9.9%	-6.5%	10.6%	15.5%	18.6%	48.7%
Speedup (Adaptive-Pref, Compr.)	39.2%	50.8%	13.1%	1.7%	10.7%	16.1%	18.5%	49.9%
Interaction(Pref., Compr.)	**15.0%**	**13.2%**	3.8%	**16.9%**	0.9%	-2.5%	0.2%	**21.5%**

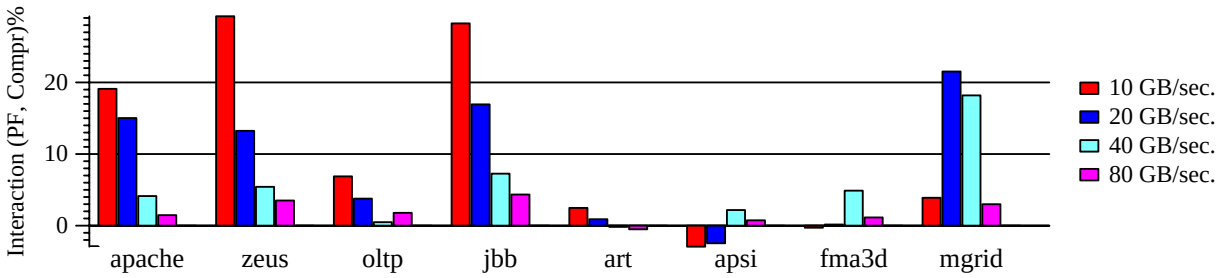

Figure 11. Interaction (%) between prefetching and compression for different pin bandwidth values (10 to 80 GB/sec.).

(Figure 7). For SPEComp benchmarks, the interaction is negative for some configurations since compression is less effective for these benchmarks. However, the negative interaction terms are limited to 3% or less. On the other hand, some configurations show significant positive interaction terms (up to 22% for mgrid) due to the impact of link compression.

5.6 Sensitivity to Number of CMP Cores

Stride-based prefetching has proven highly effective for uniprocessors and almost all commercial microprocessors implement some form of prefetching. Yet in a CMP, we have shown that prefetching increases contention for shared resources (caches and pin bandwidth), resulting in performance degradation for some workloads. Figure 1 (zeus) and Figure 12 (apache and jbb) illustrate how the impact of compression changes as a function of the number of processors (all other system parameters remain as in Table 1)[2]. Performance improvements (i.e., *Speedup - 100%*) are shown relative to a base system with the same number of processors.

For a uniprocessor, stride-based prefetching alone improves performance by 61%, 73% and 2% for apache, zeus and jbb, respectively. However, the benefit of prefetching decreases steadily for more cores. For a 16-processor CMP, stride-based prefetching provides no improvement for apache, and degrades the performance of zeus and jbb by 8% and 35%, respectively. Although adaptive prefetching always helps jbb, it degrades performance for apache and zeus on a uniprocessor (by about 8%). However, adaptive prefetching improves performance for these workloads on all CMP configurations with four processors or more. For 16-processor CMPs, performance improves 17% for apache, 16% for zeus, and jbb's degradation is reduced to 9%.

Cache and link compression alone achieve modest performance improvements for uniprocessors (20%, 7% and 6% for apache, zeus and jbb, respectively), consistent with earlier published results [4]. The improvement increases slowly for more cores (23%, 12%, and 10%, respectively for 16 processors). While compression alone provides relatively modest gains, the strong positive interaction with (non-adaptive and) adaptive prefetching results in significant performance improvements for 16-processor CMPs. Performance improves 39% for apache, 28% for zeus, and 2% for jbb.

These results clearly show that for larger CMPs, strided prefetching oversubscribes the critical cache and pin bandwidth resources. Adaptive prefetching and cache and compression provide mechanisms to increase the effective capacity of these resources. However, the strong positive interactions between these techniques strongly suggest that system designers should consider implementing both.

6 Related Work

This section briefly reviews the most directly related work.

Cache Compression. Cache compression increases a cache's effective capacity. Lee, et al. [32] propose a compressed mem-

2. OLTP shows similar (but smaller) performance improvements and degradations to other commercial benchmarks.

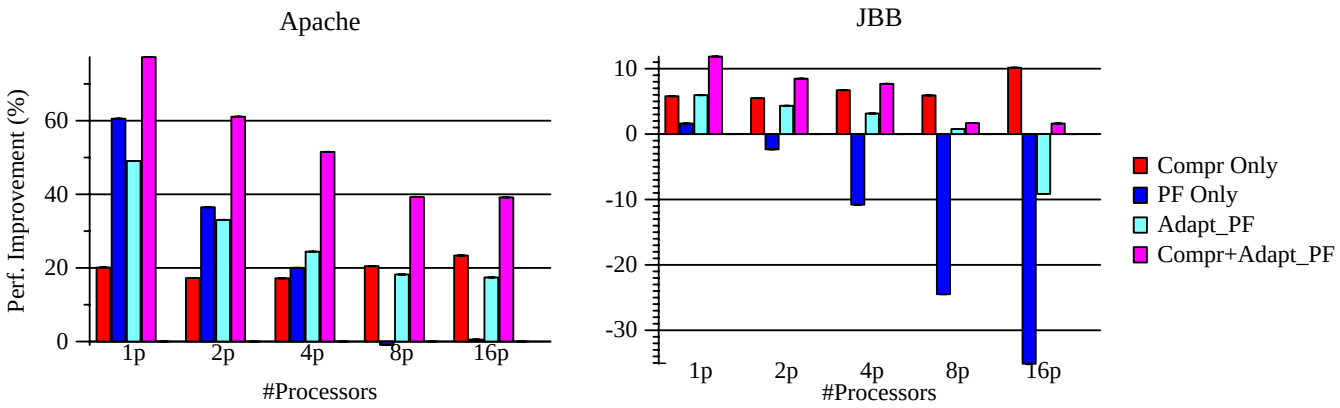

Figure 12. Performance improvement (%) for Apache and Jbb.

ory hierarchy model that selectively compresses L2 cache and memory blocks if they can be reduced to half their original size. Adaptive cache compression was proposed for uniprocessors to dynamically adapt to the costs and benefits of compression, and to compress only for workloads and workload phases when compression helps [4]. Hallnor and Reinhardt's Indirect-Index Cache decouples index and line accesses across the whole cache, allowing fully-associative placement and the storage of compressed lines [23]. This paper is the first to analyze using compression for shared CMP caches.

Link Compression. Address and data compaction have been proposed to increase the effective memory bandwidth [9, 13, 14, 19, 30]. Farrens and Park [19] use base registers to exploit the redundant information in the most-significant address bits. Kant and Iyer [30] studied the compressibility properties of address and data transfers in commercial workloads, and reported that the high-order bits can be predicted with high accuracy in address transfers but with less accuracy for data transfers. This paper is the first to study the impact of link compression on CMP systems.

Impact of Pin Bandwidth. Huh, et al. [26] identified pin bandwidth as a potential limiting factor for CMP performance, which they projected would force designers to increase the area allocated to on-chip caches at the expense of cores. This problem is exacerbated by hardware-directed prefetching schemes that target increasing the memory-level parallelism and reducing off-chip misses [8].

Hardware Prefetching. Hardware-directed prefetching has been proposed and explored by many researchers [12, 27, 37, 38, 48], and is currently implemented in many existing systems [24, 25, 39, 41]. Jouppi [28] introduced stream buffers that trigger successive cache line prefetches on a miss. Chen and Baer [12] proposed variations of stride-based hardware prefetching to reduce the cache-to-memory latency, and studied its positive and negative impacts on different benchmarks [11]. Dahlgren, et. al [15] proposed an adaptive sequential (unit-stride) prefetching scheme that adapts to the effectiveness of prefetching. Tullsen and Eggers [43] studied the negative side effects of software prefetching on bus utilization, cache miss rates and data sharing for a multiprocessor system, and proposed techniques to reduce some of these negative effects. Lin, et. al [20] mitigate the negative effects of prefetching on performance by prefetching only when the memory bus is idle (to reduce contention), and prefetching to lower replacement priorities than demand misses (to reduce cache pollution). Ki and Knowles [31] used extra cache bits to increase prefetching's accuracy. Srinivasan, et. al [40], classified prefetches according to whether they reduce or increase misses or traffic.

Prefetching and Compression. Zhang and Gupta [46] exploit their compressed cache design [47] to prefetch partial compressed lines from the next level in the memory hierarchy. Lee, et al. [32, 33, 34] use a decompression buffer between their cache levels to buffer decompressed lines, which can be viewed as storing "prefetched" uncompressed lines to reduce decompression overhead. This paper studies the interactions of a stride-based hardware prefetcher with cache and link compression in a CMP.

7 Conclusions

Chip multiprocessor design requires balancing demand on multiple critical resources, including the number of processor cores, on-chip cache size, and off-chip pin bandwidth. This paper shows that stride-based prefetching provides smaller performance improvements for CMPs than for uniprocessors, even hurting performance in some cases. We further show that cache and link compression partially compensate for the increased demand by effectively increasing cache size and pin bandwidth. For commercial workloads running on an 8-processor CMP, cache compression increases the effective capacity of the shared cache by up to 80%, thus reducing off-chip misses and improving performance by up to 18%. Link compression increases the effective off-chip communication bandwidth by up to 41%, reducing possible contention and further improving performance. We also propose a simple adaptive prefetching mechanism that uses compression's extra cache tags to throttle prefetching and improve performance of commercial workloads by 12-34%.

In a central result of this paper, we show that compression and prefetching have a strong positive interaction, improving performance by 10-51% for seven of our eight workloads on an eight-processor CMP. Compression and prefetching interact positively in three ways: link compression increases effective pin bandwidth; L1 prefetching hides part of the decompression penalty; and cache compression increases the effective cache capacity, mitigating the increase in working set size due to prefetching. The strong positive interaction between compression and prefetching suggest that CMP system designers should strongly consider implementing both.

Acknowledgements

We thank Dan Gibson, Kevin Moore, Andy Phelps, Min Xu and Luke Yen for their feedback. We also thank Brad Beckmann for his help with prefetching implementation. This work is supported in part by the National Science Foundation with grants CCR-0324878, EIA-0205286, and EIA-9971256, a Wisconsin Romnes Fellowship (Wood) and donations from IBM, Intel and Sun Microsystems. Prof. Wood has a significant financial interest in Sun Microsystems, Inc.

References

[1] Alaa R. Alameldeen. *Using Compression to Improve Chip Multiprocessor Performance*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 2006.

[2] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, February 2003.

[3] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proc. of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, February 2003.

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

[5] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[6] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[7] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[8] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. In *Proc. of the 23th Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[9] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very Low Power Pipelines Using Significance Compression. In *Proc. of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 181–190, December 2000.

[10] David Chen, Enoch Peserico, and Larry Rudolph. A Dynamically Partitionable Compressed Cache. In *Proc. of the Singapore-MIT Alliance Symposium*, January 2003.

[11] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[12] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[13] Daniel Citron. Exploiting Low Entropy to Reduce Wire Delay. *IEEE TCCA Computer Architecture Letters*, 3, January 2004.

[14] Daniel Citron and Larry Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proc. of the First IEEE Symposium on High-Performance Computer Architecture*, pages 90–99, February 1995.

[15] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[16] Karel Driesen and Urs Holzle. Accurate Indirect Branch Prediction. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 167–178, June 1998.

[17] Avinoam N. Eden and Trevor Mudge. The YAGS Branch Prediction Scheme. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 69–77, June 1998.

[18] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, June 2005.

[19] Matthew Farrens and Arvin Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *Proc. of the 18th Annual International Symposium on Computer Architecture*, pages 128–137, May 1991.

[20] Wi fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proc. of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 301–312, January 2001.

[21] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 228–241, December 2003.

[22] International Technology Roadmap for Semiconductors. ITRS 2004 Update. Semiconductor Industry Association, 2004. http://www.itrs.net/Common/2004Update/2004Update.htm.

[23] Erik G. Hallnor and Steven K. Reinhardt. A Compressed Memory Hierarchy using an Indirect Index Cache. Technical Report CSE-TR-488-04, University of Michigan, 2004.

[24] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, February 2001.

[25] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[26] Jaehyuk Huh, Stephen W. Keckler, and Doug Burger. Exploring the Design Space of Future CMPs. In *Proc. of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[27] Doug Joseph and Dirk Grunwald. Prefetching Using Markov Predictors. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[28] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[29] Stephan Jourdan, Tse-Hao Hsing, Jared Stark, and Yale N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, October 1996.

[30] Krishna Kant and Ravi Iyer. Compressibility Characteristics of Address/Data transfers in Commercial Workloads. In *Proc. of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 59–67, February 2002.

[31] Ando Ki and Alan E. Knowles. Adaptive Data Prefetching Using Cache Information. In *Proc. of the 1997 International Conference on Supercomputing*, pages 204–212, July 1997.

[32] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and Evaluation of a Selective Compressed Memory System. In *Proc. of International Conference on Computer Design (ICCD'99)*, pages 184–191, October 1999.

[33] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. An On-chip Cache Compression Technique to Reduce Decompression Overhead and Design Complexity. *Journal of Systems Architecture:the EUROMICRO Journal*, 46(15):1365–1382, December 2000.

[34] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Adaptive Methods to Minimize Decompression Overhead for Compressed On-chip Cache. *International Journal of Computers and Application*, 25(2), January 2003.

[35] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[36] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, September 2005.

[37] Alex Pajuelo, Antonio Gonz·lez, and Mateo Valero. Speculative Dynamic Vectorization. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[38] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[39] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner. Power5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4), 2005.

[40] Viji Srinivasan, Edward S. Davidson, and Gary S. Tyson. A Prefetch Taxonomy. *IEEE Transactions on Computers*, 53(2):126–140, February 2004.

[41] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[42] R.B. Tremaine, P.A. Franaszek, J.T. Robinson, C.O. Schulz, T.B. Smith, M.E. Wazlowski, and P.M. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, March 2001.

[43] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[44] Bryan Usevitch. JPEG2000 compliant lossless coding of floating point data. In *Proc. of the 2005 Data Compression Conference*, page 484, March 2005.

[45] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent Value Compression in Data Caches. In *Proc. of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–265, Dec. 2000.

[46] Youtao Zhang and Rajiv Gupta. Enabling Partial Cache Line Prefetching Through Data Compression. In *Proc. of the 2003 International Conference on Parallel Processing*, pages 277–285, Oct. 2003.

[47] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent Value Locality and Value-centric Data Cache Design. In *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, November 2000.

[48] Zheng Zhang and Josep Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 188–199, June 1995.