MemZip: Exploring Unconventional Benefits from Memory Compression *

Ali Shafiee Meysam Taassori Rajeev Balasubramonian Al Davis
University of Utah
{shafiee,taassori,rajeev,ald}@cs.utah.edu

Abstract

Memory compression has been proposed and deployed in the past to grow the capacity of a memory system and reduce page fault rates. Compression also has secondary benefits: it can reduce energy and bandwidth demands. However, most prior mechanisms have been designed to focus on the capacity metric and few prior works have attempted to explicitly reduce energy or bandwidth. Further, mechanisms that focus on the capacity metric also require complex logic to locate the requested data in memory. In this paper, we design a highly simple compressed memory architecture that does not target the capacity metric. Instead, it focuses on complexity, energy, bandwidth, and reliability. It relies on rank subsetting and a careful placement of compressed data and metadata to achieve these benefits. Further, the space made available via compression is used to boost other metrics – the space can be used to implement stronger error correction codes or energy-efficient data encodings. The best performing MemZip configuration yields a 45% performance improvement and 57% memory energy reduction, compared to an uncompressed non-sub-ranked baseline. Another energy-optimized configuration yields a 29.8% performance improvement and a 79% memory energy reduction, relative to the same baseline.

1 Introduction

Many system components (cache, memory, disk) are capacity-constrained. It is therefore natural to consider data compression techniques to boost the effective capacities of these structures. By storing data in compressed formats, there is an additional encoding/decoding delay on every read/write, but it reduces the number of accesses to the next level of the memory hierarchy. Many papers have shown the effectivness of data compression for caches [27], memory [4], and disk [39]. This paper focuses on compression applied to main memory. The work here is orthogonal to the compression algorithm in use – our focus is on the organization of compressed data within the memory system.

Most prior works focus on the higher effective capacity made possible by compression. When applied to the main memory system, it reduces the number of expensive page faults. There are other possible secondary benefits from compression: lower energy per access and lower bandwidth demand, although, these have not been explicitly targeted by most prior work. A couple of papers have attempted to reduce bandwidth and channel energy in a GPU/GDDR5 system [28], and in systems with off-chip memory controllers [31].

There are three primary hardware-based memory compression architectures for chip multiprocessors (CMPs) and DDR3 memory in recent literature. The IBM MXT technology [4] uses a memory look-up to find a pointer to a compressed block. Thus, every access requires two memory fetches. The work of Ekman and Stenstrom [16] stores metadata with every TLB entry so that the start of a cache line can be computed. The LCP architecture [26] optimizes the common case. For cache lines that can be compressed within a given size, a pointer to the compressed block is trivially computed. But a cache line that cannot be compressed within the specified size will require three memory accesses in the worst case. More details about these schemes are provided in Section 2.

All of these prior designs attempt to first increase effective capacity with compression. Since DDR3 memory chips must return data in bursts of eight, every request continues to return 64-byte blocks from memory. This 64-byte block may contain multiple compressed cache lines. There is an energy and bandwidth advantage only if applications exhibit spatial locality and indeed require the many cache lines contained in one 64-byte transfer (LCP [26] introduces an optimization to exploit this property). These prior designs also require some logic to locate and fetch a cache line because compressed blocks get re-organized in the physical memory space. Further, when a block is written to, the new block may have a different compressed size. This requires a potential re-organization of data and multiple cache line copies within memory.

Instead, in this work, we design a new compression architecture that is designed explicitly for energy- and bandwidth-efficient operation. The performance improvement comes from bandwidth efficiency, not from a reduced page fault rate. It is therefore useful even when applications don't stress the memory capacity or don't exhibit spatial locality. A similar approach was also employed for

*This work was supported in parts by NSF grant CNS-1302663, HP Labs, IBM, and Samsung. We also thank the anonymous reviewers for many useful suggestions.

GPU and GDDR5 architectures by Sathish et al. [28]. In this paper, we first show how bandwidth and energy can be saved with compression in a DDR3 memory system. Further, the extra space made available by compression is used to improve reliability and energy, by introducing ECC and energy-efficient encoding at no extra cost.

Our architecture is based on a memory system that uses rank subsetting. A compressed cache line is fetched from a single rank subset and the burst length (a multiple of eight) is a function of the compression factor. For example, an uncompressed cache line can arrive with a burst length of 64, while a highly compressed cache line can arrive with a burst length of eight. Thus, the energy and bandwidth demand of a cache line transfer is dictated entirely by its compression factor, while also complying with DDR3 standards. To reduce look-up complexity, we assume that the starting location of every block is the same as in an uncompressed baseline memory system. So the new memory system stores as many blocks as the baseline and offers no capacity advantage. Some of the spare space in the memory system can now be used to save ECC bandwidth or to further reduce energy. Energy is reduced by storing data in an encoded format that reduces the number of data bus transitions. This format is referred to as the *Data Bus Inversion (DBI)* format [30] and requires a few more metadata bits that are placed in the spare space of each block.

Thus, unlike most prior work in DDR3 memory compression, this work ignores the conventional figure of merit (capacity), and targets the unconventional metrics: complexity, energy, bandwidth, reliability.

The best performing MemZip configuration yields a 45% performance improvement and 57% memory energy reduction, compared to an uncompressed non-sub-ranked baseline. Another energy-optimized configuration yields a 29.8% performance improvement and a 79% memory energy reduction, relative to the same baseline.

2 Background

2.1 DRAM Memory Basics

A high-performance processor typically implements up to four memory controllers. Each memory controller handles a 64-bit DDR3 data channel with an address/command bus with a width in the neighborhood of 23 bits. The channel supports 1-4 ranks. A rank is a collection of DRAM chips that together feed the 64-bit data bus, i.e., in the common case, a rank may contain 8 x8 chips, or 4 x16 chips, or 16 x4 chips (xN refers to a DRAM chip with N data bits of input/output on every clock edge). DDR3 has a minimum burst length of 8, i.e., a request results in eight 64-bit data transfers on the bus. To fetch a cache line, the memory controller first issues an Activate (ACT) command, followed by a Column-Read (COL-RD). Each COL-RD results in a burst of 8 from each DRAM chip in that rank, yielding a 64-byte cache line. The ACT command brings an entire row of data (about 8 KB) into a row buffer. Adjacent cache lines in the row can be fetched with multiple COL-RDs without requiring additional ACTs. Each rank is itself partitioned into 8 banks. The 8 banks are independently controlled and have their own row buffers.

ECC-DIMMs introduce additional chips per rank to store SECDED codes. Typically, an 8-bit code is fetched along with every 64-bit data word; this introduces an energy penalty, but not a delay penalty.

2.2 Rank Subsetting

Different forms of rank subsetting have been introduced in recent years [40, 5, 33] to improve energy and performance. Rank subsetting partitions a 64-bit rank into smaller subranks. Each subrank can be independently controlled with the same single command/address bus on that channel. An ACT command only applies to a single subrank, i.e., it only causes activity in a subset of DRAM chips in the rank and limits the overfetch into the row buffer. Fetching a 64-byte cache line may require multiple COL-RDs, depending on the width of each subrank. Since the banks in each subrank can be independently controlled, the number of available banks also increases. This leads to lower queuing delays and higher data bus utilization. In short, rank subsetting lowers energy per memory access, increases the cache line transfer time, decreases data bus queuing delays, and increases command bus utilization (more COL-RDs per cache line request).

There are differing implementations of rank subsetting. The SSA design of Udipi et al. [33] designed a custom chip and interface. Ahn et al. [5] and Zheng et al. [40] remained DDR3 compliant and introduced MUXes and a buffer chip on the DIMM to activate the appropriate subrank on every command.

As an example, consider the following baseline rank that is made up of 8 x8 DRAM chips. In a 4-way subranked model, the rank and channel are partitioned into 4 subranks. Each subrank consists of 2 DRAM chips. Four COL-RDs are required to fetch an entire cache line from one subrank. 4-way rank subsetting enables a quadrupling in the number of independent banks.

Prior work has shown a performance improvement on average with rank subsetting. Section 4 reproduces some of this analysis and considers a large design space. In that section, we also propose a modification to rank subsetting that helps alleviate the data transfer time penalty, while still allowing high compressibility.

Prior work on rank subsetting also mentioned the difficulty in providing ECC support. Each subrank now needs its own extra chip to store ECC codes. Four-way rank subsetting would therefore require four extra chips instead of the one extra chip required in the baseline. Zheng et al. [40] introduced the notion of embedded-ECC to overcome this problem. The ECC for each cache line is not stored in a

separate chip, but in the same chips as the data. If we assume an 8-way sub-ranked system, every 64-byte cache line in a row is followed by eight bytes of ECC codes. The cache line is fetched with 8 consecutive COL-RDs and the ECC is fetched with a 9th sequential COL-RD. The extra COL-RD is an overhead not seen in a conventional rank with a 9th ECC chip. Embedded-ECC also makes indexing a little more complex because a row can only store 112 cache lines instead of 128. If we use 4-way and 2-way subranking, the embedded-ECC overheads grow because a single COL-RD forcibly prefetches ECC codes for 2 and 4 consecutive cache lines, respectively. As we show later, MemZip is able to support embedded-ECC without suffering from the above two drawbacks (extra COL-RDs and forced prefetch) in the common case.

2.3 Compression Algorithms

The MemZip architecture can work with any compression algorithm. In this work, we focus on two compression schemes that are easy to implement in hardware: base-delta-immediate (BΔI) [27] and frequent pattern compression (FPC) [7].

BΔI relies on the observation that words in a line only differ slightly. Hence, the words are better represented as their distance from a given base. This is best explained with an example. Consider a 64-byte line that is partitioned into 8 8-byte words. Let's assume that the first 8-byte word is the base. The difference between each 8-byte word and the base may be (represented in decimal): 0, 7, 23, 16, 104, 5, 213, 77. Since every one of these deltas is less than 255, they can each be represented by a single byte. The compressed version of this line would therefore be an 8-byte word, followed by 8 1-byte words, for a total capacity of 16 bytes. Each compressed line needs a few bits of header to indicate the size of the base and the size of each delta. BΔI also allows the use of two bases, one of which is the word zero. Every delta is therefore relative to the specified base or relative to zero. A bit mask is required in the header to indicate which of the two deltas is used for every word in the line.

FPC [7] relies on the fact that some word patterns occur frequently in many applications. Examples include all-zero and all-one bit strings in positive and negative integers with small absolute values. FPC divides a line into 4-byte words and keeps three encoding bits per word. In some cases, the three encoding bits are enough to represent the data, e.g., 000 represents a 4-byte string of zeroes. In some cases, the three encoding bits must be accompanied by additional data bits. For example, the 100 encoding represents a halfword padded with a zero halfword [7]. Representing such a 4-byte word would require 19 bits (3 bits of header, followed by 16 bits of the non-zero halfword). The entire 64-byte line is assembled as an initial 48-bit header indicating 3-bit encodings for each of the 16 4-byte words, followed by the additional bytes required by each encoding.

2.4 IBM MXT [4]

The IBM MXT architecture compresses 1 KB blocks. Every 1 KB block has a 16 byte metadata entry in physical memory that is indexed by the block address. If we're lucky, the compressed version of the 1 KB block may be found inside the metadata entry itself. If not, then the metadata entry has up to four pointers to 256 byte sectors. The compressed version of the 1 KB block is placed in 1-4 sectors. The physical memory is therefore allocated at the granularity of sectors. Data is fetched in 32 byte increments (this architecture is over 10 years old and pre-dates the DDR3 standard that requires a minimum burst length of eight). Most cache line requests require two memory accesses – one for the metadata and one for the cache block itself. The design is explicitly optimized to maximize effective capacity and reduce page faults. The memory access latency is much higher than that of a comparable baseline memory system. There is no explicit feature for bandwidth or energy efficiency. A write can be expensive as it requires the creation of new sectors and an update of the metadata entry.

2.5 Ekman and Stenstrom [16]

The work of Ekman and Stenstrom [16] addresses many of the weaknesses found in the MXT design. It associates the metadata for a page along with the TLB entry. This metadata tracks the size of each compressed block within the page. In parallel with the LLC look-up, the location of the cache line in memory is computed based on the information in the metadata. This hides the latency for address calculation, but increases the energy overhead. On every write, if the new compressed block has a very different size from the old compressed block, the blocks may have to be re-organized, requiring a page table update and even requiring a copy to a new page in the worst case. The proposal has no explicit feature for bandwidth or energy efficiency.

2.6 LCP [26]

Very recently, Pekhimenko et al. [26] proposed the LCP architecture. Each block is expected to be compressed within a fixed size field. Hence, the expected start address of a block is easy to compute. But if the block cannot be compressed within the fixed field size, it is placed in an exception region within the same page. Metadata in that page helps locate this block. In the worst case, three memory look-ups are required to fetch the requested block. The authors introduce a metadata cache at the memory controller. On metadata cache hits, the block can be fetched with a single memory access. The authors introduce a bandwidth optimization for compression. When a 64 byte block is fetched, the metadata is used to determine if multiple valid cache lines exist in this block. Any valid additional cache blocks are placed in cache, especially if recommended by a stride prefetcher. If an application exhibits spatial locality,

compression helps reduce the fetch energy and bandwidth demand for multiple blocks.

3 MemZip Architecture

3.1 The MemZip Approach

In the previous section, we discussed three competing architectures that all try to optimize effective capacity with compression. All of them try to tightly pack compressed blocks within the physical memory. This leads to overheads when accessing data. Metadata has to be consulted to locate the start of the block. The LCP architecture is most adept at quickly finding the block if there is a metadata cache hit. If there is a metadata cache miss, three memory accesses may be required. Further, when performing a write, if the new compressed block can't be fit into the page, a new larger page must be created and the contents must be moved from the old page to the new page. All of these complications arise from the fact that blocks of different sizes are being packed into a small page. To achieve a simple design, we make no attempt to save capacity. Every block has the same starting address as a baseline uncompressed memory, so there is no complication in locating a block. If the size of the compressed block varies with each new write, the block simply takes up more or less space within its allocated space; there is never a need to copy data to make room for a large compressed block.

Commodity DDR3 DRAM chips are required to provide data with a minimum burst length of eight. For a standard 64-bit DDR3 bus, we are therefore required to fetch a minimum of 64 bytes on every memory access. This limits the ability of prior work to save bandwidth and energy – even though a compressed block may only occupy (say) 32 bytes, one is forced to fetch at least 64 bytes. If the extra 32 bytes contain an entire compressed cache line and if this additional line is accessed soon after, then the prefetch enabled by compression leads to energy and bandwidth saving. Instead of relying on this accidental saving, we make an explicit effort to reduce energy and bandwidth. We access metadata that tells us the exact number of bursts required to fetch the compressed cache line. The line is then transferred over exactly that burst length, thus saving bandwidth and energy that is proportional to the compression ratio. Because of the DDR3 standard, we are limited to using burst lengths that are multiples of eight. We exploit rank subsetting to transfer a cache line over (say) a narrow 8-bit bus, instead of the standard 64-bit DDR3 bus. In a burst of eight, we can receive a 64-bit compressed cache line; in a burst of 16, we can receive a 128-bit compressed cache line, and so on. In short, we control the transfer at 8-byte granularities instead of 64-byte granularities, thus fully exploiting the energy and bandwidth potential of memory compression.

If a 64-byte cache block has been compressed into a (say) 26-byte block, we have 38 more spare bytes to store other useful information pertaining to that block. If we used all of these 38 bytes, we would negate the bandwidth and energy advantage from compression. But if we used only 6 bytes, there would be no energy or bandwidth overhead for the fetch since we are required to fetch at 8-byte granularities anyway. These 6 bytes can be used to save ECC code or for DBI codes. DBI is an encoding format that reduces the energy for data transfer.

More details of the architecture will be explained next. It is worth reiterating that compression is being performed (i) to improve performance by better utilizing memory bandwidth, and (ii) to improve energy per memory access. Our specific design choices also have favorable implications on reliability and complexity.

3.2 Basic Architecture

We first describe the design of a single channel in our memory system. We use rank subsetting to split the 64-bit data channel into N narrow data channels. We will assume $N = 8$ for this discussion; our evaluation considers several rank subsetting scenarios. The 8 narrow data channels share a single address/command bus. Such rank subsetting increases the delay to transfer a single cache line, but leads to lower queuing delays because of the parallelism and better bus utilization. Rank subsetting also leads to lower energy by shrinking the sizes of activated rows.

Data can be organized across the memory system in many different ways. Typically, an entire cache line is fetched from a single channel, rank, and bank. Consecutive cache lines can be co-located in a single row of the bank to boost row buffer hit rates or in different banks, ranks, and channels to boost memory-level parallelism. The MemZip architecture can work with either address mapping policy. Prior compression schemes have to adopt the former address mapping policy to derive any bandwidth or energy advantage from spatial locality.

We are now fetching an entire 64-byte cache line over an 8-bit data bus. This data bus is fed by a single x8 DRAM chip, or 2 x4 DRAM chips, or 4 x2 DRAM chips, or 8 x1 DRAM chips. To minimize activation energy, we will assume that a rank subset is comprised of 1 x8 DRAM chip. The 64-byte cache line is placed in consecutive columns of a single bank's row buffer. It is fetched by issuing one ACT (Activate) command, followed by 8 COL-RD (Column-Read) commands in quick succession. Each COL-RD command uses a minimum burst length of eight. The entire 64-byte cache line is transferred with 64 8-bit transfers.

In the MemZip design, each 64-byte cache block is compressed to an arbitrary size. The compressed block begins at the same address location as in the baseline rank-subset architecture. If the compressed block has a size of 26 bytes, it occupies the next 208 column bits in that row. The next 304 column bits (38 bytes) in that row are left unused, i.e., the capacity saving from compression is not being exploited

yet (we will shortly explain how some of this space can be used for reliability and energy).

Metadata structures track the size of the compressed block and the number of COL-RDs required to fetch it. Each cache line requires a few-bit metadata entry. For an 8-way subranked system, the cache line may be fetched with 0 to 8 COL-RDs, requiring a 4-bit metadata entry. If the cache line is a string of zeroes, the memory access is avoided altogether (0 COL-RDs). If 8 COL-RDs are required, the block is stored in uncompressed format.

One option for metadata storage is to attach it to the page table and the TLB. An 8 KB OS page has 128 cache lines, corresponding to 512 bits of metadata. While this level of storage may be supported by a TLB, the storage requirements grow if the OS is using large pages. Since this is a common scenario, we instead use a more general scheme. The metadata information is stored in physical memory and is cached at the memory controller in a metadata cache. Since we require 4 bits of metadata for every 512-bit data line, every 128 lines of data require 1 line of metadata storage. This metadata line is organized as follows. Every 8 KB DRAM row accommodates 128 lines, of which the first 127 are data lines and the last line is the metadata information for the entire row. The 128th data line is placed in a separate "overflow" row. One overflow row is required for every 128 data rows. This organization was selected so that metadata access yields a row buffer hit in the common case. The OS must reserve every 129th page for overflow lines, i.e., MemZip metadata introduces a storage overhead of 0.8%. The memory controller requires a new (but deterministic) indexing function when fetching the 128th line in any row. If we assume an embedded-ECC baseline, a DRAM row is composed of 112 data lines, 14 ECC lines, and 2 unused lines. The last unused line can be used to store metadata. Therefore, metadata storage is essentially free when using an embedded-ECC model.

Metadata itself is never stored in compressed format. The metadata cache stores 64-byte entries, with each entry representing 8 KB of contiguous data in a row. We later show results for different metadata cache sizes; in essence, a metadata cache sized similar to an L1 cache can represent most of the data in the LLC. The metadata cache access is on the critical path, but only adds a couple of cycles to the memory latency if there is a hit in the metadata cache.

Once the metadata information is obtained by the memory controller, the appropriate number of COL-RDs are issued to fetch the compressed cache block. It is worth pointing out one key difference from prior work. In prior designs, the metadata is used to *locate* the cache block. In MemZip, the location is known and the metadata is used to avoid bringing in other blocks, thus saving bandwidth and energy.

Once the compressed cache line is fetched, the first few header bits are examined. The compressed cache line format is shown in Figure 1. The first three bits of the header indicate which one of eight compression algorithms have been used. These algorithms are summarized in Table 1. Seven of these are BΔI algorithms. For these BΔI algorithms, the first few bits of the compressed cache line represent the BΔI bit mask, followed by the base and the deltas. Once the BΔI algorithm is known, the sizes of the bit mask, base, and deltas are known, so simple logic is required to interpret and compute the cache line. The eighth algorithm uses FPC. If FPC is being used, the first 48 bits of the compressed cache line indicate how the rest of the line should be interpreted. We assume that all of this decompression logic requires 3 cycles. Prior work has shown that BΔI can be implemented in 1 cycle [7] and FPC can be implemented in 2 cycles. We add an extra cycle for DBI conversion (discussed shortly in Section 3.4).

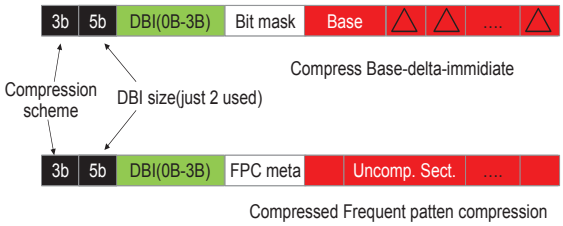

Figure 1. Compressed cache line structure.

000	fpc	001	BDel(8,0)
010	BDel(8,1)	011	BDel(8,2)
100	BDel(8,4)	101	BDel(4,1)
110	BDel(4,2)	111	BDel(2,1)

Table 1. Compression algorithms and codes.

A similar process is involved on a write, but in reverse order. Compression takes more effort because eight different compression algorithms must be evaluated. Compression latency is higher, but is off the critical path. Write operations typically wait in the write queue for hundreds of cycles and we assume that the compression is performed during this time. After compression, the appropriate number of COL-WRs are issued. The metadata is also updated (hopefully a hit in the metadata cache).

3.3 Impact on Reliability

As explained earlier, rank subsetting can increase the overhead for ECC support, requiring an extra chip for every rank subset. Embedded-ECC [40] reduces the storage overhead by storing ECC bits in the same row as the data itself. This requires an extra COL-RD for many cache line fetches and it leads to forced prefetch of ECC for neighboring lines.

The MemZip architecture can alleviate this overhead of embedded-ECC. Similar to the embedded-ECC data layout, a DRAM row contains 112 data lines, 14 ECC lines, and 2 unused lines. By default, every data line will have its ECC

codes saved among the 14 ECC lines at the end of the row. Now consider a case where a data line is compressed to say 26 bytes. A 26-byte line requires a 4-byte ECC code. This code can be placed immediately after the compressed data field instead of being placed among the 14 ECC lines at the end of the row. Since the cache line is fetched in increments of eight bytes, the ECC code is fetched without introducing an additional COL-RD. If the compressed cache line was 30 bytes in size, adding 4 bytes of ECC code would cross the 8-byte boundary and require an extra COL-RD. So there is no advantage to placing the ECC code immediately after the compressed data field. The ECC code therefore remains at the end of the row.

With this organization (compression + embedded-ECC), cache line fetches frequently do not require extra COL-RDs to retrieve their ECC codes. The memory controller can easily compute the location of the ECC code for every line. Once the header bits of the compressed cache line are examined, the memory controller can estimate if there was enough room to store the ECC codes without requiring an extra COL-RD. Note again that compression is being used not to improve capacity, but to improve other metrics (in this case, performance).

3.4 Reducing Energy with DBI

Instead of focusing on reliability, the spare bits in a compressed cache line can be used for energy efficiency. We first explain the Data Bus Inversion (DBI) technique [30]. Consider an 8-bit wide subrank that must transmit a 26-byte compressed line in 26 clock edges. If two successive bytes differ in 6 bits, 6 of the 8 bus wires will switch and dissipate dynamic energy. If we instead transmitted the inverse of the second byte, only 2 of the wires would switch. To save energy in this way, one extra bit would be required for every byte, to indicate if the byte is being sent in its original form or in its inverted form. Since there are 6 spare bytes in this example, we can easily accommodate 26 extra inversion bits. If 26 bits are not available, but 13 bits are available, every alternate byte can be considered for inversion. To keep the design simple, we allow either 0, 1, 2, or 3 bytes of DBI information. The more bytes of DBI information, the more energy we can potentially save. Of course, the overhead of sending additional DBI bits must also be factored in. Two bits are maintained in the first byte header of the compressed cache line (see Figure 1) to indicate the type of DBI encoding in use. The DBI bits follow right after. The DBI optimization is especially helpful in MemZip because compressed data tends to exhibit higher entropy (activity).

4 Analyzing Rank Subsetting

In this section, we first analyze the behavior of baseline rank subsetting (RS). It is important to first optimize RS because our compression techniques are built on top of RS. We propose a new data layout that is especially useful for MemZip. The results in this section do not consider any compression; the effects of compression are evaluated in the next section.

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	4MB/64B/8-way, shared, 10-cycle
Coherence Protocol	Snooping MESI
DRAM Parameters	
DDR3	MT41J1G4 DDR3-1600 [3],
Baseline DRAM Configuration	4 72-bit channels (ECC) 1 DIMM/channel 1 rank/DIMM, 9 devices/rank
Total DRAM Capacity	4 GB
DRAM Bus Frequency	800 MHz
T_RCD/T_RP/T_CAS	11/11/11 (memory cycles)
T_RC/T_RAS/T_RRD	39/28/5 (memory cycles)
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size	48 entries per channel
High/Low water marks	32/16

Table 2. Simulator parameters.

Memory Power	
VDD/IDD0/IDD2P0	1.5 V/55mA/16mA
IDD2P1/IDD2N/IDD3P	32mA/28mA/38mA
IDD3N/IDD4R/IDD4W/IDD5	38 mA/157mA/128mA/155mA
Metadata Cache Power	
dyn. access energy	0.19 nJ
leakage power	7.9 mW
Compressor/Decompressor	
Compression power	15.08 mW
decompression power/frequency	17.5 mW/1 GHz

Table 3. Power estimation parameters.

4.1 Methodology

Our simulations use the Simics [2] full-system simulator with out-of-order cores. Our simulation parameters are listed in Table 2. We have integrated the detailed USIMM [11] DRAM simulator with Simics. The DRAM device model and timing parameters have been obtained from Micron datasheets [3] and are also summarized in Table 2. We adopt the open-page address mapping policy that places consecutive cache lines in the same row [20]. The memory controller scheduler employs the FR-FCFS policy. It uses high/low water marks in the write queue to drain writes in batches [11]. We expect that future processors will integrate 4-8 channels shared by tens of cores. To limit simulation time, we model a system with eight cores and a single channel and rank.

For our workloads, we use 20 memory-intensive programs from SPEC2k6 (libquantum, omnetpp, xalancbmk,

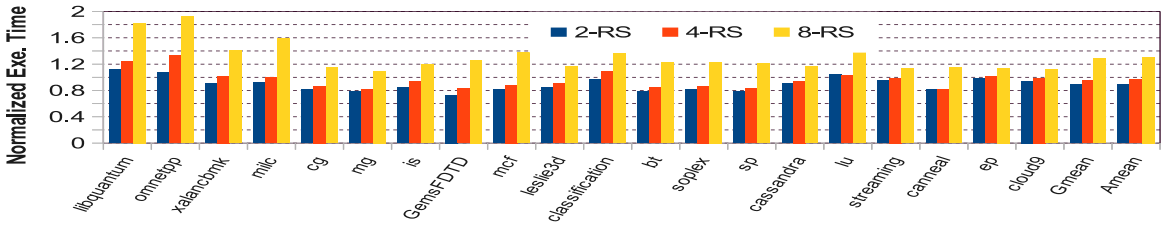

Figure 2. RS performance normalized to memory without rank sub-setting.

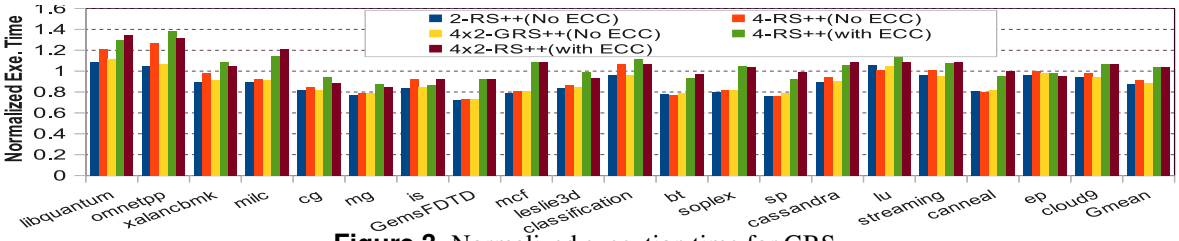

Figure 3. Normalized execution time for GRS.

milc, GemsFDTD, mcf, leslie3d, soplex), PARSEC [9] (canneal), NAS Parallel Benchmarks [8] (is, mg, bt, ep, sp), and CloudSuite [17] (classification, cassandra, cloud9). The SPEC2k6 programs are run in multi-programmed mode, with 8 copies of the same program, while the rest are run in multi-thread mode. The multi-threaded applications start detailed simulations at the start of the parallel region of interest. In the multi-programmed workloads, each core is fast forwarded for 20 billion instructions before starting simulations. The statistics for the first 100K memory transactions are ignored to account for cache warm-up effects. All of our simulations are executed until one million total DRAM accesses are encountered; this corresponds to 20-517 million committed cycles for various benchmarks. The use of DRAM access counts to terminate simulations ensures that roughly the same amount of work is done in each experiment for a given workload. This was also verified by examining other statistics such as DRAM reads/writes.

Our energy and power estimations are based on Micron's power calculator [1] for 4Gb DDR3 x8 chips, with access counts derived from our detailed simulations. Our estimates for metadata cache energy are derived with CACTI 6.5 [25] for 65 nm technology. We also synthesized the compression/decompression circuit using synopsis design compiler in 65 nm technology. Our power estimation parameters are listed in Table 3.

4.2 Rank Subset Results

Figure 2 shows the execution times for our benchmark suite for 2-way, 4-way, and 8-way RS, normalized against a conventional memory system with no RS. While rank subsetting can help reduce energy per memory access, it can have either positive or negative impacts on performance. Performance is positively impacted by affording a higher level of bank parallelism. Performance is negatively impacted by the increase in cache line transfer time and the increase in command bus contention. RS helps in several benchmarks, especially those that have higher bank conflicts and queuing delays. For example, the average queuing delay in *cg* reduces from 179 cycles in the baseline to 111 cycles in 2-way RS. In many cases, RS can degrade performance. In *cg*, in going from 4-way RS to 8-way RS, the command bus contention increases by 6x and the data transfer time increases by 2x, resulting in an overall average memory latency increase of 89 cycles. Therefore, *cg* exhibits highest performance for 2-way RS (23.4% better than the baseline), while the performance of 8-way RS is 15% worse than the baseline. On average, across all benchmarks, performance improvements over the baseline are 12.7% for 2-way RS, 4.8% for 4-way RS, and -22.5% for 8-way RS. This results trend is consistent with that shown in prior analyses of RS [5]. MemZip has the potential to do better with finer-grained memory access; we therefore first attempt to improve upon our baseline RS design. To reduce command bus contention, we consider DDR signaling for the command bus (shown by ++ in legends in subsequent figures). To reduce data transfer time, we consider a new data layout that is discussed next.

4.3 Modified Data Layout – GRS

As we move towards fine-grained RS, such as 4-way and 8-way RS, the long data transfer times tend to dominate, thus lowering performance. The fine-grained subranks are especially useful when dealing with compressed blocks because they minimize bandwidth waste. For example, if we are fetching a 38-byte compressed cache line, 2-way RS leads to 26 wasted bytes on the bus, 4-way RS leads to 10 wasted bytes, and 8-way RS leads to 2 wasted bytes. We must therefore devise a memory organization and layout that allows fine-grained memory access while supporting relatively low data transfer times. We refer to this new organization as *Generalized Rank Subsetting (GRS)*.

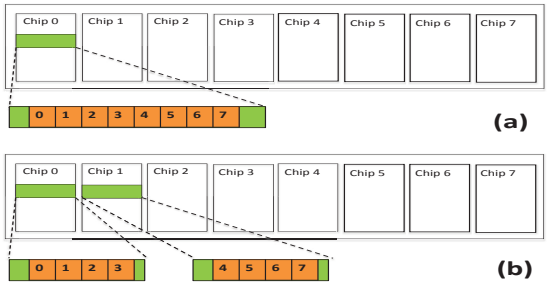

Figure 4. GRS data layout.

GRS logically combines fine-grained subranks into coarse-grained subranks. Consider the following example. Figure 4 (a) shows an 8-way subranked baseline. The 64-byte cache line is placed in a single subrank, and fetching the line requires 8 sequential COL-RDs. In GRS, shown in Figure 4 (b), we assume an 8-way subranked design, but a 64-byte cache line is placed across two subranks. Fetching the cache line still requires 8 COL-RDs, but at any time, 2 COL-RDs can be performed in parallel. So the cache line transfer time is equivalent to the delay for 4 sequential COL-RDs. This design point, referred to as GRS-8x2, is a hybrid of 8-way and 4-way RS. This design point is not meaningful for an uncompressed memory system – it has the data transfer time and parallelism of a 4-way RS system, but suffers from the command bus contention of an 8-way RS system. However, it is a meaningful design point for a compressed memory system. It offers the low data transfer time of 4-way RS, *and* the fine granularity of an 8-way RS system. For example, a 38-byte compressed cache line would be spread across the two 8-way subranks such that one subrank would provide 24 bytes (3 COL-RDs) and the second subrank would provide 16 bytes (2 COL-RDs). The delay would equal the delay for 3 sequential COL-RDs. The second subrank could have also supported another COL-RD in the same time. Instead, if we had used an 8-way RS design, the data transfer time would have equaled the delay for 5 sequential COL-RDs. If we had used a 4-way RS design, the data transfer time would have equaled the delay for 3 sequential COL-RDs, but the bus would have carried 10 empty bytes and the subrank supports no other operation during those three COL-RDs. In the GRS example above, one subrank services fewer COL-RDs than the other subrank. For load balance in this system, the start of every cache line must be shifted in a round-robin manner.

More generally stated, GRS-NxM refers to an organization that uses N-way rank subsetting, but spreads every cache line across M subranks. Figure 3 shows execution time results for each benchmark, normalized against the conventional non-subranked baseline. Unlike Figure 2, the models in Figure 3 assume DDR signaling for the address and command bus. The first three bars show execution times for 2-way RS, 4-way RS, and GRS-4x2, all without

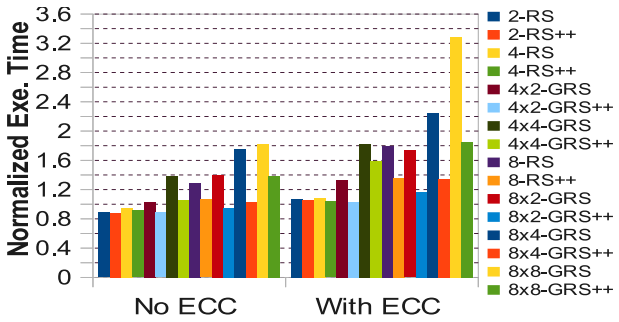

Figure 5. Geometric mean of different rank subsetting cases with and without ECC.

any ECC support. We see that GRS-4x2 is very similar to the high performance of 2-way RS, while still providing the finer granularity of 4-way RS.

Figure 5 shows normalized execution time (averaged across all benchmarks) for various RS and GRS-NxM organizations. We also separately show results for cases with and without ECC. The primary observations on this graph are: (i) Models with ECC have higher traffic rates and benefit from higher levels of subranking. (ii) The faster command bus typically improves performance by 1-2%. The improvement is higher for memory configurations supporting fine-grained access. (iii) The GRS-Nx2 configuration typically approaches the performance of the N/2-way RS configuration, e.g., GRS-4x2++ and 2-RS++ are similar.

5 Results

Figure 6 shows execution time for various rank subset configurations (without embedded-ECC) combined with memory compression. All of the bars are normalized against the execution time for a traditional baseline with no rank subsetting and no compression. Different configurations are optimal for each benchmark, but all three configurations yield overall geometric mean improvements of 30-45% over the baseline. Note that rank subsetting by itself can only yield an improvement of 15%, so most of this benefit can be attributed to MemZip's reduction of memory bandwidth pressure by fetching compressed lines. Performance varies depending on the compression ratio for the benchmark, the metadata cache hit rates, and how the program is impacted by the higher parallelism and higher latency afforded by rank subsetting. Figure 7 shows the compressibility for each benchmark. A few benchmarks show performance degradation in Figure 6. In the case of *milc*, most compressed lines are greater than 32 bytes, so there is no bandwidth reduction when using the 2-way RS configuration. It also has a high metadata cache miss rate of 37.5%. In the case of benchmark *is* with fine-grained COL-RDs, we noticed that compression frequently resulted in empty read queues, which in turn caused frequent write drains, which were interrupted by newly arriving read requests. This frequent toggle between reads and writes led to poor row buffer

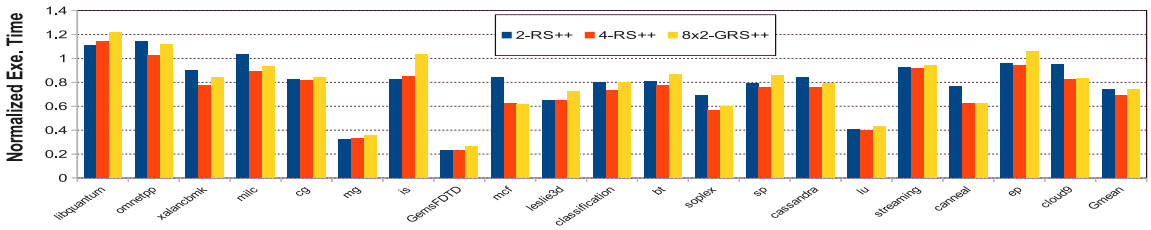

Figure 6. Normalized execution time for different MemZip configurations.

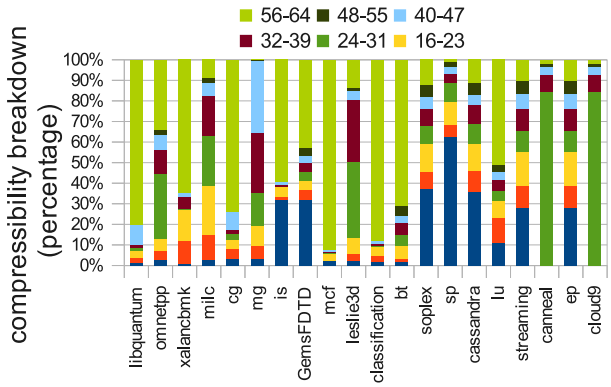

Figure 7. Breakdown of compressibility for different applications.

hit rates.

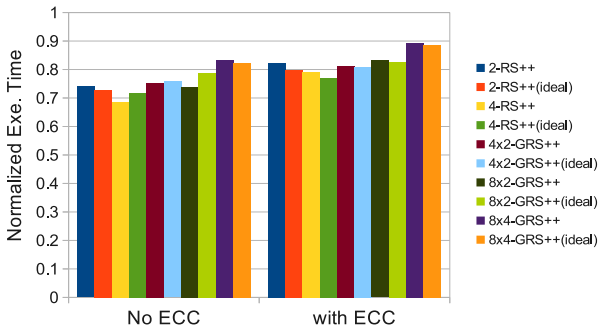

Figure 8. Geometric mean of execution times for different MemZip configurations with and without ECC. Ideal refers to a perfect metadata cache.

Figure 8 shows normalized execution times (GM for all benchmarks) for MemZip for a wide variety of rank subset configurations, with and without ECC support. We also show the idealized performance possible with a perfect metadata cache. All bars are normalized against the traditional baseline with no rank subsetting and no compression. The performance levels are all very similar, with 2-way rank subsets marginally outperforming 8-way rank subsets. As we see later, 8-way rank subsets save more energy than 2-way rank subsets. The ECC cases all have longer execution times than the non-ECC cases. This is because embedded-ECC implementations introduce extra COL-RDs to fetch the ECC information. When embedded-ECC is added to the

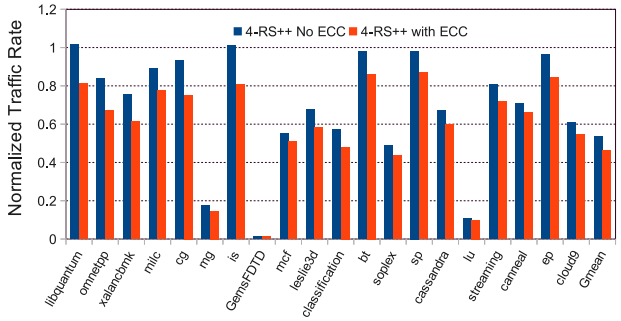

Figure 9. Memory traffic for MemZip normalized to non-compressed 4-RS with and without ECC.

8x2-GRS organization without compression, performance degrades by 21.8%. When embedded-ECC is added to the 8x2-GRS organization with compression, the performance only degrades by 12.5%, i.e., some of the ECC codes are fetched at no extra cost and don't result in performance penalties.

Figure 9 shows the normalized memory traffic with MemZip for the 4-way RS model, with and without ECC. Note that the ECC bar is normalized against a baseline with ECC, while the no-ECC bar is normalized against a baseline without ECC. Compression is able to reduce traffic by 46.6% in the no-ECC case and by 53.7% in the ECC case. The reduction is higher in the ECC case because many ECC codes can be accommodated in the spare space within a compressed line and fetched for free.

In the interest of space we don't report hit rates for our 8 KB 8-way metadata cache for our benchmark suite. We observed that benchmarks known to have large working set sizes (e.g., mcf, classification, canneal, omnetpp) show poorer hit rates (32-58%). Half the benchmark suite has metadata cache hit rates higher than 90%. On average, applications have a 93% metadata cache hit rate.

Figure 10 shows the energy for the memory system, metadata cache, and compression/decompression logic for different MemZip configurations. These bars are normalized against the energy for the traditional baseline with no rank subsetting. Note that the energy consumptions of the metadata cache and the compression/decompression logic are dwarfed by the memory energy in all cases. The energy reduction is higher for fine-grain rank subsetting, increasing

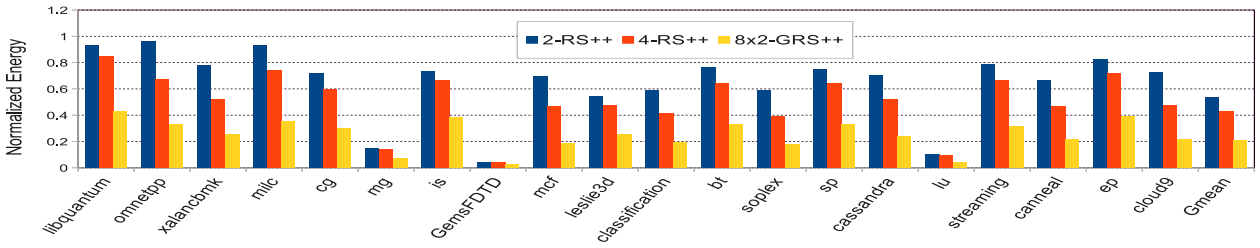

Figure 10. Memory energy for MemZip normalized to non-compressed non-sub-ranked baseline.

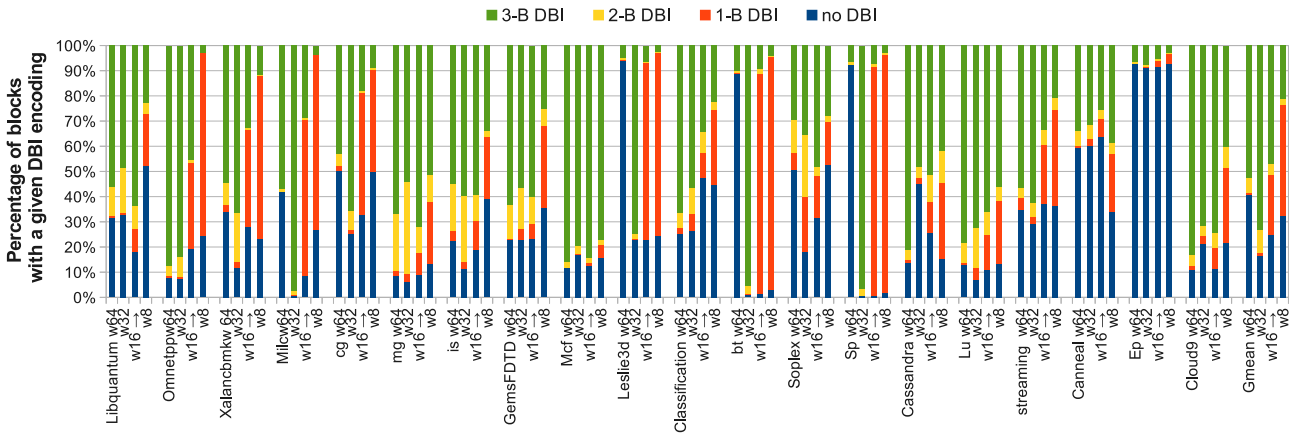

Figure 11. Breakdown of DBI encodings used for each benchmark and various bus widths in MemZip. All-zero and non-compressible lines are excluded in this breakdown.

from 46% in 2-way RS to 57% in 4-way RS, and to 79% in GRS-8x2. Some of this energy benefit comes from rank subsetting itself and some of it from fetching compressed blocks. The energy saving from compression alone is 33-40% for the different configurations.

While we don't show a figure for energy-delay product (EDP), it is clear that the GRS-8x2 configuration will emerge as a clear winner. All the configurations in Figure 8 have very similar execution times, but the 8-way rank subsets have much lower energy than 4-way and 2-way rank subsets. This also highlights the importance of the new GRS data layout. Note that 8-way RS has much poorer performance (Figure 2); a GRS-8x2 data layout is required to achieve the low energy of 8-way rank subsets, while also achieving high performance.

Finally, we show the energy savings with the DBI optimization. This is represented by the activity metric, i.e., how many bit-flips are encountered on the memory channel for all cache line transfers. We observe that moving from 64-wide buses to narrow buses (rank subsetting) increases the activity level on average. This is primarily because some applications have 64-bit data fields that align favorably on consecutive data transfers; this alignment is lost when moving to narrower buses. Once the lines are compressed, the total number of bit-flips are reduced, but this reduction is not proportional to the reduction in total traffic, i.e., compressed lines exhibit higher entropy. Figure 12 shows the reduction in activity when DBI encoding is added to various configurations with compression included. DBI is applied such that the DBI encoding bits can be accommodated in the available space in that line. For the 32-wide bus, DBI encoding causes a 30% reduction in activity. The saving reduces for 16- and 8-wide buses because they have less available space for DBI encoding bits. Figure 11 shows a breakdown of how often different DBI encodings were invoked in each configuration. It is worth noting that the DBI optimization will help reduce memory link energy, which can account for up to 40% of memory system energy [1].

6 Related Work

Data compression has been applied and evaluated for different components from the register file [22, 10] up to the hard disk drive [39]. Compression has been mainly considered to reduce the number and the size of data transfer packets. In this section we review some of these works that relate to the main memory system and last level cache.

There are several hardware and software compression approaches for main memory. We have reviewed the three main hardware approaches [4, 16, 26] in Sections 2.4, 2.5, and 2.6. MemZip yields an energy and bandwidth advan-

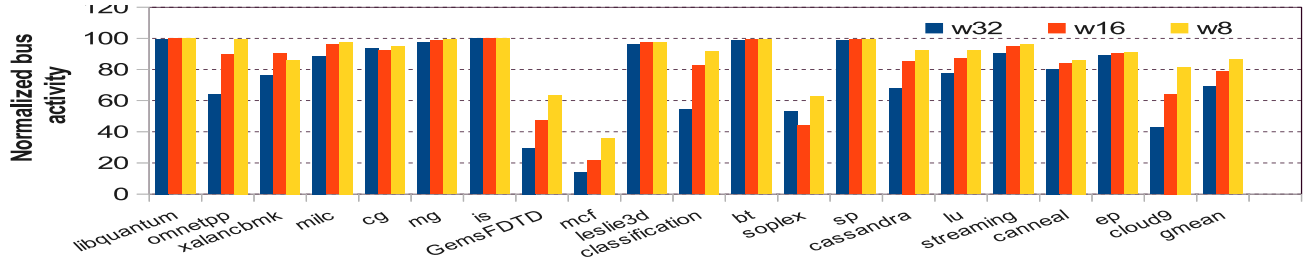

Figure 12. Bit flips on the bus for MemZip, normalized to corresponding non-compressed RS baselines.

tage over these prior works, especially if the application exhibits little spatial locality.

Some software-level high-latency compression techniques are used to save disk bandwidth when paging to disk on a page fault [14, 35, 13]. Tuduce and Gross apply compression to non-critical data in memory to mitigate performance degradation from long-latency compression [32]. Kjelso et al. [24] show that hardware compression performs better than software compression.

The other way to save memory bandwidth is to reduce miss rate. Many prior works have proposed compression-based cache designs that lead to miss reduction. Lee and Hong have used an X_RL compression algorithm for cache compression and have tried to hide high latency decompression in order to improve performance [21]. Villa et al. have proposed Frequent Value Compression for all levels of cache, especially the low-associative first level cache. Their compression scheme is based on the fact that most values read from or written to the memory hierarchy belong to a few special patterns [34]. Alameldeen and Wood also found similarities in the patterns of words of cache lines. They use compression to keep more cache lines in a set to reduce cache miss rates. This approach, however, suffers from 5-cycle decompression [6, 7]. Another approach for cache compression was proposed by Islam and Stenstrom [19], and Dusser et al. [15], which compresses cache lines with entire zero values. Yang et al. use the same approach to reduce power in the cache [36]. C-pack is a recent proposal that performs parallel decompression of multiple cache lines [12]. Pekhimenko et al. propose base-delta-immediate [27], which is an algorithm that relies on small differences between words in the same line. This algorithm is applied to compression in the cache hierarchy. Hallnor and Reinhardt investigate designs with both memory and cache compression [18].

Thuresson et al. evaluate a value locality compression scheme for the memory bus [31]. Similar to our work, they do not attempt to save memory capacity. Blocks are sent from the LLC to an off-chip memory controller in compressed format. The memory controller then stores the block in memory in its uncompressed form. This link compression approach is not applicable in most modern systems that integrate the memory controller on the processor chip.

Some papers have proposed finer granularity memory access, but have not considered compression. Mini-rank first proposed rank sub-setting for energy reduction [40]. Yoon et al. [38, 37] rely on rank subsetting to allow fine grain memory access, i.e., only part of a line is sent over the memory bus. This is primarily used to save bandwidth and power, but requires a sector cache [23].

Skerlj and Ienne trade off reliability and energy by using weaker ECC codes and using that space to store DBI bits [29].

Finally, Sathish et al. [28] consider compression for bandwidth saving in GPUs. Similar to our work, they too do not attempt to save memory capacity and rely on a metadata cache to fetch the appropriate amount of compressed data from GDDR5 memory. Our approach builds on this prior work in many ways. First, while GDDR5 allows for fine-grained memory access, DDR3 does not. To allow for fine-grained access, we combine compression with rank subsetting. We also introduce GRS to mitigate the long data transfer times inherent in rank subsetting. Second, we take advantage of unused space to send ECC information without requiring another data burst. Third, MemZip reduces bus activity with DBI, again exploiting the unused space created by compression.

7 Conclusion

The MemZip organization attempts memory compression with a focus on low complexity, reliability, and energy efficiency. It also derives high performance by reducing bandwidth pressure. It is able to achieve these goals by combining compression with rank subsetting, and with novel data layouts. It integrates the data layout with embedded-ECC codes as well as DBI codes. We show that the use of the new GRS data layout allows us to dramatically reduce energy per access, while not incurring long data transfer times. We believe this is an area of promising future work. There is the potential to further improve performance with intelligent schedulers that can prioritize the shortest job or deprioritize ECC code fetches.

References

[1] Micron System Power Calculator. `http://goo.gl/4dzK6`.

[2] Wind River Simics Full System Simulator. `http://www.windriver.com/products/simics/`.

[3] Micron DDR3 SDRAM Part MT41J1G4, 2009.

[4] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory Expansion Technology (MXT): Software Support and Performance. *IBM JRD*, 2001.

[5] J. Ahn and R. S. S. N. Jouppi. Future Scaling of Processor-Memory Interfaces. In *Proceedings of SC*, 2009.

[6] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of ISCA*, 2004.

[7] A. R. Alameldeen and D. A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme For L2 Caches. Technical report, University of Wisconsin-Madison, 2004.

[8] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. V. andd S. Weeretunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1994.

[9] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of IISWC*, 2008.

[10] R. Canal, A. Gonzalez, and J. Smith. Very Low Power Pipelines Using Significance Compression. In *Proceedings of MICRO-33*, 2000.

[11] 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. Technical report, University of Utah, 2012. UUCS-12-002.

[12] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE TVLSI*, 2010.

[13] R. S. de Castro, A. P. do Lago, and D. D. Silva. Adaptive Compressed Caching: Design and Implementation. In *Proceedings of SBAC-PAD*, 2003.

[14] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proceedings of Winter USENIX Conference*, 1993.

[15] J. Dusser, T. Piquet, and A. Seznec. Zero-Content Augmented Caches. In *Proceedings of ICS*, 2009.

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

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of ASPLOS*, 2012.

[18] E. Hallnor and S. Reinhardt. A Unified Compressed Memory Hierarchy. In *Proceedings of HPCA*, 2005.

[19] M. M. Islam and P. Stenstrom. Zero-Value Caches: Cancelling Loads That Return Zero. In *Proceedings of PACT*, 2004.

[20] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.

[21] J. Lee, W. Hong, and S. Kim. Design and Evaluation of a Selective Compressed Memory System. In *Proceedings of ICCD*, 1999.

[22] M. H. Lipasti and E. G. B. R. Mestan. Physical Register Inlining. In *Proceedings of ISCA*, 2004.

[23] J. S. Liptay. Structural Aspects of the System/360 Model 85. *The Cache. IBM Systems Journal*, 7, 1968.

[24] M.Kjelso, M. Gooch, and S. Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of of the 22nd Euromicro Conference*, 1996.

[25] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Architecting Efficient Interconnects for Large Caches with CACTI 6.0. *IEEE Micro (Special Issue on Top Picks from Architecture Conferences)*, Jan/Feb 2008.

[26] 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.

[27] 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.

[28] 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.

[29] M. Skerlj and P. Ienne. Error Protected Data Bus Inversion Using Standard DRAM Components. In *Proceedings of ISQED*, 2008.

[30] M. R. Stan and W. P. Burleson. Bus-Invert Coding for Low-Power I/O. *TVLSI*, 3, 1995.

[31] M. Thuresson, L. Spracklen, and P. Stenstrom. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans on Computer*, 2008.

[32] I. Tuduce and T. Gross. Adaptive Main Memory Compression. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[33] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *Proceedings of ISCA*, 2010.

[34] L. Villa, M. Zhang, and K. Asanovic. Dynamic Zero Compression For Cache Energy Reduction. In *Proceedings of MICRO-33*, 2000.

[35] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of USENIX Annual Technical Conference*, 1999.

[36] J. Yang and R. Gupta. Energy Efficient Frequent Value Data Cache Design. In *IEEE/ACM International Symposium on Microarchitecture*, 2002.

[37] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive Granularity Memory Systems: A Tradeoff Between Storage Efficiency and Throughput. In *Proceedings of ISCA*, 2011.

[38] D. H. Yoon, M. Sullivan, M. K. Jeong, and M. Erez. The Dynamic Granularity Memory System. In *Proceedings of ISCA*, 2012.

[39] L. You, K. Pollack, D. D. E. Long, , and K. G. Presidio. A Framework For Efficient Archival Data Storage. *ACM Transactions on Storage*, 7(2), 2011.

[40] 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.