AVR: Reducing Memory Traffic with Approximate Value Reconstruction

Albin Eldstål-Damlin, Pedro Trancoso, Ioannis Sourdis
CSE dept., Chalmers Univ. of Technology, Gothenburg, Sweden
Email: {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 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 less than 1% error to the application output.

I. 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 [1]. Besides the long memory latency, memory bandwidth severely limits performance, energy efficiency and scalability of Chip Multiprocessors (CMPs) [2]. 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 [3]–[5]. On the other hand, memory bandwidth is pin limited [3], [6] and power constrained [7] and is therefore more difficult to scale [2]. 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 [7].

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) [8]. GPUs use application-specific compression, applied to texture and color data [9], and often solve the easy part of the problem, handling read-only data [10]. Current state-of-the-art, lossless memory compression techniques achieve on average a 2:1 to 4:1 compression ratio [11]. 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 [12], [13]. Figure 1 shows the compression ratio of datasets in such cases as well as the average error introduced in the application output by the approximations (setup in Section IV). Approximately compressing parts of the data with up to 16:1 ratio reduces the overall application memory footprint up to 10 times and suffers less than 1.5% error in the application output.

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 [14]–[16], but has difficulties capturing irregular data variations. Approximate Deduplication of individual cachelines increases cache capacity [17], however, multiple values need to match cacheline granularity. Some form of lossy compression as reduced precision has been applied in approximate computing, but they are constrained to handling single values truncating their least significant bits [10], [18], [19] and therefore achieve limited compression ratio.

In this work, Approximate Value Reconstruction (AVR) is proposed for utilizing more efficiently memory bandwidth in approximation tolerant applications. 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 extracts a “summary” of the approximated data based on which it attempts to approximately reconstruct values in the processor. AVR addresses a number of challenges. Firstly, summarizing (compressing) and reconstructing (decompressing) the data needs to be generic, rather than application specific, it should also introduce minimum error, and add minimum latency and energy overheads. Secondly, managing (updating, re-packing, storing) the compressible data in the main memory as well as
on-chip needs to be performed efficiently so as to maximize the compression benefits and minimize their costs. Finally, a mechanism for regulating the error introduced by the approximations is required. In effect, memory bandwidth is better utilized improving execution time and energy efficiency.

Concisely, the contributions of this work is a novel architecture that:

- supports aggressive, approximate memory compression exploiting similarities across values;
- maximizes the effect of memory compression and minimizes its overheads 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;
  - selecting which data to store in LLC after de-compression.

Thereby, memory bandwidth is utilized more efficiently having a significant impact in system performance and energy efficiency as shown in our experiments.

The remainder of this paper is organized as follows. Section II discusses related work on approximate computing focusing on data compression. Section III describes the proposed AVR architecture. Section IV presents our evaluation results. Finally, Section V draws our conclusions.

II. RELATED WORK

Large classes of applications are inherently tolerant to approximations [12]. 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 at different aspects of computing systems. Some of them target the aforementioned memory bottlenecks.

Approximate load value prediction techniques reduce memory latency by providing a predicted value substantially faster than fetching the actual one from memory [14]–[16]. They further improve memory bandwidth utilization as they do not require to bring the actual value 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 aims to use 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 to compression is that load value prediction aims primarily at reducing load latency rather than memory bandwidth because they fetch from memory the precise values of (at least a portion of) the predicted loads.

Reducing the precision of floating point [10], [18], [20] and fixed point [19] numbers has been used to alleviate the memory bandwidth bottleneck in deep neural networks [19]. GPU workloads [10], [20] and other approximation tolerant applications [18], 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 [21].

Approximate compression has been applied to caches, too. Doppelgänger deduplicates approximately similar cachelines to compress data [17]. The subsequently described Bunker cache 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 [22]. Both designs exploit similarities between cachelines. This requires that values need to be packed in the cachelines with a fixed order and any misalignment between the data-structure and its memory mapping may prevent the deduplication of similar or identical data.

III. 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 multicore system with a shared Last Level Cache (LLC) as depicted in Figure 2.

In a nutshell, the processor sends and receives compressed memory blocks to and from the main memory and stores them compressed and partly uncompressed in the LLC. Storing compressed memory blocks in the LLC trades LLC capacity for fewer memory accesses. AVR further reduces accessing and de-compressing of memory blocks by introducing lazy evictions of uncompressed LLC cachelines. Finally, the overheads of unsuccessful compression attempts is minimized by keeping history of previous compression attempts per block.

The AVR architecture requires the following additions: a compression-decompression module to reconstruct data coming from the memory and to summarize data before sending it back to memory; a metadata table for storing information about the compressibility of the memory blocks; finally the LLC design requires changes for storing compressed data in addition to the uncompressed ones. Next, each one of the
above modules are discussed separately, after first presenting the format of the AVR memory blocks. Then, the LLC and memory operations in the AVR system are discussed.

A. Memory Blocks

Similar to most techniques that focus on data approximations [17], [18], [23], AVR considers that the programmer annotates regions of the address space that can be approximated, 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 4. 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 multiple cachelines [2] as shown in Figure 5. In our implementation, that is blocks of 16 64B cachelines, in total a quarter of a physical 4kB page. AVR compresses the 16 cachelines of the block to a single cacheline summary aiming for a 16:1 compression ratio. This ratio is suitable for the alignment of the compressed block, fits well the AVR compression method, and does not limit the compressibility for any of the applications in the study. The summary is stored in the first out of the 16 cachelines of the memory block as shown in Figure 5a. In case this compression produces approximations of some values that exceed a particular error threshold, these values are characterized as outliers and are stored explicitly, uncompressed in the compressed block. As shown in Figure 5a the outliers are placed after the summary cacheline, together with a bitmap that indicates their location in the uncompressed block. This bitmap occupies half a cacheline in a block that contains outliers and together with the...
C. 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 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 5 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 back to floating point after decompression. 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 subdivide memory blocks into a square 2D array and the second in a 1D linear array. Figure 6 illustrates an example of 2D downsampling, where the compression is performed by averaging the values of a sub-block (light-grey) into a single value, which then represents the entire sub-block. For decompression, the average values are distributed evenly and bi-linear interpolation is applied to reconstruct the approximate values in-between. As discussed above, all values are converted to a fixed-point format before compression to reduce the complexity of the arithmetic operations needed. In essence, compression computes the average value of a sub-block and requires up to five levels of 64-bit integer additions (for sub-blocks of up to 32 values) and a division by the constant number of values on the sub-block, which is reduced to a single shift when the sub-block size is a

2 In our evaluation, approximable variables that were originally 64-bit double-precision floating-point values have been converted to 32-bit single-precision - see Section IV-B

![Fig. 5: AVR compressor module.](image-url)
power of two. In our implementation, compression requires 15 cycles. Decompression is simpler, it requires two stages of linear interpolation and has a total latency of 10 cycles.

**Error calculation & Outliers selection:** Using lossy compression entails the risk of accumulating error of values that are recompressed multiple times. In order to limit the accumulated error, compression is skipped in case the introduced 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 is controlled for a single compression operation, which limits the error from exploding but does not guarantee that the accumulated error of a sequence of compression operations does not impact on the overall application output. Such guarantee would require more storage and computation which would be expensive to support.

The error per individual value is calculated in floating point format as follows:

$$
\text{Error} = \left\lfloor \frac{\text{Original Value} - \text{Approximated Value}}{\text{Original Value}} \right\rfloor \times 100
$$

For a value to be approximated with a relative error within $T_1$, a comparison between the original and approximated value should result in the exact match of (i) their exponents and (ii) their $N$ most significant bits (MSbits) of the mantissas; 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 selecting and compacting the outliers and in parallel computing the average block error for the values that are not outliers. Outliers are stored in order. Each set bit in the bitmap marks the location of a single element in the uncompressed version of the memory block to be replaced by an outlier value on decompression. 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, exponent and $N$ MSbits of mantissa are identical for the original and approximate values, otherwise they would be outliers. So, the average error is calculated by subtracting the remaining $23 - N$ least significant mantissa bits between each original and approximated value. The average error is the average of these integer subtraction results for all the non-outliers values (up to 256 values). Computing the average error also fits in 16 cycles. Finally, during the outliers selection and compaction it is determined whether the outliers fit to the maximum allocated number of cachelines and hence whether in the end the block is compressible. It is relevant to note that the outliers are stored in floating-point format, with precision reduced to half by truncating the 16 LSbits of the mantissa.

**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 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 IV, the total latency for compressing a block is 49 (processor) cycles, and for decompressing a block is 12 cycles. Decompression is more critical for the performance of the system as it is part of a memory reads. Compression is less critical because it is part of the write backs.

### D. Last Level Cache

The AVR Last Level Cache (LLC) stores uncompressed cachelines (UCL) as well as compressed memory blocks, each compressed block split in one to eight “compressed” cachelines (CCL) depending on its compressibility. In case a
memory block is uncompressed, only selected uncompressed cachelines are stored in the LLC as indicated above. 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 [28], 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 7 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, CCL0, CCL1, and CCL2; one for the summary of the block and two for the bitmap and the outliers. All three cachelines of the 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 7. 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 block cachelines. The first cacheline of the compressed block, CCL0, 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, CCL1, and CCL2, 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 CCL 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 is based on the Decoupled Sectored Caches [28] and shares some common elements with Decoupled Compressed Cache [29]. Figure 7 depicts the fields stored in each tag array and BPA entry. A tag array entry stores the following fields:

- **Block Tag:** The memory block tag.
- **CCL count:** the number of cachelines needed for storing the compressed form of the 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 that the compressed version of the block is dirty. The LRU of the tag is updated when a UCL of the block is accessed and used for tag-entries replacement. A BPA entry stores the following fields:

- **CL-type:** one bit indicating a UCL or CCL.
- **CL-id:** in case of UCL, 4-bits are used to store the cacheline tag suffix depicted in the address breakdown; in case of a CCL, 3 of the above 4 bits are used to store their 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 CCL, the compressed block part number is serving the same purpose; that is when looking up the i-th fraction (cacheline) of the compressed block the CL-id of the matching BPA entry should be i. Finally, the LRU bits of a CCL 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 only as part of a compressed block. In the first case, an UCL lookup is performed. In the second case, CCL lookup is performed. Figure 8 illustrates the AVR LLC lookups. Below we discuss each case in more detail.

A request 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 request for a compressed block in the LLC requires one or multiple accesses to the LLC, as many as the CCLs the block is composed of (CCL count). The tag array, BPA, and data array are accessed using the tag index. The block tags in the set are matched. In parallel, the entries in the BPA set that store CCLs compare their CL-id with zero. Here this field indicates the offset of the compressed cacheline in the compressed memory block and looking up for the first part of the block 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 part of the compressed block, the \textit{CCL count} is also retrieved to determine the 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, \textit{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 CCLs, compete equally. In case a CCL is evicted, then all the other CCLs of the same compressed block need to be evicted, too, and 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 \textit{CCL 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 CCLs of a block needs to be performed together at consecutive sets starting from the one indicated by the tag index.

\textit{E. 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 in the previous paragraphs.

\textbf{LLC Requests:} A cacheline request from the lower cache level to the LLC may have the following three outcomes as illustrated in Figure 8:

- 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.

\textbf{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 9.

In a 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 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 CCL, 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.

When the compressed block is no longer in memory, it is compressed again and stored in the LLC. If the recompression is skipped, the dirty UCL is written back to memory directly.

4In a more aggressive approximation approach, allowing to lose parts of the compressed memory block that store outliers could be acceptable.
TABLE I: Simulation parameters

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

IV. 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. Subsequently, we show our evaluation results and comparison with the related designs in terms of performance, energy cost, and application output error.

A. System Configuration

We evaluated the AVR using an in-house simulator, implemented on top of Pin [30], that employs an interval-based processor model, as proposed by Genbrugge et al. [31], and a cycle-accurate model of the memory hierarchy that uses DRAMSim2 for modelling main memory [32]. McPAT [33] and CACTI [34] 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 and a 28nm technology library 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 I. 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.

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 [10], [18], [19] (Truncate), and finally (ii) Doppelgänger [17], which is the closest and best performing related work on approximate data compression [17] (Dganger). 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.

B. Benchmarks

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 II. 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 [35]. For all of the applications we use the mean of the relative errors for each output value as our quality metric. Benchmarks for approximate computing (AxBench [36]) considers 10% relative output error, but it is solely up to the application provider to 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.

C. 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.

D. Experimental Results

We present next our experimental results for each benchmark comparing AVR with other related designs, namely, Doppelganger, 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 [10] [11] [12] [13] and [14] as well as in terms of application output error shown in Table III. In addition, AVR approximate LLC requests and evictions are analyzed, depicted in Figures [5] and [16]. Each application has its own particular characteristics that affect multiple performance metrics; it is therefore simpler to present our results per application rather than per metric.
Before presenting the results of each application separately, a few observations common to all applications 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 10 and 11): 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 similar to the baseline LLC achieving the same MPKI as shown in Figure 14. In our experiments AVR LLC devotes 2-16% of its capacity for storing compressed blocks.

### TABLE III: Application output error

<table>
<thead>
<tr>
<th>Application</th>
<th>dganger</th>
<th>truncate</th>
<th>AVR</th>
<th>heat</th>
<th>lattice</th>
<th>lbm</th>
<th>kmeans</th>
<th>blackscholes</th>
<th>wrf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (norm. to baseline)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
<td>50%</td>
<td>48%</td>
<td>53%</td>
<td>0.0</td>
<td>1.2%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Memory traffic (MPKI)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>43%</td>
<td>47%</td>
<td>54%</td>
<td>0.0</td>
<td>1.2%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Energy consumption (norm. to baseline)</td>
<td>0.0</td>
<td>0.0</td>
<td>15%</td>
<td>18%</td>
<td>15%</td>
<td>18%</td>
<td>0.0</td>
<td>1.2%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Total execution time</td>
<td>0.0</td>
<td>0.0</td>
<td>43%</td>
<td>47%</td>
<td>54%</td>
<td>48%</td>
<td>0.0</td>
<td>1.2%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

*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 WR*: the evicted cacheline’s block has failed to compress so the line is written back to uncompressed.

Heat is a 2D thermodynamics application that iterates over a grid of values and computes the propagation of heat. Data representing temperature are marked as approximable and have excellent compression (about $8 \times$ smaller total memory footprint and a 10:1 compression ratio of approximable data) as shown in Figure 1. AVR reduces execution time by 43% compared to the baseline introducing only 0.7% error. That is almost double the speedup 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 number of misses compared to Truncate as nearly half of its approximate LLC requests hit in compressed blocks in the LLC or in DBUF.

Lattice is a 2D Computational Fluid Dynamics (CFD) application and simulates air flow over a solid object (a car model in our experiments). Data related to pressure and air flow are marked as approximate and AVR can compress it by a factor of 9.6:1, reducing memory footprint 5$x$. AVR reduces baseline execution time by 51% introducing 0.6% output error. Doppelganger reduces baseline execution time by 54% with an error of 0.2%. This is because lattice can exploit an effectively larger LLC such as the Doppelganger as shown in the LLC MPKI results. Furthermore, Truncate is 4% slower than AVR and has 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 51% reduced from baseline. That is similar to Doppelganger’s 54%. Truncate reduces the memory traffic by 47%. It is notable 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 space in memory is exhausted. Memory latency 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** is a 3D Computational fluid dynamics (CFD) simulation implementing the Lattice-Boltzmann method. Its data, provided by SPEC2006, simulates the flow of a fluid across a sphere. The data approximated are the velocity vectors of the fluid, the final state of which is the output of the application. These data are about 98% of the memory footprint, and can be reduced more than 15× using our approximate compression method, yielding a total footprint reduction above 12×. AVR reduces execution time by 57% compared to baseline and Truncate by 42%. The two achieve 0.1% and 0.6% output error, respectively. The two have similar energy savings (22% for Truncate versus 24% for AVR). A similar relation holds for total memory traffic, which is reduced by 50% for Truncate and 67% for AVR. AVR memory latency is slightly better (30% reduction versus Truncate’s 23%) due to very low LLC MPKI, which allows it to achieve the highest performance for **lbm**. Doppelgänger yields an excess of 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.

**K-means** is a clustering algorithm applied to topological data for grouping locations in zones based on their elevation. These topological data are marked as approximate and when compressed can reduce the total memory footprint to 58% (compression rate of approximable data 2.3:1). 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 an extra iteration 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 baseline energy cost by 2%. Truncate reduces energy by 13% due to executing fewer instructions. Doppelgänger energy overhead is 3% due to its LLC design. AVR reduces baseline 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 over 55% of the AVR approximate LLC requests are hitting in the compressed blocks stored in the LLC and another 20% hits in the DBUF.

**Blackscholes** is an application for financial forecasting. It receives historical data about stock options and attempts to predict their future price. Based on the input data available in [36], some of Blackscholes input fields are identical for different entries of the history. That has been exploited by the Doppelgänger design. In our experiments, we mark these fields as approximate. Despite having about 30% of its data being approximate (AVR reaching a compression ratio of 4.7:1 for a total footprint reduction of 27%), blackscholes is not memory intensive. The baseline IPC is above 6 and the data have very little reuse. 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 [10]. 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 increases MPKI by 26%.

**WRF** is a model for weather forecasting. We have identified about 15% of its data to be approximate mostly related to geo data carrying information of various weather metrics. Although these data reach a compression ratio of 3.4:1, they reduce the total memory footprint by only 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 6% lower than the baseline as 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. It further reduces memory traffic by 5% and memory latency is unaffected. Finally, Doppelgänger reduces execution time by 1% and introduces an output error of 24.9%. Memory traffic is reduced by 2% and memory latency is left unaffected.

Figures [15] and [16] offer deeper insight of the AVR functionality showing the breakdown of the LLC requests and evictions. In general, about half 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. The overhead of hits on compressed blocks is a few tens of cycles to the AVR LLC hit latency, which otherwise is equal to a conventional LLC as indicated in Table [4]. More precisely, the average delay added to the LLC latency when hitting on a compressed block in the LLC is 20-30 cycles for wrf and kmeans, 74 for blackscholes, 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 blackscholes about 40% of the evictions require to fetch the compressed block from memory and recompress introducing overheads to the memory traffic and the remaining evictions are written back uncompressed due to bad compressibility of the block. On the contrary, the remaining benchmarks exploit the AVR lazy evictions technique as 45% to 80% of the evictions are written
back to the memory lazily substantially reducing the cases that require to fetch the compressed block on chip. Even including the lazily evicted cachelines the average block size read from memory is similar to the one indicated by the compression ratio, which is discussed per benchmark in the previous paragraphs. That is about 5.1 cachelines read per block for kmeans, 3.4-3.8 for blackscholes and wrf, and 1.5-2 for the other benchmarks. Finally, the reuse of blocks is indicative to the AVR performance gains; on average up to 2 cachelines of a block are used before eviction for kmeans, blackscholes, and wrf, while 7-9 cachelines of a block are used for the other benchmarks.

In summary, for applications with high compression ratio (heat, lattice, lbn), AVR is better than competing designs. It achieves significant reduction in execution time (20-55%) and considerable energy savings (10-20%) with less than 1% output error. Memory traffic is also reduced for these applications by 10% to 70%, although in some cases less than expected based on the compression ratio. 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. blackscholes, 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.

V. 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.

REFERENCES