Alleviating Memory Refresh Overhead via Data Compression for High Performance and Energy Efficiency

Ke Zhou, *Member, IEEE*, Wenjie Liu, Kun Tang, *Student Member, IEEE*, Ping Huang, and Xubin He, *Senior Member, IEEE*

Abstract—DRAM memory suffers from increasingly aggravating refresh penalty, which causes significant performance degradation and power consumption. As memory capacity increases, refresh penalty has become increasingly worse as more rows have to be refreshed. In this work, we propose an effective refresh approach called *Compression-Aware Refresh (CAR)* to efficiently mitigate refresh overheads. We apply a data compression technique to store data in a compressed format so that the resultant sparse banks need only be partially refreshed. Because of compression, data blocks which are originally distributed across all the constituent chips of a rank only need to be stored in a subset of those chips, leaving banks in the remaining chips not fully occupied. As a result, the memory controller can safely skip refreshing memory rows which contain no useful data without compromising data integrity. Such a compression-aware refresh scheme significantly reduces the refresh and thus improves overall memory performance and energy efficiency. To further take advantage of data compression, we adopt a rank subsetting technique to enable accesses to only those occupied chips for memory requests accessing compressed data blocks. Evaluations using benchmarks from SPEC CPU 2006 and the PARSEC 3.0 on recent DDR4 memory systems have shown that CAR achieves up to 1.66x performance improvement (11.7 percent on average) and up to 45.9 percent energy reduction (27.3 percent on average), and reduce memory traffic by up to 99.9 percent for zero cache lines intensive workloads with an average of 66.3 percent across all benchmarks.

Index Terms—DRAM refresh, energy, performance, compression

1 Introduction

Memory performance is crucial to the overall system performance as it provides an intermediate stage to bridge the huge gap between the performance anticipated by processors and the stagnant operations of disk systems or flash storage. As data-intensive workloads and big-data applications become more and more prevalent, the demand for high performance and large capacity from the memory system has steadily increased [2], [3], [4], [5]. Furthermore, the memory system contributes a significant percentage of the overall system energy consumption. Previous works [6], [7], [8] have shown that memory system accounts for up to 30 percent of the total system energy consumption, which does not account in additional energy associated with heat dissipation yet, such as power consumed by cooling facilities. Therefore, effective techniques that optimize memory performance and energy efficiency have been relentlessly researched in various aspects, spanning from system level, architectural level, to device level. Our work presented in this paper follows this effort trend to optimize memory performance and energy efficiency at the device level via optimizing refresh operations inside a memory system.

Modern DRAM memory technology is primarily built with capacitive cells, each of which contains one capacitor and one transistor. Information in a cell is represented by the charge trapped in the capacitor. Since capacitors leak over time by its very nature, DRAM cells must be *refreshed* periodically to guarantee data integrity [9], [10], [11]. The basic requirement is that every memory cell must be refreshed at least once within its so-called *retention time* window [9], [12], [13], [14] even when the whole memory system remains idle, otherwise data loss might occur. However, the needed refresh operations causes significant performance and energy overheads, causing "*Refresh Wall* " problem [9]. Even worse, as memory density increases, the refresh overheads grow substantially since there are more rows that need to be refreshed and the device unavailability time increases commensurately. It is projected that in future 64 Gb devices, refresh operations would account for 50 percent of energy consumption and degrade 50 percent memory throughput at the same time [9], [15]. High refresh

- *K. Zhou and W. Liu are with the Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education of China, and School of Computer Science & Technology, Huazhong University of Science & Technology, Wuhan 430073, China. E-mail: {k.zhou, lwj0012}@hust.edu.cn.*
- *K. Tang, P. Huang, and X. He are with Temple University, Philadelphia, PA 19122. E-mail: {kun.tang, TempleStorager, xubin.he}@temple.edu.*

Manuscript received 7 Jan. 2017; revised 18 Sept. 2017; accepted 28 Sept. 2017. Date of publication 16 Oct. 2017; date of current version 13 June 2018. (Corresponding authors: Ping Huang and Xubin He.)
Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2763141

overheads have become a critically detrimental factor to memory performance and energy efficiency.

Extensive recent research efforts have been invested on how to mitigate refresh overheads. Existing refresh mitigation techniques focus on hiding performance interference [11], [16], [17], [18] and reducing refresh operations [10], [12], [19]. A rank-level refresh operation locks up a whole rank and takes much longer time than a normal memory access, therefore, if a conflict between refresh and memory access occurs, it is beneficial to prioritize memory request over refresh via first dispatching memory access [16] or temporarily suspending on-going refresh operation [11] so as to avoid refresh interference. Enabling refresh to be performed on a finer-granularity, e.g., sub-rank [17], bank [18], subarray [20], can also reduce the probability of access conflict and hide refresh interference as other resources can service memory requests while refresh is being performed. In retention-aware refresh schemes[9], [13], refresh reduction is realized via avoiding refreshing unnecessary memory rows by taking advantage of the fact that the actual retention time of the vast majority of memory rows are much longer than the standard-defined limit (i.e., 64 ms under normal temperature and 32 ms under high temperature). Memory rows having different retention times are refreshed at differentiated rates. The potential challenge is how to accurately characterize memory cells' retention time. This is quite difficult due to *Variable Retention Time (VRT)* [13], which refers to a phenomenon that memory rows shift dynamically between a short and a long retention state. A recent research [9] has shown that such retention time-based multi-rate refresh approach could lead to unacceptable reliability. Refresh hiding can only alleviate refresh performance impact, while refresh reduction can reduce both performance and energy overheads.

In this paper, our goal is to devise a low-complexity, practical, and straightforward approach to mitigate refresh overheads in respects of both performance and energy. Based on previous works on memory compression, we propose to compress data at the memory controller and store data in a compressed format so that the memory rows in unoccupied space can be obviated from refreshes. Many previous works [21], [22], [23] and our own observations (Section 3.1) have revealed that most cache lines are reasonably compressible, implying there exist a good opportunity to achieve refresh savings via data compression. We employ the well-known low-complexity, low-latency base-delta-immediate (BDI) [21] compression algorithm as our compression engine to minimize introduced overheads.[1] With compression, a 64-byte data block[2] only requires a portion of the original space, resulting in *sparse* banks which are eligible for not being refreshed. We design a two-level vector structure to quickly identify which rows in sparse banks can be exempt from refreshes. When a bank receives a refresh command which intends to refresh a number of rows, the memory controller first checks the vector to determine which of those rows contain no useful data and skips refreshing them using a "dummy refresh" [15], which simply increases the refresh row counter but does not actually refresh a row. We rely on *Rank Subsetting* [24], [25] to partially access a rank (i.e., access the chips containing after-compression content) and obtain reduced memory traffic and access energy. Furthermore, zero cache lines are simply returned at the memory controller without accessing memory chips. Such a simple design has led to a significant amount of refresh reductions and greatly improves memory performance and energy efficiency due to the combined effects of the employed techniques, as detailed in Section 4.

In summary, we make the following contributions:

- We propose *Compression-Aware Refresh (CAR)*, a new refresh approach which employs a number of techniques (particularly data compression) to effectively reduce refresh operations. Compared with prior solutions, *CAR* is less intrusive and can be implemented in a relatively easier way as it neither requires changes to the applications or the operating system, nor relies on retention-time profiling which needs a lot of efforts to achieve due to variable retention time. To our best knowledge, our work is the first attempt to utilize the compression technique to mitigate refresh overheads.
- We have shown that the majority of memory cache blocks (64-byte in size) in all chosen benchmarks (from both SPEC CPU2006 and PARSEC 3.0) can be compressed to a much smaller size using the BDI algorithm, giving significant room for refresh reductions. As a fact, 26.1 percent of the written cache lines can be compressed to be less than 8 bytes.
- We have comprehensively evaluated the efficacy of CAR using benchmarks from SPEC CPU2006 and PARSEC 3.0 with modern DDR4 memory model. The evaluation results are very encouraging and have shown that CAR is able to dramatically mitigate refresh overheads. Overall, it achieves up to 1.66× performance improvement (with an average of 11.7 percent), while saving up to 45.9 percent energy consumption (with an average of 27.3 percent), compared to a regular non-compression, auto-refresh memory system.

The rest of the this paper is organized as following. In Section 2, we overview the necessary background knowledge. In Section 3, we elaborate on the design and implementation details of our proposed approach. Following that, Section 4 conducts comprehensive experiments to evaluate its efficacy and presents the results. Related work is then discussed in Section 5. Finally, we conclude this paper with a brief conclusion and future directions in Section 6.

2 Background

In this section, we first discuss the basics of DRAM memory. We then discuss the refresh problem in memory systems and relevant refresh mitigation techniques, followed by some background knowledge of memory compression techniques, as our work primarily relies on the memory compression technique to address the refresh problem.

1. In our paper, BDI is used as an example, other compression schemes can also be used.

2. We use "data block" and "cache line" interchangeably in this paper to indicate a data unit that transfers between the memory controller and the subordinate memory chips.

2.1 DRAM Basics

Modern memory systems are commonly built using dynamic random access memory (DRAM) technology. Memory systems are constructed in a hierarchical manner: *channel*, *rank*, and *bank* from a high level to low level perspective. Each channel can typically support 1-4 ranks, and a rank is a collection of DRAM chips that work in concert to feed the data width (64 bytes in DDR3 and DDR4). As defined by JEDEC specifications, each memory controller is responsible for a channel which has a 64-bit data bus and another 23-bit address/command bus [23]. Each memory chip internally contains multiple banks (typically 8), each of which is a two dimensional array of DRAM cells. The amount of chips comprising a rank is determined by the chip's data width. For example, if the chip data width is 8-bit, then it requires 8 chips to construct a rank (denoted as 8×8). Similarly, if the chip data width is 4-bit, then a rank contains 16 chips (denoted as 16×4). A cache line is striped across the same banks from the chips of a rank. For example, assuming an 8×8 memory system, a 64-byte cache line may be distributed across all the eight *bank0* [3] in the chips, with each *bank0* storing 8 bytes. The set of the same banks (e.g., all *bank0*) from the constituent chips work concurrently and simultaneously to service memory requests and can be perceived as a logical bank.

Accessing a cache line follows two steps. First, the address of a selected row is sent on the address bus, i.e., sending an *Active (ACT)* command. In response, the entire row content is transferred to an internal row buffer, the so-called *row activation* process. Due to a bitline's parasitic capacitance being proportional to its wire length, activating a row that is near to the row buffer takes less time than activating a distant row [26]. Next, the column address is sent on the address bus, i.e., a *Column-Read (COL-RD)* command, and in response, the data in the corresponding column is sent out to the data bus. A subsequent access that requests the data in an already opened row results in a *row buffer hit* [27], [28], [29], which is obviated from the row activation overhead and therefore results in better performance. Workloads showing good locality could significantly benefit from the *open-page* mode. As opposed, if a subsequent memory request needs to access another row rather than the currently opened row, then the memory controller must first close the opened row and then open the target row, resulting in increased service latency. Workloads having poor locality or showing high interference with other concurrently running applications would suffer from open-page policy and prefer *close-page* policy. *Rank Subsetting* technique [23], [24], [25] groups the chips in a rank into different subsets and allows individual subsets to operate independently. Our approach leverages rank subsetting to activate only the occupied chips containing valid after-compression content in response to memory requests.

2.2 DRAM Refresh

DRAM cells are leaky and therefore require periodical refresh operations to ensure data integrity. The refresh operation brings DRAM rows into the row buffer (a.k.a, sense amplifiers), precharges them in the buffer, and then restores the content back. Every cell can only sustain a limited period of time called *retention time* [13]. The retention time is defined by the JEDEC specification to be 64 and 32 ms, under normal (below $85°C$) and high temperature (above $85°C$), respectively. The memory controller needs to refresh all the memory cells before the retention time expires. The DDR specifications define that all DRAM rows are grouped into 8K refresh bundles and each refresh operation refreshes one bundle at a time, causing 8K refresh operations to be issued within the retention time [11]. The time taken to refresh a bundle is referred to as *refresh cycle* (T_{RFC}). Refresh operations can be performed in *per rank* or *per-bank* mode, where an entire rank or only a bank is locked up and becomes unavailable during the refresh cycle. It has been observed that as memory density increases, the probability of memory unavailability due to refresh operations keeps growing [9], [11], [15], [18]. Generally speaking, the negative performance impacts of refresh operations are resultant from the combination of three factors. First, the presence of refresh commands delays memory request dispatching, increasing their queuing latencies. Second, refresh operations lock up memory resources and cause device unavailability, blocking memory requests and lengthening their latencies. Third, blocked requests hog memory controller resources which otherwise could be used by regular memory requests. For example, blocked requests could cause *command queue seizure* [30] phenomenon which prevents other non-conflicting requests from progressing. Furthermore, refreshes consume a significant amount of energy, which in turn elevates the ambient temperature and threatens data reliability.

A variety of techniques have been proposed to mitigate refresh overheads. Those techniques alleviate refresh overheads via either hiding performance interference or reducing the occurrences of refresh operations. When memory requests conflict with a refresh operation, the memory controller may opt to give priority to the memory requests to avoid being blocked by appropriately changing the scheduling strategy at the memory controller [11], [16], [31]. Parallelizing the physical memory architecture and organization [17], [18] can enable memory requests to be serviced simultaneously when a refresh is being performed in another independently operating unit, such as bank, subarray, subrank. Refresh hiding techniques are mostly performance improvement oriented, with little or no energy efficiency improvement. On the other hand, reducing the total number of refresh operations can improve both performance and energy efficiency. Most of this category of techniques are based on memory retention time to eliminate unnecessary refreshes, except the *Smart Refresh* [10] that is based on history accesses. While retention time based refresh schemes have great potential of reducing refresh operations via leveraging the insightful observations of disparities in memory cells' realistic retention time [9], [12], [32], they may suffer the following two limitations. First, they may bear high implementation complexity, typically requiring changes to program codes [33] and/or operating system [32], which necessitates the expertise that belongs to a small focused community. Second, sophisticated profiling mechanisms are needed to characterize variable retention time accurately [9], [13], [14]. Our proposed refresh approach attempts to

3. If the bank id part of the translated (i.e., raw) address is 010, then the cache line is stored across all the eight *bank2*.

TABLE 1
A Summarized Comparison of DRAM Refresh Approaches

	Approach Name	Performance Improvement	Energy Saving	Retention Time-based	Compression-based	Changes
Refresh Hiding	ROP [31]	✓		no	no	memory controller
	Elastic Refresh [16]	✓		no	no	memory controller
	Refresh Pausing [11]	✓		no	no	memory controller
	CREAM [17]	✓		no	no	memory organization
	Parallel Refresh [18]	✓		no	no	memory organization
Refresh Reduction	Smart Refresh [10]	✓	✓	no	no	memory controller
	Flikker [33]	✓	✓	yes	no	program codes
	RAPID [32]	✓	✓	yes	no	operating system
	RAIDR [12]	✓	✓	yes	no	memory controller
	AVATAR [9]	✓	✓	yes	no	memory controller
	CAR	✓	✓	**no**	**yes**	**memory controller**
Others	MECC [34]		✓	no	no	memory controller
	REFLEX [15]		✓	yes	no	memory controller

address the refresh problem from a different perspective and is orthogonal to existing solutions. Table 1 gives a comparison of existing approaches and our approach (*CAR*).

2.3 Memory Compression

Memory compression has been conventionally leveraged to increase effective memory capacity [22], [35], [36], [37], [38], [39], improve the hit ratio at a higher memory level (e.g., on-chip L2 cache) [36], [38], reduce the memory traffic flowing between the processor and off-chip memory [36], [38], and between memory controller and memory chips [23]. Compression algorithms used to compress memory should be low-latency and hardware designs are preferable, as memory performance is critical to the applications. Various compression algorithms have been proposed and applied in previous works. We review three representatives which have been commonly used. Frequent Pattern Compression (FPC) [40] takes advantage of the observation that certain words occur frequently in cache lines. It thus divides a cache line into several words and uses an encoding code to represent the words, shortening the length of the cache line. Zero Value Compression (ZVC) [41], [42] is a special case of FPC. ZVC only compresses zero cache lines, as the amount of zero cache lines accounts for a significant portion in a lot of applications. Base-delta-immediate [21] leverages another important observation that the words in the same cache line exhibit a very small dynamic range, i.e., the differences between the words in the same cache line are quite small, which could result from common program behaviors such as initializing an array with a consecutive sequence of values and an array of pointers pointing to a continuous memory region. Therefore, a cache line can be represented as the combination of a base value and an array of small deltas each of which takes less space than its original value, resulting in less total storage space. As the BDI decompression is rather simple and fast, i.e., a matter of a vector addition, we employ BDI algorithm in CAR for cache line compression.

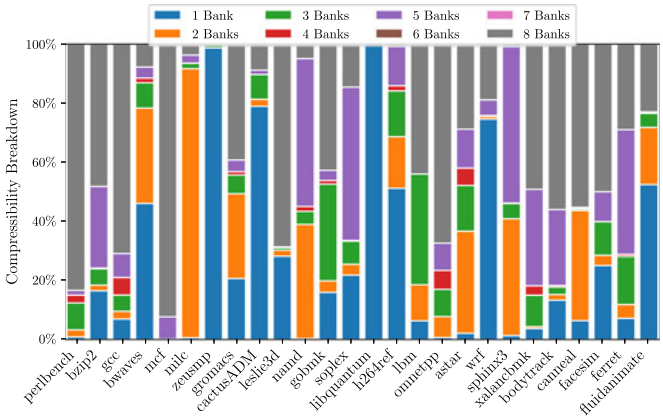
3 Compression-Aware Refresh

3.1 Motivation

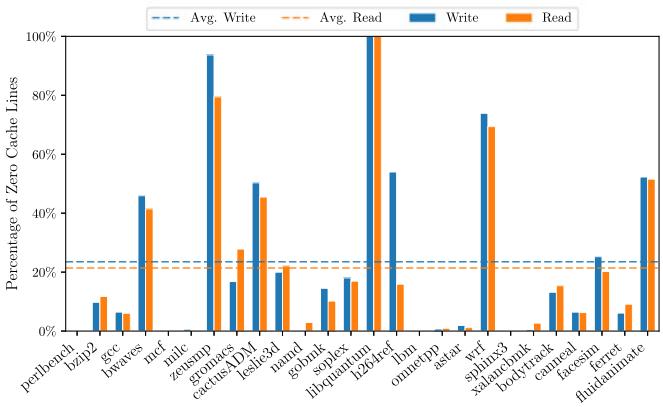
DRAM density keeps increasing, benefiting a lot to today's applications which are hungry for memory. However, the increasing internal refresh penalty is darkening the promising future sky. Observations and projections [9], [11], [15] have shown that the negative impacts of refresh operations have been steadily growing. While existing approaches, which are based on smart scheduling, parallel architecture, and retention time profiling, have been demonstrated to be effective in prior works, we believe employing data compression to reduce effectively occupied space and thus eliminate refresh operations in resultant unused space could be another simple, practical, easy-to-implement approach. The idea is straightforward. Assuming an 8×8 memory system, without compression, a 64-byte cache line is spread over eight banks among the eight constituent chips of a rank, with every bank (e.g., *bank0*) having an equal share of 8 consecutive bytes. In this situation, all eight banks need be refreshed to ensure data integrity. On the contrary, if the cache line were compressed to a smaller size, e.g., 23 bytes, then only three banks (in three different chips) would be used and the other five banks would not contain data from the cache line and therefore the corresponding rows residing in the unused space would potentially need no refresh, saving refresh operations.

To motivate our approach, we have conducted experiments (experimental details are given in Section 4) to investigate cache lines compressibility. We characterize cache lines compressibility based on all cache lines written to the memory system by applications. Please note the statistics were collected after the fast-forwarded instructions so as to rule out the biased effects of initialization phases. Fig. 1a shows the cache lines compressibility breakdown. As can be seen from the figure, most cache lines are reasonably compressible, implying using compression can produce significant sparse space in which refresh can be obviated. On average, 26.1 percent cache lines written by the benchmarks can be compressed to be less than 8 bytes, and 49.6 percent of the written cache lines can be compressed to less than 24 bytes. In particular, except for several benchmarks (e.g., *libquantum*) which exhibit pathologically high volume of zero cache lines, we have observed an impressive amount of zero cache lines[4] in most benchmarks, as demonstrated in Fig. 1b. As it shows, zero cache lines account for an average

4. A zero cache line is compressed to one byte using BDI algorithm.



(a) Cache line compressibility breakdown for applications.



(b) The percentage of read and written zero cache lines.

Fig. 1. Fig. 1a shows that most cache lines are reasonably compressible and Fig. 1b shows that zero cache lines account for an impressive amount for both read and write requests.

of 23.5 percent across all the benchmarks. Fig. 1b also shows the percentages of zero cache lines read by the applications. As it is shown, on average, 21.4 percent of the read requests were requesting zeros. Our proposed approach compression-aware refresh takes advantage of this observation by simply returning zero cache lines at the memory controller without actually accessing memory to further improve performance and energy efficiency.

3.2 Overall Architecture

Fig. 2 shows a simplified schematic view of our proposed CAR architecture. As shown in the figure, we add a compression engine in the memory controller. The compression engine performs compression and decompression for cache lines. Cache lines required by the processor are first brought into the memory controller for decompression and then returned to the processor. New cache lines coming from the processor are compressed and then stored in the memory banks. The number of banks (in different chips) that a cache block actually occupies depends on the compressed size. Depending on the compression ratio, cache lines might be compressed into variable sizes. In the figure, we assume a memory system comprising 8 chips, each of which is responsible for 8 bytes of a 64-byte cache line. Due to compression, banks in some of those chips may contain much less data than when cache lines are stored in uncompressed format. Those banks are called *sparse* banks. Memory rows

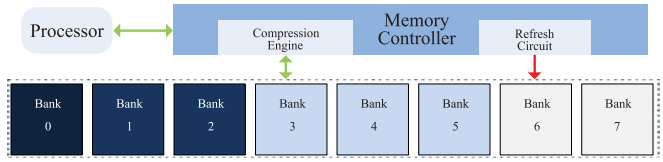


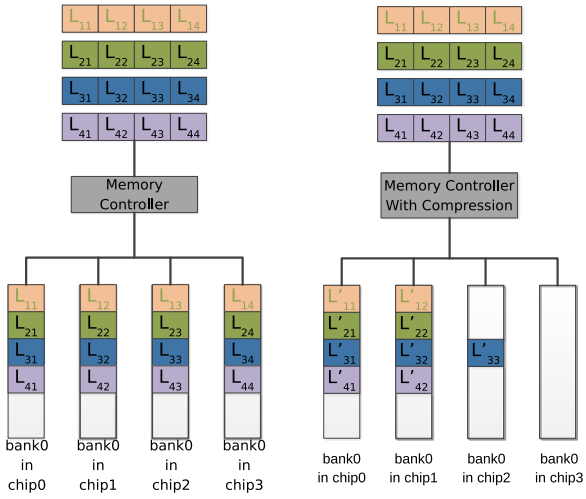
Fig. 2. A simplified schematic overview of CAR architecture. Cache line compression results in *sparse* banks in which memory rows may not need refresh. The different colors represent different degrees of sparseness of the bank.

in sparse banks may not need to be refreshed, as they contain no valid data. We also make necessary changes to the refresh mechanism to skip refreshing memory rows containing no valid data.

3.3 Memory Block Compression

Fig. 3 compares the original DRAM and a compression-capable DRAM. As it is shown in Fig. 3a, without compression, a cache line is distributed among all the four banks. On the other hand, if a cache line is compressed, then only a subset of the four banks are needed to store the compressed cache line. As it is illustrated in Fig. 3b, after cache line compression, *bank0* in *chip3* contains no valid data at all and can be completely exempt from refresh, while *bank0* in *chip2* is highly spare and thus only the memory row where L'_{33} resides needs to be refreshed. All the other memory rows in *bank0* in *chip2* need no refresh. Furthermore, since *bank0* of *chip3* contains no valid data at all, it is completely exempt from refreshes.

Due to its minimal-complexity, low-latency, we use the base-delta-immediate [21] cache line compression algorithm in our compression engine. Actually, the BDI algorithm was also used to compress cache lines in prior works [22], [23], [43] and has been demonstrated to be a practical, effective compression algorithm. Fig. 4 shows how to compress a cache line with the BDI algorithm. In the figure, a 64-byte cache line is divided into several words of equal length denoted as w. As the value differences among the words are small, the original cache line can be compressed as a chosen base word (e.g., the minimal or maximal word value), plus a vector of small deltas calculated as the differences



(a) Non-compressed DRAM (b) Compressed DRAM

Fig. 3. A cache line storage overview in non-compressed and compressed DRAM. As cache lines are shortened, fewer rows in some chips require refreshes.

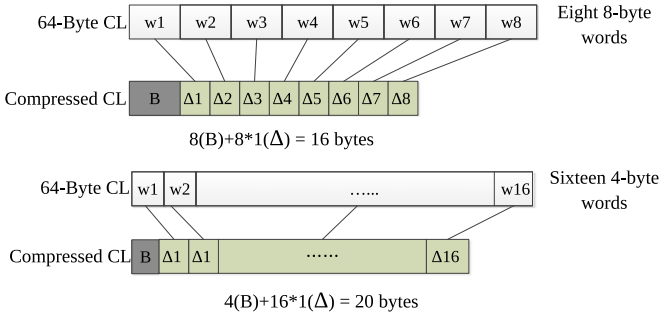


Fig. 4. The BDI compression algorithm. A cache line is perceived as a number of words, which exhibit small dynamic range. A cache line is represented as a base, plus the same number of small deltas to reduce space.

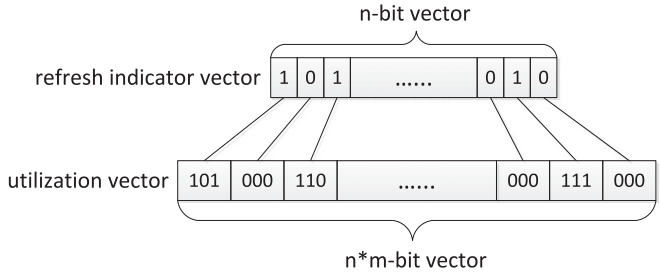


Fig. 5. A two-level data structure used to efficiently track rows that do not need refresh. The first-level vector indicates which rows need refresh. The second-level vector tracks the row utilizations.

between word values and the base word value. Typically, the memory size (denoted as k) of a delta value is much smaller than the original word length, producing small compressed cache line size. Using this compression method, a 64-byte cache line only requires $w + \frac{64}{w} \times k$ bytes. Fig. 4 shows that the same cache line can be compressed to 16 bytes or 20 bytes, assuming each delta needs one byte. If there does not exist any word partition satisfying the condition of $k < w$, then the cache line is considered to be incompressible. Otherwise, the BDI algorithm returns the result having the smallest compressed size (i.e., highest compression ratio), along with the encoding scheme information. To minimize compressed sizes, BDI uses two base words, with zero value being another implicit one. To decompress, BDI only requires to perform a vector addition, which is quite fast. As in prior works [22], [23], we also assume a 1 cycle decompression latency for our evaluation.

To support access to a compressed cache line, the encoding scheme specifying how a cache line is compressed (the values of w, k) should be preserved for references. We maintain the encoding information in the memory controller. As mentioned in the original BDI paper [21], the encoding scheme information takes four bits, requiring 0.78 percent ($\frac{4bit}{64bytes} = \frac{1}{128}$) of the memory capacity for the metadata region. To handle memory requests to cache lines, the memory controller should first check the encoding scheme to determine the compressed size and then activate the corresponding sub-rank (Section 3.5) containing the compressed cache line. As an alternative, encoding information could also be stored in the first byte of the compressed cache line. With this method, requesting a cache line needs to pay the cost of an additional access to the first byte (activating only one sub-rank) to ascertain the compressed size before obtaining the compressed cache line. If one has no interest in reducing memory traffic, the entire cache line frame can be first brought into the memory controller as if it were not compressed, and then use the encoding information to restore the correct content. Like previous works [22], [23], we also assume the compression procedure does not affect system performance since compression only happens with write requests which are typically buffered in the memory controller for a while. Furthermore, the memory controller could choose to first write cache lines in the memory and perform compression in the background at a later idle time to avoid performance interference. Another optimization *CAR* embraces is to simply return reads requesting for zero cache lines without visiting memory. This optimization adds to improve memory performance, particularly for zero-intensive applications (e.g., *libquantum*). We have observed this optimization results in up to $1.53\times$ additional performance improvement for the most zero-intensive benchmark, with an average of 8.75 percent across all the benchmarks.

3.4 Refresh Reduction

In order to safely skip refreshing invalid cache lines in a *sparse* bank, we need to track which rows in the bank are valid. To fulfill this purpose, we devise a two-level vector data structure. As it is shown in Fig. 5, the first-level is called *refresh indicator vector*, an n-bit vector. Each bit of this vector corresponds to a row in the bank and when the bit is set, the corresponding row is indicated as valid and needs refresh. The second-level is called *utilization vector*, a $(n * m)$-bit vector. Every m-bit acts as a counter to remember the number of valid cache lines the corresponding row currently contains. Whenever a compressed cache line takes space from a bank, the corresponding row counter of the bank is incremented by one. On the other hand, whenever a cache line becomes short (due to overwriting) and needs less space, then the counter corresponding to the released row is decremented by one. When a counter decreases to zero, meaning the corresponding row contains no valid cache lines, the corresponding row bit in the first-level vector is reset to 0, indicating to the memory controller to skip refreshing the corresponding row. The changes in compressed size cause counter values to be changed. Row granularity refresh controlling is possible as retention time based refresh approaches rely on similar row granularity refresh [12], [15]. As shown in Fig. 6, when a bank receives a refresh command to refresh a set of rows,[5] it checks the first-level vector to see which rows are empty and do not need to be refreshed. For those rows, it simply increments the refresh counter and does not perform refresh, as what a "dummy refresh" [15] does.

Assume an 8×8 memory system and the row size is 4 KB. As mentioned before (Section 3.1), a 64-byte cache line is evenly distributed among eight banks in the eight different chips. Each bank takes 8 bytes of the cache line. Therefore, a 4 KB row in a bank contains data from $\frac{4KB}{8B} = 512$ cache lines, resulting in each row counter in the *utilization vector* requiring $\log_2 512 = 9$ bits (i.e., $m = 9$). Besides, it requires one more bit for each bank row in the *refresh indicator vector*. In total, this two-level vector data structure incurs 10 bits overhead for each 4 KB bank row.

5. The memory controller or the bank maintains a refresh counter to remember from which row to start for refresh operations.

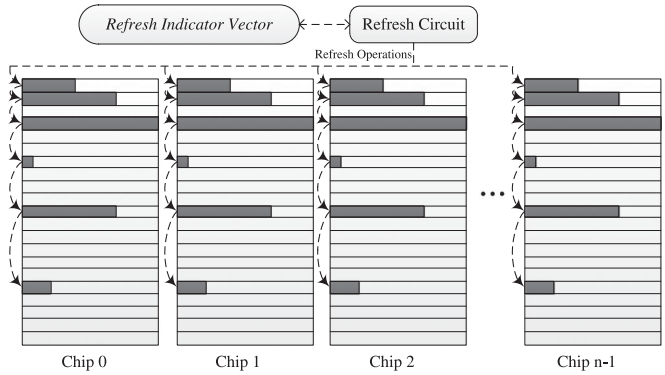


Fig. 6. The refresh operation in a sparse bank (Bank 0, with n chips). The refresh circuit issues refresh commands to the memory rows containing valid data but skips those unused memory rows. In doing so, it reduces significant amount of refreshes.

3.5 Read and Write Operations

Due to presence of the compression indirection layer, read and write operations need to be changed accordingly. To serve a cache line read, the memory controller first checks the encoding information metadata to obtain the encoding scheme and compressed size, and then decompresses the compressed content to a 64-byte cache line. To serve a cache line write, the memory controller first compresses the cache line using the BDI algorithm and updates its new encoding information in the metadata cache, which is synchronized to the physical memory when cache lines are flushed. Please note that BDI is a stateless compression algorithm, i.e., it compresses a cache line simply based on its own current content. Therefore, it does not incur additional overhead, except the compression overhead, which is minimal due to the simplicity of BDI algorithm [21], [23].

Based on that information, it decreases or increases the corresponding row counter values in the *utilization vector*. For example, if a compressed cache line originally has 23 bytes that are distributed among three banks in *chip0*, *chip1*, and *chip2*, respectively (see Fig. 2). Assume the cache line is overwritten with a new block that can be compressed to 16 bytes. The new content only needs two banks in *chip0* and *chip1*, therefore the corresponding row counter for the bank in *chip2* is decreased by one as it is no longer referred to by the overwritten cache line. Furthermore, if the counter value decreases to zero, the bit corresponding to the row in the *refresh indicator vector* is reset to zero to signify no refresh for this row. After that, it writes the new compressed content to the memory and updates encoding information.

To additionally take advantage of data compression for energy efficiency, we employ *rank subsetting* (RS) [24], [25] technique to activate part of the constituent chips for cache line accesses. With rank subsetting, we are able to activate the sub-ranks that contain the compressed content. For example, suppose an $8 * 8$ memory is partitioned to 4 sub-ranks each of which includes 2 chips and the length of a compressed cache line is 23 bytes. To read or write the cache line, it only needs to activate four chips (i.e., two sub-ranks) instead of all the 8 chips as in the case where rank subsetting is not deployed. It should be noted that the primary advantage of rank-subsetting is energy efficiency [44], [45] due to reduced memory traffic and access energy. Prior investigations [23], [24] have shown that rank-subsetting could either positively or negatively affect performance. The reason is because multiple column read commands are needed to obtain one cache line from sub-ranks due to each chip assuming more data from a cache line. In CAR, rank-subsetting would not cause such negative impacts as only one column read command is needed to obtain a compressed cache line, since the cache line data layout remains the same as in the vanilla system. In our evaluation, we conservatively assume rank-subsetting brings no positive impacts by offsetting the saved transferring time for compressed cache lines (i.e., treat them as uncompressed 64B when calculating transfer time) and other potential benefits (e.g., less activation time).

3.6 Metadata Analysis

The metadata overhead of CAR includes the encoding information of compression memory blocks and the storage for the two-level refresh indication vectors. In this section, we discuss where the medadata information is stored and what is the overhead.

As discussed in Section 3.3, using BDI algorithm each 64-byte cache line requires four bits metadata to remember its encoding scheme. Therefore, a 16 GB DRAM memory requires a total of 128 MB storage space, which is too big to be stored in on-chip storage. Our approach is to store the four bits encoding information at the beginning each compressed memory block and employ a metadata cache in the memory controller to store the encoding information of cache lines in the last level cache. The on-chip storage for metadata cache is proportional to the size of last-level cache. Each LLC cache line has a four-bit entry denoting its encoding scheme in the metadata cache. Putting it in perspective, for an 8 MB LLC, the on-chip storage is 64 KB, regardless of the memory size. When a cache line in the LLC is evicted, its encoding information in the metadata cache is updated to facilitate reaccesses to it in the future. When a new memory block is brought into the LLC, the associated metadata entry is updated with its encoding information read from memory. Storing the four-bit encoding scheme in the header of each compressed memory block does not cause the memory block to span an additional chip for the most majority of compressed blocks. It is also unlikely to introduce additional reads due to the unalignment of sub-ranking.

The two-level indication vectors are used to facilitate skipping refreshing unused memory rows. Therefore, they are only consulted when performing refreshes. So we store the vectors in a dedicated part of memory space in each of the chips, rather than in on-chip storage. As discussed in Section 3.4, each 4 KB memory row needs 10 bits in the refresh indication vector. As a result, a 4 Gbit chip requires 1.25 Mbit to store the refresh indication vectors and a whole 16 GB DRAM memory system requires a total of 5 MB memory storage. The vectors are updated appropriately when cache lines are evicted back to the memory.

4 Evaluation

4.1 Methodology

To evaluate CAR, we implement our idea in the cycle-accurate DRAMSim2 [46] memory simulator. We use the Zsim [47] processor simulator as the front-end and integrate the modified DRAM simulator with Zsim. We use the Pin

TABLE 2
Simulation Parameters

Processor	Single out-of-order core
Memory Controller	64/64-entry read/write request queue FR-FCFS, writes are scheduled in batches
DRAM	DDR4-1600, 1 channel, 1 rank per channel Memory organization: 8 x8 or 16 x4 sub-rank = 2, 4, 8
Refresh Parameters	tREFI=7.8μs/3.9μs/1.95μs for 1x/2x/4x, tRFC=350/260/160 ns for 8Gb DRAM chips with 1x/2x/4x, tRFC=480/350/260 ns for 16Gb DRAM chips with 1x/2x/4x
Cache Line Compression	Compression power=15.08 mW Decompression power=17.5mW/1 GHz Decompression delay=1 cycle

[48] tool to capture virtual addresses. As memory compression is performed on cache line basis and its efficiency does not rely on the number of ranks, for simplicity, we simulate a DDR4 memory system comprising one channel and one rank in the channel. We evaluate two memory capacities (8 and 16 GB) with two different organizations (8×8 and 16×4), as different configurations will affect how many chips a compressed cache line occupies. We also evaluate CAR working in different refresh modes of DDR4 to investigate the applicability of CAR. Otherwise specifically noted, we assume the default number of sub-ranks to be 2 and use 1x refresh mode of DDR4. Table 2 lists our main simulation parameters. The compression parameters are from [23]. We adapt the Micron Power calculator [49] to derive energy evaluations for DDR4. Table 3 shows our used IDD/IPP values for an 8 Gb DDR4 DRAM device according to the Micron specification [50]. Besides, based on the estimation using CACTI tool, we allow a 2-cycle latency for each metadata access to obtain compressed cache line size.[6]

For workloads, we select 21 benchmarks from the SPEC CPU2006 suite and 5 benchmarks from the PARSEC 3.0 suite [51]. These benchmarks can be classified as *High Compression* or *Low Compression*, and *Intensive* or *Non-intensive* based on compressibility and memory traffic intensity, respectively. High compressible benchmarks show low memory traffic, while low compressible benchmarks introduce high memory traffic. As shown in Fig. 1, the chosen benchmarks entail wide diversity in terms of compressibility and zero cache lines. Table 4 gives the benchmarks classification based on compressibility and memory intensiveness.

Before starting simulation, we fast forward one billion instructions. We then simulate another billion instructions for statistics. We comprehensively compare our approach with a non-compression, no rank subsetting, auto refresh (1x FGR mode) baseline memory system and a non-compression, no rank subsetting, no-refresh ideal memory system. In the meanwhile, we also compare with other two recent refresh schemes in respect of performance.

6. Please note the metadata access latency can be obviated if a fixed BDI scheme (i.e., fixed length for base value and deltas) is adopted based on before-hand knowledge about cache line compressibility. However, it may sacrifice compression ratio.

TABLE 3
DDR4 Power Parameters of 8 Gb Devices

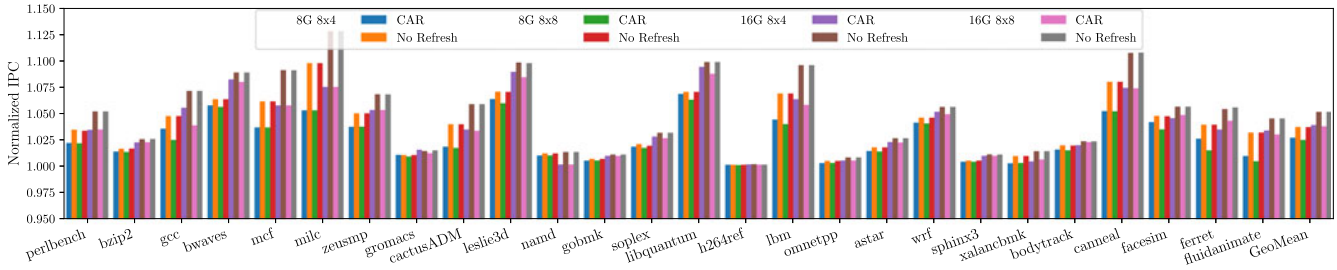
Power parameter	IDD	IPP
Supply voltage (VDD/VPP)	1.2 V	2.5 V
One bank active-precharge current (IDD0/IPP0)	45 mA	3 mA
Precharge power-down current (IDD2P/IPP2P)	25 mA	3 mA
Active standby current (IDD3N/IPP3N)	35 mA	3 mA
Active power-down current (IDD3P/IPP3P)	30 mA	3 mA
Burst read current (IDD4R/IPP4R)	100 mA	3 mA
Burst write current (IDD4W/IPP4W)	95 mA	3 mA
Burst refresh current (IDD5B/IPP5B)	250 mA	28 mA

4.2 Performance Comparison

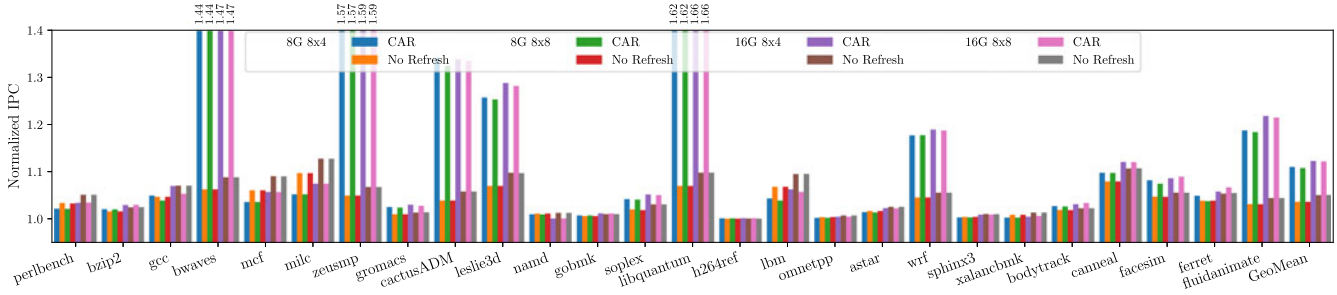
We use the Instruction Per Cycle (IPC) metric to compare the performance of our approach *CAR* with the baseline and no refresh systems. Figs. 7a and 7b demonstrate the normalized IPC comparison results without and with zero cache line optimization, respectively. As shown in the figure, with cache line optimization, CAR is able to improve performance as much as $1.66\times$ for *libquantum*, with an average of 11.7 percent, while without cache line optimization, CAR can improve performance by a maximum of 9.4 percent (3.2 percent on average). Two critical conclusions are in order. First, our approach can efficiently alleviate refresh performance penalty. Without zero cache line optimization, the IPC disparities between *no-refresh* and *baseline* represent performance degradations due to refresh operations. Generally speaking, non-intensive applications suffer from refresh penalty less than intensive applications, as fewer requests are blocked. The observed refresh penalty varies from 0.15 percent (*h264ref*) to 12.8 percent (*milc*). As can be seen in Fig. 7a, in every set of the results, the performance of CAR lies well in between *no-refresh* and *baseline*, but approaching closer to *no-refresh*. CAR improves the performance by a range of 0.16 percent (*namd*) to 9.4 percent (*libquantum*). More compressible and intensive applications benefit more from CAR. Table 5 gives a summary of refresh impacts and the efficiency of our approach in alleviating refresh overheads based on the benchmark classification (Table 4). The number in each cell denotes the average value across the benchmarks belonging to the same category. Second, zero cache line optimization can further improve CAR performance, particularly for zero-intensive applications. However, for applications which read few zero cache lines (e.g., *mcf*, *lbm* in Fig. 1b), zero cache line optimization does not bring noticeable additional performance improvement.

TABLE 4
Benchmark Characteristics and Classification

	Intensive	Non-intensive
High Compression	bwaves, cactusADM, libquantum, wrf, zeusmp, soplex canneal, fluidanimate	gromacs, namd, sphinx3, h264ref facesim, bodytrack
Low Compression	lbm, leslie3d, mcf, gcc, milc, perlbench, ferret	astar, bzip2, xalancbmk, gobmk, omnetpp



(a) Normalized IPC relative to the baseline system without zero cache line optimization.



(b) Normalized IPC relative to the baseline system with zero cache line optimization.

Fig. 7. Performance comparison of our approach with the baseline and no refresh systems. Each benchmark has four sets of results and the middle legend in each set represents CAR and labels the memory configuration for the set. With zero cache line optimization, CAR can improve performance as much as $1.66\times$ for *libquantum*, with an average of 11.7 percent. Without zero cache line, CAR can still outperform the baseline by a maximum of 9.4 percent for *libquantum*, with an average of 3.2 percent (It should be noted that the maximum and average refresh performance penalty is 12.8 and 4.4 percent, respectively).

For zero non-dominant applications, the performance improvements mainly result from refresh alleviations provided by CAR.

Refresh management schemes entail different policies and incur various overheads. To fairly compare CAR with the state-of-the-art approaches, we choose to compare our approach with Elastic Refresh [16] and CREAM [17]. For comparison, we use the metric of *Refresh Alleviation Efficiency*, which is defined as the ratio between the degree of refresh alleviation and the total refresh overheads (being the performance differences between an ideal no-refresh memory and an regular baseline refresh memory). This metric faithfully represents the alleviation efficiency of refresh schemes. Fig. 9 shows the comparison results of averaged alleviation efficiency across all benchmarks. As it is shown, both CAR and CREAM have better alleviation efficiencies than Elastic Refresh, while with our zero cache line optimization CAR also significantly outperforms CREAM. However, please note that CAR is advantageous in terms of implementation cost and complexity. Elastic Refresh requires a sophisticated prediction algorithm to anticipate forth-coming memory requests and also needs a carefully designed scheme for managing postponed refresh operations. CREAM needs the memory to support parallel architecture so that refresh operations and regular memory requests can proceed in parallel, affecting its generality. By contrast, our CAR only requires a lightweight cache lines compression/decompression engine (i.e., BDI algorithm), which imposes no changes to the memory and is highly portable.

TABLE 5
Refresh Penalty and CAR Alleviation Efficiency

	High Compression		Low Compression	
	Intensive	Non-Intensive	Intensive	Non-Intensive
Degradation	6.9%	2.0%	8.4%	1.7%
Improvement	5.5%	1.6%	5.6%	1.3%
Efficiency	78.1%	80.2%	66.3%	76.5%

4.3 Access Latency

To gain further insight on the efficiency of CAR's alleviating refresh overheads, in this section, we investigate individual memory request latencies. Refresh operations lengthen memory request service time if access conflicts occur, because refresh operations tie up memory resources and creates unavailability. The quicker refresh operations finish, the sooner memory resources become available. CAR reduces the time spent on refreshing memory rows and thus is able to shorten accesses latencies. Therefore, short memory request latency is indicative of reduced refresh overheads. Fig. 8 compares the average access latencies of the benchmarks in the three systems with different configurations. Again, compared to the *baseline* system, CAR reduces the memory request latency by up to 32.7 percent (for *namd*), with an average of 12.4 percent across all the benchmarks. The primary reason is due to refresh reduction which results in memory requests having a smaller probability of being blocked by refresh. Fig. 10 shows the average access latencies and their standard deviations across the benchmarks in different memory configurations. Two conclusions are in order. First, CAR reduces the average access latency by 11.1 , 10.4, 14.2, 13.5 percent for the four configurations, respectively, and approaching pretty closely to the values of the *no-refresh* system. Second, the average standard deviations across the four configurations are 175.2, 65.6, and 51.3 for baseline, CAR, and no refresh, respectively, implying CAR makes the memory system more stable and more consistent due to refresh reduction. In other

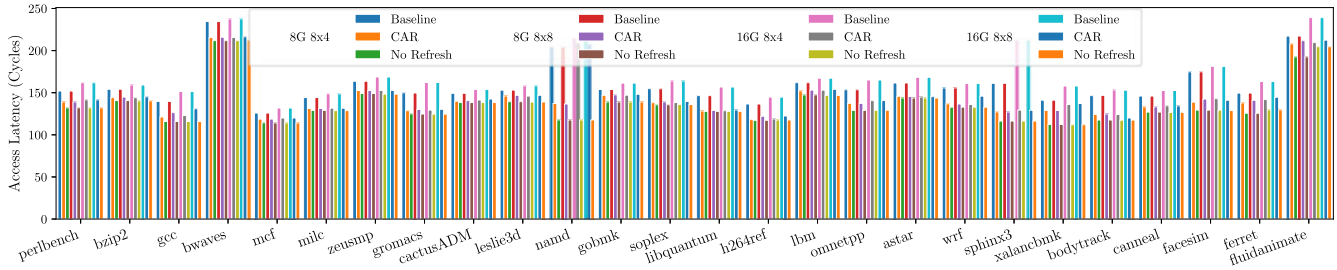


Fig. 8. Comparison of memory access latencies (in cycles) across different configurations without zero cache line optimization. CAR reduces memory request latency by an average of 12.4 percent relative to the baseline and approaches the latencies of the ideal *no-refresh* system.

words, CAR reduces latency variations and can deliver better QoS for applications which benefit from consistent memory performance.

4.4 Memory Traffic Reduction

As mentioned previously, we employ rank-subsetting to enable fine-grained accesses to compressed cache lines. In this section, we investigate the total memory traffic observed between the memory controller and memory chips. To evaluate the holistic efficacy of CAR, we enable zero cache line optimization to calculate memory traffic in this section and energy in the remaining sections. Fig. 11 shows the comparison of traffic reductions, which is normalized to the baseline. As it is shown, CAR significantly reduces memory traffic, achieving a maximum of 99.9 percent memory traffic reduction, with a geometrical mean of 66.1 percent across all the benchmarks. The memory reductions are primarily attributed to two reasons. First, the combination of memory compression and rank-subsetting results in reduced memory traffic. Second, accessing zero cache lines incurs no data traffic at all due to the employed zero cache line optimization. This figure also shows that the memory traffic reductions are insensitive to the memory configuration. We present the memory traffic results using different sub-ranks in Section 4.7.

4.5 Refresh Reduction

CAR skips refreshing memory rows containing no valid data. Fig. 12 shows the normalized number of refreshed rows. As it is clearly shown, CAR significantly reduces refreshed rows for the majority of the benchmarks. It eliminates up to 93.7 percent (for *libquantum*) of refreshed rows and achieves an average of 74 percent. As the same compressed cache line might occupy different number of chips in different memory configurations, the total amount of reduced refreshed rows varies with the memory configuration. Take *bzip2* for example, for an 8G memory, the percentage of refreshed rows of an 8×8 memory configuration is 63.7 percent of the baseline, and it goes down to 57.5 percent in a 16×4 memory configuration.

4.6 Energy Reduction

Energy efficiency is an important metric to evaluate memory systems, as energy consumption has become an increasingly critical concern in modern computer systems. As discussed so far, CAR significantly reduces refresh operations and memory traffic, which can translate to improved energy efficiency. Fig. 13 gives the energy comparisons of total energy, refresh energy, and burst energy. As it is shown, even though CAR needs to spend energy consumption on compression and decompression, on average, it cuts down 27.3, 74, and 62.6 percent of the total energy, refresh energy, and bursty energy, respectively. Refresh energy reduction is attributed to the decreases of refreshed rows as shown in Fig. 12, and burst energy reductions result from the saved memory traffic as shown in Fig. 11. Overall, the results have demonstrated that CAR is able to improve memory energy efficiency for all the applications in various memory configurations.

4.7 Sensitivity Study

In this section, we carry out two sensitivity studies on rank-subsetting and different DDR4 FGR refresh modes, respectively.Rank-subsetting affects how compressed cache lines are accessed and thus impacts memory traffic, bursty energy, and total energy. Table 6 gives the results of

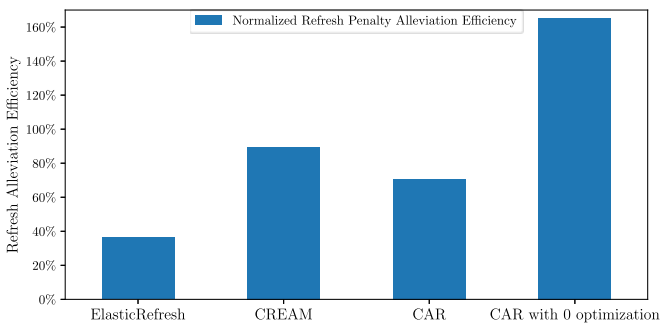


Fig. 9. Refresh alleviation efficiency comparisons among CAR, Elastic Refresh, and CREAM. Elastic Refresh, CAR, CREAM, and CAR with zero cache lines optimization achieve an alleviation efficiency of 36.4, 70.6, 89.3, and 165.6 percent, respectively.

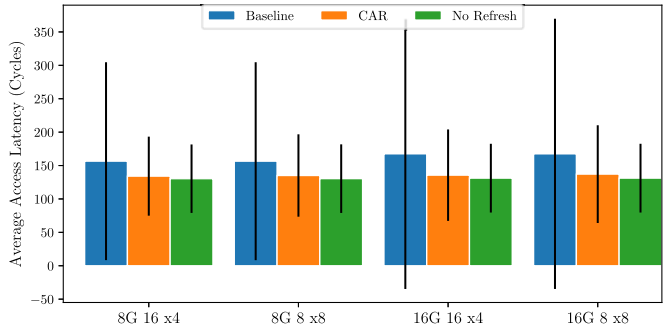


Fig. 10. Average access latencies and their deviations in different configurations. The bars compare the average access latencies and the error bars compare the standard deviations of access latencies. CAR reduces both average access latencies and standard deviations in all configurations.

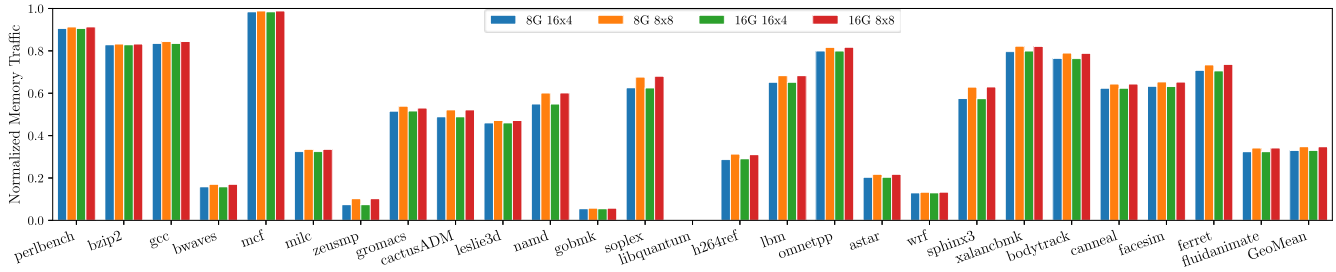


Fig. 11. Comparison of memory traffic. Values are normalized to baseline. As can be seen, in most scenarios, CAR significantly reduces memory traffic due to data compression and zero cache line traffic elimination.

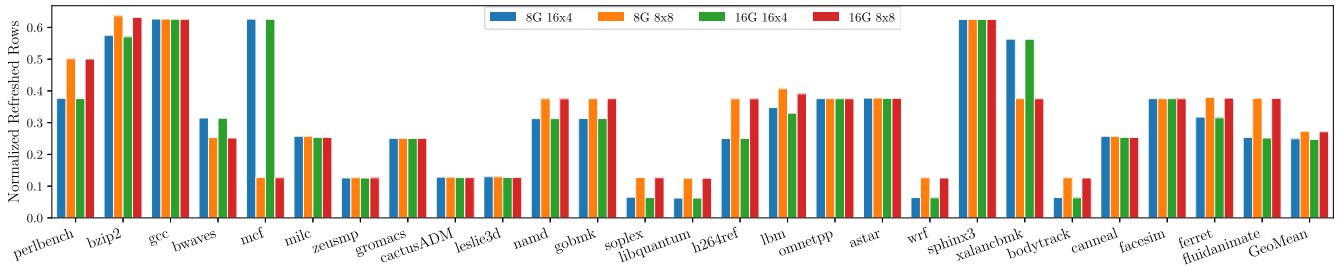
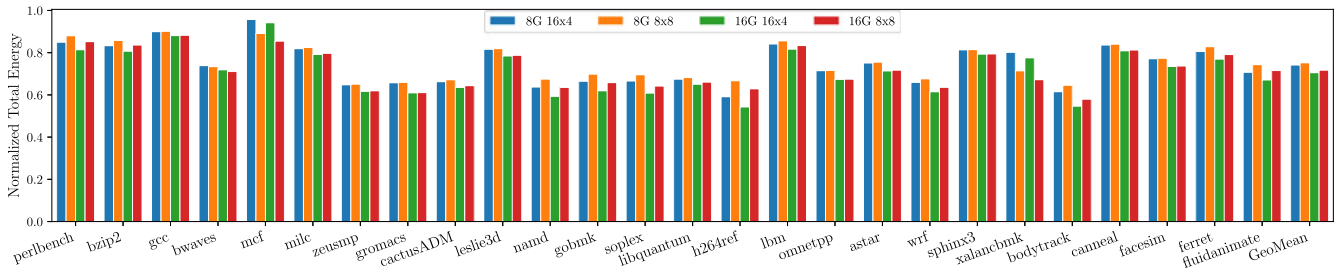
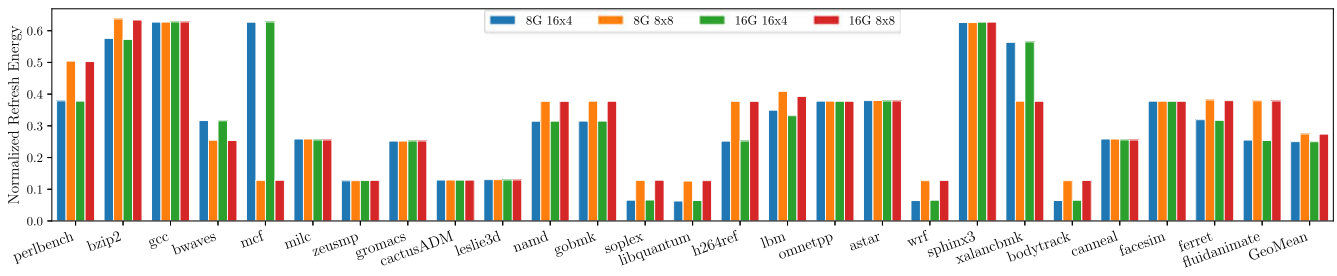


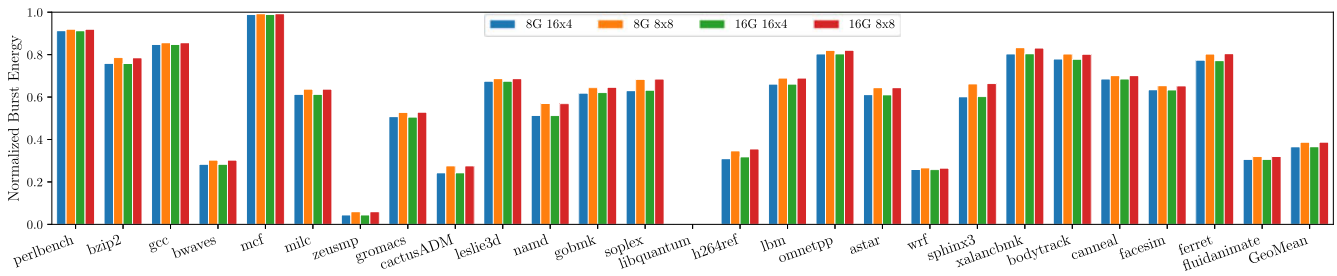
Fig. 12. Normalized refreshed rows relative to the baseline. CAR reduces 74 percent refreshed rows on average.



(a) Total Energy Consumption.



(b) Refresh Energy Consumption.



(c) Burst Energy Consumption.

Fig. 13. Energy comparison of the three systems for different benchmarks. Figs. 13a, 13b and 13c compare total energy, refresh energy, and burst energy, respectively. Values are normalized to that of baseline. On average, CAR reduces 27.3, 74, and 62.6 percent of total energy, refresh energy, and bursty energy, respectively.

sensitivity study on rank-subsetting. The percentage numbers are normalized to the numbers of the same memory organization partitioned to 2 sub-ranks. Please note that not all possible combinations in the table have been considered. For example, we do not divide an 8×8 configuration into 8 sub-ranks, as each sub-rank contains only one chip. We do

TABLE 6
The Impacts of Rank-Subsetting

	8G 8x8		8G 16x4		16G 8x8		16G 16x4	
subranks	4	8	4	8	4	8	4	8
Memory Traffic	32.2%	N/A	32.2%	29.7%	32.2%	N/A	32.3%	29.7%
Total Energy	74.7%	N/A	73.9%	73.5%	71.2%	N/A	70.3%	70%
Burst Energy	35.7%	N/A	35.6%	32.7%	35.7%	N/A	35.6%	32.7%

not present refresh energy in the table as refresh reductions only depended on data compression ratio. From this table, we know that as the number of sub-ranks increases, the memory traffic, total energy, and burst energy generally decrease due to fine-grained accesses to compressed cache lines. The reason is that having more sub-ranks allows the memory controller to activate less chips to access compressed cache lines and transfer less data, causing less memory traffic and access energy.

Fine Granularity Refresh (FGR) technique defined in DDR4 specifications, aims to alleviate refresh overheads by providing flexibility in choosing refresh duration and frequency. Fig. 14 shows the access latency and refresh energy comparison averaged across the benchmarks in different FGR refresh modes. The values under each refresh mode are normalized to the *baseline* system using the same refresh mode. As it is shown, CAR can consistently reduce access latency and improve energy efficiency across different FGR refresh modes, reducing up to 23 percent access latency and consuming only around 26.7 percent of the *baseline*'s refresh energy, demonstrating that CAR is effective in working with various DDR4 refresh modes.

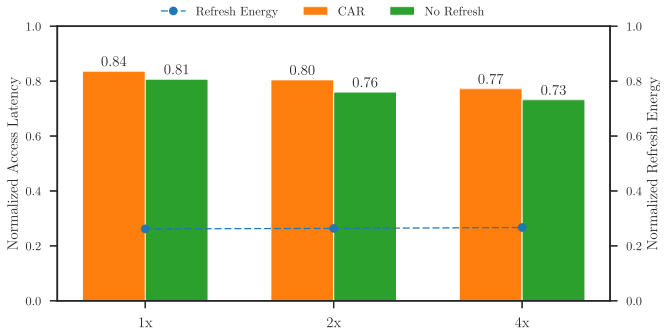


Fig. 14. The normalized access latency and refresh energy comparisons in different FGR refresh modes. CAR reduces access latency and refresh energy in all refresh modes.

5 Related Work

Refresh penalty is a critical problem in memory research. There exists a large body of research works proposed to alleviate memory refresh overheads. Prior research efforts can be broadly categorized into two types, *refresh hiding* and *refresh reduction*. As refresh operations affect system performance by blocking regular memory requests, refresh hiding aims to eliminate refresh blocking effects observed by memory requests via appropriate scheduling or parallelizing refresh and memory accesses. *Elastic Refresh* [16] postpones up to eight refresh operations in anticipation of memory requests possibly arriving soon after refreshes have been issued. If anticipated correctly, memory requests can be serviced before the refreshes, avoiding being blocked. *Refresh Pausing* [11] enables a refresh operation to be interruptible so that memory requests can be serviced when they arrive by temporarily pausing an on-going refresh operation and resuming the interrupted refresh after completing memory requests. Fine Granularity Refresh [30] is a newly proposed refresh management scheme defined in the JEDEC DDR4 standard. FGR allows refresh operations to be performed at different rates (e.g., 2x, 4x) to accommodate application patterns. Parallelizing refreshes and accesses is achieved by changing the memory architecture to support finer-grained refresh granularity, e.g., bank-level [18], sub-array level [20] and sub-rank level [17] so that other resources can be available to service accesses while refresh operations are being performed. Refresh reduction is primarily realized by eliminating unnecessary refresh operations based on data behaviors/characteristics or variable retention time. *Smart Refresh* [10] avoids refreshing rows which have been recently accessed as memory accesses implicitly perform refresh operations. *Variable retention time* [13], [14] phenomenon, i.e., memory cells exhibit widely varying retention time, provides good opportunities to reduce refresh operations. Based on variable retention time, *Flikker* [33] classifies data into critical and non-critical sections at the software layer and applies normal and slower refresh rates to those two sections, respectively. Similarly, *RAPID* [32] exposes the retention time to the operating system and prioritizes the allocation of pages having longer retention time so that the system can use a slower refresh. *RAIDR* [12] is a retention time aware refresh scheme implemented inside the memory system. It groups memory rows into different bins according to their retention time and applies disparate refresh rates to bins. As memory retention time changes dynamically, such multi-rate refresh approach can lead to unacceptable system reliability. To resolve that, *AVATAR* [9] proposes to switch back to the normal refresh rate for the memory rows which are refreshed at a slower rate and errors have occurred to. MECC [34] aims to reduce refresh power of the memory in mobile devices by leveraging the fact that mobile devices are idle most of the time. In this scheme, during idle time, it protects memory using 6-bit ECC and refreshes at an extended interval of 1 second, while during active time it returns to use single-bit ECC and refreshes at the normal interval of 64 ms. Different from the above mentioned efforts, our proposed approach achieves refresh savings from an unexplored direction, i.e., it reduces refresh operations by storing data in compressed format.

Data compression has been leveraged in memory systems to improve effective capacity and performance [36], [37], [38], [39]. LCP [22] attempts to fasten the procedure of locating a cache line when pages are compressed into variable sizes. In LCP, each cache line is compressed within a fixed size. If a cache line cannot be compressed within the boundary, it is identified as an *exception* and is stored in uncompressed form. A physical page is divided into three sections, which store compressed cache lines, metadata region and the exception region for uncompressed cache lines, respectively. Cache line locations can be quickly decided via linear calculations. MemZip [23] also employs data compression in the memory controller to store data in compressed format. However, its primary goal is not to

enlarge memory capacity, but to exploit unconventional benefits of data compression. It also relies on rank-subsetting to access a subset of chips to save bandwidth and energy but at the cost of multiple column reads for each cache line access, while our approach does not increase the number of column reads a cache line access. Another recent work [43] applies BDI algorithm in GPU to compress register file content written by a warp to achieve energy efficiency, as the register values written by the same warp instruction exhibit good value similarity. While data compression has been proved to improve system performance and energy efficiency in many previous works, a recent work [53] has observed that data compression might incur energy overheads due to the compression-caused bit toggles which consume extra dynamic transfer energy. Correspondly, the authors propose two new toggle-aware compression techniques to reduce energy consumption. Our work exploits data compression from a new perspective, to reduce refresh penalty which significantly affects performance and energy efficiency in contemporary memory systems.

6 Conclusion and Future Work

In this work, we propose a compression-aware memory refresh approach named CAR. To the best of our knowledge, this is the first work to leverage memory compression to achieve refresh savings. Cache lines are compressed to smaller sizes, creating sparse banks in which the amount of rows requiring refresh is significantly reduced. The overall implementation complexity of CAR is minimal because of the efficient compression algorithm and low space overhead. Evaluation results with SPEC CPU 2006 and PARSEC 3.0 benchmarks have demonstrated that CAR can remarkably improve memory performance and energy efficiency.

Our future work includes: (1) Compact the valid cache lines in sparse banks to further reduce rows that need refreshing. Our current implementation employs an aggressive policy which refreshes a row as long as it contains data from any cache lines belonging to the row. Our initial observations have revealed that most of the counter values in the second-level vector are far smaller than the maximum value, implying there still exist much space for refresh savings. However, compacting cache lines requires more implementation complexity and trade-offs need to be investigated. (2) Explore new ways to utilize the saved cache line space in the same sub-rank. In CAR, accessing the saved space does not impose additional performance overheads, as data in the space will be fetched in the same burst access. A possible way is to store more powerful ECC code to tolerate retention failures and improve reliability.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grants 61232004 and 61502189, the National Key Research and Development Program of China (No. 2016YFB0800402), and the US National Science Foundation (NSF) under Grant Nos. CNS-1702474, CNS-1700719, and CCF-1547804. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. A preliminary version of this work was published as a 2-page poster paper in the Proceedings of the 2016 ACM SIGMETRICS [1].

References

[1] W. Liu, P. Huang, K. Tang, K. Zhou, and X. He, "CAR: A compression-aware refresh approach to improve memory performance and energy efficiency," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Sci. Poster*, 2016, pp. 373–374.

[2] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 237–248.

[3] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation," in *Proc. 41st Int. Symp. Comput. Archit.*, 2014, pp. 349–360.

[4] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 14.

[5] V. Karakostas, et al., "Redundant memory mappings for fast access to large memories," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 66–78.

[6] I. Hur and C. Lin, "A comprehensive approach to DRAM power management," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, 2008, pp. 305–316.

[7] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "MemScale: Active low-power modes for main memory," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 225–238.

[8] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. 8th Int. Conf. Autonomic Comput.*, 2011, pp. 31–40.

[9] M. K. Qureshi, D. H. Kim, S. Khan, P. Nair, and O. Mutlu, "AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2015, pp. 427–437.

[10] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs," in *Proc. 40th ACM/IEEE Int. Symp. Microarchit.*, 2007, pp. 134–145.

[11] P. Nair, C.-C. Chou, and M. K. Qureshi, "A case for refresh pausing in DRAM memory systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 627–638.

[12] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 1–12.

[13] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 60–71.

[14] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study," in *Proc. Int. Conf. Meas. Model. Comput. Syst.*, 2014, pp. 519–532.

[15] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible auto-refresh: Enabling scalable and energy-efficient DRAM refresh reductions," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 235–246.

[16] J. Stuechelix, D. Kaseridis, H. C. Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 375–384.

[17] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, "CREAM: A concurrent-refresh-aware DRAM memory architecture," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 368–379.

[18] K. Chang, et al., "Improving DRAM performance by parallelizing refreshes with accesses," in *Proc. 20th Int. Symp. High-Perform. Comput. Archit.*, 2014, pp. 356–367.

[19] P. J. Nair, D.-H. K. Moinuddin, and K. Qureshi, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *Proc. 40th Annu. Int. Symp. Comput. Arch. (ISCA'13)*, 2013.

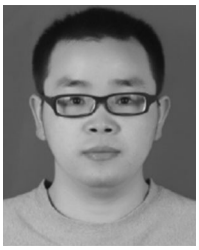
[20] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *Proc. 39th Int. Symp. Comput. Archit.*, 2012, pp. 368–379.

[21] 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 *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 377–388.
[22] G. Pekhimenko, et al., "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 172–184.
[23] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 638–649.
[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 *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2008, pp. 210–221.
[25] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 175–186.
[26] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *Proc. 19th Int. Symp. High-Perform. Comput. Archit.*, 2013, pp. 615–626.
[27] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 24–35.
[28] O. Seongil, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-buffer decoupling: A case for low-latency DRAM microarchitecture," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 337–348.
[29] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk memory access prediction and streaming," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 545–557.
[30] J. Mukundan, H. Hunter, K. H. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *Proc. 40th Int. Symp. Comput. Archit.*, 2013, pp. 48–59.
[31] P. Huang, W. Liu, K. Tang, X. He, and K. Zhou, "ROP: Alleviating refresh overheads via reviving the memory system in frozen cycles," in *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 169–178.
[32] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 155–165.
[33] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 213–224.
[34] C. Chou, P. Nair, and M. K. Qureshi, "Reducing refresh power in mobile devices with Morphable ECC," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2015, pp. 355–366.
[35] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proc. 33rd Annu. ACM/IEEE Int. Symp. Microarchit.*, 2000, pp. 258–265.
[36] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, Art. no. 212.
[37] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proc. 32nd Annu. Int. Symp. Comput. Archit.*, 2005, pp. 74–85.
[38] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 331–342.
[39] G. Pekhimenko, et al., "Exploiting compressed block size as an indicator of future reuse," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 51–63.
[40] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 Caches," Comput. Sci. Dept., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 1500, 2004.
[41] M. M. Islam and P. Stenstrom, "Zero-value caches: Cancelling loads that return zero," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, 2009, pp. 237–245.
[42] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proc. 23rd Int. Conf. Supercomput.*, 2009, pp. 46–55.
[43] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient GPUs through register compression," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 502–514.
[44] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–12.
[45] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Improving system energy efficiency with memory rank subsetting," *ACM Trans. Archit. Code Optimization*, vol. 9, no. 1, pp. 4:1–4:28, Mar. 2012.
[46] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Letters*, vol. 10, no. 1, pp. 16–19, Jan. 2011.
[47] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 475–486.
[48] C.-K. Luk, et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2005, pp. 190–200.
[49] Micron System Power Calculator. (2017). [Online]. Available: http://www.micron.com/support/power-calc.
[50] 8Gb: x4, x8, x16 DDR4 SDRAM. (2017). [Online]. Available: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf
[51] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Jan. 2011.

Ke Zhou received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1996, 1999, and 2003, respectively. He is a professor of the School of Computer Science and Technology and Wuhan National Laboratory for Optoelectronics, HUST. His main research interests include computer architecture, cloud storage, parallel I/O, and storage security. He has more than 50 publications in journals and international conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, the *Performance Evaluation (PEVA)*, FAST, USENIX ATC, MSST, ACM MM, INFOCOM, SYSTOR, MASCOTS, ICC, etc. He is a member of the IEEE and a member of the USENIX.

Wenjie Liu received the master's degree from Nanjing University, China, in 2013. He is currently working toward the PhD degree at Wuhan National Laboratory of Optoelectronics (WNLO), Huazhong University of Science and Technology, China.

Kun Tang received the BS and MS degrees in software engineering from Dalian University of Technology, China, in 2011 and 2013, respectively, and the PhD degree in computer science from Temple University, in 2017. He is currently a software engineer at Amazon. He is a student member of the IEEE.



Ping Huang received the PhD degree from Huazhong University of Science and Technology, in 2013. He is currently a research assistant in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His main research interest includes nonvolatile memory, operating system, distributed systems, DRAM, GPU, Key-value systems, etc. He has published papers in various international conferences and journals, including SYSTOR, NAS, MSST, USENIX ATC, Eurosys, IFIP Performance, INFOCOM, SRDS, MASCOTS, ICCD, the *Journal of Systems Architecture (JSA)*, the *Performance Evaluation (PEVA)*, the *Sigmetrics*, ICPP, the *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, the *ACM Transactions on Storage*, etc.

Xubin He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from the University of Rhode Island, Kingston, Rhode Island, in 2002. He is currently a professor in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His research interests include computer architecture, data storage systems, virtualization, and high availability computing. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE, a member of the IEEE Computer Society and USENIX.