CHALMERS
UNIVERSITY OF TECHNOLOGY

L2C: Combining Lossy and Lossless Compression on Memory and I/O

L^{2}C: Combining Lossy and Lossless Compression on Memory and I/O

ALBIN ELDSTÅL-AHRENS, Chalmers University of Technology, Sweden
ANGELOS ARELAKIS, ZeroPoint Technologies AB, Sweden
IOANNIS SOURDIS, Chalmers University of Technology, Sweden

In this paper we introduce L^{2}C, a hybrid lossy/lossless compression scheme applicable both to the memory subsystem and I/O traffic of a processor chip. L^{2}C employs general-purpose lossless compression and combines it with state of the art lossy compression to achieve compression ratios up to 16:1 and improve the utilization of chip's bandwidth resources. Compressing memory traffic yields lower memory access time, improving system performance and energy efficiency. Compressing I/O traffic offers several benefits for resource-constrained systems, including more efficient storage and networking. We evaluate L^{2}C as a memory compressor in simulation with a set of approximation-tolerant applications. L^{2}C improves baseline execution time by an average of 50%, and total system energy consumption by 16%. Compared to the lossy and lossless current state of the art memory compression approaches, L^{2}C improves execution time by 9% and 26% respectively, and reduces system energy costs by 3% and 5%, respectively. I/O compression efficacy is evaluated using a set of real-life datasets. L^{2}C achieves compression ratios of up to 10.4:1 for a single dataset and on average about 4:1, while introducing no more than 0.4% error.

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

ACM Reference Format:
Albin Eldstål-Ahrens, Angelos Arelakis, and Ioannis Sourdis. 2022. L^{2}C: Combining Lossy and Lossless Compression on Memory and I/O. *ACM Trans. Embedd. Comput. Syst.* 21, 1, Article 12 (January 2022), 26 pages. https://doi.org/10.1145/3481641

1 INTRODUCTION

The rapid increase of connected devices and data produced globally [1] drive numerous applications to become more data-intensive, overwhelming existing computing systems in various domains [2–4]. In high performance computing, server machines in data centers and supercomputers need to handle massive volumes of data supporting Big Data, Cloud Computing, streaming services and many other emerging applications. In the embedded domain, edge and Internet-of-Things (IoT) devices are expected to store, process and communicate data at high data rates under a tight power budget. In turn, the huge sizes and overwhelming rates of data put pressure on the memory and I/O bandwidth resources of systems and often become the bottleneck, limiting performance and wasting energy [5].

One way to alleviate the bandwidth pressure is to improve its utilization with compression. Compressing data towards bandwidth improvement has different requirements depending on the target subsystem. On one hand, in a memory subsystem, compression needs to have low latency, especially during decompression triggered by read accesses, and be effective on small block sizes,

This work is supported by the Swedish Research Council (contract number 2012-4924) under the ACE project.
Authors' addresses: Albin Eldstål-Ahrens, eldstal@chalmers.se, Chalmers University of Technology, Rännvägen 6, Gothenburg, Sweden; Angelos Arelakis, angelos.arelakis@zptcorp.com, ZeroPoint Technologies AB, Gothenburg, Sweden; Ioannis Sourdis, sourdis@chalmers.se, Chalmers University of Technology, Rännvägen 6, Gothenburg, Sweden.

i.e., a few cache lines. Then, it can reduce memory access latency offering faster processing and higher energy efficiency. Commercially available memory compression techniques are mostly application-specific, i.e., GPUs [6, 7]. Other more generic memory compression approaches use a single, lossless or lossy algorithm for compression [8–10]. However, current lossless compression algorithms offer limited compression ratios (on average, between 2x and 4x) [8, 11–15], while a lossy one is only suitable for datasets that tolerate approximations [16, 17]. The trade-off is that lossy compression is able to offer compression ratios as high as 16x [10], making it attractive where supported. On the other hand, compression of data transferred through I/O ports has different design objectives as it strives for high throughput rather than low latency and handles data in larger blocks or in streams. In turn, the combination of latency tolerance and larger block sizes enables higher compression ratios. I/O compression offers better storage utilization and more efficient data transmission improving systems efficiency. I/O compression in embedded systems is often supported by custom hardware, hence is more expensive and with limited applicability, i.e., targeting wireless communications [18]. In the HPC domain, IBM Power9 and Z15 offer user-controlled lossless-only compression acceleration [19], while software-based, hence slower, compression is used for check-pointing traffic [20].

This work describes L^2C , a new holistic compression scheme aiming to utilize more efficiently the bandwidth resources of a processor chip. The main advantage of L^2C is that it combines lossless and lossy compression to best fit the characteristics of different parts of a dataset and improve the impact of compression. In particular, L^2C offers high, lossy compression for data that can be approximated and lower, but lossless compression for data that cannot. Thereby, it is better than previous approaches that offer only lossy or only lossless compression. Combining lossless and lossy compression in the memory system is challenging as they exhibit radically different characteristics, which call for different design requirements. The compression ratio of a lossless method is 4-8× lower than that of a lossy one, as a consequence, using the same memory block size for both would either introduce excessive traffic overheads for the lossless or limit the effectiveness of the lossy method. On the other hand, supporting two different block sizes introduces challenges in the design of the memory system. L^2C addresses these challenges to preserve the benefits of both compression alternatives. Another property of L^2C is that it handles both memory and I/O traffic improving systems efficiency and simplifying integration in the uncore of a chip. However, reusing the same mechanism for memory and I/O compression introduces the challenge of supporting both low latency as well as high throughput compression, while remaining effective in handling small blocks.

In a nutshell, the contributions of this paper are the following:

- The first approach that combines Lossy and Lossless compression algorithms in a memory system. L^2C achieves this supporting:
  - two granularities of memory blocks, tailored to each compression method in order to increase its effectiveness and reduce overheads;
  - a cache structure and main memory layout that can store blocks of both granularities;
  - a mechanism to dynamically select the most suitable compression method;
  - a hybrid metadata format that supports the two methods and in addition is partially embedded with the data to reduce costs.
- Reusing the same compression mechanism for I/O traffic, too, to improve the efficiency of storage and networking functions, which is enabled by compressor designs that offer both high throughput and low latency.

- A thorough evaluation and comparison with state of the art compression techniques showing the benefits of combining lossy and lossless compression as well as the gains of reusing it for compressing I/O traffic.

The remainder of this paper is organized as follows. Sections 2 and 3 discuss related work and background. Section 4 describes the proposed L^{2}C architecture. Section 5 presents our evaluation results and Section 6 draws our conclusions.

2 RELATED WORK

Prior work on related topics is discussed next. First, existing designs for memory compression are presented and subsequently a summary of I/O compression techniques applied in data collection systems is given. Finally, in relation with lossless compression methods, an overview is provided on approximate computing techniques that improve the performance of memory systems.

2.1 Memory Compression

A wide variety of memory compression techniques have been proposed for improving memory capacity and bandwidth utilization. They employ low latency algorithms and suggest different adjustments in the memory system to increase compression efficiency and minimize overheads.

Most existing designs use lossless compression algorithms to avoid introducing changes to the data. Some example of lossless algorithms applied to memory systems use dictionary-based compression [21], exploit frequent patterns and zero-value blocks [22], use similarities of words at the same bit position [8] or offer a hybrid scheme of different lossless algorithms applied to different data [23]. In spite of these varying approaches, lossless solutions have limited compression ratio between 2:1 and 4:1. Leveraging the fact that some applications can tolerate inaccuracies in parts of their data [16, 24], lossy algorithms, such as downsampling [9] and Squeeze [10], were introduced for memory compression to improve compression ratio up to 16:1. However, lossy approaches can be applied only to data that tolerate approximations and limits their applicability.

Besides the algorithm choice, another aspect is the data placement in memory. Some approaches compact compressed data in memory to improve capacity [25]. Others avoid the overheads of data compaction, allocating the worst case storage required for the uncompressed data and focus only on memory bandwidth improvements [9, 10, 26, 27].

Another important design choice is the granularity of the memory block size used for compression, especially when random access in the compressed form of the data is limited or not supported at all. Then, the block size defines a trade-off between the maximum supported compression ratio and the traffic overheads of fetching more data than requested. To exemplify, considering that a cache line (e.g. 64B) is the standard memory access granularity, selecting a block size of eight cache lines defines the maximum compression ratio to be 8:1. However if the average achieved compression ratio is 2:1 then that means that on average a memory access will bring four cache lines on-chip, at the risk of overhead in case of lacking locality. As a consequence, previous lossless memory compression solutions use small blocks of 2-4 cache lines and lossy ones use blocks of about 16 cache lines [9, 10]. Another overhead of larger block sizes is the fact that evicting a cache line from the chip requires the entire block to be present in order to get updated; this adds traffic overheads in case the block misses. In the past, the following two techniques have been used to reduce these overheads: the first one stores recently compressed blocks in the Last Level Cache of the processor and the second uses unoccupied memory space to evict dirty cache lines in their uncompressed form, postponing the recompression of the block [9, 10].

Finally, managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [28, 29]. One approach is to

employ a metadata table and a cache of it, as in [9, 10, 25], which is updated with the TLB and adds a few bytes of bandwidth overhead at every TLB miss. Techniques like Attache [28] aim to further reduce the metadata cost by embedding the metadata directly in the compressed block.

L^2C strives to improve bandwidth utilization while avoiding compaction in main memory. Note that on the contrary, compaction in storage and networking I/O devices is one of L^2C objectives. L^2C is the first memory compression solution that addresses the challenge of combining lossy and lossless. It does so by adapting the memory system to support two block granularities; one for lossless and one for lossy compressed data. In addition, L^2C employs a mix between the two metadata approaches mentioned above, with *essential* metadata kept in a table along-side the TLB while *non-essential* metadata are embedded in the compressed block.

2.2 Link Compression

Compression has been a key technique for reducing I/O traffic in embedded as well as in HPC systems. The main design objective is high throughput and in the case of embedded systems low power is an additional requirement.

In distributed embedded data collection systems and IoT devices, compression fills a critical role due to tight constraints on power, communications and computational resources. Lossless compression has been applied to reduce the volume of off-device traffic [30], by exploiting application specific data properties [31], deduplication [32], prediction [33], and similarities between concurrent data streams [34]. General-purpose compression algorithms such as LZW have proved prohibitively expensive for such low-power devices [35] due to their excessive energy costs. A number of compression schemes have been proposed for embedded applications, utilizing data transformations [36], correlating multiple data sources [37], identifying particularly interesting (i.e. irregular) measurements [38], automatically adapting compression parameters to data features [39]. Moreover, a hybrid lossy and lossless scheme [18], the combination of which in I/O compression does not entail the challenges discussed for the memory compression counterpart.

In HPC applications, software-implemented lossy stream compression has been applied to high-volume I/O traffic without latency constraints [20] to alleviate the performance, energy and storage costs of saving checkpointed data. Moreover, IBM Power9 and z15 provide a user-controlled compressor accelerator in their DMA engine [19] to reduce data volume of DMA transfers.

In summary, embedded I/O compression techniques are mostly custom hardware designs, which increases the cost of the system and often limits their applicability to the particular targeted class of I/O devices. In the HPC domain, compression solutions are in some cases software-based, hence slower and less energy efficient, and in all cases controlled in the user space therefore cannot be exploited at regular memory and I/O operations.

L^2C exposes its proposed memory compression technique to compress I/O traffic, too, in order to alleviate I/O bandwidth pressure and improve the efficiency of storage and networking systems functions. L^2C compression is generic, hardware accelerated and handled in a transparent way without user explicit control. Finally, reusing the same compression mechanism for memory and I/O saves systems energy and area.

2.3 Approximate Computing

The aforementioned lossy compression approaches can be considered part of the broader topic of Approximate computing as they introduce approximations to the data they handle. As such, they share in common some aspects such as the mechanisms for handling errors and identifying opportunities for approximation. Below, approximate computing techniques for improving the memory system are discussed.

Large classes of applications are inherently tolerant to approximations [16]. This enables a trade-off between the quality of their results and their performance and energy efficiency. This trade-off is exploited by various approximate computing techniques, such as computation acceleration [40], memoization [41], limited fault recovery [42], and data storage [43, 44].

Several approximate computing techniques target memory system bottlenecks. Approximate load value prediction reduces memory latency by predicting rather than fetching a value from memory [45–47]. Reducing the precision of floating point [48–51] and fixed point [52, 53] numbers has been used to alleviate the memory bandwidth bottleneck in deep neural networks [52], GPU workloads [49–51, 54] and other approximation tolerant applications [48] 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. Furthermore, Doppelgänger proposed to deduplicate similar cache lines to compress data [55].

A combined approach has been proposed to increase the compression ratio offering the option to reduce precision of individual values by truncating bits and then apply lossless compression on top [49]. The compression ratio remains at roughly 2:1, due to the limited impact of single-value precision reduction and is similar to existing lossless compression schemes, offering little benefit to outweigh quality loss of approximation. Precision reduction is distinct from full lossy compression, in that it only trivially reduces storage size for each individual value rather than identifying inter-value redundancy. Furthermore, the proposed design is implemented in a GPU architecture. While GPGPU techniques extend application support beyond graphics, it is nonetheless limited. L^{2}C takes a different approach, supporting lossless compression along-side more aggressive lossy compression in a general-purpose processor, as well as dynamically switching between the two. This is a more complex problem, due to the differing properties of the two compression methods.

In the past, applications [16] and (parts of) datasets [24] that tolerate approximations have been identified. Past lossy memory compression techniques used error thresholds for maintaining the introduced approximation error in check [9, 10] and evaluated the final error caused to the application output. They also kept track of the accumulated average error per block to limit the effect of repeated approximations on the same data [10]. L^{2}C follows the same approach for handling the error introduced by lossless compression.

3 BACKGROUND

L^{2}C takes its basis in two existing compression systems: the lossy MemSZ [10] and the lossless SC^2 [14]. Lossless compression is safe to apply to all application data, but generally offers limited compression ratio. Lossy compression is only applicable to select portions of data, but provides significantly higher compression potential. By combining these two approaches, L^{2}C is able to reap the benefits of both. In this section, the two existing systems are described.

3.1 MemSZ

Memory Squeeze (MemSZ) applies lossy compression to parts of the application data, which can tolerate approximation [10]. Thereby, it reduces the volume of data transferred between main memory and processor chip, improving memory bandwidth utilization. The main component of MemSZ is a compressor and decompressor between the last level cache (LLC) and the memory controller of a processor.

Similar to most techniques that focus on data approximations [9, 48, 55, 56], data regions are annotated *approximable* by the programmer, using a specialized system call. Allocated pages are marked as approximable using one extra bit for every entry in the page table and translation lookaside buffer (TLB). The programmer also specifies two acceptable *error thresholds* for approximable data. One threshold limits the allowable error introduced in any single compression event, the

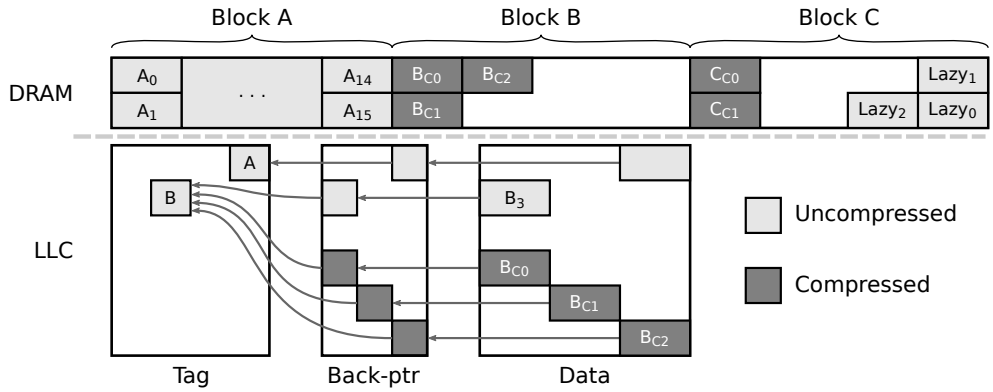


Fig. 1. MemSZ's Decoupled Sectored Cache and main memory block layout.

other limits the total accumulated error across the full application lifetime. Like other memory compression works [9, 25, 57], metadata information for compressible memory blocks is stored in a Metadata Table (MT) in main memory and cached on-chip (CMT). CMT is accessed in parallel with the LLC and updated together with the TLB. Application data which are not marked as approximable are not compressed. MemSZ does not aim to improve memory capacity. Consequently, each block is allocated enough space to remain uncompressed, and therefore memory allocation is not affected. Compressed blocks leave empty memory space between them, which remains uncompacted.

In order to achieve compression ratios of up to 16×, MemSZ applies compression at the granularity of a 1kB (16 cache lines). However, this introduces a number of challenges. The compression prevents random access to single cache lines embedded in compressed blocks, so a memory access triggers accessing the entire block. In addition, LLC evictions are burdened with the overhead of fetching and recompressing blocks. MemSZ addresses these challenges by (i) co-locating compressed memory blocks and uncompressed cache lines in the Last Level Cache (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 regular uncompressed cache lines, MemSZ employs a Decoupled Sectored Cache [58], as illustrated in Figure 1. A layer of indirection (*back-pointers*) allows a single tag to represent a block of multiple consecutive cache lines. MemSZ extends this design to store any combination of compressed memory subblocks (CMS) and uncompressed cache lines (UCL) under a shared tag.

A request to the LLC may hit in three distinct ways, with increasing latency: (i) in the buffer of the compressor, which stores the most recently decompressed block; (ii) in the LLC as an UCL, (iii) in the LLC as part of a compressed block. In the latter case, the block must be read out of the cache and decompressed, introducing additional latency compared to a regular cache hit. Otherwise, when the cache line misses, 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 from memory 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 regular, 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 on-chip in order to be updated. To reduce the overhead of such full-block fetches, MemSZ employs *lazy evictions*. The single dirty UCL

is written back to memory, utilizing the space left empty after the end of the compressed block. Figure 1 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. Lazy eviction allows MemSZ to postpone the costly recompression and mitigate its traffic overhead.

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. It is used to delay recompression until a sufficient number of updates have been carried out, with an exponential back-off. The metadata of a block also includes its compressed size, number of lazy evicted cache lines, and the total accumulated error of each block.

The compression algorithm used by MemSZ is chosen for its high compression ratio and designed for fast decompression. Compression is based on SZ [20], modified for a fixed block size and increased parallelism. Individual values which exceed a set *error threshold* are embedded in the compressed block, ensuring that each recompression meets the set threshold. This allows a variable compression ratio ranging from 2× to 16×.

MemSZ reduced memory traffic up to 81% improving system performance and energy efficiency up to 62% and 25%, respectively, introducing less than 2% application output error.

3.2 Statistical Cache Compression

Statistical Cache Compression (SC^2) is a lossless cache-compression scheme, rather than main memory, which is based on type-agnostic huffman-encoding [14]. A global *Value Frequency Table* (VFT) is populated during a sampling phase at the start of execution, forming the basis for an encoding tree. This encoding tree is then used to compress cache lines before they are written to LLC, increasing its capacity.

During the sampling phase, the VFT is populated by observing the last-level cache. The VFT is a set-associative cache structure, indexed by data values. It stores occurrence counters for the set of most frequently seen values. When a line is updated in LLC, each individual value in the cache line is *added* to VFT, i.e., its counter is incremented. When a cache line is evicted from LLC, each value in the line is *subtracted* from VFT, i.e. its counter is decremented.

Since the VFT is of finite capacity, not all possible values can be present at the same time. Newly observed values are inserted in the VFT, replacing the least-frequent value in its set. A special counter labeled *OTHER* is maintained with the sum of all replaced counts. This is used as the frequency of any data value not explicitly present in the VFT.

When the sampling phase ends, the frequencies collected in VFT are used to build a huffman tree, assigning variable-length codes to each of the observed data values. This process assigns shorter codes to the most frequently seen values, based on the assumption that common values during sampling will remain common during the rest of execution.

During execution, any line to be inserted in the LLC is compressed using the generated encoding. Known values are replaced with their variable-length code. Values not assigned an explicit encoding are stored as-is, prefixed by the code assigned to *OTHER*. The global state (VFT) being shared between all compressed blocks removes the need to embed the huffman dictionary in the compressed block. This allows SC^2 to be applied to blocks of arbitrary size, with no reduction in compression efficiency.

4 SYSTEM ARCHITECTURE

L^{2}C is a hybrid compression scheme which combines lossless compression with more aggressive, lossy compression. Lossy compression has the potential for higher compression ratios, but is limited to data annotated by the developer as *approximable*. Lossless compression offers more modest

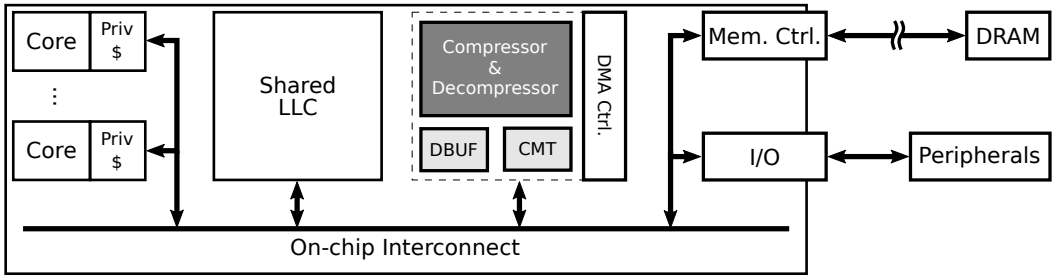


Fig. 2. Top-level view of the L^{2}C memory compression architecture. The compressor module is placed next to the DMA controller, with access to the on-chip interconnect.

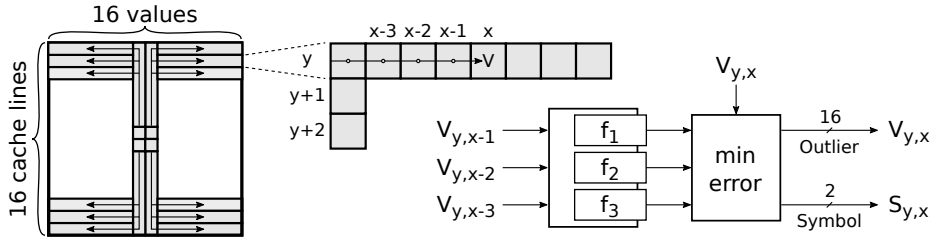
benefits, but is safe to apply to all data, even as a fallback for approximable data. The hybrid nature of L^{2}C offers benefits over either approach. Lossless compression is available for all data. For data which is marked approximable, lossy compression is employed as a primary technique. If lossy compression fails due to quality constraints, L^{2}C falls back to lossless compression. This approach makes L^{2}C applicable and beneficial to any application able to tolerate lossy memory compression.

L^{2}C adds a hardware compressor in the uncore of a processor chip as depicted in Figure 2. It uses the MemSZ [10] and SC^2 [14] compression methods for lossy and lossless compression, respectively. The L^{2}C compressor module includes a buffer that stores the most recently decompressed data (DBUF) and a cache of the metadata table (CMT) to handle the compression/decompression process. Similar to MemSZ the LLC is designed as a decoupled sectored cache able to store compressed blocks alongside the normal uncompressed data. Moreover, the L^{2}C LLC and memory support two block type of different granularity to fit the requirements of the two compression modes.

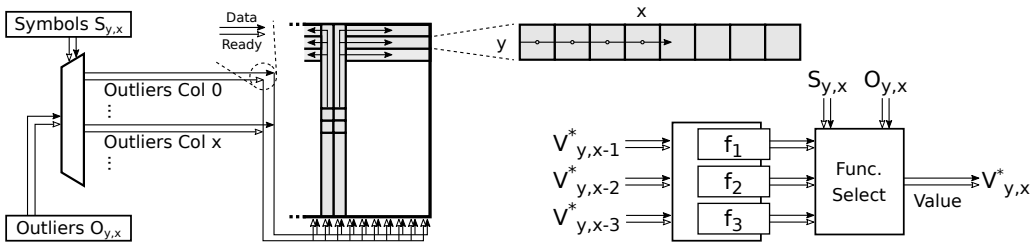
The compressor is located next to a Direct Memory Access (DMA) controller and connected to the on-chip interconnect allowing it to interact with data transfers between the Last Level Cache (LLC), Memory controller and system I/O ports. This placement allows both memory and I/O compression. In turn, this enables L^{2}C to use the same compressor for both memory and I/O compression, the latter case controlled by the DMA.

Briefly, a memory access in the L^{2}C system, is handled as follows. L^{2}C extends the page table to include metadata information about the allocated pages, including the annotation of *approximable* pages, in other words pages that can be compressed in a lossy manner. A memory access is marked as approximable or not after the TLB access. Metadata is read out in parallel with the LLC being accessed. At the LLC, an access may hit either compressed or uncompressed data; otherwise (LLC miss), an access to the main memory is triggered. The metadata indicates the size and compression state of the fetched data. Moreover, LLC evictions are handled lazily by first attempting to update the block if it resides in the LLC; if not, an uncompressed write-back is attempted, if compression has left any unused space, otherwise, the block is fetched from memory to be recompressed.

In general, data in memory are grouped into larger blocks of multiple cache lines. These blocks are kept in memory in compressed form. When a dirty cache line is evicted from the LLC, the compressed block it belongs to is eventually updated to include the fresh data. At maximum compression ratio, a block of 1kB (16 cache lines) fits in 64B (one cache line). Moreover, L^{2}C can automatically downgrade blocks from lossy to lossless compression in cases where insufficient precision can be preserved. This allows the benefits of compression to be retained, at a reduced level, rather than leaving the data uncompressed. Finally, blocks which are not explicitly marked as approximable are only compressed losslessly.



(a) Lossy MemSZ compression. Every 32-bit input value V is replaced by a 2-bit symbol S.



(b) MemSZ decompressor. Outlier placement (left) is carried out in parallel with symbol decompression (right). Dataflow signalling allows decompression to take place out of sequence.

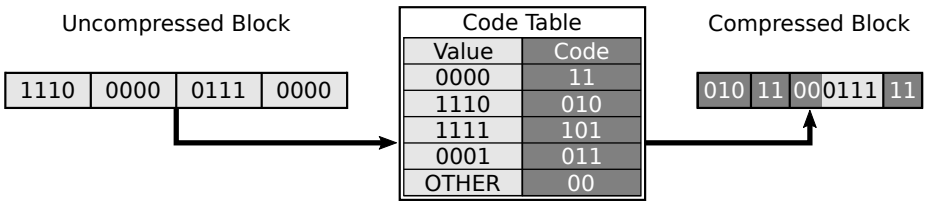
Fig. 3. Lossy compression scheme employed by L^{2}C. A data block is processed as several parallel sequences. Each value in a sequence is encoded as a function of the preceding values.

In the remaining of the section, we describe the system design in more detail. First, the design of the compressor is presented. Subsequently, the L^{2}C memory block format and memory layout are discussed. After that, it is explained how transition between block types are handled, and metadata information is organized. Then, the design of the last level cache (LLC) is described. Finally, the L^{2}C I/O compression support is explained.

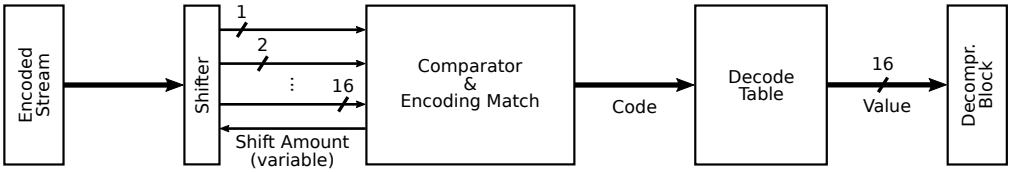
4.1 Compression Methods

The main feature of L^{2}C is the application of two separate compressors, unified in a hybrid design. In this article, we present and evaluate using the MemSZ lossy compressor [10] and the SC^2 lossless compressor [14]. MemSZ represents the state of the art in lossy memory compression, offering compression ratios of up to 16×. SC^2 is designed for cache compression, which requires low latency and hardware complexity. These features also make it suitable for memory compression. Both parts of the L^{2}C compressor are pipelined allowing high throughput. Without loss of generality, L^{2}C can be implemented using any combination of block compressors. It is also trivial to extend L^{2}C to support multiple lossy or lossless compressors and choose the most successful method for any given block.

4.1.1 Lossy compression. The lossy part of the L^{2}C compressor is based on the SZ lossy compression algorithm [20], which compresses sequences of values by describing each consecutive value as a function of the preceding values. This is done by computing three different fixed functions (constant, linear or polynomial), comparing their respective error and selecting and storing the best option (two bits) in place of the value (32 bits). MemSZ introduces several performance improvements to SZ and applies it to 1kB blocks for memory compression [10]. Data blocks are processed in a



(a) SC^2 compression of 4-bit values. More common values are assigned shorter codes.



(b) Decompression of 16-bit values. A comparator identifies a single valid code from the front of the bitstream.

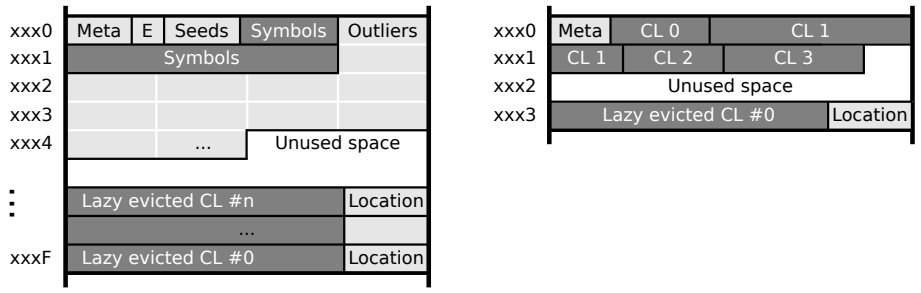
Fig. 4. Lossless SC^2 compression scheme employed by L^{2}C.

square arrangement, allowing for greater parallelism both during compression and decompression as illustrated by Figure 3. The maximum achievable compression ratio for a 1kB block is 16 : 1.

The process of lossy compression of a 16 cache line block, outlined in Figure 3a, is designed to maximize parallelism. The 16 cache lines are arranged as rows in a square block. Four *seed* values are taken from the center of the block. The block is divided into 32 parallel sequences, starting vertically from the seeds in both directions and then spreading out toward the sides. Within each sequence, the compressor attempts to describe each value V strictly as a function of the preceding three values. If one of the available functions (constant, linear, or polynomial) successfully describes the value, a two-bit *symbol* identifying that function is enough to represent the value. If none of the functions is successful, the value is an *outlier*, and is marked by a special symbol. The outlier value itself is stored at reduced precision (16 bits) in the compressed block. After this process, the completed compressed block consists of the seed values, the set of two-bit symbols and a collection of all identified outlier values. Compression of 1kB is completed in 16 cycles.

Decompression is illustrated in Figure 3b. It is optimized for minimal latency, and carried out in two parallel processes: Distribution of outlier values and decompression of symbols. Distribution of outliers is performed by decoding the sequence of symbols, identifying the location of outlier values, as well as their order. The outlier values are first assigned to their proper column. Each column is then populated, starting with the most critical center and progressing outward. The decompression of the two-bit symbols is performed in the same order as they were compressed; seed values are placed in the center of the block and 32 parallel sequences spread out vertically. Outliers may be placed throughout the block out of synchronization with these sequences, and the three decompression functions introduce differing dependencies and latencies. To exploit these irregularities, a dataflow-enabled pipeline design is used. Any one value to be decompressed is processed as soon as all its dependencies are in place. The variable decompression latency of a block, which is critical for memory reads and thus for performance, is at most 16 cycles.

4.1.2 Lossless compression. The lossless L^{2}C compressor is based on the Statistical Cache Compressor (SC^2) [14], which employs huffman-encoding. SC^2 is an inter-block compression scheme



(a) L-block compressed using MemSZ. (b) S-block compressed using SC^2.

Fig. 5. L^{2}C Memory Block formats. Large blocks (*L-blocks*) are lossily compressed, Small blocks (*S-blocks*) are losslessly compressed.

that uses a single, global, symbol table to establish the encoding, as described in Section 3.2. Hence, it does not need to add any other overhead per block and is therefore well suited to compressing blocks of arbitrary size. Figure 4 illustrates an example SC^2 compression operation, where each value of the uncompressed block (4 bits in Figure 4a) is looked up in the Code Table and replaced by the associated code. If the value is not found, it is maintained in uncompressed form preceded by the code for OTHER. The compression outcome is a compressed block of variable width. L^{2}C applies SC^2 compression using 16-bit value symbols and offers compression ratios of up to 4 : 1. SC^2 compression employs canonical Huffman codes: the codes follow the numerical sequence property, i.e., codes of the same length are numerically sequential. This is important during decompression.

The lossless L^{2}C decompressor is also based on the SC^2 decompressor [14] and is depicted in Figure 4b. Decompressing Huffman-encoded streams is inherently sequential because coded values are of variable length, thus it is not known where the next coded value starts in the encoded stream. Importantly, Huffman codes follow the prefix property, i.e., a code cannot be prefix of another code. Hence, when a bit sub-sequence matches a code, the next bit in the encoded stream determines the beginning of the next code.

The SC^2 decompressor works as follows: Part of the compressed block is inserted to a shifter. The 16 most significant bits of the bit-sequence within the shifter are inserted to the Comparator and Encoding Match engine. For each code length (1b, 2b, 3b, ..., and 16b), this engine performs numerical comparisons of the inserted bit sequence and the base value of the respective code length (i.e., the first assigned code for this length). A code of length x is matched within the bit sequence, when the comparison of x bits yields true result and the comparison of x+1 bits yields false. The matched code length determines the shift amount in the shifter and decoding can proceed with the next coded value in the stream. In parallel, the matched code is looked up in the Decode Table and the associated value is output and attached to the decompressed block. This process is repeated until all values are decompressed in the block. The decompression latency is 14 cycles per cache line at 1GHz, parallelizable for larger blocks.

4.2 Block Types

The two compression schemes employed by L^{2}C differ in their utility and application. The lossy compressor is geared toward high compression ratios, necessitating large blocks. This is in part due to a fixed per-block data overhead, in the form of *seed values* which must be included uncompressed in the compressed block. The lossless compressor, by contrast, has no such fixed overheads. Its

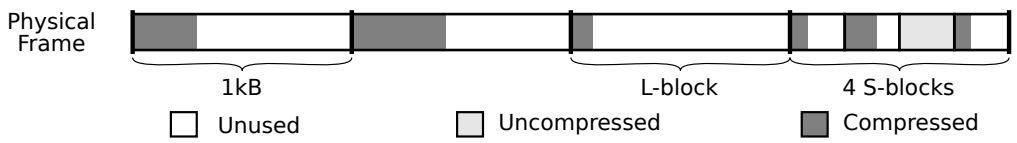


Fig. 6. Main memory with a mixture of block types. Each 1kB space is one L-block or four S-blocks.

compressed blocks consist only of re-encoded values from the original data. This allows it to be applied to blocks of any size.

The optimal block size for any memory compression scheme depends on two factors: the maximum achievable compression ratio and the minimal transfer unit of the memory bus. An undersized block may compress to a size smaller than the minimal transfer unit, leading to transfers larger than necessary. Conversely, an oversized block may compress below expectation, leading to extraneous data transferred. For these reasons, the optimal block size is such that the maximum expected compression ratio results in a compressed size equal to the minimum compression size.

The minimum transfer unit of a typical system is one cache line. The lossy compression employed by L^2C is designed for a maximum compression ratio of 16 : 1, and is thus applied to blocks of 16 cache lines. We refer to these large blocks as *L-blocks*. The lossless compression using 16-bit values has a theoretical maximum compression ratio of 16 : 1 (compressing each 16-bit value to a single-bit encoding), but typically achieves compression ratios between 2 : 1 and 4 : 1 on non-constant data. For this reason, L^2C applies lossless compressed data to blocks of 4 cache lines. We label these small blocks *S-blocks*. An *S-block* is a quarter of an *L-block*, which is convenient for their alignment and management. As L^2C combines these block types, a 1kB region of memory can either be one *L-block* or four *S-blocks*. Figure 5 illustrates the format of each block type. Both types contain a small amount of embedded block metadata, which is further described in Section 4.5.

The L-block is specifically organized to allow decompression to begin as soon as the first line is available. A single bit *E* indicates that the rest of the line has been losslessly encoded to save space. This is followed by a set of *seed* values, from which all SZ sequences begin. The first line also contains an initial set of two-bit *symbols* representing compressed values, as well as a number of *outliers* sufficient to start decompressing the center columns of the block. The remaining lines of the compressed block contains the rest of the symbols and any remaining outliers. The S-block format is simpler, consisting only of the compressed cache lines.

Both types of blocks leave unused space at the end of their allocation in physical memory, which is used for *lazy evictions*. When a compressed block is only available off-chip, any dirty uncompressed cache line evicted from LLC will be stored in this space. In order to reconstruct a block with lazily evicted cache lines, the location of each dirty line must be maintained. For approximable data, data precision is reduced by a few bits to encode the proper location of the cache line. In non-approximable data, the evicted cache line is compressed and the location information is appended to the end of the cache line.

4.3 Memory Layout

The use of multiple compression schemes with differing block sizes necessitates a flexible memory layout for compressed data. A memory location may be in one of three different states:

(1) Compressed lossily as part of a 1kB L-block
(2) Compressed losslessly as part of a 256B S-block
(3) Uncompressed as part of an uncompressed 256B S-block

L-blocks are aligned to 1kB boundaries while S-blocks always appear in groups of four, each aligned to 256B. Figure 6 illustrates L- and S-blocks coexisting in physical memory. This alignment serves dual purposes. First, the address of a cache line can be trivially translated into the physical address of the corresponding compressed block. Second, it allows an L-block to transition into four S-blocks if lossy compression fails, without affecting neighboring blocks outside the 1kB allocation. This type of transition is central to L^{2}C, enabling a fallback to less aggressive compression rather than leaving data uncompressed.

4.4 Block Type Transition

During the execution of a program, the same memory region may be dynamically selected to be compressed in a lossy or lossless manner as long as it is indicated to be approximable. The transition between lossy L-blocks and lossless S-blocks is described below.

When lossy compression of an L-block is attempted and fails, MemSZ leaves the full block uncompressed. This leads to wasted compression potential, since the data may still exhibit some amount of redundancy. L^{2}C leverages this potential by transitioning the L-block into four S-blocks and applying lossless compression. In effect, data compressibility determines a block's place within a hierarchy of compression states, from lossy L-block via lossy S-block and down to completely uncompressed S-block. Figure 7 illustrates the logic governing transitions between these states.

Uncompressed data may, with updates, become compressible again. SC^2 compression is applicable to blocks of any size, and L^{2}C uses this property to determine the compressibility of individual cache lines. A *back-off* counter ① associated with uncompressed S-blocks keeps track of the number of individual and compressible cache lines written back to the block. When the counter reaches its maximum, the S-block is expected to be compressible and a transition ② is attempted.

Analogously, after some number of updates to a compressed S-block, it is possible that compressibility changes and lossy compression becomes viable. L^{2}C uses the lossless compressibility of the S-blocks as an indicator for this (Figure 7). Every group of four S-blocks shares a *transition count* ③, which is incremented when a compressed S-block is written back to memory. If any S-block fails compression, the transition count is cleared. Once a sufficient number of consecutive lossless compression attempts have been successful, a transition ④ to a single L-block is attempted.

Transition to a lower compression state (i.e. L-block to S-Block or S-block to uncompressed data) is straight-forward. Such a transition occurs only when compression fails, and thus all data is already available on-chip. Conversely, any transition toward a higher compression state involves reading multiple cache lines from memory, in order to compress a larger block. In the worst case, this consists of three compressed S-blocks totalling nine cache lines. To reduce this traffic overhead, L^{2}C postpones the transition attempt until the next cache miss for this block. Because miss resolution requires one uncompressed cache line or one compressed S-block from memory, this reduces the total overhead of the transition. In addition, any compressed blocks which are already on-chip in the LLC do not need to be transferred.

4.5 Block Metadata

One hurdle faced by memory compression systems is the overhead of metadata. Certain information about a compressed block may be necessary in order to manipulate the block in memory or bring it on-chip for processing. This additional information is too large to keep on-chip in its entirety, and must therefore be stored in main memory.

To reduce the traffic overhead of such metadata, L^{2}C divides the compression metadata into two categories. *Essential* metadata are necessary even when the corresponding block is not on-chip, in order to fetch or update it. *Non-essential* metadata are only needed once the block is on-chip, and are embedded in the compressed block as illustrated in Figure 5.

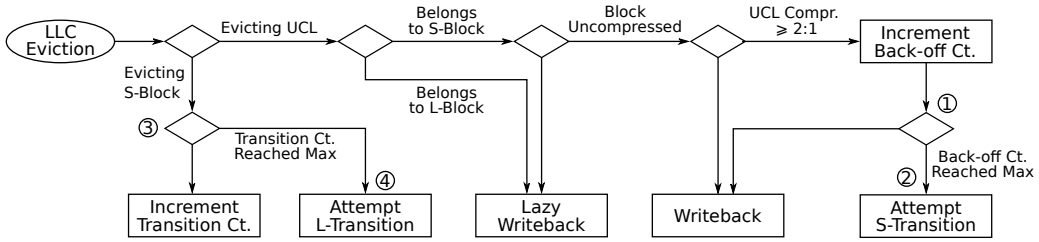


Fig. 7. Back-off and transition behavior. Data transitions from Uncompressed to lossless S-Block to lossy L-Block as compressibility tests succeed.

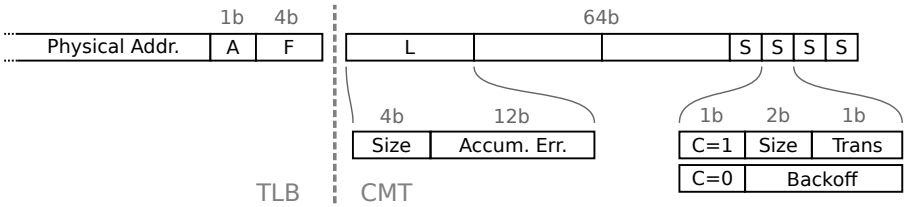


Fig. 8. Metadata table format. S-block metadata is encoded differently for Compressed (C=1) and Uncompressed (C=0) blocks.

Non-essential metadata are only needed when the full compressed block is also available for processing. This information consists of the size of the compressed block excluding lazily evicted cache lines, which is necessary in order to decompress the block. In addition, L-blocks encode the *compression method* used, to be able to differentiate between data types and potentially support other compression schemes.

L^{2}C uses a Compression Metadata Table (CMT) as an on-chip cache for essential compression metadata. CMT has a structure corresponding to the existing Translation Lookaside Buffer, and is updated in tandem with it on TLB misses. Each quarter-page is described either as one L-block or four S-blocks. Four unused bits (labeled F) in the regular Page Table Entry (PTE) are used to encode this state. An additional TLB bit is used to mark approximable pages. A CMT entry comprises 64 bits for one page, and is organized as illustrated in Figure 8.

S-blocks are afforded four bits of CMT space. These four bits are used to encode three fields: a two-bit *size* field, a 1-bit *transition* counter (described below), and a 3-bit *back-off counter* used to delay compression for uncompressed blocks. Since the size and transition fields are only needed for compressed blocks and the counter is only needed for uncompressed blocks, these two sets are overlapped. A single bit *C* is used to distinguish between the two states.

L-blocks have 16 bits of essential metadata, divided into two fields: a four-bit *size* field and a twelve-bit counter of *accumulated error*. The twelve-bit counter is a floating-point (4-bit exponent and 8-bit mantissa) representation of the accumulated error introduced by lossy compression.

4.5.1 Metadata during transitions between block types. Metadata encoding is complicated by the multiple compression states a single block may have. One cause of transition is a failure to compress. L-blocks which fail lossy compression transition into four S-blocks. S-blocks which fail lossless compression transition into uncompressed data. The opposite transitions are carried out when compression is retried successfully. These retries are controlled using *back-off counters*.

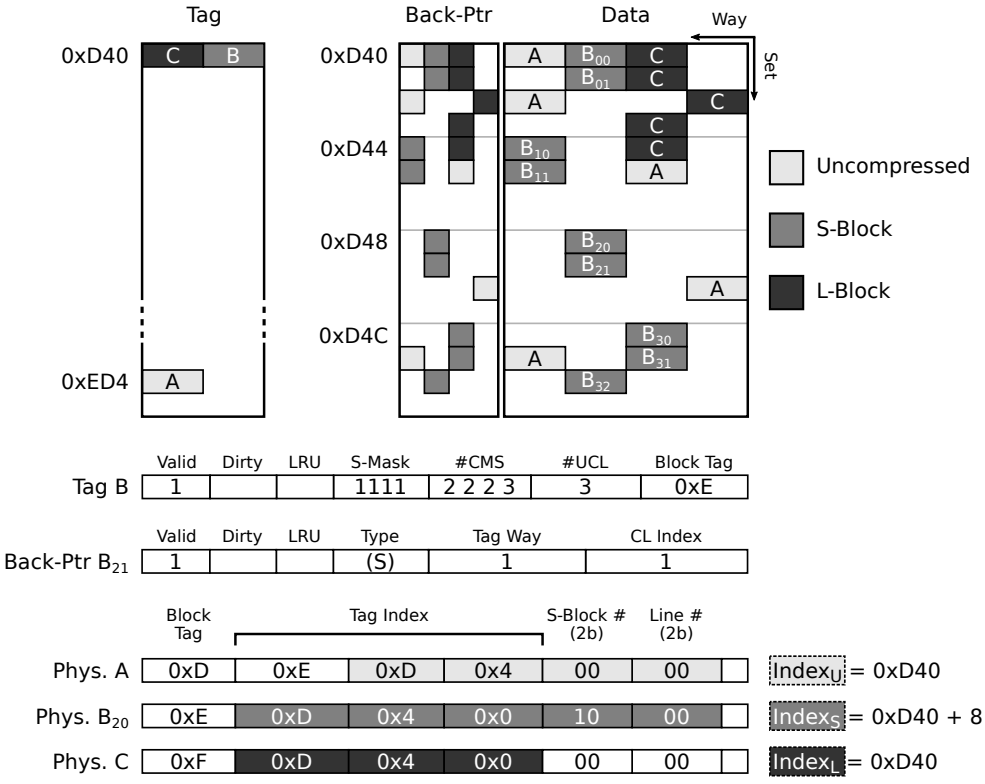


Fig. 9. Conceptual view of the L^{2}C decoupled sectored cache and its three data indexing functions. S-blocks are placed at 4-set intervals.

Transition from uncompressed to compressed S-block is tracked using the metadata for S-blocks, as discussed above. When an uncompressed eviction occurs, the compressibility of the cache line is tested. The *back-off* counter of the corresponding S-block is incremented if the evicted line has an individual compressibility at or above 2 : 1.

Transition from S-block to L-block (for pages annotated as approximable, i.e. allowing L-block lossy compression) is controlled by four *transition* bits spread out across the metadata of the S-blocks. These bits encode a counter of consecutive successful S-block compression attempts, indicating that the data is compressible. Overlapping the metadata bits this way works, since the *transition* counter is only valid if all four S-blocks have been successfully compressed.

The *Accumulated Error* counter associated with an approximable L-block must be maintained even when the block temporarily transitions to S-blocks or is left uncompressed due to failed compression. This is done by including three bits of the counter in the *non-essential* metadata embedded in each S-block, if that block is compressed. If an S-block is uncompressed, the three bits are instead embedded as the least significant bit of each of the first three data words.

4.6 Last-Level Cache

Support for two separate memory block sizes also raises the need for similar support in the last-level cache. Resolving LLC misses by fetching compressed blocks from memory introduces traffic overhead because blocks may be larger than one cache line. In order to benefit from the extra

fetched data, it must be kept on-chip for as long as possible. If the data exhibits spatial locality, the fetched block acts as a form of prefetching, at reduced traffic cost.

L^{2}C uses a Decoupled Sectored Last-level Cache [58] to store compressed and uncompressed data on-chip simultaneously. Tags are decoupled from data entries as illustrated in Figure 9 and associated using a special *back-pointer array*. This allows multiple data entries representing the same 1kB address space to share the same tag. For example, a 1kB region of physical memory may be present in the LLC as one compressed L-block and three uncompressed cache lines, simultaneously. Three separate indexing functions are used for data placement: One for compressed L-blocks ($Index_L$), one for compressed S-blocks ($Index_S$) and one for uncompressed data ($Index_U$).

L- and S-blocks all consist of one or more cache line sized CMSs. All CMSs belonging to a single compressed block are placed in consecutive LLC sets. Since a tag never represents both L- and S-blocks simultaneously, the two use similar indexing functions. If the L-block indexing function $Index_L(A)$ indicates that the compressed data for a tag A should start in set X, then $Index_S(A)$ would also place the first S-block for that same tag starting at X. The second S-block is placed at $X + 4$, the third at $X + 8$ and the last at $X + 12$. This way, L-compressed blocks and S-compressed blocks have similar behavior in the LLC. The indexing functions $Index_L(A)$ and $Index_S(A)$ are chosen to minimize interference between compressed and uncompressed data belonging to the same block, i.e. the uncompressed indexing function $Index_U(A)$ is unlikely to return the same index as $Index_L(A)$ or $Index_S(A)$.

Figure 9 illustrates a slice of the LLC with data from three 1kB memory blocks (A, B and C) present. A is uncompressed, B is compressed as four S-blocks and C is compressed as a single L-block. Their respective physical addresses are such that the indexing functions $Index_U(A) = Index_S(B) = Index_L(C) = 0xD40$, and they thus contend for the same 16 sets in LLC. The uncompressed cache lines from A are placed based on their individual addresses. The four S-blocks $B_0 - B_3$ start at four-set intervals, while the compressed L-block is placed in five consecutive sets starting at $0xD40$. Any compressed data for A or uncompressed data for B and C are placed in other sets..

The LLC supports three types of lookups (Uncompressed, S-Block, or L-Block). Lookups work similarly to a standard Decoupled Sectored Cache. The tag index is computed from the sought physical address. Based on the type of lookup (Uncompressed, S-Block, or L-Block), the corresponding indexing function ($Index_U$, $Index_S$, $Index_L$, respectively) is used to identify the proper set in the back-pointer/data arrays. Tag and BP lookups are then performed in parallel. If a Tag entry and a BP entry are both located, a tag match is confirmed using the *tag way* stored in each BP entry as well as the *block tag* from the physical address. If these comparisons all match, both tag and data have been successfully located.

L^{2}C uses a single tag to represent each contiguous 1kB region of physical memory, in both compressed and uncompressed forms. The tag entry is extended with additional fields to support the two block sizes. A four-bit mask indicates which S-blocks are present in the LLC. An 8-bit counter field is used to indicate the number of data entries present for each compressed block (four 2-bit counters for S-blocks or a single 3-bit counter for an L-block).

Compressed data has the potential to offer greater utility compared to their size. To exploit this, replacements are performed with a modified Least-Recently-Used (LRU) mechanism. When an uncompressed cache line is updated (via write-back from the L2 cache), its LRU is normally updated to record that it has been used recently. If the tag entry indicates that a compressed copy of the same block is present in the cache, the LRU counter of the *compressed block* is updated in stead of that of the UCL. This way, compressed blocks are prioritized over their uncompressed (and redundant) counterparts during cache replacements.

The decoupled sectored cache organization allows L^{2}C to store any combination of compressed and uncompressed data on-chip. The accompanying metadata enables lookups of compressed data,

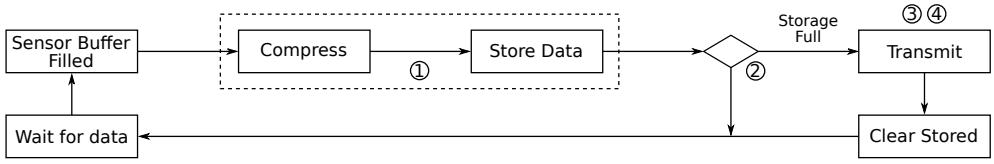


Fig. 10. Execution flow of a data collection application which benefits from compressed I/O.

increasing the effective capacity of the cache. As an additional benefit, this enables the reuse of compressed blocks, thus amortizing their memory traffic overhead.

4.7 I/O Compression

The placement of the L²C compressor, attached to the on-chip interconnect and next to the Direct Memory Access (DMA) controller, also enables the compression of I/O traffic. L²C can direct through the compressor any data transfer between two memory-mapped regions. In DMA-capable systems, the on-chip DMA controller is programmed to initiate the data movement, while in systems without DMA, a processor core performs this task. This covers both data input (e.g sensor devices) and bidirectional devices (e.g. local storage, network interfaces). L²C enables transparent compression at high bandwidth.

I/O-heavy applications which can benefit from compression include data aggregation services and remote sensor networks. These networks typically consist of low-power devices with limited performance and communication resources. Nodes of this type are strongly power constrained, and may rely on a small battery and unreliable power harvesting techniques (e.g. solar cells, RF energy harvesting). For this reason, energy efficiency is a high priority. The device typically spends as much time as possible in a low-power state, periodically waking up to collect and transmit data.

Figure 10 illustrates the execution flow of a simple embedded application. Data is collected and buffered in an off-chip sensor, while the processor itself is in a low-power sleep state. An interrupt wakes the processor when the buffer is full. The processor triggers a data transfer (via DMA or software mechanisms) to bring the sensor data on-chip. The data is stored in persistent storage, and the processor returns to its sleep state. When local storage is full, a batch of data is transmitted via radio for central aggregation. The benefits of data compression in such a system are fourfold:

① Execution time is reduced, allowing longer sleep periods.
② Longer periods of data can be logged in local storage, reducing the frequency of transmission.
③ Radio transmission and relay energy is reduced, due to smaller payloads.
④ Radio bandwidth is saved.

The data transfer from sensor to processor, be it via DMA or software mechanism, is uncompressed at the source, but passes through the compressor after arriving on-chip. As a result, the data is compressed before being written to storage, saving both time ① and storage space ② . In addition, this allows the collection period to be extended before local storage space is exhausted. Once it is, energy ③ and bandwidth ④ savings are compounded; less frequent radio transmissions, each containing more sensor data.

In addition to these benefits, digital sensors for natural phenomena (e.g. air pressure, temperature, pollutants, radiation) have finite precision, introducing some amount of quantization during data acquisition. Lossy compression can be used to exploit this approximation tolerance.

By placing the L²C compressor appropriately, compression can be applied to memory-mapped peripherals such as built-in sensors. Attaching the compressor to the on-chip interconnect as illustrated in Figure 2 also allows compression to be applied to external peripherals.

Table 1. Simulation parameters.

(a) System parameters.

Parameter	Configuration
CPU	4 core, o-o-O, 4-way issue @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	4MB shared, 16-way, 15 cycle latency
Main Memory	4GB DDR4, 1 channels, 800MHz
VFT	7kB, 8-way, 16-bit values

(b) Compressor properties.

Parameter	Compressor	Decompressor
SC^2 latency	18 cycles	42 cycles
SC^2 leakage power	33.6mW	0.4mW
SC^2 dyn. energy	0.576nJ	0.592nJ
MemSZ latency	16 cycles	8-16 cycles
MemSZ leakage power	28.8mW	144.5mW
MemSZ dyn. energy	3.94nJ	17.5nJ

I/O compression differs from memory compression by the property that data is compressed *exactly once*. As a result, block type transitions will never occur. For this reason, the I/O compressor does not need to prioritize L-blocks over S-blocks. Instead, both lossy and lossless compression are attempted, choosing whichever achieves a better compression ratio.

5 EVALUATION

In this section we evaluate the efficiency of L^2C . We first describe our experimental setup, detailing the system configuration of our experiments and the benchmarks used. Two separate evaluations are described: one applying L^2C for memory compression and one for I/O compression. Then, experimental results from each evaluation are presented.

5.1 Experimental Setup

Our evaluation of L^2C is twofold. First, we evaluate its use as a Memory Compression scheme, using a processor and memory simulator. Separately, we evaluate the potential of L^2C as a I/O Compression scheme by applying it to a selection of real-world datasets.

5.1.1 Memory Compression. We evaluated L^2C for memory compression in an in-house simulator, implemented on top of Pin [59]. The simulator employs an interval-based processor model, as proposed by Genbrugge et al. [60]. The memory hierarchy was modelled at cycle granularity, using DRAMSim2 for main memory [61]. McPAT [62] and CACTI [63] were used to model power and latency of the system considering 32nm technology. The MemSZ compression hardware modules were implemented in RTL, synthesized using Synopsys Design Compiler to determine their operating frequency, latency and power consumption; the same parameters for SC^2 are taken from [14] which were measured with the same technology node. These factors are used as configuration information for the simulations. The general properties of the simulated system are listed in Table 1a. The power and latency of each compressor are outlined in Table 1b.

As explained in Section 4, the developer is responsible for the annotation of *approximable* data structures. For this evaluation, we manually add annotations to the source code of each benchmark based on experimentation to find safe approximations. Table 2a summarizes the type of approximated data for each application.

In order to emulate the impact of the approximations on the overall application error, we emulate not only the memory accesses but also update the values of the memory contents accordingly. This is done by applying a software implementation of the compression and reconstruction methods to the data. Lossless compression is applied to all non-code pages mapped into the process. This includes heap, stack, and data segments of the application itself, as well as those of shared libraries.

Besides the baseline system, L^2C is further compared with (i) the lossy-only *MemSZ* [10] and (ii) a variation using only lossless SC^2 compression (*Lossless*). As all three compressing systems use the same decoupled sectored cache design, they are configured identically apart from the employed

Table 2. Workloads used to evaluate L^{2}C .

(a) Benchmark Applications.

Application	Approx.	Output	Footprint / core	Checkp.	Description
heat [64]	Temps	Temps	8.3MB	✓	Heat propagation through a 2D field of uniform material
lattice [65]	P and M	Vel.+Press.	5MB	✓	2D Lattice-Boltzmann simulation of air flow
lbm [66]	Velocities	Velocities	325MB		3D Lattice-Boltzmann simulation of fluid flow
orbit [67]	Phys. data	Phys. data	10MB	✓	3D simulation of the two-particle orbit problem
cdelta [67]	Phys. data	Phys. data	22MB	✓	Delta-function heat conduction model
sedov [67]	Phys. data	Phys. data	12MB	✓	Sedov explosion model
windt [67]	Phys. data	Phys. data	23MB	✓	Windtunnel with a step
kmeans [68]	Topol. [69]	Clusters	5.5MB	✓	Iterative clustering algorithm
wrf [66]	Geo data	Temp.	90MB		Weather forecasting model

(b) Datasets used to evaluate L^{2}C for Link Compression.

Dataset	Domain	Type	Size	Description
height [69]	Geo survey	2D spatial	1024×1024 samples	Geographical height map
aquarius [70]	Geo survey	2D spatial	$8 \times 512 \times 1024$ samples	Sea surface properties
gb6 [71]	Astronomical survey	2D spatial	2048×2048 samples	Radiotelescope imagery
strang [72]	Geo survey	Time series	187176 samples	Solar radiation measurement at $60°N 15°E$
hand [73]	HCI	Time series	80×400000 samples	Hand positions for gesture detection
mitbih [74]	Medical	Time Series	$9 \times 2 \times 650000$ samples	Two-channel ECG recordings
ampds [75]	Energy distribution	Time series	12×1051200 samples	Energy consumption data from a residential building
air [76]	Meteorological	Time series	13×121641 samples	Air quality measurements
gas [77]	Scientific	Time series	19×786432 samples	Carbon monoxide sensor in physics experiment
hydra [78]	Mechanical	Time series	18×1048576 samples	Condition monitoring of hydraulic system

compression mechanism. This similarity allows the isolation of lossy compression, to study its impact compared to a system with only lossless compression capability.

Each simulation is executed in the following steps: i) A *warmup* period of 50M instructions is carried out to warm up the cache hierarchy; ii) at the end of this warmup period, 10% of the compressible system memory is randomly sampled to train the SC^2 and populate the VFT. This emulates a longer sampling period. Furthermore, all compressible data in memory is compressed at the end of the warmup period, simulating an application with compressed input data; iii) the application is executed until it has finished generating output data.

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 2a.

The input data sets used for our experiments are the standard input data sets 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 [69].

Compression metadata has been identified as a significant source of memory traffic [28]. To evaluate this factor, our simulations include both the traffic of regular page table information (via TLB misses) and the additional transfer of essential compression metadata.

Benchmarks for approximate computing (AxBench) considers 10% relative output error [79]. Due to its strongly application-dependent nature, it is solely up to the application provider to define what is an acceptable error level. We evaluate and present output error using the mean relative error across the output dataset. 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, L^{2}C

provides tunable knobs to control the data approximation error and constrain application output error. These knobs allow an application provider to adjust the trade-off between output error and performance/energy improvement. Specifically, two quality thresholds are configurable. One is local to each compression attempt, controlling which values are outliers. The second is maintained over the entire execution time, limiting accumulated approximation error.

5.1.2 I/O Compression. The benefits of I/O compression (reduced execution time, reduced communication duration, reduced communication bandwidth, improved storage efficiency) are directly proportional to the achieved compression ratio. For this reason, we evaluate the use of L^2C as a I/O compression scheme by applying it to a selection of real-world datasets as outlined in Table 2b.

The datasets can be generally divided into two categories: Spatial and Time series. Spatial data represent a snapshot of samples from different locations, such as a topological survey. This type of data is typically seen at centralized collection points, such as coordinating nodes or database servers, where data are collated from multiple distributed sources. Time series represent multiple samples from the same sensor, such as a continuous energy consumption measurement. This type of data is typically seen in the individual sensor node, such as an implanted medical device.

To evaluate the efficiency of L^2C for I/O compression, each dataset is compressed using the three evaluated compression schemes: *Lossless*, *MemSZ* and L^2C. We present the achieved compression ratio of each system as well as the resulting approximation error.

5.2 Results

In the following section we present the results of both evaluations. First, we show detailed statistics acquired from simulations of memory compression. Subsequently, we show the compressibility of the datasets used to evaluate L^2C for I/O Compression.

5.2.1 Memory Compression. The primary characteristic differentiating the various compression schemes is the achieved compression ratio for any given dataset. Table 3a shows the compression ratio of each application's footprint at the end of execution. While neither lossy nor lossless alone show a clear advantage, it is clear that a hybrid approach is able to reap the benefits of each. L^2C consistently achieves a higher compression ratio than either of the two competing designs. Table 3b shows the compression ratio for the approximable subset of the footprint. We observe that lossy compression is up to 7 *times* more effective than lossless compression for the annotated data. MemSZ does, however, leave blocks uncompressed if they fail to meet quality requirements under lossy compression. L^2C falls back to lossless compression for these blocks, achieving a higher overall compression ratio. This effect is most pronounced in *lattice*, where L^2C achieves a 49% higher compression ratio compared to lossy compression alone.

The main benefit of memory compression lies in reduced traffic on the main memory bus. Figure 11c shows the total memory traffic for each design, normalized to the traffic of the baseline system. Traffic is broken down by data type: non-approximable data, approximable data, page table traffic, and metadata traffic. We find that metadata traffic comprises at most 3.9% of total traffic, twice as much as the regular page table traffic. On average, L^2C reduces the total traffic volume by 73%. This is an improvement of 18% compared to MemSZ and 56% over Lossless.

One potential cause of traffic overhead is the transition from multiple S-blocks to a single L-block. To attempt such a transition, multiple S-blocks must be read from main memory. Table 3d shows the fraction of total memory traffic caused by such reads. The maximum 2.2% is found in *lattice*, while the remaining benchmarks see at most a fraction of a percent of overhead.

The reduced traffic on the main memory bus yields lowered latency for memory accesses, which is particularly important for memory reads. Figure 11d shows the Average Memory Access Time

Table 3. Compression efficacy of the three memory compression systems.

(a) Compression ratio, all data.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf	GM
Lossless	2.5×	2.5×	2.2×	3.1×	2.9×	3.4×	2.7×	1.9×	1.5×	2.4×
MemSZ	1.5×	1.1×	4.8×	1.8×	1.1×	1.3×	1.0×	1.3×	1.2×	1.5×
L^{2}C	3.2×	2.6×	7.2×	4.1×	3.1×	4.3×	2.8×	2.5×	1.6×	3.2×

(b) Compression ratio, approximable data.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf	GM
Lossless	2.8×	1.9×	2.2×	3.7×	2.6×	3.7×	2.1×	2.3×	3.0×	2.6×
MemSZ	15.9×	5.1×	15.9×	14.9×	9.2×	15.8×	15.9×	3.6×	4.4×	9.6×
L^{2}C	16.0×	7.6×	15.9×	14.9×	9.2×	15.8×	15.9×	3.9×	5.3×	10.4×

(c) Mean relative application output error.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf
Lossless	0%	0%	0%	0%	0%	0%	0%	0%	0%
MemSZ	0.12%	0.24%	0.05%	0%	0.01%	0%	0%	0.05%	<0.01%
L^{2}C	0.13%	0.25%	0.06%	0%	<0.01%	0%	<0.01%	0.05%	<0.01%

(d) Fraction of memory traffic caused by L^{2}C block transitions.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf
L^{2}C	0.000%	2.207%	0.000%	0.000%	0.006%	0.000%	0.000%	0.001%	0.064%

(AMAT) for instructions with memory input operands, normalized against the baseline AMAT. On average, L^{2}C reduces baseline AMAT by 36%, improving on MemSZ by 5% and Lossless by 17%.

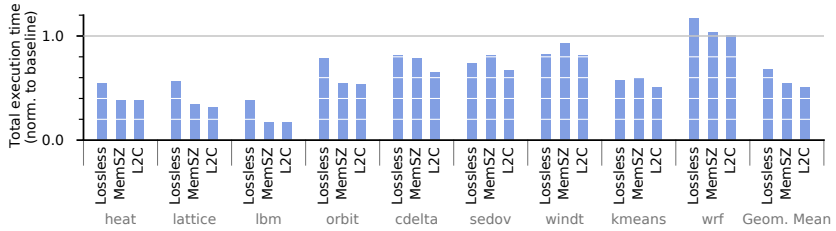
Another benefit of the three compressing designs is that they are able to maintain compressed data in the LLC, increasing its apparent capacity. Figure 11e shows the LLC Misses per Kilo-Instruction (MPKI) normalized to the baseline system. L^{2}C reduces average MPKI by 69%. This is a 16% improvement over MemSZ and 49% over Lossless.

Execution time is affected both by the reduced memory latency and the improved LLC miss rate. Figure 11a shows the execution time achieved by each system, normalized to that of the baseline system. We observe that L^{2}C equals or surpasses both competing designs in all tested applications. L^{2}C reduces execution time by an average 50%, improving on MemSZ by 9% and Lossless by 26%.

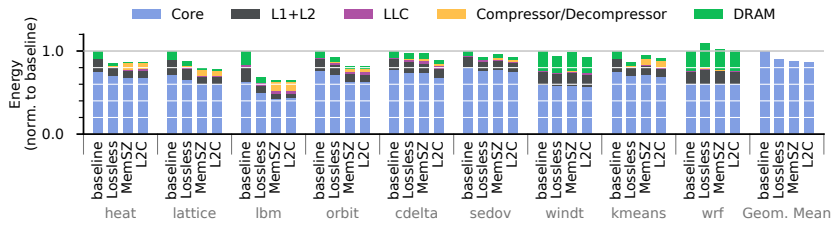
The reduced execution time coupled with reduced DRAM activity translate into a reduction of total system energy. Figure 11b shows the total energy consumption of each design, broken down by system component. The energy consumption follows the same trend as memory traffic, with L^{2}C achieving an average reduction of 16%. This is 3% and 5% better than MemSZ and Lossless, respectively. Notably, Lossless is closer in energy consumption than the other metrics, owing to the less complex compressor/decompressor.

Finally, each application's output error is presented in Table 3c. We find that for the majority of the benchmarks, approximation introduces less than 0.05% relative error compared to the baseline output. L^{2}C differs from MemSZ by at most 0.01%. This is due to cache interference effects causing slight differences in eviction timing, leading to small variations in lossy compression outcome.

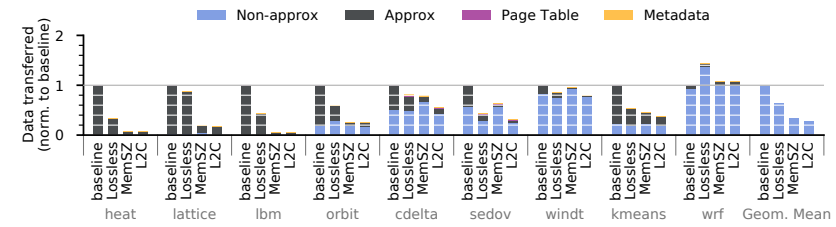
Across the tested applications, we see clear indications that the improvements gained by lossy and lossless compression have significant overlap. A hybrid approach is able to achieve the benefits of both methods, where each is most suitable. L^{2}C surpasses MemSZ by also compressing the non-approximable traffic, and outperforms Lossless by applying more aggressive compression to the subset of data which tolerate it.



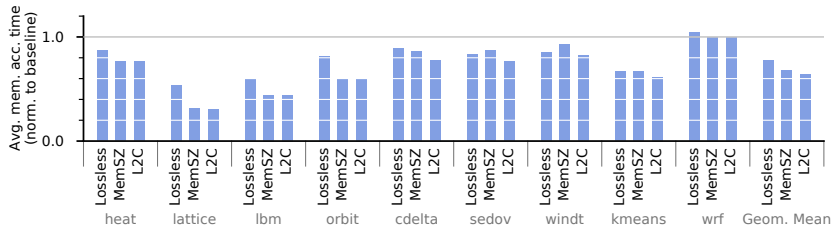
(a) Execution time.



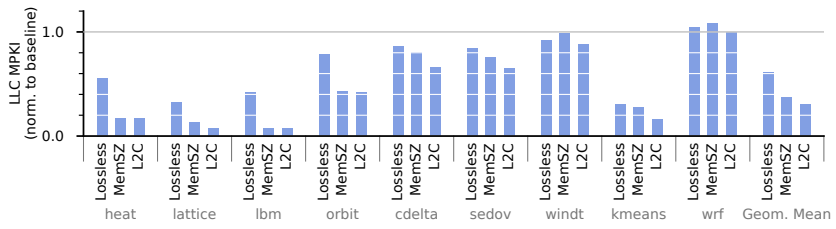
(b) System Energy Consumption.



(c) Memory Traffic.



(d) Average Memory Access Time (AMAT).



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

Fig. 11. Evaluation of the L^{2}C memory compression design and comparison with competing designs.

Table 4. I/O compression efficacy.

(a) Achieved compression ratio.

	height	aquarius	gb6	strang	hand	mitbih	ampds	air	gas	hydra	GM
Lossless	2.34×	1.54×	1.03×	2.67×	2.25×	1.61×	1.51×	1.44×	1.05×	1.52×	1.62×
MemSZ	3.59×	6.38×	2.48×	1.95×	2.17×	2.03×	7.44×	1.03×	10.35×	8.20×	3.55×
L^{2}C	3.93×	6.39×	2.48×	2.87×	2.36×	2.31×	7.55×	1.45×	10.35×	8.58×	3.96×

(b) Relative approximation error.

	height	aquarius	gb6	strang	hand	mitbih	ampds	air	gas	hydra
Lossless	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
MemSZ	0.33%	0.44%	0.33%	0.07%	0.35%	0.32%	0.25%	<0.01%	0.40%	0.36%
L^{2}C	0.33%	0.44%	0.33%	0.04%	0.09%	0.32%	0.25%	<0.01%	0.40%	0.36%

We observe that the traffic reduction achieved by L^{2}C equals or surpasses MemSZ and Lossless in all the tested benchmarks. Of note is that two of the tested benchmarks benefit more from the modest Lossless compression across all data than from more aggressive MemSZ compression on only the approximable subset. This illustrates that the memory footprint of each subset is of lesser importance than the memory activity induced by each. Compression is most beneficial on blocks which normally bounce between main memory and LLC, and this is highly application-dependent.

Wrf and *orbit* illustrate a data pattern which defeats the heuristic used by L^{2}C to determine compressibility of S-blocks. A subset of non-approximable data has interspersed cache lines showing at least 2:1 compressibility, but four-line blocks alternate between being compressible and incompressible. Each time a compressible line is written back to an uncompressed S-block, the block's *back-off* counter is incremented, bringing the block closer to a retry. The result is a large number of *failed* block writebacks which ultimately lead to new *retry* fetches.

Heat, *lattice* and *lbm* make up another interesting subset of applications, those with only or almost only approximable memory traffic. For such applications, the only room for L^{2}C to improve upon MemSZ is in approximable blocks which have failed lossy compression. As shown in Table 3b, only *lattice* has any significant opportunity like this, and L^{2}C successfully exploits it. *Kmeans* and *wrf* also show MemSZ leaving blocks uncompressed, which are successfully compressed by L^{2}C.

Sedov and *windt* both benefit more from lossless compression than lossy, in terms of memory traffic. This is a by-product of approximation tolerance. While these applications both process a large data footprint of regular data, not all of it is safe to approximate. As a result, a large portion of their memory traffic is compressible but only using lossless compression. In these applications, Lossless performs better than MemSZ, while L^{2}C capitalizes on the strengths of both.

5.2.2 I/O Compression. As explained in Section 4.7, the primary metric of interest for I/O compression is the achieved compression ratio. Table 4a shows the results for the three evaluated compression schemes, *Lossless*, *L^{2}C* , and *MemSZ*. Due to its hybrid nature, L^{2}C equals or surpasses MemSZ in all cases. This is because any block which MemSZ can compress successfully will be compressed identically in L^{2}C . The remaining blocks are guaranteed equal or better compression, since MemSZ leaves them uncompressed while L^{2}C applies additional compression. The same holds true against *Lossless*. Notably, *strang* and *hand* exhibit better compressibility with lossless than lossy in some blocks. Since L^{2}C chooses the most effective compressor, it yields a higher total compression ratio than MemSZ. On average, L^{2}C achieves a compression ratio of 3.96:1. MemSZ manages 3.55:1 and Lossless reaches 1.62:1.

A similar trend is observed in the introduced approximation error. Table 4b shows the mean relative error caused by compressing each dataset. In spite of its higher compression ratio, L^{2}C

introduces no extra error compared to MemSZ. This is because all lossily compressed blocks are compressed identically between the two, introducing the exact same error. In *strang* and *hand*, a by-product of selecting lossless compression when beneficial is that error is also reduced. No tested dataset suffers more than 0.4% error.

6 CONCLUSIONS

L^2C is a hybrid lossy/lossless memory and I/O compression scheme, the first of its kind. It applies general-purpose lossless compression alongisde state-of-the-art lossy compression to improve the bandwidth efficiency of both the system memory bus and processor I/O traffic. In memory compression experiments, L^2C achieves average memory-footprint compression of 3.2:1 across all benchmarks (up to 7.2:1 on a single one), improving by 33% over a pure-lossless solution. On approximable data, L^2C achieves an average compression ratio of 10.4:1 (up to 16:1), which is an 8% improvement over the current state-of-the-art lossy memory compression. Furthermore, compared to the best previous work, L^2C reduces off-chip memory traffic at least by 18%, execution time by 9% and total system energy by 3%. When applied to a set of real-life datasets for I/O compression, L^2C achieves an average of 4:1 compression, surpassing lossy and lossless single-method compressors by 10% and 241%, respectively.

ACKNOWLEDGEMENTS

This work is supported by the Swedish Research Council (contract number 2014-6221) under the ACE project.

REFERENCES

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, no. 2012, pp. 1–16, 2012.
[2] J. Ahn *et al.*, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA*. ACM/IEEE, 2015, pp. 336–348.
[3] ——, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*. ACM/IEEE, 2015, pp. 105–117.
[4] M. Pavlovic *et al.*, "On the memory system requirements of future scientific applications: Four case-studies," in *IISWC*, 2011, pp. 159–170.
[5] B. Rogers *et al.*, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," in *ISCA*, 2009, pp. 371–382.
[6] NVIDIA, "Nvidia tegra x1: Nvidia's new mobile superchip," whitepaper, 2015.
[7] M. Doggett, "Texture caches," in *MICRO*, vol. 32, no. 3. IEEE, 2012, pp. 136–141.
[8] J. Kim *et al.*, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *ISCA*. ACM/IEEE, 2016, pp. 329–340.
[9] A. Eldstål-Damlin *et al.*, "AVR: Reducing memory traffic with approximate value reconstruction," in *ICPP*, 2019, pp. 1–10.
[10] A. Eldstål-Ahrens and I. Sourdis, "MemSZ: Squeezing memory traffic with lossy compression," *ACM TACO*, vol. 17, no. 4, pp. 40:1–40:25, Nov. 2020.
[11] A. Alameldeen and D. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2004.
[12] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 8, pp. 1196–1208, 2010.
[13] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012, pp. 377–388.
[14] A. Arelakis and P. Stenstrom, "SC2: a statistical compression cache scheme," in *ACM SIGARCH Computer Arch. News*, vol. 42. IEEE Press, 2014, pp. 145–156.
[15] A. Arelakis *et al.*, "HyComp: a hybrid cache compression method for selection of data-type-specific compression methods," in *MICRO*. IEEE, 2015, pp. 38–49.
[16] V. K. Chippa *et al.*, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC*, 2013, pp. 1–9.

[17] J. R. Goldschneider, "Lossy compression of scientific data via wavelets and vector quantization," Ph.D. dissertation, Univ. of Washington, 1997.

[18] C. J. Deepu *et al.*, "A hybrid data compression scheme for power reduction in wireless sensors for IoT," *IEEE Trans. on Biomedical Circuits and Systems*, vol. 11, no. 2, pp. 245–254, 2017.

[19] B. Abali *et al.*, "Data compression accelerator on IBM POWER9 and z15 processors : Industrial product," in *ISCA*. IEEE, 2020, pp. 1–14.

[20] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *IPDPS*. IEEE, 2016, pp. 730–739.

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

[22] J. Dusser and A. Seznec, "Decoupled zero-compressed memory," in *Int. Conf. on HiPEAC*. ACM, 2011, pp. 77–86.

[23] S. Kim *et al.*, "Transparent dual memory compression architecture," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 206–218.

[24] A. Malek *et al.*, "Odd-ecc: on-demand dram error correcting codes," in *MEMSYS*, 2017, pp. 96–111.

[25] G. Pekhimenko *et al.*, "Linearly compressed pages: a low-complexity, low-latency main memory compression framework," in *MICRO*. IEEE, 2016, pp. 172–184.

[26] A. Shafiee *et al.*, "Memzip: Exploring unconventional benefits from memory compression," in *HPCA*, 2014, pp. 638–649.

[27] R. Kanakagiri *et al.*, "MBZip: Multiblock data compression," *TACO*, vol. 14, no. 4, pp. 1–29, 2017.

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

[29] E. Choukse *et al.*, "Compresso: Pragmatic main memory compression," in *MICRO*. IEEE, 2018, pp. 546–558.

[30] M. Vecchio *et al.*, "Adaptive lossless entropy compressors for tiny iot devices," *IEEE Transactions on Wireless Communications*, vol. 13, no. 2, pp. 1088–1100, 2014.

[31] G. Campobello *et al.*, "An efficient lossless compression algorithm for electrocardiogram signals," in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 777–781.

[32] R. Vestergaard *et al.*, "Generalized deduplication: Lossless compression for large amounts of small iot data," in *European Wireless 2019; 25th European Wireless Conference*, 2019, pp. 1–5.

[33] D. Blalock *et al.*, "Sprintz: Time series compression for the internet of things," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3264903

[34] B. R. Stojkoska and Z. Nikolovski, "Data compression for energy efficient iot solutions," in *2017 25th Telecommunication Forum (TELFOR)*, 2017, pp. 1–4.

[35] K. B. Adedeji, "Performance evaluation of data compression algorithms for iot-based smart water network management applications," *Journal of Applied Science & Process Engineering*, vol. 7, no. 2, pp. 554–563, 2020.

[36] A. Moon *et al.*, "Lossy compression on IoT big data by exploiting spatiotemporal correlation," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[37] A. Khelifati *et al.*, "Corad: Correlation-aware compression of massive time series using sparse dictionary coding," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 2289–2298.

[38] A. Ukil, S. Bandyopadhyay, A. Sinha, and A. Pal, "Adaptive sensor data compression in IoT systems: Sensor data analytics based approach," in *IEEE Int. Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5515–5519.

[39] A. Ukil, S. Bandyopadhyay, and A. Pal, "IoT data compression: Sensor-agnostic approach," in *2015 Data Compression Conference*, 2015, pp. 303–312.

[40] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*. IEEE, 2012, pp. 449–460.

[41] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 922–927, 2005.

[42] M. de Kruijf *et al.*, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*. ACM/IEEE, 2010, pp. 497–508.

[43] D. Jevdjic, K. Strauss, L. Ceze, and H. S. Malvar, "Approximate storage of compressed and encrypted videos," in *ASPLOS*, 2017, pp. 361–373.

[44] A. Sampson *et al.*, "Approximate storage in solid-state memories," *ACM TOCS*, vol. 32, no. 3, p. 9, 2014.

[45] J. San Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," in *MICRO*. IEEE, 2014, pp. 127–139.

[46] B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," in *PACT*, 2014, pp. 493–494.

[47] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *TACO*, vol. 12, no. 4, p. 62, 2016.

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

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

[50] A. Angerd *et al.*, "A framework for automated and controlled floating-point accuracy reduction in graphics applications on gpus," *TACO*, vol. 14, no. 4, pp. 1–25, 2017.
[51] ——, "A GPU register file using static data compression," in *ICPP*. ACM, 2020.
[52] P. Judd *et al.*, "Proteus: Exploiting numerical precision variability in deep neural networks," in *ICS*. ACM, 2016, pp. 1–12.
[53] A. Ranjan *et al.*, "Approximate memory compression," *IEEE TVLSI*, vol. 28, no. 4, pp. 980–991, 2020.
[54] S. Lal, "SLC: Memory access granularity aware selective lossy compression for gpus," in *DATE*. IEEE, 2019, pp. 1184–1189.
[55] J. San Miguel *et al.*, "Doppelganger: A cache for approximate computing," in *MICRO*. IEEE, 2015, pp. 50–61.
[56] A. Sampson *et al.*, "Enerj: Approximate data types for safe and general low-power computation," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 164–174, 2011.
[57] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *SIGARCH C.A. News*, vol. 33, 2005, pp. 74–85.
[58] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio," in *ISCA*. ACM/IEEE, Apr 1994, pp. 384–393.
[59] C. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, 2005, pp. 190–200.
[60] D. Genbrugge *et al.*, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA*, Jan 2010, pp. 1–12.
[61] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE CAL*, vol. 10, no. 1, pp. 16–19, 2011.
[62] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*. IEEE, 2009, pp. 469–480.
[63] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *HP lab.*, vol. 27, pp. 22–31, 2009.
[64] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc., 2004.
[65] S. Ansumali *et al.*, "Minimal entropic kinetic models for hydrodynamics," *Europhysics Letters*, vol. 63, no. 6, p. 798, 2003.
[66] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH C.A. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[67] University of Chicago ASCF Center. (2018) Flash4 user's guide. [Online]. Available: http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug_4p6.pdf
[68] A. Eldstål-Ahrens. (2018) 1D K-Means, Open Source. [Online]. Available: https://github.com/eldstal/kmeans
[69] Lantmäteriet, "Swedish topological survey hdb 50+ västra götaland, zone 63_3," 2016. [Online]. Available: https://www.lantmateriet.se/
[70] NASA/JPL. (2019) JPL SMAP level 3 CAP sea surface salinity standard mapped image 8-day running mean v4.3 validated dataset. [Online]. Available: https://doi.org/10.5067/smp43-3tpcs
[71] T. McGlynn *et al.*, "Skyview: The multi-wavelength sky on the internet," in *Symposium-International Astronomical Union*, vol. 179. Cambridge University Press, 1998, pp. 465–466.
[72] Swedish Meteorological and Hydrological Institute. (2020) STRÅNG - a mesoscale model for solar radiation. [Online]. Available: http://strang.smhi.se/
[73] S. Lobov *et al.*, "Latent factors limiting the performance of sEMG-interfaces," *Sensors*, vol. 18, no. 4, p. 1122, 2018.
[74] A. L. Goldberger *et al.*, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000 (June 13).
[75] S. Makonin *et al.*, "Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014," *Scientific Data*, vol. 3, no. 160037, pp. 1–12, 2016.
[76] S. De Vito *et al.*, "On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario," *Sensors and Actuators B: Chemical*, vol. 129, no. 2, pp. 750–757, 2008.
[77] J. Fonollosa *et al.*, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," *Sensors and Actuators B: Chemical*, vol. 215, pp. 618–629, 2015.
[78] N. Helwig *et al.*, "Condition monitoring of a complex hydraulic system using multivariate statistics," in *IEEE I2MTC*, 2015, pp. 210–215.
[79] A. Yazdanbakhsh *et al.*, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

Received February 2021; revised May 2021, August 2021; accepted August 2021