Enabling Technologies for Memory Compression: Metadata, Mapping, and Prediction

Arjun Deb
University of Utah, UT, USA
arjundeb@cs.utah.edu

Ali Shafiee
University of Utah, UT, USA
shafiee@cs.utah.edu

Rajeev Balasubramonian
University of Utah, UT, USA
rajeev@cs.utah.edu

Paolo Faraboschi
Hewlett Packard Labs, CA, USA
paolo.faraboschi@hpe.com

Naveen Muralimanohar
Hewlett Packard Labs, CA, USA
naveen.muralimanohar@hpe.com

Robert Schreiber
Hewlett Packard Labs, CA, USA
rob.schreiber@hpe.com

***Abstract*—Future systems dealing with big-data workloads will be severely constrained by the high performance and energy penalty imposed by data movement. This penalty can be reduced by storing datasets in DRAM or NVM main memory in compressed formats. Prior compressed memory systems have required significant changes to the operating system, thus limiting commercial viability. The first contribution of this paper is to integrate compression metadata with ECC metadata so that the compressed memory system can be implemented entirely in hardware with no OS involvement. We show that in such a system, read operations are unable to exploit the benefits of compression because the compressibility of the block is not known beforehand. To address this problem, we introduce a compressibility predictor that yields an accuracy of 97%. We also introduce a new data mapping policy that is able to maximize read/write parallelism and NVM endurance, when dealing with compressed blocks. Combined, our proposals are able to eliminate OS involvement and improve performance by 7% (DRAM) and 8% (NVM), and system energy by 12% (DRAM) and 14% (NVM), relative to an uncompressed memory system.**

I. Introduction

Memory system technologies are rapidly evolving, with new 3D-stacked devices and non-volatile memories (NVMs) on the horizon. As processors move towards specialization with accelerators, one can argue that memory systems should be specialized as well. Indeed, IBM already designs custom DIMMs for their Power8 line of processors [10]. These DIMMs include a Centaur memory buffer that has scheduling, caching, and other RAS features [10]. In the future, we can expect to see more system vendors designing specialized memory architectures that are not as constrained by JEDEC standards. A prominent example of such memory specialization is Near Data Processing (NDP) [14], [5], [1].

In this paper, we focus on specialized memory architectures that provide support for memory compression. Compression will likely be a key ingredient in a system's ability to grapple with emerging big data applications, regardless of whether data is resident on DRAM or NVM.

The concept of memory compression has been around for decades, with IBM's MXT architecture representing an early commercial example [22]. However, memory compression has not been widely used in modern systems. While the many benefits of compression are evident [21], [18], [20], the system-level complexity of implementing compression has been non-trivial. A number of recent works have reduced this complexity [21], [18], [16]. The work in this paper continues this progression, resulting in a memory compression architecture that is completely invisible to the OS.

Similar to the approach proposed in recent works [21], [20], we do not use compression to grow the effective memory capacity (thus avoiding OS involvement). Instead, compression is used to read/write blocks that are smaller than the standard 72-byte blocks. This reduces energy per read/write; it also reduces the impact of writes on NVM endurance. By reducing bandwidth utilization, this has the potential to improve performance as well. However, the solutions of Shafiee et al. [21] (*MemZip*) and Sathish et al. [20] require separate metadata storage to track the compressibility of every cache line, thus involving the OS.

The first contribution of this paper is to eliminate all OS involvement by encoding the compression metadata in the field typically reserved for ECC. We are able to do this without an increase in silent data corruption (SDC) or detected unrecoverable error (DUE) rates.

In this new design, since separate metadata is eliminated, the memory controller is unaware of the block size before a read operation. Therefore, the low energy and low bandwidth benefits of compression are not exploited by read operations. We introduce prediction mechanisms that allow the memory controller to guess the block size with a very high accuracy. The prediction mechanisms have an accuracy of 97% and are able to reduce memory energy by 24%. They are able to offer performance that is within 1% of an oracular scheme.

Third, we recognize that the mapping of compressed blocks to memory chips has a significant impact on wear leveling, power per chip, and performance. We therefore design new mapping policies that simultaneously improve all three of these metrics by uniformly spreading activity within a memory rank. Relative to the default mapping policy, our proposed *permuted* mapping policy is able to reduce activity variance by 19×.

This work was supported in parts by Hewlett Packard Labs and NSF grants 1302663 and 1423583.

II. Background

A. Prior Work in Memory Compression

A number of memory compression approaches have been considered in the last 15 years [22], [7], [18]. These approaches were primarily designed to boost memory capacity by compacting a large page into a smaller page. While this is an attractive feature, it inevitably requires an overhaul of the OS paging policies and mechanisms. The high complexity of this overhaul, whether real or perceived, is a major obstacle for commercial adoption.

Early compression techniques have several other drawbacks that have been progressively addressed by recent works. First, the high latency of compression and decompression [22], [7] has been reduced to a handful of cycles by the Base-Delta-Immediate (BΔI) [19] algorithm. BΔI is very effective for workloads that exhibit value locality, thus striking a sweet spot in terms of latency and compressibility. Second, early works require complex mechanisms to locate the start of a block. Third, they also suffer from the "expanding-write" problem, where a write to a block may result in lower compressibility. This causes the new version of the block to occupy more space than the old version of the block. A number of compressed blocks then have to be moved to make room for this larger block. Both of these problems were mitigated by the LCP mechanism [18] that reduces complexity for the common case.

To further reduce the complexity of compression for DDR3 memory, Shafiee et al. [21] abandon the goal of higher memory capacity with compression. In their MemZip approach, every block has the same starting address as in a baseline uncompressed system. But instead of occupying the next 72 bytes as in an uncompressed baseline, the block occupies a subset of those next 72 bytes. With support from rank-subsetting [24], [4], MemZip can issue fine-granularity reads and writes for compressed blocks. This reduces energy and bandwidth demand. When applied to NVMs, this also reduces wearout. However, like prior work, MemZip requires separate metadata storage to track the compressibility of each block. The management of this metadata in main memory requires OS involvement. One (positive) side-effect of such separate metadata is that a single read of metadata from memory fetches information for hundreds of cache lines. This can then be leveraged to efficiently fetch compressed versions of neighboring blocks (assuming spatial locality).

B. Baseline Compressed Memory System

We build on the MemZip approach to construct a baseline system for this work. We are targeting future specialized systems that are willing to deviate from the DDR protocol. We continue to use commodity DDR-compatible memory chips, but the memory controller, channel, and DIMM are customized to exploit compression (an approach similar to that of the customized memory system architecture in IBM's Power8 systems [10]). Our proposal and design are agnostic to the memory technology; as described shortly, the proposals have a higher impact for NVMs.

In our baseline system (example rank shown in Figure 1), a memory rank is partitioned into smaller sub-ranks. In Figure 1, a rank composed of 9 x8 memory chips is partitioned into 9 sub-ranks. The chips continue to share a single command/address bus. While all chips in a conventional rank share a single chip-select line, each chip in this sub-ranked design has its own chip-select line. This is the primary overhead and change to the conventional DDR protocol. With this change in place, the memory controller can issue commands just as usual, but the commands are restricted to a subset of a rank. By creating sub-ranks, we enable fine granularity reads and writes. In the example in Figure 1, we can read/write blocks at 8-byte granularity. As identified in MemZip, there are two primary ways to map a block to chips in a rank. In *Vertical Interleaving*, the block is placed entirely in one sub-rank. A single Column-Read command to that sub-rank retrieves 8 bytes of data. The number of Column-Read commands required to retrieve a compressed cache line depends on the compressibility of that block. In the worst case, an uncompressed 72-byte cache line would require 9 sequential Column-Read commands to that rank. In *Horizontal Interleaving*, the block is scattered across multiple sub-ranks and 8-byte words are fetched in parallel from all involved sub-ranks. Again, the number of involved sub-ranks is a function of the compressibility of the block.

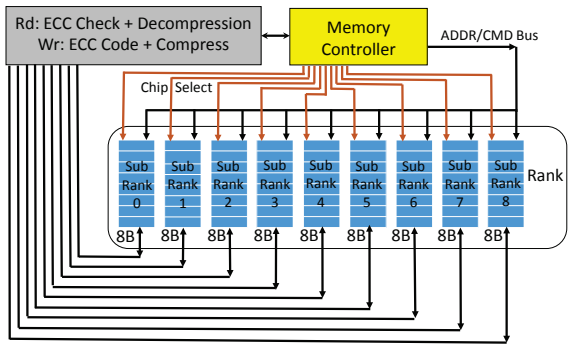

Fig. 1. Baseline memory system with sub-ranking.

We first compare the merits of Vertical and Horizontal Interleaving, while assuming an idealized compressed memory system with no OS involvement. Since compression metadata is embedded in the block itself, the size of the cache line is not known before the read is performed.

With Vertical Interleaving, one cache line is fetched from a single sub-rank, and multiple cache lines can be fetched in parallel from different sub-ranks. We'll make the optimistic assumption that the first word contains enough information to decipher the size of the block. Accordingly, additional Column-Reads are issued until the entire block has been fetched. By fetching exactly the required data, Vertical Interleaving can enjoy the low energy, low bandwidth, and high parallelism enabled by compression. A similar benefit is also experienced when handling writes. As seen in Figure 2, benchmarks LU, $GemsFDTD$, $zeusmp$ exhibit performance improvements in a DRAM system with vertical interleaving.

However, Vertical Interleaving suffers from a significant drawback – longer latency to fetch an entire cache line. In a DRAM system, or an NVM system that supports row buffers, the increase in latency is modest. Since the multiple sequential reads can be pipelined row buffer hits, the additional latency is at most 32 memory cycles. This has a noticeable impact on workloads that are largely incompressible (workloads $libquantum$, $omnetpp$, $xalancbmk$ in Figure 2).

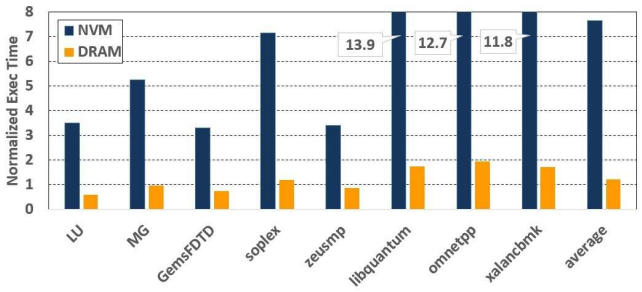

Fig. 2. Execution time for Vertical Interleaving for DRAM and NVM main memories, normalized to a baseline with no compression.

In an NVM system with no support for row buffers, e.g., a memory system based on memristor crossbars [23], multiple sequential fetches from the same chip can take significantly longer. This is because a crossbar-based chip can only read 64 bits from 64 crossbar arrays at a time – so, reading a 576-bit cache line will require 9 sequential and slow reads. Vertical Interleaving therefore performs poorly for NVM systems for all benchmarks (Figure 2). This is unlike a memory with a row buffer that can read thousands of bits in parallel into a row buffer, and quickly send bits to the output pins in successive bursts. This is a fundamental limitation of Vertical Interleaving.

With Horizontal Interleaving, a single cache line can be fetched in a single burst. But because the size of the cache line is not known beforehand, we have to conservatively fetch data from all chips in the rank. This implies that reads do not enjoy any of the benefits (low bandwidth, low energy, and high parallelism) of compression. Writes can be made more efficient because the size of a compressed block is known before performing a write. We assume a default mapping policy where the first 8 bytes of a block are placed in chip 0, next 8 in chip 1, and so on. We assume that the first 8 bytes represent the ECC/metadata for the cache line. Therefore, all writes, regardless of compressibility, involve chip 0. As a result, while compression helps reduce the energy overhead of writes, it does little to boost parallelism. Therefore, Horizontal Interleaving has the same performance as the baseline uncompressed memory system. However, we believe that the above limitations of Horizontal Interleaving can be overcome with smart prediction and mapping policies. Therefore, the goal here is to start with a Horizontal Interleaved design and augment it so it is better than the baseline and the Vertical Interleaved design in every workload, in terms of performance and energy.

III. Proposals

As described in Section II-B, without loss of generality, we are assuming a baseline memory system where a rank has been partitioned into 9 sub-ranks, and a block is placed in the rank with Horizontal Interleaving. Writes are performed on a subset of sub-ranks in parallel. Reads are performed across all sub-ranks in parallel.

For all cache lines, we assume a naive mapping policy that places the first 8 bytes of a cache line in sub-rank 0, the next 8 bytes in sub-rank 1, and so on. Later, we introduce other mapping policies. Recall that the first 8 bytes of the 72-byte data packet represent the ECC/metadata for the 64-byte cache line.

A. Managing Compression Metadata

The first step is to define a coding mechanism that can use 8 bytes of a 72-byte word to perform error correction and store compression metadata. We consider three possible approaches for constructing the codes in the 8-byte ECC/metadata field.

Approach 1: Modified Hamming Codes

There are many ways to construct ECC codes. For example, with BCH codes, a 61-bit code can protect a data field of up to 1023 bits from up to 6 errors [12]. This means that a 512-bit cache line can be protected with a 61-bit BCH code, and have 3 spare bits to track compression metadata. However, this approach cannot handle the common case of a pin failure that can cause up to eight errors in a cache line access. We therefore do not leverage BCH codes in this work.

Instead, we consider Hamming codes that are very popular in DRAM memories [8], [9]. In several commercial systems, an 8-bit Hamming SECDED code can be added to every 64-bit data transfer. This code is widely used because it can protect against single-bit random soft errors in every 72-bit field, as well as hard errors that impact one bit in every bus transfer (e.g., a pin failure). More generally speaking, m code bits can protect k information bits, where $2^{m-1} \geq m + k$. With an 8-bit code, we can protect (single error correct, double error detect) data fields as large as 120 bits; with a 7-bit code, we can protect data fields as large as 57 bits.

To make the ECC code more space-efficient, we can try to use the 8-bit code to protect data fields larger than 64 bits. But then, a single pin failure (a common error scenario) would go uncorrected. Instead, if we try to use the 8-bit code to protect data fields smaller than 64 bits, the data+code would not fit in a 576-bit transfer. So there is no way to modify the 8-bit Hamming SECDED code to be more space-efficient and include compression metadata in a 576-bit transfer.

Instead, if we use a 7-bit Hamming SECDED code to protect a 57-bit data field, we can spare a single bit in a 576-bit data transfer to store compression metadata. Nine 7-bit codes can be used to protect nine 57-bit data fields (where the data field is comprised of up to 512 bits of data and another bit to indicate if the block is compressed or not). If the block is compressed to a size of 455 bits or less (as is the case for blocks compressed with BΔI), we only need eight 7-bit codes to protect eight 57-bit data fields – this frees up an extra 7 bits in the 64-bit ECC/metadata field to store BΔI metadata.

When encoding the block, unused bits are treated as zeroes. Similarly, when decoding the block, since compression metadata is known (by inspecting the bits read from the 64-bit ECC/metadata field), the unused bits are treated as zeroes.

This new coding technique has higher resilience than the baseline. While the baseline memory system can protect from a single error in every 72-bit field, the new code can protect from a single error in every 64-bit field.

The 7-bit Hamming SECDED code is therefore an effective way to provide error correction support that is at least as strong as the baseline, and maintain compression metadata in the block itself.

Approach 2: Bamboo ECC

A recent paper [11] introduced the concept of Bamboo ECC, that uses Reed-Solomon (RS) codes. The data emerging from

a single DRAM pin in a cache line burst is treated as a single 8-bit symbol. A large number of data symbols can be protected from t symbol failures with $2t$ additional symbols. Therefore, with eight additional pins (symbols) in a 72-bit channel, an RS code can be constructed to handle four pin (symbol) failures. As shown by Kim et al. [11], such a Bamboo ECC code is very strong and can comfortably handle most common memory error patterns. Therefore, of the eight additional pins in a 72-bit channel, six can be allocated for Bamboo ECC codes, and two can be allocated for compression metadata. This guarantees protection from errors in up to three pins or symbols. The penalty imposed by tracking compression metadata in the block itself is that scenarios with four symbol errors will now go uncorrected. This already uncommon scenario can be further alleviated by using some of the 16 compression metadata bits to construct additional error detection codes, or by employing a background scrubber to prevent accumulation of errors.

Summary

The above examples show how compression metadata can be integrated with ECC codes within a 64-bit field. In some of these examples (Hamming or BCH codes), the integration does not introduce any additional penalties. In other examples (Bamboo ECC), there is a slight drop in (already strong) error coverage. With the above codes, writes are performed to a subset of memory chips, and the codes are constructed with the assumption that the data in unused chips are zeroes. Reads can be performed with a subset of memory chips, and compression metadata indicates the data that must be zeroed while verifying the ECC codes.

B. Compressibility Prediction for Reads

With the above encoding of compression metadata in the block itself, OS involvement is eliminated. Compression and ECC are entirely handled by the memory controller. Writes are performed on a subset of chips in a rank. However, reads are problematic because the size of the block is encoded in the block itself. Therefore, the read has to either be performed in two phases (read the metadata from the 0th chip, then read data from the appropriate subset of chips) or the read has to conservatively read data from all 9 chips in parallel. The former introduces a significant penalty in read latency and the latter fails to exploit the benefits of compression during reads.

To address this problem, we introduce a prediction mechanism to guess the number of chips required for every cache line read. If the prediction is accurate, we save energy and enable multiple parallel cache line fetches from a single rank. If the prediction is a conservative over-estimate, we waste energy and reduce the parallelism in the system. If the prediction is an under-estimate, we increase read latency by requiring multiple sequential data fetches for a single cache line.

We consider two prediction approaches. The first is PC-based, where the PC of the load instruction serves as the index into a predictor table. This assumes that a load tends to access the same type of data record, with relatively uniform compressibility. The second is page-based, where the physical page number serves as the index into a predictor table. This assumes that the data records in a single page are of a similar type and have uniform compressibility.

In both cases, as shown in Figure 3, each entry in the predictor table is comprised of saturating counters for each

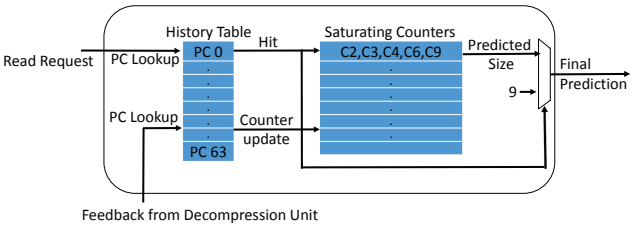

Fig. 3. Program Counter based predictor

candidate block size, i.e., up to 9 saturating counters per entry. The BΔI implementation in this study only requires five saturating counters per entry because at 8-byte granularity, only five block sizes are supported. Once the size of a block is determined, the prediction table is updated by incrementing a single saturating counter corresponding to that block size and that PC or page. In case of erroneous predictions, the saturating counter that caused the wrong prediction in that entry is decremented. On a look-up, the highest-valued saturating counter indicates the predicted size of the block. In case of a tie, we conservatively predict the larger block size. The tables are tagged, i.e., each entry stores its corresponding PC or page number. We also assume that the table is fully-associative and managed with an LRU replacement policy. In our evaluations, we observed that 2-bit saturating counters and 64-entry tables were sufficient for high accuracy. The page-based predictor can even be merged with the TLB structures.

C. Data Mapping Policies

One of the benefits of dealing with compressed blocks is that reading/writing a compressed block only involves a subset of the rank (chips and channel). This means that a rank can support multiple concurrent reads or writes as long as they are to non-intersecting subsets of chips in the rank. In our default mapping policy, every cache line starts at chip 0, and progressively uses chips 1, 2, ..., 8, depending on the size of the block. Such a naive mapping policy is clearly sub-optimal because two accesses to the same rank can never proceed in parallel.

We consider an obvious alternative mapping policy to promote parallelism within a rank. This policy reverses the mapping for adjacent cache lines. So, odd-numbered cache lines are mapped to chips 0, 1, 2, ..., 8 (in that order), while even-numbered lines are mapped to chips 8, 7, 6, ..., 0 (in that order). We refer to this as the *simple mapping policy*. While it increases the likelihood that two adjacent compressed cache lines can be fetched together, it is less effective for more general access patterns, and it leads to non-uniform activities in chips.

We therefore construct a mapping policy that maximally promotes parallelism and uniform activities in each chip in a rank. We first define two cache lines as being $m - n$ compatible if the two cache lines use non-intersecting sets of chips, assuming the first cache line uses m chips after compression and the second uses n chips after compression. For example, in the simple mapping policy, two adjacent cache lines are always 4-5 compatible.

We attempt to construct a mapping for every group of eight consecutive cache lines that somewhat preserves the compatibility properties of the simple mapping policy, but allows

some perturbation so that each chip has somewhat uniform activity. We therefore come up with mappings that permute the order of the chips as long as the following properties are preserved: (i) consecutive cache lines are 1-7, 3-5, 5-3, and 7-1 compatible, (ii) the last cache line in the group is 1-7, 3-5, 5-3, and 7-1 compatible with the first cache line of the next group. By enumerating a number of permutations, we considered a vast design space of possible mappings for a group of eight adjacent cache lines. We selected the best mapping based on the afforded parallelism and activity uniformity for a few synthetic compression patterns. This mapping policy, referred to as the *permuted mapping policy*, is indicated below in Figure 4. The memory controller statically enforces this mapping policy for every group of eight consecutive cache lines. To promote long-term uniform wearout, this mapping policy can be periodically rotated.

Cache line	Chips used by this cache line (in order)
0	0 1 2 3 4 5 6 7 8
1	7 8 5 3 6 4 2 1 0
2	0 4 1 3 2 5 8 7 6
3	6 5 8 7 2 1 3 0 4
4	4 1 3 2 0 8 7 6 5
5	5 7 6 8 0 1 2 4 3
6	3 1 2 0 4 6 7 5 8
7	8 6 7 5 4 1 2 0 3

Fig. 4. Permuted mapping policy for a group of eight consecutive cache lines.

D. Supporting Scheduling Policies

Each incoming memory transaction request (a macro-request) is partitioned into up to nine micro-requests (one per sub-rank). The micro-requests corresponding to a macro-request are handled together by the memory controller. The number of micro-requests is based on the compressibility prediction in case of reads, or based on the compression logic in case of writes. In case of mis-predictions by our compressibility predictor, additional micro-requests may be inserted into the queue.

While a compressed memory system should typically yield performance improvements, we observed slight performance degradations in some cases. This was because the above scheduling policy was sub-optimal when handling row buffer hits or compressibility mis-predictions. Consider the following example where two read requests are to the same DRAM row, i.e., in the baseline memory system, the second read is a row buffer hit. In our compressed memory system, assuming that the first cache line is compressed to use 3 chips, the first request will involve Activates and Column-Reads to only 3 chips. If the second cache line is compressed to use 5 chips, it can no longer enjoy a row buffer hit; it would have to issue Activates to 2 of the chips, followed by Column-Reads to 5 chips.

To address this problem, Activates and Precharges are performed on all nine chips in the rank, while Column-Reads and Column-Writes are performed on rank sub-sets. Figure 5 shows that such full-activates are slightly better than partial-activates in all but one benchmark.

IV. Methodology

For our simulations, we use the Simics full-system simulator [3]. We model eight out-of-order cores with 1 memory channel and 4 ranks. Detailed simulation parameters are listed in Table I. We interface a detailed memory timing model (USIMM [6]) to Simics.

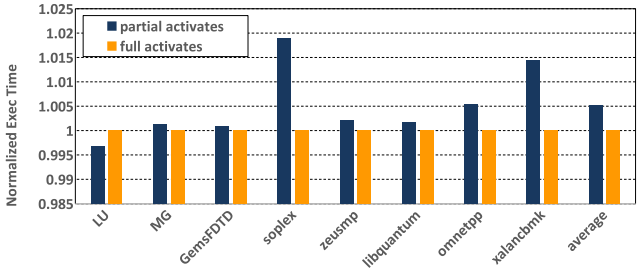

Fig. 5. Performance of a compressed DRAM system with partial and full Activates and Precharges.

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	8-core, 3.2 GHz
Re-Order-Buffer	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache	8MB(DRAM) 16MB(NVM) /64B/8-way, shared, 10-cycle
Coherence Protocol	Snooping MESI
DRAM Parameters	
DDR3	MT41J256M4 DDR3-1600 [13],
Baseline DRAM Configuration	1 72-bit Channel 4 DIMM/Channel (unbuffered, ECC) 1 Ranks/DIMM, 9 devices/Rank
DRAM Bus Frequency	800MHz
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size High/Low Watermarks	48 entries per channel 40/20
NVM Parameters	
Baseline NVM Configuration	1 72-bit Channel 4 DIMM/Channel (unbuffered, ECC) 1 Ranks/DIMM, 9 devices/Rank
NVM Bus Frequency	400MHz
tRCD/tWR/tCL	58/237/48 (CPU cycles)
NVM Read Queue	48 entries per channel
NVM Write Queue Size High/Low Watermarks	48 entries per channel 40/20

TABLE I
Simulator parameters.

Memory Power	
VDD/IDD0/IDD2P0/	1.5(V)/55(mA)/16(mA)
IDD2P1/IDD2N/IDD3P/	32(mA)/28(mA)/38(mA)
IDD3N/IDD4R/IDD4W/IDD5	38(mA)/157(mA)/128(mA)/155(mA)
Compressor/Decompressor	
Com. Power	15.08 (mW)
decom. power/cycle fre.	17.5(mW)/1 GHz

TABLE II
Power parameters.

As a sensitivity study, we also model behavior when the DRAM system is replaced by an NVM system (NVM parameters are also summarized in Table I). To model read/write timing we use the parameters from the work of Xu et al. [23].

Note that our proposals are especially compelling for NVM systems because of the positive impact on endurance and the poor behavior of Vertical Interleaving in NVMs.

For our workloads, we use eight memory-intensive programs from SPEC2k6 (libquantum, onmetpp, xalacbmk, GemsFDTD, soplex, zeusmp) and NAS Parallel Benchmarks (LU, MG). The SPEC2k6 programs are run in multi-programmed mode, with 8 copies of the same program, while the NAS benchmarks are run in multi-thread mode. We ignore the statistics for the first 100K DRAM reads to account for cache warmup effects. All our simulations are run until 1 million DRAM (or NVM) reads are completed.

To calculate the energy consumed by our memory system, we use Micron's power calculator [2]. The power model for the compression and decompression logic are based on those in MemZip. These power parameters are summarized in Table II. Similar to the methodology used by MemZip [21], system power is calculated by assuming that memory consumes 25% of baseline system power. For NVM read/write energy, we use the per-bit energy numbers of Niu et al. [15].

Figure 6 shows the compression histogram of all workloads (for BΔI). It shows the percentage of cache lines that can be stored in 2, 3, 4, 6, and 9 chips. Out of the 8 workloads that we use, 5 exhibit high compressibility (first 5 bars), whereas 3 don't (next 3 bars).

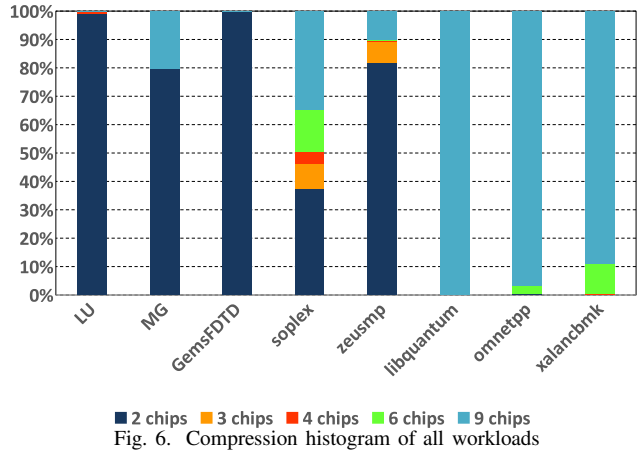

Fig. 6. Compression histogram of all workloads

V. Results

A. DRAM Results

We first evaluate the impact of our proposals on a DRAM-based memory system. Figure 7 shows normalized execution time for a number of configurations. The left bar is an oracular system that knows the compressibility of every block, but uses the default mapping policy. As a result, it is unable to issue requests to multiple cache lines in parallel. Its performance therefore represents that of a baseline uncompressed memory system. The second bar is the oracular system, but with the permuted map policy introduced in Section III-C. It promotes high parallelism and therefore represents an upper-bound in terms of performance.

The next three bars in Figure 7 represent behavior for different mapping policies (default, simple, and permuted), while assuming a PC-based compressibility predictor. The last three bars do the same for a page-based compressibility predictor.

We observe that the mapping policies have a significant impact on performance. Moving from the default to the simple map yields a performance improvement of 6% on average for the page-based predictor. Moving from simple to permuted yields an additional 1% improvement on average. When considering the workloads on the left of the graph (with more than 65% of blocks being compressible), the average improvement with the page-based predictor and permuted mapping is 11%. Both prediction-based approaches are equally effective at nearly matching the performance of the oracular scheme. The PC-based and page-based predictors are within 1% of the oracular scheme on average. Libquantum exhibits small performance degradations because of a slight decrease in row buffer hit rates.

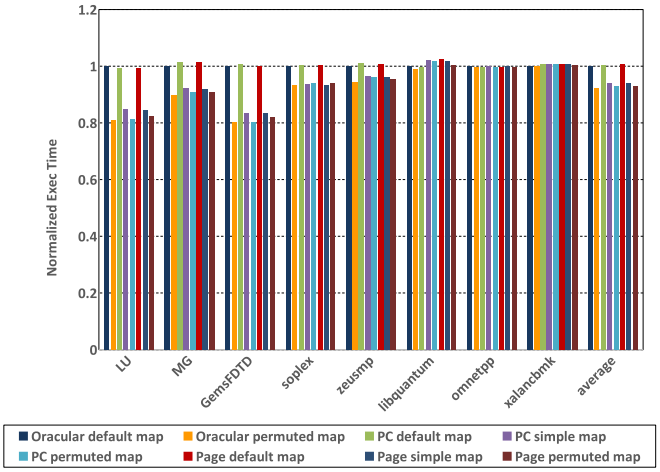

Fig. 7. Performance of Oracular, Page, and PC based predictors in a DRAM system with different mapping schemes.

To better understand the behavior of the predictors, Figures 8 and 9 characterize the prediction accuracy of the PC and Page based predictors with permuted mapping. Their average accuracies are 93% and 97% respectively. Several benchmarks have near perfect prediction accuracies. The mispredictions are split evenly as over-estimations (bad for energy) and under-estimations (bad for performance).

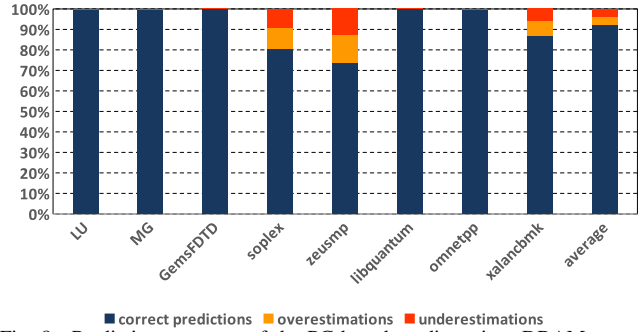

Fig. 8. Prediction accuracy of the PC based predictor in a DRAM system.

Figure 10 shows the system energy consumed by our predictor based designs, normalized to a baseline with uncompressed memory. These numbers closely follow the performance trends. The energy savings are from accessing fewer chips per memory request and reduced execution time. The predictor-based schemes with permuted mapping yield average energy reduction of 12%, and this is within 2% of the energy

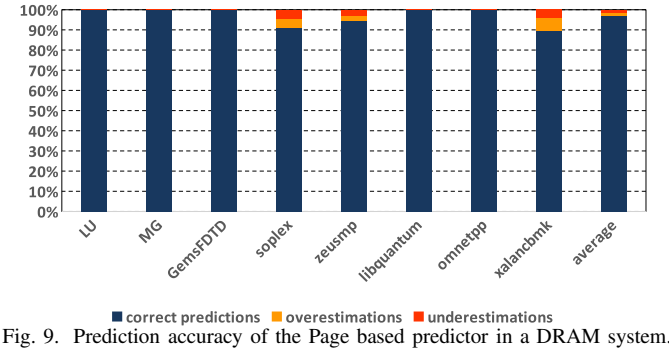

Fig. 9. Prediction accuracy of the Page based predictor in a DRAM system.

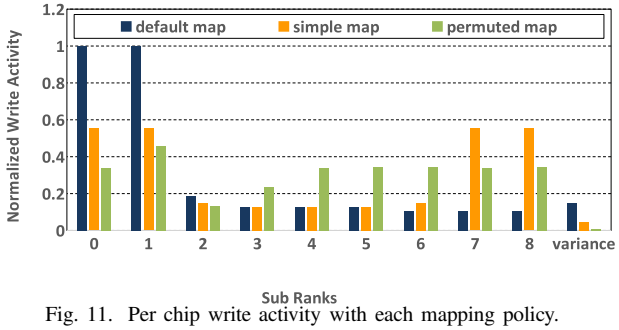

Fig. 11. Per chip write activity with each mapping policy.

saved by an oracular scheme.

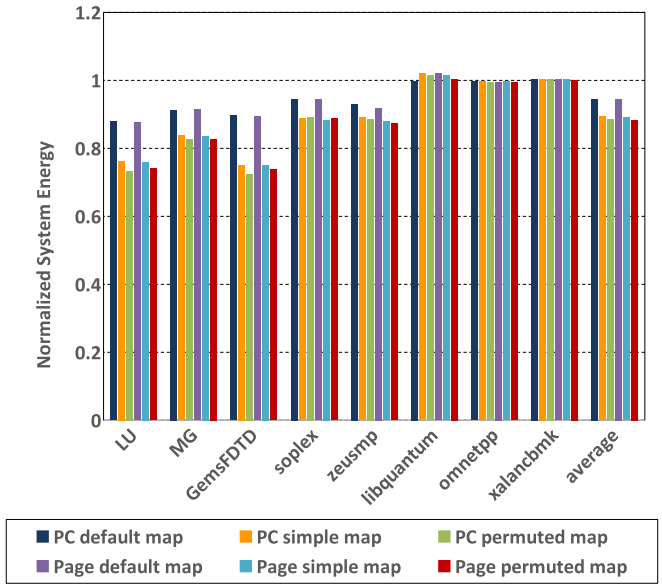

Fig. 10. System energy consumption of Page and PC based predictors in a DRAM system.

In the results so far, the difference in performance and energy between the simple and permuted mapping schemes has been relatively small. The clear benefit of the permuted mapping scheme is evident when plotting the distribution of write activity across the chips in a rank. Figure 11 shows the fraction of memory requests that touch each chip in a rank for the default, simple, and permuted mapping policies. The last bar in the graph plots the variance in these three distributions. Clearly, the default mapping involves the first two chips in every access, while the permuted mapping is best at evenly distributing accesses across all chips. The variance of default, simple, and permuted mapping are 0.15, 0.05, and 0.008 respectively. This uniformity of accesses is useful in three ways: it increases the probability of scheduling two requests together, it maximizes endurance for NVM memories, and it reduces the likelihood of thermal emergencies.

B. NVM Results

We next examine the performance impact of our proposals on an NVM based memory system. The prediction accuracies and per-chip write activities are similar to those in the DRAM system since they are not affected by DRAM timing parameters. As shown in Figure 12, the page-based predictor with the simple mapping policy shows a performance benefit of 7.2%. This increases by an additional 1% in case of permuted mapping. For workloads that have high compressibility, the page-based predictor with permuted mapping shows an average performance improvement of 13.5%. In terms of system energy consumption, the best evaluated configuration yields an average energy reduction of 14%.

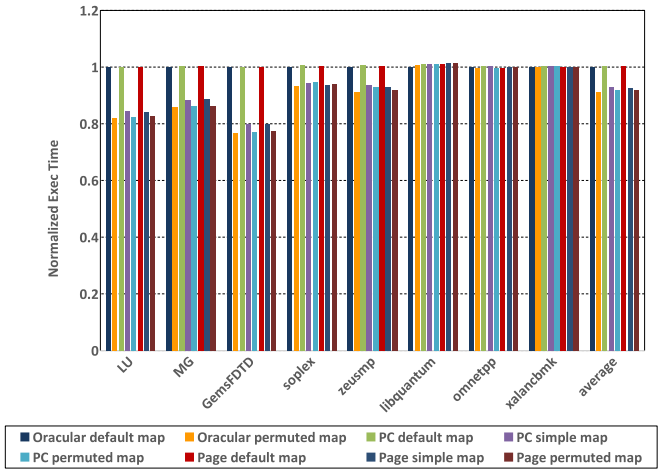

Fig. 12. Performance of Oracular, Page, and PC based predictors in an NVM system with different mapping schemes.

VI. Related Work

Section II-A has already described prior approaches in compressed memory systems. Here, we describe a few prior works that are related to our approaches of ECC/metadata integration and prediction of compressibility.

A recent paper by Palframan et al. [16] modifies data layout in non-ECC DIMMs to support ECC. It takes advantage of compression to create room in a 64-byte block to store error correction codes. The 64-byte block is partitioned into four regions, each with an ECC code. If multiple ECC codes flag errors, then with a very high probability, the block is uncompressed and stored in its raw 64-byte form. With such a data layout and code, separate metadata is not required to keep track of which blocks are compressed and which are not. However, for blocks that are uncompressed, ECC must be maintained in a separate region of memory. This requires multiple memory accesses to process one transaction, and involvement from the OS.

The LCP implementation of Pekhimenko et al. [18] has a default compressibility for every block in a page. This is tracked in page table entries and is used to efficiently handle the common case for that page. This is similar to our approach of tracking compressibility on a per-page basis with our predictor. But the applications of this compressibility prediction are very different in LCP and in our design. For example, our design is using prediction to avoid OS involvement, while LCP relies on the OS to maintain this compressibility information. Pekhimenko et al. [17] also use compressibility as a prediction of future reuse of a block, applied to cache replacement policies.

VII. Conclusions

In this work, we take the next step in memory compression by implementing it entirely in hardware with no OS involvement. Three enabling technologies are proposed: (i) Modified Hamming and Bamboo codes that integrate ECC and compression metadata in a single 64-bit field per cache line. (ii) PC and page based prediction techniques that allow read operations to be performed more efficiently. (iii) Mapping policies that evenly distribute activity across a rank to improve performance, endurance, and power/thermal profiles.

Our analysis and results show that the modified Hamming code introduces no penalties. The PC and page based predictions are both highly accurate. The page-based prediction turns out to be slightly more effective (an accuracy of 97%), and can be integrated into existing TLB hardware. We observe that the simple and permuted mappings have very similar performance, but the permuted mapping offers much lower variance. The combined techniques offer a performance improvement of 7% in DRAM, and 8% in NVM, because of the higher parallelism they enable. By limiting the number of chips accessed per memory request, and by improving performance, they also yield system energy reductions of 12% in DRAM, and 14% in NVM. The variance in write activity is also reduced from 0.15 in the default mapping to 0.008 in the permuted mapping.

References

[1] "EMU Technology," http://www.emutechnology.com/.

[2] "Micron System Power Calculator," http://www.micron.com/products/support/power-calc.

[3] "Wind River Simics Full System Simulator," 2007, http://www.windriver.com/products/simics/.

[4] J. Ahn, N. Jouppi, and R. S. Schreiber, "Future Scaling of Processor-Memory Interfaces," in *Proceedings of SC*, 2009.

[5] R. Balasubramonian, J. Chang, T. Manning, J. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insight from a Workshop at MICRO-46," in *IEEE Micro's Special Issue on Big Data*, 2014.

[6] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.

[7] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," in *Proceedings of ISCA*, 2005.

[8] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, 1950.

[9] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, 1970.

[10] Jeffrey Stuecheli, "Power Technology for a Smarter Future," 2014.

[11] J. Kim, M. Sullivan, and M. Erez, " Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory ," in *the Proceedings of HPCA-15*, February 2015.

[12] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System Implications of Memory Reliability in Exascale Computing," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 46:1–46:12.

[13] "Micron DDR3 SDRAM Part MT41J256M8," Micron Technology Inc., 2006.

[14] R. Nair *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," in *IBM Journal of R&D (to appear)*, 2014.

[15] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, , and Y. Xie, "Design of Cross-point Metal-oxide ReRAM Emphasizing Reliability and Cost," in *Proceedings of ICCAD*, 2013.

[16] D. Palframan, N. Kim, and M. Lipasti, "COP: To Compress and Protect Main Memory," in *Proceedings of ISCA*, 2015.

[17] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse," in *Proceedings of HPCA*, 2015.

[18] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *Proceedings of MICRO*, 2013.

[19] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in *Proceedings of PACT*, 2012.

[20] V. Sathish, M. j. Schulte, and N. S. Kim, "Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads," in *Proceeding of PACT*, 2012.

[21] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploiting Unconventional Benefits from Memory Compression," in *Proceedings of HPCA*, 2014.

[22] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland, "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45(2), 2001.

[23] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the Challenges of Crossbar Resistive Memory Architectures," in *Proceedings of HPCA-21*, 2015.

[24] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-Rank: Adaptive DRAM Architecture For Improving Memory Power Efficiency," in *Proceedings of MICRO*, 2008.