Technical Report 1500, Computer Sciences Dept., UW-Madison, April 2004

Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches

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

Abstract

With the widening gap between processor and memory speeds, memory system designers may find cache compression beneficial to increase cache capacity and reduce off-chip bandwidth. Most hardware compression algorithms fall into the dictionary-based category, which depend on building a dictionary and using its entries to encode repeated data values. Such algorithms are effective in compressing large data blocks and files. Cache lines, however, are typically short (32-256 bytes), and a per-line dictionary places a significant overhead that limits the compressibility and increases decompression latency of such algorithms. For such short lines, significance-based compression is an appealing alternative.

We propose and evaluate a simple significance-based compression scheme that has a low compression and decompression overhead. This scheme, Frequent Pattern Compression (FPC) compresses individual cache lines on a word-by-word basis by storing common word patterns in a compressed format accompanied with an appropriate prefix. For a 64-byte cache line, compression can be completed in three cycles and decompression in five cycles, assuming 12 FO4 gate delays per cycle. We propose a compressed cache design in which data is stored in a compressed form in the L2 caches, but are uncompressed in the L1 caches. L2 cache lines are compressed to predetermined sizes that never exceed their original size to reduce decompression overhead. This simple scheme provides comparable compression ratios to more complex schemes that have higher cache hit latencies.

1 Introduction

As semiconductor technology continues to improve, the rising disparity between processor and memory speed increasingly dominates performance. Modern processors use two or more levels of cache memories to reduce effective memory latency and bandwidth. Effectively using the limited on-chip cache resources becomes increasingly important as memory latencies continue to increase relative to processor speeds. Cache compression has previously been proposed to improve performance, since compressing data stored in on-chip caches increases their effective capacity, potentially reducing misses.

Most previous proposals in hardware cache or memory compression (Section 2) are hardware implementations of dictionary-based software compression algorithms (e.g., LZ77 [32]). Such hardware dictionary-based schemes depend mainly on maintaining a per-block dictionary and encoding words (or bytes) that match in the dictionary, while keeping words (bytes) that do not match in their original form with an appropriate prefix.

Schemes such as the Block-Referential Compression with Lookahead (BRCL) used in the IBM MXT memory compression depend on having long enough lines / pages to increase the overall compression ratio [14]. BRCL provides a good compression ratio for 1K-byte or longer blocks. However, cache lines are typically much shorter and BRCL does not perform as well for shorter lines. In addition, decompression latency is high, since the parallel implementation of BRCL decompresses data at a speed of 8 bytes per cycle [26], or 8 cycles for a 64-byte cache line. The X-Match compression scheme [18] tries to compress more data with a small dictionary by allowing partial matches of data words to dictionary entries. Frequent-value cache designs [29, 31] achieve better compression for cache lines by constructing a single dictionary (the Frequent-Value Cache, FVC) for the whole cache, which increases the chance of a single word to be found and compressed. These designs are based on the observation that a few cache values are frequent and thus can be compressed to a fewer number of bits. However, a large FVC requires an increased decompression latency due to the increased FVC access time.

Significance-based compression is based on the observation that most data types (e.g., 32-bit integers) can be stored in a fewer number of bits than the maximum allowed. For example, sign-bit extension is a commonly implemented technique to store small integers (e.g., 8-bit) into 32-bit words, while all the information in the word is stored in the least-significant few bits. In contrast with dictionary-based compression schemes, significance-based compression [9, 11, 12] does not incur a per-line dictionary overhead, which makes it more suitable for the typically-short cache lines. In addition, compression and decompression hardware is faster than dictionary-based encoding and decoding. However, compressibility can be significantly impaired for long cache lines.

In this document, we propose a significance-based compression scheme that provides reasonable compressibility for the typically short cache lines with a relatively fast hardware compression and decompression (Section 3). This scheme, the Frequent Pattern Compression (FPC) compresses a cache line on a word-by-word basis. For each word, FPC detects whether it falls into one of the patterns that can be stored in a smaller number of bits, and stores it in a compressed form with an appropriate prefix. We discuss the implementation of a hardware decompression pipeline that decompresses a 64-byte cache line in five cycles (Section 4). We evaluate this scheme and compare it with other hardware compression schemes in Section 5.

2 Related Work

Several researchers used hardware-based compression to increase effective memory size, reduce memory address and data bandwidth, and increase effective cache size.

IBM's Memory Compression. IBM's MXT technology [26] employs real-time main-memory content compression that can be used to effectively double the main memory capacity without a significant added cost. It was first implemented in the Pinnacle chip [25], a single-chip memory controller. Franaszek, et al. [13], described the design of a compressed random access memory (C-RAM), which formed the basis for the memory organization for the MXT technology, and studied the optimal line size for such an organization. Data in main memory is compressed using a hardware parallelized derivative of the Lempel-Ziv (LZ77) sequential algorithm [32]. This parallel algorithm, Parallel Block-Referential Compression with Directory Sharing, divides the input data block (1 KB in MXT) into sub-blocks (four 256-byte sub-blocks), and cooperatively constructs dictionaries while compressing all sub-blocks in parallel [14]. MXT is shown to have a negligible performance penalty compared to standard memory, and memory contents for many applications and web servers can be compressed by a factor of two to one [1].

Other Hardware Memory Compression Designs. Kjelso, et al. [18], demonstrated that hardware main memory compression is feasible and worthwhile. They used the X-Match hardware compression algorithm that maintains a dictionary and replaces each input data element (whose size is fixed at four bytes) with a shorter code in case of a total or partial match with a dictionary entry. Communication bandwidth is reduced by “compacting” cache-to-memory address streams [12] or data streams [11]. Benini, et al. [8], propose a data compression/decompression scheme to reduce memory traffic in general purpose processor systems. Data is stored uncompressed in the cache, and compressed on the fly when transferred to memory. Memory-to-cache traffic is also decompressed on the fly. They used a differential compression scheme described in [7] that is based on the assumption that it is likely for data words in the same cache line to have some bits in common. Zhang and Gupta [30] introduce a class of common-prefix and narrow-data transformations for general-purpose programs that compress 32-bit addresses and integer words into 15-bit entities. They implemented these transformations by augmenting six data compression extension (DCX) instructions to the MIPS instruction set.

Cache Compression and Related Designs. Lee, et al. [21, 19, 20], propose a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks if they can be reduced to half their original size. Their selective compressed memory system (SCMS) use a hardware implementation of the X-RL compression algorithm [18], a variant of the X-Match algorithm that gives a special treatment for runs of zeros. They propose several techniques to hide decompression overhead, including parallel decompression, selective adaptive compression for blocks that can be compressed to below a certain threshold, and the use of a decompression buffer to be accessed on L1 misses in parallel with L2 access. Ahn, et al. [2], propose several improvements on the X-RL technique that capture common values. Chen, et al. [10], propose a scheme that dynamically partitions the cache into sections of different compressibility, and they use a variant of the LZ compression algorithm. Pomerene, et al. [22], used a shadow directory scheme with more address tags than data blocks to improve upon LRU replacement.

Frequent-Value-Based Compression. Yang and Gupta [28] found out from an analysis of the SPECint95 benchmarks that a small number of distinct values occupy a large fraction of memory access values. This value locality phenomenon enabled them to design energy-efficient caches [27] and data compressed caches [29]. In their compressed cache design, each line in the L1 cache can be either one uncompressed line or two lines compressed to at least half their original sizes based on frequent values [29]. Zhang, et al., designed a value-centric data cache design called the frequent value cache (FVC) [31], which is a small direct-mapped cache dedicated to holding frequent benchmark values. They showed that augmenting a direct mapped cache with a small frequent value cache can greatly reduce the cache miss rate.

Significance-Based Compression. Farrens and Park [12] make use of the fact that many address references transferred between processor and memory have redundant information in their high-order (most significant) portions. They cached these high order bits in a group of dynamically allocated base registers and only transferred small register indexes rather than the high-order address bits between the processor and memory. Citron and Rudolph [11] store common high-order bits in address and data words in a table and transfer only an index plus the low order bits between the processor and memory. Canal, et al. [9], proposed a scheme that compresses data, addresses and instructions into their significant bytes while maintaining a two or three extension bits to maintain significant byte positions. They use this method to reduce dynamic power consumption in a processor pipeline. Kant and Iyer [16] studied the compressibility properties of address and data transfers in commercial workloads, and report that the high-order bits can be predicted with high accuracy in address transfers but with less accuracy for data transfers.

3 Frequent Pattern Compression (FPC)

We propose a compression scheme that builds on significance-based compression schemes [9, 11, 12]. It is also based on the observation that some data patterns are frequent and also compressible to a fewer number of bits. For example, many small-value integers can be stored in 4, 8 or 16 bits, but are normally stored in a full 32-bit word (or 64-bits for 64-bit architectures). These values are frequent enough to merit special treatment, and storing them in a more compact form can increase the cache capacity. In addition, special treatment is also given to runs of zeros since they are very frequent, which is similar to the special treatment in X-RL [18]. The insight behind FPC is that we want to get most of the benefits of dictionary-based schemes, while keeping the per-line overhead at a minimum.

The Frequent Pattern Compression (FPC) compresses / decompresses on a cache line basis. Each cache line is divided into 32-bit words (e.g., 16 words for a 64-byte line). Each 32-bit word is encoded as a 3-bit prefix plus data. Table 1 shows the different patterns corresponding to each prefix.

Each word in the cache line is encoded into a compressed format if it matches any of the patterns in the first six rows of Table 1. These patterns are: a zero run (one or more all-zero words), 4-bit sign-extended (including one-word zero runs), one byte sign-extended, one halfword sign-extended, one halfword padded with a zero halfword, two byte-sign-extended halfwords, and a word consisting of repeated bytes (e.g. “0x20202020”, or similar patterns that can be used for data initialization). These patterns are selected based on their high frequency in many of our integer and commercial benchmarks. A word that doesn’t match any of these categories is stored in its original 32-bit format. All prefix values as well as the zero-run length data bits are stored at the beginning of the line to speed up decompression.

3.1 Segmented Frequent Pattern Compression (S-FPC)

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

Table 1. Frequent Pattern Encoding

Prefix	Pattern Encoded	Data Size
000	Zero Run	3 bits (for runs up to 8 zeros)
001	4-bit sign-extended	4 bits
010	One byte sign-extended	8 bits
011	halfword sign-extended	16 bits
100	halfword padded with a zero halfword	The nonzero halfword (16 bits)
101	Two halfwords, each a byte sign-extended	The two bytes (16 bits)
110	word consisting of repeated bytes	8 bits
111	Uncompressed word	Original Word (32 bits)

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 [23]. Hallnor and Reinhardt's Indirect-Index Cache (IIC) does this across the whole cache, allowing fully-associative placement and a software managed replacement policy [15]. Lee, et al.'s selective compressed caches use this technique to allow two compressed cache blocks to occupy the space required for one uncompressed block [21, 19, 20]. Decoupled access is simpler if we serially access the cache tags before the data. Fortunately, this is increasingly necessary to limit power dissipation [17].

In theory, a cache line can be compressed into any number of bits. This can be achieved in a completely decoupled design across the whole cache (e.g., IIC). However, such design adds more complexity to cache management. In our compressed cache design, the decoupled variable segment cache [5], each cache line is stored as a group of one or more 8-byte *segments*. For example, a 64-byte line can be stored in 1-8 segments. A compressed line is padded with zeros till its size becomes a multiple of the segment size, and these extra zeros (that do not correspond to any tags) are ignored during decompression. While this approach doesn't permit high compression ratios for some cache lines (e.g., all zero lines), it allows for a more practical and faster implementation of cache accesses.

4 Compression and Decompression

We propose a compressed cache design in which data is stored uncompressed in the level-1 caches and compressed in the level-2 caches [5]. This helps reduce many of the costly L2 cache misses that hinder performance, while not affecting the common case of an L1 hit. However, such a design adds the overhead of compressing or decompressing cache lines when moved between the two levels. FPC allows a relatively fast implementations of both of these functions.

Compression. Cache line compression occurs when data is written back from the L1 to the L2 cache. A cache line is compressed easily using a simple circuit that checks each word (in parallel) for pattern matches. If a word matches any of the seven compressible patterns, a simple encoder circuit is used to encode the word into its most compact form. If no match was found, the whole word is stored with the prefix '111'. This can be performed in one cycle, assuming 12 FO4 delays. For zero runs, we need to detect such runs of consecutive zeros, and increment the data value of the first occurrence to represent their count. Since zero runs are limited in our design to eight zeros, this can be implemented in a single cycle using a simple multiplexer/adder circuit.

Cache line compression can be implemented in a memory pipeline, by allocating three pipeline stages on the L1-to-L2 write path (one for pattern matching, one for zero run encoding, and one for gathering the compressed line). A small victim cache that contains a few entries in both compressed and uncompressed form can be used to hide the compression latency on L1 writebacks.

Decompression. Cache line decompression occurs when data is read from the L2 to the L1 caches. This is a frequent event for most benchmarks whose working sets do not fit in the L1 cache. Decompression latency is critical since it is directly added to the L2 hit latency. Decompression is a slower process than compression, since prefixes for all words in the line have to be accessed in series, because each prefix is used to determine the length of its corresponding encoded word and therefore the starting location of all the subsequent compressed words. Figure 1 presents a sche-

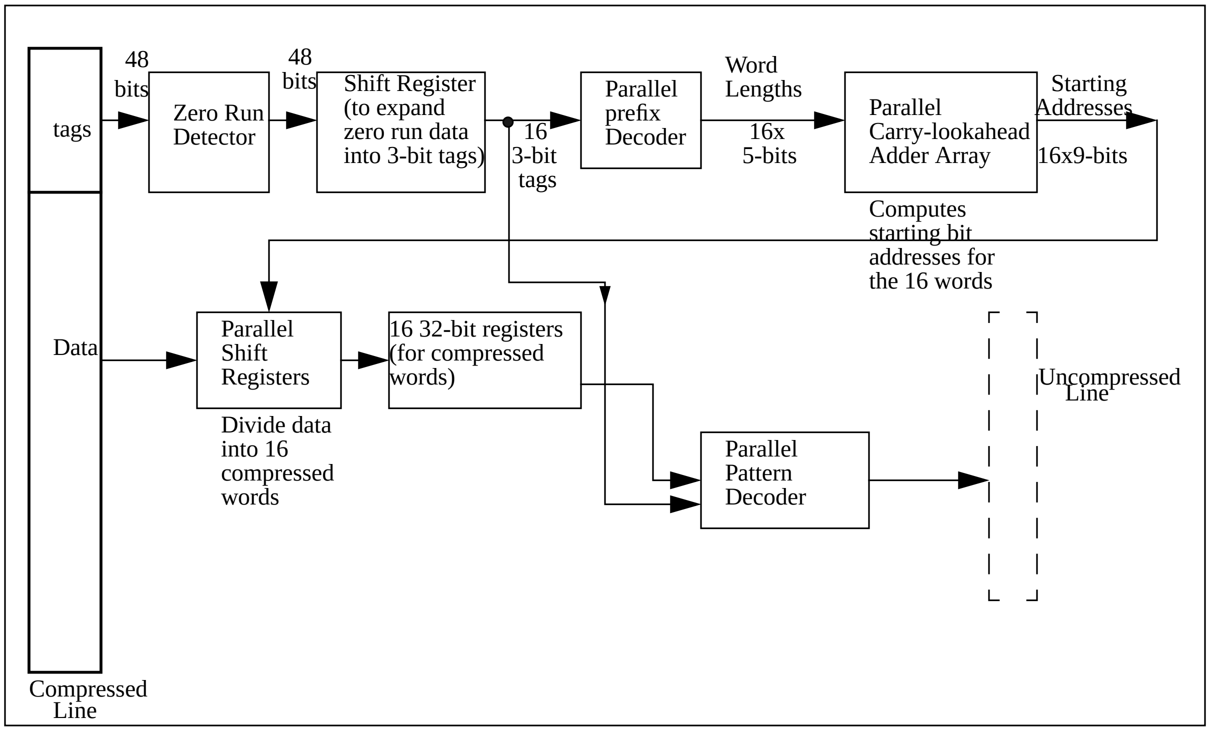
48 bits
tags
Zero Run Detector
48 bits
Shift Register (to expand zero run data into 3-bit tags)
16 3-bit tags
Parallel prefix Decoder
Word Lengths
16x 5-bits
Parallel Carry-lookahead Adder Array
Starting Addresses
16x9-bits
Computes starting bit addresses for the 16 words
Data
Parallel Shift Registers
Divide data into 16 compressed words
16 32-bit registers (for compressed words)
Parallel Pattern Decoder
Uncompressed Line
Compressed Line

Figure 1. Cache line decompression pipeline for a 64-byte (16-word) cache line.

This is a five-stage pipeline used to decompress a compressed cache line, where each stage contains 12 FO4 gate delays or less. The first pipeline stage (containing the zero run detector, shift register and parallel prefix decoder) decodes the prefix array to determine the length in bits of each word. The second and third stages (Parallel Prefix adder array) compute the starting bit address for each data word by adding the length fields of the preceding words in a hierarchical fashion. The fourth stage (parallel shift registers) contains 16 registers each of which is shifted by the starting address of its word. The fifth and last stage contains the pattern decoder, which decodes the content of each 32-bit register into an uncompressed word according to its corresponding prefix.

matic diagram for a five-stage hardware pipeline that can be used to decompress 64-byte cache lines. Each pipeline stage is 12 FO4 delays or less, assuming the parallel resources required are available for the parallel adder, shift register and pattern decoder. Assuming one processor cycle requires 12 FO4 gate delays, this means that the decompression latency is limited to 5 processor cycles.

5 Evaluation

We evaluate our FPC scheme in terms of its achieved compressibility compared to other compression schemes. We show compression results for our frequent patterns, and demonstrate that zero runs are the most frequent. We also analyze the performance of segmented compression, and the effect of restricting compressed lines to segment boundaries on compression ratios.

5.1 Workloads

To evaluate our design against alternative schemes, we used several multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [3]. We also used six of the SPEC [24] benchmarks, three from the integer suite (SPECint2000) and three from the floating point suite (SPECfp2000). All of these workloads run under the

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.
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, and a warmup interval of 200,000 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 [6] 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.
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).
SPEC. We use three integer benchmarks (bzip, gcc, and mcf) and three floating point benchmarks (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 warm up caches of each benchmark run for 1 billion instructions.

Solaris 9 operating system. These workloads are briefly described in Table 2. For each data point in our results, we present the average and the 95% confidence interval of multiple simulations to account for space variability [4].

5.2 Compression Ratio

To evaluate the success of our compression scheme, we estimated the compressibility properties of our set of benchmarks. A snapshot is taken of the L2 cache contents for each of these benchmarks after a warm-up interval. Assuming variable length cache lines that can occupy any number of bits, we compare the compression ratio from our Frequent Pattern Compression scheme (FPC) with two other memory compression schemes:

- The X-RL algorithm [18] used in some compressed cache implementations [21, 19, 20].
- The Block-Referential Compression with Lookahead (BRCL) scheme [14], which is an upper bound for the parallel compression scheme used for memory compression in the IBM MXT technology [26]. We apply it here to cache lines.

We also compare against the "Deflate" algorithm used in the gzip unix utility, which combines an LZ-variant implementation with Huffman encoding of codewords in the dictionary. For this algorithm, we run the gzip utility on the whole cache snapshot file (as opposed to 64-byte lines individually compressed by the other three schemes). The "Deflate" algorithm is used to provide a practical bound on compressibility of dictionary-based schemes for arbitrarily long cache lines.

Figure 2 shows results that compare the four compression schemes. While FPC is faster to implement in hardware, it provides comparable compression ratios to the dictionary-based XRL and BRCL, and even approaches gzip for some benchmarks. FPC is slightly better than XRL and BRCL for the four commercial benchmarks.

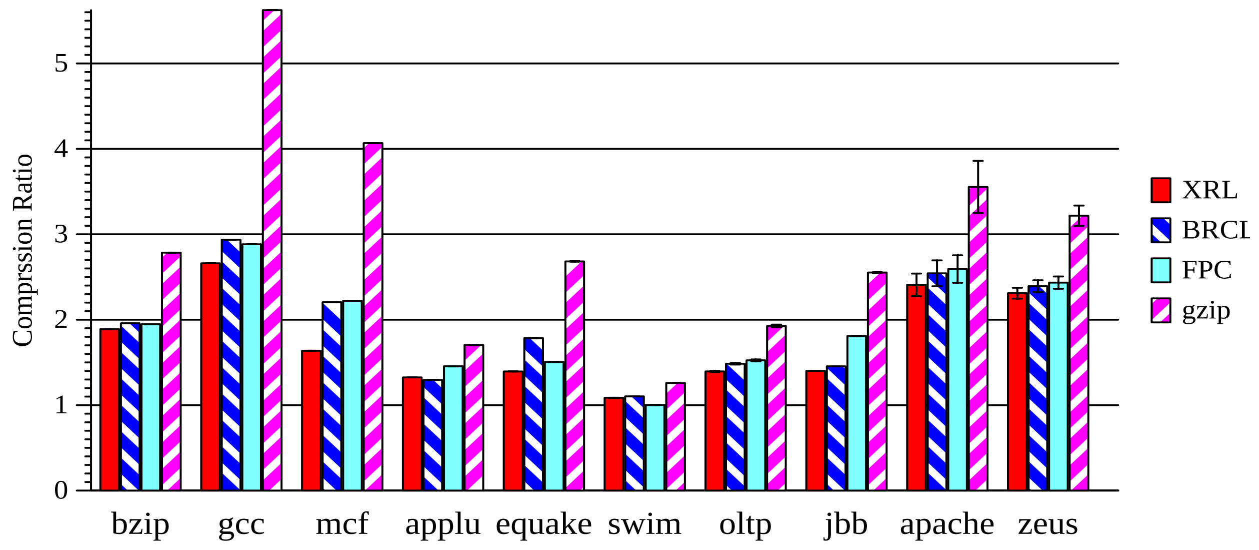

Figure 2. Compression ratios (original size / compressed size) for XRL, BRCL, FPC and gzip

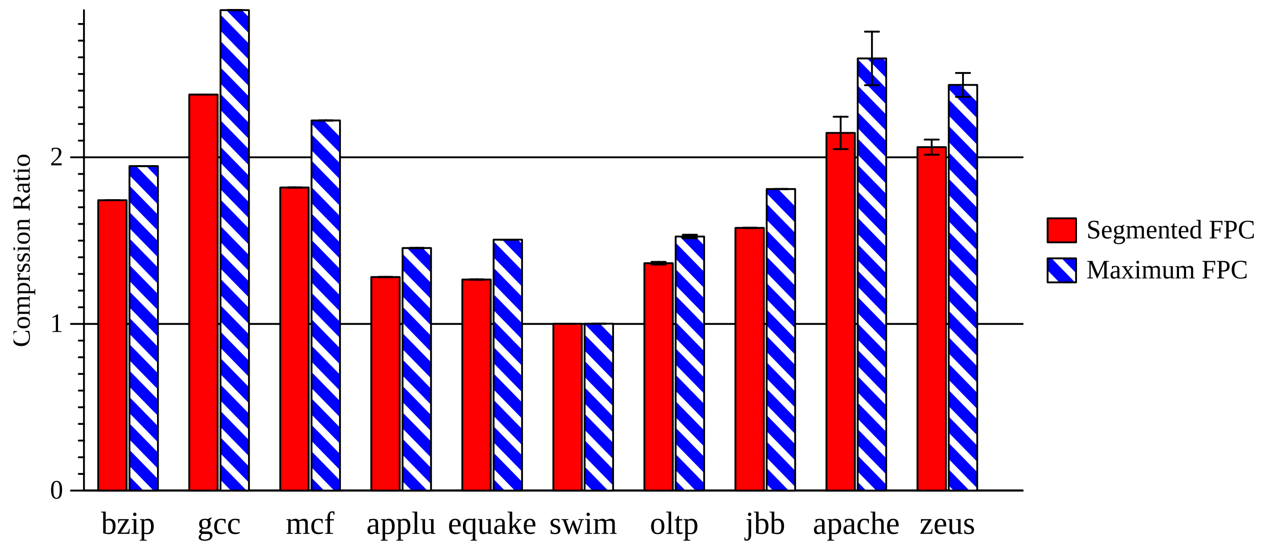

Figure 3. Compression ratios for segmented and variable-length FPC

In most practical cache designs, cache lines cannot occupy any arbitrary number of bits. Restricting the compressed line sizes to a certain subset of all possible lengths (as we do in our segmented design) partially reduces compressibility. To assess the loss in compressibility, we compare the compression ratio from our Segmented Frequent Pattern Compression scheme (Segmented-FPC) against the compression ratio from the Frequent Pattern Compression scheme assuming variable-length lines are possible (Maximum-FPC).

Figure 3 shows the compression ratios from the two schemes for our ten benchmarks. The simple scheme (Segmented-FPC) has compression ratios of 1.7-2.4 for the three SPECint2000 benchmarks, 1.0-1.3 for the three SPECfp2000 benchmarks, 1.4-2.1 for the four commercial benchmarks. OLTP had the lowest compression ratio among our set of commercial benchmarks, since its data is randomly generated. A real OLTP application would have much less randomness, and thus have a higher compression ratio.

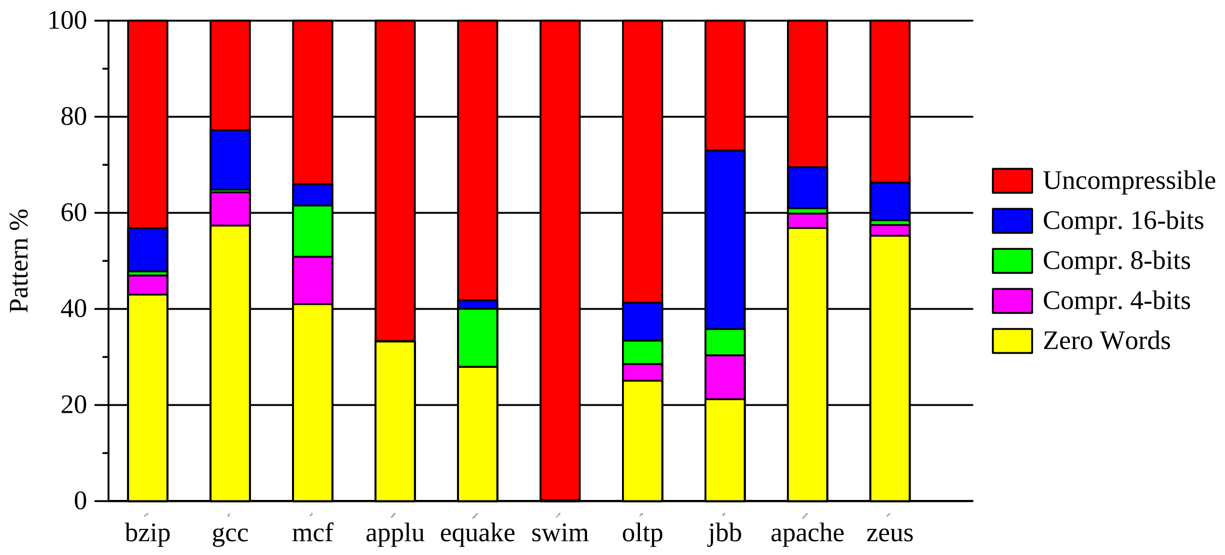

Figure 4. Frequent Pattern Histogram

Compression ratios are higher for all integer benchmarks (1.4-2.4) than floating point benchmarks. For example, only 0.4% of all cache lines in *swim* are compressible. This is because of the nature of floating point benchmarks where floating point numbers might not fit any of the frequent patterns. However, some benefit is still possible for benchmarks with lots of zero words. Segmented-FPC achieved most of the compression benefit from variable-length lines of Maximum-FPC.

5.3 Which Patterns Are Frequent?

Frequent Pattern Compression (FPC) is built on the observation that some word patterns are more frequent than others. We experimented with cache snapshots for our different benchmarks to come up with a reasonable set of frequent patterns (described in Table 1). Figure 4 shows the relative frequency of incompressible words, zero words and words compressible to 4, 8 and 16 bits. The 4-, 8-, and 16-bit patterns are present with various frequencies across our integer and commercial benchmarks. Unfortunately, most of the words in floating point benchmarks are incompressible with FPC, since our patterns are mainly integer patterns.

As Figure 4 demonstrates, zero words are the most frequent compressible pattern across all benchmarks, which is why some compression techniques (e.g., X-RL) specifically optimize for runs of zeros. Figure 5 shows the average number of zeros in a zero run for our set of benchmarks. Except for equake and jbb, the average zero run length for all benchmarks is greater than two. In developing the FPC scheme, we had two options to compress zeros. The first was to have a prefix for each zero word with no data. The second was to encode zero runs with a single prefix and save the length of the run in the data part corresponding to that prefix. However, since most zero runs have more than two (and in most cases three) words, the additional compressibility justifies having special treatment for zero runs.

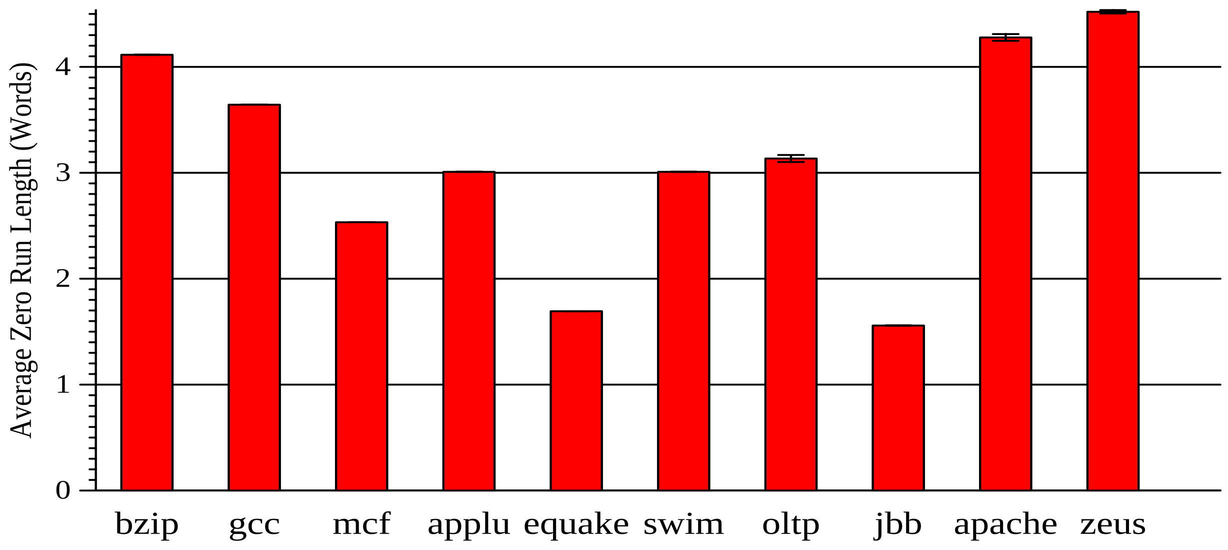

Figure 5. Average number of words in a zero run for our ten benchmarks

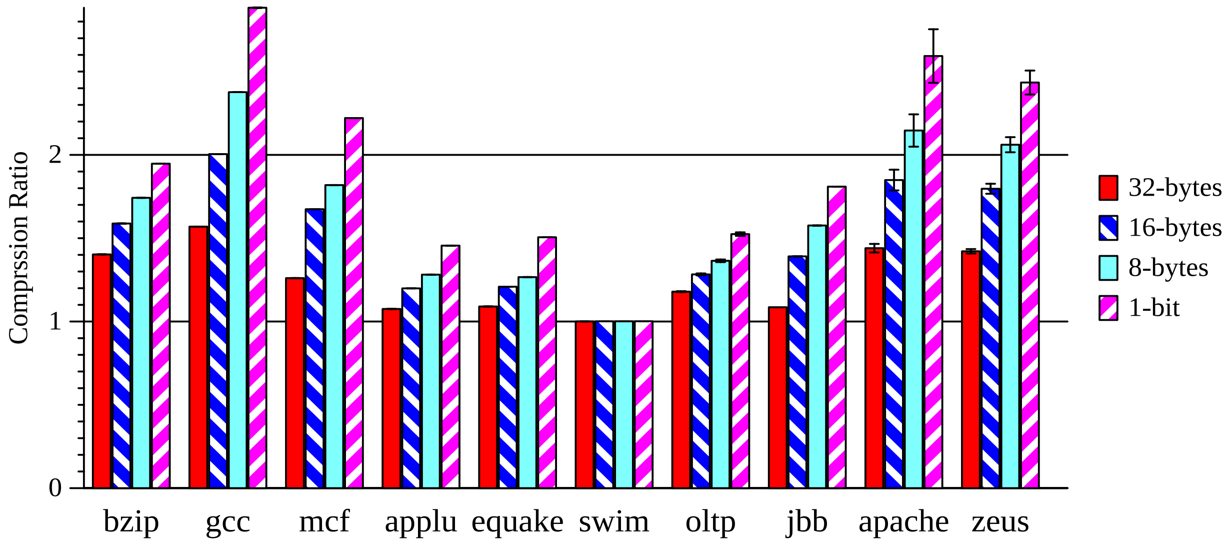

Figure 6. FPC Compression ratios for segment sizes (1 bit to 32 bytes)

5.4 Analysis of Segmented Frequent Pattern Compression

In designing a practical compressed cache implementation, selecting a specific base segment size is critical. A compressed line can only be stored in a size that is an integer multiple of the base segment size. Smaller segments allow for higher compression ratios. On the other hand, larger segments decrease the cache design complexity. Cache design should balance the tradeoff between these two conflicting issues. We selected a base segment size of 8-bytes (i.e., up to 8 segments for 64-byte lines) in our Segmented FPC design.

Figure 6 shows the sensitivity of our compression schemes to the base segment size. The four bars for each benchmark represent compression ratios if we have two possible sizes, i.e., an uncompressed line occupying two segments (32-byte segments), four (16--byte segments), eight (8-byte segments, which is the same as Segmented-FPC in

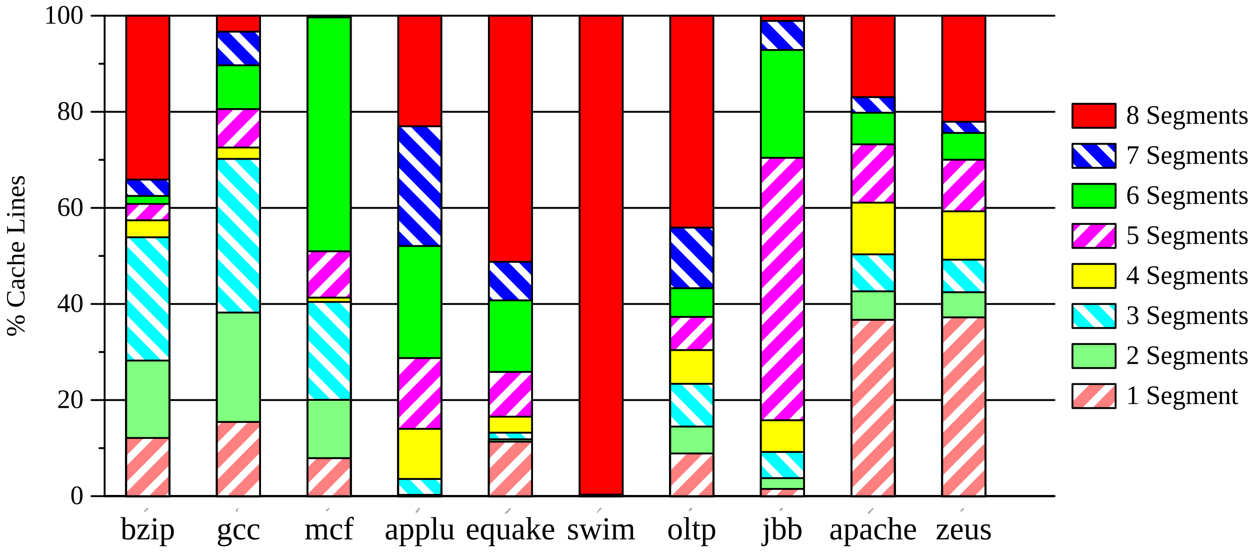

Figure 7. Segment Length Histogram: Percentage of cache lines compressed into 1-8 Segments

Figure 3), and all possible sizes (1-bit segments, the same as Maximum-FPC). Our 8-byte-segment design increases the compression ratio by up to 52% vs. 32-byte segments, and up to 19% vs. 16-byte segments. Figure 7 shows the percentage of lines that can be compressed into 1-8 segments. We show a more detailed distribution in Figure 8, demonstrating the cumulative distribution of compressed cache line sizes (1-512 bits) for our ten benchmarks, as well as the 25th, 50th and 75th percentiles.

6 Conclusion

Cache designers might consider using cache compression to increase cache capacity and reduce off-chip bandwidth. In this document, we propose and evaluate a simple significance-based compression scheme suitable for cache lines, since it has a low compression and decompression overhead. This scheme, Frequent Pattern Compression (FPC) compresses individual cache lines on a word-by-word basis by storing common word patterns in a compressed format accompanied with an appropriate prefix. This simple scheme provides comparable compression ratios to more complex schemes that have higher cache hit latencies.

References

[1] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. Performance of hardware compressed main memory. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture,* pages 73–81, January 2001.

[2] 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.

[3] 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.

[4] 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.

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

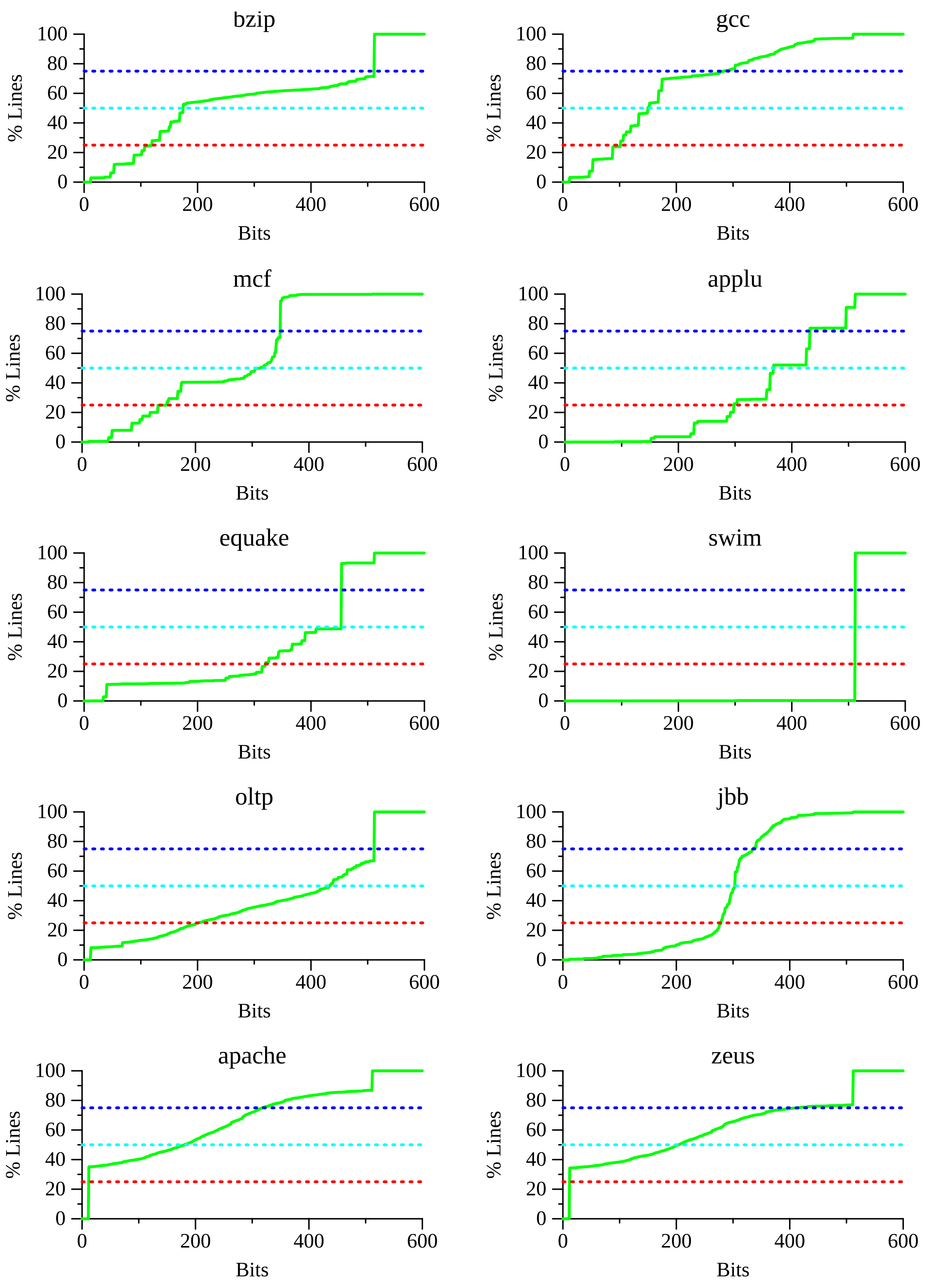

Figure 8. Cumulative Distribution of Compressed Line Lengths (1 to 512 bits). These graphs highlight the 25th, 50th and 75th percentile values.

[6] 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.

[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] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very Low Power Pipelines Using Significance Compression. In *Proceedings 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 *Proceedings of the Singapore-MIT Alliance Symposium*, January 2003.

[11] 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.

[12] 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.

[13] P.A. Franaszek and J.T. Robinson. On Internal Organization in Compressed Random-Access Memories. *IBM Journal of Research and Development*, 45(2):259–270, March 2001.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] R. Brett Tremaine, T. Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 21(2):56–68, March/April 2001.

[26] 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.

[27] Jun Yang and Rajiv Gupta. Energy Efficient Frequent Value Data Cache Design. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–207, November 2002.

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

[29] 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.

[30] Youtao Zhang and Rajiv Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 24–28, April 2002.

[31] 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.

[32] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.