MBZip: Multiblock Data Compression

RAGHAVENDRA KANAKAGIRI, Indian Institute of Technology, Madras
BISWABANDAN PANDA, Indian Institute of Technology, Kanpur
MADHU MUTYAM, Indian Institute of Technology, Madras

Compression techniques at the last-level cache and the DRAM play an important role in improving system performance by increasing their effective capacities. A compressed block in DRAM also reduces the transfer time over the memory bus to the caches, reducing the latency of a LLC cache miss. Usually, compression is achieved by exploiting data patterns present within a block. But applications can exhibit data locality that spread across multiple consecutive data blocks. We observe that there is significant opportunity available for compressing multiple consecutive data blocks into one single block, both at the LLC and DRAM. Our studies using 21 SPEC CPU applications show that, at the LLC, around 25% (on average) of the cache blocks can be compressed into one single cache block when grouped together in groups of 2 to 8 blocks. In DRAM, more than 30% of the columns residing in a single DRAM page can be compressed into one DRAM column, when grouped together in groups of 2 to 6. Motivated by these observations, we propose a mechanism, namely, MBZip, that compresses multiple data blocks into one single block (called a zipped block), both at the LLC and DRAM. At the cache, MBZip includes a simple tag structure to index into these zipped cache blocks and the indexing does not incur any redirectional delay. At the DRAM, MBZip does not need any changes to the address computation logic and works seamlessly with the conventional/existing logic. MBZip is a synergistic mechanism that coordinates these zipped blocks at the LLC and DRAM. Further, we also explore silent writes at the DRAM and show that certain writes need not access the memory when blocks are zipped. MBZip improves the system performance by 21.9%, with a maximum of 90.3% on a 4-core system.

CCS Concepts: • **Computer systems organization** → **Processors and memory architectures**; • **Information systems** → *Data compression*;

Additional Key Words and Phrases: CPU cache, memory, performance

ACM Reference format:
Raghavendra Kanakagiri, Biswabandan Panda, and Madhu Mutyam. 2017. MBZip: Multiblock Data Compression. *ACM Trans. Archit. Code Optim.* 14, 4, Article 42 (December 2017), 29 pages.
https://doi.org/10.1145/3151033

1 INTRODUCTION

In multicore systems, the shared memory resources, such as last-level cache (LLC) and memory, play a defining role in determining overall system performance. As more applications run on a multicore system, effective utilization of the LLC capacity is the key to improving system performance. Ideally, the capacity of an LLC should be large enough to hold the working set(s) of multiple applications, thus preventing the costly main memory accesses. However, increasing the

Authors' addresses: R. Kanakagiri, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417; email: raghavendra066@gmail.com; B. Panda, Office: KD 203, Department of Computer Science, Indian Institute of Technology Kanpur; email: biswap@cse.iitk.ac.in; M. Mutyam, BSB 356, Dept. of CSE, IIT Madras, Chennai - 600 036; email: madhu@cse.iitm.ac.in.

LLC size beyond a limit incurs significant area and power overhead. Thus, effectively utilizing the LLC to achieve better system performance has been an active area of research.

On the memory front, aspects such as memory latency and bandwidth have a noticeable impact on system performance. Given that accessing the memory incurs significantly longer delays, improving the on-chip caches' performances has been of great interest. But, owing to the on-chip area limitations of these caches, performance improvements at memory also are important. Since DRAM is the ubiquitous main memory technology, we consider it for this work; for the reminder of the article, we use the terms main memory and DRAM interchangeably.

Cache compression techniques play an important role in improving system performance by increasing the effective LLC capacity. These techniques compress a single cache block by exploiting various data patterns, such as zeros, frequent values [47] (frequent values stored in a cache block are encoded so that data values can be represented as series of codes), frequent patterns [4] (compresses the most commonly used data patterns), and base-delta-immediate (B△I/BDI) [38] (exploits the small deltas among the data values to compress) that are present within a cache block. Along with these cache compression techniques, there are other works that focus on developing the compressed LLC organizations. The main focus of these techniques is to reduce the hardware overhead in terms of metadata, unused cache space, and to improve the energy efficiency when compression is employed.

Similar to LLC, compression is employed at the memory by exploiting data patterns within a block of data. LCP [37] is a main memory compression technique that employs existing compression techniques to compress a block of data *individually*. When compression is employed at the main memory, the challenge is to reduce the steps involved in locating a compressed block. To locate a requested block requires additional computational steps that need to take into account the size of the blocks that are placed before it in a page. Processors employ virtual-to-physical address translation to locate a block of data in the memory. Compressed blocks are variably sized, and hence would require changes to the TLB, Operating System (OS), and memory controllers to support compression. On-chip caches (including L1 caches) typically use tag bits derived from the physical address bits to avoid aliasing. With memory compression, the physical address of a block is not fixed and hence would require a fast translation of virtual-to-physical address so as to not increase the critical L1 cache access latency. These requirements pose challenges for easy adoption of main memory compression.

In this work, we explore compression of multiple data blocks, both at the LLC and DRAM. Below, we present our insight and motivation in this regard.

Opportunity: Existing techniques on cache compression—such as BDI, frequent value compression (FVC), and frequent pattern compression (FPC)—increase cache capacity by compressing a single cache block independently, which we refer to as *intrablock* compression. However, applications exhibit data locality that spread across multiple consecutive blocks, which opens up opportunities for *interblock* compression. Such interblock compression, in which data locality is spread across multiple DRAM columns of a single DRAM page, presents an opportunity at the DRAM as well.

Key Observation: Motivated by this insight, we study 21 SPEC (2000/2006) CPU applications and find that there is significant opportunity available for compressing multiple consecutive data blocks both at the LLC and DRAM. We observe that, at LLC, across these applications, around 25% (on average) of the cache blocks can be compressed into a single block. The number of compressible cache blocks varies between 2 to 8. At DRAM, we observe a similar trend, in which more than 30% (average) of the columns residing in a single DRAM page can be compressed into one DRAM column when grouped together in groups of 2 to 6. If this available compression of multiple columns can be achieved in DRAM, the uses are multifold. First, in DRAM, per access (using

CAS), 64B of data can be transferred over 8 data-bus cycles. If multiple columns are compressed into a single column, the data transfer time is greatly reduced. Second, the number of DRAM accesses reduces because the compressed column can satisfy possible nearby block requests by itself. Third, the data-bus availability for other requests improves, improving bandwidth utilization. Last, DRAM power consumption reduces due to reduced accesses. We give a detailed description of these observations in Section 3.

Contributions: We make the following contributions:

(1) We exploit data patterns across multiple consecutive blocks both at the LLC (i.e., MBZip-C) and DRAM (i.e., MBZip-M), and compress them into a single data block called a *zipped* block.

(2) We propose a simple modification to the tag structure at the LLC that provides an effective way to access multiple cache blocks that reside in a *zipped* block without incurring any additional redirectional delay. We use a combination of conventional and superblock tags to index into the compressed cache. Our technique can handle different types of applications effectively:
   (a) If the application exhibits spatial locality and is compression friendly, superblock tags are used to track the cache blocks.
   (b) If the cache blocks are not compressible, conventional tags are used. The resulting scenario is similar to a conventional cache organization.

(3) At the DRAM, we keep the address computation logic simple, that is, to locate any block of data in our technique, the same address mapping as that of a conventional DRAM is used. Our proposed technique is lightweight and can be deployed easily.
   (a) It does not require changes to the OS. It is a hardware solution that is transparent to the OS.
   (b) Since the compressed blocks have a fixed address, on-chip caches can still be physically tagged without requiring any changes to the existing logic.
   (c) If the size of the compressed block increases, our technique does not require the compressed page to be moved to a bigger slot. It is written to the memory as any other conventional write, without generating extra accesses or page faults.
   (d) It does not make any assumptions about the data patterns within a page.
   (e) Since both uncompressed and compressed blocks have a fixed location, our technique does not require additional accesses/pointers to locate either the uncompressed or compressed blocks.

   To the best of our knowledge, this is the first work that explores multiblock data compression at the main memory and increases bandwidth, yet still retains the address mapping of a conventional DRAM.

(4) We provide a synergistic mechanism that coordinates MBZip at the LLC and DRAM together (referred to as MBZip-CM). Experimental results for a 4-core system demonstrate that MBZip-C and MBZip-CM improve system performance by 15.5%, and 21.9%, respectively. MBZip-CM reduces off-chip bandwidth requirements by 21.1%.

(5) Last, we also explore silent writes at the memory (a write to a location in memory that does not change the existing data in that location). We show that, in certain cases, a write request can simply be squashed without generating a read or write to the memory.

2 BACKGROUND

Base-Delta-Immediate (BDI) compression: BDI [38] is one of the simplest, most high-performing, and easy-to-implement cache compression techniques. The technique is based on the observation that the differences between the data values that are stored within a cache block are small. For example, a 64B cache block can be viewed as eight 8B segments. If the difference (delta) between the first segment and every other segment in the block is 1B, then BDI stores only the deltas along with the first segment. Thus, it compresses a 64B cache block to 16B (1 8B segment + 8 1B deltas). The data pattern exploited in the example is referred to as B8-$\triangle$1[1]. Similarly, BDI can compress blocks that have other frequently observed data patterns: B8-$\triangle$2, B8-$\triangle$4, B4-$\triangle$1, B4-$\triangle$2, B2-$\triangle$1, repeated values and zeros. BDI stores encoding bits along with the tag associated with the block. These bits represent the data pattern used for compressing the block. Depending on the pattern exploited, the compressed block size can vary. Both the size of the base and the deltas are determined during block compression. Since each compressed word has a specific location in the compressed block (and not relative to other words in the block), the decompression latency is very low (1 cycle) when compared to other compression techniques.

Cache compression using BDI (BDI-Cache): On a read/write request at the cache, the address of the request is split into three fields from LSB to MSB, that is, offset, index, and tag. The index bits are used to select a cache set. The tag bits of the address are compared with the tag bits of the cache blocks that are resident in the selected cache set. If there is a tag match, the offset bits are used to extract the requested bytes of data for read/write. The tag space contains one tag per data block.

When compared to a generic cache, BDI can potentially store more cache blocks per set due to compression. For example, in a 256KB, 16-way, 64B cache, if all the blocks can be compressed to half (32B) of their respective fixed sizes, BDI can store a total of 8,192 cache blocks. However, to identify these blocks, BDI needs 8,192 tags. In general, it is impossible to speculate the compressibility of applications. Thus, the number of additional tags required for a compressed cache varies. Empirically, BDI determines that doubling the tags is sufficient for most of the applications.

Each cache set in BDI is considered to be constituted by fixed 8B data segments. For a generic cache set of size 1,024B, BDI splits the data space of the set into 128 segments of 8B each. An uncompressed cache block will occupy eight segments, whereas a compressed cache block will occupy less than eight segments. Apart from the conventional tag bits, a tag entry in a BDI cache is appended with additional information, such as a segment pointer and encoding bits (4b). A segment pointer points to the location of an 8B segment in the set. The size of segment pointer is equal to $log_2(\textit{no. of 8B segments in the set})$. Similar to a generic cache, the index is used to determine the cache set for a data request. When a tag match happens, BDI uses the segment pointer and the encoding bits of the tag to access the data space. The segment pointer gives the location of the first 8B segment of the cache block. The encoding bits indicate the data pattern that was exploited by BDI to compress the block. Hence, it also gives the information on the number of segments that the cache block occupies. Figure 1(a) shows the tag and data structure of a generic cache. For illustration purposes, we consider a cache set that can accommodate two uncompressed blocks of data of 32B each. The two cache blocks UB0 and UB1 contain $b0u$ and $b1u$ as their data, respectively (u represents uncompressed data). These blocks are mapped to their respective tags $t0$ and $t1$.

Figure 1(b) shows the tag and data structure of a BDI-based compressed cache. The number of tags in BDI are doubled when compared to a generic cache and hence contain four tags: $t0$, $t1$, $t2$, and $t3$. The cache set contains four compressed blocks, CB0, CB1, CB2, and CB3. Block CB0 is compressed to 16B, shown as $b0$ per segment. Similarly, CB1, CB2, and CB3 occupy 1 ($b1$), 3 ($b2$), and 2 ($b3$) segments, respectively. Note that BDI cache does not modify the index of the cache block. Hence, the same index function as that of the generic cache is retained.

[1]B(Base)-$\triangle$(Delta).

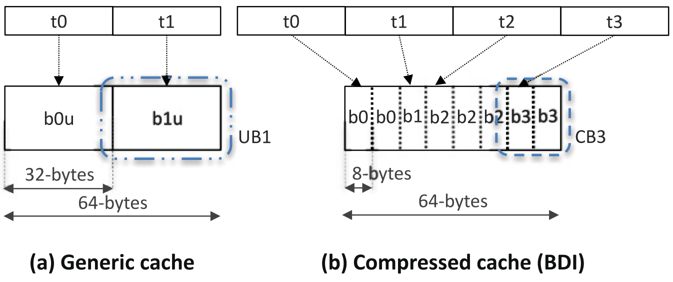


Fig. 1. Mapping of tag and data entries in (a) an uncompressed cache, and (b) BDI-based compressed cache [38]. UB1 is an uncompressed block and CB3 is a compressed block.

Main memory compression: Compressed blocks of data can be of variable sizes. The actual location of a block of data in a compressed memory page depends on the size of the blocks that are placed before it. To locate a requested block in the compressed memory requires additional computational steps that need to take into account the respective sizes of the previously placed blocks in that page. This can increase the main memory access latency further.

To solve the problem of complexity involved in locating a block of data in a compressed page of memory, LCP uses fixed size for compressed blocks in a page. This ensures that the location of a block in the compressed page is fixed and can easily be computed. To accommodate the blocks that cannot be compressed, LCP uses exception storage in the same page. Metadata associated with the page is used to locate the blocks. From the working of LCP, we note the following drawbacks: (i) LCP requires operating system support to manage variably sized compressed pages. The OS needs to maintain a pool of allocated and free pages. (ii) When a block of data is written back to the memory, the new data need not fit in its earmarked compressed space. Further, there might not be enough space in the exception storage to accommodate this block. This can result in an overflow and would require the OS to migrate the entire page to a larger-sized page.

3 MOTIVATION

Motivation through an example: Figure 2 shows the cache space that can be saved by compressing blocks independently using BDI and with a zipping technique that can compress across multiple blocks. For ease of illustration, we consider cache blocks of 16B. Figure 2(a) shows the contents of two consecutive cache blocks A and $A + 1$. Both blocks contain 4 words, each of size 4B. BDI compresses blocks A and $A + 1$ independently to 8B compressed blocks; 16B of cache space is saved in total. However, if we apply BDI across both the blocks creating a zipped block, then one 12B zipped block can contain both blocks A and $A + 1$ with a common base $0X00006000$, saving 24B of cache space in total.

Figure 2(a) shows the contents of two consecutive cache blocks B and $B + 1$ that have the same data values present in each of their 4B words. BDI compresses blocks B and $B + 1$ independently to 4B compressed blocks; 24B of cache space is saved in total. However, as in the case of A and $A + 1$, when we compress both B and $B + 1$ together in a zipped block, 28B of cache space can be saved in total. In addition to the cache space saved, compressing n blocks together requires only one set of encoding bits as opposed to n set of encoding bits.

Motivation through observations: The following observations show that there is opportunity to compress multiple blocks of data together, both at the LLC and DRAM. We use BDI to do so. The effectiveness of BDI is less for floating-point applications. Floating-point data has exponent and mantissa fields. It is difficult to split this data into fixed-size segments such that the

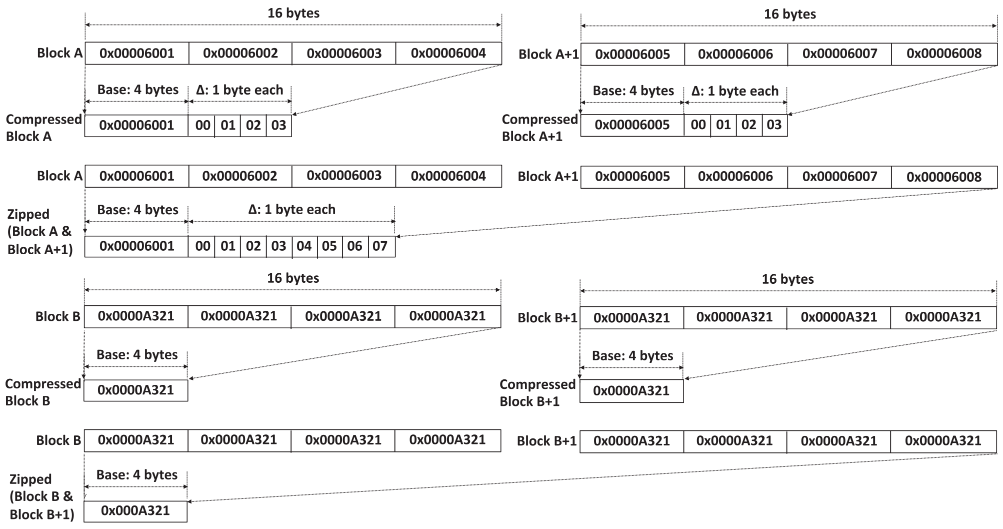


Fig. 2. Cache space that can be saved by compressing multiple blocks together into one block. A and A+1, B and B+1 are consecutive blocks that share data patterns. Compared to an independent compression of data blocks, a grouped compression that compresses multiple blocks saves additional cache space and requires only one set of encoding bits.

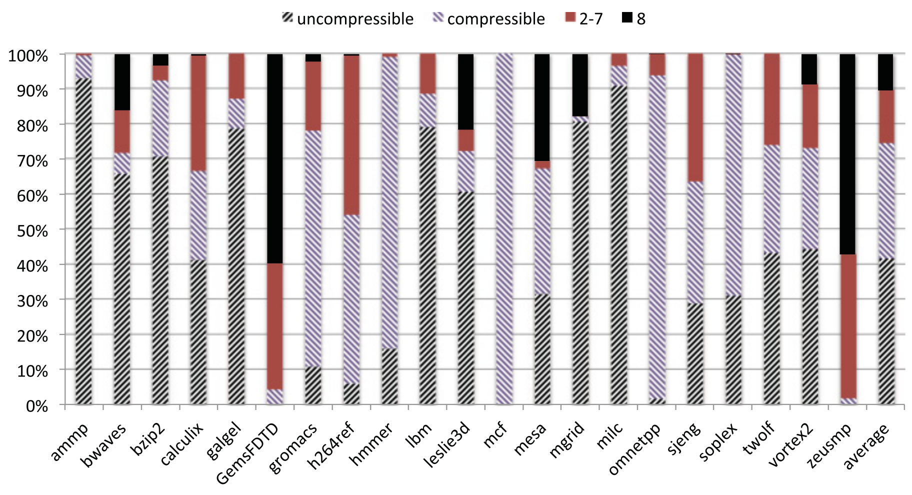


Fig. 3. Distribution of cache blocks that can be zipped to a single cache block. *a-b* signifies *a* to *b* number of blocks that are zipped into a single cache block.

difference between these segments (deltas) occupy lesser space. However, if cache blocks store constant floating-point values—for example, 0.0—or large floating-point constants, they can be exploited for compression. Also, if a floating-point application has a mixture of many pointers and floating-point values, BDI still would be effective in exploiting these pointer values.

Observation 1: Figure 3 shows the fraction of cache blocks that can be compressed into a single cache block in a 1MB LLC. In this study, we try to compress multiple (2–8) consecutive cache blocks. On average, across a wide variety of SPEC CPU 2000/2006 applications, we find that

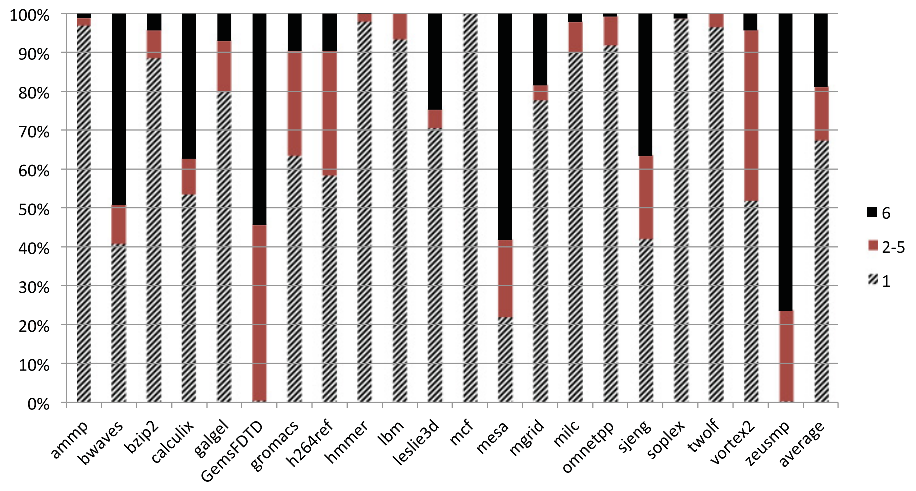


Fig. 4. Distribution of columns that can be zipped to a single column in DRAM based main memory. *a-b* signifies *a* to *b* number of columns are zipped into a single column.

around 25% of cache blocks can be zipped into a single cache block. If cache blocks that contain only zeros are excluded, we observe that, on average, around 12.6% of blocks can be compressed into a single zipped block. In floating-point applications, such as calculix and bwaves, around 16.9% and 11.7% of the blocks can be compressed into a single zipped block, respectively.

Observation 2: Figure 4 shows the opportunity for zipping multiple columns that are present within a single DRAM page. For this experiment, we consider a DRAM-page of size 8KB and only those pages that are touched during the simulation. On average, around 33% of DRAM columns can be zipped into a single column of size 64B.

4 CHALLENGES

Zipping multiple blocks of data at the LLC and main memory pose certain challenges. The zipping mechanism should be robust enough to handle these challenges:

At the LLC: A cache block at the LLC can fall into any of the following three categories:

(1) Uncompressed cache block: A block that is not compressible (64B).
(2) Compressed cache block: A block that is compressible to less than 64B, but which cannot be zipped with other blocks.
(3) Zipped cache block: Consecutive blocks that are compressed as a single cache block.

The zipped cache organization should be able to use the cache space saved by independently compressing blocks that belong to category 2. For category 3 cache blocks, the proposed organization should provide simple ways to access the cache blocks present within a zipped block without any additional redirectional delay. When the blocks are compressed, depending on the compression pattern exploited, the size of the resultant zipped block can vary. It can even be less than 64B despite containing multiple blocks of data inside it. The zipped cache organization should be able to efficiently use the free cache space when the zipped block is less than 64B.

At the DRAM: The size of a DRAM page is fixed, with multiple columns in each page. Compressing multiple columns into a single column can lead to zipped columns of different sizes. The challenges of implementing zipping at the main memory are as follows:

(1) Variably sizes zipped columns at the memory leads to variable column sizes, which leads to variably sized physical pages. This complicates the memory management unit that has to map fixed-sized virtual pages to variably sized physical pages. It can also lead to DRAM fragmentation.
(2) As the zipped columns can be of variable sizes, the computation of the actual location of a column address is dependent on the previous zipped column sizes in that page.
(3) A particular column can be part of any zipped column. For example, in a page containing columns c_0 to c_{127}, c_5 might be in the first column zipped along with c_0 to c_4, or in the 2nd column zipped with c_1 to c_4, or might be in its own column in uncompressed form. Unlike cache, memory does not have a structure like the tag that can point to the column directly. Hence, the information needs to be stored as metadata per zipped block. This metadata will, by itself, consume space and might offset the space savings obtained by zipping. Apart from offsetting the space savings, any access to the memory to fetch a column should access the metadata structure first before accessing the column. Both (2) and (3) increase the critical path length for accessing a column.
(4) A write-back to the DRAM that stores columns in a zipped form might need rezipping which, in turn, might change the address mapping. For example, if column c_5 was part of a zipped column and a write-back now updates c_5 with data that no longer can be zipped with other columns, c_5 might end up requiring more space than it previously occupied in the page.

In the following sections, we describe our zipping mechanism (MBZip) and how it handles these challenges.

5 MBZIP

5.1 MBZip at the LLC (MBZip-C)

In MBZip cache, multiple consecutive cache blocks are compressed into a single block, called a zipped block. In generic cache, if block X maps to set Y, consecutive block X+1 maps to set Y+1. If blocks X and X+1 are compressed into a single zipped block and placed in set Y, a request to block X+1 has to look up set Y rather than Y+1. Thus, the index function of MBZip-C needs modification to handle such accesses.

As categorized in Section 4, MBZip-C has to handle three categories of cache blocks: (i) uncompressed blocks, (ii) compressed blocks, and (iii) zipped blocks. To identify a block in MBZip-C, we use two different index functions, referred to as *h1* (see Figure 6) and *h2* (see Figure 7). The index function *h1* is the same as the index function of a generic cache. As shown in Figure 7, for *h2*, the index bits are not extracted right after the offset bits but rather after the zip bits. The number of zip bits is equal to log_2(maximum number of blocks that can be zipped into a single block). This ensures that consecutive blocks in a zipped block are mapped to a single set in MBZip-C. The uncompressed and the compressed blocks are indexed by *h1*, whereas zipped blocks are indexed by *h2*.

Consecutive blocks X and X+1 map to a same set if indexed using *h2*. They can be zipped and placed in the set indexed by *h2*. In general, *h2* ensures simple mapping without requiring any additional redirections for blocks like X+1. It can be observed that, in both generic and BDI caches, consecutive cache blocks get mapped to consecutive cache sets. However, in MBZip-C, using the proposed indexing mechanism, multiple cache blocks that are zipped into a single block are indexed into one cache set.

When compared to a generic cache, since BDI can potentially store more cache blocks per set due to compression, the number of tags per set is doubled. Similar to BDI, in MBZip-C, the number

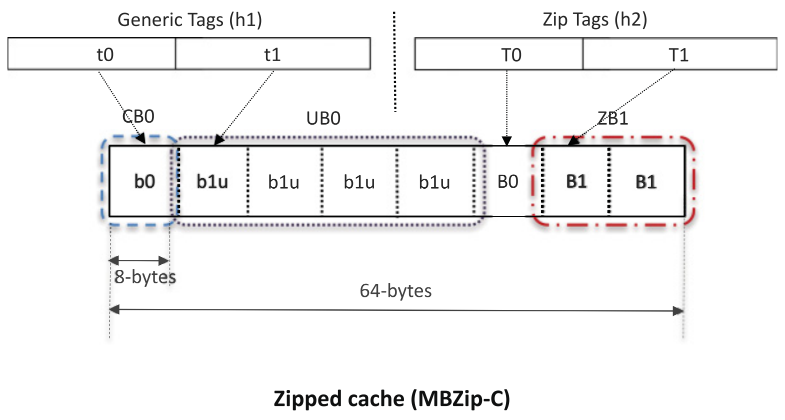


Fig. 5. Mapping of tag and data entries in MBZip cache. CB0 is a compressed block, UB0 is an uncompressed block and ZB1 is a zipped block. Tags t0 and t1 are indexed using h1, and T0 and T1 are indexed using h2.

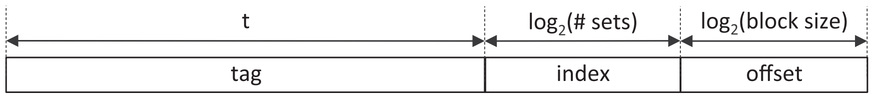


Fig. 6. Index function *h1*.

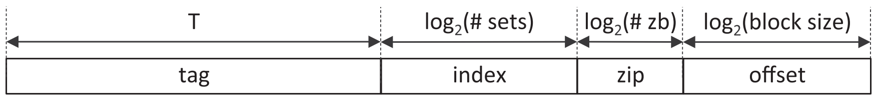


Fig. 7. Index function *h2*. *zb* stands for the number of blocks that are zipped into one cache block.

of tags are doubled. But in MBZip-C, half of these tags use the *h1* index function and the other half use *h2* as their index function. On a cache request, tags indexed using both h_1 and h_2 are searched in parallel. The total number of tags searched for one request is same as that of the BDI cache.

The indexing of both h_1 and h_2 happens simultaneously. There is no extra latency in terms of redirection to find a block in the cache whether it is stored as an uncompressed/compressed block or as part of a zipped block. A tag entry, whether indexed by *h1* or *h2*, is appended with a segment pointer and encoding bits in addition to the tag bits. The data space is split into fixed-sized segments of 8B each. We differentiate the tags indexed by *h1* and *h2* and represent them as t and T. T tags, called zip tags, will also have additional bits for each compressed block in the zipped block representing the coherence state of that block. These additional bits help us maintain/track the coherence at block level, similar to a generic cache.

Figure 5(c) shows the tag and data structure of an MBZip cache, where $t0$, $t1$ are the generic tags and $T0$, $T1$ are the zip tags of the set. $t0$ points to compressed block CB0, and $t1$ points to uncompressed block UB0. CB0 and UB0 contain $b0$ and $b1u$ as their data and occupy 8B and 32Bs, respectively. $T0$ and $T1$ point to zipped blocks ZB0 and ZB1, which contain the zipped data of several consecutive cache blocks. Zipped block ZB1 spans across two segments, shown as $B1$ per segment. ZB0 occupies only one segment and is represented by $B1$. In general, tags t can point to uncompressed or compressed cache blocks and tags T can point to uncompressed, compressed, or zipped cache blocks. Overall, in MBZip, we can store uncompressed cache blocks, compressed cache blocks, and zipped cache blocks of varying sizes. We study the possibility of zipping multiple

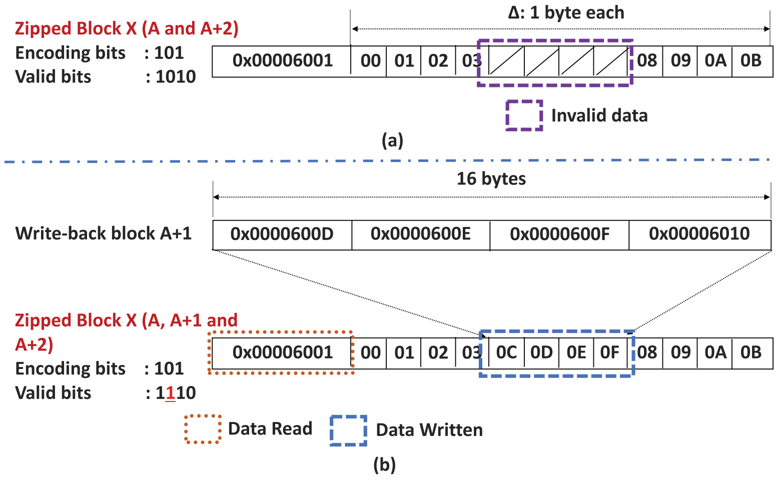


Fig. 8. (a) Zipped block X at the LLC. X contains blocks A and A+2. (b) Write-back block A+1 is rezipped with X.

cache blocks into a single block and find that zipping a maximum of eight blocks provides a good trade-off between performance gain and tag overhead.

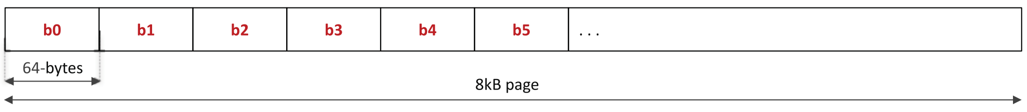
Design choice: We use a combination of both generic and zip tags. If the application exhibits spatial locality and is compression friendly, zip tags are used to track the cache blocks. If none of the cache blocks in the application are compressible, the generic tags are used. If only zip tags were used for an application that exhibits spatial locality (streaming) but is not compression friendly, there is a potential case for increase in conflict misses.

Handling write-backs to the LLC: When a cache block (X) is initially brought from DRAM to the LLC, it is allocated either as an uncompressed/compressed cache block (associated with tag t_X) or as a zipped cache block (associated with tag T_x). The cache block is transferred to upper levels of cache in uncompressed form. If this cache block X needs to be written with new data in the upper level cache, a write exclusive permission is obtained from the LLC controller. The data of X at LLC no longer contains updated data, but the tag (t_x or T_x) continues to maintain the coherence information.

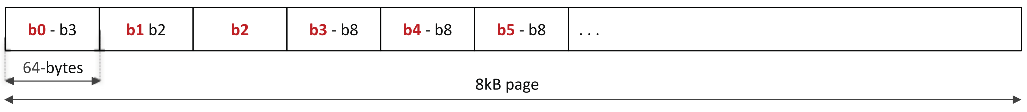
Block X, after being evicted from the upper level cache, is written back to the LLC. The LLC controller tries to rezip a write-back block with an existing zipped block at the LLC. A write-back to LLC can result in either of these two scenarios:

(1) If a write-back has an associated zip tag in the LLC, it would result in *read(base)-write*. If the write-back block cannot be zipped into the existing zipped block, a new tag is allocated. The write-back block is no longer part of the zipped block/tag (the corresponding valid bit is reset).
(2) If a write-back has an associated generic tag in the LLC, it would result only in a *write* (same scenario as that of a generic cache).

For example, in Figure 8, block A+1 is written back to the LLC from the upper levels of cache. The write-back block A+1 contains new data $0X0000600D$, $0X0000600E$, $0X0000600F$, and $0X00006010$. The base ($0X00006001$) and the encoding bits (101) of zipped block X are read. The cache controller tries to compress A+1 using the base $0X00006001$, that is, deltas are computed between the base and the new data. The compression pattern is given by the encoding bits 101. If A+2 is compressible



(a) Uncompressed page in conventional DRAM.



(b) Zipped page in DRAM.

Fig. 9. MBZip at the memory.

with the base, it is allocated as part of X in its respective position. A+2 is marked dirty in the zip tag. Note that the deltas associated with the other blocks (A, A+2) are *not touched*. Only a single compressed block is written back as part of a zipped block. If A+2 cannot be compressed with X, it is handled as an individual block.

5.2 MBZip at the Memory (MBZip-M)

MBZip-M corresponds to zipping blocks at the main memory. For explaining MBZip-M, we consider DRAM-based main memory that comprises 8KB pages, and each page comprises 64B columns. With MBZip-M, we intend to zip a maximum 6 consecutive data columns (we explain the reasoning behind this in Section 5.2.1.) into one column. There is 1:1 mapping of column addresses between conventional and MBZip-M main memory pages; that is, in MBZip-M, a particular column can be found in the same physical address space as that of a conventional uncompressed main memory page. But, the column can either be in zipped form or in uncompressed form. In zipped form, this column of data is compressed with its consecutive columns. If a column is uncompressed, it is the only column to occupy the entire column space of 64B. Figures 9(a) and 9(b) show a conventional page and a zipped page in DRAM, respectively.

To locate any b_x in MBZip-M, the same address mapping as that of a conventional DRAM is used. With MBZip-M, there is a possibility of fetching b_{x+1} to b_{x+5} in *one* single DRAM read, apart from fetching b_x. For example, as shown in Figure 9(b), when a read request is sent to b_0, a single read will fetch b_0 to b_3 as a zipped column. Similarly, a read to b_3 will fetch b_3 to b_8, whereas a read to b_2 will fetch b_2 alone in an uncompressed format. Note that there is duplication of data, that is, a particular column of data b_x, apart from being stored in its respective column space, can be part of other zipped columns between b_{x-5} to b_{x-1}. This is different from MBZip-C, in which a block can be present in only one location, either uncompressed, compressed, or as part of a zipped block. MBZip-C attempts to improve system performance by increasing the effective cache capacity, whereas MBZip-M attempts improving system performance by reducing the number of read requests sent to the main memory. When a zipped column is sent to the LLC controller, pending/future requests to other blocks in the zipped column can be serviced without generating extra memory-read requests.

Metadata per column: Figure 10(a) shows a zipped DRAM page with the metadata bits. MBZip-M stores 8b of metadata for each column. Out of the 8b, 3b are used to store compression-encoding bits. The remaining 5b store 1 valid bit for each of the 5 consecutive compressed blocks in the column. The first block b_x within a zipped column is always valid and has the same starting address as in a conventional DRAM page. We refer to this column as the *parent column*. For a zipped column consisting of 6 blocks, 5b represent the valid status of the blocks b_{x+1} to b_{x+5} in the parent

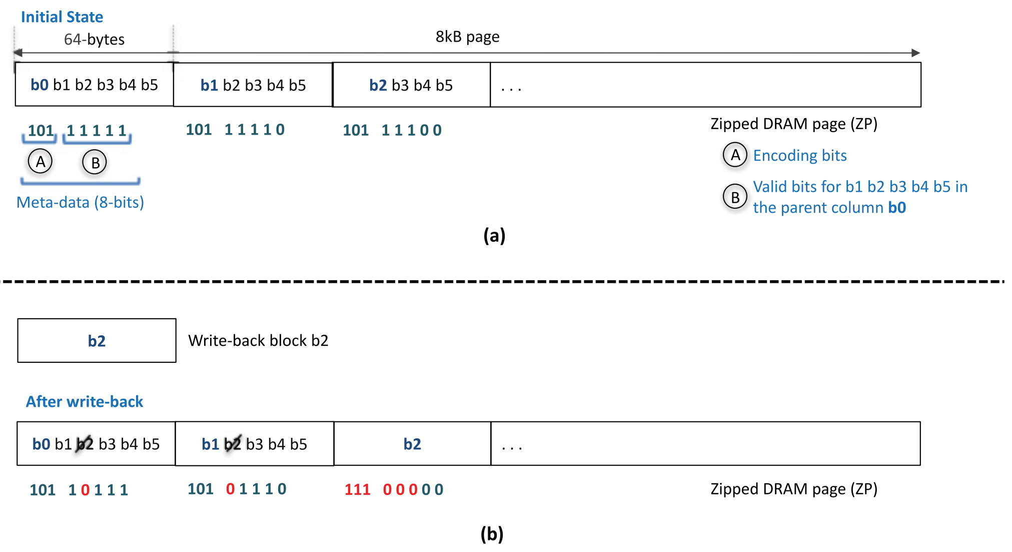


Fig. 10. (a) Zipped page in DRAM with the meta-data bits. (b) After block b2 is written back to the page, the meta-data bits are updated accordingly.

column b_x, and the 3b represent the encoding pattern used for compressing the blocks. For an uncompressed column, the 3 encoding bits indicate that the block is stored as an uncompressed block. In Figure 10(a), the parent column b_0 has blocks b_1, b_2, b_3, b_4, and b_5 zipped along with it. The valid bits are set accordingly. For parent column b_2, only the first three valid bits are set, indicating that the blocks b_3, b_4, and b_5 are part of the zipped column.

Write-back to the memory: In the case of a write-back from the LLC (say, for block b_x), the request gets added to the write queue. This block of data b_x can be part of zipped columns between b_{x-5} to b_{x-1} in the same DRAM page. The metadata corresponding to the DRAM page is updated, that is, the valid bit corresponding to b_x is reset in all the associated zipped columns. Figure 10(b) shows the status of the zipped DRAM page after write-back block b_2 has updated the memory. The valid bit corresponding to block b_2 is reset in the parent columns b_0 and b_1. Note that we do not generate extra writes to columns b_0 or b_1. Only the metadata corresponding to the page is updated. b_2 is written back to the DRAM as an uncompressed block, overwriting the previous zipped column.

In Section 5.3.1, we describe in detail the possible scenarios in MBZip-CM, in which associated zipped columns (b_{x-5} to b_{x-1}) get updated with the data of new block b_x without generating any extra writes to the DRAM.

5.2.1 *Metadata at the Main Memory.*

We store the metadata in a reserved DRAM space. Limiting the number of compressed data blocks in a zipped column to six translates to metadata of 128B per 8KB DRAM page (1B per 64B column). This, in turn, limits the number of required metadata reads to decode an entire zipped DRAM page to two (the metadata of 128B per page is stored in the reserved DRAM space across 2 columns). It also simplifies the address decoding logic of the metadata space.

Metacache: Programs exhibit spatial and temporal locality. Hence, multiple columns can get accessed in a single page within a short duration or the same pages might get accessed again in the near future. Instead of generating a metadata read to the reserved DRAM space every time a

DRAM access is made, we cache the metadata for frequently used DRAM pages. We refer to this as *metacache*.

An LLC miss for block b_x generates a request to the memory controller. The request gets added to the read queue, where it waits for its turn to read the corresponding column of data from the DRAM. When the request is added to the read queue, the metacache is searched to read its corresponding metadata. On a hit in the metacache, the column requested is read from the DRAM and sent to the LLC along with the metadata corresponding to the block. On a metacache miss, a request to fetch the metadata is issued to the DRAM. When both the metadata and the requested column data are available, the request is serviced. The metadata that was fetched is stored in the metacache. The metacache implements a simple LRU replacement policy. A write request to the DRAM might update the metadata contents. When a block is replaced from the metacache, if the victim block is dirty, a write request is generated to the reserved DRAM space.

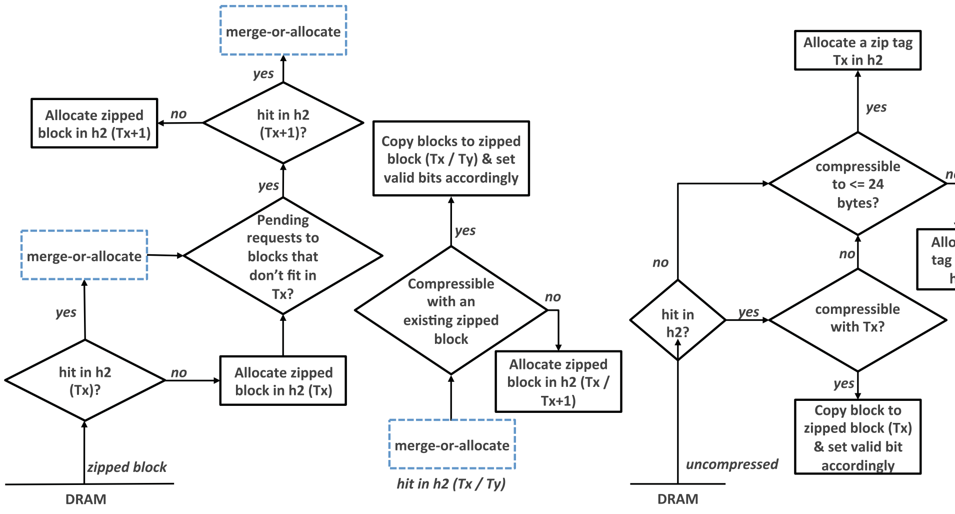
Optimization to handle metadata: The number of zipped columns in a DRAM page can vary from zero to *n*, *n* being the total number of columns in a conventional DRAM page. If there are no zipped columns in a page (nonzipped page), it follows that all the columns are stored in the uncompressed form and the respective valid bits in the metadata block are all set to 0. The number of zipped columns in a page is dependent on the application and its execution phase. We maintain a hardware structure along with the metacache, which is indexed using the page address. We refer to this structure as *metatable*. It contains a fixed number of entries and stores page addresses of frequently used nonzipped pages. When a metadata block is brought into the metacache, if the page contains no zipped column, an entry is made in the metatable. Any read/write request along with searching the metacache also indexes into the metatable. If this results in a hit, no read request is generated to the reserved DRAM space to fetch the metadata block, thus saving the extra metadata read. Alternatively, the Page Table Entry (PTE) can be extended to store a bit indicating whether the corresponding page is zipped or nonzipped. Unused PTE bits of existing systems (Intel x86-64 systems [1]) can be used to store this information.

5.3 MBZip both at Cache and Memory (MBZip-CM)

MBZip-CM is a mechanism that coordinates zipped blocks between the LLC and DRAM. For a given LLC miss request, the response from the DRAM can either be an uncompressed block or a zipped block. When a block address is indexed using *h2*, we refer to the resultant zip tag as T_x and if indexed using *h1*, we refer to it as t_x (generic tag). Figure 11 illustrates the flow of events on a response from the DRAM.

Uncompressed block response from the DRAM: At the LLC, the serviced block is indexed using *h2*. If the resultant zip tag T_x is already present in the zip tag structure, we check if the block is compressible with the existing zipped block pointed by T_x. To perform this check, only the base of the zipped block and its encoding bits are read. If the serviced block is compressible using the base that was read, it is allocated as part of the zipped block in its respective position. The corresponding valid bit is set in T_x.

If T_x is not present, or if the serviced block is not compressible with the existing zipped block, we perform an intrablock compression of the serviced block. If the block size upon compression is less than or equal to 24B, we allocate the block with a zip tag (T_x). In this case, the block can be zipped with other serviced blocks in future. If the block is not compressible, or if the size of the compressed block is more than 24B, we index using *h1* and allocate the block with a generic tag (t_x). If the size of the compressed block is greater than 24B, even if the block is compressible with other serviced blocks in future, the total compressed size would exceed 64B. Hence, it is not beneficial to use a zip tag in this scenario. Note that the allocated block with tag t_x can either be (i) a compressed block whose size is greater than 24B but still less than 64B, occupying fewer than


merge-or-allocate
yes
Allocate zipped block in h2 (Tx+1)
no
hit in h2 (Tx+1)?
yes
Pending requests to blocks that don't fit in Tx?
merge-or-allocate
yes
hit in h2 (Tx)?
no
Allocate zipped block in h2 (Tx)
zipped block
DRAM
Copy blocks to zipped block (Tx / Ty) & set valid bits accordingly
yes
Compressible with an existing zipped block
no
Allocate zipped block in h2 (Tx / Tx+1)
merge-or-allocate
hit in h2 (Tx / Ty)
Allocate a zip tag Tx in h2
yes
compressible to <= 24 bytes?
no
Allocate tag tx in h1
no
no
hit in h2?
yes
compressible with Tx?
yes
Copy block to zipped block (Tx) & set valid bit accordingly
uncompressed
DRAM

(a) Zipped block as response (b) Uncompressed block as response

Fig. 11. Response from the DRAM.

8 segments and thus contributing to the savings in cache space or (ii) an uncompressed block with no savings in cache space.

Zipped block response from the DRAM: A zipped block in LLC can contain up to 8 blocks, whereas in DRAM it can contain only up to 6 blocks. Thus, a serviced zipped block from DRAM, when indexed using *h2*, may span over two different sets in the cache. In other words, blocks in the serviced zipped block may get mapped to two different tags (either only T_x or T_x and T_{x+1}). Consider a zipped block with address X, that contains a total of six valid blocks in the zipped form, having addresses X, $X + 1$, $X + 2$, $X + 3$, $X + 4$, and $X + 5$. When indexed using *h2*, X and $X + 1$ can get mapped to zip tag T_x and the rest of the addresses can get mapped to zip tag T_{x+1}.

If T_x is already present in the zip tag structure, all the valid blocks that map to T_x are either merged with the existing zipped block in a cache or a new zipped block is allocated. We merge the two zipped blocks only if they share the same base and encoding bits, which implies that all the blocks in the two zipped blocks are compressible with each other. We refer to this step as *merge-or-allocate*. If T_x is not present, a new zipped block is allocated with blocks that map to T_x. If there are no pending requests to blocks that get mapped to T_{x+1}, we do not allocate any further blocks. But, if there are any such pending requests we check if T_{x+1} is present in the cache. If yes, we *merge-or-allocate* the serviced zipped block with the existing zipped block. If T_{X+1} is not present, a new zipped block is allocated with blocks that map to T_{x+1}. When handling both T_x and T_{x+1}, if there are no pending requests for a block that is being copied, we copy the block only if it is not already present in the cache.

To explain this process, we assume that a request has been made to DRAM for block b_6. A zipped block is serviced from the DRAM that contains blocks b_6 to b_{11}. b_6 and b_7 get mapped to a zipped tag, T_1. The remaining blocks (b_8 to b_{11}) get mapped to the consecutive zipped tag, T_2. We allocate blocks b_6 and b_7 (*merge-or-allocate*) as a single zipped block in the LLC. If there are pending requests for any of the other blocks (b_8 to b_{11}), we have two scenarios: (i) T_2 is already present in the cache, in which case we *merge-or-allocate* the blocks with the existing zipped block; or (ii) T_2 is not present in the cache, in which case we allocate a new zipped block (T_2).

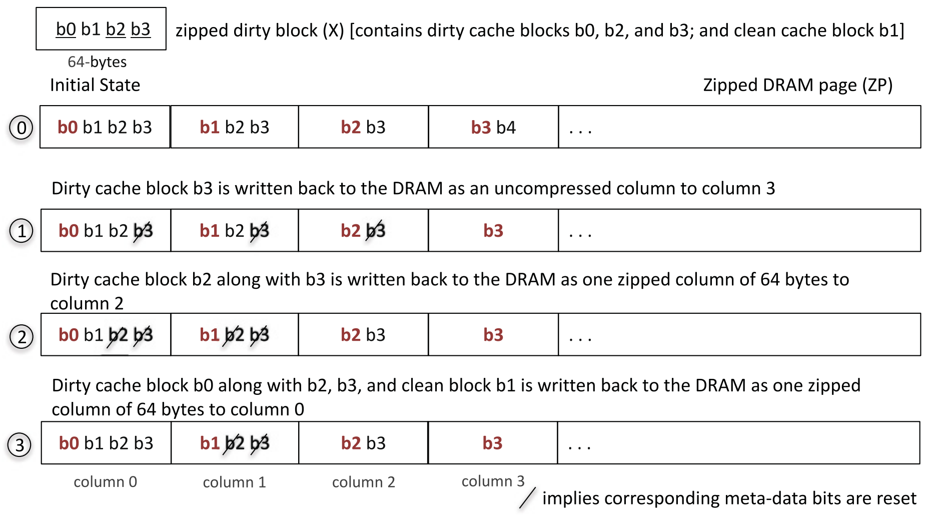

b0 b1 b2 b3
zipped dirty block (X) [contains dirty cache blocks b0, b2, and b3; and clean cache block b1]
64-bytes
Initial State
Zipped DRAM page (ZP)
b0 b1 b2 b3
b1 b2 b3
b2 b3
b3 b4
Dirty cache block b3 is written back to the DRAM as an uncompressed column to column 3
Dirty cache block b2 along with b3 is written back to the DRAM as one zipped column of 64 bytes to column 2
Dirty cache block b0 along with b2, b3, and clean block b1 is written back to the DRAM as one zipped column of 64 bytes to column 0
column 0
column 1
column 2
column 3
implies corresponding meta-data bits are reset

Fig. 12. Zipped dirty block (X) written back to the main memory. The block contains three dirty cache blocks (along with a clean block) evicted from the LLC as a single zipped block.

5.3.1 Handling Writes to the Main Memory. When a zipped block is evicted from LLC, it can contain anywhere between zero to eight dirty blocks. In the case in which the zipped block has no dirty blocks, then it is discarded (as done in the case of a generic cache). If there is even a single dirty block, the entire zipped block is sent to the write buffer (occupying not more than 64B). Note that a zipped dirty block takes up only one write buffer entry irrespective of the number of dirty blocks in it. This saves the write buffer entries. The zipped dirty block is then sent to the DRAM write queue, where it occupies only one entry. This again saves space. If there are eight dirty blocks in every zipped block in the write queue, the write queue can store up to eight times the number of entries when compared to a conventional DRAM. This generates more opportunity for row hits when writing dirty blocks to the memory. When a dirty zipped block is chosen from the write queue to be written to DRAM, the writes of the dirty blocks happen from the rightmost dirty block to the leftmost dirty block in the zipped block.

Figure 12 shows the steps followed when a zipped dirty block (X) is written back to the DRAM. X contains three dirty cache blocks b_0, b_2, and b_3 (along with a clean block b_1) evicted from the LLC as a single zipped block. In the conventional case, there will be three write requests to the uncompressed memory, one each for b_0, b_2, and b_3. In MBZip, ⓪ shows the contents of the zipped DRAM page (ZP) before the write is initiated. The metablock associated with ZP is read either from the metacache or the reserved DRAM space.

(1) In step ①, dirty cache block b_3 is written to its parent column (column 3) as an uncompressed block. This overwrites the previously stored zipped column in column 3. The corresponding valid bit of b_3 is reset in column 0, column 1, and column 2, as these zipped columns no longer contain the updated data of block b_3.
(2) In step ②, dirty cache block b_2 along with b_3 is written to its parent column (column 2) as one zipped column. This overwrites the previously stored data in column 2. Note that after step ②, column 2 contains a zipped column that has valid blocks b_2 and b_3. The corresponding valid bit of b_2 is reset in column 0 and column 1, as these zipped columns no longer contain the updated data of block b_2.

(3) In step ③, dirty cache block b_0—along with b_2, b_3, and clean block b_1—is written to its parent column (column 0) as one zipped column. Column 0 now contains a zipped column that has valid blocks b_0, b_1, b_2, and b_3.

A total of three writes were generated to the memory, which is similar to the conventional case. The zipped columns were updated with new data without generating extra writes to the memory. In general, when a dirty zipped block containing blocks b_x to b_{x+5} (out of which b_{x+4} and b_{x+1} are not dirty) is written back to the memory, b_{x+5} is first written to its parent column, followed by b_{x+3}, b_{x+2}, and b_x.

5.4 Silent Writes

A write to a location in memory may not change the existing data in that location and is thus a redundant write. Such write requests to cache have been termed as silent stores [25]/writes [21]. We explore this observation at the main memory and use the term *silent writes* for such requests. We run each benchmark for 50 billion instructions and collect silent-write statistics at the end of the simulation. We find that, on average, across 21 benchmarks, 9.6% of the writes are silent. More than 15% of the writes are silent in benchmarks such as *bwaves*, *GemsFDTD*, *lbm*, *leslie3d*, *mcf*, *mesa*, *sjeng*, *soplex*, *vortex2*, and *zeusmp*.

In order to identify a silent-write request, we first need to read the existing data from the memory. If the existing data is the same as the new data that is requested to be written, the corresponding write request can be squashed. In such a scenario, we essentially issue one read request (to read the existing data) and no write request. However, if the write request is not silent, we add the overhead of a read request to the existing write request. This overhead can be removed if we can predict that the write request is nonsilent. We observe that there is a strong correlation to a write being silent or nonsilent both across writes made to the same address during the course of the program execution and across writes to consecutive addresses. To exploit this correlation, we propose using a 2b bimodal predictor (indexed using the page addresses) to predict whether a write request is silent. The accuracy of our predictor (4kB structure) is around 94.4%, on average.

In the case of zipped columns in the main memory, exploiting silent writes is advantageous. Assume that a write request is issued for a block b_x. Let us consider that the parent zipped column associated with b_x contains blocks b_x to b_{x+5}. Exploitation of this silent write will be beneficial in the following scenarios :

(1) If a write to b_x is classified by the predictor as silent, a request is issued to read the existing data of b_x. This request reads the associated parent zipped column of b_x. If this zipped column is buffered for a small duration in the controller, any future writes to blocks b_{x+1}, b_{x+2}, b_{x+3}, b_{x+4}, or b_{x+5} can be tested if they are silent or not without issuing any extra reads to the memory. If they turn out to be silent, the corresponding write requests can be simply squashed. In these scenarios, a write request neither reads from nor writes to the main memory.
(2) A write-back block containing only b_x would invalidate b_{x+1} to b_{x+5} in the parent zipped column of b_x. A silent write, on the other hand, will not invalidate these blocks.
(3) Also, a write-back block containing only b_x would invalidate the copy of b_x in all of those previous zipped columns of which b_x is a part. However, in the case of a silent write, such invalidations are also not needed.

Figure 13 shows an example in which identification of silent writes is beneficial. The zipped dirty block X in the write queue contains blocks b_2 and b_3, both of which are dirty. In Figure 13, Ⓐ shows the initial state of the zipped page. In Ⓑ, we show the state of the page if b_2 and b_3 are written,

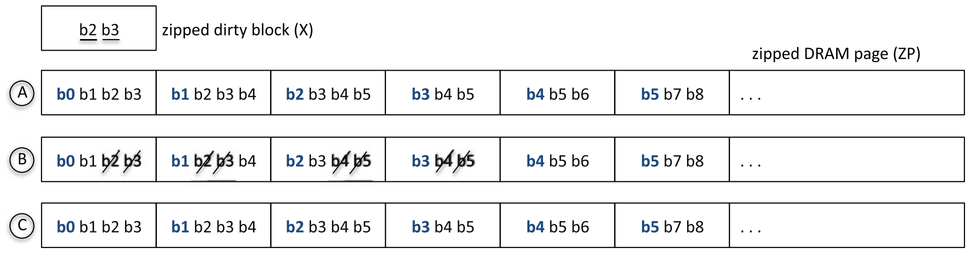


Fig. 13. Zipped dirty block written back to the main memory. b_2 and b_3 are both silent. A: Initial state of the zipped page. B: State of the page after write-back. C: State of the page after write-back, but with silent writes enabled.

without identifying whether the writes are silent. Two sets of invalidations are made : (i) other blocks (b_4 and b_5) in the parent zipped columns of b_2 and b_3 are invalidated; and (ii) copies of b_2 and b_3 in the previous zipped columns, of which they are a part, are invalidated. The invalidations are done by resetting the corresponding valid bits in the metadata associated with the zipped blocks.

In Ⓒ, we show the case in which we check whether the writes are silent : a read is generated to the parent column of b_2. If the write to b_2 is silent, the write is simply squashed. Also, note that the read would actually read a zipped column containing blocks b_2, b_3, b_4, and b_5. Thus, to detect whether b_3 is silent or not, no extra read needs to be generated. If b_3 also turns out to be silent, the write is no longer needed. In this example, when silent writes were identified for the zipped dirty block *X*, only one read and no writes were generated. The parent zipped column of b_2, which has been read, is stored in a buffer. Now, as long as the column resides in the buffer, future write requests for any block in this column (including those for b_4 and b_5) need not generate any read requests to classify them as silent. Also, enabling silent writes removes the unnecessary invalidations that are made to the parent columns of b_0, b_1, b_2, and b_3.

A write request to a zipped page is indexed into the silent predictor and searched in the silent buffer in parallel. If found in the buffer, the write can be tested if it is silent, without generating a read request. If the buffer reports a miss, the prediction is used. A read request is generated only if the predictor classifies the write as silent. Upon read, if the write is silent (right prediction), the write request is simply squashed. If the prediction is wrong, the write request is allowed to update the memory with the new data. The block that is read is stored in zipped form (as read from the memory) in the silent buffer. The buffer is a small (4kB) associative hardware structure. Silent writes can also be tackled at the LLC. Before a cache block is updated in the LLC, it is read to check if the update is silent. The dirty bit is not set if it is silent. This reduces the complexity at the memory controller.

5.5 Hardware Overhead

The metacache and metatable sizes are 68.5kB (64kB data space and 4.5kB tag space), and 8kB, respectively. The silent buffer and silent predictor table sizes are 4.3kB (4kB data space and 0.3kB tag space) and 4kB, respectively. In total, the hardware overhead at the memory controller is 84.8kB. A reserved space of 1.6% of the total memory is required to store the metadata. MBZip employs the same compressor/decompressor unit proposed for BDI, both at the LLC and memory controller. Apart from this, the memory controller should be equipped with a hardware structure to identify silent writes. At the cache, similar to BDI, we use double the number of tags. Half of these tags (zip tags) have 3 fewer tag bits. But, the zip tags also track coherence per block in the zipped

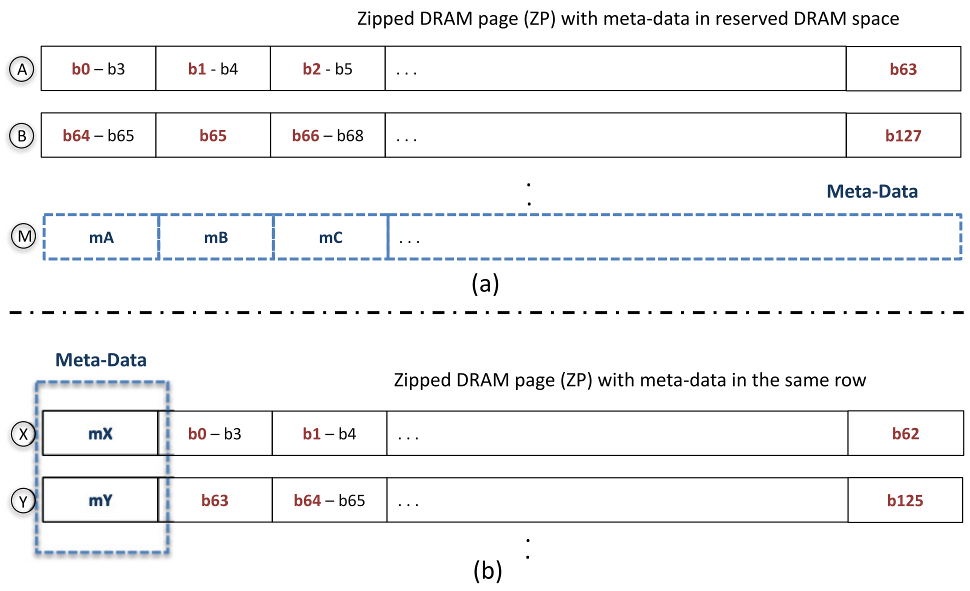


Fig. 14. (a) Address mapping in MBZip-M, in which the metadata associated with a page is stored in a reserved DRAM space. (b) Address mapping in MBZip-M, in which the metadata associated with a page is stored in the same page.

block. A zipped block can contain up to eight blocks. Extra coherence bits are needed to track the same. Thus, a zip tag uses 11/19b (MESI/MOESI coherence protocol) more when compared to a generic/BDI tag. When we consider a 1MB cache with 64B blocks (total number of tags is 16,384, which includes 8,192 zip tags, and 8,192 generic tags), MBZip would need 11kB/19kB extra space at the tag array when compared to the BDI cache.

5.6 Placement of Metadata in MBZip-M

In MBZip-M, the metadata associated with a page is placed in a reserved DRAM space. Figure 14(a) shows this address-mapping scheme. Column *mA* in page *M* stores the metadata associated with page *A*. Page *M* is part of the reserved DRAM space. The address translation is simple, as there is 1:1 mapping of column addresses between conventional and MBZip-M main memory pages. If the metadata for a request is not found in the metacache, a different page needs to be opened to read the metadata and then closed, followed by reading the zipped/uncompressed data from the actual page. There is potential for increase in DRAM page conflicts if the metacache misses are significant. The metadata can be stored in the same page and the simplicity of the memory management unit can still be retained. To reduce page conflicts, MBZip-M can use a different addressing scheme that does not retain the 1:1 mapping with a conventional DRAM page but would still be simple. We present two mechanisms in which the metadata can be stored in the same page.

Figure 14(b) shows the address-mapping scheme in which the metadata associated with a page is stored in the same page. The metadata of page *X* is stored in Column *mX*. The memory controller needs to compute a new address for any column request. The computation is not dependent on the compression pattern, that is, the location of each column in the zipped DRAM is fixed. Each page starts with a column(s) storing the metadata. The address computation logic is fixed. It is given by Equation (1), where *n* represents the address of a parent column in the conventional DRAM page, *p* represents the number of columns in a page, and *q* represents the number of columns in a page with the new address-mapping scheme (without counting the column storing the metadata). The new column address is given by n'.

$$n' = \left\lfloor \frac{n}{q} \right\rfloor * p + (n \,\%\, q) + 1 \tag{1}$$

Figure 14(b) shows a 4kB page with 64B columns. The metadata is 64B and is stored in the first column of every page. p in this case is 64 and q is 63. The other mechanism is to store the metadata in the same page, but instead of storing the displaced column (actual data) in the following page, the displaced column is stored in a reserved DRAM space. A similar approach to handle metadata is proposed in [7]. For example, consider a 4kB page with 64B columns. The metadata associated with the page can either be stored in the starting column (column 0) or at the end (column 63). This results in a block of data ($b63$ of page 0, $b127$ of page 1, and so on) that cannot be accommodated in its respective page. These blocks are stored in the reserved DRAM space. To access the 64th column in every page, the memory controller computes a new address. This address-computation logic is simple and fixed. Note that, in this mechanism, the 64th column cannot be zipped with the last five columns in the page. Any memory request to a zipped page will require a metadata read (if not found in the metacache) and the actual column read. Note that in both address-mapping schemes, there is no strict ordering between the reads as long as the metadata is sent along with the request. In this work, to keep the address mapping simple, we store the metadata in a reserved DRAM space, as shown in Figure 14(a).

5.7 MBZip in Exclusive Caches

On a miss in L2, the next level of the cache hierarchy (LLC) is searched for the requested block. On a miss in LLC, in an exclusive cache, the requested block brought from the memory is inserted only into the L2 cache. In MBZip, the memory might respond with a zipped block that contains the requested block. The zipped block can be allocated in the LLC. The valid bit for the requested block is reset in the zip tag. This is similar to the case in which the requested block is not part of the LLC. The decision to allocate the zipped block at the LLC can be based on the access pattern and the usefulness of other blocks in the zipped block. On LLC hit, the requested block is inserted into the L2 cache and the block in the LLC is invalidated. If the block was part of a zipped block, the valid bit associated with it is reset in the zip tag. On eviction from L2, the cache controller will try to zip the evicted cache block with a zipped block in LLC if they share the same zip tag T_x.

5.8 ECC in MBZip

MBZip-M handles Error-correcting Code memory (ECC) bits in the compressed domain for zipped data. If a zipped column is written to the DRAM, the ECC is computed on the zipped data. When the zipped column is read, the ECC bits are used for error correction in the zipped data. The data, if required, can be decompressed using the error-free zipped data.

6 EXPERIMENTAL EVALUATION

We use the gem5 [9] simulator to evaluate the effectiveness of the proposed techniques. We use benchmarks from SPEC CPU 2000 and 2006 suites [17] [18]. We fast-forward 10 billion instructions, warm up the system for 250 million instructions, and collect statistics for the next 250 million instructions, which is similar to earlier works [29] [39]. Table 1 provides the details of system parameters used in our evaluation. For the baseline, we consider a system that does not implement zipping. We evaluate the benefits of using MBZip-C and MBZip-CM along with the state-of-the-art cache compression technique, BDI, and memory compression technique LCP. For LCP, compressed pages are of pre-determined sizes: 512B, 1kB, 2kB, 4kB, and 8kB. The smallest available page size

Table 1. Parameters of the Simulated System

Processor	4-core and 8-core CMP, 4.7GHz, out-of-order
Fetch/Commit width	8
ROB/LQ/SQ/Issue Queue	192/96/64/64 entries
L1 D/I Cache	32kB, 4-way, 3-cycle latency
LLC	4/8MB for 4/8 cores, 16-way, 30/33 cycle latency
Cache block size	64B
MSHR entries	16, 64/128 at L1, LLC with 4/8 cores
Prefetcher at LLC	Stream prefetcher, with degree = 4 and distance = 64
DRAM Controller	On-chip, open row, 64 read and write queue entries, FR-FCFS scheduler
DRAM Bus	Split-transaction, 800MHz, BL = 8
DRAM	DDR3 1,600MHz (11-11-11), 1/2 channels for 4/8 cores, peak bandwidth = 12.8GB/s

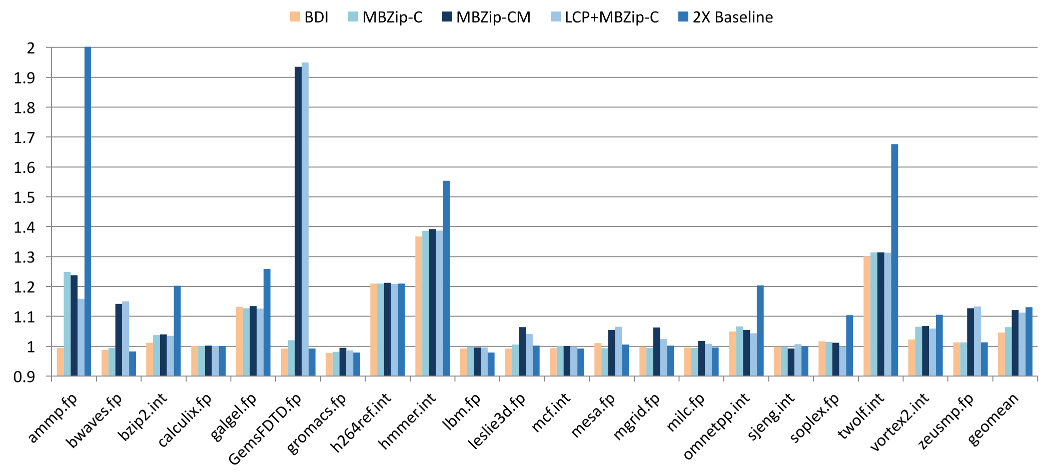


Fig. 15. Performance improvement in terms of IPC with BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C for single-core workloads. Floating-point benchmark names are appended with .fp and integer benchmarks with .int.

that fits the compressed page is chosen. For example, if a page is compressed to 1,408B, then a 2kB page is allocated. We use BDI to compress blocks in LCP. We evaluate MBZip-C with LCP, that is, LCP+MBZip-C. We increase the cache access latency in all four techniques (BDI, MBZip-C, LCP+MBZip-C, and MBZip-CM) by 2 cycles. This is to take into account the decompression latency and the extra tag lookup latency.

6.1 Single-Core Results

Figure 15 shows the improvement in IPC for 21 SPEC benchmarks. We also report the performance improvement that can be gained by doubling the cache size from 1MB to 2MB in a baseline system. On average, doubling the cache capacity results in 13.1% performance improvement. Compared to a baseline system, BDI, MBZip-C, LCP+MBZip-C, and MBZip-CM provide performance improvements of 4.5%, 6.4%, 11.2%, and 12.1%, respectively. For LCP+MBZip-C, we calculate the

Table 2. Benchmark Characteristics

Zip Friendly (ZF)	bwaves, GemsFDTD, mesa, zeusmp, calculix, gromacs, sjeng, leslie3d and mgrid
Cache Sensitive (CS)	bzip2, soplex, omnetpp, ammp, galgel and hmmer
ZF and CS	twolf, vortex2 and h264ref
Neither ZF nor CS	lbm, mcf and milc

Note: Zip Friendly: If more than 20% of the blocks can be zipped into a single block. Cache Sensitive: If the ratio of improvement in performance in terms of IPC by going from 1MB to 2MB LLC is greater than 10%.

percentage of exception blocks in a compressed page. On a metacache miss, in LCP, three memory read requests are required for exception blocks. We refer to the benchmark categories in Table 2.

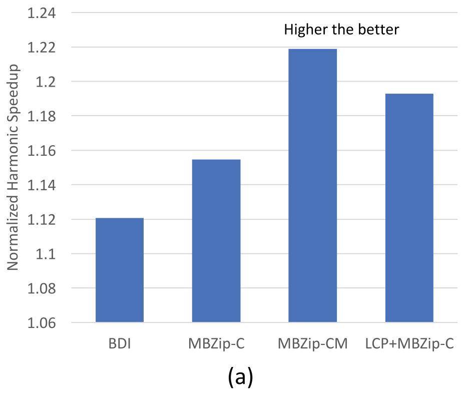
The zip friendly (ZF) benchmarks do not gain in performance either with BDI, MBZip-C, or by doubling the cache capacity (cache insensitive). Some of these benchmarks degrade in performance when compared to the baseline. This is because of the extra cache access latency when compared to the baseline. Benchmarks such as *bwaves*, *GemsFDTD*, *leslie3d*, *mesa*, *mgrid*, and *zeusmp*—though cache insensitive—gain in performance because of MBZip-CM and LCP+MBZip-C. MBZip-CM performs better than LCP+MBZip-C in benchmarks *leslie3d* and *mgrid*, and LCP+MBZip-C is slightly better in *bwaves*, *GemsFDTD*, *mesa*, and *zeusmp*. *calculix* shows a slight improvement in MBZip-CM. A fraction of the performance degradation of *gromacs*—seen with BDI, MBZip-C, and LCP+MBZip-C—is offset in MBZip-CM. *sjeng* does not gain in performance because of the meta-accesses in MBZip-CM.

In general, the cache-sensitive (CS) benchmarks gain in performance by using BDI, MBZip-C, LCP+MBZip-C, and MBZip-CM, and by doubling the cache capacity. These benchmarks have a mixture of compressed blocks (single-cache blocks with less than 64B) and zipped blocks along with the uncompressed ones. In benchmarks *bzip2*, *omnetpp*, and *ammp*, MBZip-C, LCP+MBZip-C, and MBZip-CM perform better when compared to BDI by being able to exploit the small percentage of zipped blocks along with accommodating the compressed blocks. *ammp* has 27.6% of exception blocks and is highly sensitive to cache. *hmmer* gains in performance with all of the techniques. In *hmmer*, more than 85% of the blocks are compressible. In *soplex*, which has mainly compressed blocks, BDI performs slightly better than MBZip-C, LCP+MBZip-C, and MBZip-CM. MBZip-C and MBZip-CM are better than the base case, as they are able to use the space freed up by the compressed blocks. In *galgel*, MBZip-CM performs slightly better than BDI, which, in turn, performs slightly better than MBZip-C.

The ZF-CS benchmarks gain in performance with all five techniques. In *h264ref*, BDI, MBZip-C, and LCP+MBZip-C gain in performance, as much as doubling the cache size in the baseline would gain. MBZip-CM performs slightly better than the four. In *twolf*, MBZip-C, LCP+MBZip-C, and MBZip-CM perform slightly better than BDI. In *vortex2*, both the proposed techniques perform better than BDI.

Benchmarks *lbm*, *mcf*, and *milc*, which are neither CS nor ZF, do not gain much in performance with any of the five techniques. In *lbm*, there is performance degradation in all of the techniques. The slight performance degradation in *mcf* because of BDI is offset in MBZip-C, LCP+MBZip-C, and MBZip-CM. In the case of *milc*, there is slight performance degradation because of BDI and MBZip-C and a slight performance improvement because of LCP+MBZip-C and MBZip-CM.

On average, the bandwidth of BDI, MBZip-C, LCP+MBZip-C, and MBZip-CM is 71.9%, 67.7%, 64.4%, and 63.2% of the baseline, respectively. The LLC miss count is 69.9%, 63.8%, 55.4%, and 52.9% of the baseline, respectively.



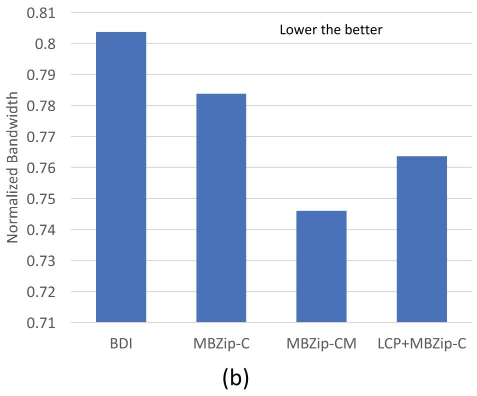


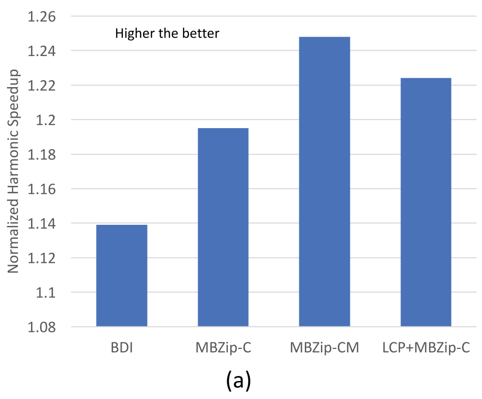
Fig. 16. (a) Normalized harmonic speedup (HS) and (b) bandwidth with BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C for 4-core workloads.

On average, the meta bandwidth is 2.1% of the total bandwidth. **Metatable:** On average, 34.2% of the meta reads are saved because of the metatable. If we use a perfect metatable, 55% of the reads to obtain the metadata associated with a page can be eliminated (unzipped pages). **Metacache:** The average misses in the metacache is 4.2%. **Silent buffer:** The average, silent buffer hits (the number of writes that hit in the silent buffer and are silent by the total number of writes) is 0.3%. In benchmarks like *bwaves*, *GemsFDTD*, *mesa*, and *zeusmp*, more than 20% of the writes hit in the silent buffer and turn out to be silent, thus eliminating the need for a memory access for such writes. **Exception blocks:** The percentage of exception blocks in compressed pages in LCP+MBZip-C is 23.1%.

6.2 Multicore Results

We create 150 4-core and 50 8-core workload mixes by mixing benchmarks based on the characteristics as shown in Table 2. We measure the system performance in terms of weighted speedup (WS) [44] (the higher the better) and harmonic mean of speedups (HS) [28] (the higher the better). We also report the LLC miss count and off-chip bandwidth consumption in terms of GB/s. HS is the reciprocal of the average normalized turn-around time and WS is equivalent to system throughput [14]. HS balances both performance and fairness. The baseline system is chosen accordingly; that is, when reporting performance improvement numbers for the techniques in a 4-core system with prefetcher enabled, the prefetcher is enabled in the baseline system as well. *The numbers reported for bandwidth and LLC are normalized to the corresponding baseline system.*

4-core: Figures 16(a) and 16(b) show the normalized HS and bandwidth, respectively. On average, BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C improve the system performance (HS) by 12.1%, 15.5%, 21.9%, and 19.3%, respectively. In terms of WS, the performance improvement is 9.9%, 12.6%, 20.5%, and 18.6%, respectively. Both BDI and MBZip-C show considerable performance improvements for workloads that have a mix of CS and ZF benchmarks in them. In a few workloads, there is performance degradation (as seen in 1-core) because of the extra access latency. MBZip-CM and LCP+MBZip-C exploit both CS and ZF applications in the workloads and have better performance gains. With MBZip-CM, maximum improvement of 90.1% is seen in the workload *h264ref-GemsFDTD-omnetpp-milc*) and a performance drop of 1.1% in *lbm-mcf-sjeng-calculix*. With LCP+MBZip-C, workloads with applications that have considerable percentage of exception blocks (such as *galgel* (21.9%), *ammp* (27.6%), *omnetpp* (23.5%)), there is an impact on performance



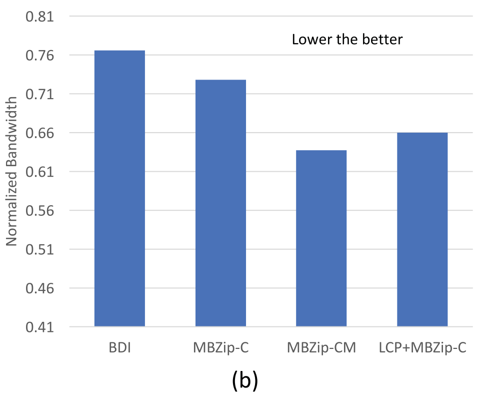


Fig. 17. (a) Normalized harmonic speedup (HS) and (b) bandwidth with BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C for 4-core workloads with stream prefetcher enabled.

and bandwidth, as exceptions require three DRAM accesses on a miss in the metacache. On average, the bandwidth of BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C is 80.4%, 78.4%, 74.6%, and 76.4% of the baseline, respectively. The LLC miss count is 70.7%, 66.8%, 53.9%, and 56.3% of the baseline, respectively.

8-core: For the 8-core system, we use two memory controllers. BDI improves the system performance by 8.8% (HS) and 8.2% (WS) with an average LLC miss count of 77.9% and bandwidth of 85.7%. MBZip-C improves the WS and HS by 10% and 11.4%, respectively, with an average LLC miss count of 75.9% and bandwidth of 84.8%. LCP+MBZip-C improves the system performance by 16.1% (HS) and 15.6% (WS) with an average LLC miss count of 66.1% and bandwidth of 83.3%. MBZip-CM outperforms the three techniques. It improves the system performance by 17.8% (HS) and 17.6% (WS). On average, the bandwidth of MBZip-CM is 81.9% of the baseline and the LLC miss count is 63.8%.

4-core with stream prefetcher enabled: We use a stream prefetcher with degree 4 and distance 64. Figures 17(a) and 17(b) show the normalized HS and bandwidth, respectively. BDI improves the HS and WS by 13.9% and 10%, respectively. MBZip-C performs fairly better than BDI in this system, with performance improvement of 19.5% (HS) and 13.9% (WS). Since a stream prefetcher brings consecutive blocks into the cache, there is more potential to zip blocks together. LCP+MBZip-C improves the HS and WS by 22.4% and 17.1%, respectively. MBZip-CM improves the system performance further. It provides a performance improvement of 24.8% (HS) and 19.1% (WS). On average, the bandwidth of BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C is 76.6%, 72.8%, 63.7%, and 66.0% of the baseline, respectively. The LLC miss count is 67.4%, 62.8%, 55.1%, and 56.8% of the baseline, respectively.

4-core with 3-levels of cache: We evaluate MBZip for a 3-level cache hierarchy. MBZip-CM remains effective for a 3-level cache, with an average performance improvement of 38.6% (HS) and 28.6% (WS). BDI, MBZip-C, and LCP+MBZip-C improve the system performance (HS/WS) by 18%/13.1%, 33.7%/23.1%, and 36.8%/27%, respectively. On average, the bandwidth of BDI, MBZip-C, MBZip-CM, and LCP+MBZip-C is 79.8%, 69.2%, 59.6%, and 62.6% of the baseline, respectively. The LLC miss count is 69.5%, 58.5%, 52.8%, and 54.2% of the baseline, respectively.

7 RELATED WORK

We briefly discuss related works in cache, memory, and link compression for both CPUs and GPUs. We conclude the section with a discussion on related works in silent writes.

At the cache: Yang et al. observe that a significant amount of values accessed by cache belong to a small set of frequently occurring values and propose Frequent Value Compression (FVC) [47]. FVC incurs the overhead of profiling needed to identify these frequent values. Alameldeen and Wood [3] identify seven frequently occurring compressible data patterns. They propose Frequent Pattern Compression (FPC) [4], which exploits these patterns to compress a cache block word by word. Chen et al. [10] propose C-Pack, which compresses frequently appearing words by using compact encodings. They also try to combine pairs of compressed blocks into a single block. Each compressed block in the pair has its own tag and size fields associated with it. Arelakis et al. [6] propose SC^2, which uses Huffman encoding to compress a cache block. The decompression latency of SC^2 is ten cycles after using a pipelined decompression engine.

Sardashti et al. [42] propose a decoupled compressed cache (DCC) that exploits spatial locality to increase the effective cache capacity. DCC uses super block tags to reduce tag overhead. However, it requires extra metadata to hold backward pointers to read an entire block. Sardashti et al. [40] propose the use of a skewed compressed cache (SCC), which eliminates the metadata required by DCC and uses direct data-tag mapping. SCC uses skewed associative mapping (to reduce conflict misses), which adds complexity and overhead. The same authors also propose yet another compressed cache (YACC) [41], which retains the benefits of SCC but eliminates the need for skewing. YACC compacts cache blocks that have the same compression factor. In the recently proposed Dictionary Sharing (DISH) ([33]), the authors observe that this can potentially result in unused data space. Their technique builds on YACC and does not change the cache layout proposed in YACC. A dictionary is shared among multiple contiguous compressed blocks that get mapped to the same set in the YACC organization, thus reducing space wastage. Both YACC and DISH use superblock tags to track blocks. For applications that exhibit spatial locality but are not compression friendly, the effective set associativity available in YACC-based cache organization is less when compared to a generic cache. MBZip has a combination of both conventional and zip tags. It does not depend on skewing to reduce conflict misses. If the application exhibits spatial locality and is compression friendly, zip tags are used to track the cache blocks. If the cache blocks in the application are not compressible, conventional tags are used. Nguyen and Wentzlaff [30] propose MORC, which builds on a log-based cache organization. MORC selects cache blocks that are filled into the cache around the same time as candidates for compression and tries to compress them. They also try to compress cache tags to further save space. In a log-based cache organization, the average decompression latency can be longer when compared to a set-based cache. The previous cache blocks in a log need to be decompressed before decompressing the requested block. This might potentially harm the performance of a single-stream application. In MBZip, the decompression latency of a block is independent of the blocks present in the zipped block. MBZip uses a set-based cache organization. In [5], the authors explore a hybrid compression scheme (HyComp), in which the best performing compression scheme is chosen based on the data type. They use heuristics to predict the data types. The authors also propose a novel compression scheme (FP-H) to exploit value locality in floating-point values (data types that have predefined semantic value fields in general). In [48], the authors explore compression for DRAM caches. They observe that the performance improvement is higher if DRAM cache is designed to provide higher bandwidth and not just optimized for capacity. They dynamically choose between spatial indexing (similar to zip indexing in MBZip) and traditional set indexing depending on the compressibility of the data.

Pekhimenko et al. [36] propose compression-aware management policies (CAMP) for caches. The importance (value) of a block is calculated using the compressed block size, which, in turn, is used both in the minimum-value eviction (MVE) scheme and a size-based insertion policy (SIP). In hot-cacheline prediction for early decompression (HoPE) [34], the authors use the compressed block size as an indicator to enable early decompression of cache blocks. Their techniques identify frequently used cache lines and use that information to minimize the read hit decompression latency in compressed cache designs. Gaur et al. [15] observe that there can be negative interactions between cache compression and replacement policies, that is, when replacement policies are modified to accommodate compression, it can lead to suboptimal replacement decisions that can hurt performance gains. To avoid these negative interactions, the authors propose Base-Victim Compression (BVC), which guarantees that all cache blocks that would have existed in an uncompressed cache at any point in the program's execution would remain in the compressed cache. BVC does not modify the data array. This is achieved by always associating two tags with one physical way. This overcomes the drawback of compressed cache architectures that require segments from multiple ways to be activated to read a single cache block, which, in turn, increases the power and area.

Instead of exploiting data redundancy for compression, in [45], the authors present a technique to detect duplicate cache blocks and store only one copy of the data. This data can be accessed through multiple addresses in the cache, that is, multiple tags can potentially point to a single data location. The technique results in an increase in effective cache capacity.

At the main memory: IBM MXT [2] uses a compressed physical memory. To perform address translation, a lookup table (compression translation table) is used, which is kept uncompressed at a reserved location in the physical memory. To offset the increased memory access latency, a large (32MB) uncompressed L3 cache managed at 1kB block granularity is used. In [13], a memory page is logically arranged as a hierarchical structure with a small slack at each level to accommodate changes in the compressed block size upon write-back from the cache. A DMA-engine is used to move the page when there is a page overflow. The page table needs to be modified to accommodate size vectors that contain the encoded slack thresholds and the sizes for the subpages and page. Pekhimenko et al. [37] propose linearly compressed pages (LCPs) that use fixed size for compressed columns within a page to simplify main memory address computation. Further, LCP improves performance and reduces bandwidth by transferring multiple compressed cache blocks using a single memory request. LCP requires OS support. A write-back to the memory can cause an overflow that requires the OS to migrate the entire page to a larger page. MBZip does not need any changes to address computation logic and works seamlessly with the conventional/existing logic. An overflow scenario is not possible in MBZip. In Buri [49], the authors propose a hardware technique to manage allocation/reallocation of compressed pages. The OS support required is minimal. Similar to LCP, compressed blocks are stored in fixed-sized slots. To read an uncompressed block, a pointer is first read from its slot, followed by a second memory access to dereference the pointer and read the data.

In COP [32], the authors propose a technique that uses block-level compression to make room for ECC check bits in DRAM. Shafiee et al. [7] propose MemZip, which combines compression with rank subsetting, and propose novel data layouts at the DRAM. In MemZip, the space saved due to compression is not used to pack other blocks of data. Instead, the goal of MemZip is to use the space saved to implement (store) error-correcting codes (ECC) or energy-efficient data encodings (Data Bus Inversion technique (DBI)). A read to MemZip would fetch only a single block of data. MemZip uses rank-subset architecture as its baseline. MBZip has the potential to read multiple blocks of data on a single read and can work with the baseline DRAM architecture. Data deduplication is tackled in HICAMP memory architecture [11]. The memory is divided into fixed-size lines that store unique content and is immutable during its lifetime. The memory system can either read a line by its ID or look up a line by its content. When a new line is written, it is checked to see if the content already exists in the memory and inserted only if the data does not already exist. This improves both cache and DRAM utilization. To incorporate the architecture requires changes to the virtual memory framework and programming model. Arjun et al. [12] use compression at the main memory to reduce energy consumed per read/write. The effective memory capacity is kept the same as that of the memory without compression. The compression metadata is encoded in the field reserved for ECC and thus does not require separate metadata storage to track the compressibility of every block. In [20], the authors propose Frugal ECC (FECC), in which the ECC information is stored along with the data. A single read would then be able to fetch both the data and the ECC bits without using redundant memory chips. Data is compressed and the leftover space is used to store the ECC information. The overflow data is stored in a separate block because of insufficiently compressed blocks. A cache line is stored evenly distributed among different chips in the DRAM. Due to compression, banks in some of these chips can contains rows that do not hold any valid data. In [27], the authors propose a compression-aware refresh mechanism in which only those rows that contain valid data are refreshed. In [31], the authors propose a compression-expansion (CompEx) coding scheme that integrates compression with expansion coding to reduce energy and latency and improve lifetime in Multilevel/Triple-level cell (MLC/TLC) nonvolatile memories (NVMs). In [26], a cache block is compressed and stored only in the soft-bit region of the MLC cells, thus requiring only one step to access the block. The leftover hard-bit region is used to store compressed cache blocks in memory-intensive applications. Hallnor and Reinhardt [16] propose a unified compressed memory hierarchy that encompasses both LLC and main memory. At the LLC, they build on the previously proposed Indirect Index Cache (IIC) to accommodate compression (IIC with compression (IIC-C)). IIC-C adds extra subblock pointers to the tag. The size of a tag entry is increased by 6B to 8B for 4 sub-blocks. They couple the compressed cache with main memory compression. A MXT-like memory compression scheme is used. In [35], the authors observe that, though compression reduces the amount of data that is transferred, there is an increase in the number of wires that switch from 1 to 0 or 0 to 1 (bit toggles). The increase in number of toggles results in higher dynamic energy consumption by on-chip and off-chip interconnects. Two toggle-aware compression techniques are proposed to mitigate this problem.

Though graphic processing units (GPUs) offer very high off-chip memory bandwidth, the performance of many GPU applications is still bandwidth bound. Kim et al. [19] propose Bit-Plane Compression (BPC), a novel compression algorithm to increase the effective memory bandwidth. BPC employs a data transformation technique, Delta-BitPlane-XOR (DBX), to improve the compressibility of data. BPC focuses on main memory link compression to address the off-chip bandwidth concerns in GPGPUs. In [43], the authors explore microarchitectural techniques to support both lossless and lossy compression of data transferred through the GPU memory I/O links. Data is stored in compressed format in the GPU's off-chip memory. Significant performance improvements are gained in memory-bound workloads by targeting link compression. Based on Huffman encoding, an entropy encoding–based memory compression (E^2MC) for GPUs is proposed in [22]. E^2MC targets memory bandwidth to achieve higher performance and energy efficiency. Lee et al. [23] propose warped compression, which employs BDI for register compression in GPUs to reduce both dynamic and leakage power in the system. Warped-compression architecture is also coupled to support compressed execution, that is, some instructions are processed without decompressing the operand values to further save energy. Vijaykumar et al. [46] propose a core-assisted bottleneck acceleration (CABA) framework for GPUs, in which assist warps are automatically generated to perform specific tasks that speed up application execution. Instead of a hardware-based implementation, CABA uses assist warps to enable flexible data compression in the memory hierarchy.

Silent writes: Lepak and Lipasti [24] show that 20% to 68% of the stores are silent. In [25], they study the problem associated with detecting silent stores. They exploit both temporal and spacial locality in a processor's load and store queues to propose a technique to identify such stores with low implementation cost. At the cache, they use idle read access ports to detect silent stores. Kishani et al. [21] propose a mechanism to use the existing parity code as block signature to detect silent writes at the L2. They avoid both, writing the cache block and updating the ECC.

Awad et al. [8] propose a silent shredder for nonvolatile memory (NVM) based on the observation that a large percentage of all memory writes can be due to data shredding in operating systems. A silent shredder eliminates such writes by manipulating initialization vectors.

8 CONCLUSION

In this article, we proposed a new technique called MBZip that exploits data patterns spread across multiple consecutive blocks for compression, both at the LLC and main memory. At the cache, MBZip uses a combination of generic and zip tags to accommodate cache blocks of various forms: uncompressed, compressed, and zipped. At the DRAM, MBZip does not need any changes to the address computation logic and works seamlessly with the conventional/existing logic. MBZip is a synergistic mechanism that coordinates zipped blocks at the LLC and DRAM. Overall, MBZip-CM is shown to improve system performance by 21.9% and 17.8%, for 4- and 8-core systems, respectively. We conclude that MBZip provides a promising way to increase system performance.

REFERENCES

[1] Intel 64 and ia32 architecture software developer's manuals. 2017. Retrieved October 24, 2017 from http://www.intel.com/products/processor/manuals/.

[2] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, Jr., C. O. Schulz, L. M. Herger, and T. B. Smith. 2001. Memory expansion technology (MXT): Software support and performance. *IBM Journal of Research and Development* 45, 2 (2001), 287–302.

[3] A. R. Alameldeen and D. A.Wood. 2004. Frequent pattern compression: A significance-based compression scheme for l2 caches. In Technical Report 1500, University of Wisconsin-Madison, Computer Sciences Department.

[4] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, Washington, DC, USA, 212–. http://dl.acm.org/citation.cfm?id=998680.1006719.

[5] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, USA, 38–49. DOI: http://dx.doi.org/10.1145/2830772.2830823

[6] Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*. IEEE Press, Piscataway, NJ, USA, 145–156. http://dl.acm.org/citation.cfm?id=2665671.2665696.

[7] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'14)*.

[8] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, USA, 263–276. DOI: http://dx.doi.org/10.1145/2872362.2872377

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Computer Architecture News* 39, 2, 1–7. DOI: http://dx.doi.org/10.1145/2024716.2024718

[10] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration Systems* 18, 8, 1196–1208. DOI: http://dx.doi.org/10.1109/TVLSI.2009.2020989

[11] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: Architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, USA, 287–300. DOI: http://dx.doi.org/10.1145/2150976.2151007

[12] A. Deb, P. Faraboschi, A. Shafiee, N. Muralimanohar, R. Balasubramonian, and R. Schreiber. 2016. Enabling technologies for memory compression: Metadata, mapping, and prediction. In *IEEE 34th International Conference on Computer Design (ICCD'16)*. 17–24. DOI: http://dx.doi.org/10.1109/ICCD.2016.7753256

[13] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Washington, DC, USA, 74–85. DOI: http://dx.doi.org/10.1109/ISCA.2005.6

[14] Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3, 42–53. DOI: http://dx.doi.org/10.1109/MM.2008.44

[15] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. 2016. Base-victim compression: An opportunistic cache compression architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, Piscataway, NJ, USA, 317–328. DOI: http://dx.doi.org/10.1109/ISCA.2016.36

[16] Erik G. Hallnor and Steven K. Reinhardt. 2005. A unified compressed memory hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE Computer Society, Washington, DC, USA, 201–212. DOI: http://dx.doi.org/10.1109/HPCA.2005.4

[17] John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7, 28–35. DOI: http://dx.doi.org/10.1109/2.869367

[18] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4, 1–17. DOI: http://dx.doi.org/10.1145/1186736.1186737

[19] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, Piscataway, NJ, USA, 329–340. DOI: http://dx.doi.org/10.1109/ISCA.2016.37

[20] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. 2015. Frugal ECC: Efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, NY, USA, Article 12, 12 pages. DOI: http://dx.doi.org/10.1145/2807591.2807659

[21] Mostafa Kishani, Amirali Baniasadi, and Hossein Pedram. 2011. Using silent writes in low-power traffic-aware ECC. In *Proceedings of the 21st International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization, and Simulation (PATMOS'11)*. Springer, Berlin,180–192. http://dl.acm.org/citation.cfm?id=2045364.2045383.

[22] S. Lal, J. Lucas, and B. Juurlink. 2017. E2MC: Entropy encoding based memory compression for GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 1119–1128. DOI: http://dx.doi.org/10.1109/IPDPS.2017.101

[23] S. Lee, K. Kim, G. Koo, H. Jeon, M. Annavaram, and W. W. Ro. 2017. Improving energy efficiency of GPUs through data compression and compressed execution. *IEEE Transactions on Computers* 66, 5, 834–847. DOI: http://dx.doi.org/10.1109/TC.2016.2619348

[24] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, USA, 182–191. DOI: http://dx.doi.org/10.1145/339647.339678

[25] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. ACM, New York, NY, USA, 22–31. DOI: http://dx.doi.org/10.1145/360128.360133

[26] L. Liu, P. Chi, S. Li, Y. Cheng, and Y. Xie. 2017. Building energy-efficient multi-level cell STT-RAM caches with data compression. In *22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. 751–756. DOI: http://dx.doi.org/10.1109/ASPDAC.2017.7858414

[27] Wenjie Liu, Ping Huang, Kun Tang, Ke Zhou, and Xubin He. 2016. CAR: A compression-aware refresh approach to improve memory performance and energy efficiency. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*. ACM, New York, NY, USA, 373–374. DOI: http://dx.doi.org/10.1145/2896377.2901498

[28] K. Luo, J. Gummaraju, and M. Franklin. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*. 164–171.

[29] R Manikantan, Kaushik Rajan, and R. Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, USA, 428–439. http://dl.acm.org/citation.cfm?id=2337159.2337208.

[30] Tri M. Nguyen and David Wentzlaff. 2015. MORC: A manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, USA, 76–88. DOI: http://dx.doi.org/10.1145/2830772.2830828

[31] Poovaiah M. Palangappa and Kartik Mohanram. 2017. CompEx++: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVMs. *ACM Transactions on Architecture and Code Optimization* 14, 1, Article 10, 30 pages. DOI: http://dx.doi.org/10.1145/3050440

[32] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. 2015. COP: To compress and protect main memory. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, USA, 682–693. DOI: http://dx.doi.org/10.1145/2749469.2750377

[33] B. Panda and A. Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12. DOI: http://dx.doi.org/10.1109/MICRO.2016.7783704

[34] Jaehyun Park, Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Vinson Young, Junghee Lee, and Jongman Kim. 2017. HoPE: Hot-cacheline prediction for dynamic early decompression in compressed LLCs. *ACM Transactions on Design Automation of Electronic Systems* 22, 3, Article 40, 25 pages. DOI: http://dx.doi.org/10.1145/2999538

[35] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A case for toggle-aware compression for GPU systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 188–200. DOI: http://dx.doi.org/10.1109/HPCA.2016.7446064

[36] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 51–63. DOI: http://dx.doi.org/10.1109/HPCA.2015.7056021

[37] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, USA, 172–184. DOI: http://dx.doi.org/10.1145/2540708.2540724

[38] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, USA, 377–388. DOI: http://dx.doi.org/10.1145/2370816.2370870

[39] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, USA, 57–68. DOI: http://dx.doi.org/10.1145/2000064.2000073

[40] Somayeh Sardashti, André Seznec, and David A. Wood. 2014. Skewed compressed caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE Computer Society, Washington, DC, USA, 331–342. DOI: http://dx.doi.org/10.1109/MICRO.2014.41

[41] Somayeh Sardashti, André Seznec, and David A. Wood. 2016. Yet another compressed cache: A low cost yet effective compressed cache. In *ACM Transactions on Architecture and Code Optimization.* ACM.

[42] Somayeh Sardashti and David A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, USA, 62–73. DOI: http://dx.doi.org/10.1145/2540708.2540715

[43] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, USA, 325–334. DOI: http://dx.doi.org/10.1145/2370816.2370864

[44] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. ACM, New York, NY, USA, 234–244. DOI: http://dx.doi.org/10.1145/378993.379244

[45] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-level cache deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. ACM, New York, NY, USA, 53–62. DOI: http://dx.doi.org/10.1145/2597652.2597655

[46] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, USA, 41–53. DOI: http://dx.doi.org/10.1145/2749469.2750399

[47] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. ACM, New York, NY, USA, 258–265. DOI: http://dx.doi.org/10.1145/360128.360154

[48] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2017. DICE: Compressing DRAM caches for bandwidth and capacity. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, USA, 627–638. DOI: http://dx.doi.org/10.1145/3079856.3080243

[49] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. 2015. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Transactions on Architecture and Code Optimization* 12, 3, Article 31, 24 pages. DOI: http://dx.doi.org/10.1145/2808233

Received September 2016; revised August 2017; accepted October 2017