DRAM Workload Behavior

Mohd Arif Pandit
Department of Computer science and engineering
IIT-Ropar
Punjab, india
mohd.22csz0017@iitrpr.ac.in

***Abstract*—The high latency associated with accessing DRAM has become a bottleneck for various applications. In response, plenty of research works in recent past has been in this direction to alleviate this issue, including the ”copy row” CROW [2], an in-DRAM cache, which involves a flexible substrate-based caching mechanism within DRAM. However, there are still questions regarding the placement of copy rows within DRAM, the replacement policy employed by the in-DRAM cache, and the timing of caching new rows. In this project, we propose a new cache management policy called CROW++, which builds upon the CROW mechanism and aims to further reduce latency and refresh overhead. Our analysis identified a potential for improvement in the retention of frequently accessed rows within the in-DRAM cache. By implementing our proposed CROW++ cache management policy, our goal is to increase the efficiency of DRAM by better retaining frequently accessed rows inside the in-DRAM cache.**

***Index Terms*—DRAM, Memory subsystem, DRAM-cache, workload behaviour, cache policy**

I. Problem Statement

Several recent studies have revealed that the high latency associated with accessing DRAM has resulted in it becoming a bottleneck for the performance of various applications. According to the authors in [1], DRAMs are facing three key problems: high access latency, significant overhead associated with refreshing, and the impact of node scaling on increasing exposure to vulnerabilties. The third issue mentioned in the previous statement is somewhat connected to the scaling of technology nodes, and there exist methods such as error correcting codes (ECC) to address this problem. However, the primary concerns in the present DRAM architecture are the first and second issues discussed. A particular piece of research in [2], has put forward potential solutions to alleviate all three of the issues mentioned earlier. The authors in [2] hereafter as CROW have introduced a solution called ”copy row” (CROW), which involves a flexible substrate-based caching mechanism within the DRAM. This approach effectively reduces the access time for frequently accessed rows in the DRAM. Furthermore, it improves the retention time of rows by copying weaker rows into the in-DRAM cache also called as CROW-cache. The researchers have also suggested that the neighboring rows of frequently accessed ones should be copied into the in-DRAM cache to mitigate the rowhammer issue, thereby increasing the overall reliability of the DRAM. In this work, I have been following the methodology outlined in CROW, as well as analyzing the accompanying code. During this process, I have identified a few questions that I would like to address. Firstly, the authors have not provided explicit details regarding the placement of copy rows within the DRAM. Based on my own observations and research, I suggest that the rows closer to the row buffer will experience lower access latency, whereas those located farther away will experience higher latency, as discussed in [3]. However, the authors of CROW have not proposed a specific location for the copy rows, nor have they explored the tradeoffs associated with different placement strategies. Furthermore, I have noticed that in [4], the researchers move the copy rows to weak rows, which can potentially be located anywhere within the DRAM. However, there is no information available regarding the level of latency variation associated with the placement of these copy rows. Although I have not been able to find any relevant details regarding this issue in their available code or the research paper, I plan to investigate this further in my future work. Another question I have raised pertains to the replacement policy employed by the in-DRAM cache proposed in CROW. The authors have used an LRU (least recently used) replacement policy, which may not be the best approach for memory-bound or memory-intensive workloads. After analyzing the results, I suggest that a modified version of the replacement policy described in [5] may be more appropriate. Specifically, I propose a policy that prioritizes retaining frequently accessed rows within the in-DRAM cache. I arrived at this conclusion after analyzing the results of memory-intensive workloads, such as SPEC CPU2006 [6] 429.mcf workload, which indicated that frequently accessed rows were being evicted from the in-DRAM cache. My third question is related to the time of caching new rows within the in-DRAM cache. In CROW, rows are copied into the CROW-cache as soon as they are accessed. However, I propose that delaying this process and copying rows during the row-close time may yield different results. Unfortunately, there is no available command or specific timing window for performing copy operations at the PRE command. As a solution, I suggest keeping track of row-hits for recently accessed rows from a subarray row. Then, when a refresh operation is performed on the same subarray, we can copy the same row into the in-DRAM cache. This approach may yield varying results,

This work was carried as term paper project for course Memory system architecture at IIT Ropar Under Prop. Dr. T.V. Kalyan and I plan to explore this further in my future work. In my current research work, I introduced a new cache management policy for in-DRAM cache which I called as CROW++. This policy builds upon the existing CROW implementation and seeks to reduce both latency and refresh overhead even further. Through my analysis of the CROW mechanism, I identified a potential for improving the retention of frequently accessed rows within the in-DRAM cache. By implementing the proposed CROW++ cache management policy, my goal is to increase the efficiency of DRAM by better retaining frequently accessed rows within the in-DRAM cache.

II. Motivation for the solution

Dynamic random-access memory (DRAM) has long been the primary memory technology used in computing systems. However, with the increasing demand for high-performance computing and the explosion of data-driven applications, DRAM's limitations, such as high latency and refresh overhead, have become critical bottlenecks in system performance. Previous work, such as CROW, has proposed in-DRAM caching mechanisms to address these issues, but there is still room for improvement.

In recent work [2] also called as CROW, a flexible substrate called Copy-Row DRAM (CROW) was proposed as a means to enhance performance and reliability, while also reducing energy consumption. The architecture of Copy-Row DRAM (CROW) includes an in-DRAM cache, which functions as a cache memory. Unlike conventional DRAMs, which have only one local row-decoder inside a subarray, CROW has two local row-decoders. One local row-decoder is used for regular rows inside the subarray, while the other local row-decoder is reserved for the copy rows that are stored in the in-DRAM cache. The proposed idea behind CROW is to utilize the timing window for activating multiple rows, which was suggested in SALP [7]. In CROW, the number of rows within a subarray that will be used for the in-DRAM cache is defined by the user as a parameter. I have conducted several experiments using different workloads and varying numbers of copy rows, and observed that the relationship between the number of rows and performance is not consistent across all the workloads. Based on my analysis, I have found that using 8 copy rows per subarray provides the highest performance gains in CROW. The aim of the CROW was to reduce latency and refresh overhead in DRAM, and they were able to achieve this in some of the workloads. However, I believe that there is still potential for further performance improvement by reducing the access time for rows inside the subarray. It is important to limit the number of rows used for the in-DRAM cache as it adds overhead and reduces the number of available rows in the subarray. Suppose we have designated 8 rows as copy rows within a subarray and have opened 8 different rows in the same subarray. When a new request for a different row is received within the same subarray, we must remove one of the rows from the 8 copy rows based on the least recently used (LRU) policy in CROW. However, if a row is frequently accessed, it may be more efficient to keep it in the in-DRAM cache copy rows, rather than repeatedly accessing it from the regular rows inside the subarray. To address the issue of thrashing, I propose implementing a cache management policy that involves two decisions: one at the time of insertion and another at the time of replacement. By tracking the requests of rows inside each subarray, we can determine which rows are frequently accessed and should be added to the copy rows. This will help reduce the likelihood of thrashing occurring within the in-DRAM cache.

III. Proposed Idea

A. Working

In this work I am proposing a replacement policy for one recent good work for mitigating high access latency and refresh overhead. In the CROW authors have used LRU as replacement policy which after many experiments I observed that there is a possiblity of performance improvement if a better cache management policy is used to retain frequently accessed rows inside DRAM cache and reduce the thrashing of hot rows from CROW cache.

1) DRAM Organisation: DRAM-based main memory is structured hierarchically with the processor at the root level. The processor is connected to multiple channels, each of which has its own memory controller. Each channel is further divided into Dual inline memory modules (DIMMs), which are logically managed and accessed as ranks. Each DIMM has two sides, one being rank-1 and the other rank-2. These DIMMs consist of multiple DRAM devices, which are logically managed and accessed as banks for achieving parallelism. Within a bank, DRAM devices are divided into rows, which are horizontal collections of groups of DRAM cells. Each DRAM cell consists of a capacitor and an access transistor. To access a capacitor, a transistor is required, which is connected to bitlines and wordlines. Bitlines connect multiple DRAM cells in a column together with a sense amplifier inside the row buffer, and wordlines are used to select the access transistor of all DRAM cells in a particular row. Multiple DRAM cells horizontally make a row, and multiple rows are organized into subarrays. Multiple subarrays make a bank, and multiple banks make a rank, which in turn is controlled by a channel and its associated memory controller. Whenever a request for read/write is received from the processor, the memory controller segments the address into multiple subsections: channel, rank, bank, row, and column.

2) CROW: With CROW authors aim to address several challenges in DRAM, including high latency, refresh overhead, and the risk of rowhammer-induced reliability issues. Unlike other proposed solutions, CROW did not require any major changes to the existing DRAM architecture. Instead, it relied on the addition of a new decoder and a small amount of memory overhead to efficiently duplicate a select group of rows within a subarray at runtime.

CROW consists of two main components: a smaller group of cells within each subarray where a duplicate copy of the currently accessed row is stored, and a table maintained at the memory controller (known as the CROW-table) for bookkeeping and cache management policies. CROW divides the DRAM architecture into two parts: regular rows and copy rows, each of which is accessed by its own row decoder. This separation allows for the possibility of opening two rows simultaneously, also known as multiple row activation (MRA).

When a new request arrives in a subarray, if a row buffer miss or row buffer conflict occurs, meaning a new row needs to be accessed from the subarray, the memory controller first activates the regular row. Once the data is latched into the local row buffer, the copy row is also activated. As the bitlines are driven by sense amplifiers inside the row buffer, any new row that is opened will receive the data properly, transferring the charge from sense amplifiers. This way, the data from the source row is copied into the copy rows from the local row buffers.

The CROW technique is based on the operation of DRAM cells. The DRAM cell operation can be broken down into several phases. In the first phase, known as the precharged state, the bitlines are in a precharged state and are maintained at V_{dd}/2. The worldline, which controls the bitlines for a particular row, is set to 0V in the precharged state. To access a cell, an ACT (Activate) command is applied to the particular row, which raises the worldline voltage from 0V to V_h and connects the capacitors to the bitlines. Depending on the charge state of the capacitor, the bitlines will be driven towards V_{dd}/2+δ or V_{dd}/2-δ. The second phase is called charge sharing. During this phase, sense amplifiers inside row-buffers are enabled to detect deviations on the bitlines. These deviations are then amplified towards V_{dd} or ground. After amplification, a memory controller can issue READ/WRITE commands to access data from the latched data inside the row-buffer. The timing constraint defined by t_{RCD} determines when a READ/WRITE command will be issued after applying the ACT command. The fourth phase is the fully-restored state, where both the bitline and the cell are at the same voltage level. The time taken to reach this state is defined by t_{RAS}, which means that all the bitlines and DRAM cells of the accessed row are at some level. At this stage, the bitlines can be precharged using the PRE command to prepare access to a different row. The timing for the PRE command is determined as t_{RP}.

In summary, there are four main commands used in dynamic random access memory (DRAM): ACT, WR, RD, and PRE. These commands are scheduled by the memory controller using a state machine and timing parameters. The ACT command opens a row inside the row buffer, which can be accessed later using WR and RD commands. Finally, the PRE command is used to close the row. Additionally, the REF command is used to mitigate capacitor leakage. It involves applying a periodically scheduled ACT+PRE, called REF, to rows after their retention time expires, which is typically within milliseconds.

In the context of CROW (in-DRAM cache), two new commands have been proposed: ACT-c and ACT-t. These commands are used to implement the proposed in-DRAM cache. To copy an opened row from the row buffer to the CROW cache, an ACT-c command is used instead of the ACT command, which has higher latency. The ACT-c command uses the same mechanism as RowClone. First, a regular row is accessed, and after the data is latched in the row buffer, a copy row is opened inside the CROW cache. By ”opened,” we mean that all the DRAM cells inside a row are connected to bitlines, which are further connected to sense amplifiers. This way, the data gets copied from the regular row to the CROW cache.

A CROW table is maintained inside the memory controller to store the details of cached rows. When a new row is opened, the CROW table is accessed to determine whether it is available inside the CROW cache. If present, instead of using the ACT command to open the row, ACT-t command (Activate two-rows) is used, which has much lower latency than the ACT or ACT-c (copy regular to copy rows) command. This reduces the access time of the rows in DRAM. Therefore, the more ACT-t commands used for row access, the lower the latency will be. To have more ACT-t commands for row access, frequently accessed rows should be retained inside the CROW cache. This forms the basis of the current work, which proposes retaining frequently accessed rows in the DRAM cache.

3) CROW++: I am proposing a cache management policy for CROW cache to improve the performance of the CROW. This policy involves cache insertion and replacement strategies. The cache insertion policy involves deciding where to copy newly accessed open rows in a subarray. There are three decisions involved in this policy: 1) Where to copy the newly inserted row: there are multiple options, and the decision needs to be fixed as it can affect performance. 2) What to evict when there is no vacant space inside the cache: if there is no space available in the cache, a decision needs to be made on which row to evict to make room for the new one. 3) How to change the priorities of rows inside the cache when there is a hit: when a row is hit inside the cache, the policy needs to change the priorities of the other rows. In CROW, LRU is used as the replacement policy. However, this policy assumes that every new row will be re-referenced in the distant future, which is not always the case. As a result, some rows are predicted as being the least recently used and are evicted from the cache even though they might have access in the near future. To address this issue, we propose maintaining a table that stores the subarray row address. When a row is accessed and added to the CROW cache, we store its address and set the count to 1. If the row is evicted from the cache but has future access, we consider it as thrashing, and the row is retained in the proposed table. The table keeps incrementing the count whenever the row is accessed again. We propose using a parameter called to_mru, which is guided by the table. On frequent access, this parameter goes from 0 to 1. A value of 0 means inserting a row at the LRU position, while 1 means inserting a row at the MRU position. The intermediate values of to_mru represent the probability of inserting a row at the MRU position. Every time a new row is inserted into the cache, we decide the to_mru variable based on the proposed table. We also calculate another parameter called ”r” randomly between 0 and 1. We compare this value against to_mru, and if (to_mru $>$ r), we insert the row at the MRU position. Otherwise, we insert it at the LRU position. We also propose adding a ”retain” bit to the CROW table. This bit will be set to 1 if a row crosses a certain threshold of frequent access. We can use this bit to retain frequently accessed rows inside the cache and remove them from the cache management policy. This reduces the unnecessary load on the cache management policy and reduces the number of ACT-c commands issued. By caching these rows inside the cache, we can issue more ACT-t commands, which helps us reach our proposed research goal.

The difference between two works CROW with LRU policy and proposed CROW++ with modified version of RRIP [5] cache policy can be dipicted from Figure 1 and Figure 2 respectively. As can be seen clearly in figure 1, we are not able to retain row-d and row-e so we will be consuming higher latency by using ACT-c commands instead of ACT-t as we are not able to retain them and they get thrashed from in-DRAM cache because of the LRU policy was not able to retain them. Now with the proposed modified RRIP policy over CROW also we call it as CROW++, where by using proposed procedure as discussed we are able to retain row-d and row-e by tracking these using crow-life-time table which allows to guide cache insertion policy using to_mru variable, so now we are able to retain row-d and row-e and are able to reduce the access latency to these frequently accessed rows

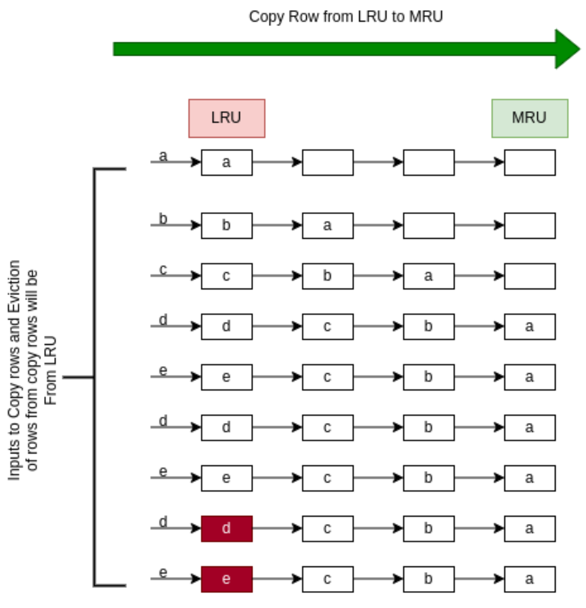
Copy Row from LRU to MRU
LRU
MRU
Inputs to Copy rows and Eviction of rows from copy rows will be From LRU
a
b
c
d
e

Fig. 1. CROW LRU Policy

4) Implementation:

a) CROW: CROW is based on the idea of SALP [7] MRA (Multiple Row Activation). In CROW, two rows are opened simultaneously during a memory access: the row requested by the processing core and another row from the CROW cache, which is opened to receive the data if it’s not already present in the cache or to supply the data if the requested row is cached in the CROW cache. To open two rows in one ACT (Activate) command, the authors utilized two row decoders within a subarray. One local row decoder is used to select from regular rows, while the other local row decoder selects a copy row from small number of rows, called copy rows. The additional local row decoder for copy rows is lightweight and doesn’t add much hardware overhead as it’s driving only a small portion of the rows. To drive two local row decoders, the authors proposed a new ACT-c command over the baseline ACT command [8]. The ACT-c command provides two addresses, one for the local row decoder for the regular row and another for selecting one of the copy rows. When a row is opened in the row buffer, its entry is made in the CROW table. Subsequent accesses to the same row are checked against the CROW table to determine if the row exists in the CROW cache. If the row isn’t present in the cache, an ACT-c command is used, as discussed earlier. If the row is present in the cache, a new ACT-t command is used, which works the same way as the ACT-c command, but with lower latency. This is because two rows are driving the bitlines, which reduces the bitline capacitance and allows for faster sensing amplifier drive towards V_{dd}, reducing the latency.

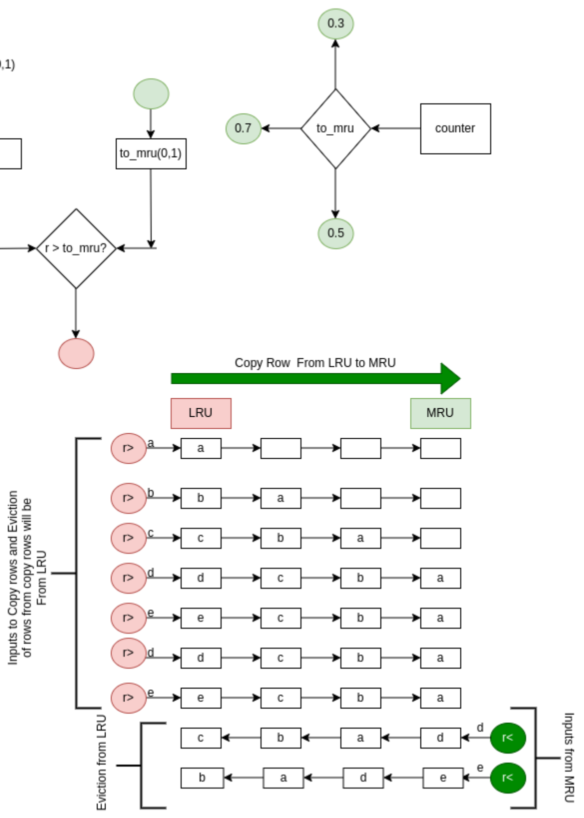

Fig. 2. CROW++ (CROW with an updated cache policy)

TABLE I
BENCHMARK DETAILS

Benchmark	**Application Type**	**Workload Type**	**DRAM Behavior**
429.mcf	**Vehicle routing problem**	**Low memory intensity**	**Accesses a large amount of memory, but with low memory intensity.**
433.milc	**Lattice quantum chromodynamics**	**Moderate memory intensity**	**Accesses a moderate amount of memory with moderate memory intensity.**
470.lbm	**Fluid dynamics**	**High memory intensity**	**Accesses a large amount of memory with high memory intensity.**
459.GemsFDTD	**Electromagnetic simulations**	**High memory intensity**	**Accesses a large amount of memory with high memory intensity.**
436.cactusADM	**General relativistic astrophysics**	**High memory intensity**	**Accesses a large amount of memory with high memory intensity.**

The CROW-table is a data structure maintained in the memory controller. It has two primary functions: (1) to store details of recently accessed rows that are allocated a copy row, and (2) to store the addresses of weak rows which are remapped to copy rows to reduce the refresh overhead. The CROW-table is indexed using a combination of the bank and subarray address, which are calculated by the memory controller from the requested address. When a request comes to the memory controller from a core, the memory controller generates an address that includes channel, rank, bank, row, and column details. The memory controller then checks whether the requested row entry exists in the CROW-table. Based on this information, the memory controller decides whether to use the ACT-c or ACT-t command. The ACT-c command takes longer latency than the ACT-t command. The CROW-table maintains details such as the row address, a valid bit, and other special fields. By storing this information, the memory controller can quickly access the required row and optimize the access time, thereby reducing latency and improving the performance of the system.

b) CROW++: The proposed work involves using a robust cache policy for the in-DRAM cache, which is implemented by leveraging the existing CROW architecture. The new cache policy comes in the form of a storage overhead to be maintained at the memory controller. The proposal introduces a new table called ”crow-life-time,” which keeps track of the details of rows that are cached inside the CROW cache. The count of a particular row inside the crow-life-time table is incremented on frequent accesses. Based on this count, the cache policy decides whether the row needs to be inserted at the most recently used (MRU) position or the least recently used (LRU) position. The proposed implementation requires two registers one for ”to_mru” as discussed in the previous section, and another for a randomly generated value. This value provides randomness in the insertion process, which helps to ensure that the copy rows are evenly distributed inside all subarrays. Overall, the proposed approach aims to optimize the in-DRAM cache by efficiently managing the frequently accessed rows in each subarray and improving the cache insertion policy

IV. EXPERIMENTAL SETUP

A. Simulator

CROW has been incorporated into Ramulator, a fast and precise DRAM simulator that features extensibility support. This simulator offers a generalized template that enables the modeling of various DRAM systems based on their architectural specifications. This is possible due to Ramulator’s decoupled and modular design. Ramulator supports a wide range of DRAM systems such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, as well as academic research proposals like SALP and Rowclone, making it a popular tool for experimentation. Organizations can use Ramulator to analyze new DRAM standards before actual production. Ramulator is primarily used in academic research where researchers experiment with DRAM systems to propose new ideas and demonstrate performance improvements. Ramulator’s design is hierarchical, with a generalized DRAM module serving as the base, and specific modules like DDR3/4 and LPDDR3/4 extending it. Ramulator provides the flexibility to reconfigure DRAM for different standards, and at compile time, users can choose which DRAM standard to experiment with. Ramulator features a memory controller that manages the channels and provides an interface for sending and receiving memory requests. Ramulator is available in standalone and integrated versions with gem5 and works with both CPU trace and memory. It is written entirely in C++11, making it an accessible tool for computer science researchers interested in experimenting with DRAM.

B. Benchmarks

In CROW authors have experimented with different workloads from SPEC CPU2006 benchmark [6]. It is a suite of standardized benchmarks developed by the Standard Performance Evaluation Corporation (SPEC) to evaluate the performance of CPUs and compilers. The benchmark includes a set of synthetic programs that represent real-world applications and cover a wide range of computing tasks, including scientific computing, media encoding, and database management. The SPEC CPU2006 benchmark measures the performance of both single-threaded and multi-threaded applications and provides a standardized way to compare the performance of different hardware platforms and software configurations. I have covered details of some of the benchmarks in Table I. I differentiated them based on workload type, and DRAM behaviour. Workload Type refers to the amount of memory accessed and the frequency of those accesses, and is categorized as low memory intensity, moderate memory intensity, or high memory intensity. DRAM behavior refers to how the benchmark accesses DRAM, and is categorized as low memory intensity, moderate memory intensity, or high memory intensity, based on the amount of memory accessed and the frequency of those accesses.

TABLE III
SOTA WORKS

Paper-Title	Author	Publisher, year	Novelty
EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM	Koppula et al.	ACM, 2019	The paper presents EDEN, a framework that uses approximate DRAM to improve energy efficiency and performance of DNN inference while meeting user-specified target accuracy.
BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly -Accessed DRAM Rows	Giray et al.	HPCA, 2021	This work describes a new hardware-based solution called BlockHammer to mitigate RowHammer attacks in DRAM memory.
A Modern Primer on Processing in Memory	Mutlu et al.	Emerging Computing: From Devices to Systems, 2022	This article explores the concept of processing-in-memory (PIM) as a solution to reduce data movement between computation units and memory by placing computation mechanisms near where data is stored.
CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations	Orosa et al.	ISCA, 2021	The article introduces CODIC, a new DRAM substrate that allows for fine-grained control over DRAM circuit timings, enabling new features and custom optimizations.

C. Configuration

As with availability of cycle-accurate simulators like ramulator [8] or DRAMSIM3 [9] we are able to experiment with various configurations with ease, also I have tried to evaluate my work by varying the configuration parameters some of the important configurations parameters used in my experiments are provided in Table II.

D. Hardware overhead

In this work, as I have provided one more table which I am referring as crow-life-time table, where in at the entry of a row inside CROW-cache I am maintaining a count which will get Incremented in further addition of same row in CROW-cache, in turn this count will guide the insertion policy where to insert the new row either at LRU or MRU. The proposed Idea comes with a hardware overhead which is equal to ($\log_2$(subarray_addr) + $\log_2$(counter)) * No. of subarrays * No. of banks * No. of ranks * No. of channels. In current configuration this equals to 12-bits (bits used for address + counter) * 2^7 (subarrays) * 2^3 (banks) * 2^0 (ranks) * 2^0 (channel) which comes as 1KB Approx.

E. Recent Works

I have discovered that there are no recent studies that have improved upon CROW to further reduce latency or refresh rates, likely due to the significant hardware overhead and substrate changes involved. However, this work has been extensively cited by state-of-the-art papers that explore in-memory computing, near data processing, and row-hammer mitigation. I have summarized some of the most notable studies in Table III.

V. RESULTS AND ANALYSIS

A. Experiment 1

I conducted an experiment on a DRAM standard with default architecture, without any subarray level implementation or in-DRAM cache, using Ramulator. Ramulator allows us to choose the DRAM standard we want to experiment with at compile time. I used the same configuration as in the analysis of CROW and proposed CROW++, with the only difference being that we did not use subarrays or in-DRAM cache. Various workloads such as 429.mcf, 433.milc, and others were run, and I compared them with different row buffer policies. However, I found that there was no consistent behavior among the workloads on running in Ramulator baseline. I choose LPDDR4 DRAM standard, as it was used in the analysis of CROW and proposed CROW++. The reason for conducting this experiment was to analyze how workloads perform in baseline, and as seen, their performance was not consistent. I later found that the same inconsistency was found with CROW and CROW++. I was trying to determine which row buffer policy would be optimal for CPU benchmark workloads, but nothing can be said in general as it shows varying nature. As can be seen in Figure 3, only 429.milc and 462.libquantum shows the maximum difference in performance. The outcome of this experiment, can be proposed as there is not specific row-buffer policy which is showing consistency in SPEC CP2006 benchmark, one policy say closed is having best performance in 436.cactusADM workload while closed policy is performance worst in case of 462.libquantum, as can be depicted in Figure 3.

B. Experiment 2

Another Experiment I did by running various workloads from SPEC CPU2006, the intention here was to see how the performance varies by running the same workloads on various DRAM standards such as DDR3, DDR4, LPDDR3, LPDDR4, GDDR5, HBM, SALP, WIDE IO , WIDE IO2. Where all are Industry standards without SALP [7] which is an academic proposal for performance Improvement over conventional DRAM by utilising the subarrays. As can been seen in the Figure 4, HBM is best performing DRAM standard for SPEC CPU2006 benchmarks, followed by WIO, and WIO2. These technologies have high memory bandwidth, low latency, and power consumption, which can improve performance for CPU-bound workloads in different system configurations. However

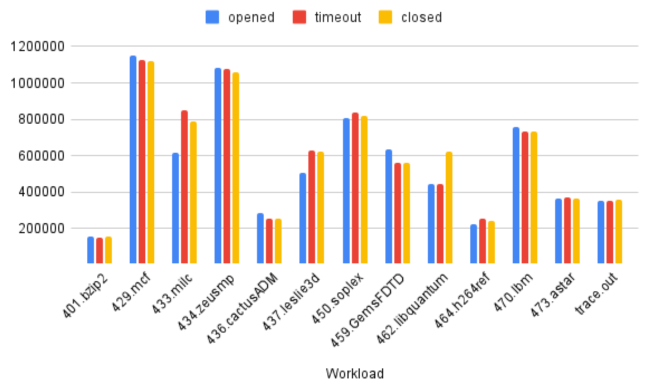

Fig. 3. Workload behaviour over various row-policies in baseline

TABLE II
CONFIGURATION PARAMETERS

Parameter	Value
org	LPDDR4_3200
cpu_tick	5
mem_tick	2
L1, L2	off
l3_size	4194304
channels	1
ranks	1
banks	8
rows	65536
subarrays	128
subarray_size	512
enable_crow_upperbound	false
row_policy	closed/opened/timeout
copy_rows_per_SA	1/2/4/8/16/32/64/128/256
weak-rows_per_SA	0/3
num_track_rows	2/4/8

LPDDR3, LPDDR4, DDR3, DDR4 and GDDR5 are more traditional DRAM standards, designed for use in desktop and server systems. The main reason for HBM and WIO and WIO2 showing better performance over other standards is HBM is utilising more channels which increases the bandwidth and reduces the latency of frequent DRAM accesses.

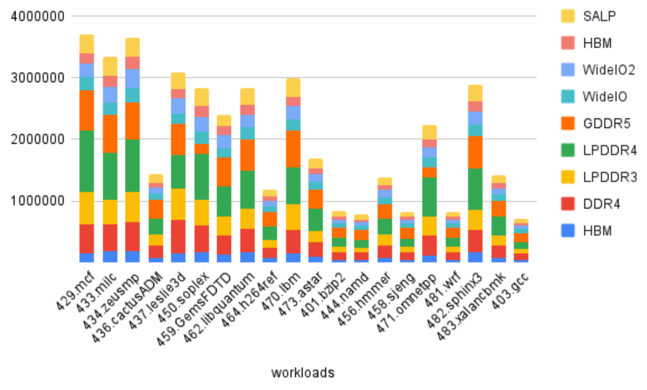

Fig. 4. workload-behavior in Baseline with various DRAM Standards

C. Experiment 3

Experiment third I have performed by comparing the SALP, TL-DRAM and COPY-rows, I have all are performing equally well, means all workloads are showing consistent performance in terms of number of cycles taken per workload, so we can say any type of performance academic proposals are equally well be it SALP, TL-DRAM, or CROW-cache, but the concern remain how well they can be actually implemented in DRAM chip, means the technological aspects are not covered in these works, and major works are ignoring the hardware overhead and the delay, power consumption, area and cost associated with these academic proposals for performance improvement, I am proposing in continuation to this work in future I will implementing all my Ideas in FPGA emulation environment supported with timing calculation from SPICE simualtions, as it will give better hand on device side also, current work and all previous works like SALP, TL-DRAM, CROW, are showing performance improvment without taking consideration into hardware side. The results of this experiment though were not much informative as all are perfoming equally well over SPEC CPU2006, the results are in Figure 5.

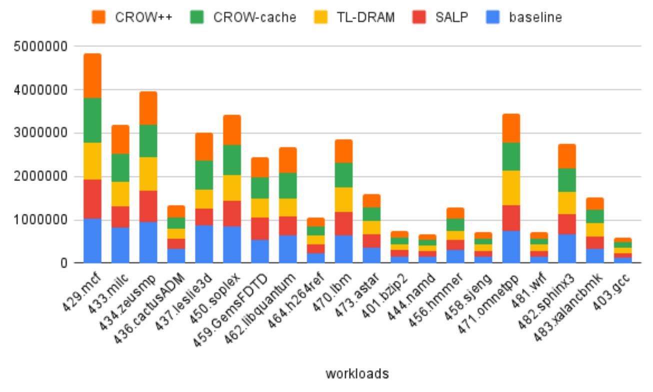

Fig. 5. workload-behavior over SALP, TL-DRAM, CROW, and CROW++

D. Experiment 4

In Experiment 4, I analyzed the proposed CROW++ cache management policy, which is a modified version of the CROW + modified cache policy. The main objective of this policy is to retain frequently accessed rows in the in-DRAM cache, which the current cache management policy is unable to achieve. I discovered that due to high subarray level parallelism, the rows in the cache are distributed widely among the subarrays, and no workload was found to overload requests to a particular subarray. Therefore, the current proposed cache management policy, CROW++, is not able to improve over all the workloads as the role of cache policy doesn't come into play. So with my present cache management policy, I was able to show performance improvements in two workloads, namely 429.mcf and 471.omnetpp. I was able to increase the CROW-cache hits with reduced number of cycles. However, due to the reasons mentioned above, there were no performance improvements in other workloads. I presented some of the results in Figures 6, 7, 8, and 9.

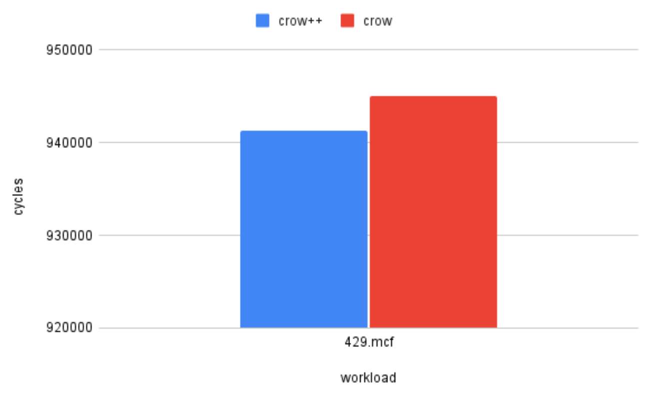

Fig. 6. workload-behavior over CROW , CROW++

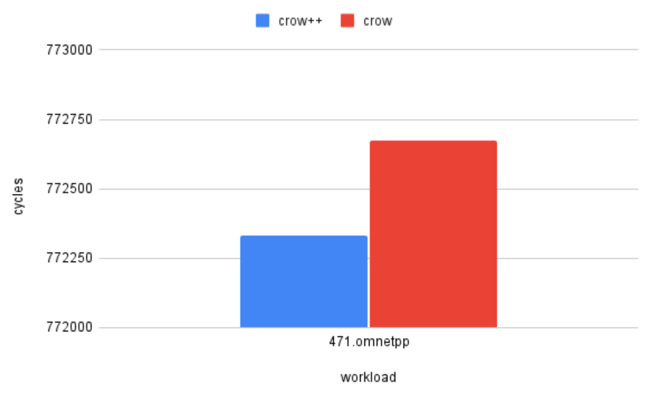

Fig. 7. workload-behavior over CROW , CROW++

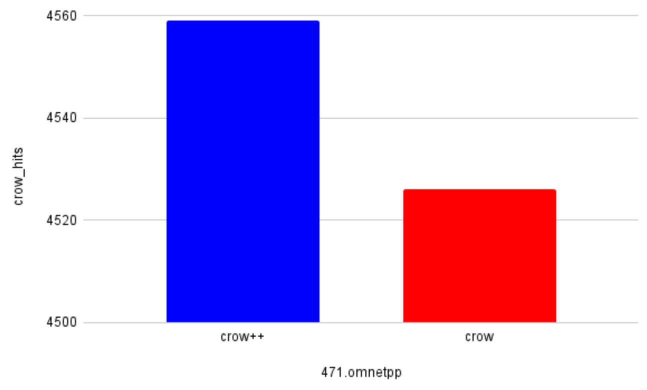

Fig. 8. workload-behavior over CROW , CROW++

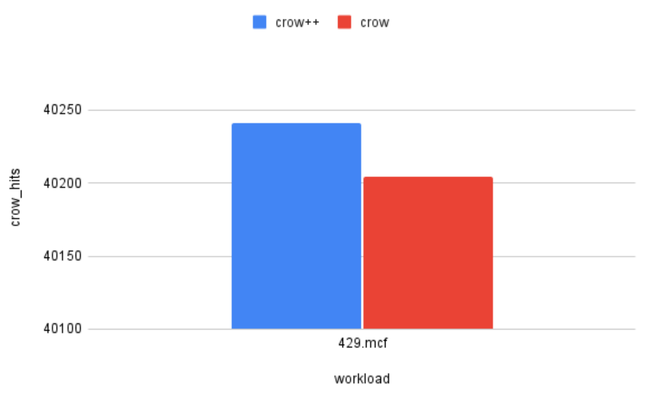

Fig. 9. workload-behavior over CROW , CROW++

VI. Conclusion

In this work, I began my exploration with a remarkable study, as presented in [1], where the authors comprehensively investigated the functioning of DRAM. They thoroughly analyzed the performance of various DRAM types, including DDR3, DDR4, LPDDR4, GDDR5, HBM, among others, under different workload conditions. In this study, I attempted to generate some results using a limited number of benchmarks due to their unavailability. However, I was able to replicate the findings of previous research, which indicated that DDR4 may not always outperform DDR3. The primary focus of this work was to address the three major challenges faced by conventional DRAM standards, namely higher access latency, high refresh overhead, and reliability issues. A similar issue was highlighted in a previous study [2], where the authors proposed a new substrate known as in-DRAM or CROW-cache, which is maintained inside subarrays to store frequently accessed rows. I aimed to reproduce their results, particularly regarding performance, and some of my findings can be found in the previous section, experiment-2. As part of my research direction, I raised several questions regarding potential improvements to the existing work. One of the primary questions I posed was where the copy-rows should be placed within the subarray. Previous studies, such as [3], have suggested that the bitline capacitance plays a critical role in determining the access latency of rows in DRAM. Specifically, shorter bitlines lead to lower latency. However, the authors of the original work did not mention where the copy-rows were placed in relation to the row-buffer where sense amplifiers are maintained. Another study, [4], proposed using weak rows as copy-rows, but did not provide any circuit simulation data to determine the impact of moving the copy-rows within the subarray. As this would require circuit simulation, As such, this will be a focus of my future work. Another question that I raised pertains to the timing of copy operations inside the in-DRAM cache. Instead of copying a row into the CROW-cache during the row-open or ACT command, I proposed delaying the operation until the future and implementing a hit counter for recently accessed rows. During the refresh operation, this hit counter would be compared against a threshold, and a decision would be made regarding whether to copy the row into the CROW-cache or not. This approach would enable rows to be added to the CROW-cache based on the number of hits they receive, rather than copying every newly opened row into the cache. The reason for scheduling this operation during the refresh (REF) window is that during this window, we perform the ACT+PRE command to a row, and there is sufficient time to precharge the copy rows as well. I started working on this idea, but encountered difficulties because I was unable to determine the appropriate delay to increase in the REF window to make this operation possible. Attempting to perform the operation directly during the REF window caused unusual output effects, indicating that the timing constraints were being violated. Therefore, I will be conducting SPICE simulations to calculate the timing for copy operations in the REF window, and using these results to refine the proposed approach. For now, this idea is a part of my future study. The third question I raised pertains to the number of copy rows that are kept inside the subarray. Rather than going for higher numbers of copy rows, such as 8 or more, which can raise bookkeeping complexity and reduce the number of available rows within the subarray, I suggested fixing the number of copy rows to a lower value, such as 2 or 4. However, this can lead to a problem of row thrashing, where a cold row is kept inside the in-DRAM cache while a hot row is replaced. To address this issue, I proposed a modified version of the cache policy, which I call CROW++ (CROW with an updated cache policy). This policy aims to retain frequently accessed rows inside the CROW-cache, and with higher subarray array level parallelism, it was observed to be effective in only some benchmarks. With this approach, I was able to retain hot rows inside the in-DRAM cache, reduce row thrashing, and ultimately reduce the number of cycles needed.

In this study, I have concluded that among the conventional DRAM types, HBM provides the best performance for SPEC CPU2006. Other DRAM types and academic proposals such as SALP [7], TL-DRAM [10], and CROW [2] did not show a significant impact on the SPEC CPU2006 benchmark results. However, if CROW is coupled with in-memory computing, it could be a better alternative to conventional DRAM architectures.

Acknowledgment

I would like to acknowledge the invaluable support and guidance provided by Respected Prop. Dr. T.V. Kalyan and Teaching-Assistant Respected Saurabh Jaiswal throughout my research work on DRAM workload behavior. Their expertise and encouragement have been instrumental in shaping my ideas and methodology, and their mentorship has been truly inspiring. I am grateful for their patience and dedication, which have made this paper possible.

References

[1] Ghose, Saugata, et al. ”Demystifying complex workload-dram interactions: An experimental study.” Proceedings of the ACM on Measurement and Analysis of Computing Systems 3.3 (2019): 1-50.

[2] Hassan, Hasan, et al. ”Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability.” Proceedings of the 46th International Symposium on Computer Architecture. 2019.

[3] Hassan, Hasan, et al. ”ChargeCache: Reducing DRAM latency by exploiting row access locality.” 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016.

[4] Kumar, Sudershan, Prabuddha Sinha, and Shirshendu Das. ”WinDRAM: Weak rows as in-DRAM cache.” Concurrency and Computation: Practice and Experience 34.28 (2022): e7350.

[5] Jaleel, Aamer, et al. ”High performance cache replacement using re-reference interval prediction (RRIP).” ACM SIGARCH computer architecture news 38.3 (2010): 60-71.

[6] Henning, John L. ”SPEC CPU2006 benchmark descriptions.” ACM SIGARCH Computer Architecture News 34.4 (2006): 1-17.

[7] Kim, Yoongu, et al. ”A case for exploiting subarray-level parallelism (SALP) in DRAM.” ACM SIGARCH Computer Architecture News 40.3 (2012): 368-379.

[8] Kim, Yoongu, Weikun Yang, and Onur Mutlu. ”Ramulator: A fast and extensible DRAM simulator.” IEEE Computer architecture letters 15.1 (2015): 45-49.

[9] Li, Shang, et al. ”DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator.” IEEE Computer Architecture Letters 19.2 (2020): 106-109.

[10] Lee, Donghyuk, et al. ”Tiered-latency DRAM: A low latency and low cost DRAM architecture.” 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2013.