DR DRAM: Accelerating Memory-Read-Intensive Applications

Yuhai Cao[1], Chao Li[1], Quan Chen[1], Jingwen Leng[1], Minyi Guo[1], Jing Wang[2], and Weigong Zhang[2]
Department of Computer Science and Engineering, Shanghai Jiao Tong University
Beijing Advanced Innovation Center for Imaging Technology, Capital Normal University
cyh-shanghai@sjtu.edu.cn, {lichao, chen-quan, leng-jw, guo-my}@cs.sjtu.edu.cn, {jwang, zwg771}@cnu.edu.cn

***Abstract*—Today, many data analytic workloads such as graph processing and neural network desire efficient memory read operation. The need for preprocessing various raw data also demands enhanced memory read bandwidth. Unfortunately, due to the necessity of dynamic refresh, modern DRAM system has to stall memory access during each refresh cycle. As DRAM device density continues to grow, the refresh time also needs to extend to cover more memory rows. Consequently, DRAM refresh operation can be a crucial throughput bottleneck for memory read intensive (MRI) data processing tasks.**

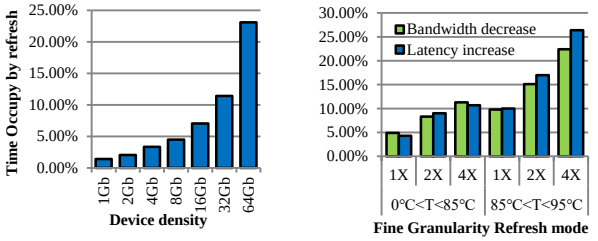
To fully unleash the performance of these applications, we revisit conventional DRAM architecture and refresh mechanism. We propose DR DRAM, an application-specific memory design approach that makes a novel tradeoff between read and write performance. Simply put, DR has two layers of meaning: device refresh and data recovery. It aims at eliminating stall by enabling read and refresh operations to be done simultaneously. Unlike traditional schemes, DR explores device refresh that only refreshes a specific device at a time. Meanwhile, DR increases read efficiency by recovering the inaccessible data that resides on a device under refreshing. Our design can be implemented on existing redundant data storage area on DRAM. In this paper we detail DR's architecture and protocol design. We evaluate it on a cycle accurate simulator. Our results show that DR can nearly eliminate refresh overhead for memory read operation and brings up to 12% extra maximum read bandwidth and 50~60% latency improvement on present DRR4 device.

Keywords— DRAM refresh; read-intensive; data analysis; memory bandwidth; redundant data storage

I. INTRODUCTION

The memory read bandwidth has become an ever-tightening computing resource today, especially for applications that generate limited write operations on very large input data. In the era of intelligent systems and edge computing, it is not unusual that the ratio between output and input data can be very small. For example, data clustering applications may involve an input audio/image file with size in many KB, while output label is only several bytes. A neural network training algorithm may take thousands of training dataset, while the model data can be stored in on-chip cache. Some of the in-memory database applications are also focused on analytic operations that require no writing to the database [1]. Many basic operations such as linked-list traversal (LLT) and adjacent table access (graph processing algorithms) exhibit low memory write-to-read ratio as well. They are typically hungry for random read bandwidth with few data update, exchange and write back.

Efficient memory system design is crucial for accelerating emerging memory read intensive (MRI) applications. They need to read raw data from the memory sequentially without frequent write back. In the above scenarios, data movement from memory to processor dominates the performance. Oftentimes, these applications are read-latency-sensitive. As memory access latency increases, the computation tasks can be easily blocked, which will greatly degrade data processing throughput.



(a) Time occupied by refresh (b) BW and latency of a 8Gb DDR4

Figure 1: Performance impact of DRAM refresh

There are mainly two ways to optimize memory bandwidth in the literature. 1) Accelerator designers use on-chip memory to perform computation locally [2, 3]. While on-chip memory has limited storage space and the content delivery speed still depends on off-chip memory bandwidth. 2) Some papers propose to move processing close to the storage system, namely, processing-in-memory (PIM) and near-data processing (NDP) [4, 5]. These techniques often require special hardware (e.g., HMC, 3D-stack memory) as the platform, which are still under rapid development and not likely to be widely adopted quickly. Although emerging persistent memory (PM) devices provide DRAM-like read latency, they could introduce huge design complexity to the OS and software stack due to the unique property of persistent data storage [6,7,8].

In this work we explore an alternative design approach that could enhance DRAM read throughput. We attack an important underlying root cause of the problem: the memory refresh itself. Memory controller generates refresh operations periodically to recharge voltage. The refresh operation locks DRAM devices and stall access.

The overhead caused by refresh cannot be ignored. In Fig. 1(a), the performance overhead due to refresh grows rapidly as DRAM density increases. For a 64Gb DDR4 device, the time spends on refresh can be over 20%. If ambient temperature increases, the refresh frequency may double. In Fig. 1(b) we compare different modes of Fine Granularity Refresh (FGR) [10] with an ideal case that has no refresh overhead. In the worst case, we observe 23% degradation of the maximum bandwidth. Thus, refresh can greatly impact read bandwidth especially when the device becomes denser.

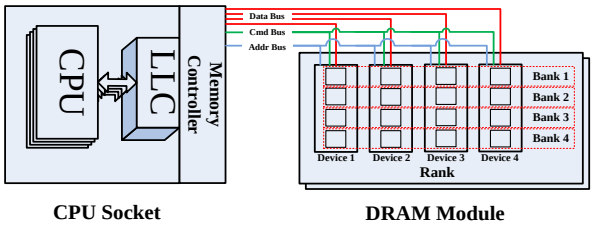


Figure 2: DRAM organization

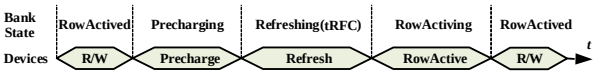


Figure 3: Refresh timing for DRAM

The driven insight of our work is that the refresh efficiency of traditional DRAM system becomes incommensurate with its performance goals for some of the emerging applications. It is important to re-think memory design and tap into application-specific DRAM architecture.

We propose DR DRAM, an enhanced DRAM module. Unlike previous studies that emphasize both read and write performance, we only focus on improving memory read efficiency to make MRI applications process faster. Our main idea is to minimize processor stalls by making read operation available during each refresh cycle. In other words, DR DRAM enables refresh and read operations to be done simultaneously.

The proposed DR mechanism highlights two features: device refresh and data recovery. DR DRAM is a device refresh based memory since it only refreshes a specific DRAM device at a time. This distinguishes our work with conventional designs that refresh the same bank/row for all devices. DR DRAM is also a data recovery based memory since it further combines memory refresh with the Parity-Check-Code (PCC) mechanism. It can recover the inaccessible data that resides on the device that is under refreshing. By making column read process independent of refresh operation, our system can unleash the full potential of DRAM read bandwidth and latency. Modern DRAMs typically have under-utilized redundant data storage area for fault-tolerance. DR can take advantage of this opportunity to boost performance without adding overhead. It does not require additional PINs on the CPU side as well.

Overall, DR DRAM makes a novel trade-off between read and write performance for emerging memory read intensive tasks. It represents a kind of application-transparent, hardware-oriented optimization scheme. It is configurable and easy to implement on traditional Dual Inline Memory Module (DIMM). It requires moderate extension on existing architecture and memory control.

We evaluate our DR based on DRAMSim2. The results show that our design yields notable performance improvement on both bandwidth and latency in synthetic trace-based simulation. We also demonstrate that DR can benefit many real-world MRI tasks. We discuss potential hardware/software methods that can further improve DR. This paper makes the following contributions:

1) We propose DR DRAM, a novel enhanced DRAM module design for emerging memory read intensive tasks. It combines device-level refresh and data recovery to make read operation available during refresh cycle.

2) We devise the protocol of DR DRAM and describe it in detail (including bank state change and state transition). The new memory protocol enables existing memory controller to effectively perform device-level refresh

3) We architect a novel memory module so as to support DR technique at the memory hardware level. We illustrate the operating process of memory module.

II. BACKGROUND

A. Refresh Mechanism

As Fig. 2 shows, the hierarchy of DIMM is made up of channel, rank, bank, row, and column. A row is the smallest refresh unit in the bank and it is composed of multiple columns. A column is the smallest addressable unit and the memory controller accesses DRAM at column granularity. The data size of a column is as same as the device width.

In a commercial *DDRx* device, the memory controller needs to send auto-refresh command during every refresh interval (*tREFI*). Typically, the retention time of data in DRAM cell is 64ms if the ambient temperature is less than 85 degree Celsius or 32ms if the ambient temperature is higher than it [10]. The memory controller needs to send 8192 refresh commands within data retention time, to make sure that all rows will be refreshed. *tREFI* for *DDRx* device is around 7.8us under 85 degree Celsius. The time of refresh cycle (*tRFC*) is related to the number of rows to be refreshed in each refresh cycle. Some devices also support Fine Granularity refresh mode (FGR) for better trade-off between *tREFI* and *tRFC* [10].

Fig. 3 shows the refresh protocol for a particular bank. Before refresh, if the bank is in *RowActivated* state，the memory controller must send precharge command to reset the row amplifier. After *precharge*, the memory controller can send a refresh command. During a refresh cycle, the DRAM first reads row data into row amplifier then restores electron charge back to the capacity cell. The above two steps may repeat several times during a refresh cycle. Once finished, the bank state will return to *Idle* and it must be activated before next column read/write.

B. MRI Applications

Memory-read-intensive (MRI) applications are applications that are sensitive to memory read reference. They normally have the following two attributes: 1) consuming large amount of memory bandwidth and sensitive to memory access latency; 2) showing limited memory write reference activities, which means low write back rate (e.g., less than 10%) in the last level cache.

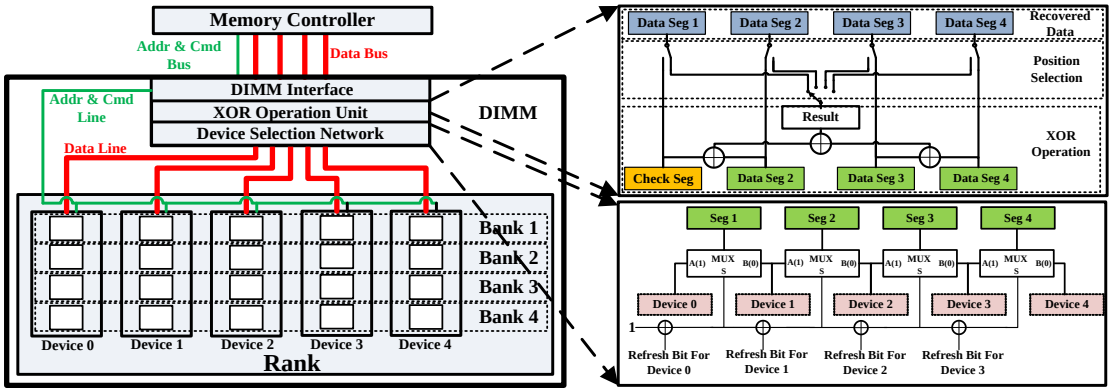


Figure 4: Overview of the DR DRAM system organization

Data analytic applications such as neural network training and inference are sensitive to memory access latency. For example, when running inference application on a high performance server, raw data such as image data or video data are read sequentially from the memory. The memory read latency is very important when computation resources are abundant, since the processer can easily stall without data input. When the DRAM is refreshing, the processor has to wait hundreds of nanosecond to receive the raw data which cause a great impact on MRI applications.

Another type of representative MRI application is link list traversal (LLT) like operations. This includes LLT, hash table look up, graph traversal, and etc. These operations are mainly bandwidth hungry due to fewer arithmetic operations. Their memory references are generally random with low spatial locality. In addition, since the required data is also smaller than the size of cache line, the unnecessary data transfer from the memory to the CPU unavoidably wastes bus bandwidth [2]. Moreover, low spatial locality also causes frequent row amplifier switch between the *RowActivated* and *Precharge* state. That will further degrade performance and increase read latency.

III. DESIGN FOR MRI TASKS

In this work we intend to improve the read performance for MRI tasks by enhancing exiting DRAM modules. Fig. 4 depicts our proposed designs.

A. DR DRAM Design

We propose DR DRAM, an application-specific DRAM that features *device refresh* and *data recovery*. Specifically, unlike tradition refresh mode (per-rank refresh or per-bank refresh), our design works at a fine-grained DRAM device level. Enlightened by the RAID3 technology, we devise a technique which enables data read during memory refresh. When a segment of data is unreachable due to refreshing, we can recover it through PCC and XOR operation.

In general, we consider a memory system in which the data bus width is 64bits and device width is 16bits. To ensure that all the 64bits data can be accessed even during refresh cycle, we enlarge data width to 80bits by adding an additional DDR device (Device 0). To ensure that we can always access 64bits of data, at most one device can be in refresh cycle at any time. During the interval between two refresh cycles, we can access 80bits of data.

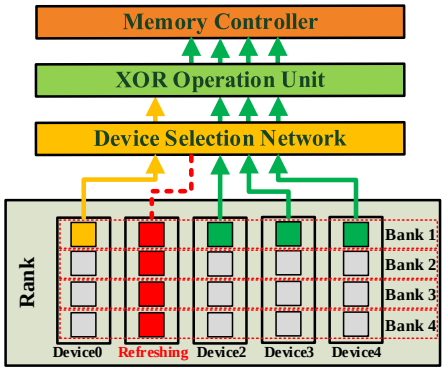


Figure 5: DR DRAM read process

As shown in Fig. 4, to support the device refresh and data recovery mechanism, we devised two modules on the memory controller: a XOR operation unit and a device selection network (DSN). The XOR operation unit can generate parity check code during a column write process and recover inaccessible data in a column read process. The device selection network is mainly composed of several two-way multiplexers (MUX). It is used to select accessible devices and redirect data path. The MUXs can select A-path by input 1 or select B-path by input 0.

1) *Column-Read Process*

Fig.4 also indicates the implement of refresh at device-level and the process of data recovery when issuing a column-read command. The DSN only links accessible devices. The input bits are “0100”, which indicates Device 1 is in refresh cycle. Then the MUXs will select route {A, B, B, B} and Device {0, 2, 3, 4} will read into Segments. The PCC (located in the Device 0) and the original data (located in Devices 1~4) are delivered to the XOR Operation Unit. As shown in Fig. 5, the inaccessible segment in Device 1

(Data Segment 1) will be recovered and cached into an output buffer/register. Then, our position selection network (in Fig. 4) sends recovered data to its correct location. Finally, we get the whole column of data.

2) *Column-Write Process*

To ensure that all the data segments can be recovered in the read process during the refresh cycle, the column write process must write all the data including PCC into the memory. The column write command is only allowed during the interval of DR refresh cycle. As DR DRAM refreshes more frequently, it may decrease write performance comparing with traditional refresh method. The original data is divided into 4 segments. The PCC is generated by the XOR Operation Unit. Our DSN only works under the column read operation, which means that all the 5 data segments are written into its individual device directly. Afterwards, The PCC is recorded in the Device 0 and the original data are stored in Devices 1~4.

3) *CPU Pin Issue*

In our design the DR DRAM has to control each device refresh individually. As our design packs devices into the memory module and the CPU interact with DIMM interface, we can easily constrain the number of PINs within the memory module even if the CPU do not have additional PINs. Conventional DRAM module typically has multiple devices and they share address and command PINs. For DR DRAM, devices within a memory module no longer share control PINs and we do not introduce additional PINs on the CPU side. Our memory controller can encode refresh command and the designated device to a new command. The DIMM interface decodes the new command and sends refresh command to the designated device directly.

B. DR Control Protocol

In this part, we describe our initial implementation of the DR mechanism at the protocol level. Our refresh protocol determines when memory controller should send DR refresh command. We modify and extend the original bank states of the DRAM. We implement these bank states in the cycle-accurate DRAMSim2 simulator [11].

The timing example of refresh protocol in DR mode is shown in figures 6(a) and 6(b). Our memory controller is designed to be fully aware of the detailed timing process specified by the protocol. In Fig. 6(a) we show the start timing of a particular device (i.e., Device-Ref). At the beginning, all devices are in the same state. The states can be one of the following: *allRowActivate*, *allPrecharge* or *allIdle*, which means all devices in *RowActive* state, *Precharge* state and *Idle* state individually. The *all* prefix indicates all devices are in same state. The postfix indicates which the state is. Once a refresh command arrives, the Device-Ref stalls and starts to refresh. However, other devices maintain their initial states and continue their operation. From a bank's perspective, its new state can be one of the following: *refRowActivate, refPrecharge* or *refIdle*. The

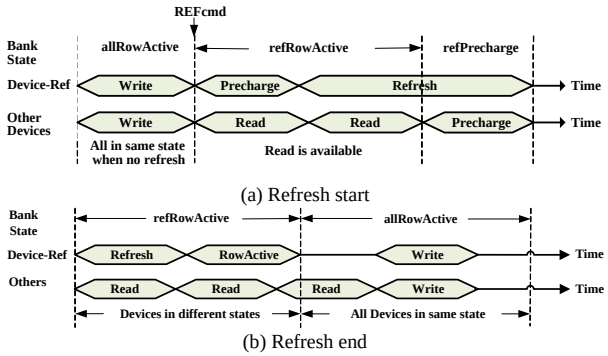


Figure 6: DR DRAM timing

ref prefix indicates that a device is in refresh cycle. The postfix indicates the state of the non-refresh devices. At the end of each refresh cycle, the memory controller determines the next bank states. Fig. 6(b) illustrates how these devices turn into *allRowActive* state when the refresh ends. Device-Ref should turn into *RowActivate* state at the end of the refresh cycle. At last all devices merge into a same state.

Note that DR should guarantee at most one device during a refresh cycle. Therefore, refresh parameters (*tRFC* and *tREFI*) must satisfy the following constraint:

$$tRFC < tREFI/Device\ number$$

Violation of the above constraint may cause undesirable things when using the DR technique. 1) The refresh cycle will occupy the rows all the time, which will totally disable column write command. 2) A row may refresh multiple banks (or multiple data segments) during a refresh cycle, and consequently, the refreshing data cannot be recovered by XOR operation. At last the memory will become inaccessible for both read and write.

Fig. 7 shows the maximum value that *tRFC* can reach across devices of different width. The restriction is large enough for *x16* and *x8*. For a *x4* device it shows that *tRFC* should be less than 433ns, which is hard to achieve. Thus, our DR technique is more suitable for memory modules using *x8* and *x16* devices.

C. DR Implementation

The detail implementation of DR may vary. In this part, we discuss the possible operating modes of DR.

1) *Original DR (DR)*

Like per-rank refresh, our original DR scheme refreshes a device across all banks in a particular rank. Similar to the per-rank refresh, DR performs synchronous refresh for all banks. Thus, it may cause performance degradation if the refresh rate becomes higher. As Fig. 8 shows, for *x16* devices (refresh rate is *tREFI*/5), the bandwidth degradation caused by synchronous refresh can be around 2.5%.

2) *DR at Asynchronous Mode (DR-A)*

To overcome the synchronous problem in original DR, one can implement DR in an asynchronous way. The refresh operation for DR-A is on a per-bank basis and it is asynchronous among different banks. Like per-bank refresh,

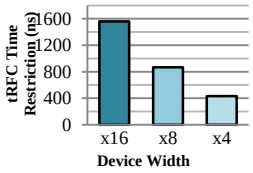


Figure 7: Timing restrictions on memory devices

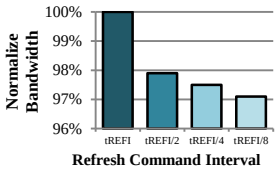


Figure 8: BW degeneration due to memory refresh

a refresh command only selects the particular bank instead of the entire rank. DR-A will not stall bank operation to waiting for refresh. Compared with original DR, DR-A is more fine-gained. Therefore, the memory controller has to generate increased number of refresh command.

3) *DR at Burst Refresh Mode (DR-B)*

The original DR and DR-A are both distributed refresh, which means a refresh cycle only refresh one or several rows. To reduce the number of refresh command, one can also apply burst refresh mode that refreshes all rows and all data cell in a single refresh cycle. The refresh command interval time (*tREFI*) for a particular device is 64ms.Also a refresh cycle (*tRFC*) can take very long time. Though DR-B can achieve the best performance in principle, it may greatly decrease write performance.

IV. OPTIMIZING DR DRAM

DR DRAM seeks a better design trade-off between read and write performance for MRI applications. In this section we introduce two optimization schemes that minimize the design overhead on memory write operation.

A. Col-Read Interrupt

Column write operation can only be achieved when no device is refreshing. Therefore, we can increase the interval time between two refresh cycles to give more time for write transaction. DR uses an optimization scheme called col-read interrupt to improve memory write performance. Traditionally, if a refresh command comes after column read operation, the refresh command should wait until the finish of read. Our optimization can interrupt read process on the device that is prepared to refresh. Once the column read operation is interrupted, the targeted device will enter the refresh cycle immediately without waiting for read. As the refresh cycle is fixed, an early start of a refresh cycle will cause an early end. Then the interval time between two refresh cycles are increased. Besides, col-read Interrupt does not influence data integrity due to the following data recovery process. It only makes one segment of data inaccessible during the read process.

B. Bank-Aware Write

Sometimes the impact of the col-read interrupt mechanism is still limited. To further improve column-write performance, we propose to schedule write transactions and trigger column write operation when bank is not busy. This DR optimization scheme is call write-only-cache (WOC).

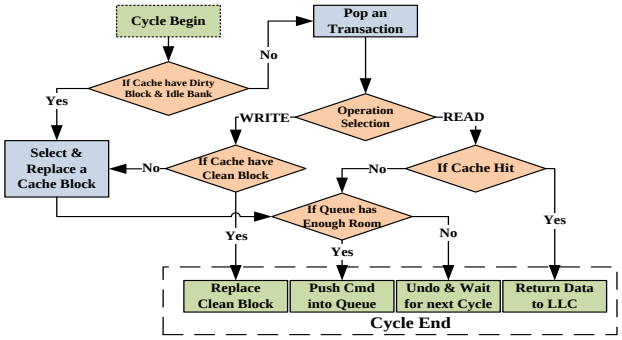


Figure 9: Control flow diagram of DR optimization

Unlike operation interrupt, WOC locates in memory controller and only manages write transaction. It stores column data and schedules write access operation. If the target bank is busy, the write transaction will be temporarily stored in WOC until the bank state becomes idle.

Fig. 9 illustrates the control flow of the memory controller with WOC. At the beginning of a cycle, the memory controller searches dirty blocks with *Idle* bank state. If such block is found, the memory controller will set clean bit for the block and push a WRITE command into the command queue. If no such block is found, memory controller will pop a transaction from transaction queue. The next step depends on the type of transaction. If it is WRITE transaction, controller will add it into cache based on LFU policy. If it is READ operation with cache miss, the column data will be read from memory without cached into WOC. This mechanism can balance the col-write workload and improve performance if there are write back operations.

V. EVALUATION METHODOLOGY

This section describes our experimental methodology. We evaluate DR from three different aspects: 1) synthetic trace-based simulation, 2) realistic workload measurement, and 3) hardware design overhead analysis.

A. Basic Configuration

We evaluate our design with a modified trace-based cycle-accurate simulator base on DRAMSim2 [11]. We experiment with different system configurations to verify the performance of DR. Table 1 summarizes some key parameters we used. The timing parameter we used are referenced from recent work [9] and industry manual [12]. The DRAM device we evaluated is configured as a commercial DDR4 system [12]. The parameters of memory controller are derived from the default setting in DRAMSim2. We use the default setting of L1 cache in gem5. The parameters related to power estimation such as current values and voltage values come from micron [12].

B. Synthetic Traces

We first use synthetic traces to verify the effectiveness of our memory module design, as shown in Fig. 10. A trace generator pushes READ/WRITE transactions into a transaction queue at fixed intervals. The transaction queue is

TABLE I: EVALUATED MEMORY SYSTEMS

	Parameter	Configuration
DRAM	Refresh Timing	tREFI=7800ns; tRFC=890ns[9], 550ns, 350ns[12];
	Architecture	DDR4: 2400E; Device Width:16bits; Columns per Row:1024; Num of Bank:16; Num of Rank:1;
Memory Controller	Policy	Open Row Policy;
	Queue Size	Command Queue Size: 256; Transaction Queue Size: 64;
	Address Mapping	channel:row:col:bank:rank;
	Scheduler	FR-BRR;
Write Only Cache	Latency(cycle)	Tag Latency: 2; Response Latency: 2; Data Latency: 2;
	Size	Cache Size: 64KB; Cache Line Size: 64B; Cache Associativity: 1;

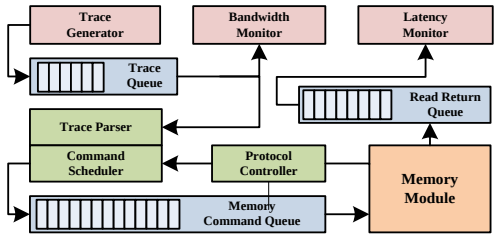


Figure 10: Implementation of trace-based simulation

out-of-order with a read first policy. A bandwidth monitor locates at the frontend of the transaction queue. If the data flow is not blocked, the transaction queue will pop one transaction at each cycle. Then the transaction will translate into memory command with the help of a trace parser, a command scheduler and a protocol controller. The command scheduler accesses bank in a round-robin manner and sends commands to the memory module. For column read command, the memory module should send response data to a read return queue (RRQ). In the RRQ, a latency monitor can calculate the latency of READ transaction.

For synthetic trace based evaluation, we experiment with four configurations: *intensive read*, *moderate read*, *synthetic combination*, and *FGR mode*. By sending traces to DRAMSim2 at different intervals, we can obtain the bandwidth and latency performance of our DRAM module.

We evaluate intensive read and moderate read by adjusting the trace generation cycle. In order to verify the results of intensive read, we set the trace generation cycle to 1ns. Such a short generation cycle will make transaction queue filling with read task. For moderate read, we make command queue and transaction queue unblocked by setting trace generation cycle larger (8ns, 16ns, 32 ns and 64ns). Although memory bandwidth will not be fully utilized, we can get more reasonable latency result due to shorter queue waiting time. We evaluate intensive read and moderate read under both random read and sequential read.

Our synthetic combination traces include both read and write operations. We use it to verify the impact of WRITE (col-write) operation. Our simulator is configured through mixing write transactions with read transaction based on intensive read. The ratio of write and read is between 10% and 0.001%. The configuration of our trace reference is derived from our target application. We store the input data sequentially in the DRAM. We assume the memory write trace is randomly distributed in the memory. We consider data analytic accelerators that have on-chip storage. The on-chip storage allows accelerators to calculate locally and avoid aggressive memory reference. The size of the output data is far less than the input data. Most of the input data can be abandoned after finishing processing.

We also evaluate MRI tasks under FGR (Fine Granularity Refresh) mode, as it is the state-of-art technique in the DRAM industry. FGR is a technique in *DDR4 SDRAM* which can make a trade-off between refresh latency and frequency. According to the JEDEC standard [10], DDR4 has *1X*, *2X* and *4X* refresh modes. From *1X* to *4X*, the frequency of sending refresh command becomes higher and the refresh cycle becomes longer.

C. Realistic Workload

We further evaluate our design using real workload on gem5 [13]. Our CP U model is single core timing CPU works at 1GHz with 1MB cache size. The memory model is modified from simple memory module. The setting of simple memory model can equal to DDR3-1600 with only one bank. We use three different workloads: CNN inference, linked-list traversal and PageRank. As previous section describes, these workloads are read intensive with low cache write back rate. We use Ligra [14] as framework to run PageRank benchmark. The input graph data is synthetic rMatGraph with more than 1M edges and generated by Ligra itself. We use LLU benchmark [15] to justify linked-list traversals applications. The linked-list we generated in LLU benchmark has 16M nodes with 32Byte node size. CNN inference workload is hand-configured with MNIST input file. To ensure accuracy, our simulation excludes data loading process to make sure simulated instructions are located in algorithm area.

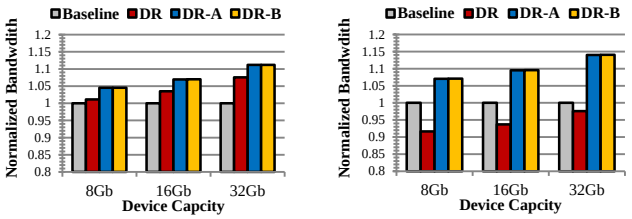
D. Area and Energy

We also evaluate the runtime cost and hardware cost of DR DRAM. We mainly look at the energy and chip area overhead. We also discuss methods for improving the DRAM chip area efficiency of DR DRAM technique. DRAMSim2 can estimate power consumption at the memory operation level. By setting DDR4 current drawn information and MRI sequence trace file, we can generate energy utilization data under different modes of DR.

VI. RESULTS

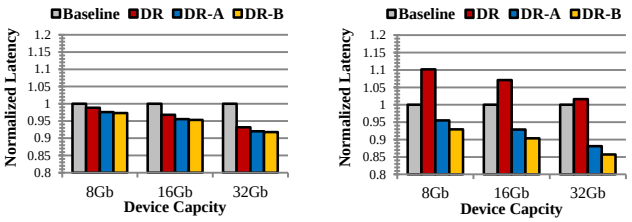
A. Impact on Intensive Read

We first evaluated intensive read operation (MRI tasks) on DR DRAM from the perspective of both bandwidth and latency. Fig. 11 and Fig. 12 present our results for both sequential access and random access. We consider different DDR4 capacities and different DR modes (detailed in Section 3.5). We use micron DDR4 with per-bank refresh as



(a) Random read (b) Sequential read

Figure 11: Bandwidth results of MRI tasks



(a) Random read (b) Sequential read

Figure 12: Latency results of MRI tasks

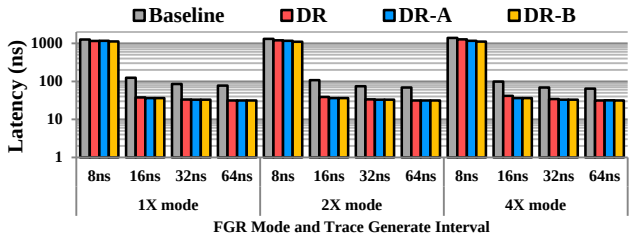


Figure 13: Latency of moderate random read

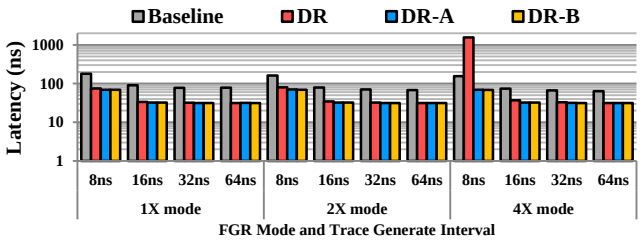


Figure 14: Latency of moderate sequential read

our baseline and adopt the refresh timing parameters of per-bank refresh for DR DRAM. Each group of result is normalized to the baseline.

As shown in the figures, the proposed DR technique has better performance on both bandwidth and latency. For random read on the 32Gb device, the bandwidth of both DR and DR-A can outperform the baseline by around 12%.

For sequential read, we observed some interesting results. The performance of original DR even worse than our baseline. The main reason is that DR uses synchronous refresh among banks, which means that all banks should reach to the idle state before refresh. As a result, it will degrade the performance on both random read and sequential read access patterns.

However, sequential read suffers much more impact than random read. This is because memory cell refresh on sequence read will block operations on all banks before the refresh cycle, while random read only block a few banks. Besides, banks activate different rows frequently due to the low spatial locality of random access. In this case, these banks are more likely to enter into the idle state which can alleviate state transition cost.

By comparing different DR modes, we can see that the way we perform device-level refresh matters. As expected, DR-A outperforms DR greatly due to the asynchronous refresh operation. Overall, DR-B shows the best latency result. This is because that DR-B always sends the fewest refresh commands during operation. DR-B can nearly eliminate the refresh overhead. It almost gets the highest performance on both bandwidth and latency for MRI tasks.

B. Impact on Moderate Read

The memory channel faces less contention under moderate read operations. The bandwidth utilization of different DR implementation methods mainly depends on the query arrival rate. We control the simulation time to be more than 30ms to improve the accuracy. In the following discussion we mainly focus on read latency (the time needed to return the data) result.

Fig. 13 and Fig. 14 show the read latency of a 32Gb DR DRAM under different trace generation cycles (8ns to 64 ns) and different FGR modes. As we can see, in most cases, the latency of moderate read in DR can get an improvement from 50% to more than 60%, which is a significant improvement compared to the result of intensive read. Normally, the read latency is between 70ns to 200ns in our baseline. While in DR, the latency is between 35ns to 80ns. Latency in random read is a little bit larger than sequential read latency due to the row activate operation.

Note that sometimes the system latency can be higher than 1000ns (e.g., when the trace generation cycle is 8ns in Fig. 14) in our design. The reason is that our evaluated system keeps stressing the memory subsystem by continuously generating memory access queries. In this case, both transaction queue and command queue are fully filled. Therefore, read transactions start to spend much more time on waiting in the queue. However, such a long latency may not happen in the real world since the processor may already stall at certain point.

C. Impact of Memory Write

For MRI tasks, the write back rate is much smaller compares to traditional applications on high performance CPU. The write back rate is 0.001% ~ 10%. It can be systems that have large memory read references with most of the temporary data updated in on-chip memory.

Fig. 15 shows the maximum bandwidth when we mix read with write transactions. The horizontal axis indicates the ratio between write number and read number in our trace file. The ratio also equals to the rate of write back of the last level cache. The device we evaluated is a 32Gb *x16* DDR4 DRAM. When the ratio of write back is 10%, DR-

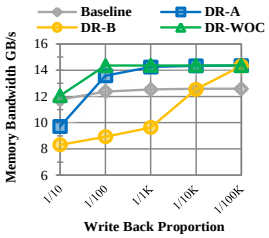


Figure 15: Impact of write

Figure 16: Impact of FGR

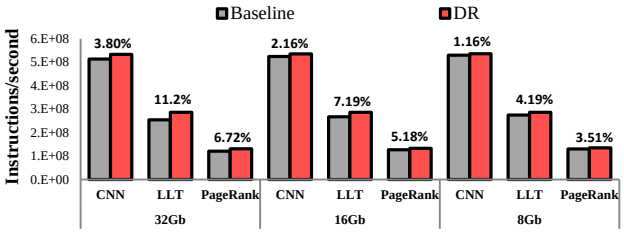


Figure 17: Realistic workload results

A and DR-B are much worse than our baseline. Only DR-WOC (DR-A plus write only cache) slightly outperforms our baseline. For a write back ratio of 1%, DR-A and DR-WOC achieve their maximum bandwidth, while DR-B is still write-sensitive due to the long refresh cycle. DR-B will match or even outperform our baseline as the proportion of write back getting smaller. It also indicates DR-B is not suitable for high write back rate. The reason is the command queue under DR-B can be easily filled.

D. FGR Read Bandwidth

We find that the FGR technique also has a great impact on read performance. Fig. 16 shows the intensive read bandwidth improvement for FGR in a 32Gb DDR4 based on DR-A mode. As we can see, the performance improvement grows from the *1X* mode to the *4X* mode. The reason is that the overall time occupied by refresh cycle becomes larger on high refresh rate mode. While the philosophy of DR DRAM is to eliminate refresh occupation time through parallelize read operation and refresh operation. Therefore, *4X* mode and *2X* mode get more improvement.

E. Real Workload Result

In addition to synthetic trace-based simulation, we also evaluate our design with realistic workload. The write back rate depends on both application feature and hardware platform. In our experiment, LLT has the smallest write back rate with nearly no cache write back. The write back rate for CNN is the biggest one, with 2.23% rate. The write back rate for PageRank is 2.03%.

There is little difference between DR-A and original DR since our memory can be treated as DRAM with only one bank. Further, we do not distinguish different DR modes. Fig. 17 illustrates the instruction per second (IPS) among different memory configurations (different density). The data labels above the bar chart indicate the performance improvement of DR. As we can see, all applications can benefit from the proposed memory optimization scheme and the degree of improvement becomes better for denser device. The instruction per second can further increase under lower write back rate.

F. Power and Area Cost

The DR DRAM design requires moderate hardware addition, which can increase energy consumption as well. We measure system efficiency using throughput per watt (TPW). Fig. 18 shows our results.

Although all DR refresh mechanisms can decrease energy utilization, the degradation level can be acceptable. For DR-A and DR-B, the energy utilization is only a bit lower than our baseline, which has around 2% degradation. However, for original DR, the energy utilization can decrease by 9%. That is mainly because the bandwidth of original DR is far less than DR-A and DR-B.

We investigate area cost on different types of DIMM. For memory modules such as registered-DIMM, they already have additional devices to store error check code (ECC) [16]. They typically occupy 8 parity bits for 64 data bits. We can share and reuse these devices for DR. In this case, we can avoid introducing area overhead on *x8* or *x4* devices. The only thing is that the ECC function and the DR function cannot be enabled simultaneously.

For other solutions, the area overhead can be illustrated in Fig. 19. DIMM without ECC function (Unbuffered–DIMM and Small-Outline-DIMM) needs 25% additional area cost on a *x16* device and 12.5% additional area cost on a x8 device. For those DIMMs that already have ECC storage, if we don't want share additional device, it will introduce 20% additional area cost on a *x16* device and 11% additional area cost on a *x8* device.

VII. RELATED WORK

In this section we discuss prior works that are most relevant to our work and highlight our novelties. The novelties of our work are as follows: 1) We propose to use device refresh and data recovery to parallelize DRAM refresh and read operation. 2) We explored hardware design at the memory module level and we devise a new memory access protocol to control DR DRAM.

A. Refresh Optimization

These works can be classified into three categories:

1) *Software-based Design*

Refresh scheduling can be controlled at the software level. Isen and Hohn show that many rows in the DRAM do not store valid data. Therefore, these rows are unnecessary to be refreshed. They propose ESKIMO [17], which use malloc/free operation and DRAM hardware design to control refresh schedule. Similar ideas have been seen in different real-world designs [18]. Flikker et al. [19] divided DRAM into several areas that have different refresh rates. If a page is important, it will be allocated into high-refresh-rate area. Kotra et al. [9] used hardware/software co-design

approach to allocate DRAM storage. Unlike the above works, we focus on system hardware. Our design modifies the memory controller and DRAM module.

2) *Command Scheduling*

If a row has been precharged or activated recently, there is no need to be refreshed. DTail [20] uses low storage cost to track storage information and maximizes refresh reduction. Flexible-Auto-Refresh [21] combines access record and data retention time. RAIDR [22] records data retention time and classifies rows into different refresh rate. Adaptive Refresh and EFGE uses fine-grained refresh (FGR) [10] which can improve refresh efficiency on DDR4. Our technique is not refresh command scheduling based. We extend DRAM module and memory controller to implement DR.

3) *Microarchitecture Solution*

Optimizing refresh overhead at the microarchitecture level is also a widely used solution. Researchers from CMU and Intel try to parallelize refresh activities with data access through microarchitecture design on DRAM device [23]. Choi et al. [24] reduce enable Fast-Refresh through Multiple Clone Row DRAM technique. Recently, non-blocking memory refresh [25] also combines data recovery and additional chip to alleviate refresh overhead. Differently, this work focuses on detailed implementation at the protocol and hardware level. Besides, we proposed a new solution to overcome memory write problem.

B. Processing-In-DRAM

Many applications today rely on memory-intensive operations such as LLT and graph processing [26]. There have been prior works improve job performance through optimizing DRAM architecture or using Processing-in-Memory (PIM) technique. Gather Scatter-DRAM [27] can access stride address pattern within a single read/write command. Ambit [28] combines row amplifier with accelerator to perform bulk bitwise operation. Some other works [5] use Hybrid Memory Cube technology and add many logic modules. Our works does not offload any arithmetic operation to memory. We optimize memory access through eliminating refresh overhead on memory read transactions.

VIII. CONCLUSIONS

The memory IO bandwidth will become a crucial bottleneck for future memory-read-intensive applications. In this paper we propose DR DRAM, a novel memory design which can parallelize refresh operations and read operations. Besides, our approach avoids the huge modification on the software stack. We implement our design both at the memory module level and the memory protocol level. Using cycle-accurate simulators and various workloads, we show that DR improves memory read bandwidth by 12% and reduces latency overhead by 50~60%. We expect that MRI applications such as CNN, PageRank and LLT can greatly benefit from our design with affordable energy and area overhead.

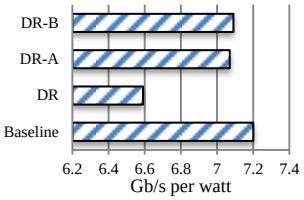


Figure 18: Energy utilization

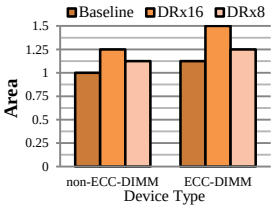


Figure 19: Normalized area

IX. ACKNOWLEDGEMENTS

This work is supported in part by the National Basic Research Program of China (973 Program, NO. 2015CB352403), the National Natural Science Foundation of China (No. 61502302, 61602301, 61632017, 61702328, 61741211, 61472260). Corresponding authors are Chao Li at Shanghai Jiao Tong University and Weigong Zhang at Capital Normal University.

REFERENCES

[1] Oracle, "When to Use Oracle Database In- Memory", Oracle *Write Paper*, 2015
[2] T. Ham et al., "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics", MICRO 2014
[3] Y. Wang et al., "Re-architecting the On-chip memory Sub-system of Machine-Learning Accelerator for Embedded Devices", ICCAD 2016
[4] H. Asghari-Moghaddam et al., "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems", MICRO 2016
[5] B. Hong et al., "Accelerating Links-list Traversal Through Near-Data Processing", PACT 2016
[6] T. Wang et al., "Hardware Supported Persistent Object Address Translation", MICRO 2017
[7] G. Chen et al. "Efficient Support of Position Independence on Non-Volatile Memory", MICRO 2017
[8] D. Xue et al., "Adaptive Memory Fusion: Towards Transparent, Agile Integration of Persistent Memory", HPCA 2018
[9] J. Kotra et al., "Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling", ASPLOS 2017
[10] JEDEC STANDARD, DDR4 SDRAM, November 2013
[11] P. Rosenfled et al., "DRAMSim2: A Cycle Accurate Memory System Simulator", *IEEE Computer Architecture Letters*, 2011
[12] Micron. 8Gb: x8, x16 Automotive DDR4 SDRAM, 2016
[13] N. Binkert et al., "The gem5 simulator", *ACM SIGARCH Computer Architecture News*, vol.30 no. 2,pp. 1-7, 2011
[14] J. Shun et al., "Ligra: A Lightweight Graph Processing Framework for Shared Memory", PPoPP 2013
[15] C. Zilles, "Benchmark Health Consider Harmful", *ACM SIGARCH Computer Architecture News*, vol. 29, no 3, pp. 4-5, 2001
[16] Micron. 4GB(x72, ECC,SR) 288-Pin DDR4 RDIMM, 2013
[17] C. Isen et al., "ESKIMO - Energy Savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem", MICRO 2010
[18] S. Song, "Method and System for Selective DRAM Refresh to Reduce Power Consumption," United States Patent 006094705
[19] S. Liu et al., "Flikker: Saving DRAM Refresh-power through Critical Data Partitioning", ASPLOS 2011
[20] Z. Cui et al., "DTail: A Flexible Approach to DRAM Refresh Management", ICS 2014
[21] I. Bhati et al., "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions", ISCA 2015
[22] J. Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh", ISCA 2012
[23] K. Chang et al., "Improving DRAM Performance by Parallelizing Refreshes with Accesses", HPCA 2014
[24] J. Choi et al., "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM", ISCA 2015
[25] K. Nguyen et al., "Nonblocking Memory Refresh", ISCA 2018
[26] P. Pan et al., "Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines", ICCD 2017
[27] V. Seshadri et al., "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses", MICRO 2015
[28] V. Seshadri et al., "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology", MICRO 2017