AVR: Reducing Memory Traffic with Approximate Value Reconstruction

AVR: Reducing Memory Traffic with Approximate Value Reconstruction

Albin Eldstål-Damlin, Pedro Trancoso and Ioannis Sourdis
Chalmers University of Technology, Gothenburg, Sweden
{eldstal,ppedro,sourdis}@chalmers.se

ABSTRACT

This paper describes Approximate Value Reconstruction (AVR), an architecture for approximate memory compression. AVR reduces the memory traffic of applications that tolerate approximations in their dataset. Thereby, it utilizes more efficiently the available off-chip bandwidth improving significantly system performance and energy efficiency. AVR compresses memory blocks using low latency downsampling that exploits similarities between neighboring values and achieves aggressive compression ratios, up to 16:1 in our implementation. The proposed AVR architecture supports our compression scheme maximizing its effect and minimizing its overheads by (i) co-locating in the Last Level Cache (LLC) compressed and uncompressed data, (ii) efficiently handling LLC evictions, (iii) keeping track of badly compressed memory blocks, and (iv) avoiding LLC pollution with unwanted decompressed data. For applications that tolerate aggressive approximation in large fractions of their data, AVR reduces memory traffic by up to 70%, execution time by up to 55%, and energy costs by up to 20% introducing up to 1.2% error to the application output.

CCS CONCEPTS

• **Computer systems organization** → *Other architectures*; • **Hardware** → *Memory and dense storage.*

KEYWORDS

Approximate Computing, Memory Compression.

ACM Reference Format:
Albin Eldstål-Damlin, Pedro Trancoso and Ioannis Sourdis. 2019. AVR: Reducing Memory Traffic with Approximate Value Reconstruction. In *48th Int'l Conf. on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337824

1 INTRODUCTION

The performance of computer systems is largely dominated by their memory hierarchy as the gap between computing speed and data transfer speed keeps increasing [46]. Besides the long memory latency, memory bandwidth severely limits performance, energy efficiency and scalability of Chip Multiprocessors (CMPs) [33]. On one hand, the demand for higher memory bandwidth increases. Adding more cores on a chip and using specialized accelerators increases the potential processing throughput and calls for higher data rates. New emerging data-intensive applications further increase the need for large volumes of data to be transferred fast [4, 5, 30]. On the other hand, memory bandwidth is pin limited [4, 49] and power constrained [28] and is therefore more difficult to scale [33]. More expensive, 3D-stacked DRAM technologies alleviate the bandwidth problem, but due to power constraints cannot keep up with the increasing demand on data rates either [28].

One way to alleviate the memory bandwidth pressure is to reduce the volume of transferred data using compression. Data can then be transferred between the main memory and the processor chip in a compressed form consuming less bandwidth and reducing energy cost. With a few exceptions, hardware main memory compression is limited to lossless methods. Commercial examples of architectures that use memory compression are graphics processing units (GPUs) [1]. GPUs use application-specific compression, applied to texture and color data [14], and often solve the easy part of the problem, handling read-only data [42]. Current state-of-the-art, lossless memory compression techniques achieve on average a 2:1 to 4:1 compression ratio [23]. However, some classes of applications, i.e., commercial, multimedia, scientific, may allow for more aggressive compression as they inherently tolerate approximations in parts of their data [11, 18] without introducing significant error.

In the past, the performance of memory subsystems has been improved for approximation-tolerant applications. Load value prediction without fetching the actual requested data has been used for improving memory latency and bandwidth [37, 45, 47], but has difficulties capturing irregular data variations. Approximate deduplication of individual cachelines increases cache capacity [39], however, multiple values need to match at cacheline granularity. A form of lossy compression has been applied in approximate computing, but is constrained to reducing precision of single values truncating their least significant bits [6, 21, 22, 42] and therefore achieves limited compression ratio.

In this work, Approximate Value Reconstruction (AVR) is proposed for reducing data volumes transferred between processor chip and main memory in approximation tolerant applications, utilizing more efficiently the memory bandwidth. AVR goes beyond reducing the precision of individual values and compresses data in a lossy manner exploiting similarities between values while capturing their variance. In essence, AVR stores a "summary" of approximated data in memory, based on which values are approximately reconstructed in the processor chip. AVR addresses a number of challenges. Summarizing (compressing) and reconstructing (decompressing) the data needs to be generic, introduce low error, and add minimum latency and energy overheads. Moreover, managing (updating, repacking, storing) compressed data needs to be efficient and impose low traffic overheads. In effect, AVR utilizes memory bandwidth better improving performance and energy efficiency.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

Concisely, the AVR architecture contributes the following. AVR supports aggressive, approximate memory compression exploiting similarities across values and reduces memory traffic improving execution time and energy efficiency. In addition, AVR improves the effectiveness and minimizes the overheads of aggressive compression by co-locating compressed memory blocks and uncompressed cachelines in the Last Level Cache (LLC); handling LLC eviction in a lazy manner; keeping track of badly compressing memory blocks; and selecting which data to store in LLC after decompression.

The remainder of this paper is organized as follows. Section 2 discusses related work on lossless memory and cache compression as well as on approximate computing with focus on memory systems. Section 3 describes the proposed AVR architecture. Section 4 presents our evaluation results and Section 5 draws our conclusions.

2 RELATED WORK

Prior work on related topics is discussed next. First, existing designs for lossless memory and cache compression are presented and subsequently an overview is provided on approximate computing techniques that improve the performance of memory systems.

Lossless Memory Compression: There is a plethora of memory compression techniques that improve memory capacity and bandwidth utilization. Various compression algorithms are used, such as dictionary-based [9], exploiting frequent patterns or zero-value blocks [15], and more recently similarities of words at the same bit position [23]. However, lossless solutions have limited compression ratio between 2:1 and 4:1, which is substantially lower than in our work (4-8×). In general, lossless compression is orthogonal to AVR as it can be used in our design to compress data that are not approximated, or even on top of AVR approximately compressed data. Another aspect is the data placement in memory. Some approaches compact compressed data in memory to improve capacity [31]. Others, like in our work, avoid data compaction, allocating the worst case storage required for the uncompressed data and focus only on memory bandwidth [44]. Finally, managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [12, 20]. AVR uses a metadata table and a cache of it, as in [31], which is updated with the TLB and adds a few bytes of bandwidth overhead at every TLB miss; still techniques like Attache [20] could be used to further reduce the metadata cost.

Lossless Cache Compression: Lossless compression has been applied to caches, too. Besides the issues of encoding and compaction of variable size blocks [41], the compression and decompression latency constraints are tighter compared to memory compression. In the past, cache compression has been supported in various ways, for instance using value-centric caches [8]. Compacting compressed cache blocks has been tackled using decoupled super-blocks and sub-blocks [40], or super-blocks without decoupling tag and data arrays [29]. In general, cache compression cannot reduce memory traffic as it compresses single cachelines separately, rather than larger memory blocks of consecutive cachelines as performed by AVR. Consequently, as opposed to AVR, cache compression techniques applied to the LLC cannot reduce the number of memory accesses and hence cannot reduce memory traffic. Furthermore, AVR uses the LLC to store compressed memory blocks alongside the uncompressed lines, but does not attempt to compress individual cachelines. As a consequence, cache compression could be considered to compress the AVR LLC contents.

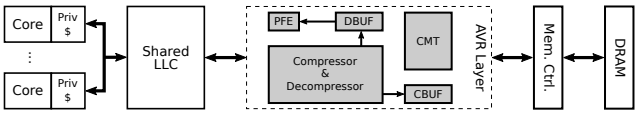


Figure 1: Toplevel block diagram of the AVR architecture.

Approximate Computing: Large classes of applications are inherently tolerant to approximations [11]. This enables a tradeoff between the quality of their results and their performance and energy efficiency. This tradeoff is exploited by various approximate computing techniques, some of them targeting the aforementioned memory bottlenecks in a lossy manner.

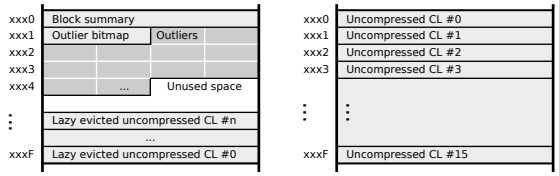
Approximate load value prediction techniques reduce memory latency by providing a predicted value substantially faster than fetching the actual one from memory [37, 45, 47]. They may further improve memory bandwidth utilization by not always bringing the actual values at all. Value prediction techniques speculate that the values loaded by the same instruction may be identical or differ by a stride. However, this does not capture any irregular variance of data such as the variance in an image where neighboring pixels may have similar values but may not necessarily differ by a fixed stride. Approximate load value prediction is applied near the core (in parallel to the L1 cache) and is therefore orthogonal to the proposed AVR compression of memory traffic. Another fundamental difference compared to AVR and in general compared to compression is that load value prediction techniques aim primarily at reducing load latency rather than memory bandwidth because in the end they do fetch the precise values from memory for error checking.

Reducing the precision of floating point [6, 21, 42] and fixed point [22] numbers has been used to alleviate the memory bandwidth bottleneck in deep neural networks [22], GPU workloads [42] and other approximation tolerant applications [21], thereby improving performance and energy efficiency. However, the compression ratio is still limited between 2:1 and 4:1 despite the loss of precision as these approaches do not exploit inter-value similarities to compress data. Closer to AVR, software techniques for lossy compression have been proposed, but have high complexity and latency and as a consequence cannot be used directly in hardware [13].

Approximate, lossy compression has been applied to caches, too. Doppelgänger deduplicates similar cachelines to compress data [39]. The subsequent Bunker cache design speculates similarities between cachelines solely based on their addresses without looking at their contents, proposing a less intrusive cache design but achieving lower compression ratio than Doppelgänger [38]. Both designs exploit similarities between cachelines. However, similar values need to have the same offset within their cachelines in order to match, which restricts deduplication opportunities.

3 AVR ARCHITECTURE

Approximate Value Reconstruction (AVR) reduces the volume of data transferred between main memory and processor chip improving bandwidth utilization and in turn system performance and energy efficiency. Without loss of generality, AVR is applied to a Chip Multiprocessor as depicted in Figure 1.



(a) Compressed block **(b) Uncompressed block**

Figure 2: AVR Memory Block.

In a nutshell, AVR handles a processor request to approximated data as follows. In case the requested cacheline misses in the LLC, the corresponding memory block, which compresses multiple cachelines including the requested one, is brought on-chip. The requested cacheline is then retrieved, stored in the LLC, and sent to the processor. The memory block is also stored in the LLC as is (compressed) so to avoid memory accesses at future requests to the block.

In general, AVR employs a number of optimizations to reduce its overheads. As mentioned above, compressed memory blocks are stored in the LLC trading LLC capacity for fewer memory accesses. In addition, every time there is an LLC eviction it would be wasteful to update the corresponding compressed block in memory. Instead, for as long as there is available space in the memory block, AVR evicts such cachelines lazily, writing them back to memory uncompressed. Then, when the space is exhausted, the block is compacted embedding all lazily evicted cachelines. Finally, the overheads of unsuccessful compression attempts are minimized by keeping a history of previous compression attempts per block.

The AVR architecture requires the following additions: a compressor and decompressor module to *summarize* data before sending them to memory and to *reconstruct* data coming back from the memory; a metadata table for storing information about the compressibility of the memory blocks; finally the LLC design requires changes for storing compressed memory blocks in addition to normal cachelines. Next, each one of the above modules is discussed separately, after first presenting the format of the AVR memory blocks. At the end of the section, the AVR LLC and memory operations are discussed.

3.1 Memory Blocks

Similar to most techniques that focus on data approximations [21, 36, 39], AVR considers that the programmer annotates memory regions that can be approximated and hence compressed in a lossy manner. This annotation also includes the size of the region as well as the datatype of the approximable data. An additional OS system call allows allocated pages to be marked as approximate at the page table requiring an extra bit for every page table and translation lookaside buffer (TLB) entry as shown in Figure 3. The programmer may further indicate an upper error threshold for acceptable approximations. In our experiments, two thresholds are used, one for the relative error of each individual value and one for the average error of all values in a block. Currently, error thresholds are common for all approximations in a program, but they could be easily extended to thresholds per allocated memory region adding a respective field to the page table.

The AVR architecture does not consider improving memory capacity and therefore memory allocation is not affected. Compression is performed at the granularity of memory blocks composed of

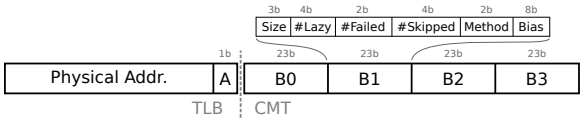


Figure 3: Format of a metadata table entry and TLB addition.

multiple cachelines as shown in Figure 2; a cacheline, i.e., 64B, being the granularity of accessing the main memory. In our implementation, a block is composed of 16 cachelines, in total a quarter of a physical 4KB page. AVR compresses the 16 cachelines of a block to a single cacheline *summary* aiming at a 16:1 compression ratio and at accessing the entire block with one memory request. The summary is stored in the first cacheline of the memory block as shown in Figure 2a. In case this compression produces approximations of some values that exceed a particular error threshold, these values are characterized as *outliers* and stored explicitly, uncompressed in the compressed block. The outliers are placed in order after the *summary* cacheline, together with a bitmap that indicates their location in the uncompressed block (one bit per 32-bit value). This bitmap occupies half a cacheline if the block contains outliers. Summary, bitmap and outliers occupy in total 1-8 out of the 16 cachelines (2:1 worst case compression ratio). The remaining space of the memory block remains available for *lazy evictions*; that is for writing back dirty uncompressed cachelines of the block, when evicted from the LLC. Thereby, AVR avoids bringing a compressed block on-chip to be updated every time a dirty cacheline is evicted. This is possible until the block space is exhausted, then the block and the lazily evicted dirty uncompressed cachelines are fetched from memory for recompaction. In case a memory block fails to be compressed in 8 cachelines or is explicitly marked as not-approximable then it is stored uncompressed as shown in Figure 2b.

3.2 Metadata Table

Each compressible block requires some metadata information in order to be handled. Similar to previous approaches on memory compression [16, 31], these metadata are stored in main memory and cached on-chip in a TLB-like Compression Metadata Table (CMT) placed and accessed in parallel with the LLC. The CMT is updated in pair with the TLB. A CMT has four 23-bit entries per 4KB page, one per 1KB memory block as shown in Figure 3. CMT stores the following information about each memory block: its size (compressed in 1-7 lines or uncompressed), number of lazy evicted cachelines stored, compression method (and datatype of values), and a bias of its values. Finally, it maintains two counters to keep the history of previous compression attempts. The first one counts the number of consecutive failed compression attempts. Then, depending on that count, a number of recompression attempts (in block updates) are skipped to reduce the overhead of badly compressed blocks.

3.3 Summarizing & Reconstruction

Summarizing and approximately reconstructing memory blocks requires knowledge of the particular value representation used in the considered dataset. Our current implementation supports standard 32-bit floating-point and fixed point formats, but can be easily extended to support other representations, too. The core

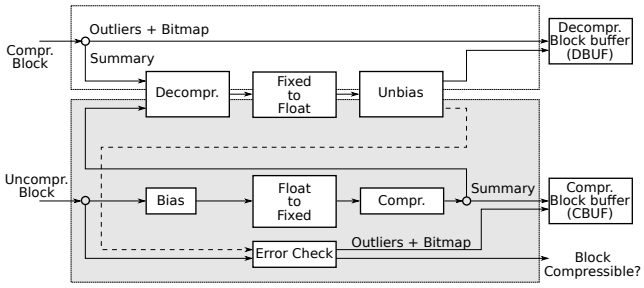


Figure 4: AVR compressor/decompressor module.

part of the compression is using fixed point arithmetic to reduce complexity. Consequently, memory blocks containing floating point numbers are converted to fixed point before compression and back to floating point after decompression. Figure 4 shows the block diagram of the AVR compressor and decompressor.

Incoming uncompressed blocks are fed to the compressor cacheline by cacheline in a pipelined fashion. Fixed point values are compressed directly. Floating point values are converted to fixed point after first having their exponent field biased to minimize loss of accuracy. Subsequently, a simple downsampling compressor is employed to generate the summary of the block replacing multiple (typically 16) uncompressed values with their average. In order to check the error of the approximated values and identify outliers, the compressed block summary is decompressed again, and if necessary converted back to floating point and unbiased. Then, each approximated value can be compared with its respective original uncompressed value stored in the input of the compressor. The result of this comparison identifies the outliers and produces the bitmap of their locations, which is part of the compressed block when outliers exist as shown in Figure 2a. This bitmap is also used to select and compact the outliers stored in the block. Thereby, the summary, bitmap and outliers of a block are produced and stored in the compressed block buffer (CBUF). Once compression completes, the metadata of the block are updated in the CMT.

Decompression is simpler. The summary of a compressed block is sent to the decompressor that produces its decompressed version and stores it to the decompressed block buffer (DBUF) after converting it to floating point and unbiasing, when needed. In addition, the outliers are placed according to their bitmap on the buffer replacing the respective decompressed values. The requested decompressed cachelines are then sent to the LLC. The remaining ones are kept in the buffer and future requests for cachelines of the same block are served from there. When the next block arrives for decompression, a prefetcher (PFE) selects a number of decompressed cachelines, not yet stored in the LLC, to be inserted in the LLC before being replaced by the new block under decompression.

Biasing & unbiasing: When dealing with extremely large or small floating-point numbers (large positive or negative exponent), the conversion to fixed-point format can cause a greater loss of precision. To avoid this, blocks are *biased* during compression. A *bias* value is determined, which, when added to the exponent of the values in the block, can bring the block's values into a representable range. Biasing is not performed on blocks where either a) the selected bias would cause special values such as *NaN* or *Inf*, or b) the selected bias would cause over- or underflow of the exponent of any value. The bias is stored with the block's metadata and used during decompression to restore the original range of values. Biasing involves finding the maximum and minimum exponent of the values in a block, determining a suitable offset, and applying it to the exponents of the block. Biasing is pipelined and performed in 4 cycles. The inverse process (unbiasing) requires an 8-bit addition to all decompressed values and requires one cycle.

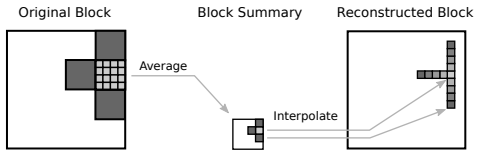


Figure 5: Downsampling and Reconstruction of a 2D block.

Float to fixed & fixed to float conversions: Converting from float to fixed point numbers and vice-versa is implemented as described in [35] requiring a single cycle.

Compression: Although various lossy compression algorithms can be considered, we opted for a method that is simple to implement. In AVR, memory blocks are compressed using downsampling [26]. This method entails dividing the block into a suitable number of sub-blocks and computing the average value of each sub-block. We aim for a 16:1 compression ratio and therefore sub-blocks of 16 values are used. In our attempt to find the best compression, a number of variations of the method are used in parallel. The main two variants differ in the considered placement of the values in the block before partitioning to sub-blocks; in particular, the first one considers the block as a square 2D array and the second as a linear 1D linear array. Figure 5 shows an example of 2D downsampling, where the compression is performed by averaging the values of a sub-block (light-grey) into a single value. For decompression, the average values are distributed evenly and bi-linear interpolation is applied to reconstruct the approximate values in-between. In our implementation, compression and decompression require 15 and 10 cycles, respectively.

Error calculation & Outliers selection: Lossy compression introduces errors in the approximated values. In order to limit this error, compression is skipped in case the error exceeds a particular threshold value. This is evaluated by comparing the original incoming uncompressed block with the approximately reconstructed block produced after compression and subsequent decompression. Two separate thresholds are used to control the approximation error of the compression operation: the relative error of each individual value may not exceed a percentage threshold T_1 and the average relative error across all values in the block may not be greater than a percentage threshold T_2. These error thresholds are exposed as a tunable knob and in our experiments $T_1 = 2T_2$. Notice that this is an error mitigation strategy of low overhead and local decision.

The error per individual value is calculated in floating point format as follows[1]. For a value to be approximated with a relative error within T_1, a comparison between the original and approximated value should result in (i) the exact match of their signs and exponents and (ii) the difference of their mantissas not exceeding the Nth most significant bit (MSbit); for an error below $1/2^N$. The above comparisons are performed in a cycle and produce the bitmap of the values that are outliers. Subsequently, this bitmap is used for

[1]For fixed point numbers a subtraction and a subsequent comparison would be required.

selecting and compacting the outliers and in parallel computing the average block error for the values that are not outliers. Selecting and compacting the outliers requires 16 cycles, one cycle per uncompressed cacheline. The relative error of each individual non-outlier value is required for computing the average error of the block. The sign and exponent are identical for the original and approximate values, otherwise they would be outliers. So, the average error is calculated by subtracting the mantissa bits of each original and approximated value. The average block error is the average of these subtractions for all the non-outliers values and computing it also fits in 16 cycles.

Prefetching decompressed cachelines: After decompressing a block, the requested cacheline(s) are stored in the LLC. Storing also the remaining cachelines could lead to the pollution of the LLC with unwanted cachelines. Consequently, they remain in DBUF until they are overwritten by another block. In the meantime, if one of these cachelines is requested it is sent directly to the LLC. When a new compressed block arrives for decompression, a prefetching engine (PFE) is consulted to decide whether any of the remaining decompressed cachelines in DBUF should be written in the LLC before they are replaced by the new block. The PFE employs a simple threshold strategy, prefetching all lines from a block where at least half have been explicitly requested.

Total compression and decompression latency: Based on the above and as confirmed by our synthesis results presented in Section 4, the total latency for compressing a block is 49 (processor) cycles, and for decompressing a block is 12 cycles. Decompression is more critical for system performance as it affects memory reads. Compression is less critical because it affects the write backs.

3.4 Last Level Cache

The AVR Last Level Cache (LLC) stores uncompressed cachelines (UCL) as well as compressed memory blocks. A compressed memory block may occupy one to eight LLC lines (64B), depending on its compressibility, it is therefore split in 64B compressed memory subblocks (CMS). When a memory block enters the processor chip and gets uncompressed, only selected uncompressed cachelines are stored in the LLC. The AVR LLC is decoupled in order to support the management of the LLC contents at two granularities, namely, that of a cacheline (64B) and that of a memory block (16 cachelines). Following the design of the Decoupled Sectored Caches [43], the AVR LLC decouples its tag array from the data array. On one hand, entries of the LLC data array have a cacheline (64B) granularity. On the other hand, the tag array has a granularity of a memory block (16 cachelines). The decoupling of tag and data arrays is facilitated by a back-pointer array (BPA) which supports the indirection between every data array entry and a tag array entry to associate the data of a cacheline with its tag. In essence, each data array entry has a respective BPA entry at the same set and way, which maintains its state-bits and a pointer to its tag in the tag array. In contrast, a tag array entry can be shared among multiple data array entries.

LLC Functionality: Figure 6 illustrates the AVR LLC functionality using an example of a memory block with tag *A*. The memory block of this example, when compressed, occupies three cachelines, CMS0, CMS1, and CMS2; one for the summary of the block and two for the bitmap and the outliers. All three cachelines of the

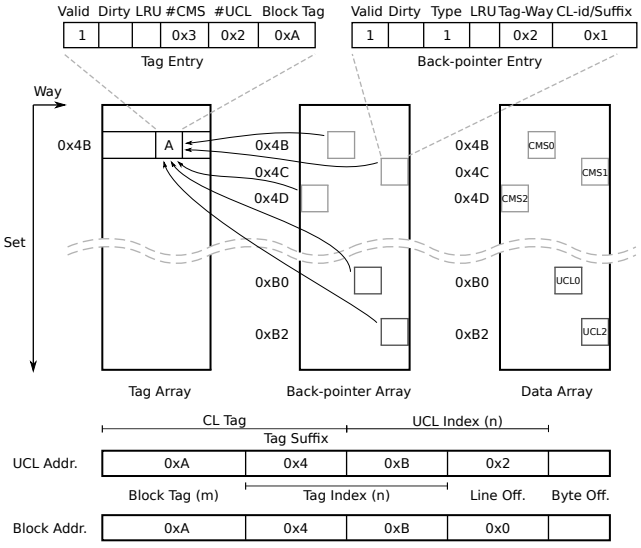


Figure 6: AVR Last Level Cache.

compressed block are stored in the LLC. In addition, two of its 16 uncompressed cachelines, UCL0 and UCL2 are also present in the LLC. The breakdown of a memory address is shown in Figure 6. After the 6-bits of byte offset, there is the 4-bit cacheline offset in the memory block. Let us consider that the LLC requires n-bits for indexing. Then, the tag array will use as index the *n* bits of the address after the cacheline offset (*tag index*) and store the remaining *m* most significant bits of the address as the memory *block tag* because it follows a memory block granularity. For example, the tag *A* for the memory block 0xA4B0 is placed in set 0x4B of the tag array. The same indexing is used for the placement of the compressed memory subblock. The first CMS of the compressed block, CMS0, is placed in a way of set 0x4B, occupying the respective entry in both the data array and the BPA. The remaining parts of the compressed block, CMS1, and CMS2, are placed in the subsequent sets 0x4C and 0x4D. Uncompressed cachelines use the indexing of a conventional cache (UCL index), in particular, the *n* bits after the byte offset. For example, uncompressed cacheline UCL2, with address 0xA4B2, is placed in set 0xB2. This LLC design has two advantages. Firstly, each UCL and CMS is mapped to different LLC sets, thereby, not affecting the effective associativity of the cache. Secondly, a single tag entry is required for all of cachelines of a block, making the management of memory blocks simpler.

LLC Structure: Structurally, the AVR LLC, depicted in Figure 6, is based on the Decoupled Sectored Caches [43] and shares some common elements with Decoupled Compressed Cache [40]. A tag array entry stores the following fields:

- **Block Tag:** The memory block tag.
- **CMS count:** the number of subblocks/cachelines needed for storing the compressed memory block (3 bits).
- **UCL count:** number of uncompressed cachelines of the block stored in the LLC (4-bits).
- **Block state bits:** valid, dirty & least recently used (LRU).

The dirty bit indicates the compressed memory block is dirty. The tag LRU is updated when a UCL of the block is accessed and used for tag-entries replacement. A BPA entry stores the following:

- **CL-type:** one bit indicating a UCL or CMS.
- **CL-id:** for a UCL, this 4-bit field stores the *cacheline tag suffix* depicted in the address breakdown; for a CMS, 3 of these 4 bits store the CMS *offset in the compressed block*.
- **Tag-way:** the way of the tag array that stores the tag of the respective block.
- **CL state bits:** valid, dirty and LRU bits.

The tag suffix of an UCL is stored in the BPA because during a lookup it needs to match together with a block tag to complete a cacheline tag match. Instead, for the BPA entries that store a CMS, the compressed memory subblock number is serving the same purpose; that is when looking up the i-th subblock (cacheline) of the compressed block the *CL-id* of the matching BPA entry should be i. Finally, the CMS LRU bits are updated when any UCL of the block is accessed.

LLC Lookup & Allocation: A request for an LLC cacheline is served by accessing in parallel the DBUF and the LLC tag array. In case the requested cacheline is in the DBUF it is returned. Otherwise the tag array access will determine whether the cacheline is available in the LLC uncompressed or its compressed memory block is present. In the first case, an UCL lookup is performed. Otherwise, in the second case a CMS lookup is performed. Figure 7 illustrates the AVR LLC lookups. Below we discuss each case in more detail.

A lookup for an uncompressed cacheline is performed as follows. The tag array is accessed using the tag index and in parallel the BPA and data array using the UCL index. The block tags in the set are matched. In parallel, the cacheline tag suffixes (*CL-id*) in the BPA set are matched for the entries in the set storing UCLs. Subsequently, the tag-way stored in each of the matching BPA entries is compared with the way of the matching block tag. There is a hit when a tag suffix matches and its tag-way points to a matching tag. The tag-way stored in the BPA entry must be equal to the way of the matching tag in order to ensure that a matching tag suffix points to its true tag, otherwise the cacheline stored in the BPA entry may have a different tag than the matching one.

A lookup for a compressed block in the LLC requires one or multiple accesses to the LLC, as many as the CMSs the block is composed of (*CMS count*). The tag array, BPA, and data array are accessed with the tag index. The block tags in the set are matched. In parallel, the entries in the BPA set that store CMSs compare their *CL-id* with zero. Here this field indicates the offset of the CMS in the compressed memory block and looking up for the first subblock requires *CL-id* to be zero. Subsequently, the tag-way stored in any of the matching BPA entries is compared with the way of the matching block tag. In this first access to the LLC, besides the first subblock (CMS), the *CMS count* is also retrieved to determine the total number of LLC accesses required for accessing the compressed block. If that number is more than one, the BPA and data array are accessed repeatedly until all parts of the block are read. At each access, *CL-id* needs to match the iteration increment, and tag-way should be the same as the matching tag entry.

When there is no available cacheline entry in the set, allocation for an UCL is performed choosing a victim cacheline based on the LRU bits stored in the BPA set. All cachelines in the set, UCLs and CMSs, compete equally. In case a CMS is evicted, then all the other CMSs of the same compressed block need to be evicted, too, and

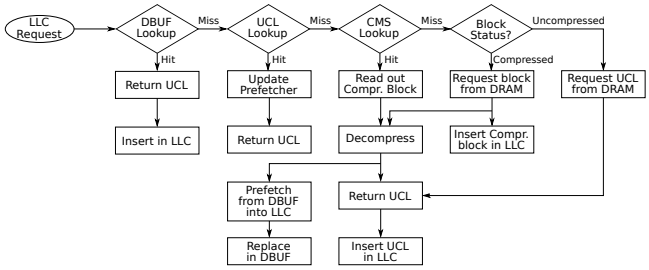


Figure 7: AVR LLC requests.

if dirty written back to memory. The tag entry of the block would remain if the LLC stores UCLs of the block. The absence of the compressed version of the block is indicated by setting to zero the field *CMS count* in its entry in tag array. Allocation for a tag entry is performed by choosing a victim tag in the set based on LRU. The LRU of a block tag is updated when one of its UCLs is accessed or when the block is recompressed. Finally, allocation for the CMSs of a block needs to be performed together at consecutive sets starting from the one indicated by the tag index.

3.5 Memory operations

We explain next the AVR memory operations at the LLC and main memory level. More precisely, we explain how a request to the LLC and an LLC eviction are handled. The details of an LLC lookup and allocation are omitted as they were described above.

LLC Requests: A request to the LLC has the following possible outcomes as shown in Figure 7:

- The requested UCL may hit either in the LLC or in the decompressed buffer (DBUF). In the latter case the UCL is also written from DBUF to the LLC.
- There is a miss of the requested UCL, but a hit to the compressed memory block stored in the LLC. Then, the compressed block is read and decompressed in the AVR compressor block to retrieve the requested cacheline.
- In case both the UCL and the compressed block miss in the LLC, the compressed block containing the requested cacheline is requested from the main memory and upon arrival decompressed to retrieve the requested cacheline. Then, the compressed block is also stored in the LLC.

Note that at a new decompression, the decompressed block previously stored in the DBUF needs to be overwritten. Before overwriting the old block, the prefetcher is consulted to potentially save some of its UCLs, storing them in the LLC.

LLC Evictions: When a cacheline is replaced from LLC, then if clean no further action is required, if dirty, the cacheline is evicted and its type is checked first as shown in Figure 8.

In case of a dirty UCL, it is checked whether its compressed memory block is also stored in the LLC. If so, the compressed block is read from the LLC, decompressed, updated with the evicted dirty UCL, compressed again and stored back to the LLC. In case the compressed block is missing from the LLC (or the compression attempt fails), the metadata table is consulted to check whether there is space in the main memory to lazily store the dirty cacheline. If so, the dirty UCL is written back to the memory and the metadata entry is updated to reflect that. Otherwise, the compressed block

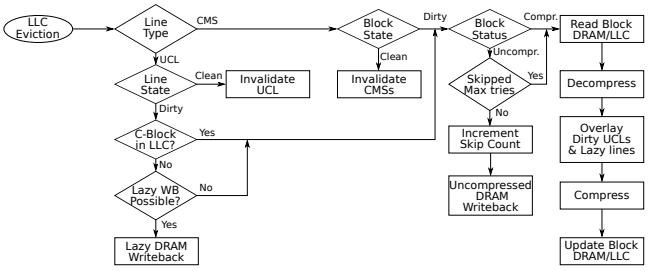


Figure 8: AVR LLC evictions.

is read from memory, decompressed, updated with all the dirty cachelines, as well as any lazy evicted lines, then compressed and written back to memory.

When bringing in a compressed block from memory, the metadata table is consulted to determine whether lazy evicted cachelines exist in memory. If so these lazy evicted cachelines are read from memory together with the block, and incorporated into the block after decompression. The block is immediately recompressed, marked dirty and stored in LLC.

When evicting a dirty CMS, the entire compressed block needs to be evicted, as partially storing it in the LLC is not useful. The dirty compressed block is first read from LLC and put in the AVR compressor/decompressor to be decompressed. Any dirty UCLs belonging to the block are read from LLC and overlaid on the decompressed block. The memory block is compressed again and written back to memory.

Note that before compressing a memory block that is currently uncompressed, because its last attempt for compression failed, its compression history and counter of skipped compressions is consulted. Based on these fields it is determined whether to proceed with the current compression or not. Accordingly, the above metadata fields are updated in the respective entry. If the recompression is skipped, the dirty UCL is written back to memory directly.

4 EVALUATION

In this section we evaluate the effectiveness of the AVR architecture. We first describe our experimental setup, presenting the system configuration of our experiments and the benchmarks used. Then, we discuss the hardware overheads of the AVR architecture. Finally, we show our evaluation results and comparison with related designs in terms of performance, energy, and application output error.

4.1 Experimental setup

We evaluated the AVR system using an in-house simulator, implemented on top of Pin[25], that employs an interval-based processor model, as proposed by Genbrugge et al. [17], and a cycle-accurate model of the memory hierarchy that uses DRAMSim2 for modelling main memory [34]. McPAT[24] and CACTI[27] were used to model power and latency of the system considering 32nm technology. The AVR compression hardware modules were implemented in RTL, synthesized using Synopsys to determine their operating frequency, latency and power consumption; this information was then fed to the simulation tool. The parameters of the simulated system are listed in Table 1. In order to correctly emulate the impact of the approximations in the overall application error, we not only emulate the memory accesses but we actually update the values of the memory contents accordingly by applying the construction and reconstruction methods to the data.

Table 1: Simulation parameters

Parameter	Configuration
CPU	8 core, out-of-order, 4-way issue/commit @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	8MB shared, 2 banks, 16-way, 15 cycle access latency
DRAM	8GB DDR4, 2 channels, 1600MHz

Besides the baseline system, AVR is further compared with (i) itself without marking any data as approximate so as to measure AVR overheads (*ZeroAVR*), (ii) a design that simply compresses approximate values to half-precision by truncating 16 bits similarly to what has been proposed in [21, 22, 42] (*Truncate*), and finally (ii) Doppelgänger [39], which is the closest and best performing related work on approximate data compression [39] (*Dganger*). As proposed, Doppelgänger is configured to have identical LLC data-array size and a 4× larger tag-array versus AVR, *i.e.* being able to index up to 4× more cachelines. Lossless compression techniques are considered orthogonal and so not included in the comparison; that is because the downsampled values and outliers of an AVR compressed block could be further compressed in a lossless way.

The benchmarks used in this evaluation are selected so each one of them (i) is able to execute until completion and generate an output, and (ii) can tolerate approximations in (parts of) its data. The above restricts us to using the benchmarks listed in Table 2; the table further presents the application domain, description of the approximated data-structures and output type as well as their memory footprint. The application code was analyzed to identify approximable data structures. In many cases, a large portion of the application's working set is dynamically allocated. For these cases, a wrapper was created to the *malloc* library call to allocate properly aligned space and register the address range as approximable. 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 [2]. We use the mean of the relative errors for each output value as our quality metric. Benchmarks for approximate computing (AxBench[48]) considers 10% relative output error, but it is solely up to the application provider do define what is acceptable. Similar to previous works, AVR provides the means to control the data approximation error as a knob to constrain application output error.

Table 2: Benchmark Applications

Application	Approx.	Output	Footprint	Description
heat[32]	Temps	Temps	8.2MB/core	2D Thermodynamics application that iterates over a grid of values and computes the propagation of heat.
lattice[7]	P and M	Vel.+Pr.	5MB/core	2D Lattice-Boltzmann method simulation of air flow over a solid object.
lbm[19]	Velocities	Velocities	325MB/core	3D Lattice-Boltzmann method simulation of fluid flow over a sphere.
orbit[10]	Phys. data	Phys. data	376MB/core	3D simulation of the two-particle orbit problem
kmeans[3]	Topol. [2]	Clusters	5.5MB/core	Clustering algorithm, applied on a geographic elevation map.
bscholes[48]	Options	Prices	6MB/core	Financial forecasting, predicts future stock option prices based on historical parameters.
wrf[19]	Geo data	Temp.	90MB/core	Weather forecasting model.

Table 3: Application output error

	heat	lattice	lbm	orbit	kmeans	bscholes	wrf
dganger	0.4%	0.2%	22.3%	>100%	<0.05%	<0.05%	24.9%
truncate	0.2%	0.5%	0.6%	<0.05%	<0.05%	1.4%	4.2%
AVR	0.7%	0.6%	0.1%	<0.05%	1.2%	0.5%	8.9%

4.2 AVR hardware overhead

AVR requires some extra hardware resources. The metadata stored in the CMT and the additional bit in the TLB add up to 93 bits per page. Compared to the unmodified TLB, which stores a virtual and a physical page address (52+36=88 bits), this is an overhead of roughly 2×. The AVR Tag array and the BPA add to the baseline set-associative LLC 18 bits per entry; that is in total 144kB and 3.2% overhead to the LLC. Moreover, the AVR compressor module occupies about 200k cells according to our synthesis report.

4.3 Experimental Results

We present next our experimental results for each benchmark comparing AVR with other related designs, namely, Doppelgänger, Truncate, and ZeroAVR (all results normalized to the baseline). The designs are evaluated in terms of execution time, system energy consumption, DRAM traffic, average memory access time (AMAT), and LLC misses per kilo-instruction (MPKI), as shown in Figures 9, 10, 11, 12 and 13 as well as in terms of application output error shown in Table 3. Table 4 shows the AVR compression ratio (for Truncate compression ratio is 2:1) as well as the overall memory footprint versus the baseline. AVR approximate LLC requests and evictions are analyzed and shown in Figures 14 and 15.

Before presenting the results of each application separately a few common observations are discussed. Analyzing the execution time and energy consumption of ZeroAVR, it is observed that AVR does not add significant overhead when it does not approximate data (Figures 9 and 10); only in *lbm*, ZeroAVR is 2% slower than baseline, adding similar energy overheads, mainly due to increased DRAM latency caused by changes in the memory access pattern. Moreover, its AVR Decoupled LLC performs similarly to the baseline LLC achieving the same MPKI as shown in Figure 13. In our experiments AVR LLC devotes 2-16% of its capacity to compressed blocks.

Heat dataset exhibits excellent compression; about 8× smaller total memory footprint and a 10:1 compression ratio.AVR reduces execution time by 43% compared to the baseline introducing only 0.7% error. That is almost double the reduction compared to Truncate that has a 0.2% error. Doppelgänger shows no speedup as the data used by heat do not have significant locality and therefore having an "effectively" larger cache does not improve performance. Improvements in execution time lead to AVR and Truncate reduction of baseline energy cost by 18% and 15%, respectively. Furthermore, Doppelgänger introduces an energy overhead of 1% due to its LLC design. AVR reduces memory traffic by 71% compared to baseline. Truncate reaches 50% and Doppelgänger achieves a 4% reduction. AVR reduces memory latency by 20%. Truncate follows with a 5% reduction. This is confirmed by MPKI, where AVR has less than half the misses compared to Truncate as over half of its approximate LLC requests hit in compressed blocks in the LLC or in DBUF.

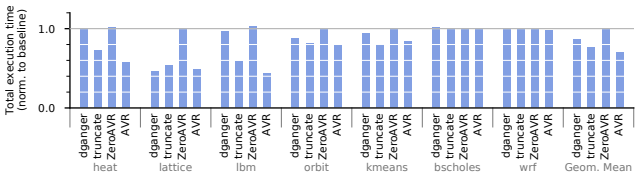


Figure 9: Execution time.

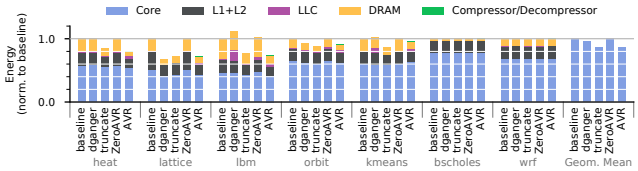


Figure 10: System Energy Consumption.

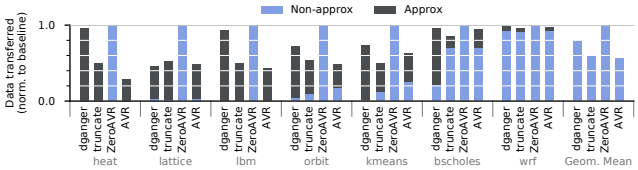


Figure 11: Memory Traffic.

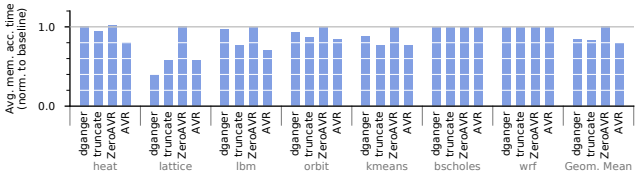


Figure 12: Average Memory Access Time (AMAT).

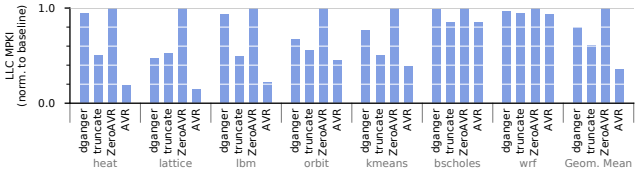


Figure 13: LLC misses per kilo-instruction (MPKI).

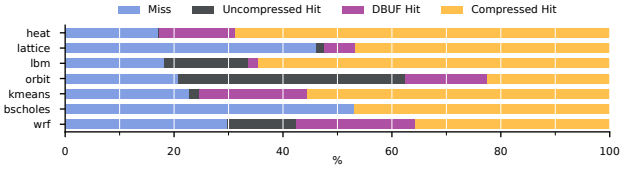


Figure 14: AVR LLC Requests on approximate cachelines.

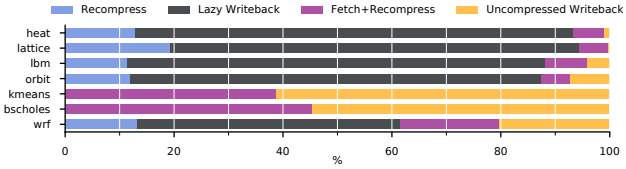


Figure 15: AVR LLC Eviction of approximate cachelines[2].

[2] *Recompress*: the evicted cacheline belongs to a compressed block available in LLC, which is updated and recompressed; *Lazy Writebacks*: the cacheline is evicted to memory uncompressed (lazily, without recompression) although it belongs to a compressed block (stored in memory); *Fetch+Recompress*: the compressed block, to which the evicted cacheline belongs, is read from memory and updated; *Uncompressed WB*: the evicted cacheline's block has failed to compress so the line is written back uncompressed.

Table 4: AVR compression ratio and footprint reduction

	heat	lattice	lbm	orbit	kmeans	bscholes	wrf
Compr. Ratio	10.5×	9.6×	15.6×	16.0×	2.3×	4.7×	3.4×
Mem. Footprint	12.6%	20.0%	7.9%	54.1%	58.5%	78.6%	89.6%

Lattice dataset is compressed by AVR by a factor of 9.6:1. AVR reduces execution time by 51% introducing 0.6% output error. Doppelgänger reduces execution time by 54% with an error of 0.2%. This is because *lattice* can exploit the effectively larger Doppelgänger LLC. Furthermore, Truncate achieves a speedup of 47% with an output error of 0.5%. Energy consumption follows the performance trends. AVR reduces baseline energy by 23%. Doppelgänger and Truncate energy consumption is reduced by 27% and 23% of the baseline, respectively. AVR memory traffic is reduced by 51% compared to baseline. That is similar to Doppelgänger's 54%. Truncate reduces the memory traffic by 47%. It is noteworthy that the large gap in MPKI between AVR (14% of baseline) and competing designs (48% and 53% for Doppelgänger and Truncate, respectively) is not reflected in the memory traffic volume. This is caused by frequent lazy writebacks leading to an inflated amount of read traffic when memory space is exhausted. AMAT follows the execution time trends. Doppelgänger leads with a 60% reduction. AVR AMAT is down by 43% compared to baseline and Truncate follows with 42%.

Lbm has about 98% of its footprint approximable, and AVR reduces it more than 15×. AVR is better than Truncate. It reduces baseline execution time by 57% vs. 42% for truncate, has 0.1% application output error (Truncate error is 0.6%), similar energy savings, lower memory traffic (33% vs. 50% for Truncate) and lower memory latency (30% reduction for AVR versus Truncate's 23%) due to very low LLC MPKI. Doppelgänger yields an excessive 22.3% output error, with a 3% improvement in execution time and no effect on total energy. The high output error is caused by edge-cases in Doppelgänger's approximation, where cache-lines at the extreme edges of their respective expected value span are considered approximately equal even though their absolute values are very different.

Orbit sees its total footprint reduced to 54% by AVR as half of its data are approximate and compress almost perfectly. AVR and Truncate introduce negligible error, while Doppelgänger causes strong artefacts leading to a runaway error exceeding 100%. AVR reduces execution time to 79% of baseline, Truncate trails behind at 82% followed by Doppelgänger with 86%. Truncate achieves the largest improvement in energy totaling 89% of baseline, AVR follows closely with a total of 92% and Doppelgänger with 93%. In spite of the high compression ratio, AVR only reduces memory traffic to 52%, outperforming by a narrow margin Truncate's 54%, while traffic for Doppelgänger is 65%. Memory latency follows a trend similar to execution time, with AVR achieving a reduction to 84%, Truncate yielding 86% and Doppelgänger 90% of the baseline.

K-means has a 58% reduction in memory footprint at the cost of 1.2% error. AVR achieves the highest instructions per cycle (IPC) count among all design points, but has the second shorter execution time after Truncate. That is because the application requires extra iterations to converge for the AVR, which increases the total number of executed instructions. Note, that k-means is the only benchmark used where the workload may vary based on the quality of the approximations, all other applications have a fixed number of instructions to execute. Doppelgänger matches the baseline execution time despite its slightly improved memory latency and reduced memory traffic and has negligible error. AVR reduces energy cost by 2%. Truncate, which runs an identical number of instructions to the baseline, reduces energy by 13%. Doppelgänger energy overhead is 3% due to its LLC design. AVR reduces memory traffic by 37% and Truncate by 50%. This difference is an artifact of AVR's higher number of executed instructions. Doppelgänger has a smaller reduction of memory traffic, 26% less than the baseline, primarily because its LLC performs better than the baseline, as confirmed by its MPKI results. Memory latency is the shortest for AVR and Truncate, each 23% lower than the baseline, Doppelgänger follows with 12%. It is noteworthy that 55% of the AVR approximate LLC requests hit in the compressed blocks stored in the LLC and another 20% in DBUF.

Blackscholes (bscholes) uses input data where some of the input fields are identical for multiple entries [48]. This has been exploited by the Doppelgänger design. About 30% of bscholes dataset is approximable and AVR reaches a compression ratio of 4.7:1. However, bscholes is not memory intensive. As a consequence, the evaluated designs have little impact. Nevertheless it is still interesting to discuss their behaviour. Indeed, the execution time of all designs is very close to the baseline as shown in Figure 9. This holds also for the energy consumption. Truncate and AVR reduce memory traffic by 15% and 6%, respectively. Doppelgänger reduces traffic by 3%, does not improve memory latency and reduces MPKI by 1%.

WRF has only 15% of its data marked as approximable, most of them geographically ordered weather metrics. AVR compresses these data with a 3.4:1 ratio reducing total memory footprint by 10%. Still it is an interesting application to discuss as a case where approximation does not offer a large benefit. AVR reduces execution time by 2% introducing 8.9% error to the application output. It reduces memory traffic only by 3% and has no effect on memory access time. Its MPKI is 7% lower than the baseline as over 50% of the approximable LLC requests hit in compressed blocks in the LLC or in the DBUF. Truncate has similar performance. It reduces execution time by 1% with 4.2% output error, reduces memory traffic by 5% and memory latency is unaffected. Finally, Doppelgänger causes 24.9% error with negligible performance impact.

Figures 14 and 15 show the breakdown of the LLC requests and evictions. In general, about 40-80% of the LLC requests hit on the DBUF or on compressed blocks. The latter case adds extra latency to the LLC hits for reading and decompressing the compressed block before serving a request. More precisely, the average LLC latency when hitting on a compressed block in the LLC is 20-30 cycles for wrf and kmeans, 74 for bscholes, and 40-50 cycles for the other benchmarks, which is still significantly faster than a DRAM access. The analysis for the AVR LLC evictions is mixed among benchmarks. For kmeans and bscholes about 40% of the evictions require fetching the block from memory and recompressing introducing traffic overheads, the remaining evictions are uncompressed written-backs because the block has failed to compress. On the contrary, the other benchmarks exploit the AVR lazy evictions in 45% to 80% of the cases avoiding fetching the compressed block on chip. Even including the lazily evicted cachelines, the average size of a block read from memory is similar to the one indicated by the compression ratio shown per benchmark in Table 4. That is about 5.1 memory accesses to read a block for kmeans, 3.4-3.8 for bscholes and wrf, and 1.2-2 for the other benchmarks. Finally, the reuse of blocks is indicative to the AVR performance gains; on average 7-10

unique cachelines of a block are used before eviction for bscholes and lattice and 13-16 cachelines for the other benchmarks.

In summary, for applications with high compression ratio (*heat, lattice, lbm*), AVR is better than competing designs. It achieves significant reduction in execution time (40-55%) and considerable energy savings (10-20%) with less than 1% output error. Memory traffic is also reduced for these applications by 50% to 70%, although in some cases less than expected based on the compression ratio. Orbit is an exception to this trend; although AVR achieves excellent compression ratio, execution time is reduced only by 20%. At medium compression ratio, i.e. in *k-means*, AVR has moderate performance gains (about 15%) despite increasing the number of executed instructions. At low compressibility, i.e. in *wrf*, AVR improvements are negligible as are its overheads. Moreover, in compute bound applications, i.e. *bscholes*, there is minimum impact. Note that AVR memory latency is substantially reduced and always lower than the compared approaches. Finally, when not approximating, AVR does not have notable overheads.

5 CONCLUSIONS

The AVR architecture improves the memory system using aggressive approximate compression. Thereby, AVR reduces memory traffic, utilizes more efficiently the off-chip bandwidth and achieves better performance and energy efficiency. AVR provides a low latency decompression scheme to reduce overheads in memory access time. Its LLC design stores both compressed and uncompressed data to increase its hit rate. AVR LLC evictions of compressible cachelines are handled in a lazy manner reducing the overhead of recompression. Moreover, keeping track of badly compressed blocks reduces unsuccessful compression attempts. Finally, the decompressed data selected to be stored in the LLC are carefully selected to avoid polluting the LLC with unwanted data. For applications with large part of the data being approximation-tolerant, AVR reduces memory latency by up to 45%, memory traffic by up to 70%, and achieves up to 55% lower execution time, up to 20% lower energy with less than 1% error to the application output.

ACKNOWLEDEGMENTS

This work is supported by the Swedish Research Council (contract number 2012-4924) under the ACE project.

REFERENCES

[1] 2015. NVIDIA Tegra X1: NVIDIA's New Mobile Superchip. whitepaper.
[2] 2016. Swedish Topological Survey HDB 50+ Västra Götaland, zone 63_3. Retrieved 2016-01-13 from https://www.lantmateriet.se/
[3] 2018. 1D K-Means, Open Source. Retrieved 2018-10-13 from https://github.com/eldstal/kmeans
[4] Junwhan Ahn et al. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA-42*. 336–348.
[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA-42*. 105–117.
[6] A. Angerd et al. 2017. A framework for automated and controlled floating-point accuracy reduction in graphics applications on GPUs. *TACO'17* (2017).
[7] Santosh Ansumali et al. 2003. Minimal entropic kinetic models for hydrodynamics. *EPL (Europhysics Letters)* 63, 6 (2003), 798.
[8] A. Arelakis et al. 2015. HyComp: a hybrid cache compression method for selection of data-type-specific compression methods. In *MICRO-48*. 38–49.
[9] Luca Benini et al. 2002. An adaptive data compression scheme for memory traffic minimization in processor-based systems. In *ISCAS*. IEEE.
[10] ASCF Center. 2018. FLASH4 User's Guide. http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug_4p6.pdf Online; Accessed 2019-04-18.
[11] V. K. Chippa et al. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *DAC*. 1–9.
[12] E. Choukse, M. Erez, and A. R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *MICRO-51*. 546–558.
[13] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *IPDPS*. IEEE, 730–739.
[14] Michael Doggett. 2012. Texture Caches. *IEEE Micro* 32, 3 (2012).
[15] Julien Dusser and André Seznec. 2011. Decoupled zero-compressed memory. In *Int. Conf. on HiPEAC*. ACM, 77–86.
[16] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *SIGARCH C.A. News*, Vol. 33. 74–85.
[17] D. Genbrugge, S. Eyerman, and L. Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA-16*. 1–12.
[18] Jill R. Goldschneider. 1997. *Lossy Compression of Scientific Data Via Wavelets and Vector Quantization*. Ph.D. Dissertation. Univ. of Washington.
[19] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.
[20] S. Hong et al. 2018. Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads. In *MICRO-51*. 326–338.
[21] Animesh Jain et al. 2016. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *MICRO-49*.
[22] Patrick Judd et al. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *Int. Conf. on Supercomputing*.
[23] J. Kim et al. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *ISCA*. 329–340.
[24] Sheng Li et al. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO-42*.
[25] Chi-Keung Luk et al. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, Vol. 40.
[26] E. Meijering. 2002. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proc. of IEEE* 90 (2002).
[27] N. Muralimanohar et al. 2009. CACTI 6.0: A tool to model large caches. *HP lab.* (2009), 22–31.
[28] M. O'Connor et al. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *MICRO-50*. 41–54.
[29] B. Panda et al. 2018. Synergistic Cache Layout for Reuse and Compression. In *PACT*.
[30] M. Pavlovic et al. 2011. On the memory system requirements of future scientific applications: Four case-studies. In *IISWC*. 159–170.
[31] G. Pekhimenko et al. 2016. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *MICRO-46*.
[32] J Quinn Michael. 2004. *Parallel Programming in C with MPI and OpenMP*. Technical Report. ISBN 0-07-058201-7.
[33] B.M. Rogers et al. 2009. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *ISCA-36*. 371–382.
[34] Paul Rosenfeld et al. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL* 10, 1 (2011), 16–19.
[35] L. Saldanha et al. 2009. Float-to-fixed and fixed-to-float hardware converters for rapid hardware/software partitioning of floating point software applications to static and dynamic fixed point coprocessors. *Des. Aut. for Emb. Sys.* 13, 3 (2009).
[36] Adrian Sampson et al. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46.
[37] Joshua San Miguel et al. 2014. Load Value Approximation. In *MICRO*.
[38] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel. 2016. The bunker cache for spatio-value approximation. In *MICRO-49*. IEEE, 1–12.
[39] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger. 2015. Doppelganger: A cache for approximate computing. In *MICRO-48*.
[40] S. Sardashti et al. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *MICRO*.
[41] S. Sardashti, A. Seznec, and D.A. Wood. 2016. Yet another compressed cache: a low-cost yet effective compressed cache. *ACM TACO* 13, 3 (2016), 27.
[42] V. Sathish et al. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *PACT*.
[43] A. Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *ISCA-21*. 384–393.
[44] A. Shafiee et al. 2014. MemZip: Exploring unconventional benefits from memory compression. In *HPCA*. 638–649.
[45] B. Thwaites et al. 2014. Rollback-free Value Prediction with Approximate Loads. In *PACT*. 493–494.
[46] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH C.A. News* 23, 1 (1995), 20–24.
[47] A. Yazdanbakhsh et al. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *ACM TACO* 12, 4 (2016), 62.
[48] A. Yazdanbakhsh et al. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test* 34, 2 (2017).
[49] Y. Zhou and D. Wentzlaff. 2016. MITTS: Memory Inter-arrival Time Traffic Shaping. In *ISCA-43*. 532–544.