Appears in the proceedings of the
31st Annual International Symposium on Computer Architecture (ISCA-31)
Munich, Germany, June 19-23, 2004

Adaptive Cache Compression for High-Performance Processors

Alaa R. Alameldeen and David A. Wood
Computer Sciences Department, University of Wisconsin-Madison
{alaa, david}@cs.wisc.edu

Abstract

Modern processors use two or more levels of cache memories to bridge the rising disparity between processor and memory speeds. Compression can improve cache performance by increasing effective cache capacity and eliminating misses. However, decompressing cache lines also increases cache access latency, potentially degrading performance.

In this paper, we develop an adaptive policy that dynamically adapts to the costs and benefits of cache compression. We propose a two-level cache hierarchy where the L1 cache holds uncompressed data and the L2 cache dynamically selects between compressed and uncompressed storage. The L2 cache is 8-way set-associative with LRU replacement, where each set can store up to eight compressed lines but has space for only four uncompressed lines. On each L2 reference, the LRU stack depth and compressed size determine whether compression (could have) eliminated a miss or incurs an unnecessary decompression overhead. Based on this outcome, the adaptive policy updates a single global saturating counter, which predicts whether to allocate lines in compressed or uncompressed form.

We evaluate adaptive cache compression using full-system simulation and a range of benchmarks. We show that compression can improve performance for memory-intensive commercial workloads by up to 17%. However, always using compression hurts performance for low-miss-rate benchmarks—due to unnecessary decompression overhead—degrading performance by up to 18%. By dynamically monitoring workload behavior, the adaptive policy achieves comparable benefits from compression, while never degrading performance by more than 0.4%.

1 Introduction

Semiconductor technology trends and microarchitectural innovations continue to exacerbate the performance gap between processors and memory. The ITRS Roadmap [16]—the semiconductor industry's detailed projection of Moore's Law [34]—predicts that transistor performance will improve at nearly 21% per year until 2007 while DRAM latency will improve at only 10% per year. Coupled with the trend toward increasingly deep pipelines [22, 23], main memory latency is expected to grow to hundreds of cycles.

Cache memories have long been used to reduce average memory latency and bandwidth. Current processors typically provide two levels of on-chip caches (e.g., separate L1 instruction and data caches and a unified L2 cache), with some recent architectures having three-level cache hierarchies [39]. Effectively organizing the limited on-chip cache resources is particularly critical for many memory-intensive commercial workloads [6].

Cache compression is one way to improve the effectiveness of cache memories [9, 27, 29, 43, 44]. Storing compressed lines in the cache increases the effective cache capacity. For example, Yang, et al. propose a compressed L1 cache design where each set can store either one uncompressed line or two compressed lines [43]. Increasing the effective cache size can eliminate misses and thereby reduce the time lost to long off-chip miss penalties. However, compression increases the cache hit time, since the decompression overhead lies on the critical access path. Depending upon the balance between hits and misses, cache compression has the potential to either greatly help or greatly hurt performance.

In this paper, we develop an adaptive cache compression scheme to dynamically optimize on-chip cache performance (Section 2). Our design has two major parts. First, we use a two-level cache hierarchy where the L1 cache holds uncompressed data and the L2 cache dynamically selects between compressed and uncompressed storage. We use a simple, significance-based compression algorithm, Frequent Pattern Compression [4] to compress L2 lines. The L2 cache is 8-way set-associative with LRU replacement, where each set can store up to eight compressed lines but has space for only four uncompressed lines. Thus compression can potentially double the effective capacity of the cache. Simulation results show that memory-intensive commercial workloads achieve average effective capacities of 5-7 MB for a 4 MB uncompressed L2 cache.

Second, our adaptive compression policy (Section 3) uses the L2 cache's LRU replacement state to track whether compression would help, hurt, or make no difference to a given reference. The key insight is that the LRU stack depth and compressed size determines whether a given reference hits because of compression, would have missed without compression, or would have hit or missed regardless. The controller updates a single, global saturating counter on each reference, increment-

ing by the L2 miss penalty when compression could have or did eliminate a miss and decrementing it by the decompression latency when a reference would have hit regardless. The controller uses the predictor when the L2 allocates a line: storing the line uncompressed if the counter is negative, and compressed otherwise.

This paper makes four main contributions:

- It shows that always compressing L2 cache lines increases the effective cache capacity for commercial benchmarks by 29-75%, which in turn reduces L2 miss ratios by 9-24% and overall run-time as much as 15% (i.e., a 17% speedup). However, the increased L2 access latency (due to decompression overhead), degrades performance for workloads with low L2 miss rates by as much as 18%.
- It proposes a novel adaptive policy that dynamically balances the benefit of compression (i.e., miss ratio reduction) with the cost (i.e., increased L2 access latency).
- It presents full-system simulation results showing that adaptive cache compression can improve performance of memory-intensive commercial workloads by up to 17%, while never degrading performance by more than 0.4%.
- To our knowledge, it presents the first quantitative evaluation of L2 cache compression for commercial workloads.

2 Compressed Cache Hierarchy

We propose a two-level cache hierarchy consisting of uncompressed L1 instruction and data caches, and an optionally compressed L2 unified cache. While many of the mechanisms and policies we develop could be adapted to other cache configurations (e.g., three-level hierarchies), we do not consider them in this study.

The goals of this design include:

- Using compression to increase effective L2 cache capacity in order to reduce L2 misses.
- Limit the impact of cache decompression overhead by providing a bypass path for uncompressed lines.
- Enable an adaptive policy to dynamically control compression based on workload demands.
- Limit impact on the cache design complexity.

2.1 Overview

Figure 1 illustrates the proposed cache hierarchy. L1 instruction and data caches store uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. This design also completely isolates the processor core from the compression hardware. The L1 data cache uses a writeback, write allocate policy to simplify the L2 compression logic. On L1 misses, the

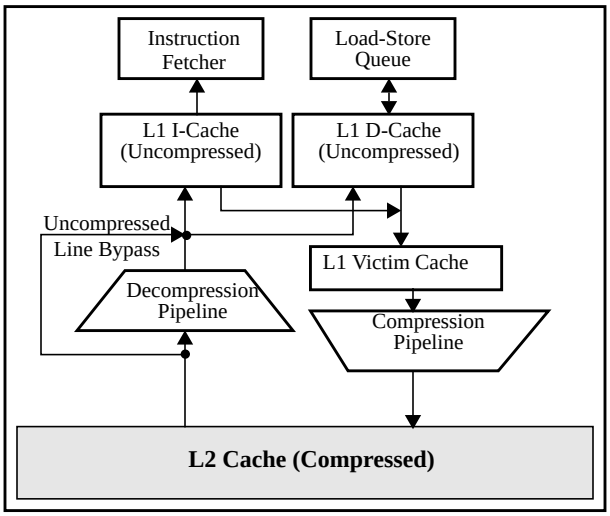

Figure 1. Compressed Cache Hierarchy.

controller checks an uncompressed victim cache in parallel with the L2 access. On an L2 hit, the L2 line is decompressed if stored in compressed form. Otherwise, it bypasses the decompression pipeline. On an L2 miss, the requested line is fetched from main memory. We assume uncompressed memory, however, this is largely an orthogonal decision. The L1 and L2 caches maintain exclusion and lines are allocated in the L2 only when replaced from the L1. In addition to its normal function, the victim cache acts as a rate-matching buffer between the L1s and the compression pipeline [29]. For design simplicity, we assume a single line size for all caches.

2.2 Decoupled Variable-Segment Cache

To exploit compression, the L2 cache must be able to pack more compressed cache lines than uncompressed lines into the same space. One approach is to decouple the cache access, adding a level of indirection between the address tag and the data storage. Seznec's decoupled sector cache does this on a per-set basis to improve the utilization of sector (or sub-block) caches [36]. Hallnor and Reinhardt's Indirect-Index Cache decouples accesses across the whole cache, allowing fully-associative placement, a software managed replacement policy, and (recently) compressed lines [20, 21]. Lee, et al.'s selective compressed caches use this technique to allow two compressed cache lines to occupy the space required for one uncompressed line [29, 27, 28]. Decoupled access is simpler if we serially access the cache tags before the data. Fortunately, this is becoming increasingly necessary to limit power dissipation [25].

Our decoupled variable-segment cache builds on these earlier concepts. As illustrated in Figure 2, each set is 8-way set-associative, with a compression information tag stored with each address tag. The data array is broken into eight-byte segments, with 32 segments statically allocated to each cache set. Thus, each set can hold no

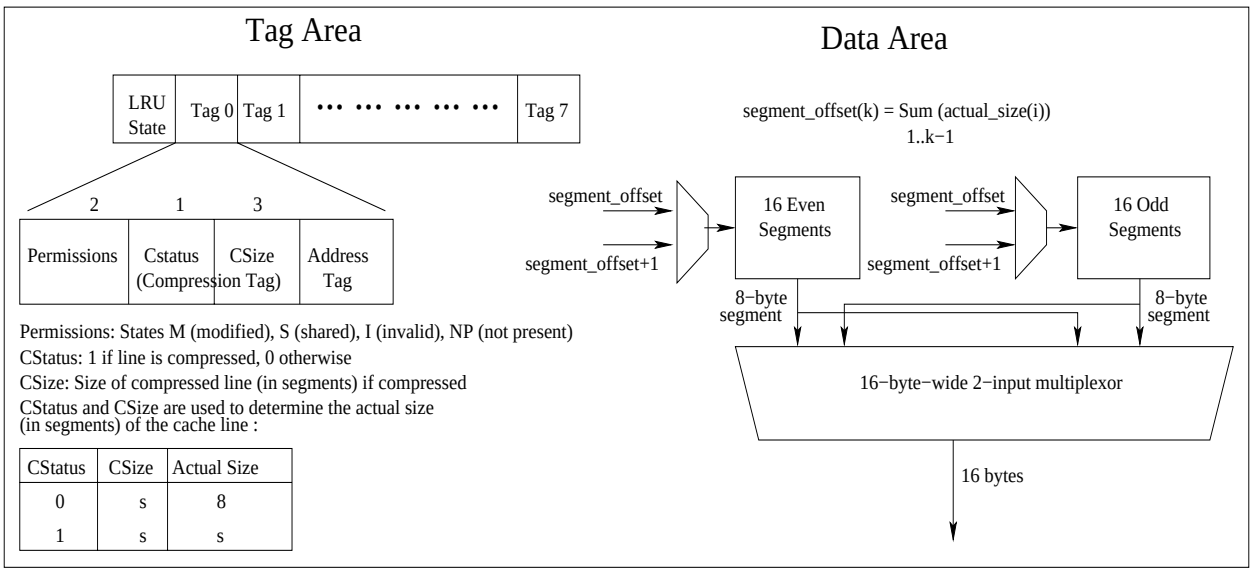

Figure 2. A single set of the decoupled variable-segment cache.

more than four uncompressed 64-byte lines, and compression can at most double the effective capacity. Each line is compressed into between one and eight segments, with eight segments being the uncompressed form. The compression tag indicates i) the compressed size of the line (CSize) and ii) whether or not the line is stored in compressed form (CStatus). A separate cache state indicates the line's coherence state, which can be M (modified), S (shared), I (invalid), or NP (not present). The NP state differentiates between a line invalidated by a coherence event and one invalidated due to exclusion or L2 replacement. Note that the compression tag is maintained even for NP lines for use by the adaptive compression policy.

Data segments are stored contiguously in address tag order. That is, the offset for the first data segment of line k is:

$$\text{segment\_offset}(k) = \sum_{i=1}^{k-1} \text{actual\_size}(i)$$

A line's actual size is determined by the compression tag (Figure 2) and the eight segment offsets are computed in parallel with the address tag match using a 5-bit parallel-prefix adder. On an address tag match, the segment offset and actual length are used to access the corresponding segments in the data array. The array is split into banks for even and odd segments, allowing two segments (16 bytes) to be fetched per cycle regardless of the alignment [19].

Because the L1 and L2 caches maintain exclusion, an L1 replacement writes back both clean and dirty lines to the L2 cache. In many cases, the writeback finds a matching address tag, with space allocated, in state NP. If the compressed size is the same as before, this writeback is trivial. However, if the address tag is not found, or the compressed size has changed, the cache controller must allocate space in the set. This may entail replacing one or more L2 lines or compacting invalid/not present lines to make space. More than one line may have to be replaced if the newly allocated line is larger than the LRU line plus the unused segments. In this case, we replace at most two lines by replacing the LRU line and searching the LRU list to find the least-recently-used line that ensures we have enough space.

Compacting a set requires moving tags and data segments to maintain the contiguous storage invariant. This operation can be quite expensive, because it may require reading and writing all the set's data segments. For this reason, compaction is deferred as long as possible and is never needed on a read (e.g., L1 fill) access. With a large L1 victim cache and sufficient L2 cache banks, compaction will have negligible impact on performance.

A decoupled variable-segment cache adds relatively little storage overhead. For example, consider a 4-way, 4 MB uncompressed cache with 64-byte lines. Each set has 2048 data bits, in addition to four tags. Each tag includes a 24-bit address tag, a 2-bit LRU state, and a 2-bit permission, for a total of 4*(24+2+2)=112 bits per set. Our scheme adds four extra tags, increases the LRU state to three bits and adds a 4-bit compression tag per line. This adds 112+8*1+8*4=152 bits per set, which increases the total cache storage by approximately 7%.

2.3 Frequent Pattern Compression (FPC)

L2 cache compression requires a low-latency hardware compression algorithm. We implemented a significance-based scheme called Frequent Pattern Compression (FPC) [4]. Compared to the dominant dictionary-based

approaches [17, 26], FPC has a lower decompression latency and comparable compression ratios. FPC decompresses a 64-byte line in five cycles, assuming 12 FO4 gate delays per cycle. The 64-byte L2 cache lines are compressed into one to eight 8-byte segments. Simulation results show compression ratios (i.e., original size divided by compressed size) of 1.3-2.4 for selected SPECint benchmarks and commercial workloads, but only 1.0–1.3 for selected SPECfp benchmarks.

3 Adaptive Cache Compression

While compression helps eliminate long-latency L2 misses, it increases the latency of the (usually more frequent) L2 hits. Thus, some benchmarks (or benchmark phases) will benefit from compression, but others will suffer. For a simple, in-order blocking processor, L2 cache compression will help if:

$$(\text{avoided L2 misses} \times \text{L2 miss penalty}) > (\text{penalized L2 hits} \times \text{decompression penalty})$$

Where penalized L2 hits are those that unnecessarily incur the decompression penalty. Rearranging terms yields:

$$(\text{avoided L2 misses} / \text{penalized L2 hits}) > (\text{decompression penalty} / \text{L2 miss penalty})$$

For a 5 cycle decompression penalty and 400 cycle L2 miss penalty, compression wins if it eliminates at least one L2 miss for every 400/5=80 penalized L2 hits. While this may be easily achieved for memory-intensive commercial workloads, smaller workloads—that fit in a large L2 cache—may suffer degraded performance.

Ideally, a compression scheme should compress data when the benefit (i.e., avoided misses) outweighs the cost (i.e., penalized L2 hits). This section describes the central innovation in this paper: an adaptive predictor that monitors the actual effectiveness of compression and uses this feedback to dynamically determine whether to store a line in a compressed or uncompressed form. Simulation results (Section 5) show that this adaptive policy obtains most of the benefit of compression when it helps, while never performing much worse than not compressing.

3.1 Classification of Cache References

The key insight underlying our adaptive compression policy is that the LRU stack depth and compressed size determine whether compression helps or hurts a given reference. The example in Figure 3 illustrates the different cases using the LRU stack of a single cache set.

Classification of hits:

- A reference to Address A hits at stack depth 1. Because the set can hold four uncompressed lines and the LRU stack depth is less than or equal to four, compression provides no benefit. Conversely, since the data is stored uncompressed, the reference incurs no decompression penalty. We call this case an *unpenalized hit*.
- A reference to Address C hits at stack depth 3. Compression does not help, since the line would be present even if all lines were uncompressed. Unfortunately, since the block is stored in compressed form, the reference incurs an unnecessary decompression penalty. We call this case a *penalized hit*.
- A reference to Address E hits at stack depth 5. In this case, compression has eliminated a miss that would otherwise have occurred. We call this case an *avoided miss*.

Classification of misses:

- A reference to Address G misses in the cache, but matches the address tag at LRU stack depth 7. The sum of the compressed sizes at stack depths 1 through 7 totals 29. Because this is less than 32 (the number of data segments per set), this reference misses *only* because one or more lines at stack depths less than 7 are stored uncompressed (i.e., Address A could have been stored in two segments). We call this case an *avoidable miss*.
- A reference to Address H misses in the cache, but matches the address tag at LRU stack depth 8. However, this miss cannot be avoided because the sum of compressed sizes exceeds the total number of segments (i.e., 35 > 32). Similarly, a reference to Address I does not match any tag in the stack. We call each of these cases an *unavoidable miss*.

The cache controller uses the LRU state and compression tags to determine the class of each L2 reference. The avoidable miss calculation is implemented using a five-bit parallel-prefix adder with 8:1 multiplexors on the inputs to select compressed sizes in LRU order. Note

Stack Depth	Address Tag	CStatus	CSize (Segments)	Perm.
1	A	Uncompr.	2	M
2	B	Uncompr.	8	M
3	C	Compr.	4	M
4	D	Compr.	3	M
5	E	Compr.	2	M
6	F	Compr.	7	M
7	G	Uncompr.	5	NP
8	H	Uncompr.	6	NP

Figure 3. A cache set example.

Address tags are shown in LRU order (Address A is the most recent). The first six tags corresponds to lines in the cache, while the last two correspond to evicted lines (Permissions = NP). Addresses C, D, E and F are stored in compressed form.

that this parallel-prefix add uses the compressed sizes, whereas the parallel-prefix add discussed in Section 2.2 uses the actual sizes. To save hardware, a single parallel-prefix adder can be time-multiplexed, since gathering compression information is not time critical, and the data array access takes longer than the tag access.

3.2 Global Compression Predictor

Like many predictors, the adaptive compression policy uses past behavior to predict the future. Specifically, the controller uses the classification above to update a global saturating counter—called the Global Compression Predictor (GCP)—to estimate the recent cost or benefit of compression. On a penalized hit, the controller biases against compression by subtracting the decompression penalty. On an avoided or avoidable miss, the controller increments the counter by the (unloaded) L2 miss penalty. To reduce the counter size, we normalize these values to the decompression latency, subtracting one and adding the miss penalty divided by decompression latency (e.g., 400 cycles / 5 cycles = 80).

The controller uses the GCP when allocating a line in the L2 cache. Positive values mean compression has been helping eliminate misses, so we store the line in compressed form. Negative values mean compression has been penalizing hits, so we store the line uncompressed. All allocated lines—even those stored uncompressed—must run through the compression pipeline to calculate their compressed size, which is used to determine avoidable misses.

The size of the saturating counter determines how quickly the predictor adapts to workload phase changes. The results in this paper use a single global 19-bit counter that saturates at 262,143 or -262,144 (approximately 3300 avoided or avoidable misses). Using a large counter means the predictor adapts slowly to phase changes, preventing short bursts from degrading long-run behavior. Section 6.4 examines the impact of workload phase behavior on compression.

While we assume LRU replacement in this paper, any stack algorithm—including random [32]—will suffice. Moreover, the stack property only needs to hold for lines that either do or might have fit due to compression (e.g., LRU stack depths 5–8 in our design). We can use any arbitrary replacement policy for the top four elements in the "stack."

4 Evaluation Methodology

We present an evaluation of adaptive compression on a dynamically-scheduled out-of-order processor using full-system simulation of commercial workloads and a subset of the SPECcpu2000 benchmarks.

Table 1. Simulation Parameters

L1 Cache Configuration	Split I & D, each 64 KB (unless otherwise specified) 2-way set associative with LRU replacement, 64-byte line, 2-cycle access time
L2 Cache Configuration	Unified 4 MB (unless otherwise specified), 8-way set associative with LRU replacement, 64-byte line
L2 Cache Hit Latency	Uncompressed: 20 cycles, Compressed: 25 cycles (20 + 5 decompression cycles)
Memory Configuration	4 GB of DRAM, 400 cycles access time (unless otherwise specified) with infinite chip-to-memory bandwidth
Processor Pipeline	4-wide superscalar, 11-stage pipeline—Pipeline stages: fetch (3), decode (3), schedule (1), execute (1 or more), retire (3)
Reorder Buffer	64-entry ROB
Branch Predictors	1 KB YAGS direct branch predictor [14], a 64-entry cascaded indirect branch predictor [13], and a 64-entry return address stack predictor [24]

4.1 System Configuration

We evaluated the performance of our compressed cache designs on a dynamically-scheduled SPARC V9 uniprocessor using the Simics full-system simulator [30], extended with a detailed processor simulator (TFSim [33]), and a detailed memory system timing simulator [31]. Our target system is a superscalar processor with out-of-order execution. Table 1 presents our basic simulation parameters.

4.2 Workloads

To evaluate our design against alternative schemes, we used several multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [2]. We also used eight of the SPECcpu2000 [38] benchmarks: four from the integer suite and four from the floating point suite. All of these workloads run under the Solaris 9 operating system. These workloads are briefly described in Table 2. We selected these workloads to cover a wide range of compressibility properties, miss rates and working set sizes. For each data point in our results, we present the average and the 95% confidence interval of multiple simulations to account for space or time variability [3]. Our runtime results for commercial workloads represent the average number of cycles per transaction (or request), whereas runtime results for SPEC benchmarks represent the average number of cycles per instruction (CPI).

Table 2. Workload Descriptions

Online Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier, with many concurrent users performing transactions. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 5 GB database with 25,000 warehouses stored on eight raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses. There are 16 simulated users, and the database is warmed up for 100,000 transactions before taking measurements for 300 transactions.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use Sun's HotSpot 1.4.0 Server JVM. Our experiments use two threads and two warehouses, a data size of ~44 MB, a warmup interval of 200,000 transactions, and a measurement interval of 20,000 transactions.
Static Web Serving: Apache. We use Apache 2.0.43 for SPARC/Solaris 9, configured to use pthread locks and minimal logging as the web server. We use SURGE [5] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB), and disable Apache logging for high performance. We simulate 400 clients, each with 25 ms think time between requests, and warm up for 50,000 requests before taking measurements for 3000 requests.
Static Web Serving: Zeus. Zeus is another static web serving workload driven by SURGE. Zeus uses an event-driving server model. Each processor of the system is bound by a Zeus process, which is waiting for web serving event (e.g., open socket, read file, send file, close socket, etc.). The rest of the configuration is the same as Apache (20,000 files of ~500 MB total size, 400 clients, 25 ms think time, 50,000 requests for warmup, 3000 requests for measurements).
SPEC. We use four integer benchmarks (bzip, gcc, mcf and twolf) and four floating point benchmarks (ammp, applu, equake, and swim) from the SPECcpu2000 set to cover a wide range of compressibility properties and working set sizes. We use the first reference input for each benchmark. We fast forward each benchmark for 1 billion instructions, and simulate the next 1 billion instructions.

5 Evaluation of Adaptive Compression

To understand the utility of adaptive compression, we compare it with two extreme policies: *Never* and *Always*. *Never* models a standard 8-way set associative L2 cache design, where data is never stored compressed. *Always* models a decoupled variable-segment cache (Section 2.2), but always stores compressible data in compressed form. Thus *Never* strives to reduce hit latency, while *Always* strives to reduce miss rate. *Adaptive* uses the policy described in Section 3 to utilize compression only when it predicts that the benefits outweigh the overheads.

5.1 Effective Cache Capacity

This section examines the compression ratio achieved in the decoupled variable-segment L2 cache. Figure 4 presents the average effective cache capacity (and 95% confidence intervals). Effective cache capacity is computed by counting the valid cache lines in samples taken every 100 million cycles. These benchmark runs use the baseline configuration of a 128 KB split L1 cache and a 4 MB unified L2 cache.

Without compression, the maximum cache size is 4 MB. Under the *Never* policy, most workloads approach this maximum, although maintaining exclusion between L1 and L2 caches prevents them from quite reaching it. The twolf benchmark illustrates that some workloads do not fully utilize large L2 caches, and hence are unlikely to benefit much from cache compression. The *Always* and *Adaptive* results show that many workloads can potentially benefit significantly from cache compression. The memory-intensive commercial workloads achieve effective cache capacities of 5–7 MB, a 25–75% increase.

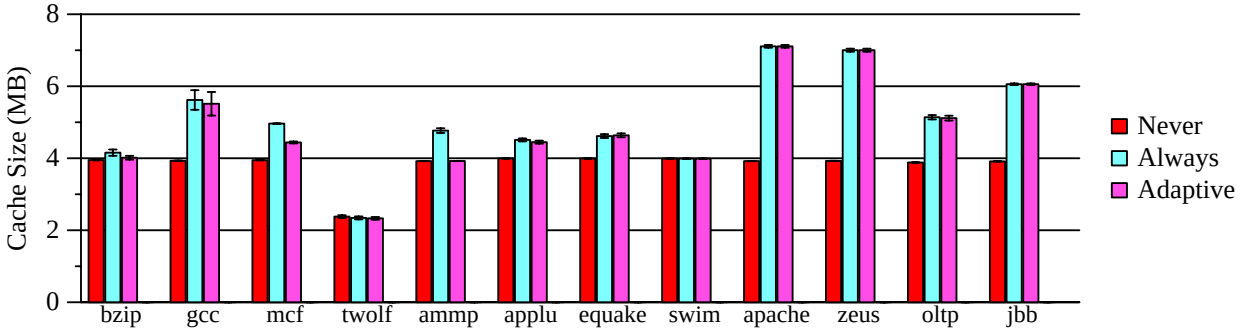

Figure 4. Average cache capacity during benchmark runs (4 MB uncompressed size).

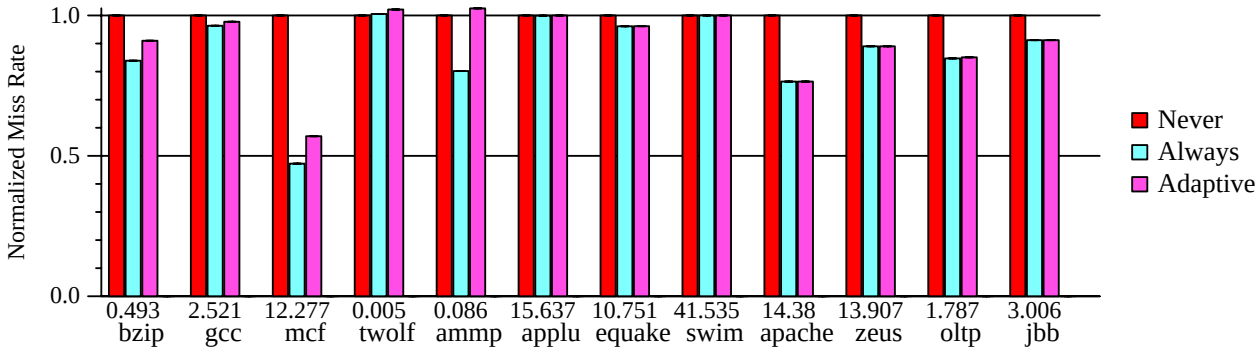

Figure 5. L2 cache miss rate for the three compression alternatives (128K/4MB configuration), normalized to the "Never" miss rate, shown at the bottom.

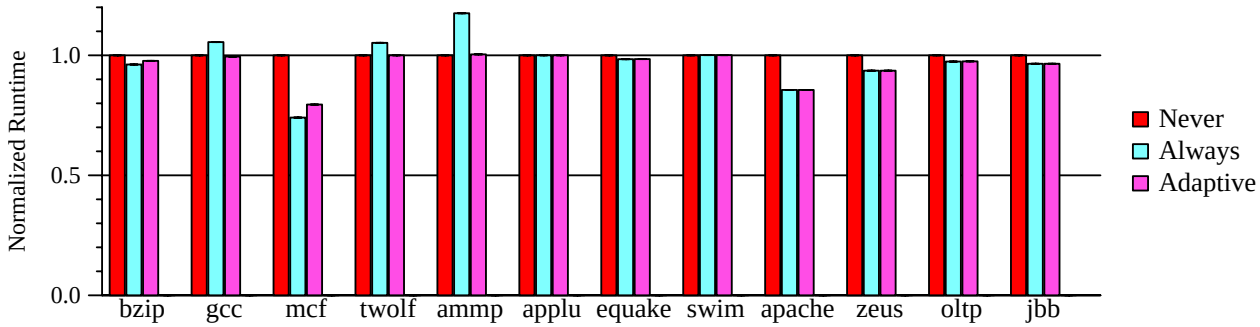

Figure 6. Runtime for the three compression alternatives, normalized to the "Never" runtime.

5.2 Miss Rates for Compressed Caches

Using compression to increase effective cache capacity should tend to decrease the L2 miss rate. Figure 5 presents the average miss rates for our benchmarks. The results are normalized to *Never* to focus on the benefit of compression, but the absolute misses per 1000 instructions for *Never* are included at the bottom. Both *Always* and *Adaptive* have lower or equal miss rates when compared to *Never*, with two exceptions. The slight increase in the miss rate for *Adaptive* when compared to *Never* in twolf and ammp is due to the difference in associativity. Since we predict no compression in both of these benchmarks, the L2 caches are 4MB, 8-way for *Never* and 4 MB, 4-way for *Adaptive.*

Not surprisingly, the commercial benchmarks achieve substantial benefits from compression, reducing the miss rates by 9–24%. More striking are the results for mcf. *Always* reduces the miss rate by over half, despite increasing the effective capacity by only 25%. This suggests that compression increases effective cache size sufficiently for a critical working set to fit in the L2 cache. Benchmarks with small working sets (e.g., twolf) get little or no miss rate reduction from compression. The four floating-point benchmarks, despite very large working sets, do not benefit from compression (except for ~4% for equake) due to the poor compression ratios our compression algorithm achieves for floating-point data.

5.3 Performance

The ultimate objective of adaptive cache compression is to achieve performance comparable to the best of *Always* or *Never.* Reducing the cache miss rate, as *Always* does for some benchmarks, may be outweighed by the increase in hit latency. Figure 6 presents the simulated runtime of our twelve benchmarks, normalized to the *Never* case. Most of the benchmarks that have substantial miss rate reductions under *Always* also improve runtime performance (e.g., a speedup of 17% for apache, 7% for zeus, and 35% for mcf). However, the magnitude of this improvement depends upon the absolute frequency of misses. For example, jbb and zeus have similar relative miss rate improvements, but since zeus has more than four times as many misses per instruction, its performance improvement is greater. On the other hand, benchmarks with smaller working sets (e.g., gcc, twolf, ammp) do not benefit from greater cache capacity. Ammp is the extreme example in our benchmark set, with *Always*'s performance degrading by roughly 18% compared to *Never.*

Figure 6 also shows that *Adaptive* achieves most of the benefit of *Always* for benchmarks that benefit from compression. In addition, for benchmarks that do not benefit from compression, it degrades performance by less than 0.4% compared to *Never.* Mcf is the one benchmark where *Adaptive* performs substantially less well than

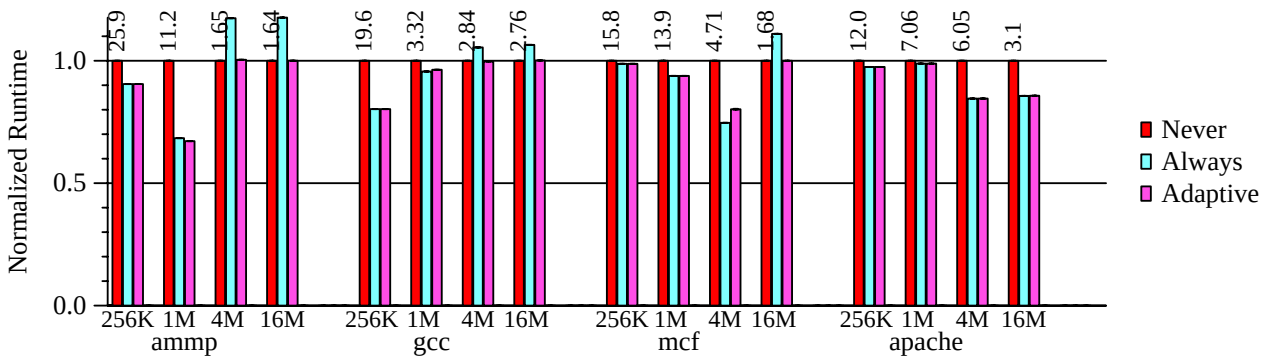

Figure 7. Sensitivity of four benchmarks to L2 cache size changes (all with 32 KB L1). Runtime (in cycles per instruction) is normalized to *Never*.

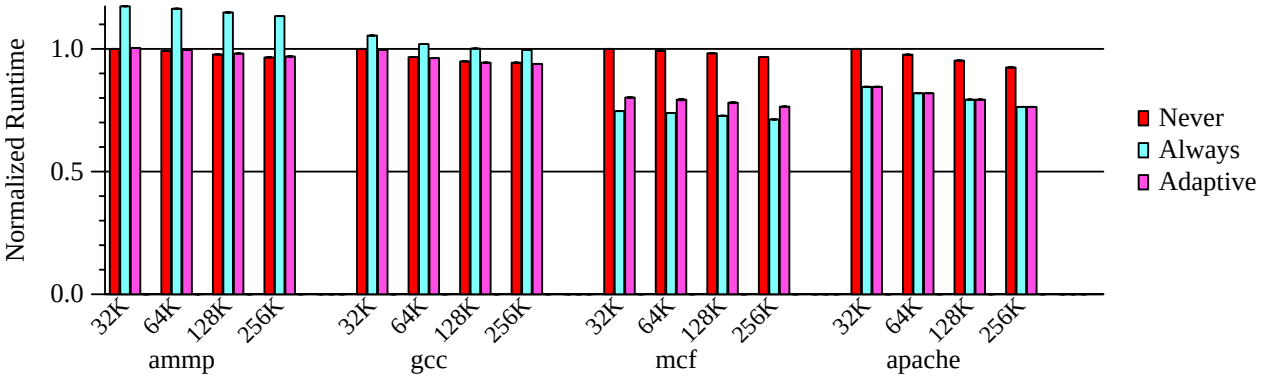

Figure 8. Sensitivity of four benchmarks to L1 cache size changes (all with 4 MB L2). Runtime is normalized to 32K/*Never*.

Always (26% vs. 35%) due to the benchmark's phase behavior, as explained in Section 6.4.

5.4 Summary

The results in this section show that while some memory-intensive benchmarks benefit significantly from compression, other benchmarks receive little benefit or even degrade significantly. For our benchmarks, *Adaptive* achieves the best of both worlds, improving performance by using compression when it helps, while not hurting performance when compression does not help.

6 Sensitivity of Adaptive Compression

The effectiveness of cache compression depends upon the interaction between a workload's working-set size and the caches' sizes and latencies. Adaptive cache compression is designed to dynamically adjust its compression decisions to approach the performance of the better of the two static policies *Always* and *Never*. In this section, we investigate how well *Adaptive* adjusts to changes in L1 and L2 cache sizes, decompression latency, and benchmark phases. We focus on four benchmarks that represent opposite sides of the spectrum: mcf and apache that are helped by compression; and ammp and gcc that are hurt by compression.

6.1 Sensitivity to L2 Cache Size

Cache compression works best when it can increase the effective L2 size enough to hold a workload's critical working set. Conversely, compression provides little or no benefit when the working set is either much larger, or much smaller, than the L2 cache size. Figure 7 illustrates this by presenting normalized runtime for various L2 cache sizes, assuming a fixed L2 access latency. For ammp and gcc, compression helps performance for smaller cache sizes, since compression allows the L2 cache to hold more data (e.g., compression allows ammp to hold an average of ~1.2 MB in a 1 MB L2, resulting in a 49% speedup). However, compression hurts performance for larger cache sizes, since compression increases the hit latency but doesn't significantly increase the effective cache size. At the other extreme, mcf and apache only benefit from compression for larger caches (4 and 16 MB), since the working set is too large to fit in the smaller cache sizes, even with compression. For all cases, *Adaptive* adapts its behavior to match the better of *Always* and *Never*.

6.2 Sensitivity to L1 Cache Size

The effectiveness of L2 cache compression depends on the overhead incurred decompressing lines on L2 hits. Since the L1 filters requests to the L2, the L1 size

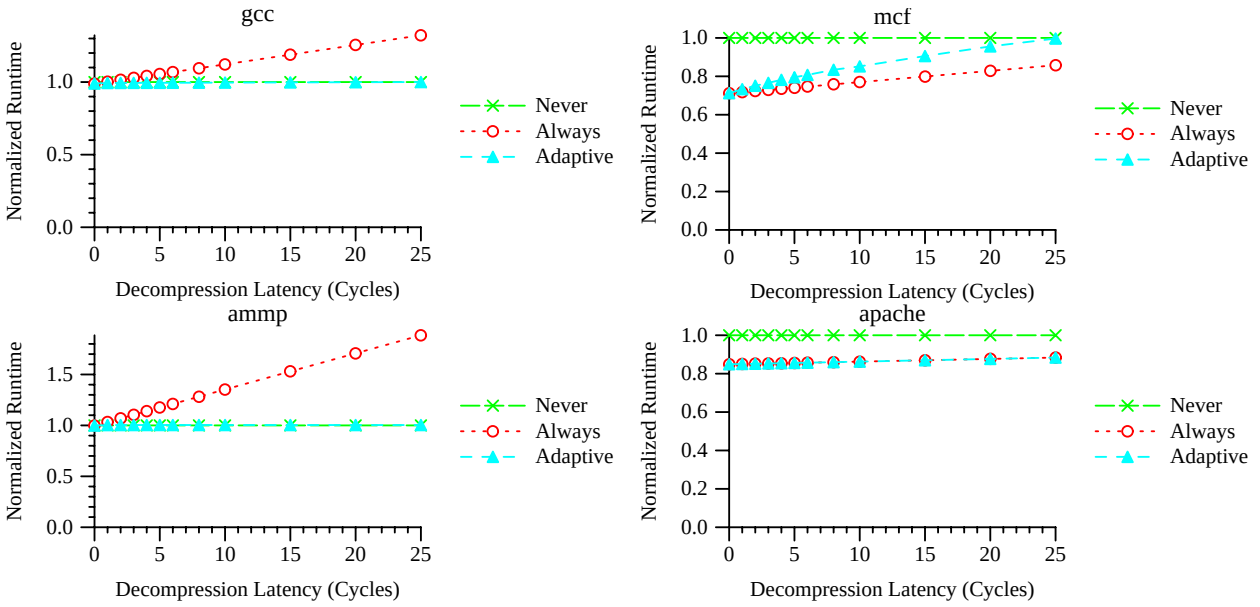

Figure 9. Sensitivity of adaptive compression to decompression latency. *Adaptive* performs the same as the better of *Never* and *Always*, except for mcf.

impacts this overhead. As the L1 cache size increases, references that would have hit in the L2 are now satisfied in the L1. Thus the decompression overhead tends to decrease. Conversely, as the L1 size decreases, the L2 incurs more penalized hits due to the increased number of L1 misses. Figure 8 illustrates this tradeoff for a 4 MB L2 cache, assuming a fixed L1 access latency. For these benchmarks and parameters, increasing L1 size has very little impact on the relative benefit of compression.

6.3 Adapting to Decompression Latency

All cache compression schemes are highly sensitive to the decompression latency. Larger decompression latencies decrease the appeal of cache compression by increasing the effective access latency. Figure 9 presents normalized runtime for four benchmarks as the decompression latency varies from 0 to 25 cycles. These results show that *Adaptive* adjusts to changes in decompression latency, and usually achieves performance comparable to the better of *Always* and *Never*. Mcf is the notable exception, where *Adaptive* degrades to *Never* for large decompression latencies. This behavior is due to rapid phase changes in the benchmark and their impact on the global predictor. We examine this further in the next section.

6.4 Adapting to Benchmark Phases

Many benchmarks exhibit phase behavior [37], and a benchmark's working set size may change between different phases. These changes can disrupt the *Adaptive* policy, since the past (the previous phase) may not be a good predictor of the future (the next phase). However, even an ideal predictor—one which instantaneously detects phase changes and accurately predicts future behavior—cannot adapt immediately. This is because the cache state (i.e., which lines are currently compressed) depends upon the predictions made during the previous phase. Thus the adaptive policy may incur many avoidable misses or penalized hits before it adapts to the new phase.

Figure 10 illustrates the phase behavior exhibited during our simulation runs. The top graphs show changes in the global predictor (GCP) value over time, while the lower graphs show the effective cache size over time. Two benchmarks, ammp and apache, exhibit no phase changes during these relatively short simulation runs. Thus, the global predictor and cache size remain roughly constant. For ammp, GCP is negative and the effective cache sizes holds steady at 4 MB. For apache, GCP stays positive and the effective cache size fluctuates around 7 MB.

The other two benchmarks exhibit distinct working set phases. Gcc's working set changes slowly from a size that fits in less than 4 MB for the first 1.5 billion cycles to a size that benefits from compression for the remainder of the run. Adaptive compression adjusts to this change and compresses lines to increase the effective cache size. For mcf, however, the phase changes are much more frequent. In this case, the adaptive policy alternates between predicting for and against compression. Thus, *Adaptive* only compresses some cache lines, resulting in worse performance than the *Always* policy (as shown in Section 5.3). Using a larger counter as our predictor reduces this effect by increasing the hysteresis, thus increasing the fraction of compressed lines. How-

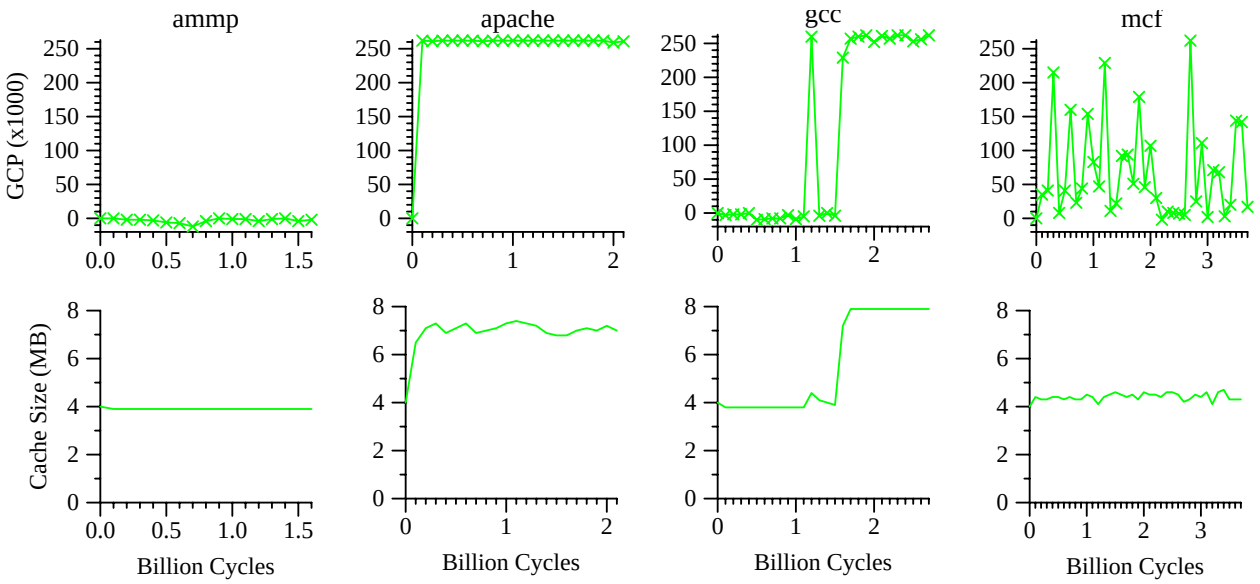

Figure 10. Sensitivity of adaptive compression to benchmark phase changes. This figure shows temporal changes in global compression predictor values (top) and effective cache size (bottom).

ever, a larger predictor also delays the response to more well-behaved phase changes, such as the change exhibited by gcc.

7 Related Work

Hardware-based compression has been used to increase effective memory size, reduce memory address and data bandwidth, and increase effective cache size. Although adaptive compression has been previously applied in software to virtual memory systems, this paper presents the first adaptive scheme for hardware caches.

Hardware Memory Compression Designs. IBM's MXT technology [40] employs real-time main-memory compression that can effectively double the main memory capacity. MXT compresses main memory data using a parallel algorithm, Parallel Block-Referential Compression with Directory Sharing, which divides each input data block into sub-blocks, and constructs a dictionary while compressing all sub-blocks in parallel [17].

Kjelso, et al. [26] use the X-Match hardware compression algorithm that maintains a dictionary and replaces each input data element with a shorter code in case of a total or partial match with a dictionary entry. Such compression reduces communication bandwidth by compacting cache-to-memory address streams [15] or data streams [10]. Benini, et al. [8] propose a data compression/decompression scheme to reduce memory traffic in general-purpose processor systems. They store uncompressed data in the cache, and compress/decompress on the fly when data is transferred to/from memory. They use a differential compression scheme based on the assumption that it is likely for data words in the same cache line to have some bits in common [7].

Cache Compression and Related Designs. Lee, et al. [27, 28, 29] propose a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks that can be reduced to half their original size. Their Selective Compressed Memory System (SCMS) uses a hardware implementation of the X-RL compression algorithm [26], a variant of the X-Match algorithm that gives a special treatment for runs of zeros. Ahn, et al. [1] propose several improvements on the X-RL technique that capture common values. Chen, et al. [9] propose a scheme that dynamically partitions the cache into sections of different compressibility. Hallnor and Reinhardt [21] modify their indirect-index cache design to allocate variable amounts of storage to different cache lines based on their compressibility. Pomerene, et al. [35] use a shadow directory scheme with more address tags than data lines to improve upon LRU replacement.

Yang and Gupta show that a small number of distinct values occupy a large fraction of memory access values in the SPECint95 benchmarks [42]. This value locality motivates their "Compression Cache" design [43]. Zhang, et al., design a value-centric data cache design called the frequent value cache (FVC) [44], which is a small direct-mapped cache dedicated to holding frequent benchmark values. They show that augmenting a direct mapped cache with a small FVC can greatly reduce the cache miss rate.

Adaptive Compression in Virtual Memory Systems. Adaptive compression has been used in virtual memory management schemes to compress portions of main memory (called compression caches) to avoid I/O operations caused by page faults. Douglis observes that different programs need compressed caches of different

sizes [12]. He implements a simple adaptive scheme that dynamically split main memory pages between uncompressed and compressed portions. Both portions compete for the LRU page in memory, and allocating a new page is biased towards the compression cache. Cortes, et al. [11] classify reads to the compression cache according to whether they were caused by swapping or prefetching, and propose optimized mechanisms to swap pages in/out. Wilson, et al. [41] propose dynamically adjusting the compressed cache size using a cost/benefit analysis that compares various target sizes, and takes into account the compression cost vs. the benefit of avoiding I/Os. Their system uses LRU statistics of touched pages to compare the costs and benefits of target sizes, and adjusts the compression cache size on subsequent page accesses. Freedman [18] optimizes the compression cache size for handheld devices according to the energy costs of decompression vs. disk accesses.

8 Conclusions

In this paper, we propose an adaptive compression policy to improve the performance of high-performance processors running memory-intensive workloads. We use a two-level cache hierarchy where the L1 holds uncompressed data while the L2 can optionally store data in compressed form. Our adaptive policy dynamically adjusts to the costs and benefits of compression. A single global saturating counter predicts whether the L2 cache should store a line in compressed or uncompressed form. The L2 controller updates the counter based on whether compression could (or did) eliminate a (potential) miss or incurs an unnecessary decompression overhead.

We show that compressing *all* compressible cache lines can improve performance for some memory-intensive workloads, while hurting the performance of other applications that have low miss rates or low compressibility. Our adaptive scheme successfully predicts workload behavior, thus providing a performance speedup of up to 26% over an uncompressed cache design for benchmarks that benefit from compression, while limiting the performance degradation of other benchmarks to less than 0.4%.

Acknowledgements

We thank Brad Beckmann, Mark Hill, Kevin Moore, Min Xu, the Wisconsin computer architecture affiliates, Virtutech AB, the Wisconsin Condor group and our anonymous reviewers for their feedback and support. 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] Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. Effective Algorithms for Cache-level Compression. In *Proceedings of the 2001 Conference on Great Lakes Symposium on VLSI*, pages 89–92, 2001.

[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 *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, February 2003.

[4] Alaa R. Alameldeen and David A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. Technical Report 1500, Computer Sciences Department, University of Wisconsin–Madison, April 2004.

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

[6] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[7] Luca Benini, Davide Bruni, Alberto Macii, and Enrico Macii. Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors. In *Proceedings of the IEEE 2002 Design Automation and Test in Europe*, pages 449–453, 2002.

[8] Luca Benini, Davide Bruni, Bruno Ricco, Alberto Macii, and Enrico Macii. An Adaptive Data Compression Scheme for Memory Traffic Minimization in Processor-Based Systems. In *Proceedings of the IEEE International Conference on Circuits and Systems, ICCAS-02*, pages 866–869, May 2002.

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

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

[11] Toni Cortes, Yolanda Becerra, and Raul Cervera. Swap Compression: Resurrecting Old Ideas. *Software - Practice and Experience Journal*, 46(15):567–587, December 2000.

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

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

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

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

[16] International Technology Roadmap for Semiconductors. 2002 Update. Semiconductor Industry Association, 2002. http://public.itrs.net/Files/2002Update/2002Update.pdf.

[17] Peter Franaszek, John Robinson, and Joy Thomas. Parallel Compression with Cooperative Dictionary Construction. In *Proceedings of the Data Compression Conference, DCC'96*, pages 200–209, March 1996.

[18] Michael J. Freedman. The Compression Cache: Virtual Memory Compression for Handheld Computers. Technical report, Parallel and Distributed Operating Systems Group, MIT Lab for Computer Science, Cambridge, 2000.

[19] Gregory F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

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

[21] 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.

[22] 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.

[23] M. S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The Optimal Logic Depth Per Pipeline Stage is 6 to 8 Inverter Delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

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

[25] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[26] Morten Kjelso, Mark Gooch, and Simon Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd EUROMICRO Conference*, 1996.

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

[28] 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.

[29] 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.

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

[31] Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, February 2002.

[32] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[33] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[34] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, April 1965.

[35] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks, February 1989. U.S. Patent 4,807,110.

[36] Andre Seznec. Decoupled Sectored Caches. *IEEE Transactions on Computers*, 46(2):210–215, February 1997.

[37] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[38] Systems Performance Evaluation Cooperation. SPEC Benchmarks. http://www.spec.org.

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

[40] 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.

[41] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 101–116, June 1999.

[42] Jun Yang and Rajiv Gupta. Frequent Value Locality and its Applications. *ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.

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

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