CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability

Hasan Hassan[†] Minesh Patel[†] Jeremie S. Kim[†§] A. Giray Yaglikci[†]
Nandita Vijaykumar[†§] Nika Mansouri Ghiasi[†] Saugata Ghose[§] Onur Mutlu[†§]

[†]*ETH Zürich* [§]*Carnegie Mellon University*

ABSTRACT

DRAM has been the dominant technology for architecting main memory for decades. Recent trends in multi-core system design and large-dataset applications have amplified the role of DRAM as a critical system bottleneck. We propose Copy-Row DRAM (CROW), a flexible substrate that enables new mechanisms for improving DRAM performance, energy efficiency, and reliability. We use the CROW substrate to implement 1) a low-cost in-DRAM caching mechanism that lowers DRAM activation latency to frequently-accessed rows by 38% and 2) a mechanism that avoids the use of short-retention-time rows to mitigate the performance and energy overhead of DRAM refresh operations. CROW's flexibility allows the implementation of both mechanisms at the same time. Our evaluations show that the two mechanisms synergistically improve system performance by 20.0% and reduce DRAM energy by 22.3% for memory-intensive four-core workloads, while incurring 0.48% extra area overhead in the DRAM chip and 11.3 KiB storage overhead in the memory controller, and consuming 1.6% of DRAM storage capacity, for one particular implementation.

KEYWORDS

DRAM, memory systems, performance, power, energy, reliability

1 INTRODUCTION

DRAM has long been the dominant technology for architecting main memory systems due to the high capacity it offers at low cost. As the memory demands of applications have been growing, manufacturers have been scaling the DRAM process technology to keep pace. Unfortunately, while the density of the DRAM chips has been increasing as a result of scaling, DRAM faces three critical challenges in meeting application demands [78, 82]: (1) high access latencies and (2) high refresh overheads, both of which degrade system performance and energy efficiency; and (3) increasing exposure to vulnerabilities, which reduces the reliability of DRAM.

First, the high DRAM access latency is a challenge to improving system performance and energy efficiency. While DRAM capacity increased significantly over the last two decades [6, 35, 37, 57, 58, 105], DRAM access latency decreased only slightly [6, 58, 82]. The high DRAM access latency significantly degrades the performance of many workloads [14, 23, 28, 30, 123]. The performance impact is particularly large for applications that 1) have working sets exceeding the cache capacity of the system, 2) suffer from high instruction and data cache miss rates, and 3) have low memory-level parallelism. While manufacturers offer latency-optimized DRAM modules [72, 98], these modules have significantly lower capacity and higher cost compared to commodity DRAM [8, 53, 58]. Thus, reducing the high DRAM access latency *without trading off capacity and cost* in commodity DRAM remains an important challenge [17, 18, 58, 78].

Second, the high DRAM refresh overhead is a challenge to improving system performance and energy consumption. A DRAM cell stores data in a capacitor that leaks charge over time. To maintain correctness, *every* DRAM cell requires periodic *refresh* operations that restore the charge level in a cell. As the DRAM cell size decreases with process technology scaling, newer DRAM devices contain more DRAM cells than older DRAM devices [34]. As a result, while DRAM capacity increases, the performance and energy overheads of the refresh operations scale unfavorably [7, 40, 64]. In modern LPDDR4 [73] devices, the memory controller refreshes *every* DRAM cell every 32 ms. Previous studies show that 1) refresh operations incur large performance overheads, as DRAM cells *cannot* be accessed when the cells are being refreshed [7, 64, 76, 84]; and 2) up to 50% of the total DRAM energy is consumed by the refresh operations [7, 64].

Third, the increasing vulnerability of DRAM cells to various failure mechanisms is an important challenge to maintaining DRAM reliability. As the process technology shrinks, DRAM cells become smaller and get closer to each other, and thus become more susceptible to failures [56, 64, 65, 70, 78, 91, 118]. A tangible example of such a failure mechanism in modern DRAM is RowHammer [52, 79, 80]. RowHammer causes *disturbance errors* (i.e., bit flips in vulnerable DRAM cells that are not being accessed) in DRAM rows physically adjacent to a row that is repeatedly activated many times.

These three challenges are difficult to solve efficiently by directly modifying the underlying cell array structure. This is because commodity DRAM implements an extremely dense *DRAM cell array* that is optimized for *low area-per-bit* [57, 58, 105]. Because of its density, even a small change in the DRAM cell array structure may incur non-negligible area overhead [58, 78, 112, 119]. **Our goal** in this work is to lower the DRAM access latency, reduce the refresh overhead, and improve DRAM reliability with *no changes* to the DRAM cell architecture, and with only *minimal* changes to the DRAM chip.

To this end, we propose *Copy-Row DRAM (CROW)*, a flexible in-DRAM substrate that can be used in multiple different ways to address the performance, energy efficiency, and reliability challenges of DRAM. The key idea of CROW is to provide a fast, low-cost mechanism to *duplicate select rows* in DRAM that contain data that is most sensitive or vulnerable to latency, refresh, or reliability issues. CROW requires only small changes in commodity DRAM chips. At a high level, CROW partitions each DRAM subarray into two regions (*regular* rows and *copy* rows) and enables independent control over the rows in each region. The key components required for independent control include 1) an independent decoder for the copy rows and 2) small changes in the memory controller interface

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322231

to address the copy rows. To store information about the state of the copy rows (e.g., which regular rows are duplicated in which copy rows), CROW implements a small CROW-table in the memory controller (Section 3.3).

CROW takes advantage of the fact that DRAM rows in the same subarray share *sense amplifiers* to enable two types of interaction between a regular row and a copy row. First, CROW can activate (i.e., open) a copy row *slightly after* activating a regular row in the same subarray. Once the copy row is activated, the charge restoration operation in DRAM (used to restore the charge that was drained from the cells of a row during activation) charges both the regular row *and* the copy row at once. The simultaneous charge restoration of the two rows copies the contents of the regular row into the copy row, similar to the in-DRAM row copy operation proposed by RowClone [100]. Second, CROW can *simultaneously* activate a regular row and a copy row that contain the *same data*. The simultaneous activation reduces the latency required to access the data by driving each sense amplifier using the charge from *two* cells in each row (instead of using the charge from only one cell in one row, as is done in commodity DRAM).

The CROW substrate is flexible, and can be used to implement a variety of mechanisms. In this work, we discuss and evaluate two such novel mechanisms in detail: 1) *CROW-cache*, which reduces the DRAM access latency (Section 4.1) and 2) *CROW-ref*, which reduces the DRAM refresh overhead (Section 4.2). We also briefly discuss a third mechanism that mitigates RowHammer errors (Section 4.3).

Reducing DRAM Access Latency with CROW-cache. Prior works [26, 39, 116] observe that a significant fraction of the most-recently-accessed rows are activated again soon after the rows are closed (i.e., the rows exhibit high reuse). To take advantage of the reuse of recently-accessed rows, we use CROW to implement CROW-cache, which reduces the DRAM access latency when a recently-accessed row is activated again. **The key idea of CROW-cache** is to 1) duplicate data from recently-accessed regular rows into a small cache made up of copy rows, and 2) simultaneously activate a duplicated regular row with its corresponding copy row when the regular row needs to be activated again. By activating both the regular row and its corresponding copy row, CROW-cache reduces the time needed to open the row and begin performing read and/or write requests to the row by 38%.

Reducing DRAM Refresh Overheads with CROW-ref. Prior works [24, 41, 64, 65, 87, 88] show that only a small fraction of DRAM cells in a DRAM chip (e.g., fewer than 1000 cells in 32 GiB of DRAM [64]), called *weak* cells, have to be refreshed at the standard-specified refresh interval (32 ms in LPDDR4 [36], 64 ms in DDR3/DDR4 [35, 37]), and that an overwhelming majority of cells (called *strong* cells) can be refreshed much less frequently than the standard-specified rate. To take advantage of typical-case DRAM cell data retention behavior, we use CROW to implement CROW-ref, which remaps weak *regular* DRAM rows to strong *copy* rows. **The key idea of CROW-ref** is to avoid storing any data in DRAM rows that contain weak cells, such that the *entire* DRAM chip can use a longer refresh interval. During system boot or runtime, CROW-ref uses an efficient profiling mechanism [87] to identify weak DRAM cells, remaps the *regular* rows containing weak cells to strong *copy* rows, and records the remapping information in the CROW-table. When the memory controller needs to activate a row, it first checks the CROW-table to see if the row is remapped, and, if so, activates the corresponding copy row instead of the regular row. Because only a very small fraction of DRAM cells are weak, remapping only a few rows can significantly reduce the refresh overhead, as the weakest DRAM cell in use determines the minimum required refresh rate of the entire DRAM chip. CROW-ref has two important properties that prior works [2, 3, 11, 12, 19, 33, 38, 49, 50, 64, 66, 69, 76, 83, 84, 86, 88, 107] fail to address at the same time. First, to avoid using weak rows, CROW-ref does *not* require system software changes that complicate data allocation. Second, once our versatile CROW substrate is implemented, CROW-ref does not require any additional changes in DRAM that are specific to enabling a new refresh scheme.

Results Overview. Our evaluations show that CROW-cache and CROW-ref significantly improve performance and energy efficiency. First, our access latency reduction mechanism, CROW-cache, provides 7.4% higher performance and consumes 13.3% less DRAM energy, averaged across our 20 memory-intensive four-core workloads, while reserving *only* 1.6% of the DRAM storage capacity for copy rows on a system with four LPDDR4 channels. Second, our refresh rate reduction mechanism, CROW-ref, improves system performance by 8.1% and reduces DRAM energy by 5.4%, averaged across the same four-core workloads using the same system configuration with futuristic 64 Gbit DRAM chips. CROW-cache and CROW-ref are complementary to each other. When combined without increasing the number of copy rows available, the two mechanisms together improve average system performance by 20.0% and reduce DRAM energy consumption by 22.3%, achieving greater improvements than either mechanism does alone.

To analyze the activation and restoration latency impact of simultaneously activating multiple rows, we develop a detailed circuit-level DRAM model. We release this circuit-level DRAM model and our simulation infrastructure that we use to evaluate the performance and energy saving benefits of CROW as publicly available tools [96]. We hope that this will encourage researchers to develop other novel mechanisms using the CROW substrate.

This paper makes the following **key contributions:**

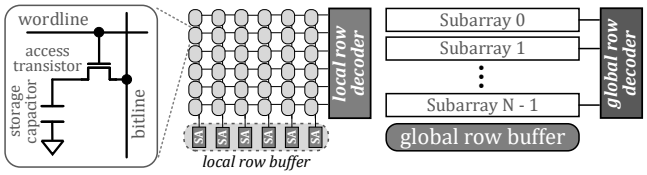
- We propose CROW, a flexible and low-cost substrate in commodity DRAM that enables mechanisms for improving DRAM performance, energy efficiency, and reliability by providing two sets of rows that have independent control in each subarray. CROW does not change the extremely-dense cell array and has *low* cost (0.48% additional area overhead in the DRAM chip, 11.3 KiB storage overhead in the memory controller, and 1.6% DRAM storage capacity overhead).
- We propose three mechanisms that exploit the CROW substrate: 1) CROW-cache, an in-DRAM cache to reduce the DRAM access latency, 2) CROW-ref, a remapping scheme for weak DRAM rows to reduce the DRAM refresh overhead, and 3) a mechanism for mitigating the RowHammer vulnerability. We show that CROW allows these mechanisms to be employed at the same time.
- We evaluate the performance, energy savings, and overhead of CROW-cache and CROW-ref, showing significant performance and energy benefits over a state-of-the-art commodity DRAM chip. We also compare CROW-cache to two prior proposals to reduce DRAM latency [53, 58] and show that CROW-cache is more area- and energy-efficient at reducing DRAM latency.

2 BACKGROUND

We describe the low-level organization of DRAM and illustrate how DRAM is accessed to provide the necessary background for understanding how CROW operates. We refer the reader to prior work [6–9, 26, 32, 53, 57–61, 64, 65, 100–102, 119] for a more detailed treatment of various aspects of DRAM operation.

2.1 DRAM Organization

A DRAM device is organized hierarchically. The smallest building block, i.e., the DRAM cell, consists of a *capacitor* and an *access transistor*, as shown in Figure 1a. The capacitor encodes a single bit



(a) DRAM cell (b) DRAM subarray (c) DRAM bank

Figure 1: DRAM organization.

of data by storing different charge levels, i.e., empty or full. During an access to a cell, the *wordline* enables the access transistor, which connects the cell capacitor to the *bitline*, so that the cell's state can be read or modified.

A *subarray* consists of a 2D array of DRAM cells, as shown in Figure 1b. The cells that share the same wordline in a subarray are referred to as a DRAM *row*. Each subarray is typically composed of 512–1024 rows. To prepare a row for an access, the *local row decoder* selects the row by enabling the corresponding wordline in the subarray. In conventional DRAM, only one row in a subarray can be enabled at a time, and, thus, only one cell is enabled on each bitline. When a cell is enabled, *charge sharing* takes place between the cell and the bitline (which is initially *precharged* to a reference voltage). This charge sharing shifts the bitline voltage up or down based on the voltage of the cell. The bitline's *sense amplifier* detects the bitline voltage shift and amplifies it to a full 0 or 1 (i.e., V_{ss} or V_{dd}). All sense amplifiers that detect and amplify the charge levels of the cells of the open row inside a subarray are referred to as the subarray's *local row buffer*.

As illustrated in Figure 1c, multiple subarrays are organized into a group to form a *bank*. To select a particular row, the memory controller sends the row address to the *global row decoder*, which partially decodes the address to select the local row decoder of the subarray that contains the row. Each bank includes a *global row buffer*. On a read operation, the requested portion of the target row is copied from the local row buffer of the subarray that contains the target row into the global row buffer. Once this is done, the global row buffer sends its data to the memory controller.

A single DRAM *chip* contains multiple banks that operate in parallel. To provide high bandwidth, multiple chips are grouped together into a *rank*. The chips in a rank share the same address/command bus, which causes the chips to operate in lockstep (i.e., the chips receive the same commands and perform the same operations on different portions), but each chip has its own data bus connection. To increase the system's total memory capacity, multiple ranks are connected to the same DRAM *channel*, operating in a time-multiplexed manner by sharing the channel bus (i.e., address/command/data pins). In a typical system, each memory controller interfaces with a single DRAM channel and sends addresses/data/commands over the channel bus to manipulate the data stored within all ranks that are part of the channel.

2.2 DRAM Operation

There are four major commands that are used to access DRAM: ACT, WR, RD, and PRE. DRAM command scheduling is tightly regulated by a set of *timing parameters*, which guarantee that enough time has passed after a certain command such that DRAM provides or retains data correctly. Figure 2 illustrates the relationship between the commands issued to perform a DRAM read and their governing timing parameters. The memory controller enforces these timing parameters as it schedules each DRAM command. Aside from the DRAM access commands, the memory controller also periodically issues a REF command to prevent data loss due to leakage of charge from the cell capacitors over time.

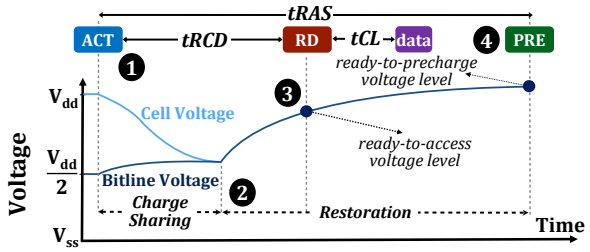


Figure 2: Commands, timing parameters, and cell/bitline voltages during a DRAM read operation.

Activate (ACT). The ACT command *activates* (opens) a DRAM row by transferring the data contained in the cell capacitors to the row buffer. ACT latency is governed by the tRCD timing parameter, which ensures that enough time has passed since the ACT is issued for the data to stabilize in the row buffer (such that it can be read).

ACT consists of two major steps: 1) capacitor-bitline *charge sharing* and 2) *charge restoration*. Charge sharing begins by enabling the wordline (❶ in Figure 2), which allows the cell capacitor to share charge with the bitline, and thus perturb the precharged bitline voltage. Once the cell and bitline voltages equalize due to charge sharing, charge restoration starts (❷). During charge restoration, the sense amplifiers are enabled to first detect the bitline voltage shift, and later restore the bitline to a full V_{ss} or V_{dd} depending on the direction of the shift. Once the bitline is restored to a *ready-to-access* voltage level (❸), restoration is complete, and the other DRAM commands (e.g., RD, WR) can be issued to the bank.

Read (RD). After a row activation, the memory controller reads data from the open row by issuing an RD command. The RD includes a column address, which indicates the portion of the open row to be read. When a DRAM chip receives an RD command, it first loads the requested portion of the open row into the global row buffer. After the data is in the global row buffer, the DRAM chip sends the data across the data bus to the memory controller. The RD command is governed by the timing parameter tCL, after which the data appears on the data bus.

Write (WR). The WR command (not shown in Figure 2) modifies data in an open DRAM row. The operation of WR is analogous to ACT in that both commands require waiting enough time for the sense amplifiers to restore the data in the DRAM cells. Similar to how a sense amplifier restores a cell capacitor during the second step of ACT, in case of a WR, the sense amplifier restores the capacitor with the new data value that the WR command provides. The restoration latency for WR is governed by the tWR timing parameter. For both ACT and WR commands, the restoration latency originates from the sense amplifier driving a bitline to replenish the charge of the DRAM cell capacitor [40, 53, 57, 120]. This makes any optimizations to the charge restoration step of ACT equally applicable to WR. Thus, using such optimizations we can decrease both tRAS and tWR.

Precharge (PRE). PRE is used to *close* an open DRAM row and prepare the DRAM bank for activation of another row. The memory controller can follow an ACT with PRE to the same bank after at least the time interval specified by tRAS. tRAS ensures that enough time has passed to fully restore the DRAM cells of the activated row to a *ready-to-precharge voltage* (❹ in Figure 2). The latency of PRE is governed by the timing parameter tRP, which allows enough time to set the bitline voltage back to reference level (e.g., $V_{dd}/2$). After tRP, the memory controller can issue an ACT to open a new row in the same bank.

Refresh (REF). A DRAM cell cannot store its data permanently, as the cell capacitor leaks charge over time. The *retention time* of a DRAM cell is defined as the length of time for which the data can

still be correctly read out of the cell after data is stored in the cell. To ensure data integrity, the memory controller periodically issues REF commands to the DRAM chip. Each REF command replenishes the charge of several DRAM rows starting from the one that a *refresh counter* (implemented in the DRAM chip) points to. The memory controller is responsible for issuing a sufficient number of REF commands to refresh the entire DRAM during a manufacturer-specified refresh interval (typically 64 ms in DDR3 and DDR4 DRAM, and 32 ms in LPDDR4).

3 COPY-ROW DRAM

To efficiently solve the challenges of 1) high access latency, 2) high refresh overhead, and 3) increasing vulnerability to failure mechanisms in DRAM, without requiring any changes to the DRAM cell array, we introduce *Copy-Row DRAM* (CROW). CROW is a new, practical substrate that exploits the existing DRAM architecture to efficiently duplicate a small number of rows in a subarray at runtime. CROW is versatile, and enables new mechanisms for improving DRAM performance, energy efficiency, and reliability, as we show in Section 4.

3.1 CROW: A High-Level Overview

CROW consists of two key components: 1) a small number of *copy rows* in each subarray, which can be used to duplicate or remap the data stored in one of the remaining rows (called *regular rows*) of the subarray, 2) a table in the memory controller, called the *CROW-table*, that tracks which rows are duplicated or remapped.

CROW divides a DRAM subarray into two types of rows: regular rows and copy rows. A copy row is similar to a regular row in the sense that it has the same row width and DRAM cell structure. However, copy rows have their own small local row decoder within the subarray, separate from the existing local row decoder for regular rows. This enables the memory controller to activate copy rows independently from regular rows (which continue to use the existing local row decoder). By allowing copy rows and regular rows to be activated independently, we enable two DRAM primitives that make use of *multiple-row activation* (MRA) in CROW.

First, CROW can perform *bulk data movement*, where the DRAM copies an entire row of data at once from a regular row to a copy row. To do so, the memory controller first activates the regular row, and next activates the copy row, immediately after the local row buffer latches the data of the regular row. After the second activation, the sense amplifiers restore the data initially stored only in the regular row to both the regular row and the copy row (similar to the RowClone [100] mechanism).

Second, CROW can perform *reduced-latency DRAM access*. When a regular row and a copy row contain *the same data*, the memory controller activates both rows *simultaneously*. This causes two cells on the same bitline to inject charge into the bitline at a faster total rate than a single cell can. Thereby, the sense amplifier to operate faster, which leads to lower activation latency via an effect similar to increasing the amount of charge stored in a single cell [10].

CROW makes use of a table in the memory controller, CROW-table, that tracks which regular rows are duplicated or mapped to copy rows. A mechanism that takes advantage of the CROW substrate checks the CROW-table and uses the information in the table to either 1) simultaneously activate duplicate regular and copy rows or 2) activate the copy row that a regular row is remapped to. For example, the CROW-cache mechanism (Section 4.1) updates a CROW-table entry with the address of the regular row that has been copied to the corresponding copy row. Prior to issuing an ACT command to activate a target regular row, the memory controller queries the CROW-table to check whether the regular row has a duplicate copy row. If so, the memory controller issues a custom reduced-latency DRAM command to simultaneously activate both the regular row and the copy row, instead of a single activate to the regular row. Doing so enables faster access to the duplicated data stored in both rows. Next, we explain CROW in detail.

3.2 Copy Rows

CROW logically categorizes the rows in a subarray into two sets, as shown in Figure 3: (1) *regular rows*, which operate the same as in conventional DRAM; and (2) *copy rows*, which can be activated independently of the regular rows. In our design, we add a small number of copy rows to the subarray, and add a second local row decoder (called the *CROW decoder*) for the copy rows.[1] Because the CROW decoder drives a much smaller number of rows than the existing local row decoder, it has a much smaller area cost (Section 6.2). The memory controller provides the regular row and copy row addresses to the respective decoders when issuing an ACT command.

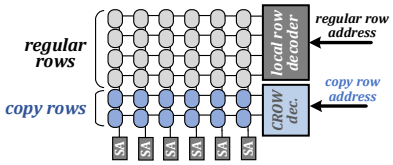


Figure 3: Regular and copy rows in a subarray in CROW.

3.3 CROW-table

As shown in Figure 4, the CROW-table in the memory controller stores information on 1) whether a copy row is allocated, and 2) which regular row is duplicated or remapped to a copy row. For example, in our weak row remapping scheme (Section 4.2), a CROW-table entry holds the address of the regular row that the copy row replaces. The CROW-table stores an entry for each copy row in a DRAM channel, and is *n*-way set associative, where *n* is the number of copy rows in each subarray.

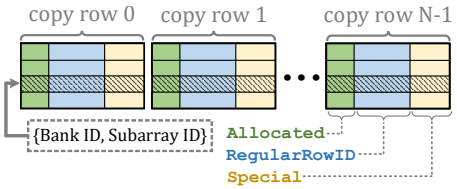


Figure 4: Organization of the CROW-table.

The memory controller indexes the CROW-table using a combination of bank and subarray addresses, which are part of the physical address of a memory request. The memory controller checks the table for an entry associated with the regular row that the memory request is to access. Our current design stores three fields in each CROW-table entry:[2] 1) the Allocated field indicates whether the entry is valid or not; 2) the RegularRowID field stores a pointer to the regular row that the corresponding copy row is associated with 3) the Special field stores additional information specific to the mechanism that is implemented using CROW.

[1] Alternatively, CROW can use a small set of the existing rows in a conventional subarray as copy rows, but this requires changing the existing local row decoder, and may not keep the number of addressable rows in the subarray as a power of two.

[2] The entry can be expanded to contain more information if needed by other mechanisms that make use of CROW.

4 APPLICATIONS OF CROW

CROW is a versatile substrate that enables multiple mechanisms for improving DRAM performance, energy efficiency, and reliability. We propose three such new mechanisms based on CROW: 1) CROW-cache, a mechanism that reduces DRAM access latency by exploiting CROW's ability to simultaneously activate a regular row and a copy row that store the same data; 2) CROW-ref, a mechanism that reduces DRAM refresh overhead by exploiting CROW's ability to remap weak rows with low retention times to strong copy rows. This mechanism relies on retention time profiling [65, 87, 88] to determine weak and strong rows in DRAM; 3) a mechanism that protects against the RowHammer [52] vulnerability by identifying rows that are vulnerable to RowHammer-induced errors and remapping the vulnerable rows to copy rows.

4.1 In-DRAM Caching (CROW-cache)

Our in-DRAM caching mechanism, CROW-cache, exploits both of our new multiple-row activation primitives in CROW (Section 3.1). The key idea of CROW-cache is to use a copy row as a duplicate of a recently-accessed regular row within the same subarray, and to activate *both* the regular row and the copy row. By simultaneously activating both rows, CROW-cache reduces activation latency and starts servicing read and write requests sooner.

4.1.1 Copying a Regular Row to a Copy Row. Given N copy rows, we would like to duplicate the N regular rows that will be most frequently activated in the near future to maximize the benefits of CROW-cache. However, determining the N most frequently-activated rows is a difficult problem, as applications often access main memory with irregular access patterns. Therefore, we follow a similar approach used by processor caches, where the most-recently-used rows are cached instead of the most frequently-accessed rows. Depending on how many copy rows are available, CROW-cache maintains copies of the N *most-recently-activated* regular rows in each subarray. When the memory controller needs to activate a regular row that is not already duplicated, the memory controller copies the regular row to an available copy row, using multiple-row activation. While this is a simple scheme, we achieve a significant hit rate on the CROW-table with a small number of copy rows (see Section 8.1.1).

To efficiently copy a regular row to a copy row within a subarray, we introduce a new DRAM command, which we call *Activate-and-copy* (`ACT-c`). The `ACT-c` command performs a three-step row copy operation in DRAM, using a technique similar to RowClone [100]. First, the memory controller issues `ACT-c` to DRAM and sends the addresses of 1) the *source* regular row and 2) the *destination* copy row, over multiple command bus cycles. Second, upon receiving an `ACT-c` command, DRAM enables the wordline of only the regular row. The process of reading and latching the row's data into the sense amplifiers completes as usual (see Section 2.2). Third, as soon as the data is latched in the sense amplifiers, DRAM enables the wordline of the copy row. This causes the sense amplifiers to restore charge to both the regular row *and* the copy row. On completion of the restoration phase, both the regular row and the copy row contain the same data.

The `ACT-c` operation slightly increases the restoration time compared to regular `ACT`. This is because, after the wordline of the copy row is enabled, the capacitance that the sense amplifier drives the bitline against increases, since two cell capacitors are connected to the same bitline. In our circuit-level SPICE model, for the `ACT-c` command, we find that the activate-to-precharge latency (`tRAS`) increases by 18%. However, this activation latency overhead has a very limited impact on overall system performance because CROW-table typically achieves a high hit rate (Section 8.1.1), which means that the latency of duplicating the row is amortized by the reduced activation latency of future accesses to the row.

4.1.2 Reducing Activation Latency. CROW-cache introduces a second new DRAM command, *Activate-two* (`ACT-t`), to simultaneously activate a regular row and its duplicate copy row. If the CROW-table (see Section 3.3) contains an entry indicating that a copy row is a duplicate of the regular row to be activated, the memory controller issues `ACT-t` to perform MRA on both rows, achieving low-latency activation. In Section 5, we perform detailed circuit-level SPICE simulations to analyze the activation latency reduction with `ACT-t`.

Note that the modifications needed for `ACT-c` and `ACT-t` in the row decoder logic are nearly identical, as they both perform multiple-row activation. The difference is that rather than activating the copy row*after* the regular row as with `ACT-c`, `ACT-t` activates both of the rows concurrently.

4.1.3 Reducing Restoration Latency. In addition to decreasing the amount of time required for charge sharing during activation (see Section 2.2), the increased capacitance on the bitline due to the activation of multiple cells enables the reduction of `tRAS` by using *partial restoration* [116, 121]. The main idea of the partial restoration technique is to terminate row activation earlier than normal, i.e., issue `PRE` by relaxing (i.e., reducing) `tRAS`, on a multiple-row activation such that the cells are not fully restored. While this degrades the retention time of each cell individually, the partially restored cells contain just enough charge to maintain data integrity until the next refresh operation [93] (we verify this using SPICE simulations in Section 5). We combine partial cell restoration (i.e., reduction in `tRAS`) with the reduction in `tRCD` in order to provide an even greater speedup than possible by simply decreasing `tRCD` alone (see Section 8).

Taking the concept one step further, we can also apply partial restoration to write accesses, decreasing the amount of time required for the `WR` command. Since the write restoration latency (`tWR`) is analogous to cell restoration during activation (Section 2.2), we can terminate the write process early enough such that the written cells are only partially charged.

4.1.4 Evicting an Entry from the CROW-table. As described in Section 4.1.3, CROW-cache relaxes `tRAS` for the `ACT-t` command to further improve the DRAM access latency. Relaxing `tRAS` may terminate the restoration operation early, which puts the precharged regular and copy rows into a *partially-restored* state.

Note that there are two cases where relaxing `tRAS` does not necessarily lead to partial restoration. First, the memory controller can exploit an open regular row + copy row pair to serve multiple memory requests, if available in the request queue, from different portions of the row. The time needed to serve these requests can delay precharging the associated regular and copy rows until they are fully restored. Second, if there are no other memory requests to different rows in the same bank, the currently-opened regular row and copy row can be left open, providing enough time for full charge restoration.

CROW-cache maintains the restoration state of the paired regular and copy rows by utilizing a single bit of the `Special` field of the CROW-table, which in the context of CROW-cache we refer to as the `isFullyRestored` field. CROW-cache sets `isFullyRestored` to `false` only if 1) `ACT-t` was used to activate the currently open row, and 2) the memory controller issues a `PRE` command to close the rows *before* the default `tRAS` is satisfied. In contrast, CROW-cache sets `isFullyRestored` to `true` if the memory controller issues the next `PRE` after the default `tRAS` (i.e., a time interval sufficient for fully restoring the open regular row and copy row). Note

that existing memory controllers already maintain timing information for conventional command scheduling, e.g., when the last ACT to each bank was issued. By taking advantage of the existing timing information, CROW-cache does not incur significant area overhead for managing the isFullyRestored field.

Partially restoring the duplicate rows helps to significantly reduce restoration latency (see Section 5). However, partial restoration introduces a new challenge to the design of CROW-cache, because a partially-restored row can only be correctly accessed using ACT-t, which activates a regular row along with its duplicate copy row. Duplicating a new regular row (RR_{new}) to a copy row that is already a duplicate of another regular row (RR_{old}), i.e., evicting RR_{old} from the CROW-table, causes RR_{old} to be activated as a single row during a future access. If RR_{old} is partially restored prior to eviction from the CROW-table, performing a single-row activation on RR_{old} in the future may lead to data corruption. Thus, the memory controller needs to guarantee that a partially-restored regular row is never evicted from the CROW-table.

To prevent data corruption due to the eviction of a partially-restored row from the CROW-table, if the *isFullyRestored* field is *false*, the memory controller first issues an ACT-t and fully restores RR_{old} by conforming to the *default* tRAS. Since this operation sets *isFullyRestored* to *true*, RR_{old} can now safely be evicted from the CROW-table to be replaced with RR_{new}. The disadvantage of this approach is the overhead of issuing an additional ACT-t followed by a PRE to perform full restoration. However, in Section 8.1.1, we show that this overhead has a negligible performance impact since CROW-table has a high hit rate.

4.1.5 Implementing the New DRAM Commands.

To implement CROW-cache, we introduce the ACT-c and ACT-t DRAM commands, both of which activate a regular row *and* a copy row. The wordline of a regular row is enabled in the same way as in conventional DRAM. To enable the wordline of a copy row, we extend the existing local row decoder logic such that it can drive the wordline of a copy row independently of the regular row. As we show in Section 6.2, for the default configuration of the CROW substrate with eight copy rows, our modifications increase the decoder area by 4.76%, leading to 0.48% area overhead in the entire DRAM chip. The additional eight copy rows per subarray require only 1.6% of the DRAM storage capacity.

Unlike ACT, which specifies a single row address, ACT-c and ACT-t need to specify the addresses of both a regular and a copy row. ACT-c and ACT-t need only a small number of copy row address bits in addition to the regular row address bits since the corresponding copy row is in the same subarray as the activated regular row. For CROW-8, where each subarray has eight copy rows, we need only three additional bits to encode the copy row address. We can either add three wires to the address bus (likely undesirable) or send the address over multiple cycles as done in current LPDDR4 chips for the ACT, RD, and WR commands [36].[3]

4.2 Reducing Refresh Overhead (CROW-ref)

To reduce the DRAM refresh overhead, we take advantage of the CROW substrate to develop CROW-ref, a software-transparent weak row remapping scheme. CROW-ref extends the refresh interval of the *entire* DRAM chip beyond the worst-case value defined in DRAM standards (64 ms for DDR3 and DDR4, 32 ms for LPDDR4) by avoiding the use of the very small set of weak rows in a given DRAM chip, i.e., rows that would fail when the refresh interval is extended (due to the existence of at least one weak cell in the row). CROW-ref consists of three key components. First, CROW-ref uses retention time profiling at system boot or during runtime to identify the weak rows. Second, CROW-ref utilizes the strong copy rows provided by CROW in each subarray to store the data that would have originally been stored in weak regular rows. Third, CROW-ref uses the CROW-table to maintain the remapping information, i.e., which strong copy row replaces which weak regular row.

4.2.1 Identifying Weak Rows.

To identify the weak rows in each subarray, we rely on retention time profiling methodologies proposed by prior work [41–43, 52, 65, 87, 88, 92, 118]. A retention time profiler tests the DRAM device with various data patterns and a wide range of operating temperatures to cover all DRAM cells that fail at a chosen target retention time. Prior works find that very few DRAM cells fail when the refresh interval is extended by 2x-4x. For example, Liu et al. [65] show that *only* ~1000 cells in a 32 GiB DRAM module fail when the refresh interval is extended to 256 ms. Assuming the experimentally-demonstrated uniform random distribution of these weak cells in a DRAM chip [2, 64, 65, 87, 88], we calculate that the *bit error rate (BER)* is $4 \cdot 10^{-9}$ when operating at a 256 ms refresh interval.

Based on this observation, we can determine how effective CROW-ref is likely to be for a given number of copy rows. First, using the BER, we can calculate P_{weak_row}, the probability that a row contains at least one weak cell, as follows:

$$P_{weak\_row} = 1 - (1 - BER)^{N_{cells\_per\_row}} \quad (1)$$

where $N_{cells\_per\_row}$ is the number of DRAM cells in a row. Second, we can use P_{row} to determine $P_{subarray-n}$, which is the probability that a subarray with N_{rows} rows contains more than n weak rows:

$$P_{subarray-n} = 1 - \sum_{k=0}^{n} \binom{N_{rows}}{k} P_{row}^{k} (1 - P_{row})^{N_{rows}-k} \quad (2)$$

Using these equations, for a DRAM chip with 8 banks, 128 subarrays per bank, 512 rows per subarray, and 8 KiB per row, the probability of any subarray having more than 1/2/4/8 weak rows is $0.99/3.1 \times 10^{-1}/3.3 \times 10^{-4}/3.3 \times 10^{-11}$. We conclude that the probability of having a subarray with a large number of weak rows is extremely low, and thus CROW-ref is highly effective even when the CROW substrate provides only 8 copy rows per subarray. In the unlikely case where the DRAM has a subarray with more rows with weak cells than the number of available copy rows, CROW-ref falls back to the default refresh interval, which does not provide performance and energy efficiency benefits but ensures correct DRAM operation.[4]

4.2.2 Operation of the Remapping Scheme.

In each subarray, CROW-ref remaps weak regular rows to *strong* copy rows in the same subarray.[5] CROW-ref tracks the remapping using the CROW-table. When a weak regular row is remapped to a strong copy row, CROW-ref stores the row address of the regular row into the RegularRowID field of the CROW-table entry that corresponds to the copy row. When the memory controller needs to activate a row, it checks the CROW-table to see if any of the RegularRowID fields contains the address of the row to be activated. If one of the RegularRowID fields matches, the memory controller issues an activate command to the copy row that replaces the regular row, instead of to the regular row. On a CROW-table miss, which

[3] In our evaluations, we assume an additional cycle on the command/address bus to send the copy row address to DRAM. This additional cycle does not always impact the activation latency, as the memory controller can issue the ACT-c and ACT-t commands one cycle earlier if the command/address bus is idle. Also, DRAM does not immediately need the address of the copy row for ACT-c.

[4] Alternatively, CROW-ref can be combined with a heterogeneous refresh-rate scheme, e.g., RAIDR [64] or AVATAR [88].

[5] We do *not* use a weak copy row to replace a weak regular row, as a weak copy row would also not maintain data correctly for an extended refresh interval. The retention time profiler also identifies the weak copy rows.

indicates that the row to be activated contains only strong cells, the memory controller activates only the original regular row. This row remapping operates transparently from the software, as the regular-to-copy-row remapping is *not* exposed to the software.

4.2.3 Support for Dynamic Remapping. CROW-ref can dynamically change the remapping of weak rows if new weak rows are detected at runtime. This functionality is important as DRAM cells are known to be susceptible to a failure mechanism known as *variable retention time* (VRT) [42, 43, 65, 74, 87, 88, 92, 118]. As a VRT cell can nondeterministically transition between high and low retention states, new VRT cells need to be continuously identified by periodically profiling DRAM, as proposed by prior work [41, 87, 88]. To remap a newly-identified weak regular row at runtime, the memory controller simply allocates an unused strong copy row from the subarray, and issues an ACT-c to copy the regular row's data into the copy row. Once this is done, the memory controller accesses the strong copy row instead of the weak regular row, as we explain in Section 4.2.2.

4.3 Mitigating RowHammer

We propose a third mechanism enabled by the CROW substrate that protects against the RowHammer vulnerability [52, 79, 80]. Due to the small DRAM cell size and short distance between DRAM cells, electromagnetic coupling effects that result from rapidly activating and precharging (i.e., hammering) a DRAM row cause bit flips on the physically-adjacent (i.e., victim) rows [52, 79, 80]. This effect is known as *RowHammer*, and has been demonstrated on real DRAM chips [52]. Aside from the decreased reliability of DRAM due to RowHammer, prior works (e.g., [20, 21, 63, 90, 99, 110, 113, 117]) exploit RowHammer to perform attacks that gain privileged access to the system. Therefore, it is important to mitigate the effects of RowHammer in commodity DRAM.

We propose a CROW-based mechanism that mitigates RowHammer by remapping the victim rows adjacent to the hammered row to copy rows. By doing so, the mechanism prevents the attacker from flipping bits on the data originally allocated in the victim rows. Several prior works [16, 45, 62, 103] propose techniques for detecting access patterns that rapidly activate and precharge a specific row. Typically, these works implement a counter-based structure that stores the number of ACT commands issued to each row. Our RowHammer mitigation mechanism can use a similar technique to detect a RowHammer attack. When the memory controller detects a DRAM row that is being hammered, it issues two ACT-c commands to copy the victim rows that are adjacent to the hammered row to two of the available copy rows in the same subarray. Similar to the CROW-ref mechanism (Section 4.2), our RowHammer mitigation mechanism tracks a remapped victim row using the RegularRowID field in each CROW-table entry, and looks up the CROW-table every time a row is activated to determine if a victim row has been remapped to a copy row.

CROW enables a simple yet effective mechanism for mitigating the RowHammer vulnerability that can reuse much of the logic of CROW-ref. We leave the evaluation of our RowHammer mitigation mechanism to future work.

5 CIRCUIT SIMULATIONS

We perform detailed circuit-level SPICE simulations to find the latency of 1) simultaneously activating two DRAM rows that store the same data (ACT-t) and 2) copying an entire regular row to a copy row inside a subarray (ACT-c). Table 1 shows a summary of change in the tRCD, tRAS, and tWR timing parameters that we use for the two new commands, based on the SPICE simulations. We discuss in detail how the timings are derived for ACT-t in Section 5.1, and for ACT-c in Section 5.2.

Table 1: Timing parameters for new DRAM commands.

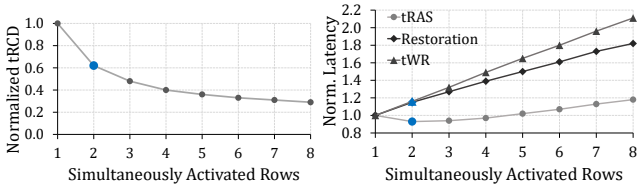
DRAM Command		tRCD	tRAS	tWR
ACT-t	*activating fully-restored rows*	-38%	-7%[a] (-33%[b])	+14%[a] (-13%[b])
	activating partially-restored rows	-21%	-7%[a] (-25%[b])	+14%[a] (-13%[b])
ACT-c		0%	+18%[a] (-7%[b])	+14%[a] (-13%[b])

[a] When fully restoring the charge. [b] When terminating charge restoration early.

In our simulations, we model the entire cell array of a modern DRAM chip using 22 nm DRAM technology parameters, which we obtain by scaling the reference 55 nm technology parameters [89] based on the ITRS roadmap [34, 115]. We use 22 nm PTM low-power transistor models [1, 122] to implement the access transistors and the sense amplifiers. In our SPICE simulations, we run 10^4 iterations of Monte-Carlo simulations with a 5% margin on every parameter of each circuit component, to account for manufacturing process variation. Across all iterations, we observe that the ACT-t and ACT-c commands operate correctly. We report the latency of the ACT-t and ACT-c commands based on the Monte-Carlo simulation iteration that has the highest access latency for each of these commands. We release our SPICE model as an open-source tool [96].

5.1 Simultaneous Row Activation Latency

Simultaneously activating multiple rows that store the same data accelerates the *charge-sharing* process, as the increased amount of charge driven on each bitline perturbs the bitline faster compared to single-row activation. Figure 5a plots the reduction in activation latency (tRCD) for a varying number of simultaneously-activated rows. As seen in the figure, we observe a tRCD reduction of 38% when simultaneously activating *two* rows. tRCD reduces further when we increase the number of simultaneously-activated rows, but the latency reduction per additional activated row becomes smaller. We empirically find that simultaneously activating only two rows rather than more rows achieves a large portion of the maximum possible tRCD reduction potential (i.e., as we approach an infinite number of rows being activated simultaneously) with low area and power overhead (see Section 6.2).



(a) tRCD (18 ns) **(b) tRAS (42 ns), tWR (18 ns), and restoration (24 ns)**

Figure 5: Change in various DRAM latencies with different number of simultaneously-activated rows, normalized to baseline DRAM timing parameters (absolute baseline values in parentheses).

Change in Restoration Latency. Although two-row activation reduces tRCD, *fully* restoring the capacitors of two cells takes more time compared to fully restoring a single cell. Therefore, two-row activation can potentially increase tRAS and tWR, which could offset the benefits of the tRCD reduction. Figure 5b shows the change in tRAS, restoration time, and tWR for a varying number of simultaneously-activated rows. Although restoration time always increases with the number of rows, we see a slight decrease in tRAS when the number of rows is small. This is because the reduction in tRCD, which is part of tRAS (as we explain in Section 2.2), is larger than the increase in restoration time. However, for five or more

simultaneously-activated rows, the overhead in restoration time exceeds the benefits of tRCD reduction, and, as a result, tRAS increases. tWRalways increases with the number of simultaneously-activated rows, because writing to a DRAM cell is similar to restoring a cell (Section 2.2).

Terminating Restoration Early. We use CROW to enable in-DRAM caching by duplicating recently-accessed rows and using two-row activation to reduce tRCD significantly for future activation of such rows. However, to make the in-DRAM caching mechanism more effective, we aim to reduce tRAS further, as it is a critical timing parameter that affects how quickly we can switch between rows in the event of a row buffer conflict. We make *three* observations based on two-row activation, which leads us to a trade-off between reducing tRCD and reducing tRAS. First, when two DRAM rows are used to store the same data, the data is correctly retained for a longer period of time compared to when the data is stored in a single row. This is due to the increased aggregate capacitance that storing each bit of data using two cells provides. Second, since data is correctly retained for a longer period of time when stored in two rows, we can terminate the restoration operation early to reduce tRAS and still achieve the target retention time, e.g., 64 ms. Third, as the amount of charge stored in a cell capacitor decreases, the activation latency increases [26, 57]. Therefore, terminating restoration early reduces tRAS at the expense of a slight increase in tRCD (due to less charge).

We explore the trade-off space between reducing tRCD and reducing tRAS for a varying number of simultaneously-activated rows using our SPICE model. In Figure 6, we show the different tRCD and tRAS latencies that can be used with multiple-row activation (MRA) while still ensuring data correctness. For two-row activation, we empirically find that a 21% reduction in tRCD and a 33% reduction in tRAS provides the best performance on average for the workloads that we evaluate (Section 8.1.1).

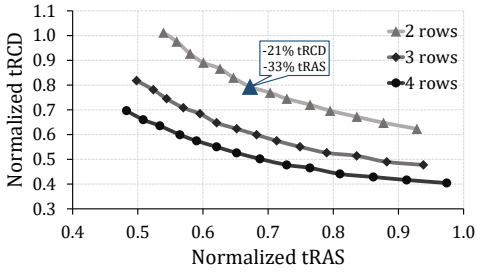


Figure 6: Normalized tRCD latency as a function of normalized tRAS latency for different number of simultaneously activated DRAM rows.

We also eliminate the tWR overhead of MRA by terminating the restoration step early when writing to simultaneously-activated DRAM rows. Since reducing tWR increases the tRCD latency for the next activation of the row, there exists a trade-off between reducing tRCD and reducing tWR, similar to the trade-off between reducing tRCD and reducing tRAS. In our evaluations, we find that to achieve a 21% reduction in tRCD, we can reduce tWR by 13% (see Table 1) by terminating the restoration step early during a write to simultaneously-activated DRAM rows.

5.2 MRA-based Row Copy Latency

As we explain in Section 4.1.1, CROW-cache uses the new ACT-c command to efficiently copy a regular row to a copy row in the same subarray by activating the copy row slightly after activating the regular row. This gives the sense amplifiers sufficient time to correctly read the data stored in the regular row. Using our SPICE model, we find that ACT-c does not affect tRCD because the copy row is activated after satisfying tRCD. This is because, to start restoring the data of the regular row to both the regular row and the copy row, the local row buffer first needs to correctly latch the data of the regular row. In contrast to tRCD, the ACT-c command increases tRAS by 18% (reduces tRAS by 7% when terminating restoration early), as restoring two DRAM rows requires more time than restoring a single row. We model this additional time in all of our evaluations.

6 HARDWARE OVERHEAD

Implementing the CROW substrate requires only a small number of changes in the memory controller and the DRAM die.

6.1 Memory Controller Overhead

CROW introduces the CROW-table (Section 3.3), which incurs modest storage overhead in the memory controller. The storage requirement for each entry ($Storage_{entry}$) in CROW-table can be calculated in terms of bits using the following equation:

$$Storage_{entry} = \lceil log_2(RR) \rceil + Bits_{Special} + Bits_{Allocated} \quad (3)$$

where RR is the number of regular rows in each subarray (we take the log of RR to represent the number of bits needed to store the RegularRowID field), $Bits_{Special}$ is the size of the Special field in bits, and $Bits_{Allocated}$ is set to 1 to indicate the single bit used to track whether the entry is currently valid. Note that the RegularRowID field does not have to store the entire row address. Instead, it is sufficient to store a pointer to the position of the row in the subarray. For an example subarray with 512 regular rows, an index range of 0–511 is sufficient to address all regular rows in same subarray. We assume one bit for the Special field, which we need to distinguish between the CROW-cache and CROW-ref mechanisms (Section 4).

We calculate $Storage_{CROW\text{-}table}$, the storage requirement for the entire CROW-table, as:

$$Storage_{CROW\text{-}table} = Storage_{entry} * CR * SA \quad (4)$$

where CR is the number of copy rows in each subarray, and SA is the total number of subarrays in the DRAM. We calculate the storage requirement of the CROW-table for a single-channel memory system with 512 regular rows per subarray, 1024 subarrays (8 banks; 128 subarrays per bank), and 8 copy rows per subarray to be 11.3 KiB. Thus, the processor die overhead is small.

The CROW-table storage overhead is proportional to the memory size. Although the overhead is small, we can further optimize the CROW-table implementation to reduce its storage overhead for large memories. One such optimization shares one set of CROW-table entries across multiple subarrays. While this limits the number of copy rows that can be utilized simultaneously, CROW can still capture the majority of the benefits that would be provided if the CROW substrate had a separate CROW-table for each subarray. From our evaluations, when sharing each CROW-table entry between 4 subarrays (i.e., approximately a factor of 4 reduction in CROW-table storage requirements), we observe that the average speedup that CROW-cache provides for single-core workloads reduces from 7.1% to only 6.1%.

We evaluate the access time of a CROW-table with the configuration given in Table 2 using CACTI [77], and find that the access time is only 0.14 ns. We do *not* expect the table access time to have any impact on the overall cycle time of the memory controller.

6.2 DRAM Die Area Overhead

To activate multiple rows in the same subarray, we modify the row decoder to drive multiple wordlines at the same time. These modifications incur a small area overhead and cause MRA to consume more power compared to a single-row activation.

In Figure 7 (left), we show the activation power overhead for simultaneously activating up to nine DRAM rows in the same subarray. The `ACT-c` and `ACT-t` commands simultaneously activate two rows, and consume only 5.8% additional power compared to a conventional `ACT` command that activates only a single row. The slight increase in power consumption is mainly due to the small copy row decoder that CROW introduces to drive the copy rows.

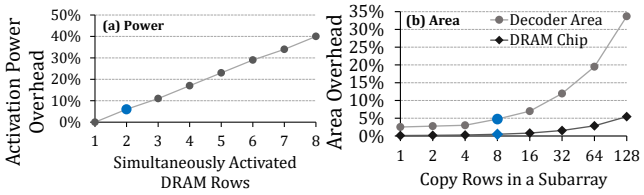


Figure 7: Power consumption and area overhead of MRA.

Figure 7 (right) plots the area overhead of a copy row decoder that enables one of the copy rows in the subarray independently from the existing regular row decoder. The figure shows the copy row decoder area overhead as we vary the number of copy rows in a subarray. For CROW-8, which has eight copy rows per subarray, the decoder area increases by 4.8%, which corresponds to only a 0.48% area overhead for the entire DRAM chip. The area overhead of CROW-8 is small because the decoder required for eight copy rows is as small as 9.6 μm^2, while our evaluations show that the local row decoder for 512 regular rows occupies 200.9 μm^2.

7 METHODOLOGY

We use Ramulator [55, 97], a cycle-accurate DRAM simulator, to evaluate the performance of the two mechanisms that we propose based on the CROW substrate. We run Ramulator in CPU-trace-driven mode and collect application traces using a custom Pin-tool [68]. The traces include virtual addresses that are accessed by the application during the trace collection run. In Ramulator, we perform virtual-to-physical address translation by randomly allocating a 4 KiB physical frame for each access to a new virtual page, which emulates the behavior of a steady-state system [85].

Table 2 provides the system configuration we evaluate. Unless stated otherwise, we implement CROW with eight copy rows per subarray. We analyze the performance of CROW-cache and CROW-ref on single- and multi-core system configurations using typical LPDDR4 timing parameters, as shown in the table. Although our evaluation is based on LPDDR4 DRAM, which is predominantly used in various low-power systems [17], the mechanisms that we propose can also reduce access latency in other DRAM-based memories, including 3D-stacked DRAM [59].

Workloads. We evaluate 44 single-core applications from four benchmark suites: SPEC CPU2006 [106], TPC [111], STREAM [71], and MediaBench [15]. In addition, we evaluate two synthetic applications [75] (excluded from our average performance calculations): 1) *random*, which accesses memory locations at random and has very limited row-level locality; and 2) *streaming*, which has high row-level locality because it accesses contiguous locations in DRAM such that the time interval between two consecutive memory requests is long enough for the memory controller to precharge the recently-open row.

Table 2: Simulated system configuration.

Processor	1-4 cores, 4 GHz clock frequency, 4-wide issue, 8 MSHRs per core, 128-entry instruction window
Last-Level Cache	64 B cache-line, 8-way associative, 8 MiB capacity
Memory Controller	64-entry read/write request queue, FR-FCFS-Cap[6] scheduling policy [81], timeout-based row buffer policy[7]
DRAM	LPDDR4 [36], 1600 MHz bus frequency, 4 channels, 1 rank, 8 banks/rank, 64K rows/bank, 512 rows/subarray, 8 KiB row buffer size, `tRCD`/`tRAS`/`tWR` 29 (18)/67 (42)/29 (18) cycles (ns)

We classify the applications into three groups based on the misses-per-kilo-instruction (MPKI) in the last-level cache. We obtain the MPKI of each application by analyzing SimPoint [25] traces (200M instructions) of each application's representative phases using the single-core configuration. The three groups are:

- L (low memory intensity): $MPKI < 1$
- M (medium memory intensity): $1 \leq MPKI < 10$
- H (high memory intensity): $MPKI \geq 10$

We create eight multi-programmed workload groups for the four-core configuration, where each group consists of 20 multi-programmed workloads. Each group has a mix of workloads of different memory intensity classes. For example, *LLHH* indicates a group of 20 four-core multi-programmed workloads, where each workload consists of two randomly-selected single-core applications with low memory intensity (L) and two randomly-selected single-core applications with high memory intensity (H). In total, we evaluate 160 multi-programmed workloads. We simulate the multi-core workloads until each core executes at least 200 million instructions. For all configurations, we initially warm up the caches by fast-forwarding 100 million instructions.

Metrics. We measure the speedup for single-core applications using the instructions per cycle (IPC) metric. For multi-core evaluations, we use the weighted speedup metric [104], which prior work shows is a good measure of job throughput [13].

We use CACTI [77] to evaluate the DRAM area and power overhead of our two mechanisms (i.e., CROW-cache and CROW-ref) and prior works (i.e., TL-DRAM [58] and SALP [53]) that we compare against. We perform a detailed evaluation of the latency impact of MRA using our circuit-level SPICE model [96]. We use DRAMPower [5] to estimate DRAM energy consumption of our workloads.

8 EVALUATION

We evaluate the performance, energy efficiency, and area overhead of the CROW-cache and CROW-ref mechanisms.

8.1 CROW-cache

We evaluate CROW-cache with different numbers of copy rows per subarray in the CROW substrate. We denote each configuration of CROW in the form of CROW-C_r, where C_r specifies the number of copy rows (e.g., we use CROW-8 to refer to CROW with eight copy rows). To show the potential of CROW-cache, we also evaluate a hypothetical configuration with a 100% CROW-table hit rate, referred to as *Ideal CROW-cache*.

[6] We use a variation of the FR-FCFS [95, 124] policy, called FR-FCFS-Cap [81], which improves fairness by enforcing an upper limit for the number of read/write requests that a row can service once activated. This policy performs better, on average, than the conventional FR-FCFS policy [95, 124], as also shown in [54, 81, 108, 109].

[7] The timeout-based row buffer management policy closes an open row after 75 ns if there are no requests in the memory controller queues to that row.

8.1.1 Single-core Performance. Figure 8 (top) shows the speedup of CROW-cache over the baseline and the CROW-table hit rate (bottom) for single-core applications. We also list the last-level cache MPKI of each application to indicate memory intensity. We make four observations from the figure.

First, CROW-cache provides significant speedups. On average, CROW-1, CROW-8, and CROW-256 improve performance by 5.5%, 7.1%, and 7.8%, respectively, over the baseline.[8] In general, the CROW-cache speedup increases with the application's memory intensity. Some memory-intensive applications (e.g., *libq*, *h264-dec*) have lower speedups than less-memory-intensive applications because they exhibit high row buffer locality, which reduces the benefits of CROW-cache. Applications with low MPKI (< 3, not plotted for brevity) achieve speedups below 5% due to limited memory activity, but *no* application experiences slow down.

Second, for most applications, CROW-1, with *only* one copy row per subarray, achieves most of the performance of configurations with more copy rows (on average, 60% of the performance improvement of Ideal CROW-cache).

Third, the CROW-table hit rate is very high for most of the applications. This is indicative of high in-DRAM locality and translates directly into performance improvement. On average, the hit rate for CROW-1/CROW-8/CROW-256 is 68.8%/85.3%/91.1%.

Fourth, the overhead of fully restoring rows evicted from the CROW-table is negligible (not plotted). For CROW-1, which has the highest overhead from these evictions, restoring evicted rows accounts for only 0.6% of the total activation operations.

8.1.2 Multi-core Performance. Figure 9 shows the weighted speedup CROW-cache achieves with different CROW substrate configurations for four-core workload groups. The bars show the average speedup for the workloads in the corresponding workload group, and the vertical lines show the maximum and minimum speedup among the workloads in the group.

We observe that *all* CROW-cache configurations achieve a higher speedup as the memory intensity of the workloads increase. On average, CROW-8 provides 7.4% speedup for the workload group with four high-intensity workloads (i.e., *HHHH*), whereas it provides only 0.4% speedup for *LLLL*.

In contrast to single-core workloads, CROW-8 provides significantly better speedup compared to CROW-1 on four-core configurations. This is because simultaneously-running workloads are more likely to generate requests that compete for the same subarray. As a result, the CROW-table cannot achieve a high hit rate with a single copy row per subarray. In most cases, CROW-8 performs close to Ideal CROW-cache with 100% hit rate, and requires only 1.6% of the DRAM storage capacity for in-DRAM caching.

[8] We notice that, for some applications (e.g., *jp2-encode*), using fewer copy rows slightly outperforms a configuration with more copy rows. We find the reason to be the changes in memory request service order due to the fact that memory controller makes different command scheduling decisions for different configurations.

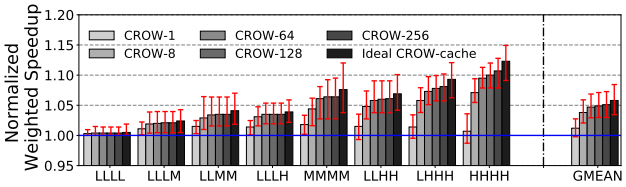


Figure 9: CROW-cache speedup (160 four-core workloads).

8.1.3 DRAM Energy Consumption. We evaluate the total DRAM energy consumed during the execution of single-core and four-core workloads. Figure 10 shows the average DRAM energy consumption with CROW-cache normalized to the baseline. Although each `ACT-c` and `ACT-t` command consumes more power than `ACT`, CROW-cache reduces the total DRAM energy due to the improvement in execution time. On average, CROW-cache decreases DRAM energy consumption by 8.2% and 6.9% on single- and four-core systems, respectively.

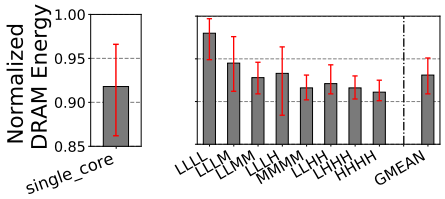


Figure 10: DRAM energy consumption with CROW-cache.

8.1.4 Comparison to TL-DRAM and SALP. We compare CROW-cache to two previous works that also enable in-DRAM caching in different ways: TL-DRAM [58] and SALP [53].

TL-DRAM uses isolation transistors on each bitline to split a DRAM subarray into a *far* and a *near* segment. When the isolation transistors are turned off, the DRAM rows in the *near segment*, which is composed of a small number of rows close to the sense amplifiers, can be accessed with low `tRCD` and low `tRAS` due to reduced parasitic capacitance and resistance on the short bitline. In contrast, accessing the far segment requires a slightly higher `tRCD` and `tRAS` than in conventional DRAM due to the additional parasitic capacitance and resistance that the isolation transistor adds to the bitline. We extend our circuit-level DRAM model to evaluate the reduction in DRAM latencies for different far and near segment sizes. We use the notation TL-DRAM-N_r, where N_r specifies the number of rows in the near segment. TL-DRAM uses the rows in

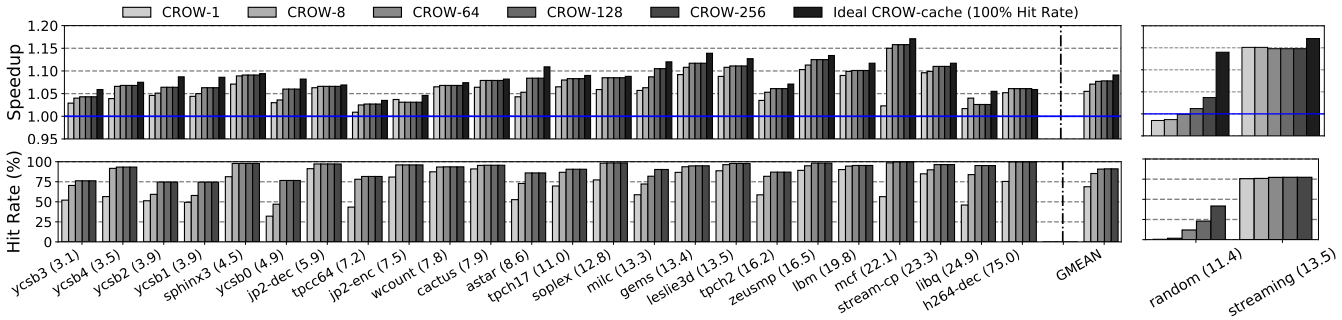


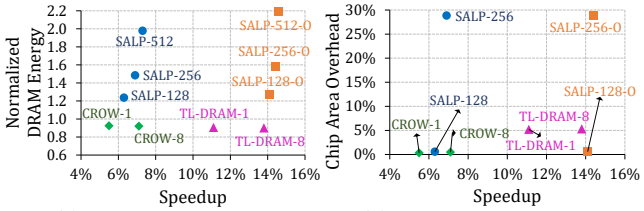
Figure 8: Speedup and CROW-table hit rate of different configurations of CROW-cache for single-core applications. The MPKI of each application is listed in parentheses.

the near segment as a cache by copying the most-recently accessed rows in the far segment to the near segment. Thus, similar to our caching mechanism, TL-DRAM requires an efficient in-DRAM row copy operation. We reuse the ACT-c command that we implement for CROW-cache to perform the row copy in TL-DRAM.

SALP [53] modifies the row decoder logic to enable parallelism among subarrays. As opposed to a conventional DRAM bank where only a single row can be active at a time, SALP enables the activation of multiple local row buffers independently from each other to provide fast access to the most-recently-activated row of each subarray. We evaluate the SALP-MASA mode, which outperforms the other two modes that Kim et al. [53] propose. We evaluate SALP with a different number of subarrays per bank, which we indicate as SALP-N_s, where N_s stands for the number of subarrays in a bank. For SALP, we use both the timeout-based and the open-page (denoted as SALP-N_s-O) row buffer management policies. The open-page policy keeps a row open until a local row buffer conflict. Although this increases the performance benefit of SALP by preventing a local row buffer from being precharged before it is reused, SALP with open-page policy consumes more energy since rows remain active for a longer time. Note that, in SALP, the in-DRAM cache capacity changes with the number of subarrays in the DRAM chip as each subarray can cache a single row in its local row buffer. To evaluate SALP with higher cache capacity, we reduce the number of rows in each subarray by increasing the number of subarrays in a bank, thus keeping the DRAM capacity constant.

Figure 11 compares the performance, energy efficiency, and DRAM chip area overhead of different configurations of CROW-cache, TL-DRAM [58], and SALP [53] for single-core workloads. We draw four conclusions from the figure.

First, all SALP configurations using the open-page policy outperform CROW-cache. However, SALP increases the DRAM energy consumption significantly as it frequently keeps multiple local row buffers active, each of which consume significant static power (as also shown in [53]). An idle LPDDR4 chip that has only a single bank with an open row draws 10.9% more current ($I_{DD}3N$) compared to the current ($I_{DD}2N$) when all banks are in closed state [73]. In SALP, the static power consumption is much higher than in conventional DRAM, since multiple local row buffers per bank can be active at the same time, whereas only one local row buffer per bank can be active in conventional DRAM.



(a) Energy vs. speedup **(b) Chip area vs. speedup**

Figure 11: CROW-cache vs. TL-DRAM [58] & SALP [53].

Second, increasing the number of subarrays in a bank increases the in-DRAM cache capacity and, thus, the performance benefits of SALP. However, with the open-page policy, SALP-256 has a 58.4% DRAM energy and 28.9% area overhead over the baseline, which is much higher than the 27.3% DRAM energy and 0.6% area overhead of SALP-128. SALP-512 has even larger DRAM energy (119.3%) and area (84.5%) overheads (not plotted). In contrast, CROW-8 *reduces* DRAM energy (by 8.2%) at very low (0.48%) DRAM chip area overhead.

Third, TL-DRAM-8 provides a higher speedup of 13.8% compared to CROW-8, which provides 7.1% speedup (while reserving the same DRAM storage capacity for caching as TL-DRAM-8). This is because the latency reduction benefit of a very small TL-DRAM near-segment, which comprises only one or eight rows, is higher than the latency reduction benefit of CROW's two-row activation. According to our circuit-level simulations, a TL-DRAM near-segment with eight rows can be accessed with a 73% reduction in tRCD and an 80% reduction in tRAS. However, this comes at the cost of high DRAM chip area overhead, as we explain next.

Fourth, in TL-DRAM, the addition of an isolation transistor to each bitline incurs high area overhead. As seen in Figure 11b, TL-DRAM-8 incurs 6.9% DRAM chip area overhead, whereas CROW-8 incurs *only* 0.48% DRAM chip area overhead.

We conclude that CROW enables a more practical and lower cost in-DRAM caching mechanism than TL-DRAM and SALP.

8.1.5 CROW-cache and Prefetcher. We evaluate the performance benefits of CROW-cache on a system with a stride prefetcher, which we implement based on the RPT prefetcher [31]. In Figure 12, we show the speedup that the prefetcher, CROW-cache, and the combination of the prefetcher and CROW-cache achieve over the baseline, which does not implement prefetching. For brevity, we only show results for a small number of workloads, which we sampled to be representative of workloads where the prefetcher provides different levels of effectiveness. We observe that, in most cases, CROW-cache operates synergistically with the prefetcher, i.e., CROW-cache serves both read and prefetch requests with low latency, which further improves average system performance by 5.7% over the prefetcher across all single-core workloads.

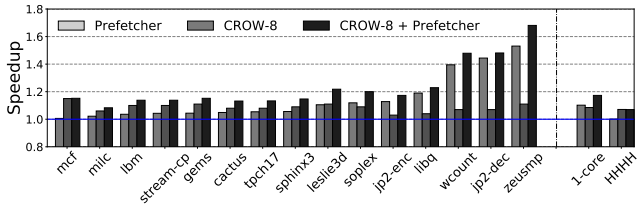


Figure 12: CROW-cache and prefetching.

8.2 CROW-ref

Our weak row remapping scheme, CROW-ref, extends the refresh interval from 64 ms to 128 ms of DRAM chip by eliminating the small set of rows that have retention time below 128 ms. For our evaluations, we assume three weak rows for each subarray, which is much more than expected, given data from real DRAM devices [51, 64, 87].[9] Extending the refresh interval provides performance and energy efficiency benefits as fewer refresh operations occur and they interfere less with application requests. CROW-ref does not incur any additional overhead other than allocating a few of the copy rows that are available with the CROW substrate. The rest of the copy rows can potentially be used by other CROW-based mechanisms, e.g., CROW-cache.

Figure 13 shows CROW-ref's average speedup and normalized DRAM energy consumption for all single-core workloads and memory intensive four-core workloads (*HHHH*) for four DRAM chip densities (8, 16, 32, 64 Gbit). We observe that, for a futuristic 64 Gbit DRAM chip, CROW-ref improves average performance by 7.1%/11.9% and reduces DRAM energy consumption by 17.2%/7.8% for single-/multi-core workloads. The energy benefits are lower for four-core workloads as DRAM accesses contribute a larger portion of the overall DRAM energy compared to single-core workloads that inject relatively few requests to the DRAM.

[9] Based on Equation 2 in Section 4.2.1, the probability of a having subarray with more than 3 weak rows across the entire DRAM is 9.3×10^{-3}.

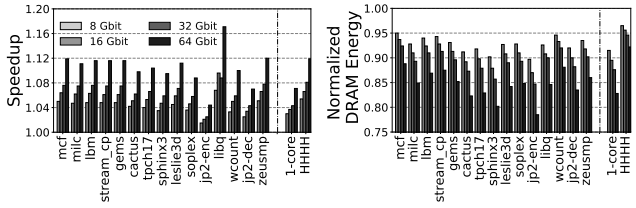
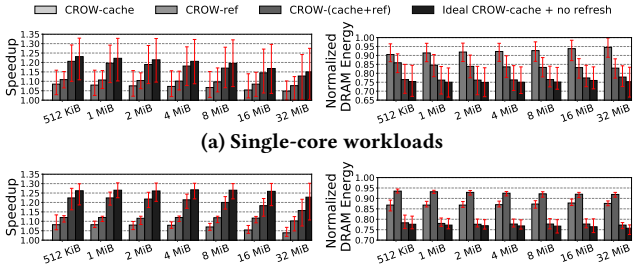


Figure 13: CROW-ref speedup and DRAM energy.

8.3 Combining CROW-cache and CROW-ref

In Section 8.1, we note that not all applications effectively use all available copy rows in CROW-cache since CROW-1 provides speedup close to CROW-cache with more copy rows. Similarly, for CROW-ref, previous work shows that there are only a small number (e.g., < 1000 on a 32 GiB DRAM [64]) of DRAM rows that must be refreshed at the lowest refresh intervals, so it is very unlikely to have more than a few weak rows (or even one) in a subarray. These two observations lead us to believe that the two mechanisms, CROW-cache and CROW-ref, can be combined to operate synergistically. We combine the two mechanisms such that CROW-cache utilizes copy rows that remain available *after* the weak row remapping of CROW-ref. Combining the two mechanisms requires only a single additional bit per CROW-table entry to indicate whether a copy row is allocated for CROW-cache or CROW-ref.

Figure 14 summarizes the performance and energy efficiency benefits of CROW-cache, CROW-ref, and their combination for an LLC capacity ranging from 512 KiB to 32 MiB and 64 Gbit DRAM chip density. The figure compares the benefits of the two mechanisms against a hypothetical CROW-based mechanism that has an ideal CROW-cache with 100% hit rate and that does not require any DRAM refresh. We make three observations from the figure.



(a) Single-core workloads

(b) Four-core workloads

Figure 14: CROW-(cache+ref) speedup and DRAM energy for different LLC capacities.

First, the two mechanisms combined together provide higher performance and energy efficiency than either mechanism alone. This is because each mechanism improves a different aspect of DRAM. Using an 8 MiB LLC, the combination of the two mechanisms provides a 20.0% performance improvement and 22.3% DRAM energy reduction for four-core workloads. The performance and energy benefits of the combined CROW-cache/CROW-ref mechanisms is larger than the the *sum* of the benefits of each individual mechanism. This is because by eliminating many refresh operations, CROW-ref decreases the interference of refresh operations with DRAM access operations. This helps CROW-cache, as a row that would have been closed by a refresh operation may now stay open, and no longer needs to be reactivated. As a result, CROW-cache can use its limited copy rows to help accelerate the activation of other rows than the ones that would have been closed by refresh.

Second, CROW-cache and CROW-ref significantly improve performance and reduce DRAM energy for *all* LLC capacities. For four-core workloads, we observe 22.4%/20.0%/15.7% performance improvement and 22.0%/22.3%/22.8% DRAM energy reduction for 1/8/32 MiB LLC capacity.

Third, the combination of CROW-cache and CROW-ref achieves performance and DRAM energy improvements close to the ideal mechanism (100% hit rate CROW-cache and no refresh). Using an 8 MiB LLC, the combination of the two mechanisms achieves 71% of the performance and 99%[10] of the DRAM energy improvements of the ideal mechanism.

We conclude that CROW is a flexible substrate that enables multiple different mechanisms to operate simultaneously, thereby improving both DRAM performance and energy efficiency.

9 RELATED WORK

To our knowledge, this paper is the first to propose a flexible and low-cost DRAM substrate that enables *multiple* mechanisms for improving DRAM performance, energy efficiency, and reliability. We briefly discuss closely related prior work that propose in-DRAM caching, latency reduction, refresh reduction, and RowHammer protection mechanisms.

In-DRAM Caching Mechanisms. In Section 8.1.4, we qualitatively and quantitatively compare CROW to the two most-closely related prior works, TL-DRAM [58] and SALP [53], which also propose in-DRAM caching. We show that CROW-cache is lower-cost to implement and consumes less DRAM energy compared to the two mechanisms. Since TL-DRAM and SALP exploit *different* observations to enable low-latency regions in DRAM, CROW-cache can be implemented in combination with both TL-DRAM and SALP to further improve system performance.

Chang et al. [8] propose LISA to enable efficient data movement between two subarrays. They also propose *LISA-VILLA*, which changes the bank architecture to include small (but fast) subarrays [67] to enable dynamic in-DRAM caching. Their approach 1) requires more changes to the DRAM chip compared to CROW-cache and 2) is orthogonal to CROW-cache as a copy row can be implemented in both the small and regular subarrays of LISA-VILLA.

Hidaka et al. [29] propose to add SRAM memory cells inside a DRAM chip to utilize as a cache. Implementing SRAM-based memory in DRAM incurs very high area overhead (e.g., 38.8% of the DRAM chip area for 64 KiB SRAM cache as shown in [53, 58]). Prior works [22, 27, 94] that propose implementing multiple row buffers (that can be used as cache) also suffer from high area overhead. CROW-cache can be used in combination with all these prior mechanisms to reduce the latency of fetching data from a DRAM row to the in-DRAM cache.

Multiple Clone Row (MCR-DRAM) [10] is based on an idea that is similar to simultaneously activating multiple duplicate rows. The key idea is to dynamically configure a DRAM bank into different access-latency regions, where a region stores data in a single row or duplicates data into multiple rows. MCR-DRAM does *not* propose a hardware-managed in-DRAM cache as it requires support from the operating system (or the application) to manage 1) the size of each region and 2) data movement between different regions. However, both operations are difficult to perform for the operating system due to memory fragmentation, which complicates dynamic resizing of a region and determining which data would benefit

[10]The combination of CROW-cache and CROW-ref almost reaches the energy reduction of the ideal mechanism because the ideal mechanism *always* hits in the CROW-table, and thus *always* uses an ACT-t command to activate a row (which consumes more energy than a regular ACT).

from a low-latency region. In contrast, CROW is much more practical to implement, as it is hardware managed and thus completely transparent to the software.

Mitigating Refresh Overhead. Many prior works tackle the refresh overhead problem in DRAM. RAIDR [64] proposes to bin DRAM rows based on their retention times and perform refresh operations at a different rate on each bin. Riho et al. [93] use a technique based on simultaneous multiple-row activation. However, their mechanism is specifically designed for optimizing only refresh operations, and thus it is not as flexible as the CROW substrate, which provides copy rows that enable multiple orthogonal mechanisms. Other prior works [2, 3, 7, 11, 12, 19, 33, 38, 41, 42, 44, 49, 50, 66, 69, 76, 83, 84, 86–88, 107, 114] propose various techniques to optimize refresh operations. These works are specifically designed for *only* mitigating the refresh overhead. In contrast, CROW provides a versatile substrate that can simultaneously reduce the refresh overhead, reduce latency, and improve reliability.

Reducing DRAM Latency. ChargeCache [26] reduces the average DRAM latency based on the observation that recently-precharged rows, which are fully restored, can be accessed faster compared to rows that have leaked some charge and are about to be refreshed soon. Note that ChargeCache enables low-latency access when DRAM rows are repeatedly activated in very short intervals, e.g., 1 ms. In contrast, a copy row in CROW-cache enables low-latency access to a regular row for an indefinite amount of time (until the row is evicted from CROW-table). Thus, CROW-cache captures higher amount of in-DRAM locality. CROW-cache is also orthogonal to ChargeCache and the two techniques can be implemented together.

Recent studies [4, 6, 47, 48, 57] propose mechanisms to reduce the DRAM access latency by reducing the margins present in timing parameters when operating under appropriate conditions (e.g., low temperature). Other works [7, 9, 22, 46, 57, 60, 101, 105, 119] propose different methods to reduce DRAM latency. These works are orthogonal to CROW, and they can be implemented along with our mechanism to further reduce DRAM latency.

10 CONCLUSION

We propose CROW, a low-cost substrate that partitions each DRAM subarray into two regions (regular rows and copy rows) and enables independent control over the rows in each region. We leverage CROW to design two new mechanisms, CROW-cache and CROW-ref, that improve DRAM performance and energy-efficiency. CROW-cache uses a copy row to duplicate a regular row and simultaneously activates a regular row together with its duplicated copy row to reduce the DRAM activation latency (by 38% in our experiments). CROW-ref remaps retention-weak regular rows to strong copy rows, thereby reducing the DRAM refresh rate. CROW's flexibility allows us to simultaneously employ both CROW-cache and CROW-ref to provide 20.0% speedup and 22.3% DRAM energy savings over conventional DRAM. We conclude that CROW is a flexible DRAM substrate that enables a wide range of mechanisms to improve performance, energy efficiency, and reliability. We hope future work exploits CROW to devise more use cases that can take advantage of its low-cost and versatile substrate.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for feedback. We thank the SAFARI Research Group members for feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: Alibaba, Facebook, Google, Huawei, Intel, Microsoft, and VMware. This research was supported in part by the Semiconductor Research Corporation.

REFERENCES

[1] Arizona State Univ., NIMO Group, "Predictive Technology Model," http://ptm.asu.edu/, 2012.
[2] S. Baek *et al.*, "Refresh Now and Then," *TC*, 2014.
[3] I. Bhati *et al.*, "Coordinated Refresh: Energy Efficient Techniques for DRAM Refresh Scheduling," in *ISLPED*, 2013.
[4] K. Chandrasekar *et al.*, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
[5] K. Chandrasekar *et al.*, "DRAMPower: Open-Source DRAM Power & Energy Estimation Tool," http://www.drampower.info.
[6] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[7] K. K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[8] K. K. Chang *et al.*, "Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM," in *HPCA*, 2016.
[9] K. K. Chang *et al.*, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," *SIGMETRICS*, 2017.
[10] J. Choi *et al.*, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
[11] Z. Cui *et al.*, "DTail: A Flexible Approach to DRAM Refresh Management," in *ICS*, 2014.
[12] P. G. Emma *et al.*, "Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications," *IEEE Micro*, 2008.
[13] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[14] M. Ferdman *et al.*, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *ASPLOS*, 2012.
[15] J. E. Fritts *et al.*, "Mediabench II Video: Expediting the Next Generation of Video Systems Research," in *Electronic Imaging*, 2005.
[16] M. Ghasempour *et al.*, "Armor: A Run-Time Memory Hot-Row Detector," http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer, 2015.
[17] S. Ghose *et al.*, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.
[18] S. Ghose *et al.*, "Understanding the Interactions of Workloads and DRAM Types: A Comprehensive Experimental Study," arXiv:1902.07609 [cs.AR], 2019.
[19] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in *MICRO*, 2007.
[20] D. Gruss *et al.*, "Another Flip in the Wall of Rowhammer Defenses," in *SP*, 2018.
[21] D. Gruss *et al.*, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," in *DIMVA*, 2016.
[22] N. D. Gulur *et al.*, "Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities," in *SC*, 2012.
[23] A. Gutierrez *et al.*, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *IISWC*, 2011.
[24] T. Hamamoto *et al.*, "On the Retention Time Distribution of Dynamic Random Access Memory (DRAM)," *TED*, 1998.
[25] G. Hamerly *et al.*, "SimPoint 3.0: Faster and More Flexible Program Phase Analysis," *JILP*, 2005.
[26] H. Hassan *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[27] E. Herrero *et al.*, "Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments," *TC*, 2012.
[28] J. Hestness *et al.*, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *IISWC*, 2014.
[29] H. Hidaka *et al.*, "The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory," *IEEE Micro*, 1990.
[30] Y. Huang *et al.*, "Moby: A Mobile Benchmark Suite for Architectural Simulators," in *ISPASS*, 2014.
[31] S. Iacobovici *et al.*, "Effective Stream-Based and Execution-Based Data Prefetching," in *ICS*, 2004.
[32] E. Ipek *et al.*, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.
[33] C. Isen and L. John, "ESKIMO - Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem," in *MICRO*, 2009.
[34] ITRS Reports, http://www.itrs2.net/itrs-reports.html.
[35] JEDEC Solid State Technology Assn., "JESD79-3F: DDR3 SDRAM Standard," July 2012.
[36] JEDEC Solid State Technology Assn., "JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard," March 2017.
[37] JEDEC Solid State Technology Assn., "JESD79-4B: DDR4 SDRAM Standard," June 2017.
[38] M. Jung *et al.*, "Omitting Refresh: A Case Study for Commodity and Wide I/O DRAMs," in *MEMSYS*, 2015.
[39] M. Kandemir *et al.*, "Memory Row Reuse Distance and Its Role in Optimizing Application Performance," in *SIGMETRICS*, 2015.
[40] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.

[41] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[42] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[43] S. Khan *et al.*, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
[44] S. Khan *et al.*, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[45] D.-H. Kim *et al.*, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2015.
[46] J. Kim *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[47] J. S. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.
[48] J. S. Kim *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.
[49] J. Kim and M. C. Papaefthymiou, "Dynamic Memory Design for Low Data-Retention Power," in *PATMOS*, 2000.
[50] J. Kim and M. C. Papaefthymiou, "Block-based Multiperiod Dynamic Memory Design for Low Data-Retention Power," *TVLSI*, 2003.
[51] K. Kim and J. Lee, "A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs," *EDL*, 2009.
[52] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[53] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[54] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[55] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2015.
[56] Y. Konishi *et al.*, "Analysis of Coupling Noise Between Adjacent Bit Lines in Megabit DRAMs," *JSSC*, 1989.
[57] D. Lee *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[58] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[59] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
[60] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," *SIGMETRICS*, 2017.
[61] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[62] E. Lee *et al.*, "TWiCe: Time Window Counter Based Row Refresh to Prevent Row-Hammering," *CAL*, 2018.
[63] M. Lipp *et al.*, "Nethammer: Inducing Rowhammer Faults Through Network Requests," *arXiv*, 2018.
[64] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[65] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[66] S. Liu *et al.*, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," *ASPLOS*, 2012.
[67] S.-L. Lu *et al.*, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
[68] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
[69] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
[70] J. Mandelman *et al.*, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," *IBM JRD*, 2002.
[71] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," https://www.cs.virginia.edu/stream/.
[72] Micron Technology, Inc., "RLDRAM 2 and 3 Specifications," http://www.micron.com/products/dram/rldram-memory.
[73] Micron Technology, Inc., "x64 Mobile LPDDR4 SDRAM Datasheet," https://prod.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr4/272b_z9am_qdp_mobile_lpddr4.pdf.
[74] Y. Mori *et al.*, "The Origin of Variable Retention Time in DRAM," in *IEDM*, 2005.
[75] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[76] J. Mukundan *et al.*, "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems," in *ISCA*, 2013.
[77] N. Muralimanohar *et al.*, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.
[78] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
[79] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
[80] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.
[81] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[82] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2015.
[83] P. Nair *et al.*, "A Case for Refresh Pausing in DRAM Memory Systems," in *HPCA*, 2013.
[84] P. J. Nair *et al.*, "Refresh Pausing in DRAM Memory Systems," *TACO*, 2014.
[85] H. Park *et al.*, "Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems," in *ASPLOS*, 2013.
[86] K. Patel *et al.*, "Energy-Efficient Value-Based Selective Refresh for Embedded DRAMs," in *PATMOS*, 2005.
[87] M. Patel *et al.*, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," *ISCA*, 2017.
[88] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[89] Rambus Inc., "DRAM Power Model," http://www.rambus.com/energy/. 2016.
[90] K. Razavi *et al.*, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Sec.*, 2016.
[91] M. Redeker *et al.*, "An Investigation into Crosstalk Noise in DRAM Structures," in *MTDT*, 2002.
[92] P. J. Restle *et al.*, "DRAM Variable Retention Time," in *IEDM*, 1992.
[93] Y. Riho and K. Nakazato, "Partial Access Mode: New Method for Reducing Power Consumption of Dynamic Random Access Memory," *TVLSI*, 2014.
[94] S. Rixner, "Memory Controller Optimizations for Web Servers," in *MICRO*, 2004.
[95] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.
[96] SAFARI Research Group, "CROW — GitHub Repository," https://github.com/CMU-SAFARI/CROW.
[97] SAFARI Research Group, "Ramulator: A DRAM Simulator — GitHub Repository," https://github.com/CMU-SAFARI/ramulator.
[98] Y. Sato *et al.*, "Fast Cycle RAM (FCRAM); A 20-ns Random Row Access, Pipe-Lined Operating DRAM," in *VLSIC*, 1998.
[99] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.
[100] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[101] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[102] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
[103] S. M. Seyedzadeh *et al.*, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *ISCA*, 2018.
[104] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor," *ASPLOS*, 2000.
[105] Y. Son *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," *ISCA*, 2013.
[106] Standard Performance Evaluation Corp., "SPEC CPU® 2006," http://www.spec.org/cpu2006/, 2006.
[107] J. Stuecheli *et al.*, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *MICRO*, 2010.
[108] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[109] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
[110] A. Tatar *et al.*, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.
[111] Transaction Processing Performance Council, "TPC Benchmarks," http://www.tpc.org/.
[112] A. N. Udipi *et al.*, "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," in *ISCA*, 2010.
[113] V. van der Veen *et al.*, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.
[114] R. Venkatesan *et al.*, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.
[115] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.
[116] Y. Wang *et al.*, "Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration," in *MICRO*, 2018.
[117] Y. Xiao *et al.*, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Sec.*, 2016.
[118] D. Yaney *et al.*, "A Meta-Stable Leakage Phenomenon in DRAM Charge Storage - Variable Hold Time," in *IEDM*, 1987.
[119] T. Zhang *et al.*, "Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation," in *ISCA*, 2014.
[120] X. Zhang *et al.*, "Exploiting DRAM Restore Time Variations in Deep Sub-micron Scaling," in *DATE*, 2015.
[121] X. Zhang *et al.*, "Restore Truncation for Performance Improvement in Future DRAM Systems," in *HPCA*, 2016.
[122] W. Zhao and Y. Cao, "New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration," *TED*, 2006.
[123] Y. Zhu *et al.*, "Microarchitectural Implications of Event-Driven Server-Side Web Applications," in *MICRO*, 2015.
[124] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent No. 5,630,096, 1997.