TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/12 pp347–360
DOI: 10.26599/TST.2019.9010077
Volume 26, Number 3, June 2021

RBC: A Memory Architecture for Improved Performance and Energy Efficiency

Wenjie Liu, Ke Zhou, Ping Huang, Tianming Yang, and Xubin He*

Abstract: DRAM-based memory suffers from increasing row buffer conflicts, which causes significant performance degradation and power consumption. As memory capacity increases, the overheads of the row buffer conflict are increasingly worse as increasing bitline length, which results in high row activation and precharge latencies. In this work, we propose a practical approach called Row Buffer Cache (RBC) to mitigate row buffer conflict overheads efficiently. At the core of our proposed RBC architecture, the rows with good spatial locality are cached and protected, which are exempted from being interrupted by the accesses for rows with poor locality. Such an RBC architecture significantly reduces the overheads of performance and energy caused by row activation and precharge, and thus improves overall system performance and energy efficiency. We evaluate RBC architecture using SPEC CPU2006 on a DDR4 memory compared to a commodity baseline memory system. Results show that RBC improves the overall performance by up to 2.24× (16.1% on average) and reduces the memory energy by up to 68.2% (23.6% on average) for single-core simulations. For multi-core simulations, RBC increases the overall performance by up to 1.55× (17% on average) and reduces memory energy consumption by up to 35.4% (21.3% on average).

Key words: memory system; Dynamic Random Access Memory (DRAM); row buffer conflict

1 Introduction

The Dynamic Random Access Memory (DRAM)-based memory system is a vital component of all computing systems, ranging from small hand-held systems to large-scale cloud platforms. As the memory system provides a buffer to alleviate the huge gap between the performance required by the processor and the relatively dormant operations of the storage system, the memory system performance is critical to the overall system performance. With the scaling of data centers and web services, applications are becoming increasingly memory hungry, which makes the memory system have a significant impact on the overall performance and energy efficiency of the data centers. Due to the advancement of memory technologies, DRAM capacity and bandwidth have been dramatically improved across generations during the past decades[1,2]. However, memory latency remains almost constant across generations[3], exacerbating the performance gap between the memory system and the processor, which is known as the "memory wall" problem. Extensive recent researches have been conducted on mitigating the "memory wall" problem via reducing memory latency. Various factors can affect the memory latencies, such as DRAM refresh[4–7], row buffer overfetch[2,8], and interference between competing processes[9]. In fact, the increase in memory capacity can hurt memory latency[3]. The primary reason is that the bitlines connected to the row buffer become longer as capacity grows, which leads to dramatically increased associated parasitic capacitance,

• Wenjie Liu, Ping Huang, and Xubin He are with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: wenjie.liu@temple.edu; templestorager@temple.edu; xubin.he@temple.edu.
• Ke Zhou is with the Wuhan National Laboratory of Optoelectronics (WNLO), Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: k.zhou@hust.edu.cn.
• Tianming Yang is with Huanghuai University, Zhumadian 463000, China. E-mail: ytmzqyy@163.com.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-12-09; accepted: 2019-12-19

and thus increases DRAM access latency[2,3]. This paper focuses on reducing the latencies caused by row buffer overfetch[10,11], which is one of the most performance and energy inefficient DRAM operations in modern multi-core systems.

DRAM access starts with locating data from thousands of rows according to the row address, and then sends the entire row to the row buffer via bitlines. After the row buffer senses the charges stored in the cells, 64 bits are transferred from the row buffer to the data bus according to the column address. Once the request is completed, the data temporarily stored in the row buffer is either waiting for next memory requests targeting at the same row or written back to the corresponding row according to the row buffer policy, which is known as "open page" or "close page" row buffer policy[12], respectively. The size of the row buffer is typically 4 KB or 8 KB. However, a recent study[13] shows that less than 4% of the data in the row buffer is accessed before the row buffer is closed in modern multi-core systems, which implies the underutilization of the row buffer. The low utilization of the row buffer can be attributed to the row buffer conflict between consecutive memory requests that target at different rows. The overheads of row buffer conflict are expensive, as the row buffer conflict introduces additional latencies and energy consumption by invoking row precharge and activation operations. With the scaling of DRAM[14], the overheads caused by row buffer conflict are nontrivial, due to the almost constant memory access latencies.

To improve the row buffer utilization, various approaches have been proposed. Some works suggest to maximize the utilization of the row buffer by scheduling[2], or group the memory requests by the targeting rows at the memory controller[8]. Other works try to mitigate the latencies by taking advantage of DRAM characteristics that accessing the rows near the row buffer incurs less latencies[3]. Similarly, the half-page[8] takes advantage of the new feature of DDR4[15] to reduce the granularity of row activation and precharge, resulting in more valuable data being kept the row buffer.

In this paper, we suggest a new row buffer conflict mitigation architecture, i.e., a new Row Buffer Cache architecture, called RBC. The central idea is motivated by the critical observation on how row buffer conflict affects memory performance. We propose to add an SRAM-based cache in the memory controller to cache the rows with good locality but frequently interrupted by the requested rows with poor locality. RBC essentially enables parallel access to both poor locality rows (in the row buffer) and good locality rows (in the cache) without the potential risk of row buffer conflict, leading to significant performance improvement. Our evaluation results have shown that RBC architecture significantly alleviates the row buffer conflict overheads in modern memory systems.

This paper makes the following contributions:

- We present a comprehensive analysis of the row buffer interference problem from different angles. The findings motivate the design of our proposed row buffer conflict alleviation approach.
- We design a new RBC architecture, which caches the contents of the row buffer into an SRAM cache in the memory controller, such that the locality can be preserved even in the case of a row buffer conflict occurrence.
- We conduct extensive evaluations with benchmarks from SPEC CPU2006. The results have shown that our system successfully alleviates row buffer conflict overheads, improving both performance and energy efficiency relative to the baseline memory system.

2 Background and Motivation

In this section, we first discuss the basics of DRAM and the role of row buffer in DRAM operations, followed by the possible interference existing in modern DRAM memory systems, as our work primarily relies on the reduction of row buffer conflict to address the interference problem.

2.1 DRAM basics

Modern memory systems are commonly built with DRAM technology. As shown in Fig. 1, DRAM-based memory systems are constructed hierarchically: channels, ranks, and banks from high to low levels. A memory system consists of multiple channels, and each channel is controlled by a separate memory controller. Each channel can support multiple ranks, and a rank is a collection of DRAM chips that work in concert to either receive commands from memory controller or feed data to the data bus. Inside the DRAM chips, multiple

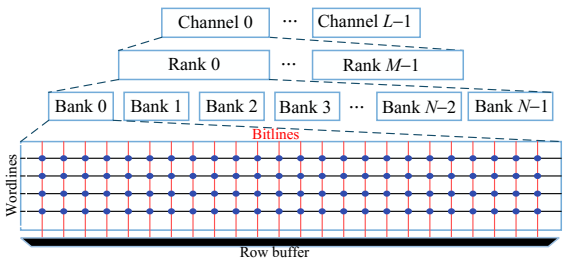


Fig. 1 Overview of memory organization.

banks are included, each of which is a two-dimensional array of DRAM cells. The cells are connected both vertically and horizontally through the bitline and wordline, respectively. The wordline connects all cells on the same row while cells on the same column are connected through the bitline. As the DRAM cell cannot be accessed directly, a sense amplifier is placed at the end of each bitline to amplify the charges in the corresponding DRAM cell through bitline, which composes the row buffer.

DRAM access follows two steps. First, the decoded row address is sent on the address bus, in the meanwhile, an activation command (activation) is sent on the command bus. In response, the data in the corresponding row is transferred to the row buffer through bitlines, which is the so-called row activation operation. Next, the column address is sent on the address bus followed by a column access command, i.e., a column-read (col-rd) command, and the data in the corresponding column can be accessed. After the column access command, the row buffer policy determines whether the data cached in the row buffer should be written back to the corresponding row or kept for the subsequent memory requests. The following access that requests data from an already opened row results in a row buffer hit and escapes the overheads introduced by additional row activation and precharge operations results in better performance. On the contrary, if the subsequent request targets a different row, the row buffer conflict happens, which increases the access latency of the latter request. Apparently, workloads with good locality could significantly benefit from the "open page" policy, while workloads with poor locality prefer the "close page" policy.

2.2 Role of the row buffer

Since the DRAM cells cannot be accessed directly, the row buffers are employed as the front-end of the DRAM-based memory system. All DRAM operations are related to the row buffer, which can be classified into the following three categories. First, data transmission between cells and the row buffer. Both activation and precharge operations are required to access the designated data, since the row activation operation brings data to the row buffer for data access while the precharge operation closes the row and writes the data back to the corresponding row. Second, data access from the data bus. As the row buffer caches the requested data, the column-read and column-write operations are used to get/put data from/to the row buffer. Third, the DRAM refresh operation needs the row buffer. Typically, the DRAM refresh operates at row granularity, which brings the data into the row buffer with an activation operation and writes the data back with a precharge operation. During the refresh, the row buffer is occupied by the refresh operation, and no memory access is allowed, which increases the memory services latency dramatically.

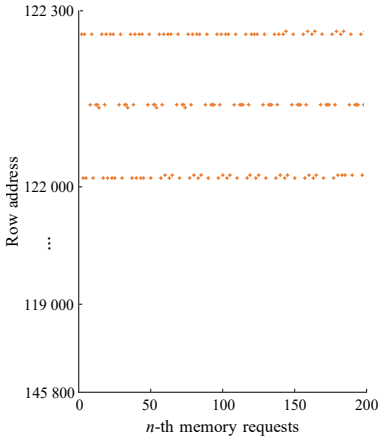
Due to the critical role of the row buffer, the row buffer utilization determines the memory system performance. An important problem that affects the row buffer utilization is the row buffer conflict. A row buffer conflict happens when two consecutive memory requests target at different rows, then the row which served for the prior request should be written back and brings the requested row into the row buffer. As a result, the latter request suffers from additional latency of both precharge and activation operations caused by the row buffer conflict. Moreover, the precharge and activation operations triggered by the row buffer conflict increase energy consumption. A variety of techniques[8, 9, 16–18] have been proposed to address the row buffer conflict problem by improving the row buffer utilization. Shao and Davis[16] proposed to group the requests targeting the same rows to increase row buffer hit rate and utilization. A page migration method is proposed to collocate frequently accessed data in the same row buffers to increase row buffer hit rate[17]. Similarly, Seshadri et al.[18] enabled the memory controller to access multiple values that belong to a stridden pattern using a single read/write command. The TCM addressed this problem by grouping the applications by its characteristics of memory access, and the spatial locality is preserved[9]. Also, to reduce the granularity of memory operations, Half Page is leveraged to enable the row buffer to contain data from two rows[8].

2.3 Row buffer interference

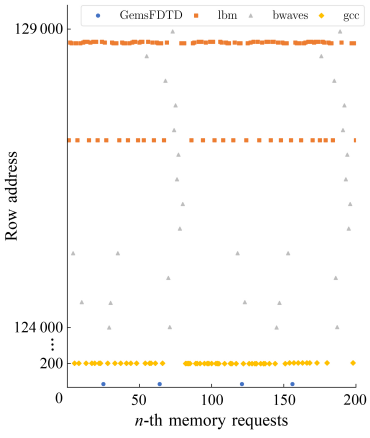
As the row buffer is on the critical path of many DRAM operations, the memory system performance depends on the row buffer utilization. However, the row buffer interference leads to low row buffer utilization. The row buffer interference occurs when two consecutive memory requests target at different rows, which results in additional row precharge and activation commands. Due to the extra row precharge and activation commands, the latter memory request suffers from increased access latency and consumes more energy during this process. Besides, as the t_{FAW} timing constrain defined by the DDR specification[15] limiting only four activation commands

can be issued during t_{FAW} memory cycles, the additional row activations caused by row buffer conflict may introduce extra damage to the system performance. With the scaling of DRAM and the more memory-hungry applications, the overheads of the row buffer interference are not trivial and become unbearable to the entire system.

The row buffer interference is not rare in today's applications as the structure of today's applications becomes increasingly complicated, which consists of a lot of loops or jump statements. From the view of the row buffer, the loop statement behaves as repeated accesses to several rows, which results in frequent row buffer conflict, while the jump statement changes the flow of execution and hurts the locality of the row buffer. Figure 2a shows a slice of the consecutive memory requests



(a) lbm



(b) WL1

Fig. 2 A snapshot of memory requests for lbm and WL1, which are benchmarks, details are introduced in Section 4. (a) lbm shows strong row buffer interference within the benchmark and (b) four kinds of dots represent four benchmarks in WL1. Due to the varying characteristics of benchmarks, WL1 suffers from severe row buffer interference.

from lbm. As shown, memory request of lbm shows a loop pattern which is always interrupted by the jump statements, as the requests jump from row 122 173 to row 118 926. As a result, the row buffer suffers from the interference during runtime. To better demonstrate the characteristic of the rows during runtime, Fig. 3 shows the percentage of the rows with different characteristics. As shown, all rows are classified as three groups: interference-free, interference-victim, and interference, based on the spatial locality and row buffer conflict of each row. Rows with poor locality usually lead to row buffer conflict and are labeled as interference. For rows with good locality, the interference-free rows are rows seldom interfered by the row buffer conflict, while the interference-victim rows are severely impacted by the row buffer confliction. According to Fig. 3, only 12.8% of rows accessed can avoid the row buffer conflict, while 29.2% of them are identified as interference-victim due to the severe row buffer interference.

The situation is getting worse in the scenario of multiprogramming workloads, as the applications not only suffer from their intra-interference but also from the interference caused by other applications. As a result, the application with good spatial locality could be interrupted by another application with poor locality, which results in severe aggregated row buffer interference. Besides, the application with few memory requests is delayed by the memory requests issued by applications with good locality, which leads to additional queuing latency. To demonstrate the row buffer interference in the scenario of multi-programmed workloads, Fig. 2b shows a slice of consecutive memory requests during the execution of WL1. As shown, the intermingled memory requests queue seems chaos due to the existence of row buffer interference. The memory requests of both gcc and GemsFDTD are

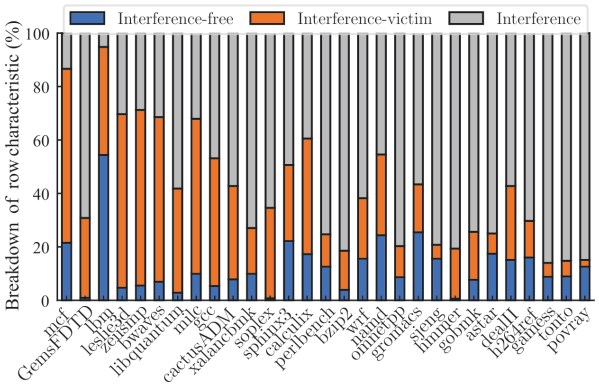


Fig. 3 Row characteristic breakdown for various benchmarks (details of benchmarks are shown in Section 4).

continuously accessing the same row, which exhibits good locality. However, such locality can be easily interfered by the memory requests issued by lbm and bwaves, as either lbm or bwaves accesses multiple rows and shows poor locality during run-time. The row buffer interference jeopardizes the performance and energy efficiency improvements brought by the spatial locality via introducing additional activation and precharge operations, which lead to in-negligible degradation of performance and energy efficiency.

2.4 Motivation

Due to the critical role of the row buffer, we propose to alleviate the row buffer conflict and protect the rows with good locality by caching them into an SRAM-based RBC. By doing so, the requests of rows with good locality are served with the low latency RBC and escape from the row buffer confliction. To motivate our approach, experiments are conducted to investigate the impact of RBC capacity on the system performance (in terms of Instructions-Per-Cycle (IPC)) and energy efficiency. Figure 4 shows the trends of both performance and energy efficiency when the RBC capacity increases. As shown, both system performance and energy efficiency are significantly improved with the increase of the RBC capacity. The improvement is attributed to the reduction of row buffer conflict, as the RBC reduces the additional activation and precharge operations introduced by the row buffer conflict via caching DRAM pages with good locality. With the increasing of RBC capacity, both performance and energy efficiency improvement trends can be divided into three segments, as shown in Fig. 4. In Segment I, both performance and energy efficiency are significantly improved by the newly added RBC, which shows significant relaxation of the row buffer confliction. However, the improvement pace slows down in Segment II and almost stays constant in Segment III. The main reason for this can be attributed to the limited work-set size during run-time, as the application does not require to access all the rows at the same time. For Segment I, the added RBC significantly improves both performance and energy efficiency by reducing row buffer conflict and accommodating more DRAM pages. As the size of work-set changes during run-time, improvements can still be observed with the increasing of RBC capacity, as shown in Segment II. For Segment III, the RBC capacity is larger than the work-set of the running application, which implies the improvement up-bound for both performance and energy efficiency. Based on the trends of improvement, the RBC architecture leverages the benefits enabled by the added RBC to protect the locality and reduce row buffer conflict to achieve the improvement of both system performance and energy efficiency.

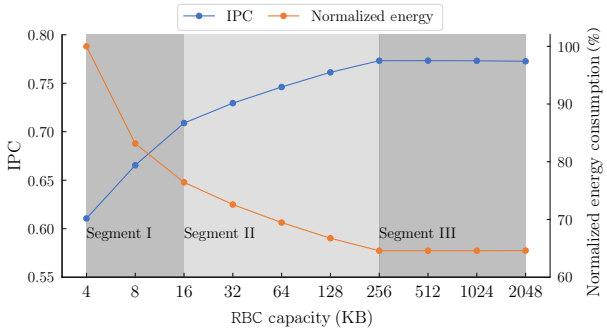


Fig. 4 Improvement trends of the performance and energy efficiency when the RBC capacity increases.

3 Design of RBC

In this section, we present the detailed design of the proposed RBC architecture. To maximize the benefits, several key components are introduced, and their interplay is demonstrated.

3.1 Overall architecture

Figure 5 shows a simplified schematic view of our proposed RBC architecture. As shown in Fig. 5, an RBC is added in the memory controller to hold the data with severe row buffer interference. The size of the RBC is determined by the size of a DRAM page and the number of banks in the memory systems. To better exploit the potential benefits enabled by the RBC, several key components are added to guide the execution of the proposed RBC architecture. To identify the interference-victim, the row buffer interference detector is designed to quantify the severity of row buffer interference of each row. Also, the quantified row interference is referenced by the row buffer locality predictor to

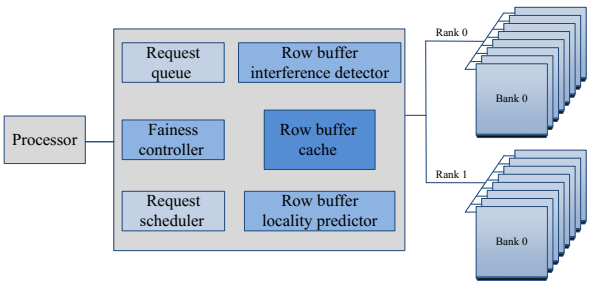


Fig. 5 Simplified architectural view of RBC.

evaluate the potential performance gain for each RBC swap-in candidate. By doing so, the rows initially profoundly affected by the row buffer interference can preserve their spatial locality and therefore improve the memory system performance. Additionally, the fairness allocator is added to ensure a fair allocation of the RBC, which improves the fairness in the multi-programming scenario.

3.2 Row buffer interference detector

The row buffer interference detector detects the row buffer interference for the running applications by using a sliding observation window, which records the row address of the last N memory requests. To accurately detect the row buffer interference, the row buffer interference detector monitors the frequently accessed DRAM rows in the observation window and uses two metrics: row buffer access rate and access count, to quantify the severity of row buffer interference. According to Section 2, each memory request can access the data until the requested data is transferred to the row buffer via an activation operation. Thus, the number of memory requests served by a single activation operation can effectively monitor the row buffer interference, as an additional activation operation is issued when the row buffer conflict happens. Based on this, the row buffer access rate is defined as the ratio between the number of memory requests served and the number of activation operations. The minimum value of the row buffer access rate is 1, as each memory request needs at most one activation operation. By comparing the row buffer access rate of the frequently accessed rows with the average row buffer access rate of the corresponding application, the rows can be clearly divided into two categories: interference-free (with higher row buffer access rate) and possible interference-victim (with lower row buffer access rate). Intuitively, the spatial locality of the rows labeled as interference-free does not severely impacted by the row buffer interference and can be exempted from the candidate of the row buffer locality predictor. However, the row buffer access rate is insufficient to distinguish whether a row is interference-victim or interference. To distinguish the rows with low row buffer access rate, the access count is employed to identify the row buffer interference-victim as the row interfered by others that can be accessed for multiple times. The access count identifies the interference-victim based on the property that the interference-victim has higher spatial locality and can be accessed for multiple times. Thus, for the row labeled as interference-victim, higher access count indicates more significant row buffer interference it suffered. With both row buffer access rate and access count, row buffer interference detector quantifies the severity of row buffer interference for each row, and classifies the row with high row buffer access rate as interference-free. For the row with low row buffer access rate, the product of row buffer access rate and access count is used, and rows with higher value are labeled as interference-victim while the rest as interference.

3.3 RBC

The RBC is an SRAM-based cache within the memory controller and lays at the core of the entire RBC architecture, which provides a cache to protect the spatial locality of the interference-victim, as shown in Fig. 6. The RBC is set-associated, and the set size is determined by the number of cache lines within a DRAM page. To manage the cache, the RBC metadata uses the row address as the tag along with a valid bit for quick detection of whether a designated cache line is cached or not. Also, the metadata includes the row buffer interference statistics for each cached row. To maximize the benefits enabled by the RBC, two major management components: the row buffer locality predictor and the fairness allocator, are proposed to manage the cache replacement and cache allocation.

3.3.1 Row buffer locality predictor

After the row buffer interference detector quantifies the interference severity of all the frequently accessed rows during the sliding window, the row buffer locality predictor begins to look for the candidate row to perform cache replacement. For each opened row, the row buffer locality predictor compares the interference severity between the row temporarily buffered by the row buffer and the least accessed row resides in the RBC belonging to the same application. By comparing the row buffer interference severity of the two rows, the row buffer

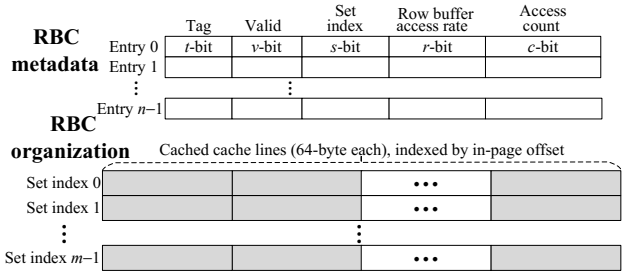


Fig. 6 Organization of RBC.

locality predictor can determine if the opened row suffers from more severe interference. The reason for doing this lies in two observations. First, the rows with low row buffer access rate and high access count are severely impacted by the row buffer interference. By caching such rows in the RBC, the row buffer interference can be significantly reduced, which alleviates the overheads caused by the row buffer confliction. Second, high interference severity indicates the potential high row buffer access rate when the spatial locality is well preserved by RBC.

When the row buffer locality predictor finishes the interference severity comparison, the RBC evicts the cached row with less interference and brings in the replacement candidate. Since the RBC architecture proactively forwards write requests to DRAM, the evicted row can be discarded. For the row that needs to be swapped into the RBC, eight consecutive cache lines are transferred to the RBC. The reason for transferring only a portion of the row to the RBC lies in two aspects. First, to bring the entire row into the RBC incurs significantly increased column access operations. Existing research has shown the access frequency of each cache line within one page can be highly imbalanced[13], which suggests the potential inefficiency of caching the entire row into RBC. Second, transferring only eight consecutive cache lines balances the overhead between caching the entire row and RBC miss, as the row chosen by the row buffer locality predictor can be accessed for multiple times. Even for the worst case that each cache line in the page frame needs to be accessed, the total transfer is only eight times.

3.3.2 Fairness allocator

As the RBC protects row buffer locality from the row buffer interference, the RBC hits significantly improve performance and energy efficiency by reducing unnecessary activation and precharge operations. However, the benefits brought by the RBC could be diminished in the multi-programmed workload scenarios, as the RBC may not reasonably be allocated. Suppose a workload contains two applications, A and B. If the RBC is not reasonably allocated between A and B, the fairness of execution can be harmed as the application with more RBC allocated suffers from less row buffer interference. To ensure a fair allocation of the RBC, the fairness allocator is designed to manage RBC space allocation. The fairness allocator divides the RBC into two parts: the fairness-oriented section and the performance-oriented section. For fairness, each application can be allocated a mandatory quota in the fairness-oriented section to ensure the severely interfered rows can be cached into the RBC. As can be inferred from Fig. 4, the increasing number of cached rows significantly improves both performance and energy efficiency. Thus, the basic quota allocated in the fairness-oriented section not only ensures fairness but also improves system performance and energy efficiency. Meanwhile, the performance-oriented section is allocated to each application in a round-robin manner, which maximizes the performance and energy efficiency of the application by caching more rows in the RBC. With the combination of the two RBC allocation policies, the fairness allocator protects the fairness and improves system throughput.

3.4 Read&write and overhead analysis

Due to the presence of the RBC, read and write operations need to be changed accordingly. To serve memory requests, the memory controller first checks if the requested row is cached in the RBC by comparing the row address with the tag of the RBC metadata. For the requested row resides in the RBC, the corresponding valid bits are checked to find out if the requested cache line is cached or not. If the requested cache line is invalid, the request is forwarded to DRAM, and the requested cache line along with the eight consecutive cache lines are transferred to the RBC to serve future memory accesses. If the requested cache line is valid, an RBC hit occurs, and the RBC handles the request. For read requests, the data is directly served by the RBC with low latency enabled by the characteristic of SRAM. For write requests, the modified data is directly written into the data entry in the RBC; also, the write request is forwarded to DRAM to keep the data consistent. Otherwise, if the requested row is not cached in the RBC, the request is forwarded to the DRAM and proceeded like a typical memory request.

Besides the potential run-time overhead, the RBC architecture incurs the storage overhead caused by the RBC. Suppose the DRAM page size is 4 KB for an RBC architecture-based memory system. Then, the capacity of an entry in the RBC is also 4 KB. According to Fig. 4, both performance and energy efficiency improve with the increased RBC capacity, while the improvements slow down when the row buffer capacity is larger than four times of the DRAM page size. Then, the total storage overhead of the RBC is $4 \times 4\,\text{KB} = 16\,\text{KB}$ for

the single-core scenario. For the multi-core scenario, assuming there are n cores within the system. As the fairness allocator requires additional capacity for the performance-oriented section, then the total storage overhead of the RBC is $(n+1)\times 4\times 4\,\text{KB} = 16\times(n+1)\,\text{KB}$.

4 Evaluation

In this section, we conduct experiments to evaluate the RBC architecture. We first introduce our evaluation methodology, including experimental testbed, related simulation parameters, and workloads. Then we present and discuss the evaluation results of various metrics.

4.1 Experimental methodology

We implement the RBC architecture in the popular DRAMSim2[19] simulator. We use the Zsim[20] simulator to run benchmarks. The processor simulator is coupled with the Pin[21] tool as the front-end. Benchmarks from the SPEC CPU2006 suite are used for evaluations. The benchmarks include both memory-intensive (marked as bold) and non-intensive applications by using the Misses Per Kilo-Instructions (MPKI) as the metric, as shown in Table 1. We conduct both single-core and multi-core evaluations. For single-core evaluations, only one benchmark runs at a time. For multi-core evaluations, four benchmarks run concurrently on a 2-rank memory. To ensure the diversity of workloads, we use six benchmark combinations, which represent a diverse mixture of the memory-intensive and non-intensive benchmarks, as shown in Table 2. The

Table 1 Benchmark groundtruth (bold for memory-intensive).

Benchmark	MPKI	Benchmark	MPKI	Benchmark	MPKI
mcf	**57.69**	xalancbmk	9.88	sjeng	4.11
GemsFDTD	**54.99**	soplex	8.59	hmmer	3.86
lbm	**39.24**	sphinx3	8.43	gobmk	3.39
leslie3d	**36.13**	calculix	8.01	astar	3.27
zeusmp	**35.47**	perlbench	7.60	dealII	2.99
bwaves	**34.92**	bzip2	7.37	h264ref	2.91
libquantum	**32.25**	wrf	7.15	gamess	2.12
milc	**24.04**	namd	6.47	tonto	1.34
gcc	**22.87**	omnetpp	6.39	povray	1.10
cactusADM	**11.72**	gromacs	5.60		

Table 2 Multi-core workloads configuration.

Workload	Benchmark
WL1	**GemsFDTD**, **lbm**, **bwaves**, **gcc**
WL2	**bwaves**, **gcc**, **libquantum**, **cactusADM**
WL3	**libquantum**, **cactusADM**, wrf, bzip2
WL4	**bwaves**, **gcc**, perlbench, astar
WL5	wrf, bzip2, perlbench, astar
WL6	perlbench, astar, omnetpp, gobmk

system parameters are given in Table 3, with the RBC parameters obtained by using the CACTI 5.3 tool[22]. Micro power calculator[23] is used to estimate memory power consumption. As inferred from Fig. 4, the performance improvements are slowing down when the row buffer number is larger than 4. Thus, the capacity of the RBC is set to 4 entries (each entry of the RBC is the same size as the DRAM page, which is typically 4 KB) for single-core and 20 for multi-program. Unless otherwise noted, the size of the Last Level Catch (LLC) is set to 2 MB. For each benchmark, we fast-forward 1 billion instructions for cache warm-up and run 1 billion instructions for comparison. For comparison, we compare our proposed RBC architecture with the state-of-the-art method, DICE[24], which focuses on improving the row buffer utilization. In addition, we also compare the RBC architecture with the state-of-the-art Bingo prefetcher[25].

4.2 Single-core experimental result

4.2.1 Performance improvement

Figure 7 compares the performance improvement in the single-core configuration using IPC as the metric. As shown, the proposed RBC architecture achieves a performance improvement of up to 2.24× (16.1% on average) which outperforms the state-of-the-art methods (the DICE improves performance up to 1.49× with 7.5% on average and the Bingo improves performance up to 2.33× with 10.5% on average). The reason that RBC architecture outperforms the two state-of-the-art methods is the RBC utilizes the spatial locality of the catched catch lines to alleviate the row buffer confliction, while two existing approaches depend on either cache line compressibility or memory access patterns. Also, as the DICE is based on cache line compression, we

Table 3 Simulation parameter.

Processor	Memory controller	DRAM	RBC
Single core/four cores	64/64-entry read/write request queue, writes are scheduled in batches	DDR4, 1 channel, 2 ranks per channel, DRAM page size is 4 KB	RBC read/write energy=0.0152 nJ, RBC access latency=5 cycles, Capacity=4 entries

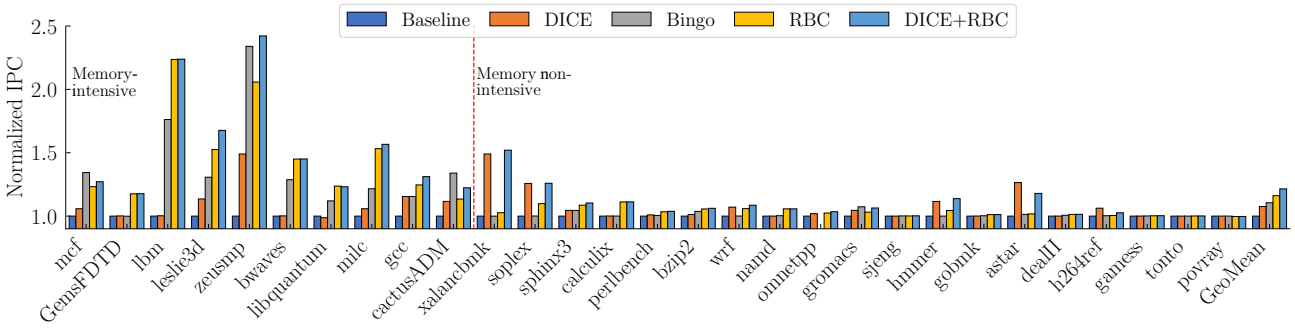


Fig. 7 Performance comparison in singe-core scenario. Values are normalized to the baseline.

integrate DICE with the RBC architecture and achieve up to 2.42× (21.5% on average) performance improvement, which shows our proposed RBC architecture can be used with other methods targeting the row buffer utilization problem. Additional, two conclusions can be drawn from Fig. 7. First, the benchmarks with more memory requests benefit a lot from the RBC architecture. To demonstrate the relationship between the memory intensity and performance improvement, the benchmarks in Fig. 7 are sorted by the memory intensity. For the memory-intensive benchmarks, the performance has improved up to 2.24× (44.4% on average). For those memory non-intensive benchmarks, the performance improvements are less significant, only up to 9.8% (3.5% on average). The reason behind this is the memory-intensive workloads suffer from greater row buffer interference, according to Section 2. Also, more memory access results in higher access count in the row buffer interference detector, which enables more performance improvements from the RBC. Second, the performance improvement comes from the RBC hit, as accessing the RBC incurs less access latency. Second, the RBC hits contribute a lot to the system performance improvements, as those benchmarks with higher RBC hit rates achieve better performance, as shown in Fig. 8.

Figures 8 and 9 have shown RBC hit rate and the average memory access latency comparison. As shown in Fig. 8, the RBC architecture delivers an RBC hit rate of up to 66.1% with 38% on average, which

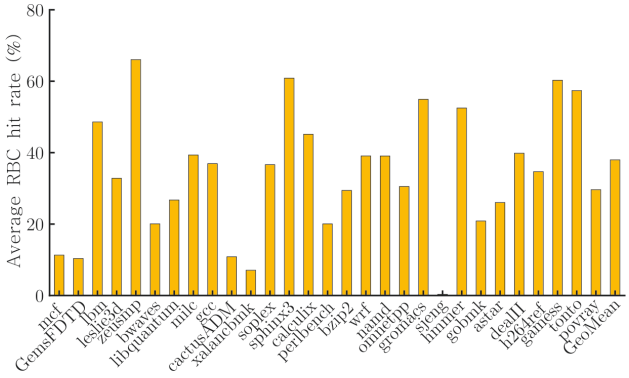


Fig. 8 Average hit rate of the RBC in single-core scenario.

indicates the row buffer interference detector is able to identify the interference severity of the accessed rows and the row buffer locality predictor is able to cache the most benefitial rows into the RBC. As the RBC servers 38% of the memory requests on average, the average memory access latency is significantly reduced by the RBC architecture by up to 84% with 50.8% on average, as shown in Fig. 9. The reason for the reduced memory access latency is due to the hits of RBC, which serves memory requests with the low latency SRAM-based RBC and avoids additional DRAM activate and precharge operations caused by row buffer conflict. Additional, the memory request latency of the RBC architecture is more stable as the standard variations drop up to 64.1% (22.3% on average), which provides more predictable performance.

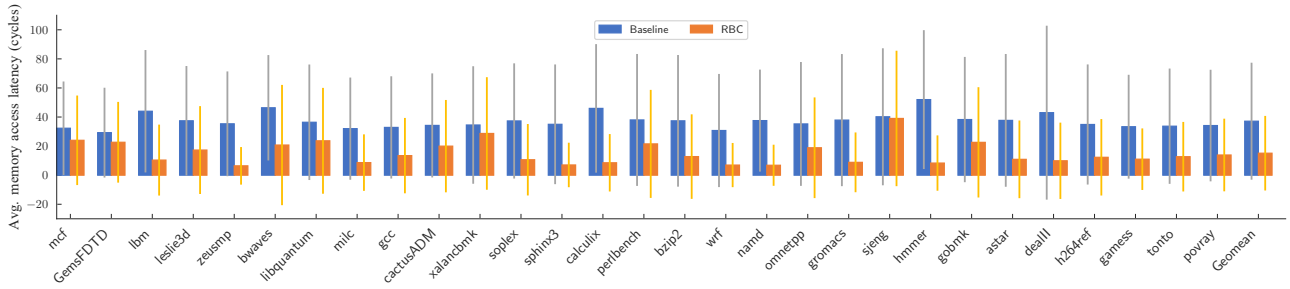


Fig. 9 Memory access latency comparison in singe-core scenario.

4.2.2 Energy consumption

Figure 10 shows the memory system energy comparison. As shown, the RBC architecture reduces the total amount of memory energy consumption relative to the baseline memory system. Similar to the performance improvements, the memory-intensive benchmarks reduce more energy than those non-intensive benchmarks, as up to 68.2% (42.2% on average). The saved energy consumption comes from two aspects. First, the energy reduction comes from the reduced row buffer conflict overheads. Due to the protection of spatial locality offered by the RBC, the row buffer interference is significantly reduced, which avoids unnecessary DRAM activate and precharge operations caused by the row buffer confliction. Second, the energy reduction comes from the reduced dynamic energy, as the RBC architecture improves the system performance and reduces the overall running time.

4.3 Multi-core experimental result

4.3.1 Weighted speedup

To evaluate the effectiveness of our proposal, the Weighted Speedup (WS) is employed to measure the performance improvement in the multi-core scenario. Figure 11 compares the normalized WSs of the five systems. As shown, the RBC architecture improves the performance for all workloads by a maximum of 1.55× with an average of 16.7%, comparing with the baseline memory system. By comparing with the state-of-the-art approaches, the RBC architecture outperforms the DICE and Bingo by the maximum of 33% (with an average of 10.1%) and 60.4% (with an average of 4.7%), respectively. The reason behind this is the RBC is allocated to each running application enabled by the fairness allocator, so that the RBC architecture results in much fair performance improvements for all the running applications. Same as the single-core results, the memory-intensive workloads get more performance improvements than the non-intensive workloads. The

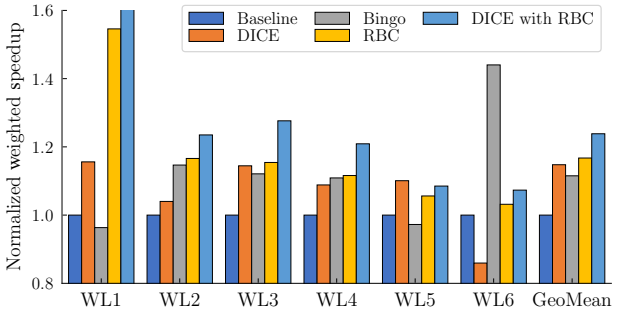


Fig. 11 Normalized weighted speedups of multi-programming workloads. The RBC architecture achieves an average of 16.7% speedup improvement relative to the baseline memory system.

memory-intensive benchmarks suffer the row buffer interference severely and benefit more from the RBC architecture, which can be inferred from the average memory access latency given in Fig. 12. As shown, the RBC architecture reduces the memory request latency by up to 49.1% with an average of 29.2%. Similar to the single-core configuration, the reduced latency comes from the RBC hits, which alleviate unnecessary activate and precharge operations caused by row buffer conflict. Also, the memory request latency becomes more stable and predictable, as the standard variation of the memory request latency drops up to 30.9% (5.2% on average).

4.3.2 Energy consumption

Figure 13 compares the normalized energy consumption

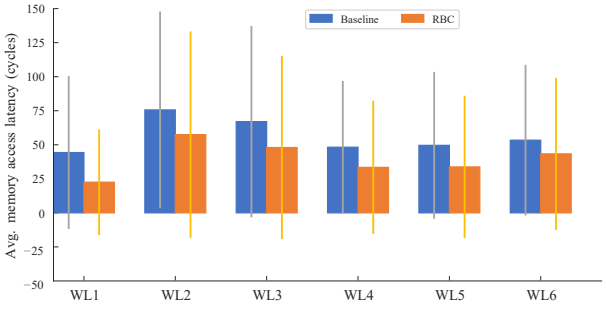


Fig. 12 Memory access latency comparison in multi-core scenario.

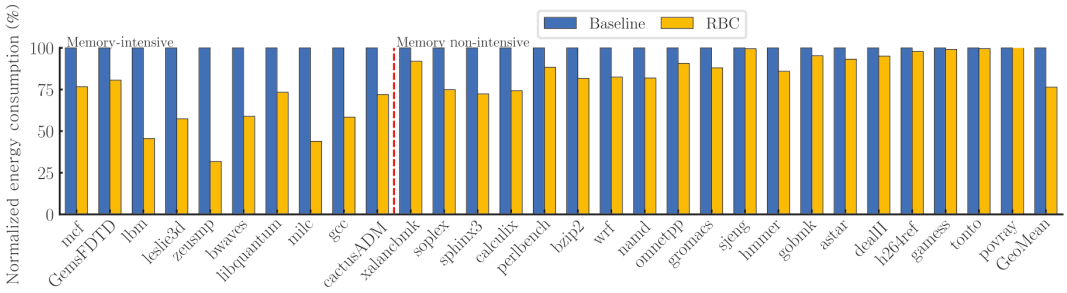


Fig. 10 Comparison of the memory system energy consumption in the single-core scenario. Values are normalized to the baseline. The RBC architecture saves energy consumption by up to 68.2% and 23.6% on average.

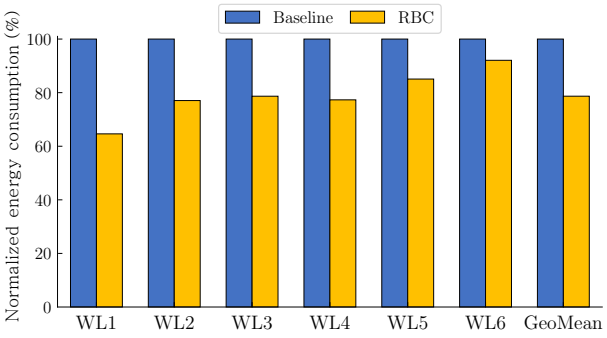


Fig. 13 Normalized energy consumption of the multi-programming scenario.

of the proposed RBC architecture against the baseline memory system. As shown, the RBC architecture reduces energy consumption by up to 35.4% (WL1) and with a geometric mean of 21.3% across all the workloads. Similar to the performance improvements, the more memory-intensive benchmarks a workload contains, the more energy-saving can be achieved. The reduced row activation and precharge commands are credited for the energy consumption reduction. Besides, the energy reduction can be attributed to the reduced background energy, due to the faster execution enabled by the RBC architecture.

4.3.3 Sensitive study on LLC size

The LLC within the processor impacts the volume of memory system traffic and affects the spatial locality presented at the memory level. To investigate the impacts that LLC may impose to the row buffer interference, Figs. 14a – 14c have shown the performance improvement, energy consumption, and RBC hit rate with various LLC sizes (ranging from 1 MB to 32 MB), respectively. Two main conclusions are in order from Fig. 14. First, the row buffer interference cannot be eliminated by the increasing LLC size, which provides opportunities for the proposed RBC architecture. As shown, the RBC architecture can improve performance by up to 1.69× for WL1 with 1 MB LLC (with a geometric mean of 15% across all the workloads in all LLC sizes) and reduce energy consumption by up to 42% for WL1 with 1 MB LLC (with a geometric mean of 19.9% across all the workloads in all LLC sizes) comparing to the baseline memory system. With larger LLC capacity, more cache lines can be cached within the LLC, which decreases the off-chip memory traffic and also changes the pattern of memory requests. Thus, the improvement of both performance and energy efficiency drops with the larger LLC capacity, as shown in Figs. 14a and 14b, respectively. Also, the enlarged LLC can not escape from the row buffer interference, as the spatial locality of the memory requests still exist, which can be harmed by both intra- and inter-application row buffer interference. Based on this, the RBC architecture protects the spatial locality and makes further performance improvements and energy efficiency. Second, depending on the memory request intensity, the trends of the row buffer hit ratio varies with the increase of the LLC size among all workloads. As the increased LLC size reduces both the memory requests volume and the spatial locality, the RBC hit rate increases with the increasing of the LLC size for the workloads with less memory traffic (e.g., WL6), which can be attributed to the enlarged LLC filtering out the memory requests with less locality. However, for the memory-intensive workloads (e.g., WL1 and WL2), the filtration effect is less significant as the memory requests of such workload can diminish the improvements brought by the enlarged LLC capacity. It is noteworthy that, with the increase of LLC capacity, the RBC hit rate increases while the overall normalized weighted speedup is decreasing. The reason behind this is larger LLC capacity leads to less memory accesses which potentially increases the chance that a memory request can be served by the RBC, however, the performance improvement brought by the RBC decreases due to the less intensive memory accesses. Overall, the *RBC* architecture brings both performance improvement and energy efficiency under various LLC capacity, as the increased LLC can not eliminates the row buffer interference.

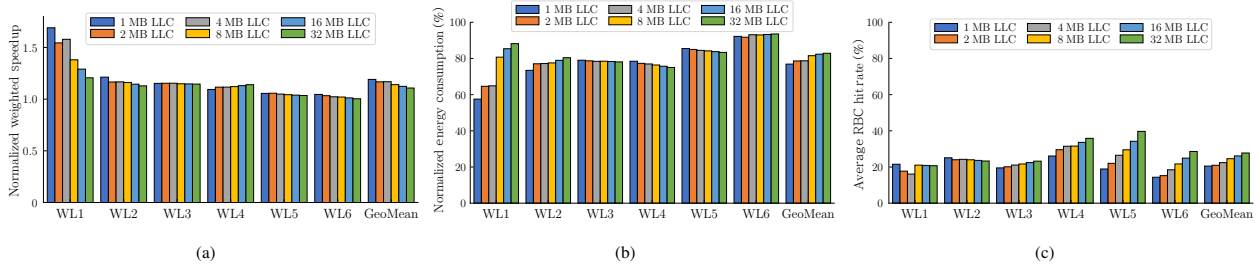


Fig. 14 Sensitive study on the impact of different LLC sizes to the effectiveness of the RBC architecture.

5 Related Work

To address the "memory wall" problem, various solutions have been suggested. DRAM refresh slows down memory performance and has notably received a lot of research interest. Refresh pausing[5] suggests interruptible refreshes and temporarily suspends ongoing refresh to give way to regular memory requests. Parallelizing refresh and memory requests[26,27] in different memory sources can also decrease refresh performance impacts. CAR[7] reduces the rows needed to refresh by compressing the data stored in the DRAM. As the refresh blocks the process of memory requests, ROP[6] alleviates the refresh overheads by prefetching data into an SRAM buffer and using the data in the buffer to serve the memory requests arriving during the refresh period.

Researchers have also conducted studies and taken advantage of the rich internal parallelism of the DRAM. Kim et al.[28] proposed a memory architecture with subarray level parallelism and explored several ways to leverage the parallelism to improve performance. Jeong et al.[29] employed bank partitioning to assign separate banks to distinct applications, so that accesses to the same bank may have better row buffer locality. Tiered latency DRAM[3] splits bitlines to near and far segments via adding isolation transistors on the bitlines, so that near-segment has shorter wire length and less parasitic capacitance. Therefore, accesses in the near-segment have shorter latency. In order to improve the system performance, LAMS[30] take advantage of the TLDRAM architecture and schedule the requests in the near-segment with a higher priority. Seshadri et al.[1] proposed a row clone memory architecture, in which a row can be moved to another row efficiently via row buffer. Also, the half-DRAM[31] re-routed wordlines to select two different rows simultaneously, which increases the utilization of the row buffer. Half Page[8] exploits half-page access granularity to reduce row buffer overfetch cost. To better exploit the benefits of Phase Change Memory (PCM), Row Buffer Locality-Aware (RBLA) caching mechanism[32] was proposed to take advantage of the lower row buffer miss latency by placing the rows with poor locality to the DRAM while leaving the rows with good locality in the PCM. Substantially, our proposed RBC architecture also benefits from the increased utilization of the row buffer, which is enabled by caching the rows with good spatial locality in the RBC. The RBC architecture distinguishes from others in the following aspects. First, the RBC architecture does not rely on the modification of the organization of cells on the DRAM chips. Second, no matter how fine-grained the accessing granularity is, the row buffer interference still exists; thus, the proposed RBC architecture can mitigate the row buffer interference with the evolution of the DRAM.

6 Conclusion

The overheads of DRAM row buffer interference become more critical as the scaling of DRAM. In this paper, we propose a new row buffer interference mitigation approach RBC, which effectively alleviates row buffer interference overheads. The RBC protects the spatial locality by caching the rows with good locality in the RBC. To better exploit the RBC, row buffer interference detector and row buffer locality predictor are used to accurately identify interference venerable data and maximize the benefits brought by the RBC. In addition, the fairness controller is used to ensure fair allocation of the RBC for all interference-sensitive applications. Our extensive evaluations in both single-core and multi-core systems have demonstrated that the RBC successfully alleviates the row buffer interference overheads under various workloads. By comparing RBC with the state-of-the-art methods, RBC outperforms both DICE and Bingo.

Acknowledgment

This work was supported by the US National Science Foundation (Nos. CCF-1717660 and CNS-1828363).

References

[1] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, et al., Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization, in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, Davis, CA, USA, 2013, pp. 185–197.

[2] O. Seongil, Y. H. Son, N. S. Kim, and J. H. Ahn, Row-buffer decoupling: A case for low-latency dram microarchitecture, in *Proceedings of ACM/IEEE 41st International Symposium on Computer Architecture* , Minneapolis, MN, USA, 2014, pp. 337–348.

[3] 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 *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture* , Shenzhen, China, 2013, pp. 615–626.

[4] J. Stuecheli, D. Kaseridis, H. C Hunter, and L. K. John, Elastic refresh: Techniques to mitigate refresh penalties in high density memory, in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010, pp. 375–384.

[5] P. Nair, C.-C. Chou, and M. K. Qureshi, A case for refresh pausing in DRAM memory systems, in *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture*, Shenzhen, China, 2013, pp. 627–638.

[6] P. Huang, W. Liu, K. Tang, X. He, and K. Zhou, Rop: Alleviating refresh overheads via reviving the memory system in frozen cycles, in *Proceedings of 45th International Conference on Parallel Processing*, Philadelphia, PA, USA, 2016, pp. 169–178.

[7] W. Liu, P. Huang, K. Tang, K. Zhou, and X. He, CAR: A compression-aware refresh approach to improve memory performance and energy efficiency, *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 373–374, 2016.

[8] H. Ha, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, Improving energy efficiency of DRAM by exploiting half page row access, in *Proceedings of 49th Annual IEEE/ACM International Symposium on Microarchitecture* , Taipei, China, 2016, pp. 1–12.

[9] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, Thread cluster memory scheduling: Exploiting differences in memory access behavior, in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010, pp. 65–76.

[10] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, Improving system energy efficiency with memory rank subsetting, *ACM Transactions on Architecture and Code Optimization* , vol. 9, no. 1, p. 4, 2012.

[11] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, Multicore DIMM: An energy efficient memory module with independently controlled DRAMs, *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 5–8, 2008.

[12] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, Memory access scheduling, in *Proceedings of ACM/IEEE 27th International Symposium on Computer Architecture*, Vancouver, Canada, 2000, pp. 128–138.

[13] D. Kaseridis, J. Stuecheli, and L. K. John, Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era, in *Proceedings of 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Porto Alegre, Brazil, 2011, pp. 24–35.

[14] O. Mutlu, Memory scaling: A systems architecture perspective, in *Proceedings of 5th IEEE International Memory Workshop*, Monterey, CA, USA, pp. 21–25.

[15] DDR4 SDRAM standard, http://www.jedec.org/standards-documents/results/jesd79-4%20ddr4, 2012.

[16] J. Shao and B. T. Davis, A burst scheduling access reordering mechanism, in *Proceedings of IEEE 13th International Symposium on High Performance Computer Architecture*, Phoenix, AZ, USA, 2007, pp. 285–294.

[17] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, Micro-pages: Increasing DRAM efficiency with locality-aware data placement, in *Proceedings of the 15^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, PA, USA, 2010, pp. 219–230.

[18] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, Gather-scatter dram: In-DRAM address translation to improve the spatial locality of non-unit strided accesses, in *Proceedings of the 48th International Symposium on Microarchitecture*, Waikiki, HI, USA, 2015, pp. 267–280.

[19] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, DRAMSim2: A cycle accurate memory system simulator, *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[20] D. Sanchez and C. Kozyrakis, Zsim: Fast and accurate microarchitectural simulation of thousand-core systems, *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 475–486, 2013.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005, pp. 190–200.

[22] P. Shivakumar and N. P. Jouppi, Cacti 3.0: An integrated cache timing, power, and area model, Report, WRL, 2001.

[23] Micron system power calculator, http://www.micron.com/support/power-calc, 2019.

[24] V. Young, P. J. Nair, and M. K. Qureshi, Dice: Compressing dram caches for bandwidth and capacity, in *Proceedings of ACM/IEEE 44th Annual International Symposium on Computer Architecture*, Toronto, Canada, 2017, pp. 627–638.

[25] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, Bingo spatial data prefetcher, in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, Washington, DC, USA, 2019, pp. 399–411.

[26] K. K.-W. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, Improving DRAM performance by parallelizing refreshes with accesses, in *Proceedings of IEEE 20th International Symposium on High Performance Computer Architecture*, Orlando, FL, USA, 2014, pp. 356–367.

[27] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, CREAM: A concurrent-refresh-aware DRAM memory architecture, in *Proceedings of IEEE 20th International Symposium on High Performance Computer Architecture*, Orlando, FL, USA, 2014, pp. 368–379.

[28] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, A case for exploiting subarray-level parallelism in DRAM, *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 368–379, 2012.

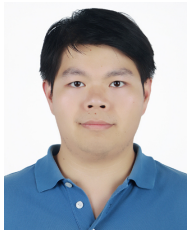
[29] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, Balancing DRAM locality and parallelism in shared memory cmp systems, in *Proceedings of IEEE International Symposium on High-Performance Comp Architecture*, New Orleans, LA, USA, 2012, pp. 1–12.

[30] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He, LAMS: A latency-aware memory scheduling policy for modern dram systems, in *Proceedings of IEEE 35th*

International Performance Computing and Communications Conference, Las Vegas, NV, USA, 2016, pp. 1–8.

[31] 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 *Proceedings of the 41st International Symposium on Computer Architecture*, Minneapolis, MN, USA, 2014, pp. 349–360.

[32] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, Row buffer locality aware caching policies for hybrid memories, in *Proceedings of IEEE 30th International Conference on Computer Design* , Montreal, Canada, 2012, pp. 337–344.

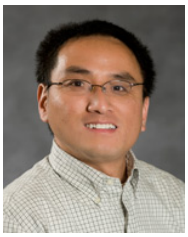


Wenjie Liu received the MS degree from Huazhong University of Science and Technology, Wuhan, China in 2018. He is currently working toward the PhD degree in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA. His main reserach interest includes DRAM, distributed systems, and nonvolatile memory. He has published papers in various international conferences and journals, including MASCOTS, ICPP, IPCCC, Sigmetrics, *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), etc. He is a student member of the IEEE.

Ke Zhou received the BEng, MEng, and PhD degrees from Huazhong University of Science and Technology (HUST), Wuhan, China in 1996, 1999, and 2003, respectively. He is a professor of Wuhan National Laboratory for Optoelectronics and the School of Computer Science and Technology, Huazhong University of Science and Technology. 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 *TPDS*, *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.

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

Tianming Yang received the BS degree from the Department of Computer Science, Zhengzhou Institute of Technology, Zhengzhou, China in 1991, and the PhD degree from Huazhong University of Science and Technology, Wuhan, China in 2010. Currently, he is an associate professor with the Department of Information Engineering, Huanghuai University, China. His current interests include data backup, networking storage, parallel file systems, disk array, and solid state disk. He has more than 20 publications in international conferences and journals, including IPDPS, GRID, NAS, ICCD, IPCCC, ICS, *Journal of Zhejiang University-SCIENCE C*, 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, USA in 2002. He is currently a professor with the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. 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 Awards (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE and a member of the IEEE Computer Society and USENIX.