Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches

Gennady Pekhimenko
gpekhime@cs.cmu.edu

Vivek Seshadri
vseshadr@cs.cmu.edu

Onur Mutlu
onur@cmu.edu

Todd C. Mowry
tcm@cs.cmu.edu

Phillip B. Gibbons
phillip.b.gibbons@intel.com

Michael A. Kozuch
michael.a.kozuch@intel.com

Carnegie Mellon University

SAFARI Technical Report No. 2012-001

June 19, 2012

Abstract

Cache compression is a promising technique to increase cache capacity and to decrease on-chip and off-chip bandwidth usage. Unfortunately, directly applying well-known compression algorithms (usually implemented in software) leads to high hardware complexity and unacceptable decompression/compression latencies, which in turn can negatively affect performance. Hence, there is a need for a simple yet efficient compression technique that can effectively compress common in-cache data patterns, and has minimal effect on cache access latency.

In this paper, we propose a new compression algorithm called **Base-Delta-Immediate (BΔI)** compression, a practical technique for compressing data in on-chip caches. The key idea of the algorithm is that, for many cache lines, the values within the cache line have a low dynamic range – i.e., the differences between values stored within the cache line are small. As a result, a cache line can be represented using a base value and an array of differences whose combined size is much smaller than the original cache line (we call this the **base+delta** encoding). Moreover, many cache lines intersperse such base+delta values with small values – our BΔI technique efficiently incorporates such **immediate** values into its encoding.

Compared to prior cache compression approaches, our studies show that BΔI strikes a sweet-spot in the tradeoff between compression ratio, decompression/compression latencies, and hardware complexity. Our results show that BΔI compression improves performance for both single-core (8.1% improvement) and multi-core workloads (9.5% / 11.2% improvement for two/four cores). For many applications, BΔI provides the performance benefit of doubling the cache size of the baseline system, effectively increasing average cache capacity by 1.53X.

1 Introduction

To mitigate the latency and bandwidth limitations of accessing main memory, modern microprocessors contain multi-level on-chip cache hierarchies. While caches have a number of design parameters and there is a large body of work on using cache hierarchies more effectively (e.g., [10, 16, 20, 21]), one key property of a cache that has a major impact on performance, die area, and power consumption is its *capacity*. The decision of how large to make a given cache involves tradeoffs: while larger caches often result in fewer cache misses, this potential benefit comes at the cost of a longer access latency and increased area and power consumption.

As we look toward the future with increasing numbers of on-chip cores, the issue of providing sufficient capacity in shared L2 and L3 caches becomes increasingly challenging. Simply scaling cache capacities linearly with the number of cores may be a waste of both chip area and power. On the other hand, reducing the L2 and L3 cache sizes may result in excessive off-chip cache misses, which are especially costly in terms of latency and precious off-chip bandwidth.

One way to potentially achieve the performance benefits of larger cache capacity without suffering all disadvantages is to exploit *data compression* [2, 9, 11, 12, 33, 34]. Data compression has been successfully adopted in a number of different contexts in modern computer systems [13, 35] as a way to conserve storage capacity and/or data bandwidth (e.g., downloading compressed files over the Internet [24] or compressing main memory [1]). However, it has not been adopted by modern commodity microprocessors as a way to increase effective cache capacity. Why not?

The ideal cache compression technique would be *fast*, *simple*, and *effective* in saving storage space. Clearly, the resulting compression ratio should be large enough to provide a significant upside, and the hardware complexity of implementing the scheme should be low enough that its area and power overheads do not offset its benefits. Perhaps the biggest stumbling block to the adoption of cache compression in commercial microprocessors, however, is *decompression latency*. Unlike cache *compression*, which takes place in the background upon a cache fill (after the critical word is supplied), cache *decompression* is on the critical path of a *cache hit*, where minimizing latency is extremely important for performance. In fact, because L1 cache hit times are of utmost importance, we only consider compression of the L2 caches and beyond in this study (even though our algorithm could be applied to any cache).

Because the three goals of having *fast*, *simple*, and *effective* cache compression are at odds with each other (e.g., a very simple scheme may yield too little compression, a scheme with a very high compression ratio may be too slow, etc.), the challenge is to find the right balance between these goals. Although several cache compression techniques have been proposed in the past [2, 6, 7, 11, 33], they suffer from either a small compression ratio [7, 33], high hardware complexity [11], or large decompression latency [2, 6, 11, 33]. To achieve significant compression ratios while minimizing hardware complexity and decompression latency, we propose a new cache compression technique called **Base-Delta-Immediate (BΔI)** compression.

1.1 Our Approach: BΔI Compression

The key observation behind **Base-Delta-Immediate (BΔI)** compression is that, for many cache lines, the data values stored within the line have a *low dynamic range*: i.e., the relative difference between values is small. In such cases, the cache line can be represented in a compact form using a common *base* value plus an array of relative differences ("*deltas*"), whose combined size is much smaller than the original cache line. (Hence the *"base"* and *"delta"* portions of our scheme's name).

We refer to the case with a single arbitrary base as *Base+Delta* (B+Δ) compression, and this is at the heart of all of our designs. To increase the likelihood of being able to compress a cache line, however, it is also possible to have *multiple bases*. In fact, our results show that for the workloads we studied, the best option is to have *two bases*, where one base is always *zero*. (The deltas relative to zero can be thought of as small *immediate* values, which explains the last word in the name of our BΔI compression scheme) Using these two base values (zero and something else), our scheme can efficiently compress cache lines containing a mixture of two separate dynamic ranges: one centered around an arbitrary value chosen from the actual contents of the cache line (e.g., pointer values), and one close to zero (e.g., small integer values). Such mixtures from two dynamic ranges are commonly found (e.g., in pointer-linked data structures, etc.), as we will discuss later.

As demonstrated later in this paper, BΔI compression offers the following advantages: (i) a *high compression ratio* since it can exploit a number of frequently-observed patterns in cache data (as shown using examples from real applications and validated in our experiments); (ii) *low decompression latency* since decompressing a cache line only requires a simple masked vector addition; and (iii) *relatively modest hardware overhead and implementation complexity*, since both the compression and decompression algorithms involve only simple vector addition, subtraction, and comparison operations.

This paper makes the following contributions:

- We propose a new cache compression algorithm, Base-Delta-Immediate Compression (BΔI), which exploits the low dynamic range of values present in many cache lines to compress them to smaller sizes. Both the compression and decompression algorithms of BΔI have low latency and require only vector addition, subtraction and comparison operations.

- Based on the proposed BΔI compression algorithm, we introduce a new compressed cache design. This design achieves a high degree of compression at a lower decompression latency compared to two state-of-the-art cache compression techniques: Frequent Value Compression
(FVC) [33] and Frequent Pattern Compression (FPC) [2], which require complex and long-latency decompression pipelines as shown in [3].

- We evaluate the performance benefits of BΔI compared to a baseline system that does not employ compression, as well as against three state-of-the-art cache compression techniques [2, 33, 7]. We show that BΔI provides a better or comparable degree of compression for the majority of the applications we studied. It improves performance for both single-core (8.1%) and multi-core workloads (9.5% / 11.2% for two- / four-cores), and for many applications the benefit from compression with BΔI provides the performance benefits of doubling the uncompressed cache size of the baseline system.

2 Background and Motivation

Data compression is a powerful technique to store large amounts of data in a smaller space. Applying data compression to an on-chip cache can potentially allow the cache to store more cache lines in compressed form than it could have if the cache lines were not compressed. As a result, a compressed cache has the potential to provide the benefits of a larger cache at the area and the power of a smaller cache.

Prior work has observed that there is a significant amount of redundancy in the data accessed by real-world applications. There are multiple patterns that lead to such redundancy. We summarize the most common of such patterns below.

Zeros: Zero is by far the most frequently seen value in application data [4, 8, 33]. There are various reasons for this. Zero is most commonly used, for example, to initialize data, used to represent NULL pointers or false boolean values, and is widely present in sparse matrices. In fact, a majority of the compression schemes proposed for compressing memory data either base their design fully around zeros [8, 7, 14, 31], or treat zero as a special case [2, 34, 32].

Repeated Values: A large contiguous region of memory may contain a single value repeated multiple times [23]. This pattern is widely present in applications that use a common initial value for a large array, or in multimedia applications where a large number of adjacent pixels have the same color. Such a repeated value pattern can be easily compressed to significantly reduce storage requirements. Simplicity, frequent usage, and high compression ratio make repeated values an attractive target for a special consideration in data compression [2].

Narrow Values: A narrow value is a small value stored using a large data type – e.g., a one-byte value stored as a four-byte integer. Narrow values appear commonly in application data due to over-provisioning or data alignment. Programmers, in general, provision the data types in various data structures for the worst case even though a majority of the values may fit in a smaller data type. For example, storing a table of counters requires the data type to be provisioned to accommodate the maximum possible value for the counters. However, it can be the case that the maximum possible counter value needs four bytes, while one byte might be enough to store the majority of the counter values. Optimizing such data structures in software for the common case necessitates significant overhead in code, thereby increasing the complexity of the program and programmer effort to ensure correctness. Therefore, most programmers over-provision data type sizes. As a result, narrow values present themselves in many applications, and are exploited by different compression techniques [2, 32, 15].

Other Patterns: There are a few other common data patterns that do not fall into any of the above three classes – a table of pointers that point to different locations in the same memory region, an image with low color gradient, etc. Such data can also be compressed using simple techniques and has been exploited by some prior proposals for main memory compression [32] and image compression [28].

In this work, we make two observations. First, we find that the above described patterns are widely present in many applications (SPEC CPU benchmark suites, and some server applications, e.g., Apache, TPC-H). Figure 1 plots the percentage of cache lines that can be compressed using different patterns.[1] As the figure shows, on average, 43% of all cache lines belonging to these applications can be compressed. This shows that there is significant opportunity to exploit data compression to improve on-chip cache performance.

Second, and more importantly, we observe that all the above commonly occurring patterns fall under the general notion of *low dynamic range* – a set of values where the differences between the values is much smaller than the values themselves. Unlike prior work, which has attempted to exploit each of these special patterns individually for cache compression [2, 33] or main memory compression [8, 32], our **goal** is to exploit the general case of values with *low dynamic range* to build a simple yet effective compression technique.

Summary comparison: Our resulting mechanism, base-delta-immediate (BΔI) compression, strikes a sweet-spot in the tradeoff between decompression latency (Decomp. Lat.), hardware complexity of the implementation (Complexity), and compression ratio (C. Ratio), as shown in Table 1. The table qualitatively compares BΔI with three state-of-the-art mechanisms: ZCA [7], which does zero-value compression, Frequent Value Compression (FVC) [33], and Frequent Pattern Compression (FPC) [2]. (These mechanisms are described in further detail in Section 6.) It also

[1]The methodology used in this and other experiments is described in Section 7. We use a 2MB L2 cache unless otherwise stated.

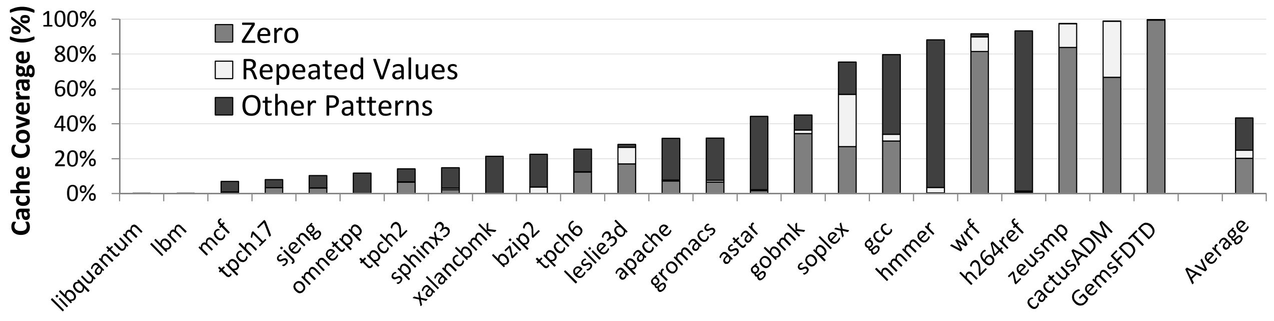


Figure 1: Cache lines distribution with different compressible data patterns in 2MB L2 cache. "Other Patterns" includes "Narrow Values".

Mechanism	Characteristics			Compressible data patterns			
	Decompression Latency	Complexity	Compression Ratio	Zeros	Repeated Values	Narrow	Low Dynamic Range
ZCA [7]	**Low**	**Low**	Low	✔	×	×	×
FVC [33]	High	High	Modest	✔	Partly	×	×
FPC [2]	High	High	**High**	✔	✔	✔	×
BΔI	**Low**	Modest	**High**	✔	✔	✔	✔

Table 1: Qualitative comparison of BΔI with prior work. Bold font indicates desirable characteristics.

summarizes which data patterns (zeros, repeated values, narrow values, and other low dynamic range patterns) are compressible with each mechanism. For modest complexity, BΔI is the only design to achieve both low decompression latency and high compression ratio.

We now explain the design and rationale for our scheme in two parts. In Section 3, we start by discussing the core of our scheme, which is *Base+Delta (B+Δ)* compression. Building upon B+Δ, we then discuss our full-blown BΔI compression scheme (with multiple bases) in Section 4.

3 Base + Delta Encoding: Basic Idea

We propose a new cache compression mechanism, *Base+Delta* (B+Δ) compression, which unlike prior work [2, 7, 33], looks for compression opportunities at a cache line granularity – i.e., B+Δ either compresses the entire cache line or stores the entire cache line in uncompressed format. The key observation behind B+Δ is that many cache lines contain data with low dynamic range. As a result, the differences between the words within such a cache line can be represented using fewer bytes than required to represent the words themselves. We exploit this observation to represent a cache line with low dynamic range using a common *base* and an array of *deltas* (differences between values within the cache line and the common base). Since the *deltas* require fewer bytes than the values themselves, the combined size of the *base* and the array of *deltas* will be much smaller than the size of the original uncompressed cache line.

The fact that some values can be represented in base+delta form has been observed by others, and used for different purposes, e.g. texture compression in GPUs [28] and also to save bandwidth on CPU buses by transferring only deltas from a common base [9]. To our knowledge, no previous work examined the use of base+delta representation to improve on-chip cache utilization in a general-purpose processor.

To evaluate the applicability of the B+Δ compression technique for a large number of applications, we conducted a study that compares the effective compression ratio (effective cache size increase, see Section 7) of B+Δ against two common data patterns (zeros and repeated values[2]). Figure 2 shows the results of this study for a 2MB L2 cache with 64-byte cache lines for applications in the SPEC CPU2006 benchmark suite, database and web-server workloads. We assume a design where a compression scheme can store up to twice as many tags compressed cache lines than the number of cache lines stored in the uncompressed baseline cache (Section 5 describes a practical mechanism that

[2]Zero compression compresses an all-zero cache line into a bit that just indicates that the cache line is all-zero. Repeated values checks if a cache line has the same 1/2/4/8 byte value repeated. If so, it compresses the cache line to the corresponding value.

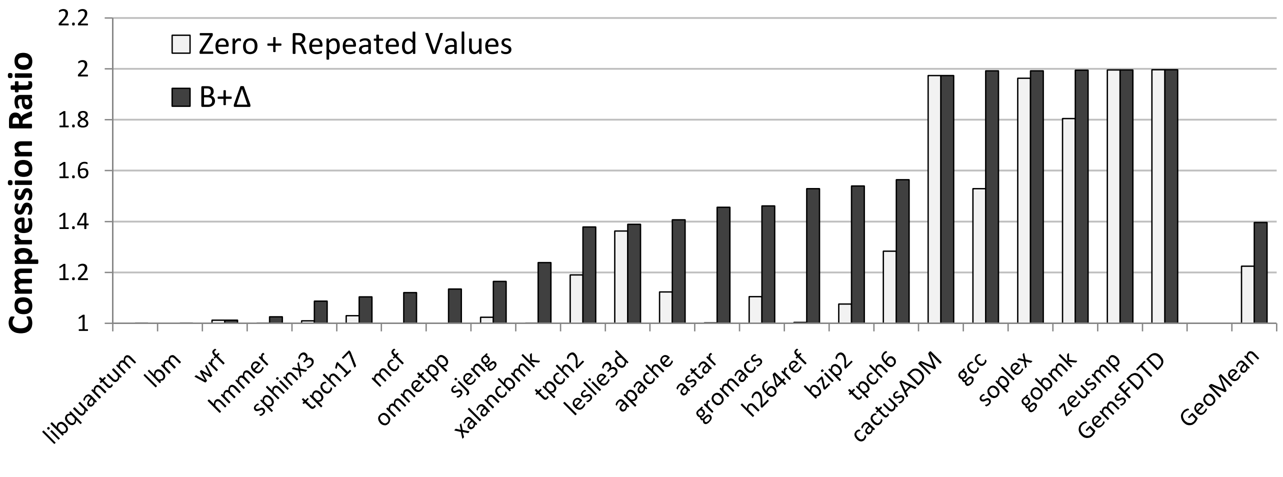


Figure 2: Effective compression ratio with different value patterns.

achieves this by using twice the number of tags). As the figure shows, for a number of applications, B+Δ provides significantly higher compression ratio (1.4× on average) than just using the simple compression scheme. However, there are some benchmarks for which B+Δ provides very little or no benefit (e.g., *libquantum*, *lbm*, and *mcf*). We will address this problem with a new compression technique called BΔI in Section 4. We now provide examples from real applications to show why B+Δ works.

3.1 Why Does B+Δ Work?

B+Δ works because of: (1) regularity in the way data is allocated in the memory (similar data values and types grouped together), (2) low dynamic range of cache/memory data. The first reason is typically true due to the common usage of arrays to represent large pieces of data in applications. The second reason is usually caused either by the nature of computation, e.g., sparse matrices or streaming applications; or by inefficiency (over-provisioning) of data types used by many applications, e.g., 4-byte integer type used to represent values that usually need only 1 byte. We have carefully examined different common data patterns in application programs that lead to B+Δ representation and summarize our observations in two examples.

Figures 3 and 4 show the compression of two 32-byte[3] cache lines from the applications *h264ref* and *perlbench* using B+Δ. The first example from *h264ref* shows a cache line with a set of narrow values stored as 4-byte integers. As Figure 3 indicates, in this case, the cache line can be represented using a single 4-byte base value, 0, and an array of eight 1-byte differences. As a result, the entire cache line data can be represented using 12 bytes instead of 32 bytes saving 20 bytes of the originally used space. Figure 4 shows a similar phenomenon where nearby pointers are stored in the same cache line for the *perlbench* application.

We now describe more precisely the compression and decompression algorithms that lay at the heart of the B+Δ compression mechanism.

3.2 Compression Algorithm

The B+Δ compression algorithm views a cache line as a set of fixed size values i.e., 8 8-byte or 32 2-byte values for a 64-byte cache line. It then determines if the set of values can be represented in a more compact form as a base value with a set of differences from the base value. For analysis, let us assume that the cache line size is C, the size of each value in the set is k bytes and the set of values to be compressed is $S = (v_1, v_2, ..., v_n)$, where $n = \frac{C}{k}$. The goal of the compression algorithm is to determine the value of the base, B^* and the size of values in the set, k, that provide maximum compressibility. Once B^* and k are determined, the output of the compression algorithm is $\{k, B^*, \Delta = (\Delta_1, \Delta_2, ..., \Delta_n)\}$, where $\Delta_i = B^* - v_i \ \ \forall i \in \{1, .., n\}$.

Observation 1: The cache line is compressible *only if*
$\forall i, \max(\text{size}(\Delta_i)) < k$, where $size(\Delta_i)$ is the smallest number of bytes that is needed to store Δ_i.

[3]We use 32-byte cache lines in our examples to save space. 64-byte cache lines were used in all evaluations (see Section 7).

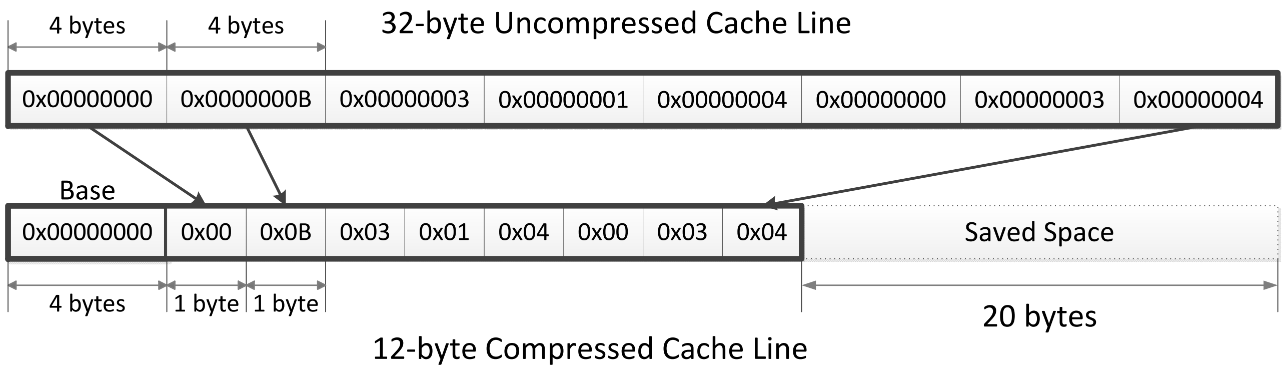


Figure 3: Cache line from *h264ref* compressed by B+Δ.

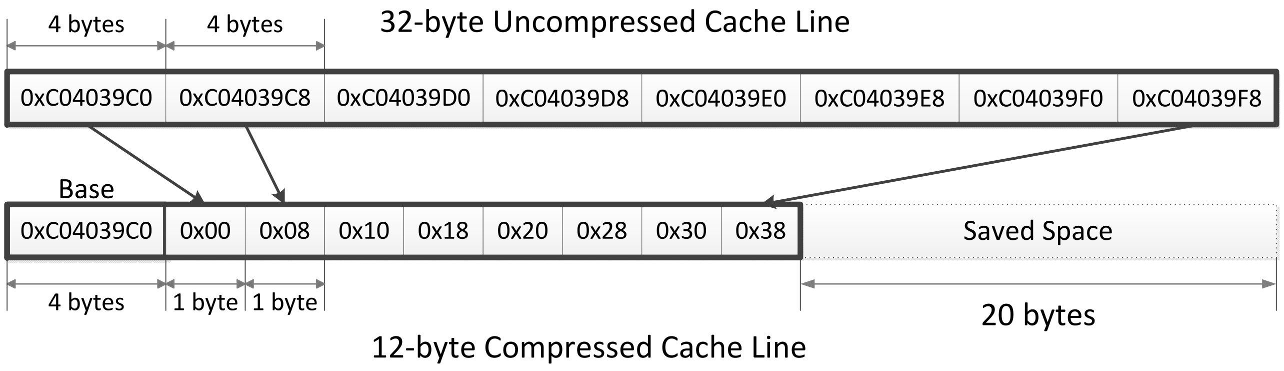


Figure 4: Cache line from *perlbench* compressed by B+Δ.

In other words, for the cache line to be compressible, the number of bytes required to represent the differences must be strictly less than the number of bytes required to represent the values themselves.

Observation 2: To determine the value of B^*, either the value of $\min(S)$ or $\max(S)$ needs to be found.

The reasoning, where $\max(S)/\min(S)$ are the maximum and minimum values in the cache line, is based on the observation that the values in the cache line are bounded by $\min(S)$ and $\max(S)$. And, hence, the optimum value for B^* should be between $\min(S)$ and $\max(S)$. In fact, the optimum can be reached only for $\min(S)$, $\max(S)$, or exactly in between them. Any other value of B^* can only increase the number of bytes required to represent the differences.

Given a cache line, the B+Δ compression algorithm needs to determine two parameters: (1) k, the size of each value in S, and (2) B^*, the optimum base value that gives the best possible compression for the chosen value of k.

Determining k. Note that the value of k determines how the cache line is viewed by the compression algorithm – i.e., it defines the set of values that are used for compression. Choosing a single value of k for all cache lines will significantly reduce the opportunity of compression. To understand why this is the case, consider two cache lines, one representing a table of 4-byte pointers pointing to some memory region (similar to Figure 4) and the other representing an array of narrow values stored as 2-byte integers. For the first cache line, the likely best value of k is 4, as dividing the cache line into a set of of values with a different k might lead to increase in dynamic range and reduce the possibility of compression. For a similar reason, the likely best value of k for the second cache line is 2.

Therefore, to increase the opportunity for compression by catering to multiple patterns, our compression algorithm attempts to compress a cache line using three different potential values of k simultaneously: 2, 4 and 8. The cache line is then compressed using the value that provides the maximum compression rate or not compressed at all.[4]

Determining B^*. For each possible value of $k \in \{2, 4, 8\}$, the cache line is split into values of size k and the best value for the base, B^* can be determined using Observation 2. However, computing B^* in this manner requires computing the maximum or the minimum of the set of values, which adds logic complexity and significantly increases

[4]We restrict our search to these three values as almost all basic data types supported by various programming languages have one of these three sizes.

the latency of compression.

To avoid compression latency increase and reduce hardware complexity, we decide to use the *first* value from the set of values as an approximation for the B^*. For a compressible cache line with a low dynamic range, we find that choosing the first value as the base instead of computing the optimum base value reduces the average compression ratio only by 0.4%.

3.3 Decompression Algorithm

To decompress a compressed cache line, the B+Δ decompression algorithm needs to convert the base value B^* and an array of differences $\Delta = \Delta_1, \Delta_2, ..., \Delta_n$ and generate the corresponding cache line values $S = (v_1, v_2, ..., v_n)$. For B+Δ, the value v_i is simply given by $v_i = B^* + \Delta_i$. As a result, all the values of the cache line can be computed in parallel using a SIMD-style vector adder. Consequently, the entire cache line can be decompressed in a single cycle using simple adders.

4 BΔI Compression

4.1 Why Could Multiple Bases Help?

Although B+Δ proves to be generally applicable for many applications, it is clear that not every cache line can be represented in this form, and, as a result, some benchmarks do not have high compression ratio, e.g., *mcf*. One common reason why this happens is that some of these applications can mix data of different types in the same cache line, e.g., structures of pointers and 1-byte integers. This suggests that if we apply B+Δ with multiple bases we can improve compressibility for some of these applications. Figure 5 shows a 32-byte cache line from *mcf* that is not

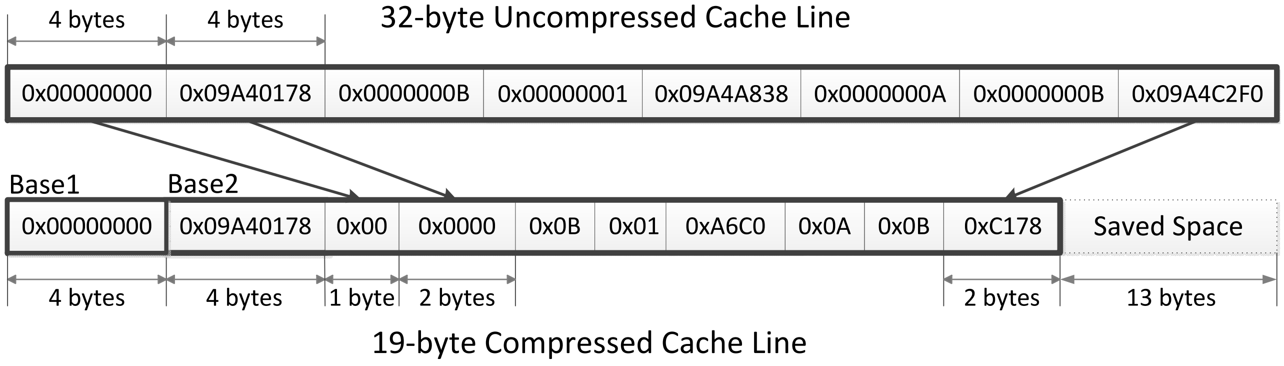


Figure 5: Example cache line from *mcf* compressed by B+Δ (two bases).

compressible with a single base using B+Δ, because there is no single base value that effectively compresses this cache line. At the same time, it is clear that if we use two bases, this cache line can be easily compressed using a similar compression technique as in the B+Δ algorithm with one base. As a result, the entire cache line data can be represented using 19 bytes: 8 bytes for two bases (`0x00000000` and `0x09A40178`), 5 bytes for five 1-byte narrow values from the first base, and 6 bytes for three 2-byte offsets from the second base. This effectively saves 13 bytes of the 32-byte line.

As we can see, multiple bases can help compress more cache lines, but, unfortunately, more bases can increase overhead (due to storage of the bases), and, hence, decrease effective compression ratio that can be achieved with one base. So, it is natural to ask *how many bases are optimal for B+Δ compression*?

In order to answer this question, we conduct an experiment where we evaluate the effective compression ratio with different numbers of bases (selected suboptimally using a greedy algorithm). Figure 6 shows the results of this experiment. The “0” base bar corresponds to a compression using simple patterns (zero and repeated values). These patterns are simple to compress and common enough, so we can handle them easily and efficiently without using

B+Δ,[5] e.g., a cache line of only zeros compressed to just one byte for any number of bases.

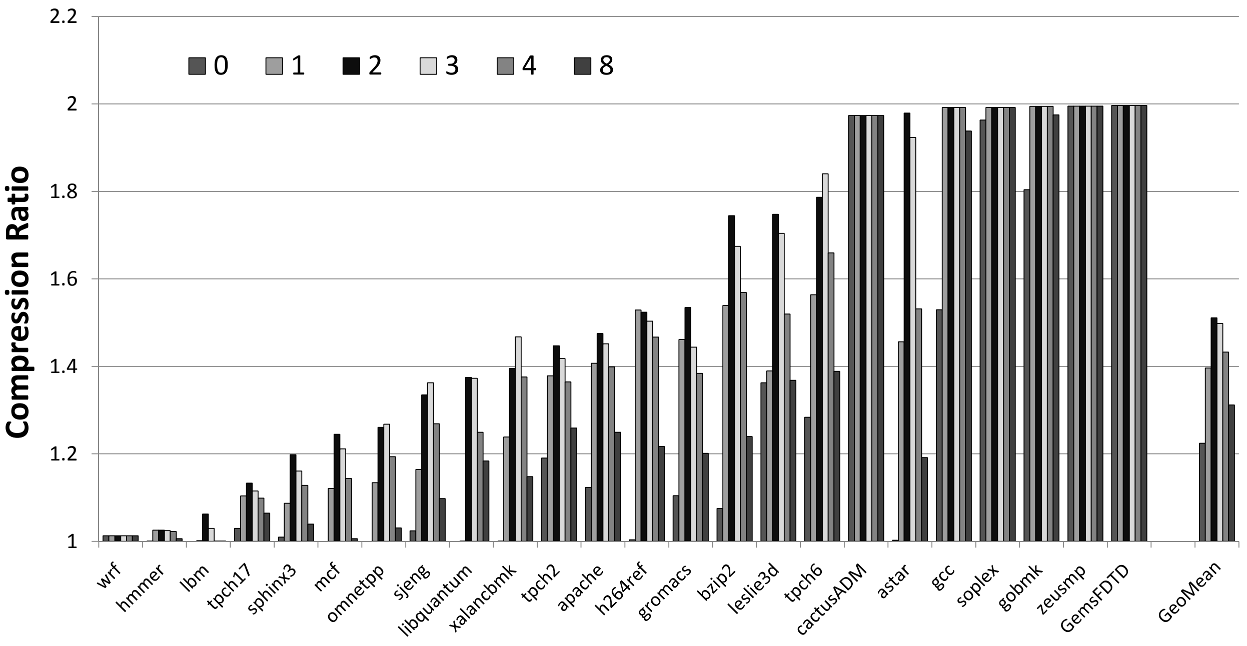


Figure 6: Effective compression ratio using different number of bases.

Results in Figure 6 show that the empirically optimal number of bases in terms of effective compression ratio is 2, with some benchmarks having optimums also at one or three bases. The key conclusion is that the B+Δ with two bases significantly outperforms one base (compression ratio 1.51 vs. 1.40 on average), suggesting that it is worth considering for implementation. Note that having more than two bases does not provide additional improvement in compression ratio for these workloads, because the overhead of storing more bases is higher than the benefit of compressing more cache lines.

Unfortunately, B+Δ with two bases has a serious drawback - the necessity to find a second base. The search for a second arbitrary base value (even a sub-optimal one) can add significant complexity to the compression hardware. This opens the question of how to find two base values efficiently. We next propose a mechanism that can get the benefit of compression with two bases with minimal complexity.

4.2 BΔI: Refining B+Δ with Two Bases and Minimal Complexity

Results from Section 4.1 suggest that the optimal (on average) number of bases to use is two, but having an additional base has major shortcomings as we described. We observe that setting the second base to zero gains most of the benefit of having an arbitrary second base value. Why is this the case?

Most of the time when the data of different types are mixed in the same cache line it comes from an aggregate data type e.g., struct. In many cases this leads to the mixing of wide values with low dynamic range (e.g., pointers) with narrow values (e.g., small integers). A first arbitrary base helps to compress wide values with low dynamic range using base+delta encoding, while a second zero base is efficient enough to compress narrow values separately from wide values. Based on this observation, we refine the idea of B+Δ by adding an additional implicit base that is always set to zero. We call this refinement **Base-Delta-Immediate** or **BΔI** compression.

There is a tradeoff involved in using BΔI instead of B+Δ with two arbitrary bases. BΔI uses an implicit zero base as the second base, and, hence, it has less storage overhead, which means potentially higher average compression ratio for cache lines that are compressible with both techniques. B+Δ with two general bases uses more storage to store an arbitrary second base value, but can compress more cache lines because the base can be any value. As such, compression ratio can be better with either mechanism, depending on the compressibility pattern of cache lines. In

[5] It is more fair, because otherwise, compression with multiple bases will have very low compression ratio for common simple patterns.

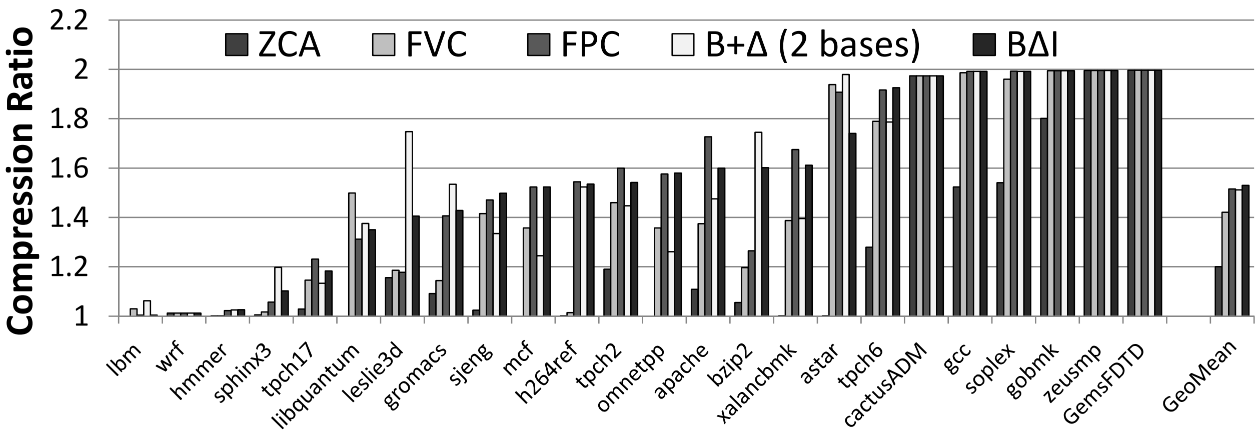


Figure 7: Compression Ratio: ZCA vs. FVC vs. FPC vs. B+Δ (two arbitrary bases) vs. BΔI.

order to evaluate this tradeoff, we compare the effective compression ratio of BΔI, B+Δ with two arbitrary bases, and three prior approaches: ZCA [7] (zero-based compression), FVC [33] and FPC [2] (Figure 7).[6]

Although there are cases where B+Δ with two bases is better, e.g., leslie3d and bzip2, on average, BΔI performs slightly better than B+Δ in terms of compression ratio (1.53 vs. 1.51). We can also see that both mechanisms are better than previously proposed FVC mechanism, and competitive in terms of compression ratio with a more complex FPC compression mechanism. Taking into an account that B+Δ with two bases is also a more complex mechanism than BΔI, we conclude that our cache compression design should be based on the refined idea of BΔI.

Now we will describe the design and operation of a cache that implements our BΔI compression algorithm.

5 BΔI: Design and Operation

5.1 Design

Compression and Decompression. We describe the detailed design of the corresponding compression and decompression logic.[7] The compression logic consists of eight distinct compressor units: six units for different base sizes (8, 4 and 2 bytes) and Δ sizes (4, 2 and 1 bytes), and two simple units for zero and repeated value compression (Figure 8). Every compressor unit takes a cache line as an input, and outputs whether or not this cache line can be compressed with this unit. If it can be, the unit outputs the compressed cache line. The compressor selection logic is used to determine a set of compressor units that can compress this cache line. If multiple compression options are available for the cache line (e.g., 8-byte base 1-byte Δ and zero compression) the selection logic chooses the one with the smallest compressed cache line size. Note that all potential compressed sizes are known statically and described in Table 2. All compressor units can operate in parallel.

Name	Base	Δ	Size	Encoding		Name	Base	Δ	Size	Encoding
Zeros	1	0	1/1	0000		Repeated Values	8	0	8/8	0001
Base8-Δ1	8	1	12/16	0010		Base8-Δ2	8	2	16/24	0011
Base8-Δ4	8	4	24/40	0100		Base4-Δ1	4	1	12/20	0101
Base4-Δ2	4	2	20/36	0110		Base2-Δ1	2	1	18/34	0111
No Compression	N/A	N/A	32/64	1111						

Table 2: BΔI encoding. All sizes are in bytes. Compressed sizes are given for 32-/64-byte cache lines.

[6] All mechanisms are covered in detail in Section 6. The comparison results are provided here to give the intuition about BΔI's relative compression ability.

[7] For simplicity we present the compression and decompression logic for B+Δ. Compression for BΔI requires one more step, where elements are checked to be compressed with zero base; decompression logic only requires additional selector logic to decide which base should be used in the addition. We keep the size of Δ (1, 2, or 4 bytes) the same for both bases.

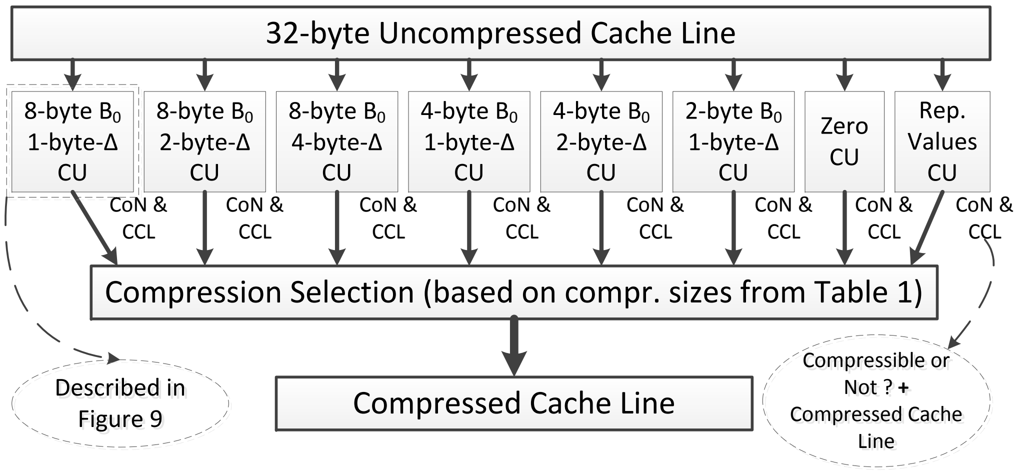


Figure 8: Compressor design. CU - compressor unit.

Figure 9 describes the organization of the 8-byte-base 1-byte-Δ compressor unit for a 32-byte cache line. The compressor "views" this cache line as a set of four 8-byte elements (V_0, V_1, V_2, V_3), and in the first step, computes the difference between the base element and all other elements. Recall that the base (B_0) is set to the first value (V_0), as we describe in Section 3. The resulting difference values ($\Delta_0, \Delta_1, \Delta_2, \Delta_3$) are then checked to see whether their first 7 bytes are all zeros or ones (1-byte sign extension check). If so, the resulting cache line can be stored as the base value B_0 and the set of differences $\Delta_0, \Delta_1, \Delta_2, \Delta_3$, where each Δ_i requires only 1 byte. The compressed cache line size in this case is 12 bytes instead of the original 32 bytes. If the 1-byte sign extension check returns false (i.e., at least one Δ_i cannot be represented using 1 byte), then the compressor unit cannot compress this cache line. The organization of all other compressor units is similar. This compression design can be potentially optimized, especially if hardware complexity is more critical than latency, e.g., all 8-byte-base value compression units can be united into one to avoid partial logic duplication.

Figure 10 shows the latency-critical decompression logic. Its organization is simple, and for a compressed cache line that consists of a base value B_0 and a set of differences $\Delta_0, \Delta_1, \Delta_2$, Δ_3, only additions between the bases and differences are performed to get the uncompressed cache line. Such decompression will take as long as the latency of an adder, and allows the BΔI cache to perform decompression very quickly.

BΔI Cache Organization. In order to obtain the benefits of compression, the conventional cache design requires certain changes. Cache compression potentially allows more cache lines to be stored in the same data storage than a conventional uncompressed cache. But, in order to access these additional compressed cache lines, we need a way to address them. One way to achieve this is to have more tags [2], e.g., twice as many,[8] than the number we have in a conventional cache of the same size and associativity. We can then use these additional tags as pointers to more data elements in the corresponding data storage.

Figure 11 shows the required changes in the cache design. The conventional 2-way cache with 32-byte cache lines (shown on the top) has a tag store with two tags per set, and a data store with two 32-byte cache lines per set. Every tag directly maps to the corresponding piece of the data storage. In the BΔI design (at the bottom), we have twice as many tags (four in this example), and every tag also has 4 additional bits to represent whether or not the line is compressed, and if it is, what compression type is used (see "Encoding" in Table 2). The data storage remains the same in size as before (2*32 = 64 bytes), but it is separated into smaller fixed-size segments (e.g., 8 bytes in size in Figure 11). Every tag stores the starting segment (e.g., Tag_2 stores segment S_2) and the encoding for the cache block. By knowing the encoding we can easily know the number of segments used by the cache block. This organization potentially allows storing twice more cache lines in the same data storage, because the number of tags in a set is doubled. It requires modest increase in the tag store size[9] (similar to some other designs [3, 10, 22]) with only 1-2 cycle access latency increase depending on the cache size (based on data from CACTI 5.3 [29]).

[8]We describe an implementation with the number of tags doubled and evaluate sensitivity to the number of tags in Section 8.

[9]According to CACTI 5.3 [29] the area increase is only 1.1% of the total area occupied by a 2MB 16-way L2, in contrast to the 2.37 times increase if we double both the size of the data storage and associativity.

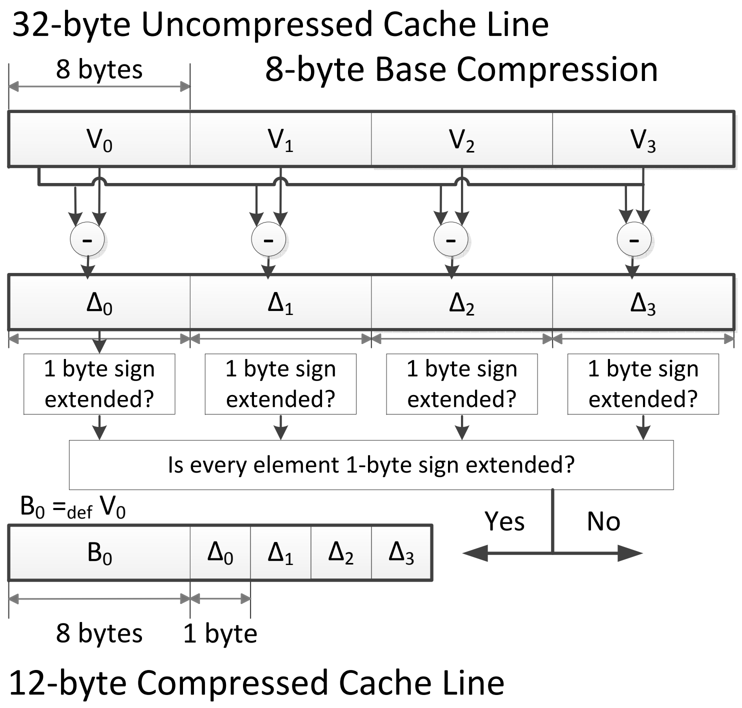

32-byte Uncompressed Cache Line
8 bytes
8-byte Base Compression
V0
V1
V2
V3
-
Δ0
Δ1
Δ2
Δ3
1 byte sign extended?
Is every element 1-byte sign extended?
B0 =def V0
Yes
No
B0
8 bytes
1 byte
12-byte Compressed Cache Line

Figure 9: Compressor unit design.

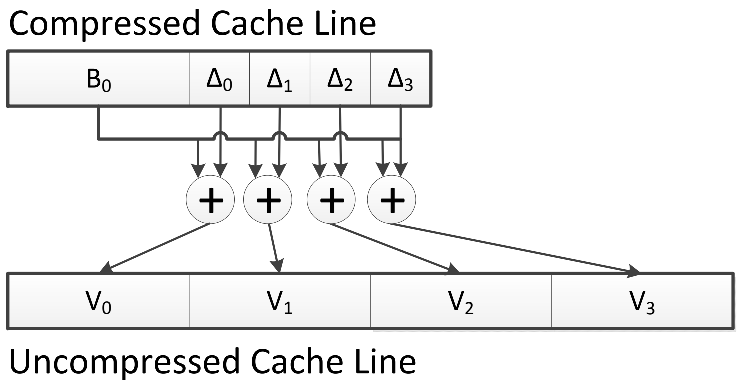


Figure 10: Decompressor design.

We also need a way to evict cache lines in this new cache design. We propose to use a slightly modified version of the LRU policy (although more effective replacement policies that can take cache line sizes into account are possible). When there is a need to store a new cache line in the set and there are not enough segments left to store it, we start evicting the least recently used cache lines until we have enough space to add a new cache line. This can result in multiple evictions per cache line insertion. However, this is acceptable, since the eviction is not on the critical path of execution.

5.2 Operation

We propose using our BΔI compression design at higher cache levels than L1 (e.g., L2 and L3). It is possible to compress data in the L1 cache [33], and memory [32, 1, 8]; however, this might lead to low performance, because an L1 cache hit is on the latency-sensitive critical path of execution, we do not want to increase L1 latency due to decompression. Let us describe how a BΔI cache fits into a system with L1, L2 and memory, assuming the L2 cache

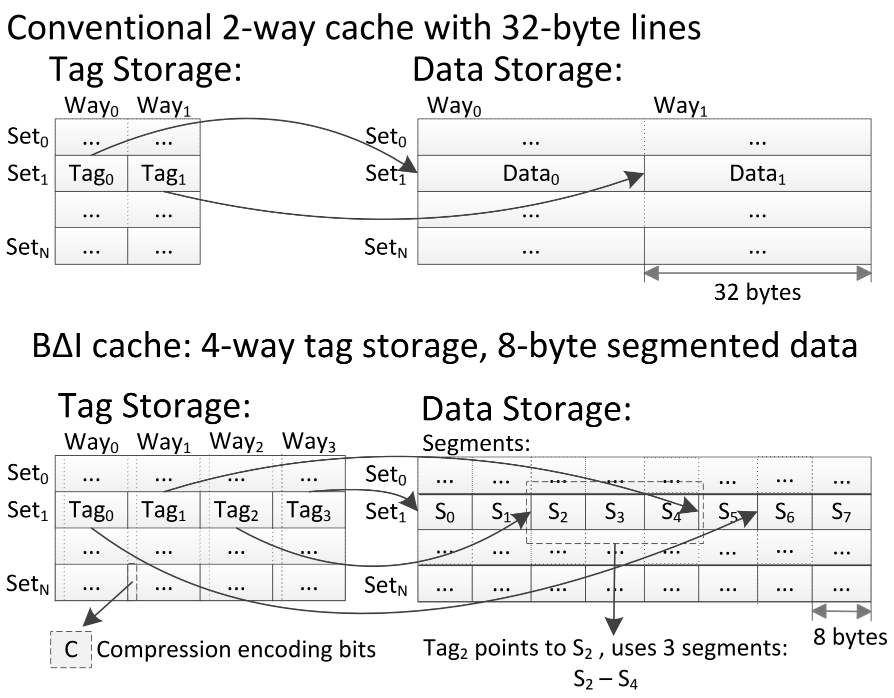


Figure 11: BΔI vs. conventional cache organization. Number of tags is doubled, compression encoding bits are added to every tag, data storage is the same in size, but partitioned into segments.

is compressed with BΔI. The only changes are to the L2 cache. Compression and decompression happens at the L2 cache. On an L2 hit the corresponding cache line can be in either compressed form (then it requires decompression) or decompressed form (then it can be simply returned to the L1 cache). On an L2 miss, we need to bring the cache line from memory, which requires compression to store it in the L2. When a line is written back from L1 to L2, it needs to be compressed first. When a line is written back from L2 to memory it needs to be decompressed.

Note that decompression latency is on the critical path for L2 hits. Hence any compressed cache design should have low decompression latency. As we have previously shown, BΔI satisfies this requirement.

6 Related Work

Multiple previous works investigated the possibility of using compression for on-chip caches [34, 2, 7, 14, 10, 6] and/or memory [32, 1, 8]. All proposed designs have different tradeoffs between compression ratio, decompression/compression latency and hardware complexity. The spectrum of proposed algorithms ranges from general-purpose compression schemes e.g., the Lempel-Ziv algorithm [35], to specific pattern-based schemes, e.g., zero values [7, 14] and frequent values [33].

The fundamental difference between BΔI and all previous cache compression mechanisms is that instead of searching for patterns at a word-granularity, which typically serializes decompression, BΔI aims to build a compression/decompression mechanism for a whole cache line that works in a *parallel* manner for all values. This simplifies both compression and especially decompression logic, since only simple masked vector operations (e.g., addition) are needed, and, at the same time, allows compression of cache lines that are otherwise not compressible (e.g., an array of pointers). As already summarized in Table 1, previous cache compression mechanisms have one or more of the following shortcomings: (1) their decompression latency is high mainly because of the sequential nature of the pattern-based or dictionary-based compression algorithms, (2) they have low average effective compression ratio mostly because only special patterns are covered, and, (3) they have relatively high hardware complexity and/or overhead.

6.1 Zero-based Designs

Dusser et al. [7] propose Zero-Content Augmented (ZCA) cache design where a conventional cache is augmented with a specialized cache to represent zero cache lines. Decompression and compression latencies as well as hardware complexity for the ZCA cache design are low. However, only applications that operate on a large number of zero cache lines can benefit from this design. In our experiments, only 6 out of 24 applications have enough zero data to get benefits from ZCA (Figure 7), leading to relatively small performance improvements (as we show in Section 8).

Islam et al. [14] observe that 18% of the dynamic loads actually access zero data, and propose a cache design called Zero-Value Canceling where these loads can be serviced faster. Again, this can only improve performance for the applications with substantial amount of zero data. Our proposal is more general than these designs that are based only on zero values.

6.2 Frequent Value Compression

Zhang et al. [34] observe that a majority of values read or written by memory operations come from a small set of frequently occurring values. Based on this observation, they propose a compression technique [33] that encodes frequent values present in cache lines with fewer bits. They apply this technique to a direct-mapped L1 cache wherein each entry in the cache can store either one uncompressed line or two compressed lines.

Frequent value compression (FVC) has three major drawbacks. First, since FVC can only compress frequent values, it cannot exploit other commonly found patterns, e.g., narrow values or stride patterns in application data. As a result, it does not provide a good degree of compression for most applications as shown in Section 8. Second, FVC compresses only the frequent values, while other values stay uncompressed. Decompression of such a cache line requires sequential processing of every element, significantly increasing the latency of decompression, which is undesirable. Third, the proposed mechanism requires profiling to identify the frequent values within an application. Our quantitative results in Section 8 shows that BΔI outperforms FVC due to these reasons.

6.3 Pattern-Based Compression Techniques

Alameldeen and Wood [2] propose frequent pattern compression (FPC) that exploits the observation that a majority of words fall under one of a few compressible patterns, e.g., if the upper 16 bits of a 32-bit word are all zeros or are all ones, all bytes in a 4-byte word are the same. FPC defines a set of these patterns [3] and then uses them to encode applicable words with fewer bits of data. For compressing a cache line, FPC first divides the cache line into 32-bit words and checks if each word falls under one of seven frequently occurring patterns. Each compressed cache line contains the pattern encoding for all the words within the cache line followed by the additional data required to decompress each word.

The authors propose a compressed cache design [2] based on FPC which allows the cache to store two times more compressed lines than uncompressed lines, effectively doubling the cache size when all lines are compressed. For this purpose, they maintain twice as many tag entries as there are data entries. Similar to frequent value compression, frequent pattern compression also requires serial decompression of the cache line, because every word can be compressed or decompressed. To mitigate the decompression latency of FPC, the authors use a five-cycle decompression pipeline [3]. They also propose an adaptive scheme which avoids compressing data if the decompression latency nullifies the benefits of compression.

Chen et al. [6] propose a pattern-based compression mechanism (called C-Pack) with several new features: (1) multiple cache lines can be compressed into one, (2) parallel compression of multiple words; but, no parallel decompression. Although the C-Pack design is more practical than FPC, it still has a high decompression latency (8 cycles), and its average compression ratio is lower than that of FPC.

7 Evaluation Methodology

We use an in-house, event-driven 32-bit x86 simulator whose front-end is based on Simics [18]. All configurations have either a two- or three-level cache hierarchy, with private L1D caches. Major simulation parameters are provided in Table 3. All caches uniformly use a 64B cache block size and LRU policy for replacement. All cache latencies were evaluated using CACTI [29], and provided in Table 3. We also checked that these latencies match the existing last level cache implementations from Intel and AMD, when properly scaled to the corresponding frequency.[10] For evaluations, we use benchmarks from the SPEC CPU2006 suite [26], three TPC-H queries [30], and an Apache web server (Table 4). All results are collected by running a representative portion of the benchmarks for 1 billion instructions.

Processor	1–4 cores, 4GHz, x86 in-order	L2/L3 hit (cycles)
L1-D cache	32kB, 64B cache-line, 2-way, 1 cycle	512kB - 15, 1MB - 21
L2 caches	0.5 – 16 MB, 64B cache-line, 16-way	2MB - 27, 4MB - 34
L3 caches	2 – 16 MB, 64B cache-line, 16-way	8MB - 41, 16MB - 48
BΔI	+1 cycle for 0.5 – 4MB (+2 cycle for others)	
Memory	300 cycle latency	

Table 3: Major parameters of the simulated system.

Metrics. We measure performance of our benchmarks using IPC (instruction per cycle), effective compression ratio (effective cache size increase, e.g., 1.5 for 2MB cache means effective size of 3MB), and MPKI (misses per kilo instruction). For multi-programmed workloads we used the weighted speedup performance metric: [25] ($\sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$). For bandwidth consumption we used BPKI (bytes transferred over bus per thousand instructions [27]).

We conducted a study to see applications' performance sensitivity to the increased L2 cache size (from 512kB to 16 MB). Our results show that there are benchmarks that are almost insensitive (IPC improvement less than 5% with 32x increase in cache size) to the size of the L2 cache: dealII, povray, calculix, gamess, namd, milc, and perlbench. This typically means that their working sets mostly fit into the L1D cache, leaving almost no potential for any L2/L3/memory optimization. Therefore we do not present data for these applications, although we verified that our mechanism does not affect their performance.

Parameters of Evaluated Schemes. For FPC, we used a decompression latency of 5 cycles, and a segment size of 1 byte (as for BΔI) to get the highest compression ratio as described in [3]. For FVC, we used static profiling for 100k instructions to find the 7 most frequent values as described in [33], and a decompression latency of 5 cycle. For ZCA and BΔI, we used a decompression latency of 1 cycle.

8 Results & Analysis

8.1 Single-core Results

Figure 12a shows the performance improvement of our proposed BΔI design over the baseline cache design for various cache sizes, normalized to the performance of a 512KB baseline design. The results are averaged across all benchmarks. Figure 12b plot the corresponding results for MPKI also normalized to a 512KB baseline design. Several observations are in-order. First, the BΔI cache significantly outperforms the baseline cache for all cache sizes. By storing cache lines in compressed form, the BΔI cache is able to effectively store more cache lines and thereby significantly reduce the cache miss rate (as shown in Figure 12b). Second, in most cases, BΔI achieves the performance improvement of doubling the cache size. In fact, the 2MB BΔI cache performs better than the 4MB baseline cache. This is because, BΔI increases the effective cache size *without* significantly increasing the access latency of the data storage. Third, the performance improvement due to the BΔI cache decreases with increasing cache size. This is expected because, as cache size increases, the working set of most of our benchmarks start fitting into the cache. Therefore, storing the cache lines in compressed format has little benefit. Based on our results, we conclude that BΔI is an effective compression mechanism to significantly improve single-core performance, and can provide the benefits of doubling the cache size without incurring the area and latency penalty associated with a cache of twice the size.

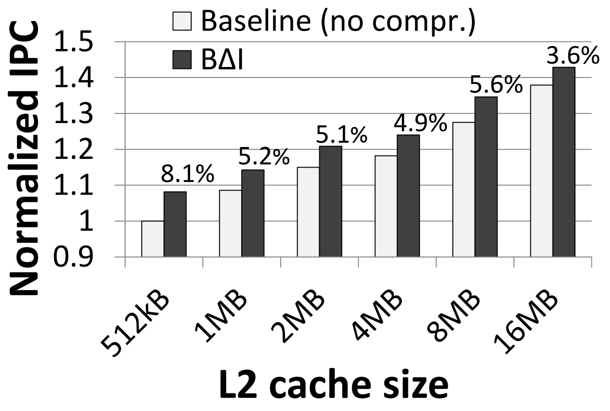
8.2 Multi-core Results

When the working set of an application fits into the cache, the application will not benefit significantly from compression even though its data might have high redundancy. However, when such an application is running concurrently with another cache-sensitive application in a multi-core system, storing its cache lines in compressed format will create

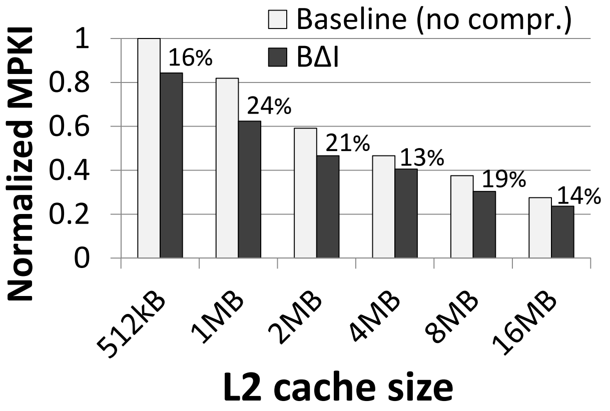
[10]Intel Xeon X5570 (Nehalem) 2.993GHz, L3 8MB - 35 cycles [19]; AMD Opteron 2.8GHz, L2 1MB - 13 cycles [5].

Cat.	**Name**	**C. Ratio**	**Sens.**	**Name**	**C. Ratio**	**Sens.**	**Name**	**C. Ratio**	**Sens.**
LCLS	gromacs	1.43 / L	L	hmmer	1.03 / L	L	lbm	1.00 / L	L
	libquantum	1.25 / L	L	leslie3d	1.41 / L	L	sphinx	1.10 / L	L
	tpch17	1.18 / L	L	wrf	1.01 / L	L			
HCLS	apache	1.60 / H	L	zeusmp	1.99 / H	L	gcc	1.99 / H	L
	GemsFDTD	1.99 / H	L	gobmk	1.99 / H	L	sjeng	1.50 / H	L
	tpch2	1.54 / H	L	tpch6	1.93 / H	L	cactusADM	1.97 / H	L
HCHS	astar	1.74 / H	H	bzip2	1.60 / H	H	mcf	1.52 / H	H
	xalancbmk	1.61 / H	H	omnetpp	1.58 / H	H	soplex	1.99 / H	H
	h264ref	1.52 / H	H						

Table 4: Benchmarks characteristics and categories: **C. Ratio** (effective compression ratio for 2MB BΔI L2) and **Sens.** (cache size sensitivity). Sensitivity is the ratio of improvement in performance by going from 512kB to 2MB L2 (L - low ($\leq$ 1.10) , H - high ($>$ 1.10)). For compression ratio: L - low ($\leq$ 1.50), H - high ($>$ 1.50). **Cat.** means category based on compression ratio and sensitivity.



(a) IPC



(b) MPKI

Figure 12: Comparison for different cache sizes. Percentages show improvement over the baseline cache of the same size.

additional cache space for storing the data of the cache-sensitive application, leading to significant overall performance improvement.

To study this effect, we classify our benchmarks into four categories based on their compressibility using BΔI (low (LC) or high (HC)) and cache sensitivity (low (LS) or high (HS)). Table 4 shows the sensitivity and compressibility of different benchmarks along with the criteria used for classification. None of the benchmarks used in our evaluation fall into the low-compressibility high-sensitivity (LCHS) category. Therefore, we generate six different categories of 2-core workloads (20 in each category) by randomly choosing benchmarks with different characteristics (LCLS, HCLS and HCHS).

Figure 13 shows the performance improvement provided by four different compression schemes, namely, ZCA, FVC, FPC, and BΔI, over a 2MB baseline cache design for different workload categories. We draw three major conclusions.

First, BΔI outperforms all prior approaches for all workload categories. Overall, BΔI improves system performance by 9.5% compared to the baseline cache design.

Second, as we mentioned in the beginning of this section, even though an application with highly compressible data may not itself benefit from compression (HCLS), it can enable opportunities for significant performance improvement for the co-running application. This effect is clearly visible in the figure. When at least one benchmark is sensitive to cache space, the performance improvement of BΔI increases with increasing compressibility of the co-running benchmark (as observed by looking at High Sensitivity label). In fact, BΔI provides the highest improvement (18%) when *both* benchmarks in a workload are highly compressible and highly sensitive to cache space (HCHS-HCHS).

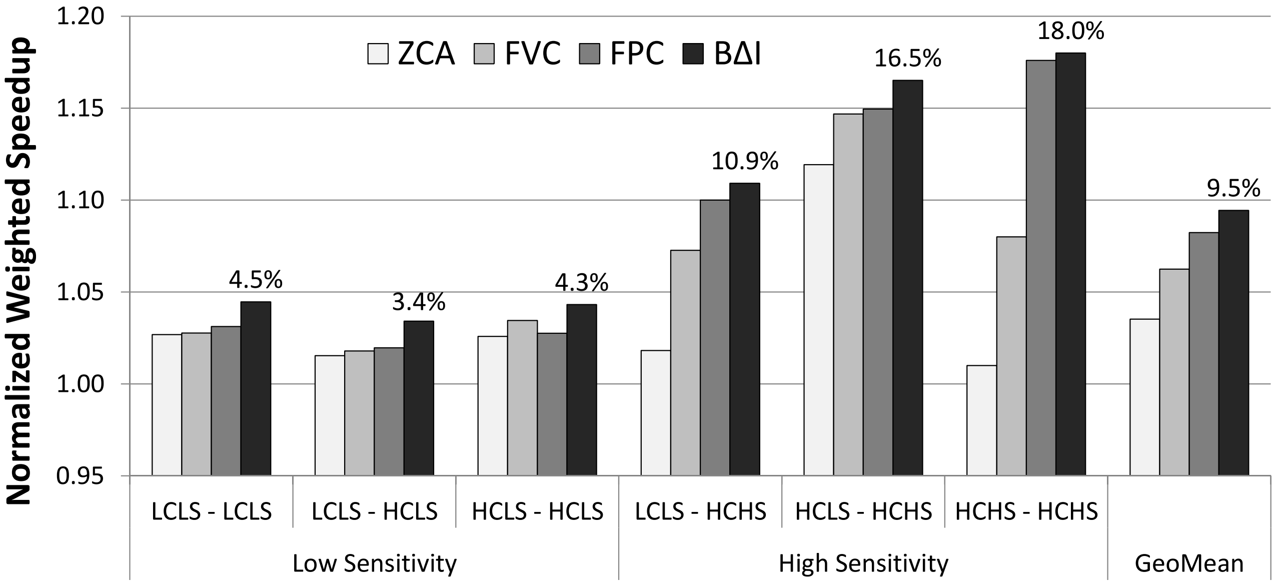


Figure 13: Normalized weighted speedup for 2MB L2 cache, 2-cores. Percentages show improvement over the baseline cache of the same size.

As the figure shows, the performance improvement is not as significant when neither benchmark is sensitive to cache space irrespective of their compressibility.

Third, although FPC provides a degree of compression similar to BΔI for most benchmarks (as we showed in Section 4.2, Figure 7) its performance improvement is lower than BΔI for all workload categories. This is because FPC has a more complex decompression algorithm with higher decompression latency compared to BΔI. On the other hand, for high sensitivity workloads, neither ZCA nor FVC is as competitive as FPC or BΔI in the HCLS-HCHS category. This is because both ZCA and FVC have a significantly lower degree of compression compared to BΔI. However, a number of benchmarks in the HCLS category (*cactusADM*, *gcc*, *gobmk*, *zeusmp* and *GemsFDTD*) have high occurrences of zero in their data. Therefore, ZCA and FVC are able to compress most of the cache lines of these benchmarks, thereby creating additional space for the co-running HCHS application.

We conducted a similar experiment with 100 4-core workloads with different compressibility and sensitivity characteristics. We observed trends similar to the 2-core results presented above. On average, BΔI improves performance by 11.2% for the 4-core workloads and it outperforms all previous techniques. We conclude that BΔI, with its high compressibility and low decompression latency, outperforms other state-of-the-art compression techniques for both 2-core and 4-core workloads, making it a better candidate for adoption in modern multi-core processors.

We summarize BΔI performance improvement against the baseline 2MB L2 cache (without compression) and other mechanisms in Table 5.

Cores	No Compression	ZCA	FVC	FPC
1	5.1%	4.1%	2.1%	1.0%
2	9.5%	5.7%	3.1%	1.2%
4	11.2%	5.6%	3.2%	1.3%

Table 5: Average performance improvement of BΔI over other mechanisms.

8.3 Effect on Cache Capacity

Our proposed BΔI cache design aims to provide the benefits of increasing the cache size while not incurring the increased latency of a larger data storage. To decouple the benefits of compression using BΔI from the benefits of reduced latency compared to a larger cache, we perform the following study. We compare the performance of the baseline cache design and the BΔI cache design by progressively doubling the cache size by doubling the cache associativity. We fix the latency of accessing all caches.

Figure 14 shows the results of this experiment. With the same access latency for all caches, we expect the performance of the BΔI cache (with twice the number of tags as the baseline) to be strictly between the baseline cache of the same size (lower limit) and the baseline cache of double the size (upper limit, also reflected in our results). However, with its high degree of compression, the BΔI cache's performance comes close to the performance of the twice as-large baseline cache design for most benchmarks (e.g., *h264ref* and *zeusmp*). On average, the performance improvement due to the BΔI cache is within 1.3% – 2.3% of the improvement provided by a twice as-large baseline cache. We conclude that our BΔI implementation (with twice the number of tags as the baseline) achieves performance improvement close to its upper bound potential performance of a cache twice the size of the baseline.

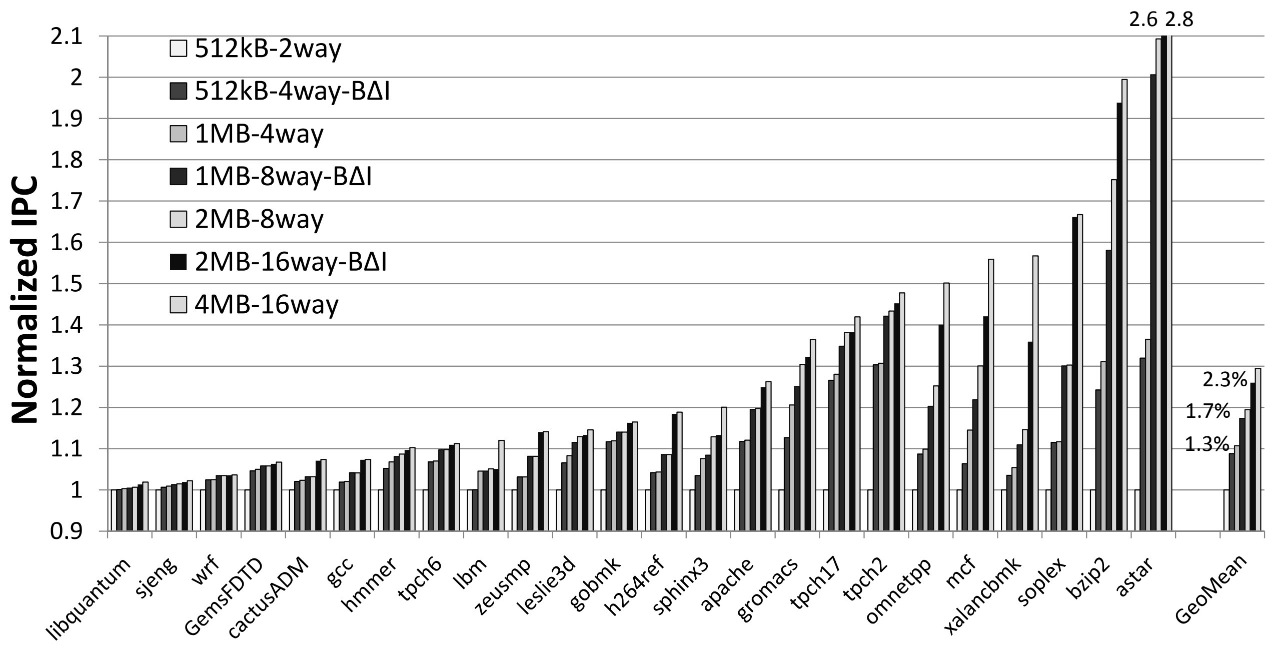


Figure 14: IPC comparison of BΔI against lower and upper limits in performance (from 512kB 2-way - 4MB 16-way L2 cache).

For an application with highly compressible data, the compression ratio of the BΔI cache is limited by the number of additional tags used in its design. Figure 15 shows the effect of varying the number of tags (from 2× to 64× the number of tags in the baseline cache) on compression ratio for a 2MB cache. As the figure shows, for most benchmarks, except *soplex*, *cactusADM*, *zeusmp*, and *GemsFDTD*, having more than twice as many tags as the baseline cache does not improve the compression ratio. The improved compression ratio for the four benchmarks are primarily due to the large number of zeros and repeated values present in their data. At the same time, having more tags does not benefit a majority of the benchmarks and also incurs higher storage cost and access latency. Therefore, we conclude that these improvements likely do not justify the use of more than 2× tags in the BΔI cache design compared to the baseline cache.

8.4 Effect on Bandwidth

In a system with a 3-level cache hierarchy, where both the L2 and the L3 caches store cache lines in compressed format, there is an opportunity to compress the traffic between the two caches. This has two benefits: 1) it can lead to reduced latency of communication between the two caches, and hence, improved system performance, and 2) it can lower the dynamic power consumption of the processor as it communicates less data between the two caches [17]. Figure 16 shows the reduction in L2-L3 bandwidth (in terms of bytes per kilo instruction) due to BΔI compression. We

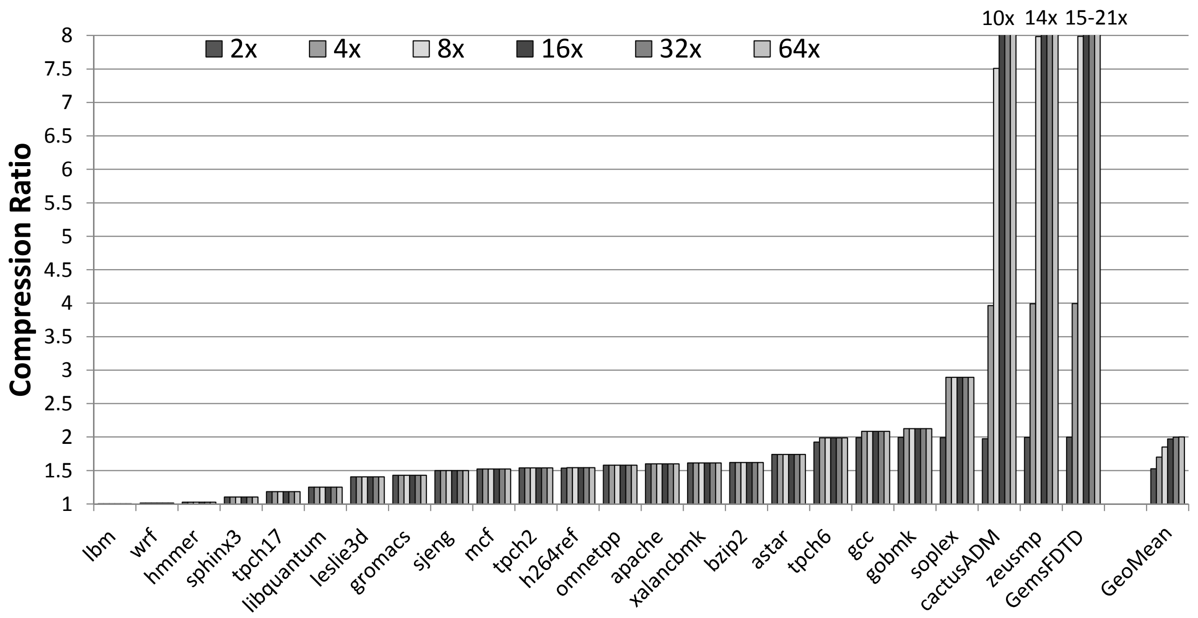


Figure 15: Effective compression ratio with increased numbers of tags.

observe that the potential bandwidth reduction with BΔI is as high as 53X (for *GemsFDTD*), and 2.31X on average. We conclude that BΔI can not only increase the effective cache size, but it can also significantly decrease the on-chip traffic.

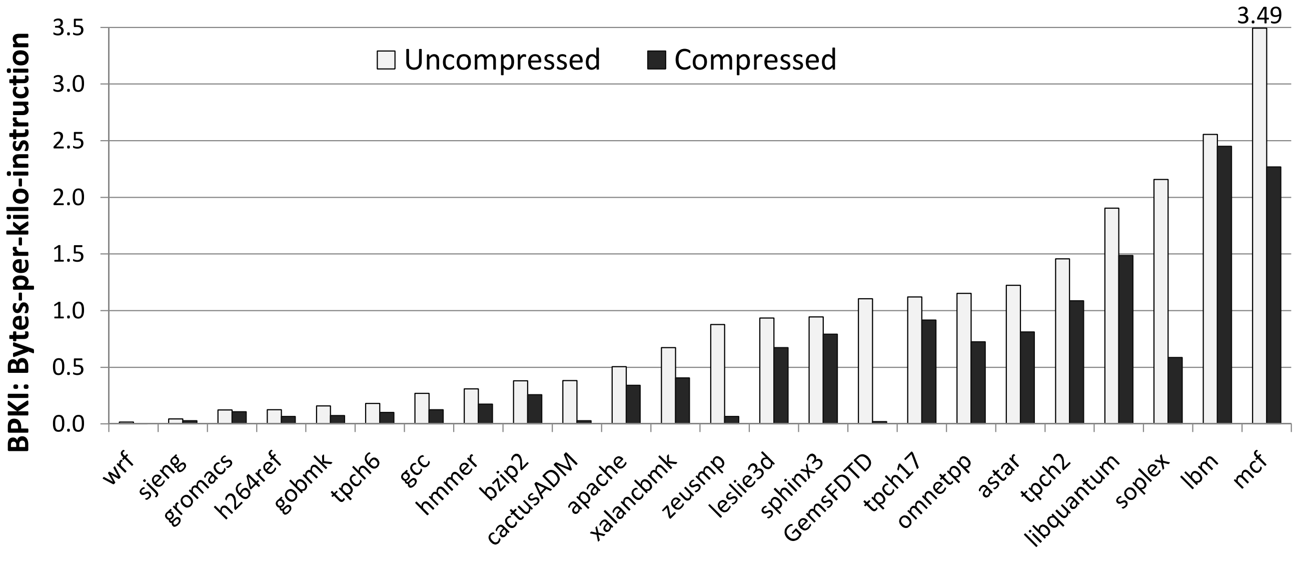


Figure 16: Effect of compression on BPKI between L2 (256kB) and L3 (8MB).

8.5 Detailed Comparison with Prior Work

To compare the performance of BΔI against state-of-the-art cache compression techniques, we conducted a set of studies and evaluated IPC, MPKI and effective compression ratio (Figure 7) for single core workloads, and weighted speedup (Figure 13) for two- and four-core workloads.

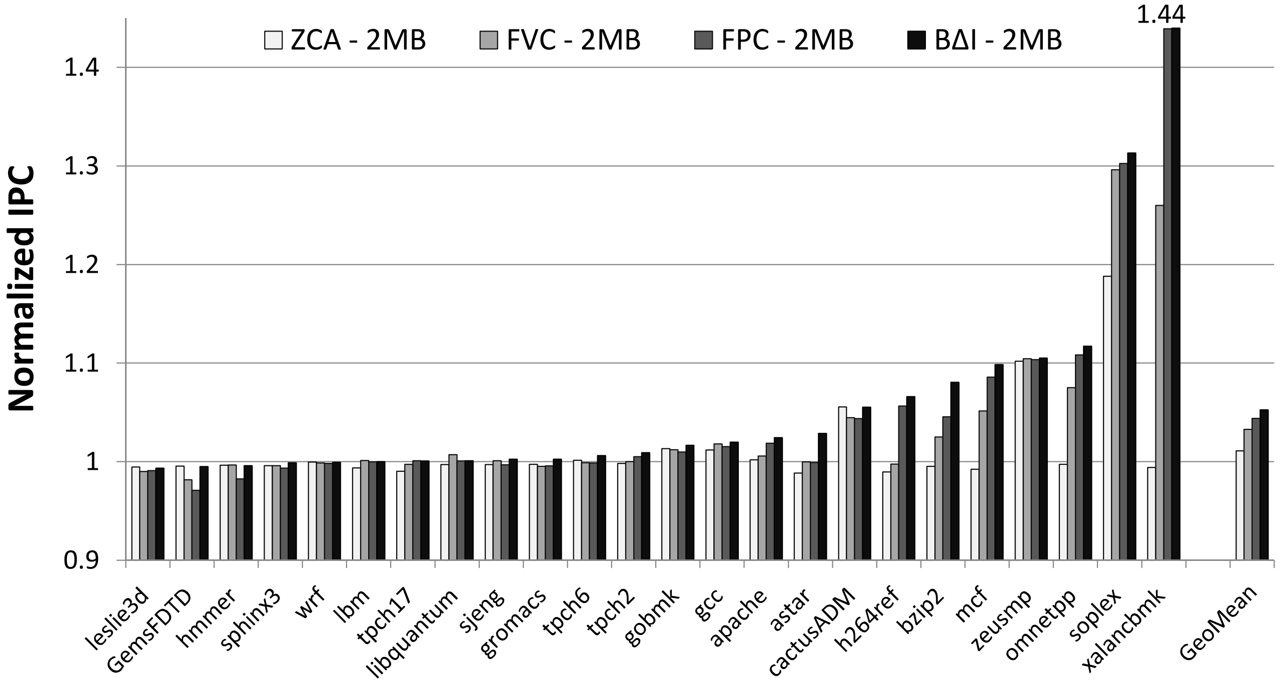


Figure 17: ZCA vs. FVC vs. FPC vs. BΔI on 2MB L2 cache.

Figure 17 shows the improvement in IPC using different compression mechanisms over a 2MB baseline cache. As the figure shows, BΔI outperforms all prior approaches for most of the benchmarks. For benchmarks that do not benefit from compression (e.g, *leslie3d*, *GemsFDTD*, and *hmmer*), all compression schemes degrade performance compared to the baseline. However, BΔI has the lowest performance degradation with its low 1-cycle decompression latency, and never degrades performance by more than 1%. On the other hand, FVC and FPC degrade performance by as much as 3.1% due to their relatively high 5-cycle decompression latency. We also observe that BΔI and FPC considerably reduce MPKI compared to ZCA and FVC, especially for benchmarks with more complex data patterns like *h264ref*, *bzip2*, *xalancbmk*, *hmmer* and *mcf* (not shown due to space limitation).

Based on our results, we conclude that BΔI, with its low decompression latency and high degree of compression, provides the best performance compared to all examined compression mechanisms.

9 Conclusions

This paper presents BΔI, a new and simple, yet efficient hardware cache compression technique that provides high effective cache capacity increase and system performance improvement compared to three state-of-the-art cache compression techniques. BΔI achieves these benefits by exploiting the low dynamic range of in-cache data and representing cache lines in the form of two base values (with one implicit base equal to zero) and an array of differences. We provide insights into why BΔI compression is effective via examples of existing in-cache data patterns from real programs. BΔI's key advantage over previously proposed cache compression mechanisms is its ability to have low decompression latency (due to parallel decompression) while still having a high average compression ratio.

We describe the design and operation of a cache that can utilize BΔI compression with relatively modest hardware overhead. Our extensive evaluations across a variety of workloads and system configurations show that BΔI compression in an L2 cache can improve system performance for both single-core (8.1%) and multi-core workloads (9.5% / 11.2% for two/four cores), outperforming three state-of-the-art cache compression mechanisms. In many workloads, the performance benefit of using BΔI compression is close to the performance benefit of doubling the L2/L3 cache size. We conclude that BΔI is an efficient and low-latency data compression substrate for on-chip caches in both single- and multi-core systems.

References

[1] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory expansion technology (MXT): software support and performance. *IBM J. Res. Dev.*, 45:287–301, 2001.

[2] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA-31*, 2004.

[3] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Tech. Rep.*, 2004.

[4] S. Balakrishnan and G. S. Sohi. Exploiting value locality in physical register files. In *MICRO-36*, 2003.

[5] J. Chen and W. W. Iii. Multi-threading performance on commodity multi-core processors. In *Proceedings of HPCAsia*, 2007.

[6] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. In *VLSI Systems, IEEE Transactions on*, volume 18, pages 1196 –1208, Aug. 2010.

[7] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *ICS*, 2009.

[8] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *ISCA-32*, 2005.

[9] M. Farrens and A. Park. Dynamic base register caching: a technique for reducing address bus width. In *ISCA-18*, 1991.

[10] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *ISCA-27*, 2000.

[11] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA-11*, 2005.

[12] D. W. Hammerstrom and E. S. Davidson. Information content of CPU memory referencing behavior. ISCA-4, 1977.

[13] D. Huffman. A method for the construction of minimum-redundancy codes. *IRE*, 1952.

[14] M. M. Islam and P. Stenstrom. Zero-value caches: Cancelling loads that return zero. In *PACT*, 2009.

[15] M. M. Islam and P. Stenstrom. Characterization and exploitation of narrow-width loads: the narrow-width cache approach. CASES '10, 2010.

[16] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ISCA-37*, 2010.

[17] Y. Jin, K. H. Yum, and E. J. Kim. Adaptive data compression for high-performance low-power on-chip networks. In *MICRO-41*, 2008.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[19] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*, 2009.

[20] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.

[21] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *HPCA-13*, 2007.

[22] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: Demand based associativity via global replacement. ISCA-32, 2005.

[23] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO-30*, pages 248–258, 1997.

[24] A. B. Sharma, L. Golubchik, R. Govindan, and M. J. Neely. Dynamic data compression in multi-hop wireless networks. In *SIGMETRICS '09*.

[25] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.

[26] SPEC CPU2006 Benchmarks. http://www.spec.org/.

[27] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, pages 63–74, 2007.

[28] W. Sun, Y. Lu, F. Wu, and S. Li. DHTC: an effective DXTC-based HDR texture compression scheme. In *Symp. on Graphics Hardware*, GH '08.

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, 2008.

[30] Transaction Processing Performance Council. http://www.tpc.org/.

[31] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *MICRO-33*, 2000.

[32] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Tech. Conf.*, 1999.

[33] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO-33*, 2000.

[34] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *ASPLOS-9*, 2000.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.