AVR: Reducing Memory Traffic with Approximate Value Reconstruction

Albin Eldstål-Damlin, Pedro Trancoso, Ioannis Sourdis
CSE dept., Chalmers Univ. of Technology, Gothenburg, Sweden

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% and achieves up to 70% reduction in execution time, up to 50% in energy consumption 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, however, 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 2% 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 at 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 after decompression.
- Thereby, memory bandwidth is utilized more efficiently having a tremendous 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.

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

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 [20]. 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.

As an introduction, 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\(^1\) as shown in Figure 3. 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. The summary is stored in the first out of the 16 cachelines of the memory block as shown in Figure 3a. 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 3a, the outliers are placed after the summary cacheline, together with a bitmap that indicates their position in the uncompressed block. This bitmap occupies half a cacheline and together with the outliers 1-7 additional cachelines of the block. As a consequence, a compressed block occupies up to half of its available memory space (1-8 out of the 16 cachelines) achieving a 2:1 worst case compression ratio. The remaining part of the space allocated for the memory block can be utilized for lazy evictions; that is for writing back dirty uncompressed cachelines of the block, which are evicted from the LLC. Thereby, AVR avoids bringing a compressed block on-chip to recompress it every time one of its dirty cachelines is evicted from the LLC. This is possible until the block space is exhausted, then the block and the lazily evicted dirty uncompressed cachelines are called on-chip for recompression. In case a memory block fails to fit in 1+7 cachelines, it remains uncompressed as shown in Figure 3b. A memory block is stored uncompressed in memory also when it is marked as non-approximable.

\(^1\)We consider a cacheline, i.e., 64B, being the granularity of accessing the main memory.

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\(^2\) 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 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.

\(^2\)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.
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 unbiased, 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 compression.

**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. For example, a floating-point number with a very high exponent may overflow the integer portion of a fixed-point representation, leading to truncation of its most-significant bits. 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. The bias is stored with the block’s metadata and reverted 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 requires an 8-bit addition to all decompressed values and requires one cycle under compression.

**Float to fixed & fixed to float conversions:** Converting from float to fixed point numbers and vice-versa is implemented as described in [24] requiring a single cycle.

**Compression:** Although various lossy compression algorithms can be considered, we opted for a method that is simple to implement without introducing very high error. In AVR, memory blocks are compressed using downsampling [25]. 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 considered. The main two variants differ in the considered placement of the values in the block before partitioning them to sub-blocks of 16 values. In our experiments all datasets are in floating point format. For fixed point numbers a subtraction and a subsequent comparison would be required.
bit map is used for 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 cacheline is required for computing the average error of the block. The 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 LSBs 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 prefetcher 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.

\[D. \text{ Last Level Cache} \]

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 [25], 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 third 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 occupies when compressed 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 0xC and 0xD. On the contrary, 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

![Fig. 7: AVR Last Level Cache.](attachment:image.png)
management of memory blocks simpler.

**LLC Structure:** Structurally, the AVR LLC is based on the Decoupled Sectored Caches [26] and shares some common elements with Decoupled Compressed Cache [27]. 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 the tag way 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 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 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 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, 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 in the LLC 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 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.

**E. Memory operations**

We explain next the AVR memory operations at the LLC and main memory level. More precisely, we explain how a request in 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.

**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.
Eviction
LLC
Writeback
Possible?
Lazy WB
C-Block in LLC?

the compressed memory block that store outliers could be acceptable.
Accordingly, the above metadata fields are updated in the
representation. If the recompression is skipped, the dirty UCL
whether to proceed with the current compression or not.
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 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.
Note that before compressing a memory block that is cur-
rently uncompressed, because its last attempt for compression
failed, its compression history and counter of skipped compres-
sions 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.

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

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 9.

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 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.

**IV. Evaluation**

In this section we evaluate the effectiveness of the AVR
architecture. We first describe our experimental setup, pre-
senting 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 system using an Pin-based instru-
mation tool [28] coupled with DRAMSim2 [29], McPat
[30] and CACTI [31] were used to model power and latency of
the system considering 32nm technology. The AVR compres-
sion 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 [12], [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, full 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
downsampling 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 with using the benchmarks listed in
Table I. 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4 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, 9 clk latency</td>
</tr>
<tr>
<td>L3 cache</td>
<td>4MB shared, 16-way, 15 clk access latency</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB DDR4, 1 channels, 1600MHz</td>
</tr>
</tbody>
</table>

**Table I: Simulation parameters**

**Fig. 9: AVR evictions.**
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 [32] as explained next. For all of the applications we use the mean of the relative errors for each output value as our quality metric.

### 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, 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 10, 11, 12, 13, and 14 as well as in terms of application output show in Table III. In addition, AVR approximate LLC requests and evictions are analyzed, depicted in Figures 15 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 any performance overhead when it does not have data to approximate (Figure 10). ZeroAVR energy overhead is negligible and below 1% as shown in Figure 11. Moreover, its AVR Decoupled LLC performs similar to the baseline LLC achieving the same MPKI as shown in Figure 14. Another observation is that in our experiments AVR LLC devotes 2-14% of its capacity for storing compressed blocks.

*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× smaller memory footprint) as shown in Figure 1. AVR reduces execution time by almost 40% compared to the baseline introducing only 0.3% error. That is almost double the speedup compared to Truncate that has similar error. Doppelgänger has an 8% slowdown 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 to 87% and 86%, respectively. AVR has slightly higher LLC energy cost than Truncate. Furthermore, Doppelgänger introduces an energy overhead of 3% due to its LLC design. An interesting observation is that despite the significantly better compression ratio of AVR in approximate data (10:1) versus Truncate (2:1), AVR has about 1/3 more (approximate) memory traffic. This is partly because of the large number of lazy LLC evictions, which increase the size of compressed memory blocks in memory. Still, a design without lazy evictions would require at an eviction the compressed block to be brought on chip for recompression, introducing equal if not higher memory traffic. AVR reduces memory latency below 60% of the baseline. Truncate follows with 80% of baseline AMAT. This is confirmed by MPKI, where AVR has half the number of misses compared to Truncate as 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 reduce memory footprint 5×. AVR reduces baseline execution time to 30% introducing only 0.6% output error. Although 1.8× slower than AVR, Doppelgänger reduces baseline execution time by 46% with an error of 0.2%. This is because *lattice* can exploit an effectively larger LLC such as the Doppelgänger as shown in the LLC MPKI results. Furthermore, Truncate is 2.3× slower than AVR and has 4× higher output error. Energy consumption follows the performance trends. AVR reduces baseline energy to half. Doppelgänger and Truncate energy consumption is 81% and 86% of the baseline, respectively. AVR memory traffic is 70% lower than the baseline. That is similar to Doppelgänger memory traffic. In *lattice*, we can observe the opposite phenomenon than in *heat*. Here, AVR is the one that has less approximate traffic than Truncate. This is explained from the breakdown of the AVR LLC evictions in the two applications shown in Figure 16. In *lattice*, 24% of the AVR LLC evictions are served by recompressing the respective compressed block stored in the LLC; for *heat* this is only 1% forcing AVR to access DRAM more often. Memory latency follows the execution time trends. AVR AMAT is down to 26% of the baseline and the second best design is Doppelgänger that has 36% of the baseline AMAT. Truncate follows with 67%. The above are in pair with their MPKI results.

*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.
TABLE III: Application output error

<table>
<thead>
<tr>
<th></th>
<th>heat</th>
<th>lattice</th>
<th>lbm</th>
<th>kmeans</th>
<th>blackscholes</th>
</tr>
</thead>
<tbody>
<tr>
<td>dganger</td>
<td>3.0%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>truncate</td>
<td>0.2%</td>
<td>2.7%</td>
<td>0.6%</td>
<td>0.0%</td>
<td>1.4%</td>
</tr>
<tr>
<td>AVR</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0.1%</td>
<td>1.2%</td>
<td>0.5%</td>
</tr>
<tr>
<td>ZeroAVR</td>
<td>3.9%</td>
<td>4.2%</td>
<td>8.7%</td>
<td>4.2%</td>
<td>4.2%</td>
</tr>
<tr>
<td>Non-approx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These data are about 98% of the memory footprint, which can be reduced more than $12\times$ using our approximate compression method. AVR reduces execution time to 62% of the baseline and Truncate to 68% with $6\times$ higher error. Although Truncate has similar performance with AVR, it has significantly more energy savings (21% versus 9% for AVR). That is due to AVR’s higher memory traffic (88% versus 50% for Truncate). The additional AVR traffic is due to additional approximate accesses and the reason is similar to heat. The high compressibility of lbm data leaves more memory space for more lazy evictions which increase the size of a compressed block in memory and hence the AVR memory traffic. Still, AVR memory latency is substantially better than the other designs (27% of the baseline AMAT) due to very low LLC MPKI, which allows it to achieve the highest performance for lbm.

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 memory footprint to 58%. 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 increases baseline energy cost by 1%. On the contrary, Truncate reduces energy by 11% due to executing fewer instructions. Doppelgänger energy overhead is 2% due to its LLC design. AVR reduces memory traffic to 63% of the baseline and Truncate to 51%. This difference is an artifact of AVR’s higher number of executed instructions. Doppelgänger has a smaller reduction of memory traffic, about 20% 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, 75% of the baseline, Truncate follows with 82% and Doppelgänger with 92%. It is noteworthy that over 55% of the AVR approximate LLC requests are hitting in the compress 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 [59], 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, blackscholes is not memory intensive. The baseline IPC is above 3 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 16% and 6%, respectively. Doppelgänger adds about 2% memory traffic, does not improve memory latency and increases MPKI by 11%.

WRF is a model for weather forecasting. We have identified about 40% of its data to be approximate mostly related to geo data carrying information of various weather metrics. Although approximate, these data do not compress very well as they reduce the memory footprint by only 10%. Still it is an interesting application to discuss as a case where AVR does not compress well. AVR reduces execution time by 4% introducing 8.7% error to the application output. It reduces memory traffic only by 3% and memory access time by 9%. Its MPKI is 7% lower than the baseline as more than 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 2% with 4.2% output error. It further reduces memory traffic by 5% and memory latency by 3%. Finally, Doppelgänger reduces execution time by 1% and introduces an output error of 3.9%. Memory traffic and memory latency are reduced by 2% and 4%, respectively.

In summary, for applications with high compression ratio (heat, lattice, lbm), AVR is better than competing designs. It achieves significant reduction in execution time (40-70%) and considerable energy savings (10-50%) 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 10%) 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 traffic and memory latency substantially and achieves up to 70% lower execution time, up to 50% lower energy with less than 1% error to the application output.

References