2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00085

GBDI: Going Beyond Base-Delta-Immediate Compression with Global Bases

Alexandra Angerd*, Angelos Arelakis*, Vasilis Spiliopoulos*, Erik Sintorn†, Per Stenström*†
*ZeroPoint Technologies
†Dept. of Computer Science and Engineering, Chalmers University of Technology

Abstract—**Memory bandwidth is limiting performance for many emerging applications. While compression techniques can unlock a higher memory bandwidth, prior art offers only modestly better bandwidth. This paper contributes with a new compression method – Global Base Delta Immediate compression (GBDI) – that offers substantially higher memory bandwidth by, unlike prior art, selecting base values across memory blocks. GBDI uses a novel clustering algorithm through data analysis in the background. The presented accelerator infrastructure offers low area overhead and latency. This paper shows that GBDI offers a compression ratio of 2.3×, and yields 1.5× higher bandwidth and 1.1× higher performance compared with a baseline without compression support, on average, for SPEC2017 benchmarks requiring medium to high memory bandwidth.**

I. Introduction

With processors improving performance exponentially comes an exponential growth in the demand for memory bandwidth. While high-bandwidth memories (HBM) provide substantially higher bandwidth, they are expensive. As an alternative, lossless memory compression techniques are emerging to improve memory bandwidth.

Memory compression techniques [21] abound to maximize the compression ratio (CR[1]) while maintaining a low latency and area overhead of the compression/decompression accelerators. Recently, Bitplane Compression (BPC) [16] was proposed to improve bandwidth. BPC is built on top of Base-Delta-Immediate (BDI) compression [19]. It utilizes value similarity within a block of n values by encoding them as one base value and $n-1$ deviations (*deltas*) from the base. BDI compression efficiency relies on two hypotheses (H1 and H2): H1) intra-block values are numerically similar and H2) deltas can be encoded compactly. This paper makes the surprising insight that these hypotheses are not true, in general, especially not for floating-point values (floats), as the following example illustrates.

Figure 1 shows 32-bit values within six 64-B cache blocks from a memory snapshot of the application `parest` in SPEC2017 [8]. It shows two interpretations of the same input data: as floats (green dots, left y axis) and as signed integers (orange dots, right y axis). Floats expose numerical similarity within a block since all values are within a quite small interval (i.e. [-2.0, 1.0]). Hence, H1 seems to hold. However, BDI interprets data as integers. Consequently, values within a block span a wide range (approximately [-2e9, 1e9]), which invalidates H1. Rather, values cluster around a few base values, illustrated by the purple horizontal lines. We show that this observation extends to integers.

Numerical similarity of floats breaks down because they are represented as a sign, an exponent and a mantissa with 1, 8 and 23 bits, respectively, according to the IEEE754 standard [2]. Clearly, what is perceived as numerically similar values cannot always be encoded compactly using deltas in BDI. For example, 0.0 = 0x00000000 and 1.3 ≈ 0x3FA66666, which differ in the 30th bit position, need 31 bits. Therefore, H2 is typically invalid for floats as well.

This paper contributes with a lossless compression algorithm – Global Base Delta Immediate compression (GBDI) – that uses *global* bases, shared by all input data, instead of a single intra-block base value as in BDI. It compresses and decompresses individual blocks on-the-fly at run-time. Global bases are established through a data analysis phase, by software algorithms, in the background.Thus, GBDI belongs to the family of statistical compression techniques [6]. Apart from attempting to select inter-block base values optimally, this paper also applies a few optimizations to maximize compressibility. A first optimization is to do a per-block decision whether to use an intra-block base, as BDI does, or a global base as GBDI. A second optimization is to depart from a fixed size of the deltas. A third optimization is to store the base pointers compactly by compressing them using Huffman encoding. We show that GBDI can offer substantial performance improvements at a small area and latency overhead of the hardware accelerators.

Thesaurus [12], tailored for cache compression, also uses dynamic clustering to exploit value similarities between cache lines by storing the difference between a similar cache line and a master copy by pointers to it. It is unclear how Thesaurus can be used for memory compression as the master copy will have to be retrieved from memory resulting in extra memory accesses. In contrast, GBDI uses clustering to establish global bases off-line in the background. While intuitively any clustering method could be used, such as kmeans [17], they typically minimize the distance between the base and cluster values. We show that this approach does not maximize compression and we propose to minimize the number of required bits to encode the distance instead. We contribute with *histogram binning* that focuses on the

[1]CR is defined as the ratio of the size of uncompressed data to compressed data.

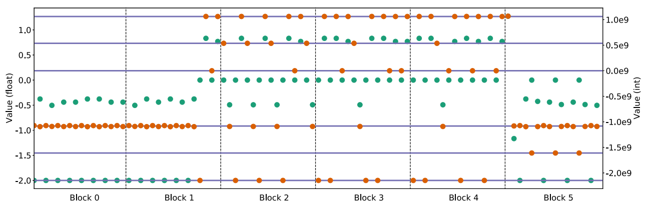


Figure 1: Example: data interpreted as floats (green) and ints (orange).

clusters with the most frequent values and selects bases to maximize compression.

The contributions of this paper are the following:

- GBDI, a generalization of Base-Delta-Immediate (BDI), selects base values across memory blocks as well as within each memory block.
- A novel clustering algorithm, referred to as histogram binning, that is more efficient than kmeans clustering and offers lower time complexity.
- A data analysis methodology that, apart from selecting global bases additionally, on a per-block basis, selects the best among BDI and GBDI and encodes bases efficiently using Huffman encoding.
- An evaluation of GBDI that shows that GBDI offers a compression ratio of 2.3×, and yields 1.5× higher bandwidth and 1.1× higher performance compared with a baseline without compression support, on average, for the applications in the SPEC2017 benchmark suite that require medium to high bandwidth.

Section II introduces the architectural framework. Then, Section III and IV introduce GBDI and its implementation. We present the experimental results describing the methodology in Section V and the results in Section VI. We end the paper by positioning our work in relation to prior art in Section VII before we conclude in Section VIII.

II. Architecture Framework

We assume a multicore System-on-Chip (SoC) according to Figure 2 with a number of processors (cores) each connected to a private cache hierarchy. All cores with their private caches (denoted L1 and L2 in Figure 2) are connected to a shared last-level cache (LLC, denoted L3 in Figure 2) and optionally to special-purpose accelerators (e.g. GPUs) and to a number of memory channels, each controlled by a memory controller (denoted MC Figure 2) connected to off-chip DRAM devices.

Compressing memory data can provide significant benefits. First, memory capacity can be expanded which offers higher performance at a lower cost by avoiding expensive page faults. Second, and the focus of this study, it can offer higher memory bandwidth by bringing multiple compressed memory blocks onto the chip in response to a single memory request. In both cases, compressed data must be located in

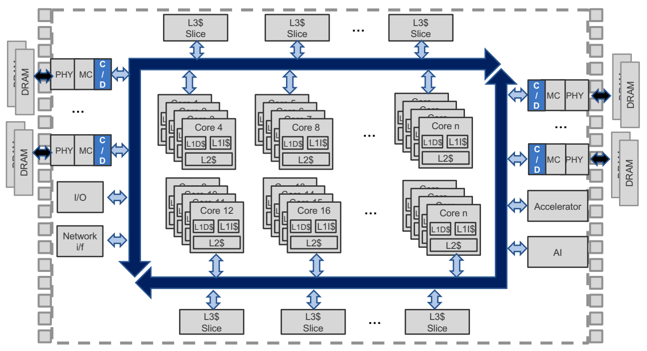


Figure 2: System architecture framework.

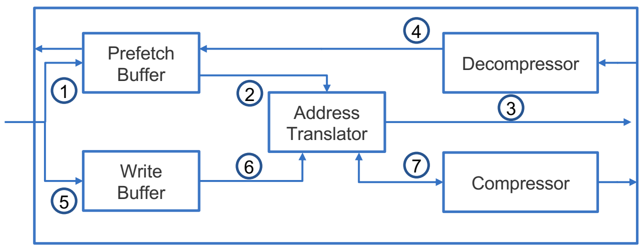


Figure 3: The architecture of the C/D IP block.

memory and that data must be compressed/decompressed at a low latency to have negligible impact on performance.

Our extended baseline system for enabling higher bandwidth, shown in Figure 2, comprises hardware functionality in terms of an IP block attached to the memory controller (denoted C/D in Figure 2) and software functionality for data analysis. We first focus on the hardware functionality.

A. Hardware Support

Figure 3 shows a block diagram of C/D in Figure 2. It comprises five building blocks to handle memory read and write requests: A Prefetch buffer (PB), an Address translator (AT), a Decompressor, a Write buffer (WB) and a Compressor. An LLC miss injects a read request into C/D. The AT maintains metadata to locate a requested block. On an AT hit, a memory request for this block is issued to the memory controller. As data is compressed, memory can respond with several compressed blocks that fit into the same physical block frame of typically 64B. These blocks are decompressed by the Decompressor and are stored uncompressed in the PB.

Let's follow the flow of an LLC miss through C/D to see how it can improve bandwidth. The request first probes the PB whether the block was prefetched due to compression (① in Figure 3). The WB is also probed in case the read block was written back recently. On a hit, the memory request will be cancelled leading to memory traffic savings. Higher spatial locality leads to more traffic saving because more data blocks will be transferred in one go. Conversely, if the requested block is not in the PB ②, the physical address will be translated to establish the location in memory where the compressed block resides and ③ a request is sent to memory. The block returned from memory may contain several compressed blocks that are decompressed ④. All the blocks contained will be inserted in PB uncompressed.

Similar to previous memory compression techniques [11], [10], the Address translator uses metadata in C/D to locate the compressed block. C/D organizes memory blocks in *sectors*, where each sector contains a fixed number of neighboring compressed memory blocks, typically four so its metadata size is typically smaller than state of the art. Note: a sector's blocks are consecutive in the memory space linked to a specific memory controller but can originate from different OS pages due to memory interleaving. C/D is agnostic to this. The location of the first memory block of each sector corresponds to its physical address and is aligned to the DRAM block boundaries; the second memory block starts where the first compressed memory block ends, etc. Hence, C/D stores only the memory-block-size metadata, referred to as *CS*. Contrary to prior art [11], [10], [18], C/D allows a compressed memory block to be of any size in the [0,64)-byte spectrum, requiring 6b metadata/block. However, if a compressed memory block is $\geq$ 64 bytes, it is stored uncompressed. Moreover, it requires 1b/block for encoding transition (refer to Section IV-B). Since the compression metadata per compressed block is 7b, the metadata of 64 consecutive blocks (i.e., 64 x 7b = 448b) is aggregated in one metadata entry so that when a piece of metadata is fetched from memory, prefetch the metadata of neighboring compressed data contained in that DRAM block is prefetched. All metadata is stored in main memory. However, to avoid adding an extra memory access to fetch metadata for every memory data read/write, C/D caches the most recently used metadata entries on chip in the Address translator. We empirically have found that the Address translator needs to cache 1-K entries of page metadata in order to achieve an average hit ratio of $\geq 95\%$.

Memory write-back requests are intercepted by C/D and complete in the Write buffer ⑤. The Write buffer is also organized to store sectors so that adjacent compressed memory blocks belonging to the same sector are compacted upon eviction from the write buffer. Efficient compression and compaction of written memory data is crucial for improving memory bandwidth. If a compressed memory block (evicted from the write buffer) exceeds in size compared to the current size in memory (according to the CS metadata), writing the compressed memory block back to memory will result in overwriting of adjacent memory block(s) leading to data corruption, known as *compression overflow*. Hence, C/D checks whether a compression overflow occurs upon a write-buffer eviction and fetches extra memory blocks in order to resolve it if the affected memory blocks are not present in the Write buffer or in the Prefetch buffer. This results in read-modify-write (RMW) transactions. We have empirically found that the extra RMW traffic to handle compression overflows is only 1-2% of the overall traffic.

Finally, for memory reads that neither hit in the Prefetch nor in the Write buffer, the C/D IP block calculates i) the location of the beginning (B) of the memory block requested by aggregating the CS of prior memory blocks and ii) the end (E) of the memory block requested by adding the CS of the requested memory block to the beginning B ⑥. Based on these, C/D establishes the amount of DRAM blocks needed to be accessed in order to fetch the requested memory block and compares this to the total DRAM traffic needed to fetch the whole compressed sector ⑦. The latter is multiplied to the prefetch buffer hit ratio so that if the hit ratio is high enough, fetching the whole compressed sector might be preferable as to fetching only the DRAM block(s) that contain(s) the requested memory block.

B. Software support

Many compression techniques in prior art, such as Base-Delta-Immediate (BDI) compression [19], are fixed to specific patterns at design time and do not need any training beforehand and can compress/decompress blocks on the fly. By contrast, SC^2, proposed by Arelakis and Stenstrom [6] uses Huffman coding, where the encoding is established by analyzing a (small) sample of data blocks in memory. Global Base-Delta-Immediate compression (GBDI), as proposed in this paper, also adopts a data analysis phase in which a set of bases are established for GBDI across all the memory blocks. This phase is triggered when the compression ratio drops below a given threshold. As this can be done in the background and is computationally light-weight, it is done in software. Moreover, as we will show in this paper, the frequency of individual values changes slowly making recompression a rare event (evaluated in Section VI-D2). When a new set of bases is established, the compressor is updated and starts compressing with this set immediately, while in the decompressor there are two base-table variants; one containing the old bases and another the new ones (Section IV-B). Recompression happens automatically for the newly written data, while for other data, C/D schedules recompression commands. This is done similarly to DRAM refresh commands, as suggested by prior art [6] so that recompression is done smoothly.

III. Global Base Delta-Immediate (GBDI) Compression

A. Baseline GBDI

GBDI aims at selecting a number of global bases, across all targeted data, that minimizes deltas when each value is encoded with a pointer to the closest global base and a delta with respect to that base. In contrast, BDI assigns a base *within* each block. Moreover, GBDI, unlike BDI, allows deltas within the same block to vary in size. The size of each delta is established by which global base the original value belongs to. To achieve this, each global base is paired with a *maximum delta*, which indicates the maximum distance which is allowed for a value using this base. For example, if a value V is closest to global base B_k with the corresponding maximum delta Δ_k, V can only be compressed using B_k if the number of bits needed to represent the delta between B_k and V is less than or equal to Δ_k.

Figure 4 shows an overview of the baseline compression algorithm, divided into five steps. First, for each value, the closest global base B_k is determined where the global bases are stored in a table structure. The table is filled by a software routine which establishes global bases and maximum deltas through data analysis in the background. The process to do that is described in Sections III-A1 and III-A2.

In Step 2, the delta for each value is calculated with respect to its closest base. Then, in Step 3, each delta is compared with the maximum delta Δ_k corresponding to the global base used to calculate the delta. If this delta is less than Δ_k, the value is compressed and marked as such in a mask. If not, it is considered an *outlier* and will not be compressed. The mask and the deltas are then conveyed to Step 4, in which the calculated deltas and the outliers are packed into two bit arrays. For each value, if the outlier mask bit is not set, the delta is packed with Δ_k bits into the *deltas* array; otherwise the full 32-bit value is packed into the *outliers* array.

Finally, in Step 5, the base pointers are packed into a bit array, optionally in a compressed format if it reduces their size. First, the number of unique bases used by any non-outlier value is identified. If this number is large, the base pointers, bp, are simply concatenated, one for each non-outlier value. If only a few unique bases (≤ 4) are used, it is more efficient to introduce another layer of indirection. In this case, the number of unique base pointers is stored first, followed by the unique base pointers. Finally, an internal base pointer mask (denoted ip in Figure 4), is concatenated for each value. The compressed base pointer format is used only if it improves compression and is indicated by a single bit in the packed data.

1) Establishing Global Base Values: Global bases are established through background data analysis in software. Kmeans clustering [17] can establish a base value that minimizes the distance to all cluster values. However, this is sub-optimal for compression. To see this, consider selecting one global base for 10 input values (five 1 and five 15). Kmeans establishes the base value as the average: $\frac{(5\cdot1+5\cdot15)}{10}=8$. The number of bits needed to encode the distances for all values is: $5\cdot\lceil\log_2(|8-1|)\rceil+5\cdot\lceil\log_2(|8-15|\rceil)=5\cdot3+5\cdot3=30$. However, choosing one of the values as a base, e.g. 1, yields $5\cdot0+5\cdot\lceil log_2(|15-1|)\rceil=5\cdot4=20$ bits. Hence, choosing one of the values as a base, instead of the average, results in a higher CR.

Finding an optimal base b among M values in a multiset $x_i \in X$ is equivalent to minimize $\sum_{i=0}^{M-1}\log_2(|b-x_i|)$. Without loss of generality, we consider instead $f(b)=\sum_{i=0}^{M-1}log_2(1+|b-x_i|)$ with the same characteristics. This function has a local minimum only in each x_i, since each term of the sum is a function with a global minimum at x_i and concave on both sides of x_i. The sum of such terms, $f(b)$, is also concave for any interval between two consecutive points, since the sum of concave functions is still concave. A concave function can only have minima at the endpoints of the interval; hence, it follows that $f(b)$ only has a local minimum at each point x_i.

Kmeans can be modified to use the global minimum of $f(b)$ as the cluster centers. However, this is expensive since $f(b)$ must be evaluated for each local minimum x_i. Instead, we modify kmeans to first search for a minimum by sampling $M_s=\sqrt{M}$ of the original points, and then search M_s points locally around the sampled minimum. Section VI-B shows that this results in a substantially higher CR than using unmodified kmeans to establish global bases.

While sampling yields a lower time complexity than evaluating $f(b)$ for each point ($\mathcal{O}(M^{\frac{3}{4}})$ vs $\mathcal{O}(M^2)$ per iteration), it is still computationally heavy. Therefore, we propose *histogram binning (HB)*, which empirically yields a good CR at a lower computational cost (time complexity and wall clock time).

In HB, we construct a histogram of all values, with a number of equal-ranged bins, N. Among the bins, we identify the B ($B<N$) bins that contain most values. Within each of the B bins, we locate the single value with the highest occurrence, and choose that as a base resulting in B base values. This heuristic ensures that a majority of values are close to one of the base values.

Selecting the number of bins (N) that maximizes CR is challenging as it affects the size of the deltas. Our approach is to sweep over different configurations of N, compress the data, and choose N with the highest CR. Also, while a large number of bases (B) increases CR, the higher amount of metadata may limit CR. In Section III-B, we present a solution which reduces the size of the global base pointers by using Huffman coding. The number of bases also affects the size and complexity of the hardware compressor and decompressor. We have found experimentally that 2048 global bases is a good choice. A sensitivity analysis of this is presented in Section VI-D.

Time complexity of HB is $\mathcal{O}(N_nM(1+\log(M)+B))$, where N_n and M are the number of bin ranges and inputs, resp. In practise, we test 15 bin ranges in steps of 2 bits ($N_n=15$). As GBDI relies on finding bases that compresses the majority of data, we find that stochastically sampling M values from the input data is sufficient, as long as $M \gg B$. Empirically, for 2048 bases, we have found that a sample size of 200K values (800KB of data) performs equivalently to evaluating the whole input data.

In our study $M=200K$, and the time complexity is

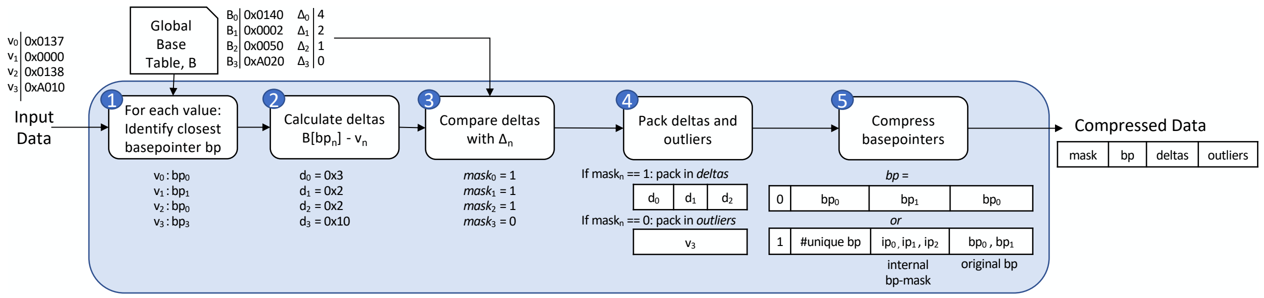


Figure 4: Overview of the main steps in the GBDI compression algorithm.

dominated by $\mathcal{O}(M\log(M))$, since $M \gg N_n$. The wall-clock time of histogram binning on this input size is less than a second.

2) Establishing Maximum Deltas: Having established the global bases, we set the maximum delta for each global base by considering all values that are closest to that base. The maximum delta is the number of bits needed to establish the distance to the furthest of these values. However, the maximum delta may become large limiting compression. To improve compression, we establish an *upper bound* on the maximum delta. If the maximum delta is greater than the upper bound, it is lowered to be equal to the upper bound. The algorithm will consider values that lie outside the upper bound as outliers. We have found that an upper bound of $16-\log_2(B)$ bits, where B is the number of global bases, works well. This allows each individual value to be compressed to $deltasize + ptrsize = (16 - \log_2(B)) + \log_2(B) = 16$ bits, which corresponds to a CR of 2x (assuming the input values are 32 bits).

B. Optimizations

To increase compression, we augment GBDI with three optimizations: intra-block sizes, a fixed delta size within each block, and compression of base pointers.

1) Exploiting Intra-Block Bases: Using an intra-block base (as in BDI) is sometimes advantageous. We identify two such cases: 1) all block values are equal and 2) values differ by a small amount as we sweep across the memory addresses. In the presence of large stretches of such memory, data contains no clear clusters which makes it challenging to identify suitable global base values, whereas the values within a single block have a very limited range. We address 1) by adding an additional step to the compression algorithm which checks whether all the values are equal and compress them as a format encoding plus the unique value. We address 2) by combining BDI with GBDI on a per block basis to select which compression method achieves the best CR.

2) Selecting a Fixed Delta Size Within Each Block: Baseline GBDI uses the maximum delta, Δ, for each base to establish the size of each delta value. When all delta values in a block are small compared to their corresponding Δ, this will be a suboptimal choice of the number of delta bits. We

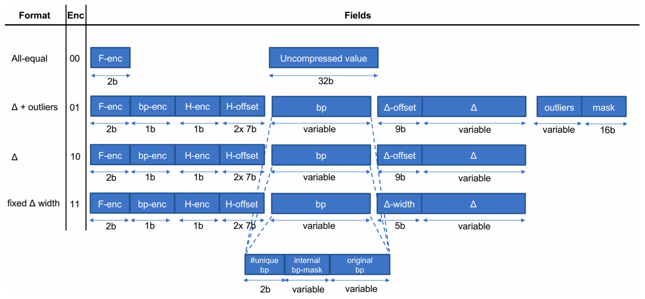


Figure 5: Compressed block formats.

choose, on a per block basis, the maximum number of delta bits needed and use that within each block.

3) Compression of base pointers: The size of the base pointer increases with the number of bases. We alleviate this cost by encoding the base pointers with Huffman coding [15]. The frequency of which each base is used is recorded while establishing the global base values (Section III-A1) and a Huffman tree is constructed. During block compression, a string of Huffman codes is stored rather than the base pointers. The compression is only applied when beneficial, and a single additional bit in the block metadata signifies whether the pointers are encoded with Huffman or not. While this encoding adds latency, it can also significantly reduce the traffic and therefore improve performance.

C. Overview of the GBDI Compression Formats

The compressed block format is identified with a 2-bit encoding prefix (see Figure 5). Prefix 0b00 identifies the *all-equal* format, which uses an intra-block base if all the values within the block are equal. Prefix 0b01 identifies the baseline GBDI format (see Section III-A), where the values are stored either as a pointer to a global base and a delta, or as a 32-bit outlier. A 16-bit mask, assuming a 64B cache block, identifies which 32-bit values are outliers.

Prefix 0b10 identifies the special case where there are no outliers, while prefix 0b11 reflects the fixed delta format (see Section III-B2).

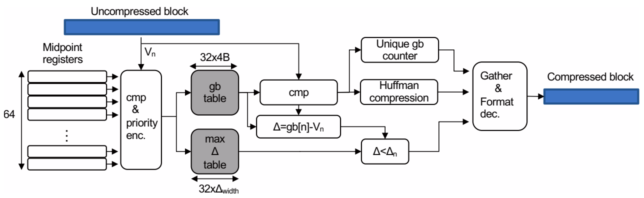


Figure 6: GBDI compression engine.

IV. Implementation

A. Compression Engine and Timing Analysis

The compression engine follows the steps of Figure 4 in Section III-A. Step 1 is the most computationally intensive of the algorithm, wherein the closest global base must be identified for each value. We do this as follows. Histogram binning sorts and organizes the global bases numerically into groups of 32 values. The midpoint between two groups (i.e., the midpoint between the last value of the previous group and the first value of the next group) is calculated. The compression engine consists of two identical compressors, each compressing half of the block. The compression engine with one compressor (for clarity) is depicted in Figure 6.

The generated midpoints are stored in registers (shared by both compressors), while the group of base values are stored in a direct-mapped SRAM, where each block is of size of 128B (=32x4B). The maximum delta value Δ_n per global base is stored in another direct-mapped SRAM. Both SRAMs are accessed with the same address index generated based on the matched midpoint register, by comparing each register value to a data value V_k of the memory block.

Next, the group of global bases returned from the SRAM are compared to the data value V_k. All global bases are compared to V_k, in parallel, using a tree of comparators to identify the closest global base. The delta Δ calculated between V_k and its closest global base is then compared with the corresponding Δ_n. Also, the closest global base is Huffman-compressed, and a counter notes the global base value and keeps track of how many unique global bases the memory block uses.

The last step gathers all global bases and delta comparisons, and compresses all formats described in Figure 5, in parallel. This follows Steps 4 and 5 in Figure 4 (Section III-A). The best format is selected and returned as the compressed block.

The compression engine is implemented in RTL. It can be clocked at 1.6GHz in a 7nm technology node. Compressing a 64-B memory block takes 12 cycles. The compression engine is pipelined and can compress memory blocks back to back.

B. Decompression Engine and Timing Analysis

The decompression engine in Figure 7 is pipelined and divided into 5 steps described below. Decompression is parallelized using 4 identical decompressors for Steps 3-5 (only one is depicted for clarity), which all are used when F-enc is not "00"; otherwise, decompression is quite simple as one 32-bit uncompressed value is replicated 16 times.

1) *Format decoding*: The header of the compressed block (F-, bp-, H-enc and H-offset) is scanned to identify the compression format and generate the control signals for the next steps for decompressing the input block.
   - *bp-enc=0*: bp contains only original base pointers, which are either N-bit wide (*H-enc=0*), e.g., N=11 bits if the number of global bases is 2048, or compressed with Huffman encoding.
   - *bp-enc=1*: bp is further broken down to unique_bp, internal_bp mask and original bp. The unique_bp field determines the number of original bp and is used in the next step. The internal_bp mask is described below. The original bp are N-bit wide or Huffman compressed depending on the H-enc.

   The bitfield of H-offset contains two offsets (offset to middle and offset to end of bp) is used to speculatively (if *H-enc=1*) calculate the start/end of the bp streams that are used in the next stage. Moreover, the outlier mask is decoded to measure the number of outliers.
2) *Stream extraction*: The base pointers are divided into 4 streams, depicted as "bp (1/4)", "bp (2/4)", etc; each stream is fed into one of the decompressors. Depending on *bp-enc* and *H-enc*, we have the following cases:
   - 00: The bp streams are generated by extracting the respective bp values of fixed width (i.e., N).
   - 01: The streams are generated using the H-offset fields, which determine the bp middle and bp end. Stream 1 starts from the left and extends to the first H-offset, while stream 2 is the reversed bit-stream of stream 1. Similarly, stream 3 starts after skipping as many bits as determined by the first H-offset and extends to the second H-offset, while stream 4 is the reversed bit-stream of stream 3. The Huffman-based streams will contain >4 compressed values, but the decompressor will stop after 4 values.
   - 10: The bp streams are skipped and the original_bp value array is stored in the register *Unique bp reg*.
   - 11: Streams are generated as in the case 01. However, each stream will contain 2 compressed values, but it will stop after decompressing one.

   In the same decompression step, the outliers are shifted out of the compressed block based on the decoded mask of the previous step.
3) *Huffman-word extraction*: Huffman-based compressed bps are passed through the Huffman decoder. Because Huffman coded symbols are of variable length, the decoder must first detect and extract a Huffman

word out of the bitstream. Then the detected word is subtracted to a pre-defined offset to determine the location that keeps the uncompressed base pointer. We use Canonical Huffman encoding and a similar decompression engine as SC^2 [6]. The pre-defined offsets are generated when the canonical Huffman coding is generated during the data analysis phase. In the same step, the Δ bitstream is also divided into 4 streams, which are organized similarly to the bp streams. The Δ streams are always of variable-length. If *F-enc=11*, the Δ width determines the width of each Δ value. For any other case, the end point is known, i.e., as depicted in Figure 5, the end of the block after having extracted any potential outliers. The mid point, on the other hand, is determined by Δ offset. Note: when *bp-enc=1*, each of the 4 positions of the *Unique bp reg* will be populated by each of the Huffman decompressors.

4) *Global-base table access*: This table contains the global bases and Δ widths. Depending on *bp-enc* and *H-enc*, the index is:
   - 00: the original base pointer originating from the bp stream;
   - 01: calculated from the Huffman decompressor;
   - 1x: one of the 4 values of the *Unique bp reg* based on the value of the internal_bp_mask array.

   This table is implemented as a dual-port SRAM so that it can handle reads from two stream-decompressors in parallel. Note: this table is separate from the global-base table in the compressor despite the same content.
5) *Value reconstruction*: This step reconstructs the original value of the memory block by adding the global base read from the table in the previous step and the respective Δ extracted out of the Δ stream. The delta value is extracted using the Δ width read from the table (also in previous step) or the fixed one (if *F-enc=11*).

The decompressor is implemented in RTL. It can be clocked at 1.6GHz in a 7nm cell library. Each step corresponds to a pipeline stage. Decompressing a memory block takes 9 cycles. Steps 1 and 2 are used once for each memory block, while Steps 3-5 are used, iteratively, for each value in the respective stream. The decompressed values are added on the fly in a decompressed memory block buffer, but a final stage is needed to output the block to the C/D IP. Even if some memory blocks take less time to be decompressed (e.g., when *F-enc=00* or *H-enc=0*), we have designed the decompressor to have the same decompression latency for all formats. Back-to-back compressed blocks can be decompressed with a gap of 4 cycles, which can be bridged by adding more decompressors. Finally, each decompressor is enhanced with an extra set of SRAMs for keeping an extra variant of a Global-base table.

C. Area

The C/D IP block and the GBDI compression and decompression engines are fully implemented in RTL. The design is clocked at 1.6GHz. The simulated C/D block has an area footprint of 0.4mm^2 in a 7nm technology process. GBDI adds about 0.1mm^2 to it; the compression engine is 0.0652mm^2 and the decompression engine is 0.0613mm^2. The amount of SRAM for the GBDI compressor is 68KB (=2x($8KB + 2KB + 24KB$)), while for the decompressor it is 40KB (20KB for each Global-base table variant).

V. Experimental Methodology

A. Architecture Model

We use gem5 [7] with the DRAM controller timing model [13] to evaluate the impact of different compression schemes on performance and memory traffic. We implement C/D (Figure 3) as a standard gem5 MemObject, which is placed before the memory controller (see Figure 2). The various subcomponents of the IP are modelled with sufficient detail, and we use realistic latencies for the different IP pipeline stages, based on the RTL implementation of C/D. GBDI compression/decompression takes 12 and 9 cycles, resp., per block although pipelining allows back-to-back compressed/decompressed block to allow for overlap of latencies. BDI compression and decompression take 3 cycles. Moreover, accessing the Address translator takes 3 cycles. A read that hits in the Prefetch Buffer takes 7 cycles, while a read that misses pays a latency of 19 cycles on top of the DRAM latency. The DRAM latency (tCAS-tRCD-tRP=14.17-14.17-14.17 ns) is modeled accurately by gem5.

We use the default gem5 O3 CPU model, which resembles a typical out-of-order CPU used in server systems. The detailed system parameters including C/D can be seen in Table I [22]. To keep the simulation time within reasonable limits, we scale down the number of cores and memory channels using a 4:1 ratio (4 cores, 1 channel). This assumption complies with a typical server system today. For example, recent Intel Xeon servers [1] with 24 and 48 cores have 8 channels establishing a 3:1/4:1 core:channel. Future trends (e.g., the recent CPUs from AMD [4] and Ampere [5]) show that the core pressure to memory will increase even more (8:1 and 16:1).

B. Applications

We use SPEC2017 with reference inputs to drive the experiments. We compare CR for all benchmarks except `cam4` which we could not run in our simulation framework. For detailed evaluation we use a subset of the applications that demand high bandwidth which is justified in detail in Section VI-C. Gem5 KVM-CPU model [20] is used to generate simulation points, which allows an application to run at native speed and collect the checkpoints, then restoring these checkpoints with a detailed CPU model to perform the actual simulation. This way, we can evaluate

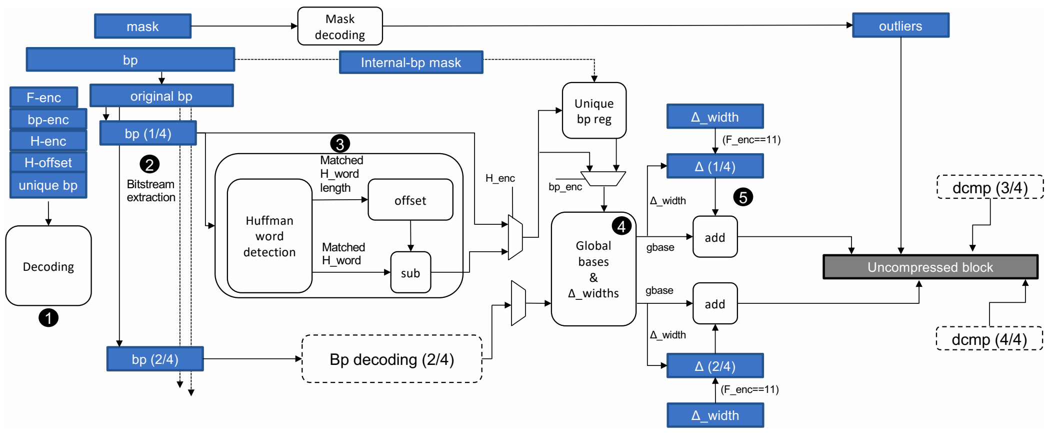


Figure 7: GBDI decompression engine.

Table I: System configuration.

Parameter	Value
Number of cores	4
CPU Clock Frequency	3GHz
Core Width/ROB/IQ/LQ/SQ	8/192/64/32/32
L1 Instruction Cache (private)	32KB, 2-way, 4c latency
L1 Data Cache (private)	64KB, 2-way, 4c latency
L2 Cache (private)	512KB, 8-way, 6c latency
L3 Cache (shared)	4MB, 16-way, 28c latency
DRAM	DDR4-2400 16x4, 1 channel
—	—
ATT size	256KB, 8-ways
Prefetch Buffer Size	64KB, 4-ways
Write Buffer Size	128KB, 4-ways
Request Queue Size	128 entries

simulation points that are spread across the whole execution of the benchmark, instead of only evaluating one phase of the benchmark after skipping a few billion instructions at the beginning.

For our experiments, we randomly generate 10 simulation points across the whole application's execution. Then, we restore from each of these checkpoints and run 1B warm-up instructions, and 100-M instructions of detailed simulation.

VI. Results

A. Compression Ratio

Here, we compare the compression ratio achieved from three different compression algorithms: BDI, BPC[2], and GBDI. For GBDI, all optimizations described in Section III-B are enabled. In addition, we also explore the resulting compression ratio of combining GBDI with BDI.

The input data to the compression algorithms consist of memory snapshots taken across the execution of the applications. The snapshots are captured with a fixed interval. For each benchmark, we aggregate the compression ratio across the snapshots by calculating the geometric mean.

The data analysis phase of GBDI, which establishes the global bases, is carried out on the first snapshot of each benchmark. The same global bases are then used for all the other snapshots. For GBDI, we use 2048 global bases.

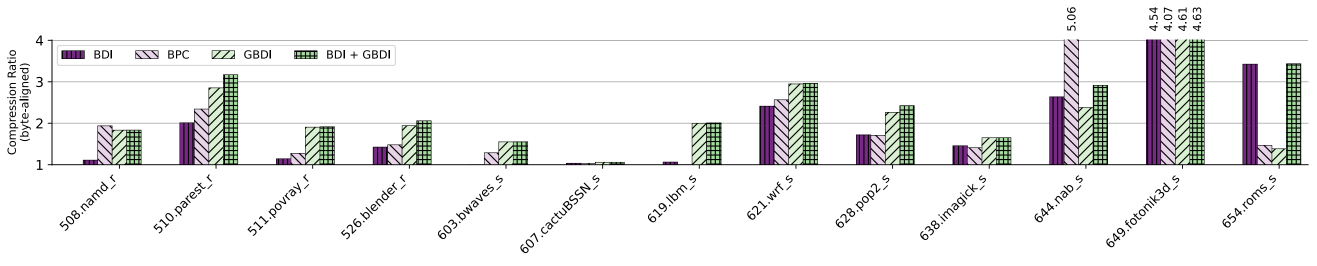
Figure 8 shows the resulting compression ratio, divided into two groups: floating-point (Figure 8a) and integer (Figure 8b) applications. Figure 8b also shows the geometric mean for all applications in both groups combined.

First, we compare GBDI, BDI, and BPC. For most benchmarks, GBDI outperforms both BDI and BPC, which shows the benefit of using global bases instead of intra-block bases. On average, across all benchmarks, GBDI reaches a compression ratio of 2.29x, while the corresponding numbers for BDI and BPC are 1.62x and 1.67x, respectively.

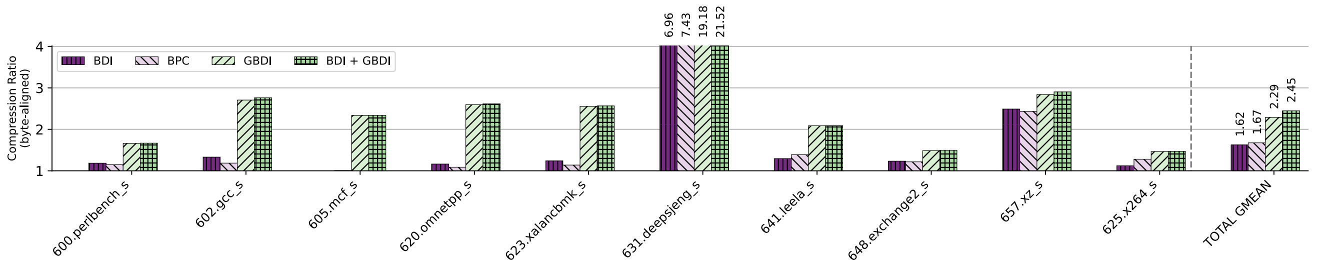
There are only two exceptions where using intra-block bases are beneficial: `roms`, where BDI performs well, and `nabs`, where BPC and BDI perform well. The memory snapshots of `roms` consist of 64-bit floating-point data where many values only differ in the least significant mantissa bits. This works especially well for BDI since it is the only compression algorithm which considers other formats than 32-bit values. For `nabs`, the values gradually increase as we sweep through the memory. An intra-block base yields small deltas which is reflected in the compression ratio for BDI and BPC.

Next, we investigate the combination of BDI and GBDI. On average, the combination performs slightly better than GBDI (2.45x vs 2.29x). There are a few exceptions where the combination is significantly better than any of the compression algorithms alone (e.g. `parest`, `pop2`, `nab`). However, in many cases, there is no or limited increase in compression ratio for the combination as compared to using GBDI alone.

[2]The original BPC algorithm was developed for GPUs, using a block size of 128B. In this section we have adapted the algorithm to consider 64B to match the common cache-line size of CPUs. This has a small negative impact on compression ratio because of higher influence from metadata. The reduction is approximately 7% for the geometric mean across all benchmarks.



(a) SPEC2017 Floating-Point



(b) SPEC2017 Integer + total geometric mean (both integer and floating-point applications).

Figure 8: Compression ratio for BDI, BPC, GBDI (with histogram binning), and the combination BDI and GBDI.

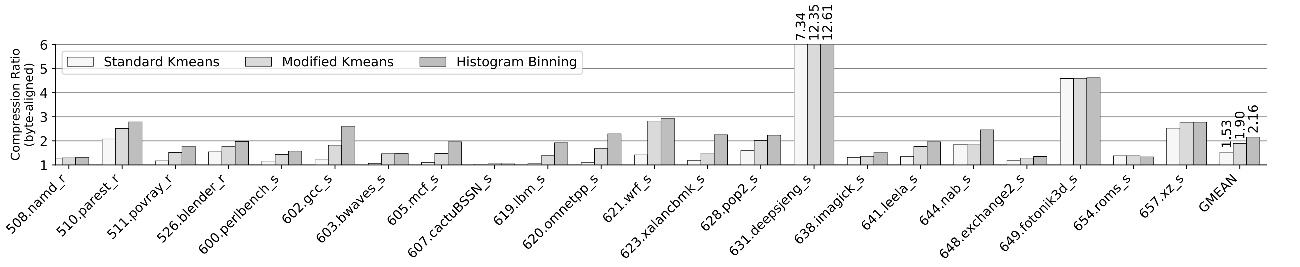


Figure 9: GBDI compression ratio when using different methods to establish the global bases. All methods use 128 global bases.

B. Histogram Binning vs Kmeans Clustering

Here, we show that histogram binning is key to efficient compression with global bases. Figure 9 shows a comparison between establishing global bases using standard kmeans clustering, our modified version of kmeans, as discussed in Section III-A1), and histogram binning. All methods are configured to find 128 global bases across the same snapshot, which are then used as global bases in GBDI.

Standard kmeans clustering almost always results in a lower compression ratio than modified kmeans and histogram binning. This confirms that establishing base values by minimizing the distance to all cluster values is suboptimal for compression (as discussed in Section III-A1). Using standard kmeans to establish the global bases for GBDI yields an average compression ratio of 1.53x.

In comparison, the modified version of kmeans, which instead minimizes the number of *bits* needed to encode the distance between the cluster center and the clustered values, yields an average compression ratio of 1.90x. However, histogram binning almost always performs better than modified kmeans (2.16x on average). This is because histogram binning continuously evaluates the compression ratio returned from GBDI, and chooses the best performing bases. Since compression ratio depends on both the number of deltabits *and* the grouping of values within each compressed block, histogram binning takes more dimensions into consideration than modified kmeans, and might find better bases.

C. Impact on Bandwidth, Traffic, and Performance

Here, we consider to what extent BDI, GBDI and the combined scheme BDI plus GBDI manage to reduce memory traffic and how this translates into performance improvements. Server systems run typically several instances of the same application. Henceforth, we consider workload mixes of four instances of the same application for all simulated SPEC2017 applications, on our four-core baseline system.

However, not all of these mixes will benefit from compression to reduce memory traffic for several reasons. One reason is that the bandwidth (b/w) pressure of some of the mixes will be so low that traffic reduction would have a negligible impact on performance. We note that `povray`, `nab`, `exchange`, `deepsjeng`, `imagick` and `x264` consume a memory bandwidth that is $\ll$1GB/s. Indeed, the memory bandwidth for these mixes is typically a few dozens of MB/s except for `x264` that consumes 350 MB/s. For this reason, we exclude all mixes containing these applications from the analysis. We also exclude `cactusBSSN` because it does not show any benefit from compression; the CR is close to 1, as shown in Figure 8a.

A third reason for not being able to benefit from traffic reduction using compression is when an application does not exhibit sufficient spatial locality. If so, the C/D IP disables compression if it detects a low hit ratio in the Address translator (typically less than 80%). For `xz` and `omnetpp`, the hit ratio of the Address translator is only 50%, while for the rest of SPEC2017 applications it is typically $\geq$ 95%. Other memory compression schemes [14], [23] have addressed this problem differently by blending the metadata with data and using predictors. GBDI is orthogonal to these techniques.

Altogether, we analyze the impact on memory traffic in detail for 14 out of the 23 workload mixes with four instances of the same application in the SPEC2017 suite. These 14 workload mixes are divided into two categories:

- Workload mixes that consume high bandwidth ($\geq$ 5GB/s). `Fotonik`, `mcf`, `lbm`, `parest`, `gcc` and `wrf` belong to this category.
- Workload mixes that consume medium bandwidth. `Roms`, `bwaves`, `pop`, `xalan`, `namd`, `blender`, `perlbench` and `leela` belong to this category.

Figure 10 shows the bandwidth consumed for the 14 selected workload mixes with the mix consuming the most/least bandwidth to the left/right, respectively. Four bars are shown for each mix from left to right corresponding to the baseline, GBDI, BDI and GBDI + BDI, respectively. As we can see, the six leftmost mixes consume a bandwidth that exceeds 5 GB/s which potentially can exhaust the available memory bandwidth. These mixes are referred to as *high bandwidth*. The next nine mixes also consume significant bandwidth but it is unclear whether they suffer from queuing delays that impact performance. These mixes are referred to as *medium bandwidth*. Overall, the bandwidth for all mixes is reduced when employing compression, regardless of the compression scheme due to traffic reduction. If an application is bandwidth limited and traffic is reduced, it will run faster and hence b/w consumption will go up.

Figure 11 shows the traffic for the three compression schemes relative to the baseline. Overall, we can see that GBDI reduces traffic for the high-bandwidth mixes substantially more than BDI alone with a geometric mean reduction of 32%. However, BDI does quite well and manages to reduce traffic, on average, by 19% compared to the baseline. The combination BDI with GBDI does not provide additional gains because the CR is improved by about 6%; i.e., a relatively small improvement that cannot materialize to traffic savings mainly due to misalignment of compressed blocks to DRAM block boundaries. The medium-bandwidth mixes also benefit from traffic reduction of 13% for GBDI and less for BDI (8%).

Figure 12 shows the speedup for each mix and compression scheme relative to the baseline. For the high-bandwidth mixes, we can see that GBDI provides an average improvement in performance of 16%. BDI, on the other hand, also provides a speedup but a substantially lower one (5%). GBDI plus BDI offers a slightly higher speedup (17%). By lumping together the high and medium bandwidth mixes, the average improvement in performance is 9%.

When zooming into the detailed results for each mix, a first observation is that `fotonik` exhibits a significant speedup (70%) for all compression schemes because of a large traffic reduction (63%). This is explained by the large amount of zero-pages, especially in the beginning of the execution (refer to the compressibility analysis in Section VI-D2). For `lbm`, `parest` and `gcc`, the high compression ratio from GBDI offers a substantial increase in speedup and a reduction in traffic but not so much reduction in bandwidth. This is because they will run faster and hence utilize the freed up b/w. For `namd`, `xalanc`, `leela`, `pop2` and `bwaves`, we note significant traffic reduction but no noticeable speedup. This is because b/w is not a bottleneck. The same explanation applies to `roms` under BDI where significant traffic reduction leads to limited speedup. For `mcf`, we notice performance degradation (5.5% for GBDI and 11% for BDI); this is because `mcf` is latency sensitive and the traffic savings are low. We experimentally find that `mcf` requires at least 10% traffic reduction for performance improvement.

D. Sensitivity Analysis

In this section we investigate the impact on performance for different settings of global bases (Section VI-D1), as well as the compressibility over time (Section VI-D2).

1) Number of Global Bases: Figure 13 shows the compressibility for all benchmarks when targeting 128, 1024 and 2048 global bases. Note that the compressibility for 2048 bases differ from Figures 8 and 9 since here, we compress a single memory snapshot (the first). On average, more global bases increases the CR: 128 bases give a geometric mean CR of 3.08x, while the corresponding number for 2048 bases is 3.51x. We also observe that some benchmarks benefit more from many global bases than others. For example, `namd`, `imagick`, `nab`, `mcf`, and `leela` see a substantial increase in CR when using 2048 bases as compared to 128 bases.

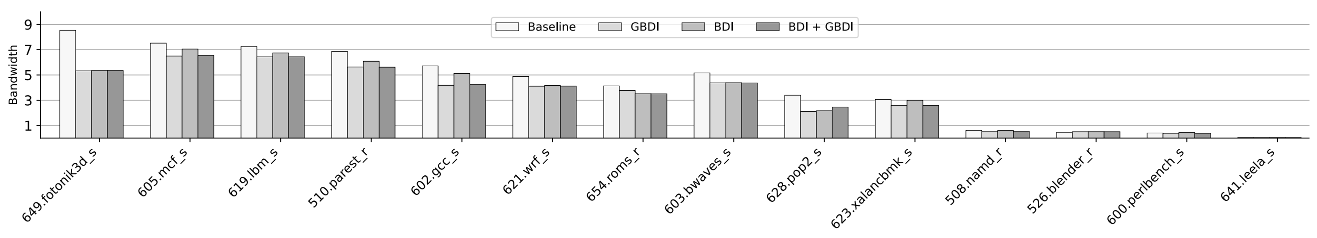


Figure 10: Simulated Bandwidth.

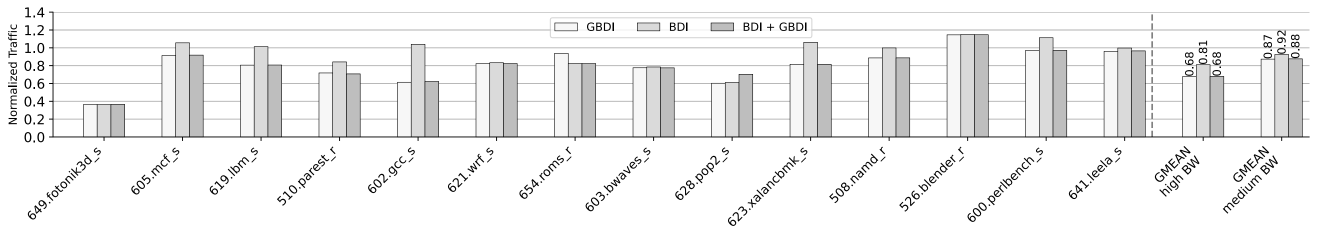


Figure 11: Total traffic normalized to baseline (without compression).

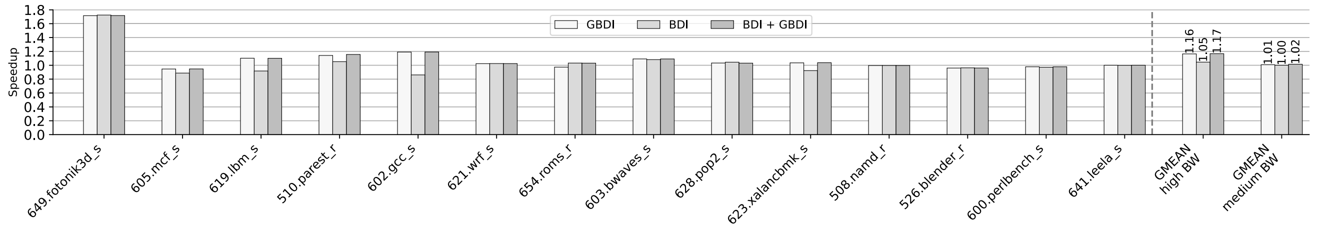


Figure 12: Speedup normalized to baseline (without compression).

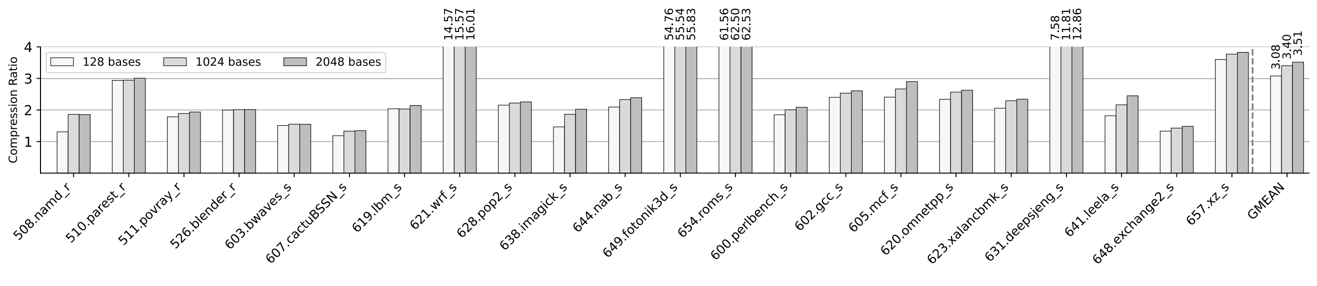


Figure 13: Sensitivity analysis: GBDI when varying the number of bases (one snapshot)

However, other benchmarks (e.g. parest, blender, and bwaves) are less sensitive to varying the number of global bases. This is because increasing the number of global bases is not beneficial if only a few value clusters are present in the data.

2) Compressibility over Time: GBDI adopts a data analysis phase in which the global bases are established. In this section, we explore the compressibility over time when the global bases are established in the beginning of the execution. Figure 14 shows the compressibility of memory snapshots taken in the beginning, the middle, and the end of the execution for each benchmark. The bases are established using data from the first snapshot. The figure shows, for each benchmark, the CR for each snapshot normalized to the CR of the first snapshot.

We observe that in many cases, the compressibility does not vary much across execution. This is similar to the observations made by Arelakis and Stenstrom [6], where they did a similar experiment for compressibility of SC^2. However, there are exceptions: for wrf, fotonik3d, and roms, the compressibility drops drastically. These benchmarks have a very high CR in their first snapshot due to a high occurrence

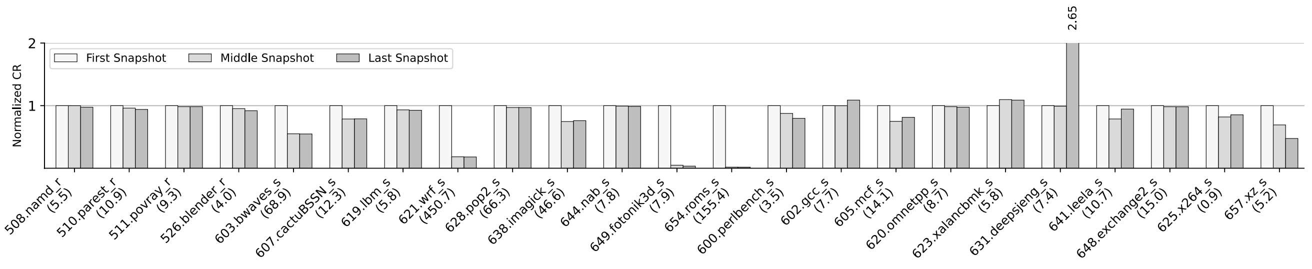


Figure 14: CR at the start, middle, and end of execution. The numbers in parantheses below each name is its runtime in minutes.

of zero-initialized memory. Hence, while the *normalized* compression ratio is low for the following snapshots, the actual compression ratio is still relatively high.

As mentioned in Section II-B, C/D may trigger re-training of the global bases if the CR drops below a given threshold. However, the overhead of this process is negligible since the training is light-weight (less than a second of wall-clock time) as compared to the runtime of the benchmarks (annotated in parentheses below each benchmark name in Figure 14). For example, a retraining may be issued in the middle of the execution for `wrf`, `fotonik3d`, and `roms`. The runtime of these benchmarks are in the order of tens to hundreds of minutes, which makes the time it takes to re-train the global bases negligible in comparison. Overall, the average impact is less than 0.1% across all benchmarks.

VII. Related Work

Compression for memory systems has been rigorously explored [21], and many compression algorithms have been proposed and/or adapted for this purpose [21]. Some examples include FPC [3], LZ [24], and C-Pack [9].

As previously discussed, the algorithms most related to GBDI are BDI [19] and BPC [16]. They rely on the assumption that the values stored in a memory block have value locality. This is common when data is stored in a *Structure Of Arrays* (SOA) format. It is also common, however, that data is stored in an *Array Of Structures* (AOS) format where more complex structures, with members of varying magnitude and word size, lie contiguously in memory.

In our approach, AOS data is instead supported by using many global bases, and individual values of frequently occurring magnitudes can be compressed as a delta to one such base. Hence, GBDI exploits *inter-block* (global bases) as well as *intra-block* value similarities.

SC^2 [6] is a statistical compression technique which adopts a data analysis phase in which a Huffman encoding is established. This phase is similar to the training phase which GBDI uses to establish global bases. However, while SC^2 targets compression of frequently occurring values, GBDI focuses on compression of similar values which form clusters.

VIII. Conclusions

This paper makes the important observation that Base-Delta-Immediate (BDI) compression, although simple and elegant, misses out on opportunities to increase compression ratio because it only exploits intra-block value locality. In contrast, this paper proposes Global-Base-Delta-Immediate Compression (GBDI) that exploits inter-block as well as intra-block value locality. It does so by doing data analysis in the background using a novel clustering algorithm called histogram binning that establishes a set of global bases so as to increase the compression ratio. We show that accelerators to support GBDI can be built with low latency and area requirements. The paper shows, experimentally based on detailed architectural simulation, how GBDI can be used to free up precious memory bandwidth and how this translates into higher performance for server workloads. It is shown that GBDI offers a compression ratio of 2.3×, as compared to 1.6× for BDI, yielding 1.5× higher bandwidth, as compared to 1.2× for BDI and 1.1× higher performance, on average, for the applications in the SPEC2017 benchmark requiring medium to high bandwidth compared with a baseline without compression support.

Acknowledgment

The authors would like to thank Professor Devdatt Dubhashi and Docent Morteza Haghir Chehreghani who provided valuable insights and expertise in the area of clustering algorithms. We are indebted to the European Commission for funding the development of the architectural simulation model of the C/D IP (Project MaxiMem, no: 858498, SME Instrument, H2020), and to the Swedish Foundation for Strategic Research for support of this research in the PRIDE project.

References

[1] Intel xeon w-3375 processor. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/217246/intel-xeon-w3375-processor-57m-cache-up-to-4-00-ghz/specifications.html

[2] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[3] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. USA: IEEE Computer Society, 2004, p. 212.

[4] AMD. Amd epyc 7003 series processors. [Online]. Available: https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf

[5] Ampere. Ampere altra max datasheet. [Online]. Available: https://amperecomputing.com/wp-content/uploads/2021/06/Altra_Max_Rev_A1_DS_v0.80_2021061291.pdf

[6] A. Arelakis and P. Stenstrom, "Sc^2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. IEEE Press, 2014, p. 145–156.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[8] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[9] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[10] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 546–558. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00051

[11] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 74–85.

[12] A. Ghasemazar, P. Nair, and M. Lis, "Thesaurus: Efficient cache compression via dynamic clustering," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 527–540. [Online]. Available: https://doi.org/10.1145/3373376.3378518

[13] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating dram controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 201–210.

[14] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. B. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 326–338. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00034

[15] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[16] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 329–340. [Online]. Available: https://doi.org/10.1109/ISCA.2016.37

[17] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[18] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.

[19] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 377–388. [Online]. Available: https://doi.org/10.1145/2370816.2370870

[20] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 183–192.

[21] S. Sardashti, A. Arelakis, P. Stenstom, and D. A. Wood, *A Primer on Compression in the Memory Hierarchy*. Morgan & Claypool, 2015.

[22] Wikichips. Neoverse n1 - microarchitectures - arm. [Online]. Available: https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1

[23] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 570–581.

[24] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.