MemSZ: Squeezing Memory Traffic with Lossy Compression

Citation for the original published paper (version of record):
Eldstål-Ahrens, A., Sourdis, I. (2020). MemSZ: Squeezing Memory Traffic with Lossy Compression. Transactions on Architecture and Code Optimization, 17(4). http://dx.doi.org/10.1145/3424668

MemSZ: Squeezing Memory Traffic with Lossy Compression

ALBIN ELDSTÅL-AHRENS and IOANNIS SOURDIS, Chalmers University of Technology, Sweden

This article describes Memory Squeeze (MemSZ), a new approach for lossy general-purpose memory compression. MemSZ introduces a low latency, parallel design of the Squeeze (SZ) algorithm offering aggressive compression ratios, up to 16:1 in our implementation. Our compressor is placed between the memory controller and the cache hierarchy of a processor to reduce the memory traffic of applications that tolerate approximations in parts of their data. Thereby, the available off-chip bandwidth is utilized more efficiently improving system performance and energy efficiency. Two alternative multi-core variants of the MemSZ system are described. The first variant has a shared last-level cache (LLC) on the processor-die, which is modified to store both compressed and uncompressed data. The second has a 3D-stacked DRAM cache with larger cache lines that match the granularity of the compressed memory blocks and stores only uncompressed data. For applications that tolerate aggressive approximation in large fractions of their data, MemSZ reduces baseline memory traffic by up to 81%, execution time by up to 62%, and energy costs by up to 25% introducing up to 1.8% error to the application output. Compared to the current state-of-the-art lossy memory compression design, MemSZ improves the execution time, energy, and memory traffic by up to 15%, 9%, and 64%, respectively.

CCS Concepts: • **Computer systems organization** → **Architectures**; **Processors and memory architectures**;

Additional Key Words and Phrases: Approximate computing, memory compression, lossy compression

ACM Reference format:
Albin Eldstål-Ahrens and Ioannis Sourdis. 2020. MemSZ: Squeezing Memory Traffic with Lossy Compression. *ACM Trans. Archit. Code Optim.* 17, 4, Article 40 (November 2020), 25 pages.
https://doi.org/10.1145/3424668

1 INTRODUCTION

Memory bandwidth is a critical resource in modern systems and has an increasing demand [50]. The large number of on-chip cores and specialized accelerators improves the potential processing throughput but also calls for higher data rates. In addition, new emerging data-intensive applications further increase memory traffic [4, 5, 47]. On the other hand, memory bandwidth is pin limited [4, 65] and power constrained [45] and is therefore more difficult to scale [50]. More expensive, 3D-stacked DRAM technologies alleviate the bandwidth problem, but have capacity and power limitations [45].

This work is supported by the Swedish Research Council (Contract No. 2012-4924) under the ACE project.

Authors' addresses: A. Eldstål-Ahrens and I. Sourdis, Chalmers University of Technology, Rännvägen 6, Gothenburg, Sweden; emails: {eldstal, sourdis}@chalmers.se.

1544-3566/2020/11-ART40
https://doi.org/10.1145/3424668

One way to alleviate the memory bandwidth pressure is to use it more efficiently by compressing the transferred data. Data can then be transferred between the main memory and the processor chip in a compressed form consuming less bandwidth and reducing energy cost. Commercial examples of architectures that use memory compression are graphics processing units (GPUs) [1]. GPUs, however, use application-specific compression, e.g., applied to texture and color data [12], and therefore are not applicable to other datasets. With a few exceptions such as the above, which is, however, not general purpose, hardware memory compression is mostly lossless offering limited compression ratio. Current state-of-the-art, lossless memory compression techniques achieve on average a 2:1 to 4:1 compression ratio [40]. However, some classes of applications, e.g., multimedia, scientific, and forecasting, may allow for more aggressive compression as they inherently tolerate approximations in parts of their data [10, 34] without introducing significant error.

In the past, the performance of memory subsystems has been improved for approximation-tolerant applications. Load value prediction was used to avoid fetching requested data and improve memory latency [53, 62, 63], but has difficulties capturing irregular data variations. Approximate deduplication of individual cache lines increased cache capacity [27]; however, multiple values need to match in the order dictated by their cache line position. Truncating the least significant bits of values was proposed as a form of lossy compression [6, 17, 38, 39, 57], but offered limited compression ratio. Recently, our previously proposed Approximate Value Reconstruction (AVR) design used downsampling as a general-purpose—as opposed to application-specific—method for compressing data that can be approximated [15]. In order to offer a high compression ratio, AVR compressed large memory blocks of 16 cache lines to a single cache line "summary" and any outlier values. However, compressing multiple cache lines together causes the loss of random access to a single line within a block. Then, accessing a cache line in memory would require accessing the entire block. AVR reduces this overhead by (i) co-locating compressed memory blocks and uncompressed cache lines in the Last Level Cache (LLC), as well as by (ii) utilizing the empty (due to compression) memory space to evict LLC cache lines uncompressed. It further keeps track of badly compressing blocks to avoid frequent unsuccessful attempts.

This work describes MemSZ, a new more aggressive general-purpose lossy memory compression approach. MemSZ is based on the Squeeze (SZ) compression [11], a very effective, although sequential, algorithm introduced for compressing check-pointed data transferred between memory and disk. MemSZ introduces a new parallel version of SZ that is able to compress/decompress a block of n values in $O(\sqrt{n})$ time, rather than in $O(n)$, in practice reducing the compression latency by 50× for the particular block size used. This low latency design of our compressor is then applied to an improved AVR design, which further offers better control of approximation error and a more efficient LLC replacement policy. Finally, in order to avoid AVR's LLC modifications, we apply the MemSZ compression to a system with a 3D-stacked DRAM cache with a line size that matches the granularity of the compressed memory blocks. As a result, both MemSZ designs offer significantly better compression ratio without increasing approximation error, in effect, reducing memory traffic and improving system performance and energy efficiency.

Concisely, the contributions of this article are the following:

- A new low latency parallel version of the SZ lossy compression algorithm that offers substantially higher compression ratio than previous lossy memory compression techniques.
- An extension of the existing AVR architecture, with better cache replacement policy and additional error control mechanisms.
- A new architecture that combines memory compression with a 3D-stacked DRAM to more efficiently handle compression blocks.

The remainder of this article is organized as follows. Section 2 discusses background and related work. Section 3 describes the proposed MemSZ architecture. Section 4 presents our evaluation results and Section 5 draws our conclusions.

2 BACKGROUND AND RELATED WORK

This section provides related work on compression and approximate computing, as well as background knowledge related to the SZ compression algorithm and the AVR architecture, both used in the proposed MemSZ approach.

2.1 Related Work

Prior work on memory and cache compression is presented and subsequently an overview is provided on approximate computing techniques that improve the performance of memory systems.

Memory Compression: There is a plethora of lossless memory compression techniques that improve memory capacity and bandwidth utilization. Various compression algorithms are used, such as dictionary-based [8], exploiting frequent patterns or zero-value blocks [13], and more recently similarities of words at the same bit position [40]. However, lossless solutions typically have a limited compression ratio, achieving between 2× and 4×. This is substantially lower than in our work (4–16×). In general, lossless compression is orthogonal to our lossy approach as it can be used to compress data that are not approximated, or even on top of the MemSZ lossy compression. Another aspect is the data placement in memory. Some approaches compact compressed data in memory to improve capacity [48]. Others, like in our work, avoid data compaction, allocating the worst-case storage required for the uncompressed data and focus only on memory bandwidth [29, 59]. Finally, managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [21, 36]. MemSZ uses a metadata table and a cache of it, as in [48], which is updated with the translation lookaside buffer (TLB) and adds a few bytes of bandwidth overhead at every TLB miss; still, techniques like Attaché could be used to further reduce the metadata cost [36].

The above memory compression schemes are lossless. Until recently, lossy compression was limited to application-specific purposes, i.e., in GPUs [12, 41], or truncating bits of individual values [6, 17, 38, 39, 57], which offered limited compression ratio (2:1 to 4:1).

The first effort departing from the above offering more aggressive (up to 16:1) general-purpose lossy memory compression was AVR [15]. The AVR architecture is discussed in more detail in Section 2.2 and is extended in MemSZ by a more powerful compressor, a mechanism to keep track of the approximation error, and a more efficient LLC replacement. In addition, MemSZ explores using a DRAM cache with large lines to avoid storing compressed blocks on the processor chip.

Cache Compression: Lossless compression has been applied to caches, too. Besides the issues of encoding and compaction of variable size blocks [56], the compression and decompression latency constraints are tighter compared to memory compression. In the past, cache compression has been supported in various ways, for instance using value-centric caches [7]. Compaction of compressed cache blocks has been tackled using decoupled super-blocks and sub-blocks [32, 55, 56], or super-blocks without decoupling tag and data arrays [46]. These designs employ compression methods such as Base-Delta-Immediate (BDI) [24] on single cache blocks, or DISH [18], which allow a small number of adjacent blocks to share a dictionary. The replacement policy of compressed caches has also been investigated, weighing cache blocks by their expected utility compared to their size [25, 31]. By replacing the data that is least likely to be useful in the near future, the efficacy of the compressed cache can be increased. In general, cache compression cannot reduce memory traffic as it compresses single cache lines separately, rather than larger memory blocks of consecutive cache lines as performed by MemSZ. Consequently, as opposed to MemSZ, cache compression techniques applied to the LLC cannot reduce the number of memory accesses and hence cannot reduce memory traffic.

Approximate Computing: Large classes of applications are inherently tolerant to approximations [10]. This enables a tradeoff between the quality of their results and their performance and energy efficiency. This tradeoff is exploited by various approximate computing techniques, some of them targeting the aforementioned memory bottlenecks in a lossy manner.

Approximate load value prediction techniques reduce memory latency by providing a predicted value substantially faster than fetching the actual one from memory [53, 62, 63]. They may further improve memory bandwidth utilization by not always bringing the actual values at all. Value prediction techniques speculate that the values loaded by the same instruction may be identical or differ by a stride. However, this does not capture any irregular variance of data such as the variance in an image where neighboring pixels may have similar values but may not necessarily differ by a fixed stride. Approximate load value prediction is applied near the core (in parallel with the L1 cache) and is therefore orthogonal to the proposed MemSZ compression of memory traffic. Another fundamental difference compared to MemSZ and in general compared to compression is that load value prediction techniques aim primarily at reducing load latency rather than memory bandwidth because in the end they do fetch the precise values from memory for error checking.

Reducing the precision of floating point [6, 38, 57] and fixed point [17, 39] numbers has been used to alleviate the memory bandwidth bottleneck of various approximation-tolerant applications, thereby improving performance and energy efficiency. However, the compression ratio is still limited between 2:1 and 4:1 despite the loss of precision as these approaches do not exploit inter-value similarities to compress data. Closer to MemSZ, software techniques for lossy compression have been proposed, but have high complexity and latency and as a consequence cannot be used directly for memory compression [11, 61].

Approximate, lossy compression has been applied to caches, too. Doppelgänger deduplicates similar cache lines to compress data [27]. The subsequent Bunker cache design speculates similarities between cache lines solely based on their addresses without looking at their contents, proposing a less intrusive cache design but achieving lower compression ratio than Doppelgänger [54]. Both designs exploit similarities between cache lines. However, similar values need to have the same offset within their cache lines in order to match, which restricts deduplication opportunities.

2.2 Background

Next, we discuss the two primary approaches MemSZ is based on. First, AVR is presented—the lossy memory compression scheme extended by MemSZ. Subsequently, SZ is described—a lossy compression algorithm introduced for software compression of checkpoints, which is parallelized in this work to support the MemSZ memory compression scheme.

2.2.1 AVR. AVR compresses in a lossy way parts of the application data, which can tolerate approximation. In general, AVR adds a compressor and decompressor between the LLC and the memory controller of a processor as shown in Figure 1. Thereby, it reduces the volume of data transferred between main memory and processor chip, improving memory bandwidth utilization.

Similar to most techniques that focus on data approximations [27, 38, 52], data regions are marked by the programmer for approximation upon allocation, using a specialized system call. Then, allocated pages can be marked as *approximable* in the page table requiring an extra bit for every entry in the page table and TLB. The programmer also specifies an acceptable *error threshold* for the approximable data, which AVR uses to limit its impact on application quality. Alongside the page table and TLB, like other memory compression works [14, 48], metadata information per

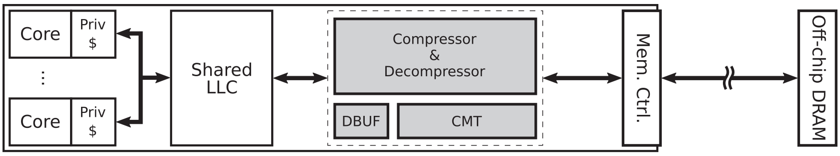


Fig. 1. Top level lossy memory compression architecture used by AVR and MemSZ.

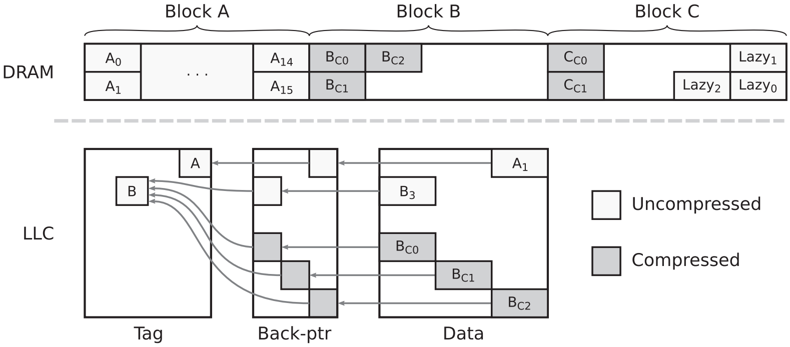


Fig. 2. AVR's Decoupled Sectored Cache and main memory block layout.

compressible memory block is stored in a Metadata Table (MT) in main memory and cached on-chip (Compression Metadata Table (CMT)). CMT is accessed in parallel with the LLC and updated in pair with the TLB. Application data that are not marked as approximable are not compressed. The AVR architecture does not aim for improving memory capacity. Consequently, each block is allocated enough space to remain uncompressed, and therefore memory allocation is not affected. Compressed blocks, then, leave empty memory space between them, which remains uncompacted.

Choosing to compress at a granularity larger than a cache line (64 B), i.e. in blocks of 1 KB/16 cache lines, is necessary for exploiting a larger compressibility potential. However, random access to a single cache line within a larger compressed block in memory is no longer possible. Furthermore, evictions suffer the overhead of attempted compression, which may in some cases be unsuccessful. AVR addresses these challenges by (i) co-locating compressed memory blocks and uncompressed cache lines in the LLC, (ii) handling LLC eviction in a lazy manner, and (iii) keeping track of badly compressing memory blocks. These three points are explained next.

In order to store compressed memory blocks alongside the uncompressed cache lines, AVR employs a Decoupled Sectored Cache [58]. A layer of indirection (*back-pointers*) is introduced to allow a single tag to represent a block of multiple consecutive cache lines. The design is augmented to store both compressed memory blocks partitioned in subblocks (CMS) as well as uncompressed cache lines (UCLs) under the same tag, in any combination. Figure 2 illustrates an example using three data blocks, denoted A, B, and C. Block A is uncompressed in memory, and thus contains 16 distinct UCLs. A recently accessed UCL from block A is stored in the LLC. The back-pointer of this UCL associates it with the tag of block A. Block B is compressed in memory, and thus only occupies a part of its allocated space in main memory. The compressed block B is also currently in the LLC. It occupies three CMSs (B_{C0}, B_{C1}, B_{C2}), each of which has a back-pointer associating it with the single tag representing block B. Furthermore, a recently accessed UCL of block B is in the cache. Its back-pointer also associates it with the same tag for block B. The compressed block C is not on-chip at the moment.

A request to the LLC may hit in three distinct ways, each with increasing latency:

(1) the requested UCL may be in the compressor block, in a buffer storing the most recently decompressed block (Decompression Buffer (DBUF));
(2) the requested UCL may be in the cache in its uncompressed form; or
(3) the compressed block containing the requested UCL may be in the cache.

In a case in which the request hits in a compressed block stored in LLC, the block must be read out of the cache and decompressed, introducing additional latency compared to a regular cache hit. In a case in which all the above options miss, a memory request is issued. The metadata for the block indicates whether the memory location is compressed in main memory or not. If not, the requested UCL can be fetched directly. If the data in the memory location are compressed, the entire compressed block is fetched and decompressed. The compressed block is inserted in LLC, as is the requested UCL. Writebacks to LLC are inserted as in a non-compressing cache. When a dirty line is evicted from LLC, the corresponding compressed block is updated if available on-chip. If the block is only available in memory, it may be brought in and updated.

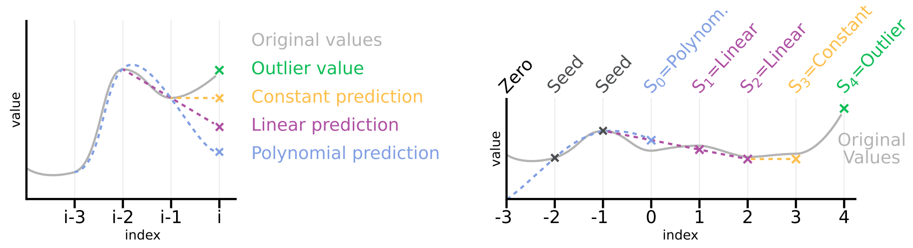
It would, however, be wasteful to update the corresponding compressed block in memory every time there is an LLC line eviction. In the worst case, such an update would require the full compressed block to be read from memory, decompressed on-chip, updated with the evicted data, recompressed and finally written back to memory. The traffic overhead of such an operation would outweigh the benefits of compression. Instead, AVR exploits the unused space between compressed blocks. Since each block is allocated 1 kB of physical memory, compressed blocks leave some of this space empty. As long as there is available empty space, AVR employs *lazy evictions*, writing the dirty cache-line back to memory uncompressed. The next time the block is brought on-chip to satisfy a cache miss, it is updated with all lazily evicted data and recompressed. This technique allows AVR to postpone the costly recompression and mitigate its traffic overhead. Figure 2 illustrates three lazily evicted cache lines in the empty space of block C. These lines were dirty in the LLC in the past, and at eviction time the compressed block C was no longer on-chip. As a result, *lazy evictions* wrote the dirty cache-lines back to memory.

The overhead of unsuccessful compression attempts is minimized by keeping a history of previous compression attempts per block. This history is maintained in the metadata of each memory block. The metadata of a block also includes its compressed size and number of lazy evicted cache lines.

The compression algorithm used by AVR is chosen for its low complexity and low decompression latency. Compression is based on *downsampling*, i.e., computing numeric averages between neighboring values. Decompression requires only interpolation between these averages. Individual values that exceed a set *error threshold* are stored explicitly, ensuring that each recompression meets the threshold. This allows a variable compression ratio ranging from 2× to 16×.

AVR reduced memory traffic up to 70%, improving system performance and energy efficiency up to 55% and 20%, respectively, introducing less than 2% application output error. However, the downsampling compression was in some cases too simple to effectively compress some datasets. Moreover, AVR did not provide a mechanism to keep track of the cumulative error introduced to a memory block by multiple compressions. Finally, the LLC replacement policy treated lines of compressed and uncompressed data equally, leading to sub-optimal utilization of LLC capacity.

2.2.2 SZ Compression. SZ is an algorithm that lossily compresses a sequence of numeric values by describing each value X_i as a function of the three preceding values $[X_{i-3}, X_{i-2}, X_{i-1}]$, according to a predefined function model [11]. Four such function models are supported, as shown in Figure 3(a): (1) A *constant* value is approximated as equal to the nearest preceding one



(a) The four function models of SZ.

(b) Reconstruction of an SZ value sequence.

Fig. 3. The SZ compression scheme.

(Equation (1)), (2) a *linear* value is extrapolated from the preceding two values (Equation (2)), (3) a *polynomial* value fits on the cubic curve described by the preceding three values (Equation (3)), or (4) if none of these models describes a value with an acceptable error, the value is an *outlier* and stored explicitly.

$$X_i^C = X_{i-1}, \tag{1}$$

$$X_i^L = 2X_{i-1} - X_{i-2}, \tag{2}$$

$$X_i^P = 3X_{i-1} - 3X_{i-2} + X_{i-3}. \tag{3}$$

By adopting this classification, a sequence of values can be described using a two-bit *symbol* S_i per value, indicating one of the four functions used to generate the respective value, together with any outlier values, if option (4) is selected. To provide input for decompression, an initial set of *seed* values are explicitly stored. These values represent the first values in the sequence and allow decompression to begin with the first non-seed value.

Figure 3(b) illustrates how a sequence of values can be approximately reconstructed using only these *seeds*, the *symbols* chosen during compression, and the explicit *outlier* values stored. Each value depends on preceding values, forcing both compression and decompression to be sequential.

This sequential dependency between consecutive values increases the processing latency. Moreover, the computations needed for generating the polynomial value fit may be too complex to be performed in a single (processor) cycle. In the past, a hardware (FPGA) implementation of SZ, called GhostSZ has been proposed for accelerating the I/O compression [28]. GhostSZ compresses multiple streams of data in parallel to increase throughput, but the processing of each stream remains sequential. In essence, compressing or decompressing a single block of values is still *O(n)* and when implemented in ASIC would require at least two processor cycles per value due to SZ's complex arithmetic computations. As a consequence, previous SZ approaches are too slow and therefore impractical for memory compression. MemSZ introduces a new low latency parallel version of SZ able to compress/decompress a block of 256 values (16 cache lines) within 16 cycles.

3 MemSZ ARCHITECTURE

MemSZis a new lossy memory compression approach. It targets the parts of application datasets that tolerate approximations and substantially reduces their volume when transferred between main memory and processor chip. Thereby, MemSZ utilizes the off-chip bandwidth more efficiently and in turn improves system performance and energy efficiency. Without loss of generality, MemSZ is applied to a Chip Multiprocessor, between the main memory controller and the LLC. A memory access to a location with compressed data brings on-chip a compressed block of multiple (16) cache lines. After decompression, the requested cache line is stored in the LLC and sent to the

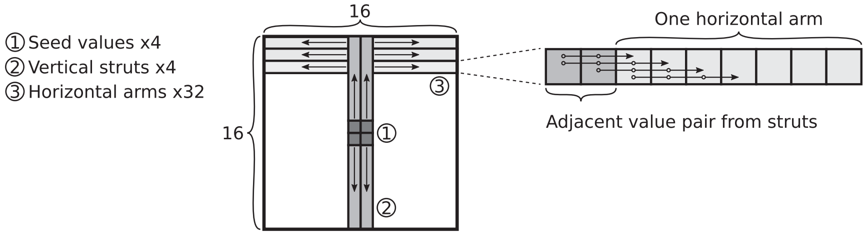


Fig. 4. SZ Compression adapted to a 2D block.

processor, while the memory block is also stored in the LLC, to avoid memory accesses at future requests to the block.

MemSZ builds upon some of the AVR concepts described in Section 2.2.1. Data regions of applications are annotated by the programmer as approximable and marked as such in the page table and TLB. In addition, metadata information is maintained per memory block, stored in memory, and cached on-chip in a CMT. Moreover, one of MemSZ's design alternatives (Figure 1) considers the same Decoupled SRAM LLC organization, storing both compressed and uncompressed data. Finally, like AVR, this design employs a 1 kB *DBUF* to store the most recently decompressed block.

MemSZ extends our previous AVR design, as follows. First, it offers a new more effective lossy compressor, which improves compression ratio. For this purpose, we introduce the first parallel design of the SZ algorithm [11]. Second, a more effective error limiting mechanism is used that is able to keep track of the error of a memory block, accumulated across multiple compressions. Third, it improves AVR's LLC replacement policy. Considering AVR's SRAM LLC on the processor die, MemSZ prioritizes the replacement of uncompressed lines versus compressed blocks; in doing so, LLC capacity is utilized more efficiently. Finally, as an alternative to AVR's modified Decoupled SRAM LLC, MemSZ also explores the use of a DRAM cache with a cache line size that matches the memory blocks considered for compression (1 KB = 16 × 64 B); then the DRAM cache needs to store only uncompressed data.

In the rest of this section, we describe the MemSZ compression and error handling as well as the two alternative LLC designs.

3.1 MemSZ Parallel Lossy Compressor

MemSZ introduces the first parallel hardware implementation of the SZ compression algorithm [11]. SZ is inherently sequential, using the three last produced values to compute the next value in the sequence. This linear dependency is an obstacle to low-latency implementation, since it limits parallelism. MemSZ breaks the linear sequence into a $(2\sqrt{n})$-way parallel operation, where n is the number of values in the sequence, reducing the complexity of the algorithm from $O(n)$ to $O(\sqrt{n})$. In the following section, we present the optimizations behind this improvement.

The MemSZ compressor processes the values stored in a (memory) block in their native arithmetic representation, i.e., floating-point arithmetic is used for blocks of floating-point numbers, and fixed-point arithmetic for fixed-point blocks. Our current implementation supports standard IEEE754 32-bit floating-point formats, but can be extended to integer and fixed-point.

Incoming blocks of 16 cache lines are arranged in a square of 16×16 values, as shown in Figure 4, before being fed to the compressor. In accordance with standard SZ, the generated compressed block consists of a two-bit *symbol* for each input value. Each two-bit *symbol* indicates the selected function used to reconstruct the input value, given its preceding values. The compressor also identifies individual *outlier* values that are not described by the summary with sufficient accuracy. These are explicitly stored alongside the symbol sequence. The proper location of each outlier is encoded in the compressed block using one designated *symbol*.

Decompression is performed in the opposite order. The summary is decoded, using each two-bit *symbol* to reconstruct a value based on its preceding neighbors. Outliers are stored explicitly in the compressed block, and re-inserted in the decompressed output. The complete decompressed block is stored in a DBUF and the requested cache line is inserted in the LLC.

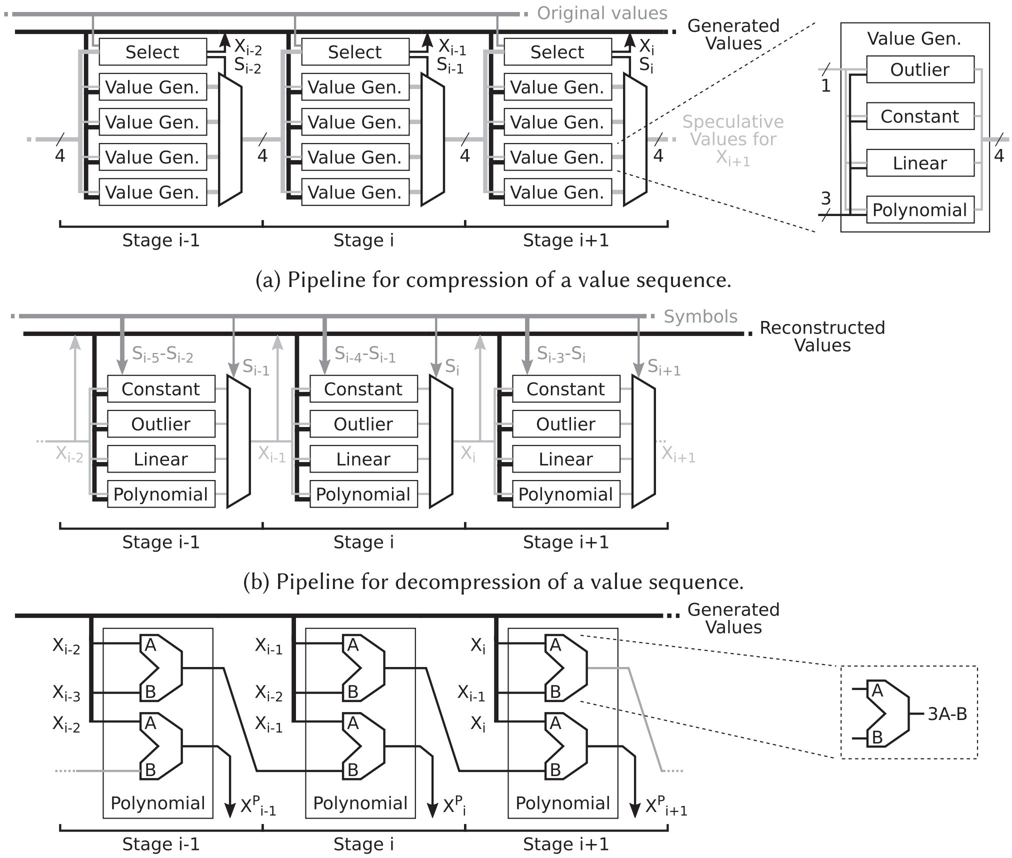
As previously outlined, SZ describes each value in a sequence as a function of the preceding values. Four set function models are supported as described in Section 2.2.2, allowing each value to be described using a two-bit *symbol*. To start the sequence, a set of *seed* values are chosen and explicitly stored. Due to the dependency between the next produced value and the preceding ones, both SZ compression and decompression are sequential and have *O(n)* latency, where *n* is the number of compressed/decompressed values. In fact, a direct hardware implementation of the SZ algorithm would require two to three processor cycles for each value generated. Such high latency is an obstacle for using SZ for memory compression. To counteract this, MemSZ applies the following four techniques:

- Values are generated in multiple parallel sequences in two (block) dimensions requiring $O(\sqrt{n})$ steps.
- The two processing steps of (i) computing the four alternative functions output for the next value and (ii) selecting the appropriate function (the one with the lowest error) are pipelined.
- Value generation of complex functions (i.e., polynomial) are also partitioned and performed in two consecutive pipeline stages.
- Generated values are fed forward in a *dataflow* fashion, allowing each computation to begin as soon as all dependencies are ready.

We use a subdivision of the 16 × 16 block into shorter sequences, which allows values to be generated in parallel. Both compression and decompression are divided into three phases as shown in Figure 4. In phase ①, the four values in the center of the block are used as *seed* values. These serve to start phase ②, consisting of four parallel vertical sequences of seven values each, called *struts*. The seeds and struts together form two complete columns of 16 values down the middle of the block. In phase ③, each horizontally adjacent pair of values from these columns serves to start two horizontal *arms*, making a total of 32. The arms are independent of each other and can therefore be processed in parallel starting as soon as the values they depend on are available—this may be well before phase 2 is complete. Using this method, the critical path to processing an entire block of 256 values is one full strut (seven operations) followed by one full arm (seven operations) for a total of $7 + 7 = 14 = 2 \cdot (\frac{1}{2}\sqrt{n} - 1) = \sqrt{n} - 2$ operations.

Compression: Figure 5(a) outlines a slice of our pipelined hardware implementation of an SZ compressor for a seven-value input sequence $[V_0 \ldots V_6]$. A *Value Generator* module implements the four function model options (*outlier*, X^C, X^L, and X^P) to approximately reconstruct a single value given the three preceding values. In one cycle, all four options are speculatively generated for value V_{i-1} based on its preceding values. In the next cycle, each possible option (Outlier, Constant, Linear, or Polynomial) for value V_{i-1} is used to speculatively generate values describing V_i. In parallel with this, a *Select* module chooses the most accurate reconstruction option X_{i-1} for value V_{i-1}. This yields the two-bit symbol S_{i-1} and resolves the speculation on V_i, resulting in its four options. These four options can then be used to speculatively generate X_{i+1} and so on.

Since compression includes trying all function models for each value, compression latency is determined by the most complex function model. The *polynomial* function shown in Equation (3) is expressed as $X_i^P = 3X_{i-1} - 3X_{i-2} + X_{i-3}$. Considering that these are floating-point computations,



(a) Pipeline for compression of a value sequence.

(b) Pipeline for decompression of a value sequence.

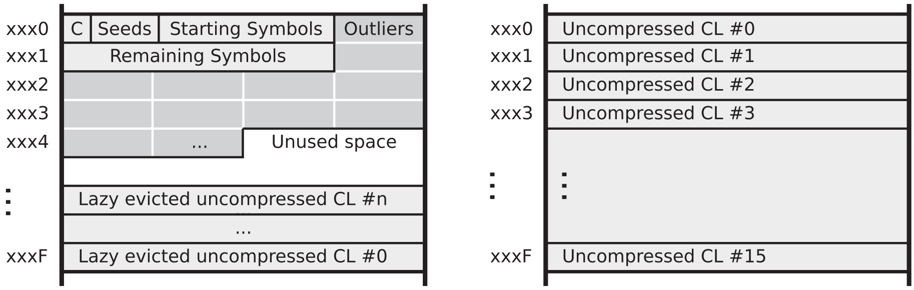
(c) Pipelining of the *polynomial* function.

Fig. 5. Design of the MemSZ compression pipeline.

it would be too complex to fit it in a single cycle. In MemSZ, we break down the polynomial equation as $X_i^P = 3X_{i-1} - (3X_{i-2} - X_{i-3})$; the part within parentheses can then be precomputed a cycle earlier, as soon as X_{i-2} is ready. By implementing the function $3A - B = 2A + A - B$ using a three-operand adder as described in [60], the polynomial function can be pipelined in two stages as illustrated in Figure 5(c), without violating the clock period of our processor as shown in Section 4.1. This allows the polynomial X_i^P to be ready in the same cycle as the other generated options for X_i, and the total latency for the compression of a full block is $2 \cdot (\frac{1}{2}\sqrt{n} - 1) + 2 = 14 + 2 = 16$ cycles, including two additional cycles, one for precomputing the first polynomial and one more for the function selection of the last generated value.

Decompression: Decompression is performed in a similar pipeline (Figure 5(b)), without the speculation. Since *constant* values require no further computation, these are forwarded up to three steps ahead in a single cycle. For example, if S_{i-1} is *constant* and S_i is *constant*, then $X_i = X_{i-1} = X_{i-2}$ and all three can be ready on the same cycle as X_{i-2}. Thus, several consecutive *constant* values can be reconstructed in a single cycle.

Because of this optimization, total decompression latency is variable. In the best case, the compressed block contains a large number of *constant* symbols and can be decompressed in six cycles. The longest latency is in a block containing an unbroken sequence of 14 *polynomial* values, which takes 16 cycles to decompress.



(a) The MemSZ compressed memory block format.

(b) Uncompressed memory block.

Fig. 6. MemSZ memory block.

Compression of constants: As an additional method to reduce the size of the compressed block, individual constants can be compressed by reducing their precision. If permitted within acceptable error thresholds, MemSZ applies a fixed 2:1 compression to seeds and outliers stored alongside the compressed block. This complements the compression achieved by SZ and reduces the constant overhead of the seed values.

Symbol compression: The compression scheme outlined above requires a minimum of four *seeds* (stored in half precision) and 252 *symbols* to describe a block. The total size of this is $4 \cdot 16 + 2 \cdot 252 = 568$ bits $= 71$ bytes, which exceeds the size of a cache line. As a result, the maximum achievable compression ratio would be $16 : 2$ or 8×. In order to increase the compression ratio to $16 : 1$, blocks with very few outliers undergo an additional pass of lossless encoding. The occurrences of each unique *symbol* are counted. New symbol encodings are assigned, such that the most common symbol requires only one 0 bit to be represented. The second-most common symbol is represented with two bits (10), outliers are given the three-bit code 111, and the remaining symbol is represented by 110. The total size of the compressed block with this encoding is the size of stored seeds and outliers plus the size of the re-encoded symbols plus a four-bit *dictionary* and a single bit to indicate that the block has been encoded this way. If this total size is less than one cache line, the block is stored using the smaller encoding. If the total size exceeds one cache line, the block is stored using the block format previously described. This re-encoding takes three cycles during compression, while it takes two cycles to decode the first symbols before decompression can start.

Memory Blocks: To further facilitate low-latency decompression, we choose a format for the compressed block that allows decompression to begin as soon as the very first compressed memory subblock (CMS) is available. This format is shown in Figure 6(a), and packs the following into the very first 64-byte CMS: (1) a single bit *C* indicating if the symbols themselves are compressed; 2) the four seed values from the center of the block, at half precision; 3) the 28 two-bit symbols for the vertical struts; 4) up to the 24 first outlier values, at half precision; and (5) any additional symbols for other positions (at least four). Using this information, the decompressor can start reconstructing the center two columns of the 2D block on the very first cycle after these first 64 B become available. The summary of a block compressed with *SZ* implicitly contains the required information to locate outliers, rendering AVR's outlier bitmap obsolete. As in the previous design, the empty space following a compressed memory block is used for lazy eviction of dirty cache lines, a technique designed to reduce and postpone recompression overhead, discussed in Section 2.2.1.

Total compression and decompression latency: Based on the above and as confirmed by our synthesis results presented in Section 4, the total worst-case latency for compressing a block is 32 processor cycles; that is, 16 cycles for the actual compression and another 16 for packing the

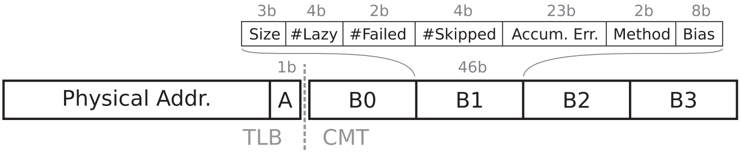


Fig. 7. Format of a metadata table entry and TLB addition.

outliers. The worst-case latency for decompressing a block is 18 cycles; 2 for decoding the first symbols and 16 for decompressing the block. Decompression is more critical for system performance as it affects memory reads. Compression is less critical because it affects only the writebacks.

3.2 Error Limiter

Lossy compression introduces errors in the approximated values. In order to limit this error, compression is skipped in cases in which the error exceeds a particular threshold value. AVR employed the following two thresholds during compression to control the approximation error: the relative error of each individual value may not exceed the threshold T_1 and the average relative error across all values in the block may not exceed the threshold T_2. However, AVR's approach accounts only for the error introduced during the current compression. Oft-transferred blocks may accumulate error over their lifetime due to multiple compressions and AVR offers no mechanism to control this.

To limit the impact of accumulated error over time on the output quality of the application, MemSZ introduces an *error limiter* strategy. An accumulating counter is maintained for each block, updated with the average block error after each compression. If this accumulated error exceeds an absolute threshold T_3, compression is permanently disabled for the offending block.

MemSZ accounts for the two local thresholds T_1 and T_2 in two steps during compression. First, if none of the function models can approximate a value to within T_1, the value is kept as an outlier. Second, the average error of the compressed block is computed during compression and if it exceeds T_2 the block is left uncompressed.

The three error thresholds are exposed as knobs and in our experiments $T_1 = 2T_2$ while T_2 and T_3 are selected by profiling each individual application.

3.2.1 *Metadata Table.*

Similarly to AVR, MemSZ uses a single bit in the page table and TLB to identify pages marked as approximable. Further metadata for each individual page is stored alongside the page table and cached on-chip in a dedicated CMT. These structures are shown in Figure 7. The CMT is updated in pair with the TLB. A CMT has four 46-bit entries per 4 KB page; one per 1 KB memory block as shown in Figure 7. CMT stores the following information about each memory block: its size (compressed in one to seven lines or uncompressed), number of lazy evicted cache lines stored, compression method (and datatype of values), and a bias of its values. In addition, it maintains two counters to keep the history of previous compression attempts. The first one counts the number of consecutive failed compression attempts. Then, depending on that count, a number of recompression attempts (in block updates) are skipped to reduce the overhead of badly compressed blocks. Finally, a running count of accumulated error is kept, to enable the *error limiter* outlined above.

3.3 LLC

After a memory access, keeping the entire accessed compressed block, rather than only the requested cache line, on-chip is important for reducing future memory accesses. This needs to be supported by the LLC. MemSZ explores two alternative system designs with different LLC organizations. The first one is based on AVR's Decoupled Sectored SRAM Cache placed on the processor die, where MemSZ proposes a new replacement policy. In an attempt to avoid AVR's LLC modifications, MemSZ also explores a DRAM LLC with a cache line size that matches the size of the memory blocks considered for compression. This second alternative uses a regular DRAM cache that stores uncompressed lines.

3.3.1 *SRAM LLC on the Processor Die.*

A MemSZ system with an SRAM LLC on the processor die, employs AVR's Decoupled Sectored Cache, which is able to store UCLs and the compressed memory block (fragmented in CMS), as described in Section 2.2.1. Cache lines already accessed by the processor are stored as UCLs, while the rest of the memory block is compressed and stored in one or multiple CMSs. Then, as explained in Section 2.2.1, an LLC hit in the compressed block has a longer access time than a regular UCL hit because it requires decompression.

In the AVR system, there was a fair least recently used (LRU) replacement policy between UCL and CMS. This practically meant that cache lines already accessed by the processor could be found in LLC uncompressed and accessed again without a decompression overhead. However, it also meant that the LLC would often store a compressed memory block as well as (some) of its cache lines uncompressed (UCL), a redundancy which can waste LLC capacity.

MemSZ chooses a different tradeoff between LLC capacity and access latency. It modifies the LLC replacement policy so that uncompressed approximable lines have higher priority to be replaced than compressed memory subblocks because they are redundant. Then, the LLC capacity is better utilized at the cost of a longer access latency to previously accessed lines. Normally, accessing an uncompressed cache line would cause an update to its LRU bits. However, in a case in which the LLC also contains the respective compressed block (indicated by the common tag entry), MemSZ updates the LRU bits of the CMSs instead. This marks the compressed block as recently used, delaying its replacement as opposed to the UCL. As a consequence, compressed blocks may remain on-chip longer, thereby using the LLC capacity more efficiently.

This reprioritization of replacing compressed blocks may be detrimental to applications whose full working set fits in LLC. These can be divided into three classes: (A) applications where the full footprint fits in LLC, (B) applications where the working set fits only due to compression, and (C) applications where the working set would fit in the LLC, but was evicted and thus compressed.

Class (A) suffers no added latency under MemSZ, since the working set is never evicted from LLC and therefore never compressed. MemSZ performs like a normal decoupled sectored cache in this case. Class (B) benefits from compression, since a non-compressing system would suffer memory accesses that are avoided using compression. Class (C) is the case where compressed blocks compete for LLC space with uncompressed lines. This occurs when (1) the application footprint is large enough to fill most of the LLC and (2) the average compression ratio is too low to fit the compressed blocks in the remaining space. For example, at an average compression ratio of 8×, this happens with a footprint between 90% and 100% of LLC. Ideally, the full working set should be decompressed and kept in LLC for such applications. One possible mitigation mechanism for this is for the LLC to maintain a counter for each compressed block, incremented when the block is decompressed. If a compressed block is decompressed on-chip multiple times, the replacement reprioritization can be selectively disabled for that block. Experiments targeted at class (C) indicate that a majority of LLC hits are to already uncompressed lines, for a total performance degradation of only 1%–2%. For these reasons, MemSZ does not include this mitigation mechanism.

One concern when implementing a sectored cache is the handling of multi-banked cache designs, where slices of the LLC serve separate parts of the shared physical address space. Since MemSZ uses different hash functions for the tag and data indices, the two risk being placed in different slices, complicating cache management. To prevent this, the hash functions are chosen such that the most significant bits of the tag index come from the same part of the address as the

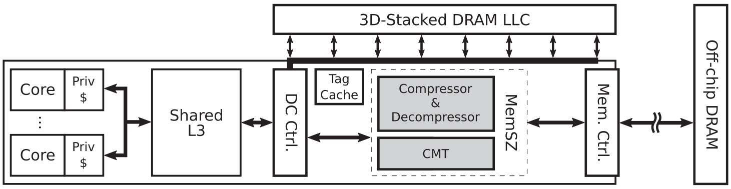


Fig. 8. The architecture of MemSZ applied with a 3D-stacked DRAM cache (MemSZ-DC).

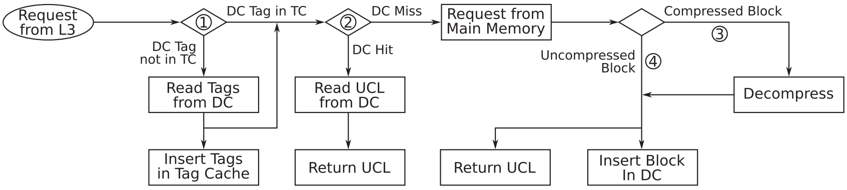


Fig. 9. The handling of a request in a MemSZ system with a DRAM cache.

leading bits of the data index, similar to the design described in [23]. This way, MemSZ ensures that a tag and its associated data are always located in the same LLC slice.

3.3.2 *3D-Stacked DRAM LLC.*

A system may use 3D-stacked DRAM, rather than SRAM, for implementing the last level cache. A DRAM cache offers larger capacity than an SRAM LLC and higher bandwidth than the off-chip main memory [20, 22, 26, 33, 37]. DRAM caches often use larger cache lines than the SRAM caches [22, 33]. MemSZ exploits this characteristic and employs a DRAM cache with line size equal to the size of the compressed memory blocks (1 KB). This significantly simplifies the implementation of the compression system, since full memory blocks read from the memory can be stored uncompressed in the DRAM cache. This removes the necessity of co-locating compressed blocks in LLC. In addition, making the DRAM cache inclusive eliminates the need for lazy evictions. This is because the full dirty block is available for recompression on eviction. Both of these properties simplify the control logic.

Our MemSZ system with DRAM cache (MemSZ-DC) is depicted in Figure 8. The compression-decompression remains between the main memory controller and the (DRAM) LLC. The DRAM cache is organized as a set-associative cache with a 1 kB block size. Tags for the DRAM cache are kept in a reserved area of the DRAM cache itself, with an SRAM tag cache on the processor die to store recently accessed tags.

A request from the shared (L3) cache for an approximable line (illustrated in Figure 9) can have the following outcomes:

① Miss in the Tag Cache, indicating that the DRAM cache lookup cannot be performed. The relevant tags must be fetched from the DRAM cache before handling the request.

② Hit in last level DRAM cache, The line is read out and returned to L3.

③ Miss in last level DRAM cache, compressed in main memory: The compressed block is read from main memory and decompressed. The requested line is returned to L3. The entire decompressed block is inserted in the last level DRAM cache.

④ Miss in last level DRAM cache, uncompressed in main memory: The entire block is read from main memory and inserted in DRAM cache. The requested line is returned to L3.

MemSZ-DC performs compression in the same manner as MemSZ, and maintains the same metadata. Compression failure (failure to meet the error threshold) is handled by storing the block uncompressed in main memory, and disabling compression for a set number of future attempts.

Table 1. Simulation Parameters

Parameter	Configuration
CPU	8 core, out-of-order, 4-way issue/commit @ 3.2 GHz
L1 cache	64 kB per core, 4-way, 1 cycle latency
L2 cache	256 kB per core, 8-way, 8 cycle latency
L3 cache	8 MB shared, 2 banks, 16-way, 15 cycle access latency
L4 cache	256 MB HBM2, 4 channels, 1,066 MHz, 8-way
Main Memory	8 GB DDR4, 2 channels, 800 MHz

4 EVALUATION

In this section, we evaluate the effectiveness of the two proposed MemSZ architectures. We first describe our experimental setup, detailing the system configuration of our experiments and the benchmarks used. Then, we discuss the hardware overheads of the MemSZ architecture. Subsequently, we show the results of our evaluations and comparison with related designs in terms of performance, energy, and application output error. Results are first presented for designs with an SRAM LLC and then for designs with a 3D-stacked DRAM cache (MemSZ-DC). Finally, we evaluate the effect of the individual MemSZ features, namely, the compressor, error limiter, and replacement policy.

4.1 Experimental Setup

We evaluated the MemSZ system using an in-house simulator, implemented on top of Pin [43], that employs an interval-based processor model, as proposed by Genbrugge et al. [19], and a cycle-accurate model of the memory hierarchy that uses DRAMSim2 for modeling main memory and DRAM caches [51]. McPAT [42] and CACTI [44] were used to model power and latency of the system considering 32 nm technology. The MemSZ compression hardware modules were implemented in RTL, synthesized using Synopsys to determine their operating frequency, latency, and power consumption; this information was then fed to the simulation tool. The parameters of the simulated system are listed in Table 1.

In order to correctly emulate the impact of the approximations in the overall application error, we emulate not only the memory accesses but we actually update the values of the memory contents accordingly by applying the compression and reconstruction methods to the data.

Besides the baseline system, MemSZ is further compared with (i) the original design (*AVR*) [15], (ii) a design that simply compresses approximate values to half-precision by truncating 16 bits similarly to what has been proposed in [38, 39, 57] (*Truncate*), and (iii) Doppelgänger [27], which is the closest and best performing related work on approximate data compression (*Dganger*). As proposed, Doppelgänger is configured to have identical LLC data-array size and a 4× larger tag-array versus the other designs, i.e., being able to index up to 4× as many cache lines. Lossless compression techniques are considered orthogonal and so not included in the comparison; that is because the symbols and outliers of a MemSZ compressed block could be further compressed in a lossless way, in addition to compressing the non-approximable data that MemSZ leaves untouched. Finally, the individual features of MemSZ are evaluated by comparing it with (i) itself replacing its compressor with AVR's downsampling compressor (*MemDS*), (ii) itself with equal LRU priority between compressed and uncompressed lines (like in AVR) (*MemLRU*), and (iv) itself without the error limiter (*MemUnL*).

MemSZ with DRAM cache (MemSZ-DC) is compared to (i) the baseline system and (ii) an approximating design that halves precision of data transferred across the main memory bus

Table 2. Benchmark Applications

Application	Approx.	Output	Footprint / core		Checkp.	Description
			Small	Large		
heat [49]	Temps	Temps	8.3 MB	128.4 MB	✓	Heat propagation through a 2D field of uniform material
lattice [30]	P and M	Vel.+Press.	5 MB	160 MB	✓	2D Lattice-Boltzmann simulation of air flow
lbm [35]	Velocities	Velocities	325 MB	325 MB		3D Lattice-Boltzmann simulation of fluid flow
orbit [9]	Phys. data	Phys. data	10 MB		✓	3D simulation of the two-particle orbit problem
cdelta [9]	Phys. data	Phys. data	22 MB		✓	Delta-function heat conduction model
sedov [9]	Phys. data	Phys. data	12 MB		✓	Sedov explosion model
windt [9]	Phys. data	Phys. data	23 MB		✓	Windtunnel with a step
kmeans [3]	Topol. [2]	Clusters	5.5 MB	802 MB	✓	Clustering, applied on a geographic elevation map
bscholes [64]	Options	Prices	6 MB	1368 MB		Financial stock option price forecasting model
wrf [35]	Geo data	Temp.	90 MB	90 MB		Weather forecasting model

(*Trunc-DC*). Each of the three has an identical cache hierarchy and DRAM cache. AVR and Doppelgänger are not included in this comparison as they do not support a DRAM cache.

MemSZ is applicable to workloads that satisfy two requirements. First, the application must be able to tolerate inaccuracy in some subset of its data. While some applications are explicitly approximate in their nature (e.g., lossy image compression), others are still able to tolerate some approximation without failure [16]. Second, this subset of data must be structured in a way compatible with the compression scheme described in Section 3. In addition to this, in order to evaluate the design, the applications chosen must be able to execute to completion and generate an output for verification. This restricts us to the benchmarks listed in Table 2; the table further presents the application domain, description of the approximated data-structures and output type, as well as their memory footprint.

In some applications, the selection of safely approximable data is affected by the technique applied. For this reason, the approximable footprint may differ between designs for the same application. Furthermore, some of the applications have a footprint smaller than the evaluated DRAM cache, and are therefore excluded from these experiments. The application code was analyzed to identify approximable data structures. In many cases, a large portion of the application's working set is dynamically allocated. For these cases, a wrapper was created around the *malloc* library call to allocate properly aligned space and register the address range as approximable.

One common source of memory traffic in scientific workloads is *checkpointing*. Checkpoints are occasional snapshots of the application's state, for the purpose of resuming execution after errors or outages. Such snapshots generate large bursts of data transfers to non-volatile storage, and contain approximable data from the application's working set. To reflect the effect of compression on these data, iterative benchmarks with checkpointing support have it enabled as indicated in Table 2.

The input datasets used for our experiments are the standard input datasets provided with the benchmarks with the exception of (i) *lattice* for which we used a silhouette of a car as the input data set, and (ii) *k-means* where the input is topological data [2]. Benchmarks for approximate computing considers 10% relative output error [64], but it is solely up to the application provider to define what is acceptable. We use the mean of the relative errors for each output value as our quality metric. The only exception to this is *k-means*, whose output is discrete and strongly bounded. For this application, we normalize each individual error to the maximum possible error for that value, such that the maximum possible error is 100%. Similar to previous works, MemSZ provides tunable knobs to control the data approximation error and constrain application output error.

Table 3. Application Output Error

(a) SRAM LLC designs

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	bscholes	wrf
dganger	0.4%	0.2%	0.1%	<0.05%	>100%	<0.05%	0.4%	<0.05%	<0.05%	56.8%
truncate	0.9%	0.6%	0.2%	<0.05%	<0.05%	<0.05%	6.3%	<0.05%	2.8%	6.2%
AVR	0.3%	0.5%	0.1%	<0.05%	4.8%	<0.05%	1.8%	0.7%	0.5%	8.9%
MemUnL	0.5%	0.5%	0.5%	<0.05%	3.8%	<0.05%	1.9%	1.0%	1.8%	crash
MemSZ	0.5%	0.5%	0.5%	<0.05%	2.6%	<0.05%	1.8%	0.6%	1.8%	0.6%

(b) DRAM LLC designs

	heat-large	lattice-large	lbm	kmeans-large	bscholes-large	wrf
Trunc-DC	<0.05%	0.1%	0.1%	<0.05%	0.6%	<0.05%
MemSZ-DC	<0.05%	0.3%	0.4%	0.2%	1.7%	<0.05%

4.2 Hardware Overhead

MemSZ requires additional hardware resources compared to a baseline cache hierarchy. The metadata stored in the CMT and the additional bit in the TLB add up to 173 bits per page. Compared to an unmodified TLB, which stores a virtual and a physical page address (52 + 36 = 88 bits), the CMT is roughly 2× its size. The MemSZ Tag array and the BPA add to the baseline set-associative LLC 18 bits per entry; that is in total a 3.2% overhead to the LLC. One method to alleviate this overhead is to reduce the associativity of the tag array. The sectored nature of the MemSZ LLC allows fewer tags than data entries, as long as there is sufficient spatial locality. While the presented results correspond to an equal-associativity (16-way) tag array, MemSZ with an 8-way tag array suffers only a 0.7% performance reduction. In such a design, the total overhead of tags and BPA is 14 bits per entry, 2.5% of the LLC capacity.

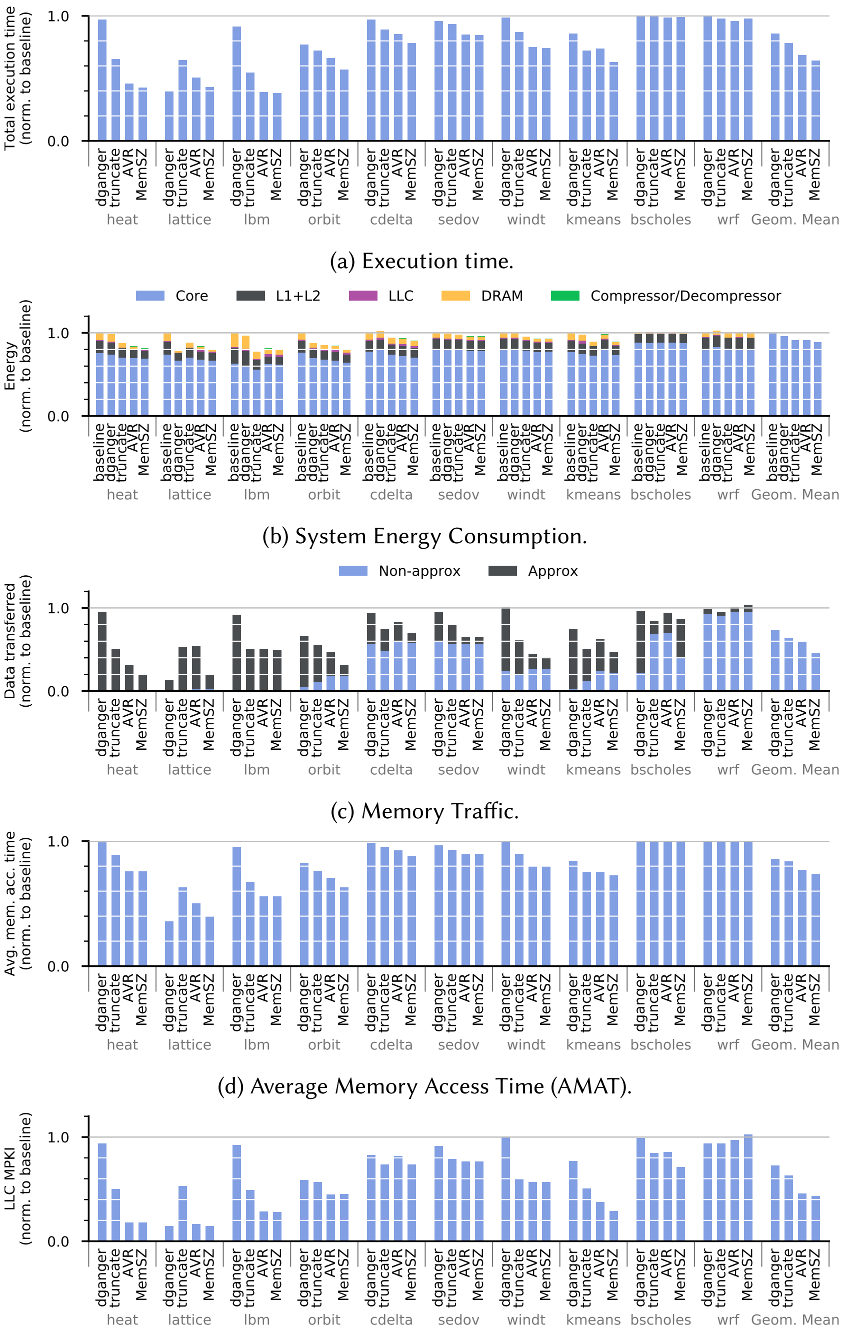
The MemSZ compressor module occupies about 860k cells according to our synthesis report. Any number of compressor modules can be employed in parallel to meet the demand of large-scale systems. The pipelined nature of our compressor design allows a single module to handle the request load of multiple processor cores.

4.3 Experimental Results

Next, we present our experimental results, comparing the two MemSZ designs against related approaches. The designs are evaluated in terms of execution time (Figures 10(a) and 12(a)), system energy consumption (Figures 10(b) and 12(b)), main memory traffic (Figures 10(c) and 12(c)), average memory access time (AMAT; Figures 10(d) and 12(d)), and LLC misses per kilo-instruction (MPKI; Figure 10(e)), as well as in terms of application output error (Table 3(a) and (b)). Table 4 shows the compression ratios achieved by MemSZ and AVR as well as the reduction of the memory footprint versus the baseline. Finally, the individual features of the MemSZ architecture are evaluated separately in Figure 11.

4.3.1 MemSZ with SRAM LLC.

On average, MemSZ reduces the baseline execution time by 36% and system energy by 12%, the highest reduction across all the competing designs. That is mainly due to the fact that it has the lowest total memory traffic, with an average 46% of the baseline, as well as the lowest LLC MPKI, which is 43% of the baseline due to storing compressed memory blocks in the SRAM LLC. These two factors lead to a reduction of the baseline AMAT by 26%. It is worth noting that for some benchmarks the baseline execution time is reduced by about 60% (heat,



(a) Execution time.

(b) System Energy Consumption.

(c) Memory Traffic.

(d) Average Memory Access Time (AMAT).

(e) LLC misses per kilo-instruction (MPKI).

Fig. 10. Evaluation of the MemSZ design with SRAM LLC and comparison with competing designs.

Table 4. Compression Efficacy

(a) MemSZ and AVR compression ratios. Truncate has a fixed 2× ratio.

	heat	**lattice**	**lbm**	**orbit**	**cdelta**	**sedov**	**windt**	**kmeans**	**bscholes**	**wrf**	**GM**
AVR	10.5×	6.8×	15.3×	16.0×	4.2×	15.6×	11.4×	2.3×	9.4×	2.3×	7.6×
MemSZ	15.8×	15.5×	16.0×	16.0×	9.2×	15.6×	15.5×	3.4×	6.3×[1]	1.6×	9.3×

(b) MemSZ and AVR memory footprints compared to baseline.

	heat	**lattice**	**lbm**	**orbit**	**cdelta**	**sedov**	**windt**	**kmeans**	**bscholes**	**wrf**	**GM**
AVR	17.5%	27.2%	8.0%	48.1%	71.3%	72.5%	50.7%	61.8%	78.6%	93.1%	43.0%
MemSZ	14.3%	20.1%	7.7%	48.1%	66.4%	72.5%	49.5%	51.5%	54.4%[1]	95.4%	38.3%

lattice, lbm) or by 40% (orbit, kmeans) and the energy consumption by up to 25% (heat, lattice, lbm, orbit) primarily due to very high memory traffic reduction, 2–5×.

The AVR design comes second with an average 31% reduction in execution time and 9% reduction in energy, which is 14% and 25% less impact than MemSZ on execution time and energy, respectively. AVR reduces memory traffic to 60% of the baseline as it uses a less efficient compressor, and has slightly higher MPKI than MemSZ due to its LLC replacement policy, which keeps redundant uncompressed lines and their compressed memory blocks. Overall, its average AMAT is up to 25% higher, with an average of 4% higher than MemSZ.

Truncate offers an average 22% reduction in execution time and 9% in energy despite the limited compression ratio (2:1). Memory traffic is reduced to 64% of the baseline on average, and MPKI is at 63% of the baseline because it offers double the capacity for approximate LLC cache lines. This yields an AMAT of 84% on average for Truncate.

Doppelgänger has an average execution time that is only 14% lower than the baseline and system energy reduced only by 4%. It reduces MPKI (due to higher effective capacity) by 26%, which is lower than all other competing designs. This has an effect on memory traffic and AMAT, which are reduced by 26% and 14%, respectively.

Some of the benchmarks used have little approximate data or are already significantly improved/compressed by previous work (AVR). This has an impact on the above presented average results; however, for some other benchmarks MemSZ provides a significant improvement over the current state of the art. Below, we analyze different groups of benchmarks separately and compare with the best previous designs.

As shown in Table 4(a), for some benchmarks MemSZ improves compression ratio compared to AVR by 1.5× (heat, windt, kmeans) or 2× (lattice, cdelta). For these benchmarks, memory traffic of approximable data is reduced by about the same percentage. Note that in a couple of cases (cdelta, wind) the reduction in total memory traffic is less pronounced because the largest part of the traffic is non-approximable. On the other hand, MPKI for these benchmarks is less affected. For heat, lattice, and windt, AVR had already reduced the MPKI significantly and MemSZ maintains or slightly improves AVR's results. In cdelta and kmeans, MemSZ improves AVR's MPKI by about 10–25%. Reducing traffic and MPKI reduces AMAT compared to AVR by 5–25% for lattice, cdelta, and kmeans, but does not show further improvement in heat and windt. Overall, for these five benchmarks MemSZ improves execution time compared to AVR by 2–15% and system energy by 1–9%. It is worth noting that the application error for all the above benchmarks remains stable (lattice, windt), slightly changes ±0.2% (heat, kmeans), and in cdelta reduces to almost half of AVR's.

[1] MemSZ safely approximates a larger portion of the footprint compared to the other designs.

For other benchmarks, AVR's compression ratio is already close to the maximum 16:1 (lbm, orbit, sedov) and therefore MemSZ does not have significant room for improvement. Although for lbm and sedov, compared to AVR, MemSZ memory traffic and MPKI are not reduced, for orbit it reduces to $\frac{2}{3}$ due to the better LLC replacement policy, which yields significantly fewer writebacks as explained in Section 4.3.2. As a result, MemSZ does not improve AVR's AMAT for lbm and sedov, but it does so by 21% for orbit. Execution time and system energy follow the same trend, AVR and MemSZ have similar performance and energy efficiency for lbm and sedov, but for orbit MemSZ reduces execution time by 14% and energy costs by 6%. Finally, MemSZ increases lbm's output error by 0.4% and maintains negligible error (<0.05%) in the other two benchmarks.

For wrf the approximable data are only a small fraction of the dataset (12%) and therefore there is little room for improvement. As a consequence, there are negligible (±2%) changes in performance and energy efficiency; however, MemSZ reduces wrf's output error by 15× compared to AVR.

Finally, for bscholes AVR can approximate a smaller fraction of the dataset (27%) without crashing compared to MemSZ that approximates about 50% of the data and therefore achieves higher reduction of the memory footprint (46% vs. 21%) despite the lower compression ratio (6.3× vs. 9.4× of AVR). As a result, MemSZ reduces memory traffic and MPKI compared to AVR, but this has negligible effect in execution time and energy consumption because the performance of bscholes is not limited by memory bandwidth.

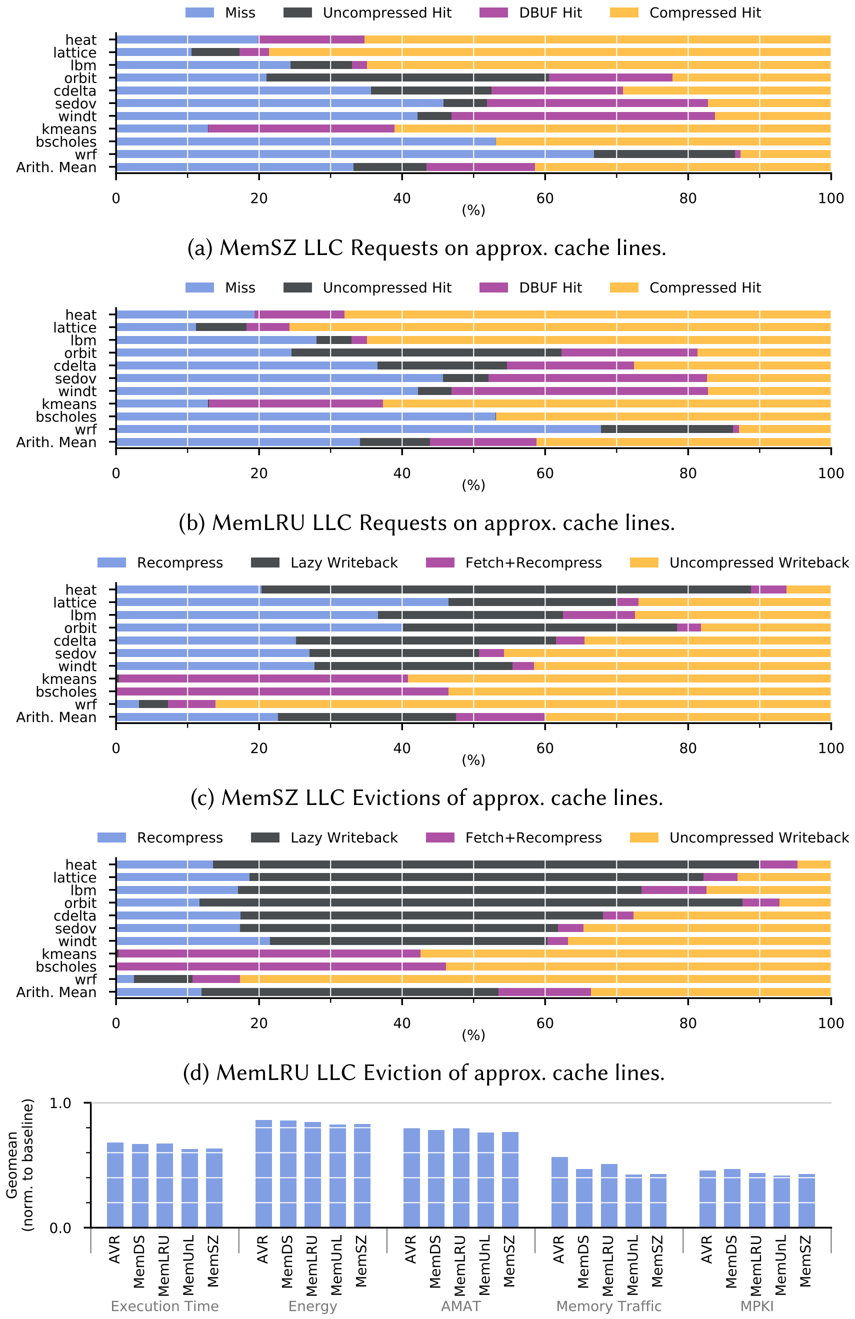
In summary, for applications with high compressibility and approximation tolerance (*heat, lattice, lbm, orbit, sedov, windt*), MemSZ outperforms competing designs. It achieves significant reduction in baseline execution time (up to 62%) and considerable energy savings (up to 25%) with less than 2% output error. Memory traffic is also reduced for these applications by up to 81%. Even when achieving medium compression ratios, i.e., in *kmeans*, MemSZ improves performance by up to 37%. MemSZ improves on the previous best design, AVR, by 23% in memory traffic, 7% in performance, and 3% in energy on average.

4.3.2 Evaluation of Individual MemSZ Features. We investigate the effect of each of the three new primary MemSZ features by disabling one of them at a time. The resulting designs (i) *MemDS*: using the downsampling compressor of AVR, (ii) *MemLRU*: using AVR's LLC replacement policy, and (iii) *MemUnL*: using no global error limiter (only AVR's local error checks), are compared to the original *AVR* design, lacking all these features, as well as with *MemSZ*, which has all features enabled.

Figure 11(e) shows this comparison for the key metrics of execution time, total system energy, average memory access time, main memory traffic, and LLC MPKI. Each metric is presented as a geometric mean across all the benchmarks from Table 2, normalized to the baseline. We observe that MemSZ outperforms AVR as well as all variations except *MemUnL*, losing 1% in execution time when enabling the error limiter and disabling some compression attempts. Table 3(a) illustrates the additional output error caused by disabling the error limiter. Note that when disabling the error limiter, for cdelta and kmeans the output error increases by about 50% and wrf crashes. Using the proposed MemSZ compressor rather than downsampling offers 10% lower memory traffic and 12% lower MPKI. Moreover, using MemSZ's LLC replacement instead of AVR's improves MPKI by 5% and reduces memory traffic by 15%.

Taking a closer look at the impact of the MemSZ LLC replacement policy in comparison to that of AVR (MemLRU), we analyze the LLC requests and LLC evictions of the two designs in Figure 11(a) and (b) and 11(c) and (d), respectively.

As shown in Figure 11(a) and (b), LLC requests may result in one of the following: a miss, a hit to an uncompressed line, a hit to the DBUF of the compressor, or a hit to a compressed memory block stored in the LLC. Compared to the old replacement policy, MemSZ slightly reduces the misses by



(a) MemSZ LLC Requests on approx. cache lines.

(b) MemLRU LLC Requests on approx. cache lines.

(c) MemSZ LLC Evictions of approx. cache lines.

(d) MemLRU LLC Eviction of approx. cache lines.

(e) MemSZ with each addition disabled.

Fig. 11. Analysis of the compressor, error limiter, and LLC replacement policy introduced by MemSZ.

3%, increasing the hits to the compressed blocks. This is more evident in heat, lattice, orbit, and sedov. In some cases like lbm and orbit, the hits to uncompressed lines also increase.

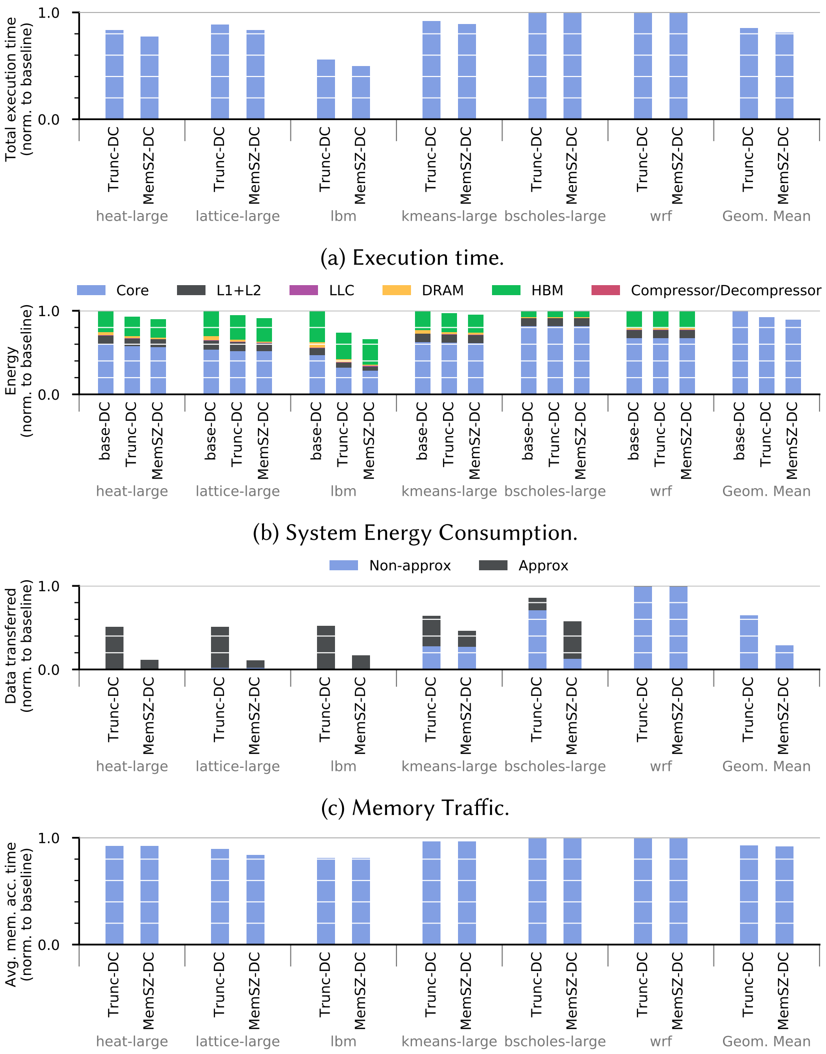
LLC evictions, shown in Figure 11(c) and (d), can be handled in one of the following ways: update of the respective compressed memory block stored in the LLC (recompress), perform a lazy writeback of the evicted line uncompressed (when there is available space in the memory), the memory block may need to be fetched from memory to be updated (fetch+recompress), or there may be a common uncompressed writeback (if block is not compressed). Comparing the breakdown of evictions for the two designs, we can observe the following. MemSZ increases the number of evictions served with a recompression by 89%, avoiding memory accesses. MemSZ also reduces lazy evictions by 40%, increasing the benefit of compression. Finally, the unwanted fetch+recompress cases are reduced by 4%.

4.3.3 Impact of Hardware Prefetching. MemSZ creates an effect similar to prefetching, since an LLC miss may lead to a full 1 kB block being brought on-chip. To realize the effect of this prefetching benefit, we compare MemSZ to a baseline system equipped with a streaming prefetcher in the L1 cache. The performance impact of adding a prefetcher to the baseline system varies between a 10% improvement (lbm) and an 18% deterioration (cdelta), with a mean performance of 0.5% worse than the baseline. Since several of the evaluated applications are bandwidth-bounded, the effect of excessive memory accesses is pronounced. The performance gap between baseline and MemSZ increases by 6% when a prefetcher is added.

4.3.4 MemSZ Modifications to SZ. MemSZ introduces modifications to the SZ compression scheme in order to reduce latency. These changes may have an adverse effect on compressibility, since the vertical *struts* of MemSZ may lack the value correlation of horizontal (adjacent) values.

To investigate this, we compare the 2D MemSZ approach to *SZ1D*, a slightly optimized sequential version of SZ. SZ1D uses four seeds, in two pairs positioned at 1/4 and 3/4 of the sequence. From each pair of seeds, processing can be performed both forward and backward. This arrangement still does not compete with MemSZ in terms of performance (the decompression latency is $O(n)$) but maintains the value correlation across the entire block sequence. The mean compression ratio achieved by SZ1D exceeds MemSZ by 3%.

4.3.5 MemSZ-DC Evaluation. As expected, adding a DRAM cache (DC) to the system significantly reduces the traffic to main memory. This makes it more difficult to evaluate the proposed MemSZ-DC as it would require having longer running benchmarks and larger memory footprints making simulation times prohibitively long. As a result, our experiments show a smaller impact in systems with DC. We use all benchmarks with memory footprint larger than the DC size to evaluate our MemSZ approach with DC (MemSZ-DC) and compare it with a Truncate system with DC of the same size (Trunc-DC). Both designs are normalized to a baseline system with no compression and a DC of the same size. The other competing designs are not included in this comparison as they do not support DRAM caches. Figure 12 shows the execution time, system energy consumption, memory traffic, and AMAT. MemSZ-DC reduces baseline memory traffic by 70% and Trunc-DC only by 35%. AMAT is less affected as MemSZ-DC reduces it to 92% of the baseline and Trunc-DC to 93%. This reduction in traffic and AMAT allows MemSZ-DC to reduce baseline execution time to 81% and energy consumption to 90% on average, while Trunc-DC reduces execution time to 85% and energy to 93% of the baseline. It is worth noting that the most pronounced improvements are achieved in heat, lattice, and lbm, where memory traffic is reduced to 10–20% of the baseline traffic and execution time to 50–85% of the baseline.



(a) Execution time.

(b) System Energy Consumption.

(c) Memory Traffic.

(d) Average Memory Access Time (AMAT).

Fig. 12. Evaluation of the MemSZ design with DRAM LLC and comparison with competing designs.

5 CONCLUSIONS

MemSZ is a new more effective lossy memory compression approach. It introduces a new parallel design for the SZ algorithm, lowering its compression latency by 50× and enabling for the first time its use in memory compression. The MemSZ architecture offers aggressive compression ratio (9.3:1 on average) reducing memory traffic and hence improving system performance and energy efficiency. Compared to the previous AVR lossy memory compression approach, MemSZ has a better compressor offering up to 2× better compression ratio, up to 64% lower memory traffic, up to 15% lower execution time, and up to 9% lower system energy; on average the improvement on the above metrics is 23%, 23%, 7%, and 3%, respectively. Finally, MemSZ with a DRAM cache

(MemSZ-DC) improves baseline execution time, energy, and off-chip memory traffic by up to 57%, 33%, and 89%, and on average by 19%, 11%, and 71%, respectively.

REFERENCES

[1] NVIDIA. 2015. NVIDIA Tegra X1: NVIDIA's New Mobile Superchip. *whitepaper*.

[2] Lantmäteriet. 2016. Swedish Topological Survey HDB 50+ Västra Götaland, zone 63_3. Retrieved January 13, 2016 from https://www.lantmateriet.se/.

[3] 2018. 1D K-Means, Open Source. Retrieved October 13, 2018 from https://github.com/eldstal/kmeans.

[4] J. Ahn et al. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*. ACM/IEEE, 336–348.

[5] J. Ahn et al. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*. ACM/IEEE, 105–117.

[6] A. Angerd et al. 2017. A framework for automated and controlled floating-point accuracy reduction in graphics applications on GPUs. *TACO* 14, 4 (2017), 1–25.

[7] A. Arelakis et al. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *MICRO*. IEEE, 38–49.

[8] L. Benini et al. 2002. An adaptive data compression scheme for memory traffic minimization in processor-based systems. In *ISCAS*, Vol. 4. IEEE, IV–IV.

[9] ASCF Center. 2018. FLASH4 User's Guide. Retrieved April 18, 2019 from http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug_4p6.pdf.

[10] V. K. Chippa et al. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *DAC*. 1–9.

[11] S. Di and F. Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *IPDPS*. IEEE, 730–739.

[12] M. Doggett. 2012. Texture caches. *Micro* 32, 3 (2012), 136–141.

[13] J. Dusser and A. Seznec. 2011. Decoupled zero-compressed memory. In *International Conference on HiPEAC*. ACM, 77–86.

[14] M. Ekman and P. Stenstrom. 2005. A robust main-memory compression scheme. In *SIGARCH C.A. News*, Vol. 33. 74–85.

[15] A. Eldstål-Damlin et al. 2019. AVR: Reducing memory traffic with approximate value reconstruction. In *ICPP*. 1–10.

[16] A. Malek et al. 2017. Odd-ECC: On-demand DRAM error correcting codes. In *MEMSYS*. 96–111.

[17] A. Ranjan et al. 2020. Approximate memory compression. *IEEE TVLSI* 28, 4 (2020), 980–991.

[18] B. Panda et al. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *MICRO*. 1–12.

[19] D. Genbrugge et al. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA*. 1–12.

[20] D. Jevdjic et al. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *MICRO*. IEEE, 25–37.

[21] E. Choukse et al. 2018. Compresso: Pragmatic main memory compression. In *MICRO*. IEEE, 546–558.

[22] E. Vasilakis et al. 2019. Decoupled fused cache: Fusing a decoupled LLC with a DRAM cache. *TACO* 15, 4 (2019), 65:1–65:23.

[23] E. Vasilakis et al. 2019. LLC-guided data migration in hybrid memory systems. In *IPDPS*. IEEE, 932–942.

[24] G. Pekhimenko et al. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*. 377–388.

[25] G. Pekhimenko et al. 2015. Exploiting compressed block size as an indicator of future reuse. In *HPCA*. IEEE, 51–63.

[26] H. Jang et al. 2016. Efficient footprint caching for tagless dram caches. In *HPCA*. IEEE, 237–248.

[27] J. San Miguel et al. 2015. Doppelganger: A cache for approximate computing. In *MICRO*. IEEE, 50–61.

[28] Q. Xiong et al. 2019. GhostSZ: A transparent FPGA-accelerated lossy compression framework. In *FCCM*. IEEE, 258–266.

[29] R. Kanakagiri et al. 2017. MBZip: Multiblock data compression. *TACO* 14, 4 (2017), 1–29.

[30] S. Ansumali et al. 2003. Minimal entropic kinetic models for hydrodynamics. *Europhysics Letters* 63, 6 (2003), 798.

[31] S. Baek et al. 2013. ECM: Effective capacity maximizer for high-performance compressed caching. In *HPCA*. IEEE, 131–142.

[32] S. Sardashti et al. 2014. Skewed compressed caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 331–342.

[33] Y. Lee et al. 2015. A fully associative, tagless DRAM cache. In *ISCA (ISCA'15)*. ACM, New York, NY, 211–222.

[34] J. R. Goldschneider. 1997. *Lossy Compression of Scientific Data Via Wavelets and Vector Quantization*. Ph.D. Dissertation. University of Washington.

[35] J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH C.A. News* 34, 4 (Sept. 2006), 1–17.

[36] S. Hong et al. 2018. Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *MICRO*. IEEE, 326–338.

[37] C. Huang and V. Nagarajan. 2014. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *PACT*. ACM, 51–60.

[38] A. Jain et al. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *MICRO*. IEEE, 1–13.

[39] P. Judd et al. 2016. Proteus: Exploiting numerical precision variability in deep neural networks. In *ICS*. ACM, 1–12.

[40] J. Kim et al. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *ISCA*. ACM/IEEE, 329–340.

[41] S. Lal. 2019. SLC: Memory access granularity aware selective lossy compression for GPUs. In *DATE*. IEEE, 1184–1189.

[42] S. Li et al. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*. IEEE, 469–480.

[43] C. Luk et al. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, Vol. 40. 190–200.

[44] N. Muralimanohar et al. 2009. CACTI 6.0: A tool to model large caches. *HP lab.* 27 (2009), 22–31.

[45] M. O'Connor et al. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *MICRO*. 41–54.

[46] B. Panda et al. 2018. Synergistic cache layout for reuse and compression. In *PACT*. 1–13.

[47] M. Pavlovic et al. 2011. On the memory system requirements of future scientific applications: Four case-studies. In *IISWC*. 159–170.

[48] G. Pekhimenko et al. 2016. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *MICRO*. IEEE, 172–184.

[49] J. Quinn Michael. 2004. *Parallel Programming in C with MPI and OpenMP*. Technical Report. ISBN 0-07-058201-7.

[50] B. M. Rogers et al. 2009. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *ISCA*. ACM/IEEE, 371–382.

[51] P. Rosenfeld et al. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL* 10, 1 (2011), 16–19.

[52] A. Sampson et al. 2011. EnerJ: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices* 46, 6 (2011), 164–174.

[53] J. San Miguel et al. 2014. Load value approximation. In *MICRO*. IEEE, 127–139.

[54] J. San Miguel et al. 2016. The bunker cache for spatio-value approximation. In *MICRO*. IEEE, 1–12.

[55] S. Sardashti et al. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *MICRO*. IEEE, 62–73.

[56] S. Sardashti, A. Seznec, and D. A. Wood. 2016. Yet another compressed cache: A low-cost yet effective compressed cache. *TACO* 13, 3 (2016), 27.

[57] V. Sathish et al. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *PACT*. 325–334.

[58] A. Seznec. 1994. Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio. In *ISCA*. ACM/IEEE, 384–393.

[59] A. Shafiee et al. 2014. MemZip: Exploring unconventional benefits from memory compression. In *HPCA*. 638–649.

[60] J. Sohn and E. E. Swartzlander. 2014. A fused floating-point three-term adder. *IEEE Transactions on Circuits and Systems I: Regular Papers* 61, 10 (Oct 2014), 2842–2850.

[61] G. Sun and S. Jun. 2019. ZFP-V: Hardware-optimized lossy floating point compression. In *ICFPT*. IEEE, 117–125.

[62] B. Thwaites et al. 2014. Rollback-free value prediction with approximate loads. In *PACT*. 493–494.

[63] A. Yazdanbakhsh et al. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *TACO* 12, 4 (2016), 62.

[64] A. Yazdanbakhsh et al. 2017. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test* 34, 2 (2017), 60–68.

[65] Y. Zhou and D. Wentzlaff. 2016. MITTS: Memory inter-arrival time traffic shaping. In *ISCA*. ACM/IEEE, 532–544.

Received May 2020; revised July 2020; accepted September 2020