Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration

Yaohua Wang[†§] Arash Tavakkol[†] Lois Orosa[†*] Saugata Ghose[‡] Nika Mansouri Ghiasi[†]
Minesh Patel[†] Jeremie S. Kim[‡†] Hasan Hassan[†] Mohammad Sadrosadati[†] Onur Mutlu[†‡]

[†]*ETH Zürich* [§]*National University of Defense Technology*
[‡]*Carnegie Mellon University* [*]*University of Campinas*

Long DRAM access latency is a major bottleneck for system performance. In order to access data in DRAM, a memory controller (1) activates (i.e., opens) a row of DRAM cells in a cell array, (2) restores the charge in the activated cells back to their full level, (3) performs read and write operations to the activated row, and (4) precharges the cell array to prepare for the next activation. The restoration operation is responsible for a large portion (up to 43.6%) of the total DRAM access latency.

We find two frequent cases where the restoration operations performed by DRAM do not need to fully *restore the charge level of the activated DRAM cells, which we can exploit to reduce the restoration latency. First, DRAM rows are periodically refreshed (i.e., brought back to full charge) to avoid data loss due to charge leakage from the cell. The charge level of a DRAM row that will be* refreshed soon *needs to be only* partially *restored, providing just enough charge so that the refresh can correctly detect the cells' data values. Second, the charge level of a DRAM row that will be* activated again soon *can be only partially restored, providing just enough charge for the activation to correctly detect the data value. However, partial restoration needs to be done carefully: for a row that will be activated again soon, restoring to only the* minimum *possible charge level can undermine the benefits of complementary mechanisms that reduce the activation time of* highly-charged *rows.*

To enable effective *latency reduction for both activation and restoration, we propose* charge-level-aware look-ahead partial restoration *(CAL). CAL consists of two key components. First, CAL accurately predicts the* next access time*, which is the time between the current restoration operation and the next activation of the same row. Second, CAL uses the predicted next access time and the next refresh time to reduce the restoration time, ensuring that the amount of partial charge restoration is enough to maintain the benefits of reducing the activation time of a highly-charged row. We implement CAL fully in the memory controller, without any changes to the DRAM module. Across a wide variety of applications, we find that CAL improves the average performance of an 8-core system by 14.7%, and reduces average DRAM energy consumption by 11.3%.*

1. Introduction

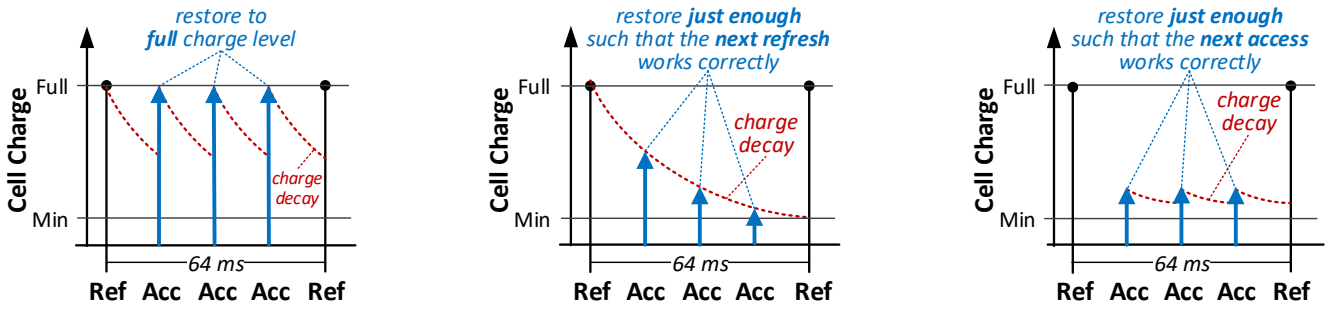
Main memory (i.e., DRAM) access latency has become a critical bottleneck for system performance [57, 58]. While DRAM capacity has increased due to manufacturing process technology scaling, DRAM access latency has not decreased significantly for decades [9, 11, 42, 46]. Due to a combination of increasing core counts, modern applications that are increasingly data-intensive, and inherent limitations to increasing memory bandwidth, the DRAM access latency is becoming a growing obstacle to improving overall system performance [13, 17, 24, 29, 41, 45, 46, 52, 54, 57, 58, 61, 70, 73, 75, 85].

DRAM access latency is predominantly composed of the latency of three fundamental DRAM operations: activation, restoration, and precharge. DRAM stores data in two-dimensional arrays of *cells*, where each cell holds one bit of data by storing a small amount of charge in a capacitor. In order to read from or write to DRAM cells, the memory controller must first *activate* (i.e., open) the array row that contains the target cells. During activation, each cell in the row shares its charge with its corresponding *bitline*. A *sense amplifier* detects the correct data value on the bitline, and the data value is stored in a latch (as part of the *row buffer*), after which the memory controller can issue read and write commands. The process of sharing charge between the cell and the bitline depletes the amount of charge in the cell. To compensate for this charge depletion and avoid data loss, DRAM must *restore* the charge level of the cell, as shown in Figure 1a. Once restoration is complete, and the memory controller is done issuing read and write commands to the activated row, the memory controller *precharges* the array, which empties the row buffer and prepares the array for the next activation operation.

Because a DRAM cell is made up of a capacitor, the cell leaks charge even when it is not accessed. In order to prevent data loss, the DRAM issues periodic *refresh* operations to *all* cells. A refresh operation brings the charge level of a cell back to its full value. In DDR DRAM, each cell must be refreshed every 64 ms under normal operating conditions [25, 26, 51].

Prior works [24, 45, 73, 87] show that the charge level of cells can be exploited to reduce the activation and restoration latencies. A DRAM cell that has been accessed or refreshed recently (e.g., within the last 1 ms) contains a high amount of charge, as very little charge leakage has occurred since the last restoration or refresh operation. A number of mechanisms [24, 45, 73] exploit this observation to reduce the *activation latency* of a highly-charged cell.[1] Likewise, if an activated cell is scheduled to be refreshed soon, the restoration operation does *not* need to *fully* restore the cell's charge, as the refresh operation will take care of this. As a result, the *Restore Truncation* mechanism [87] only *partially restores* the charge after an access, such that the amount of charge is just enough to ensure that the refresh operation can still correctly read the data, as shown in Figure 1b. While Restore

[1] For clarity and consistency, we describe the opportunities at the *cell level*. In reality, opportunities occur at the *row level*.



(a) Normal Restoration (b) Refresh-Based Partial Restoration (c) CAL Partial Restoration

Figure 1: (a) Normal charge restoration for each memory access (*Acc*) and refresh (*Ref*), (b) charge restoration using Restore Truncation [87], (c) charge restoration using CAL.

Truncation helps to reduce DRAM access latency, we find that it does *not* achieve the full benefits of partial restoration. We make two new key observations about restoration in DRAM.

First, in addition to using partial restoration for soon-to-be-refreshed DRAM cells, we can use partial restoration for soon-to-be-*reactivated* DRAM cells as well, to enable further access latency reduction. Recent work has shown that DRAM access patterns have high temporal locality, and that a recently-accessed row is likely to be accessed again soon [24]. If we can *predict* that a currently-activated cell will be activated again soon (e.g., in the next 16 ms), we can partially restore the charge level of the cell, to a level *just high enough* for the future activation to correctly read the data.

Second, while partial restoration for soon-to-be-reactivated DRAM cells can provide performance benefits, we observe that some of the latency savings from partial restoration can *work against* those mechanisms that reduce the activation latency of highly-charged cells [24, 45, 73]. If we were to use a naive approach to partial restoration, where the charge level of a cell is reduced to the *minimum* charge level necessary for the next activation, most cells that were previously highly charged can *no longer* benefit from activation latency reduction mechanisms. We find that there is an optimal amount of charge for a cell that allows us to still perform partial restoration and thus reduce the restoration latency, while also taking advantage of the activation latency reduction mechanisms, to maximize the *overall* access latency reduction.

Based on our two observations, **our goal** in this paper is to design a mechanism that can effectively enable partial restoration for both soon-to-be-refreshed and soon-to-be-reactivated cells, while still effectively exploiting the benefits of activation latency reduction.

To this end, we propose *charge-level-aware* look-ahead partial restoration (CAL). The key idea of CAL is to provide balanced activation and restoration latency reductions for soon-to-be-reactivated rows, in a way that maximizes the overall reduction in DRAM latency. CAL consists of two major components. First, CAL *looks ahead* to future memory accesses, by predicting whether an activated DRAM row will be reactivated soon (e.g., in the next 16 ms). This predictor achieves a very high accuracy of 96% for single-core applications, and 98% for multi-core workloads, on average. Second, CAL uses this prediction and the scheduled refresh time of the activated row to apply *charge-level-aware* partial restoration. Using our second observation about the trade-off between restoration latency reduction and activation latency reduction, CAL partially restores the charge of the DRAM cells in the activated row to a level that maximizes the *overall* access latency reduction (as shown in Figure 1c), due to both restoration and activation.

Our evaluations show that CAL improves the performance of a system using DDR4 DRAM by 14.7% and reduces DRAM energy by 11.3%, on average across a wide range of 8-core workloads. We show that the performance benefits of CAL are robust across many system and mechanism parameters (e.g., page management policy, address mapping scheme, restoration level, operating temperature). CAL outperforms both Restore Truncation [87] and ChargeCache [24], two state-of-the-art works that exploit partial restoration and highly-charged rows, respectively, and outperforms a system that employs a simple combination of Restore Truncation and ChargeCache by 9.8% for our 8-core workloads. CAL requires no changes to the DRAM chip or module. It can be combined easily with other complementary DRAM architecture optimizations (e.g., [11, 12, 13, 14, 23, 39, 46, 61, 67, 75, 86]) to provide further performance and energy improvements.

We make the following major contributions in this work:

- We observe that for DRAM rows that are accessed again in the near future, we can safely apply partial restoration to reduce the restoration latency. However, we observe that there is a trade-off between restoration latency and activation latency reduction for highly-charged rows, and that the largest overall DRAM access latency reduction can be achieved by balancing the restoration and activation latency reductions for such rows.
- We propose charge-level-aware look-ahead partial restoration (CAL), a mechanism that effectively reduces the DRAM access latency by carefully exploiting both partial restoration and activation latency reduction. CAL requires no changes to the DRAM chip or module, and can be implemented completely in the memory controller.
- We comprehensively evaluate the performance and energy efficiency of CAL. We show that CAL substantially improves both the performance and energy efficiency of systems with DDR4 DRAM. We also demonstrate that CAL's performance gains are robust across many system and mechanism parameters.

2. Background

We first introduce necessary background on DRAM organization and fundamental DRAM operations. We next discuss state-of-the-art mechanisms that exploit the charge level of a DRAM cell to reduce the access latency of DRAM.

2.1. DRAM Organization

In the main memory subsystem, DRAM is organized in a hierarchical manner, The topmost level of the hierarchy is the memory *channel*. Each channel connects to one or more DRAM modules that share a pin-limited bus to the processor. Each channel has its own *memory controller* that sends commands to, and manages the state of, all modules that are connected to the channel.

Each module consists of multiple DRAM *chips*, where each chip contains millions of DRAM *cells*. As shown in Figure 2, a DRAM cell consists of a single capacitor, which holds one bit of data in the form of stored charge, and an access transistor. These cells are organized into *banks*, where each bank can perform operations in parallel with other banks.[2] A bank contains multiple two-dimensional arrays of DRAM cells, which are called *subarrays* [39]. Figure 2 also shows the high-level design of a subarray. In a subarray, each cell in a *row* shares a common *wordline*, which is used to enable the access transistor of all of the cells in the row (i.e., activate the row), and each cell in a *column* shares a common *bitline*, which is used to sense data from the cell in the activated row. Each bitline is connected to its own local *sense amplifier*, where the sense amplifiers of the subarray serve as the *row buffer*.

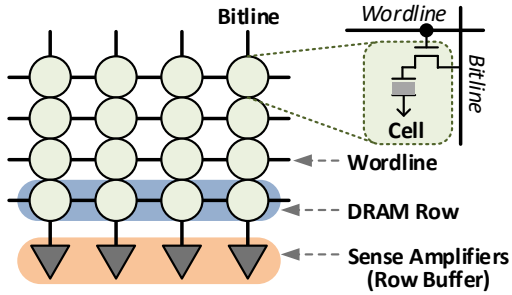


Figure 2: DRAM subarray organization and cell structure.

Several chips in the DRAM module are grouped together into a *rank*. Each cache line stored in DRAM is distributed across all of the chips in a rank. As a result, in order to operate on a single cache line, all chips in a rank are accessed concurrently, and they respond to the same DRAM commands.

2.2. Fundamental DRAM Operations

There are five fundamental operations that need to be performed when accessing data in DRAM. (1) *Activation* opens one of the DRAM rows in a bank, and copies the data in the opened row into the row buffer. (2) *Restoration* ensures that the charge that is drained from each cell in the DRAM row during activation is restored to its full level, to prevent data loss. (3) *Reads* and (4) *writes* can be performed once the data of an activated row is copied to the row buffer. (5) *Precharge* releases the data from the row buffer when the memory controller is done issuing reads and writes to the activated row, and prepares the bank to activate a different row.

Figure 3 shows a timeline of the commands issued to perform a read (top) or a write (bottom) to a single cache line of data. The memory controller issues four commands: (1) *ACT* (activate), (2) *READ* or (3) *WRITE*, and (4) *PRE* (precharge). Note that restoration does *not* have an *explicit* command, and is instead triggered *automatically* after an *ACT* command. The time spent on each operation is dictated by a set of *timing parameters* that are determined by DRAM vendors [25, 26]. While each command operates at a row granularity, for simplicity, we describe how the DRAM operations affect a single DRAM cell.

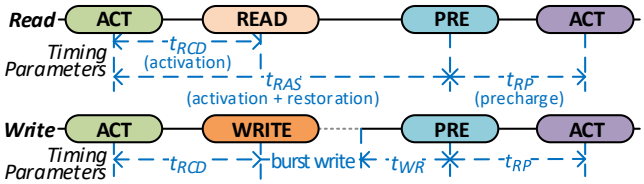


Figure 3: Commands and timing parameters for reading (top) and writing (bottom) data from/to DRAM.

Figure 4 shows the charge levels of a cell that is activated and restored. In the initial *precharged* state (❶ in Figure 4), the bitline is held at a voltage level of $V_{DD}/2$, where V_{DD} is the full DRAM supply voltage. The wordline is at 0 V, and therefore the bitline is disconnected from the capacitor. After the memory controller issues an *ACT* command, the wordline is raised to V_h, thereby connecting the DRAM cell capacitor to the bitline. As the voltage of the capacitor (in this example) is higher than that of the bitline, charge flows into the bitline, raising the voltage level of the bitline up to $V_{DD}/2 + \delta$ (❷). This process is called *charge sharing*. The sense amplifier then senses the deviation on the bitline and amplifies that deviation correspondingly. This phase, referred to as *sense amplification*, eventually drives the voltage level of the bitline and the cell back to the original voltage state of the cell (V_{DD} in this example).

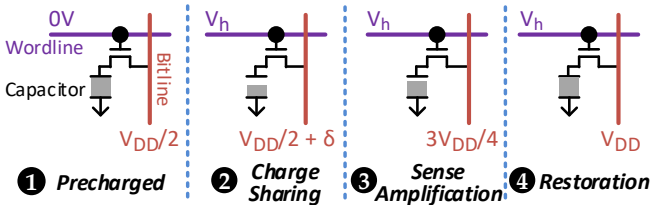


Figure 4: Cell charge levels during DRAM operations.

As soon as the sense amplifier has sufficiently amplified the data on the bitline (e.g., the voltage level reaches $3V_{DD}/4$; ❸), the memory controller can issue a *READ* or *WRITE* command to access the cell's data in the row buffer. The time taken to reach this state (❸) after the *ACT* command is specified by the timing parameter t_{RCD}, as shown in Figure 3. After the *READ* or *WRITE* command is issued, the sense amplification phase continues to drive the voltage on the bitline until the voltage level of the bitline and the cell reaches V_{DD}

[2] Modern DRAM architectures such as DDR4 DRAM employ *bank groups* [26], where each bank group contains several banks. Bank grouping is used to provide more banks at low cost for a DRAM module.

(❹). In other words, the original charge level of the cell is *fully restored* to its original value for a *READ*, or correctly updated to the new value for a *WRITE*. For DRAM read requests, the latency for the cell to be fully restored after *ACT* is determined by the timing parameter t_{RAS}. For DRAM write requests, the time taken to fully update the cell is determined by t_{WR}. After restoration, the bitline can be precharged using the *PRE* command to prepare the subarray for a future access to a different row. This process disconnects the cell from the bitline by lowering the voltage on the wordline. It then resets the voltage of the bitline to $V_{DD}/2$. The time to complete the precharge operation is specified by the timing parameter t_{RP}.

2.3. Charge Leakage and Refresh

A DRAM cell starts leaking charge after it is disconnected from the bitline [50, 51]. As described above, a DRAM access fully restores the charge in a cell, but if the cell is *not* accessed for a long time, the charge level of the cell eventually decays to a point where the original data value cannot be sensed correctly. To maintain data integrity, DRAM cells have to be periodically *refreshed* to a full charge level [50, 51, 62].

According to the DDR3 and DDR4 DRAM specifications, a DRAM cell must be refreshed every 64 ms [25, 26]. To achieve this, the DRAM rows in a chip are assigned to one of 8K refresh bins. One bin of rows is refreshed at a time by an auto-refresh command (*REF*). The interval between *REF* commands is known as the refresh interval (t_{REFI}), which is 7.8 µs in DDR3/DDR4 (i.e., 64 ms/8K). The DRAM chip maintains an internal counter to determine which bin to refresh when it receives a *REF* command. As in prior work [5, 22, 36, 51, 87], we assume that the memory controller knows the mapping between bin and row addresses. In other words, the memory controller has complete knowledge of when each bin is refreshed and the corresponding rows in each bin.

2.4. Exploiting Charge Levels to Reduce Latency

Conventional DRAM chips perform activation and restoration operations for a *fixed latency*, which is determined by the value of the timing parameters shown in Figure 3. However, prior works [24, 45, 73, 87] show that the *current charge level* of a cell can be exploited to reduce the activation and restoration latencies for a cell in several cases.

The charge level of a DRAM cell has a noticeable effect on the *activation latency* [24, 45, 73]. If a cell has a high charge level, the corresponding voltage perturbation process on the bitline during activation is faster, and consequently, the sense amplifier takes less time to reach states ❸ and ❹ in Figure 4. ChargeCache [24] is a state-of-the-art mechanism that exploits this insight to safely reduce the t_{RCD} and t_{RAS} timing parameters (see Figure 3) for a highly-charged cell. ChargeCache tracks rows that have been recently accessed. The charge level of the cells in such a row is high, as only a short amount of time has elapsed since the cells were last restored to the *full* charge level. Therefore, if a recently-accessed row is activated again within a short time interval (e.g., 1 ms), ChargeCache uses smaller t_{RCD} and t_{RAS} values for the row, which reduces the overall DRAM access latency.

A similar approach can be applied to reduce the *restoration* latency. In a conventional DRAM chip, each *ACT* command triggers a restoration operation that *fully* restores the charge level of the cells in the activated row. Likewise, each refresh operation *fully* restores the charge level of a cell at a fixed time interval (every 64 ms in DDRx DRAM). Zhang et al. [87] observe that it is *not* necessary for a *restoration* operation to *fully* restore the charge level of a cell if the cell is scheduled to be *refreshed* soon after the restoration operation takes place. They propose the *Restore Truncation* mechanism, which *partially* restores the cell charge level such that the charge in the cell is just enough to retain correct data until the next refresh of the cell. In order to determine the latency of partial restoration, Restore Truncation divides the 64 ms refresh interval into four 16 ms *sub-windows*. A cell that is activated in the last sub-window (i.e., the one closest to the next refresh operation) will be refreshed in less than 16 ms, and is therefore restored to the lowest of the partial charge levels possible in Restore Truncation. A cell that is activated in an earlier sub-window is restored to a higher partial charge level, as it would take longer before the next refresh operation takes place on the cell. The level of the restored charge can be reduced by decreasing the values of t_{RAS} and t_{WR} (shown in Figure 3). As these timing parameter values are reduced, the overall DRAM access latency reduces.

3. Motivation

As we discuss in Section 2.4, prior works propose mechanisms that reduce either the *activation* latencies of highly-charged cells *or* the *restoration* latencies of soon-to-be-refreshed cells. We find that we can use similar principles to take advantage of *new opportunities* for latency reduction that are complementary to these existing approaches. In particular, we can apply the concept of partial restoration to *soon-to-be-reactivated* cells in addition to soon-to-be-refreshed cells.

In this section, we perform a detailed study of opportunities and trade-offs related to performing partial restoration on *soon-to-be-reactivated* cells across a wide range of single-core and multi-core workloads (see Section 5 for our experimental methodology). We discuss *two key observations* that emerge from our study: (1) there are many unexploited opportunities to apply partial restoration to soon-to-be-reactivated cells (Section 3.1); and (2) there is a trade-off between reducing the restoration latency and reducing the activation latency of highly-charged cells, and these two reductions can work against each other (Section 3.2).

3.1. Partial Restoration of Soon-to-Be-Reactivated Rows

As we discuss in Section 2.4, both the restoration operation during a memory access and a refresh operation restore the charge level of a cell. The Restore Truncation mechanism [87] uses this insight to reduce the restoration operation latency when a row activation (which triggers restoration; see Section 2.2) takes place shortly before a refresh operation. Likewise, if a row activation takes place shortly before *another activation* of the same row (which we call a *soon-to-be-reactivated* row), we can similarly reduce the latency of the restoration operation that is triggered by the *earlier* row activation.

To understand the potential of partial restoration for soon-to-be-reactivated rows, we study the distribution of the *access-to-access* interval, which we define as the time between a PRE command to the bank that has the row open and the next ACT command to the *same* row, across the memory operations of an application. We compare the access-to-access interval with the *access-to-refresh* interval, which is the time between a PRE command to the bank that has the row open and the next refresh operation to the same row in the bank. As we discuss in Section 2.4, Restore Truncation [87] categorizes the *access-to-refresh* interval into one of four 16 ms sub-windows. We use a similar methodology.

Figure 5 shows the distribution of the access-to-refresh (left bar) and access-to-access (right bar) intervals for both single-core and 8-core workloads. We break down the interval distributions into five categories based on 16 ms sub-windows. We make two observations from the figure. First, for both single-core and 8-core workloads, only 25% of memory accesses are followed by a refresh in less than 16 ms, and, thus, can use the smallest restoration latency. This is because refresh happens at *fixed time intervals*, independent of the memory access pattern. As a result, when we divide the refresh interval into four 16 ms sub-windows, the number of memory accesses that fall into each sub-window tends to be similar. Second, unlike the access-to-refresh interval, the *vast majority* of the access-to-access intervals (95% for single-core, 97% for 8-core workloads) are less than 16 ms. This is due to a combination of the access locality of applications, and the high number of row conflicts that occur due to bank conflicts [24, 38, 39, 46, 55, 59, 60, 73, 88]. As a result, a row that is to be accessed again is likely to be first closed due to a bank conflict and then *reactivated* for the next access. Prior work [87] demonstrates that partially restoring the charge level of a cell, such that it has enough charge to be sensed correctly in the next 16 ms, yields a 57.1% decrease in the restoration latency.

In order to use partial restoration on a soon-to-be-reactivated row, a mechanism must be able to *predict* whether a row will be reactivated soon (e.g., in the next 16 ms).[3] This prediction needs to be made at the time the restoration operation starts. We find that we can accurately predict the next access-to-access interval of a row based on the *last* access-to-access interval length. Figure 6 shows the relationship between last and next access-to-access intervals of the same row for single-core and 8-core workloads. In the figure, we say that an access-to-access interval is *small* if it is less than 16 ms, and that it is *large* otherwise. We classify last and next interval pairs into four types: (small, small), (small, large), (large, small), and (large, large). DRAM rows that are accessed only once during the entire execution are *not* included. As shown in the figure, 96% of the pairs for single-core workloads, and 98% of the pairs for 8-core workloads, are classified as (small, small). In other words, if the last access-to-access interval is less than 16 ms, the next access-to-access interval is *highly likely* to also be less than 16 ms. We conclude that we can simply use the last access-to-access interval type to very accurately predict the next access-to-access interval type.

3.2. Balancing Activation and Restoration Latency Reductions

As we discuss in Section 2.4, prior works show that the activation latency can be reduced for a DRAM cell with a *high* charge level [24, 45, 73]. We can combine activation latency reduction for highly-charged cells with partial restoration for soon-to-be-reactivated cells. However, partial restoration can reduce or even eliminate the ability to reduce the activation latency for a highly-charged cell that is soon to be reactivated. This is because partial restoration reduces the charge level of the cell, which can cause the cell to *no longer* be highly charged by the time the reactivation takes place. If we naively combine an activation latency reduction mechanism [24] with a restoration latency reduction mechanism [87], where the partial restoration mechanism works to maximize the restoration latency reduction, the combined mechanisms yield only a slight improvement (up to 2.6%, see Section 6.1) over the individual mechanisms.

We find that there is a fundamental trade-off between restoration latency reduction and activation latency reduction. We can exploit this trade-off to maximize the overall reduction in DRAM access latency. To understand the trade-off, we perform SPICE simulations of the core DRAM circuitry (i.e., row decoder, cell capacitor, access transistor, sense amplifier, bitline capacitance, and resistance) based on the PTM low-power transistor models [3], with parameters taken from the Rambus model [65]. These SPICE simulations allow us to observe how the charge level of a cell at the time an ACT command is issued affects the required activation latency.

[3] Even though we find significant diversity in the access-to-access interval distribution if we study smaller sub-windows (e.g., 8 ms; not shown in Figure 6), we also find that the additional benefit of classifying the intervals into smaller sub-windows is small. As a result, we can significantly simplify a mechanism that makes use of access-to-access intervals, by tracking only *whether the interval for a row falls within 16 ms.*

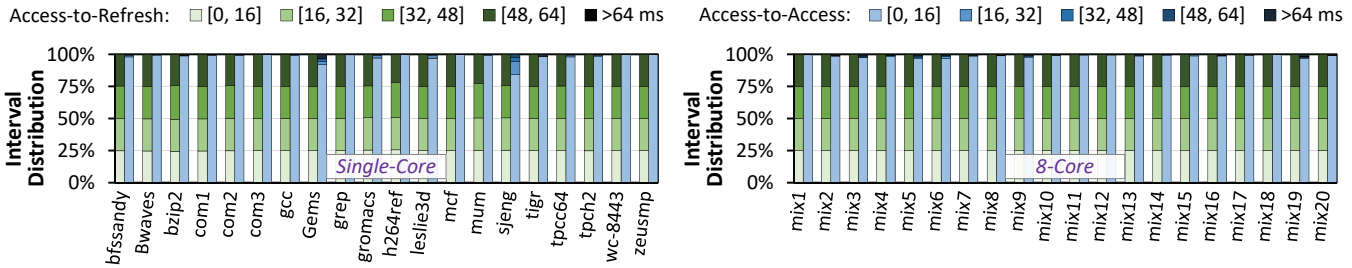


Figure 5: Distribution of access-to-access and access-to-refresh intervals for single-core (left) and 8-core (right) workloads. $[a, b]$ represents all intervals of length t where $a \leq t < b$, with a, b, and t expressed in milliseconds.

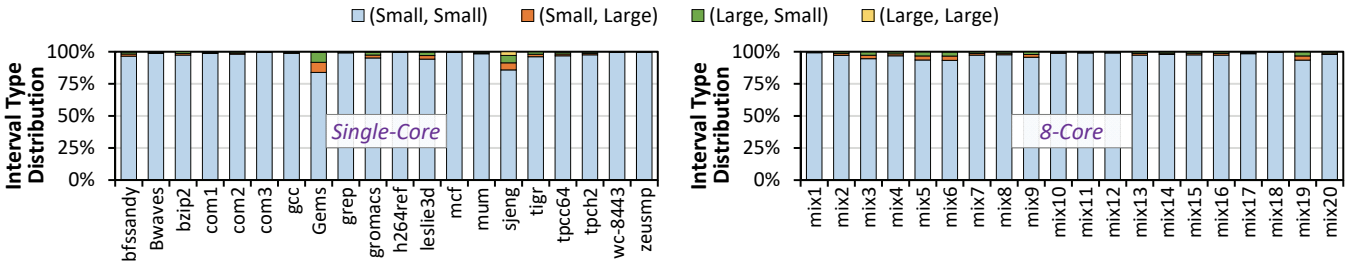


Figure 6: Distribution of *last* and *next* access-to-access interval type pairs for single-core (left) and 8-core (right) workloads.

Figure 7 shows the bitline voltage during the activation of three DRAM cells with three different initial charge levels right before the activation starts. ❶ shows a cell that is activated 1 ms after a *full* restoration (i.e., a cell with a *high* charge level). ❷ shows a cell that is activated 1 ms after a *partial* restoration, which restores the voltage of the cell to only $V_{partial-restoration-level}$. ❸ shows a cell that is activated almost 64 ms after full restoration but before refresh, which means that the cell starts activation at the *minimum* possible charge level. Compared to cell ❸, cell ❶ can use smaller activation (t_{RCD}) and restoration (t_{RAS}) latencies by exploiting the fact that the cell is highly charged. This is because a highly-charged cell reaches (1) the voltage level at which a read or write operation can begin ($V_{ready-to-access}$ in the figure) and (2) the full restoration voltage V_{DD} *more quickly* than a cell that starts activation at the minimum possible charge level. We observe from the figure that for cell ❷, by using partial restoration, we lose a small amount of the t_{RCD} reduction compared to cell ❶, but this allows us to *significantly* reduce t_{RAS}. Thus, there is a trade-off between t_{RCD} (A in the figure) and t_{RAS} (B) reductions.

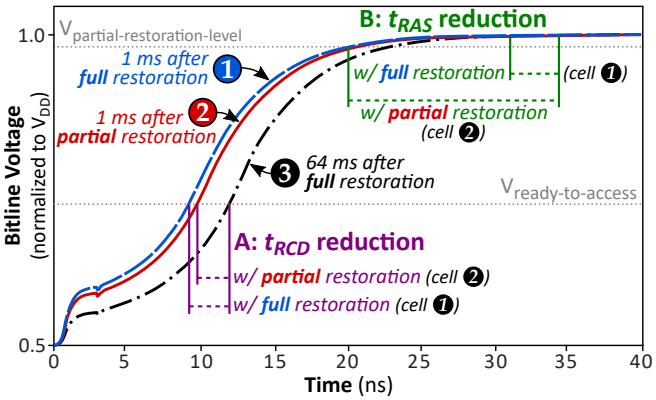


Figure 7: Effect of charge level and restoration level on t_{RCD} (activation) and t_{RAS} (restoration) latency reductions.

We can use a simple heuristic to approximate the maximum benefit of the trade-off between t_{RCD} and t_{RAS} reductions. We quantify $PRBenefit(V_x)$, the benefit of partially restoring a cell to voltage level V_x, as:

$$PRBenefit(V_x) = \frac{\Delta t_{RAS}(Vx)}{\Delta t_{RCD}(V_{full}) - \Delta t_{RCD}(Vx)} \quad (1)$$

In the equation, $\Delta t_{RCD}(V_{full})$ is the t_{RCD} reduction of a fully-restored cell (❶), and $\Delta t_{RCD}(V_x)$ and $\Delta t_{RAS}(V_x)$ are the reductions of t_{RCD} and t_{RAS} for a cell that is partially restored to voltage level V_x.

In order to minimize the overall DRAM access latency, we want to *maximize* $PRBenefit(V_x)$ without sacrificing data integrity. To do this, we must first determine V_{x_min}, the lowest value of voltage level V_x for which we can guarantee data integrity after partial restoration. Recall from Section 3.1 that we want to apply partial restoration only to cells that are reactivated or refreshed within 16 ms after their last precharge. In other words, after the precharge operation, there must be enough charge left in a cell such that at the end of the 16 ms window, the voltage level of the cell must be at least V_{min}, the lowest voltage at which data can be sensed correctly. Because a DRAM cell uses a capacitor to store data, the amount of charge in the cell *decays* over time. Thus, V_{x_min} should be set to ensure that at a charge decay rate of r_{decay}, the cell voltage reaches V_{min} after 16 ms:

$$V_{x\_min} = V_{min} + r_{decay} \times 16\,\text{ms} \quad (2)$$

Using Monte Carlo simulations of our SPICE model, with up to ±5% process variation on each component, we find that $V_{min} = 0.67 V_{DD}$.[4] To model the charge level decay rate r_{decay}, we employ a linear decay model, which is more conservative than the actual documented exponential decay behavior [87]:

$$r_{decay} = \frac{(V_{full} - V_{min})V_{DD}}{64\,\text{ms}} \quad (3)$$

where V_{full} is the voltage level of a cell after full restoration ($0.975 V_{DD}$).

We solve Equation 3 to determine the value of r_{decay}, and then use the values of V_{min} and r_{decay} in Equation 2 to find that $V_{x\_min} = 0.75 V_{DD}$. Now, we can find the maximum value of $PRBenefit(V_x)$ for all values of $V_x > 0.75 V_{DD}$. For each value of V_x, we use SPICE simulations to determine the values of $\Delta t_{RCD}(V_x)$ and $\Delta t_{RAS}(V_x)$, and then use these in Equation 1 to determine the value of $PRBenefit(V_x)$. We find that setting the partial restoration voltage $V_x = 0.85 V_{DD}$ provides the best trade-off between restoration latency reduction and activation latency reduction. We perform a sensitivity study of different partial restoration voltages in Section 6.5.

3.3. Summary and Our Goal

From our motivational study, we conclude that (1) there is a new opportunity to apply partial restoration to soon-to-be-reactivated rows (Section 3.1); (2) soon-to-be-reactivated

[4] This is more conservative than prior work [1], which assumes that $V_{min} = 0.4 V_{DD}$.

rows can be predicted with very high accuracy (Section 3.1); and (3) we can trade off restoration latency reduction with activation latency reduction for highly-charged, soon-to-be-reactivated rows to maximize the overall DRAM access latency reduction (Section 3.2). Based on these conclusions, our goal in this work is to design a mechanism that can effectively exploit partial restoration on both soon-to-be-reactivated and soon-to-be-refreshed rows.

4. CAL Design

Based on our observations from Section 3, we propose *charge-level-aware look-ahead partial restoration* (CAL) for DRAM. The key idea of CAL is to predict which rows in DRAM will be reactivated or refreshed soon, and to perform careful partial restoration on the cells in these rows that maximizes the overall access latency due to the reduction in restoration and activation latencies. CAL consists of two major components. First, CAL uses the last access-to-access interval length (see Section 3.1) of a row to predict whether the row will be activated again soon (e.g., within 16 ms). Second, CAL uses this prediction, along with information about the next scheduled refresh to the row, to decide by how much the restoration latency of the row should be reduced.

Both of these components are enabled by a hardware structure called the *timer table*, which we implement in the memory controller. The timer table tracks the last access-to-access interval of the most-recently-accessed rows. To do so, CAL allocates an entry in the timer table when a row is precharged (i.e., when the current access to the row finishes), and sets a 16 ms timer. The timer value indicates (1) whether the cell is highly charged, and (2) whether the cell is likely to be accessed again in the next 16 ms. When the row is accessed again, CAL looks up the timer table. If an entry exists for the row, and the timer is non-zero, the row is likely to continue having a small access-to-access latency, and CAL (1) reduces both the activation and restoration latencies if the cells in the row are highly charged, or (2) reduces the restoration latency otherwise.

4.1. Timer Table Design

Figure 8 illustrates the basic structure of the timer table. The timer table uses a set-associative structure, indexed by the DRAM row address, to record the last access-to-access interval for the most-recently-accessed DRAM rows. Each timer table entry contains (1) a tag, which stores the DRAM row address; (2) a timer, which records the time elapsed since the row was last precharged; (3) a partially restored (PR) bit, which is set to zero when an entry is initially allocated, and is set to one whenever a row is partially restored; and (4) a valid bit, which is set to zero for an unallocated entry.[5]

Figure 8 shows the four major operations performed by CAL using the timer table:

1. **Insertion:** CAL inserts a new entry into the table whenever the memory controller issues a PRE command to the row (❶ in Figure 8), potentially evicting an existing entry from the table (❷). CAL sets the tag of the new entry to the current row address, sets the valid bit to one, and sets the PR bit to zero.
2. **Timer Initialization:** Whenever a row is precharged (❸), CAL initializes the timer for the row to 15 ms (see Section 4.4).
3. **Timer Update:** Every 1 ms, the table checks each entry's timer (❹). If the timer is greater than zero, the timer table decrements the timer by 1 ms.
4. **Lookup:** Every time the memory controller issues an ACT or WRITE command to a row, CAL queries the table to see if an entry exists for the row (❺). If an entry exists, CAL uses this information to decide if it should reduce the activation and/or restoration latencies for the row (see Section 4.3). CAL then sets the PR bit of the entry to one if it partially restores the row. CAL uses the PR bit to determine if a partially-restored row needs to be fully restored when its access-to-access interval is large (see Section 4.4).

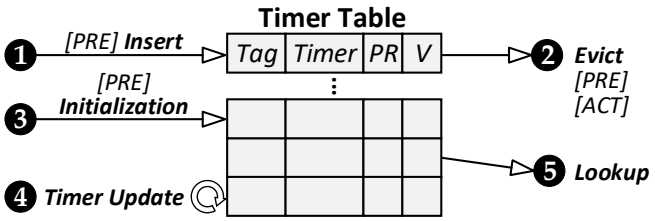


Figure 8: High-level overview of timer table operations.

4.2. Maintaining Row Timers

Whenever a row is precharged, CAL starts tracking the time until the row is activated again. This allows CAL to determine the current access-to-access interval type (i.e., small or large), which it uses to predict the next access-to-access interval type of the row (see Section 3.1). To track the access-to-access interval, CAL queries the timer table to see if there is an entry for the row that is precharged. If an entry does not exist, CAL inserts a new entry. The timer table contains a fixed number of entries, and the new entry insertion may evict an existing entry, using LRU replacement.[6]

CAL then initializes the entry timer to start tracking the access-to-access time. The timer table uses a 1 ms resolution for the timer, where it decrements the timer of each entry every millisecond. As shown by prior work [24], a 1 ms resolution is small enough for meaningful charge level exploitation. We find that this resolution provides a good trade-off between charge level exploitation and table size.

When the timer table decrements the entry timers, it checks to see if any of the timers are now 0. If a timer reaches 0, this means that the row corresponding to the entry has *not* been activated in a 16 ms window (i.e., the row has a large access-to-access interval). If this row was partially restored in the past (i.e., if the PR bit of the timer table entry is set

[5]The timer table can be implemented as either a single shared table or as per-core tables. In our evaluation, we assume that each core has a dedicated timer table. We choose per-core timer tables to avoid the need to tune the optimal size of a shared table based on the core count, and to simplify table organization.

[6]When an existing entry is evicted, if the evicted entry is for a row that was partially restored (i.e., the PR bit is set), CAL must fully restore the charge level of the row as soon as the entry is evicted to maintain data integrity. We describe this in detail in Section 4.4.

to 1), CAL must fully restore the charge level of the row to maintain data integrity (see Section 4.4).

4.3. Using the Timer to Reduce Latencies

CAL uses the current value of the timer at the time that a row is activated or written to, in order to determine if it can reduce the activation and/or restoration latencies for a row. This is because the current timer value indicates both (1) the current charge level of the cells in the row, and (2) the last access-to-access interval type for the row. There are one of three possible categories for the row based on the current value of the timer (recall from Section 4.1 that the timer starts counting down from 15 ms):

1. If the timer is 15 ms, this indicates that less than 1 ms has passed since the row was precharged. As a result, the row is currently at a high charge level, *and* has a short access-to-access interval. For such a row, CAL reduces *both the activation and restoration latencies* (see Table 1 in Section 5 for the reduced latency values), and sets the PR bit to 1.
2. If the timer is less than 15 ms but greater than zero, this indicates that between 1–15 ms have passed since the row was precharged. As a result, the row may no longer be at a high charge level, but the row has a short access-to-access interval. For such a row, CAL reduces *only the restoration latency* (see Table 1 in Section 5), as it may not be able to reduce the activation latency without the possibility of data loss, and sets the PR bit to 1.
3. If the timer is at zero, this indicates that more than 15 ms have passed since the row was precharged. As a result, the row is likely not at a high charge level, and the row has a large access-to-access interval. For such a row, CAL *does not reduce the activation or restoration latencies*, and sets the PR bit to 0. Note that if CAL cannot find an existing entry for a row in the timer table, it assumes that the row falls into this category to ensure reliable operation.

4.4. Handling Large Access-to-Access Intervals

When a row timer reaches zero after decrementing, or when a timer table entry is evicted, CAL checks to see if the row corresponding to the entry was partially restored (i.e., if the PR bit is set to 1). If the row was partially restored, the row may not have enough charge anymore. This is because CAL expected the row to have a short access-to-access interval, but the access-to-access interval of the row either (1) actually is long (in the case where the row entry's timer reaches zero), or (2) can no longer be accurately tracked (in the case where the entry is evicted).

In order to maintain data integrity for such a row, CAL must fully restore the charge level before the charge level drops below the minimum amount that can be sensed correctly. CAL immediately issues an (ACT, PRE) command pair to the row, which preempts other pending commands in the memory controller. The (ACT, PRE) pair uses *full* activation and restoration latencies, ensuring that the row is fully restored. In order to ensure that the two commands can be issued without violating any requirements, CAL *guarantees* that there is at least 1 ms left for the command pair to take place, which is plenty of time to schedule the two commands. To ensure this, CAL initializes the timer to count down for only 15 ms, even though it uses a partial restoration latency that provides enough charge for a 16 ms window.

For the workloads evaluated in this paper, the extra commands performed to fully restore a row that was initially only partially restored have a very small performance overhead of less than 0.1% for a 256-entry timer table per core. Our evaluation in Section 6.1 shows that even when we account for this small overhead, CAL provides significant speedup and outperforms other mechanisms that reduce activation and restoration latencies.

5. Methodology

We use a modified version of Ramulator [40], an open source cycle-accurate DRAM simulator, in conjunction with a front-end CPU and cache model based on Pin [53], to evaluate CAL and other mechanisms. Table 1 provides detailed configuration parameters of the simulated system, including the latency parameters used by CAL and other mechanisms. We obtain timing parameters from the SPICE model described in Section 3.

Components	Parameters
Processor	1–8 cores, 4 GHz, 3-wide issue, 128-entry inst. window, 8 MSHRs/core
Last-Level Cache	2 MB/core, 64 B cache line, 16-way
Memory Controller	64-entry RD/WR request queues, FR-FCFS scheduling [66, 89], closed-row policy
DRAM	DDR4-1600, 800 MHz bus frequency, 1–2 channels, 1 rank, 4 bank groups, 4 banks, 512 K rows/bank, 1 kB row, baseline $t_{RCD}/t_{RAS}/t_{WR}$: 13.75/35/15 ns
ChargeCache [24]	256-entry/core, 8-way associative, LRU replacement policy, 1 ms caching duration, reduced t_{RCD}/t_{RAS}: 9.7/23.8 ns
Restore Truncation [87]	reduced t_{RAS}/t_{WR} for the four 16 ms sub-windows: 35/15, 24.6/10.8, 19.4/8.4, 15.9/6.6 ns
CAL	256 entries/core, 8-way associativity, LRU replacement policy, reduced $t_{RCD}/t_{RAS}/t_{WR}$: - hit within 1 ms: 11.2/16.1/6.8 ns - hit between 1–16 ms: 13.75/19.4/8.4 ns

Table 1: Simulated system configuration.

To measure system energy, we build our energy model based on prior work [6], which captures all major components of our evaluated system, including the CPU core, caches, off-chip interconnect, and DRAM. The model makes use of multiple tools, including McPAT 1.0 [49] for the CPU core, CACTI 6.5 [56] for the caches, Orion 3.0 [28] for the interconnect, and a version of DRAMPower [8] for DRAM, which we modify to accurately capture the power consumption of CAL. We estimate the area overhead of CAL using McPAT 1.0 [49].

Mechanism Parameters. We use an 8-way cache-like set-associative structure for the CAL timer table, with 256 entries per core. To ensure a fair comparison, we evaluate ChargeCache [24] using the same capacity, set associativity, and replacement policy for the highly-charged row access cache (HCRAC).

Workloads. We evaluate twenty single-thread applications from the TPC [83], MediaBench [19], Memory Scheduling Championship [27] BioBench [2], and SPEC CPU2006 [76] benchmark suites.[7] We use PinPoints [63, 72] to identify the representative phases of each application. We classify the applications into two categories: memory intensive (greater than 10 last-level cache misses per kilo-instruction, or MPKI) and memory non-intensive (less than 10 MPKI). To evaluate the effect of CAL on a multicore system, we form twenty 8-application multiprogrammed workloads. To vary the load on the memory system, we generate workloads where 25%, 50%, 75%, and 100% of the applications are memory intensive. For both the single-thread applications and multiprogrammed workloads, each core executes at least one billion instructions. We report the instruction-per-cycle (IPC) speedup for single-thread applications, and weighted speedup [74] as the system performance metric [18] for the multiprogrammed workloads.

6. Evaluation

We compare *CAL* to a conventional baseline memory subsystem (*Base*), two state-of-the-art memory latency reduction mechanisms:

- ChargeCache [24] (*CC*), which reduces t_{RCD} and t_{RAS} for highly-charged rows;
- Restore Truncation [87] (*RT*), which reduces t_{RAS} and t_{WR} for soon-to-be-refreshed rows;

and two mechanisms that *naively* combine techniques to reduce both activation and restoration latency:

- *CCRT*, a simple combined mechanism that (1) partially restores *all* soon-to-be-refreshed rows using Restore Truncation and (2) reduces the activation latency of *only* the highly-charged rows that are *not* partially restored using ChargeCache;
- *GreedyPR*, a partial restoration mechanism that is similar to *CAL* but *greedily* maximizes the restoration latency reduction *without* considering its impact on the potential for reducing the activation latency (see Section 3.2).

Table 1 lists the reduced latencies used by CAL, ChargeCache, and Restore Truncation. In addition, we provide performance comparisons to three *idealized* (i.e., unimplementable) versions of the latency reduction mechanisms, which provide upper bounds to the potential speedup of various mechanisms:

- *IdealCC*, where *all rows* are accessed with the same reduced t_{RCD} and t_{RAS} latencies used by ChargeCache (see Table 1);
- *IdealRT*, where *all rows* are restored with the same reduced t_{RAS} and t_{WR} latencies used by Restore Truncation; and
- *IdealCAL*, where *all rows* are activated and restored with the same reduced t_{RCD}, t_{RAS}, and t_{WR} latencies used by CAL.

6.1. Impact on System Performance

Figure 9 shows the performance improvement over *Base* for our single-core system. Figure 10 shows the weighted speedup improvement over *Base*, for our 8-core system. In both figures, we group the applications and workloads based on memory intensity (see Section 5). We make four observations from Figures 9 and 10.

[7] We do not simulate the entire software stack with system services, such as context switches and VM exits. In the event of a context switch or VM exit, the timer table may need to be flushed, which can incur performance overhead. However, since we employ a timer table with *only* 256 entries, the worst-case overhead of one context switch would be 256 additional ACT and PRE commands to fully restore all of the rows whose entries are flushed. We calculate this overhead to be approximately 0.1% of the overall execution time for a high context switching frequency of 100 times per second [16]. The corresponding energy overhead is around 0.12%.

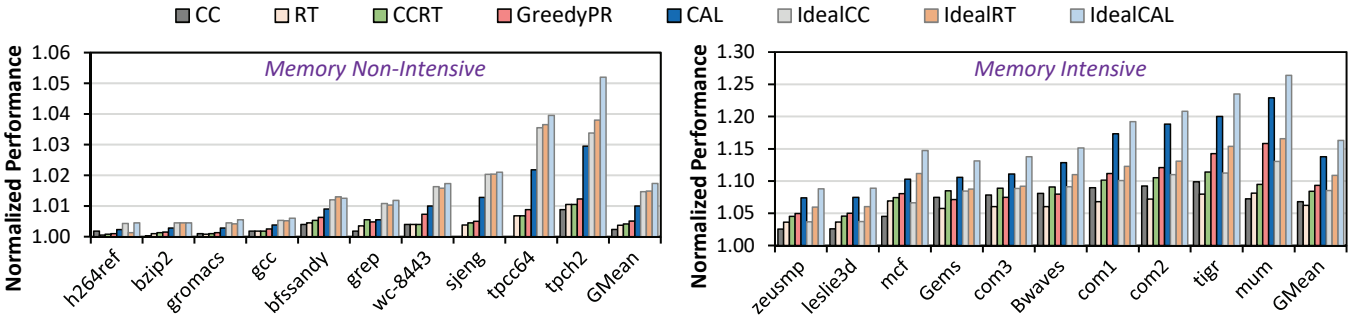


Figure 9: Speedup for our single-core system over *Base*.

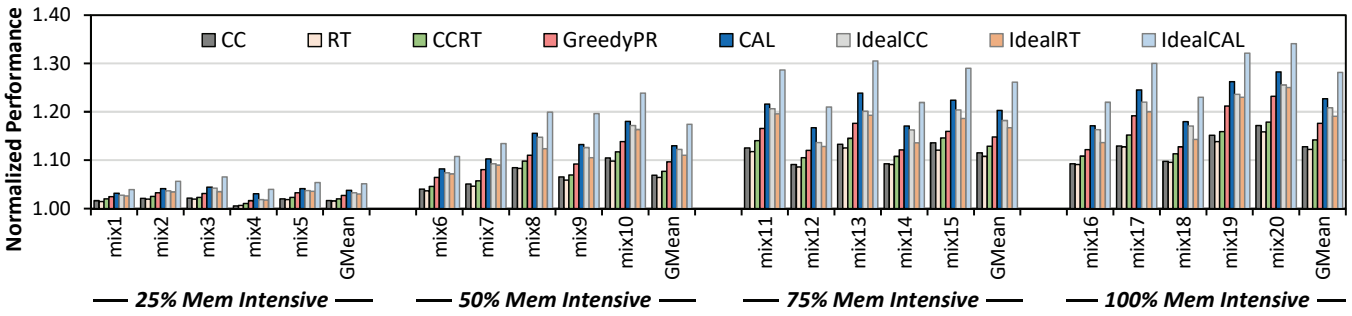


Figure 10: Speedup for our 8-core system over *Base*.

First, we observe that *CAL always* outperforms the other mechanisms (*Base*, *CC*, *RT*, *CCRT*, and *GreedyPR*). For our single-core system, *CAL* provides an average speedup over *Base* of 1.0% (up to 3.0%) for memory non-intensive applications, and 13.8% (up to 22.9%) for memory intensive applications. For our 8-core system, *CAL* improves the weighted speedup over *Base* by an average of 3.8%, 13.0%, 20.3%, and 22.7% for workloads in the 25%, 50%, 75%, and 100% memory intensive categories, respectively. Across all twenty 8-core workloads, the average performance improvement of *CAL* is 14.7%. By carefully balancing the benefits and trade-offs of activation latency reduction and restoration latency reduction, *CAL* is able to outperform *CCRT* and *GreedyPR*, which naively combine activation and restoration latency reduction mechanisms. This demonstrates that the judicious trade-off made by CAL is important to improve system performance.

Second, we observe that finding a good trade-off between restoration latency reduction and activation latency reduction can provide a much higher performance gain than a naive combination of both. As we have already discussed, *CAL* outperforms *GreedyPR* and *CCRT* because of this trade-off. We can also understand the importance of this trade-off on overall memory access latency by comparing *IdealCC*, *IdealRT*, and *IdealCAL*. *IdealCAL always* achieves higher performance than *IdealCC* and *IdealRT* (by 7.3% and 9.1%, respectively, for 8-core, 100% memory intensive workloads), even though *IdealCC* provides a *larger* decrease in t_{RCD} and *IdealRT* provides a *larger* decrease in t_{RAS} and t_{WR} for all memory accesses.

Third, the benefits of *CAL* increase as workload memory intensity increases. With a higher memory intensity, there is a higher rate of *bank conflicts* (not shown due to space constraints). This is particularly true for our multiprogrammed workloads, where multiple concurrently-executing applications often interfere with each other for access to the same bank. When a bank conflict occurs, the open row is precharged, and a new row is activated. It is highly likely due to temporal and spatial locality that the precharged row will be accessed again in the near future. We observe that when the bank conflict rate increases, the number of row activations also increases (not shown). This creates more opportunities for CAL. We run experiments on a 16-core system with *four memory channels*, and find that the average speedup of *CAL* over *Base* increases by 3% (not shown) compared to our baseline 8-core 2-channel configuration, because there are more bank conflicts in the 16-core system.

Fourth, *CAL* approaches the ideal performance improvement of exploiting *both* partial restoration and highly-charged rows (*IdealCAL*), coming within 4.5%, on average, for our 8-core system. *IdealCAL* improves performance over *CAL* because it *assumes*, ideally, that *every* row can be accessed with *minimal* activation and restoration latencies (i.e., every row has an access-to-access interval of less than 1 ms), and that the timer table is *perfect* (i.e., no entries are evicted). We explore the sensitivity of CAL to the timer table size in Section 6.4.

We conclude that CAL significantly reduces the DRAM latency and outperforms state-of-the-art mechanisms for DRAM latency reduction and their combinations.

6.2. Impact on System Energy

Figure 11 shows the overall system energy consumption for *Base*, *CAL*, *CC*, and *RT*, averaged across each workload category. We break down the system energy into the energy consumed by the CPU, caches, off-chip interconnect (*OLink*), and DRAM. DRAM energy is further broken down into activate and precharge (*Act/Pre*), read and write (*Rd/Wr*), refresh (*Ref*), and idle energy.

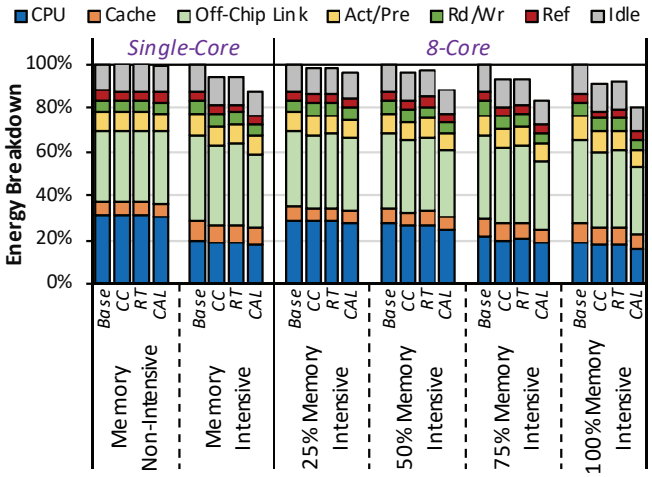


Figure 11: System energy breakdown.

We observe from the figure that *CAL* reduces the average system energy in each workload category over *Base*. For the memory intensive single-thread applications, *CAL* reduces the system energy consumption by an average of 10.1% compared to *Base*. The energy reduction increases with memory intensity: the average reduction is 18.9% for 8-core workloads where 100% of the applications in the workload are memory intensive. *CAL* also achieves lower energy consumption than *CC* and *RT*, by 11.2% and 12.3% respectively, for 8-core 100% memory intensive workloads. *CAL* reduces CPU, cache, and interconnect energy by reducing the execution time required for each application. We observe that the primary source of energy reduction in DRAM is the reduced energy consumed by activation and precharge, since CAL reduces the amount of time spent on activating and restoring a row when it reduces the activation and/or restoration latencies. Overall, we conclude that CAL is effective at reducing system energy consumption, due to its ability to reduce DRAM latency.

6.3. Area and Power Overhead

The area overhead of CAL predominantly consists of the storage required for the timer table (see Section 4.1) in the memory controller. Each entry of the table consists of a row address tag, a 4-bit timer, and two bits for the PR and valid bits. The width of the tag is dependent on the number of DRAM rows in the system. For our 8-core system, which contains 16 GB of DRAM, there are 16 M rows, which requires a tag size of 24 bits. In total, each entry consumes 30 bits.

We assume per-core timer tables, where each table has 256 entries. Therefore, each timer table consumes 960 B. For our 8-core system, the total area consumed by all of the timer tables is 7.7 kB. Using McPAT, we compute the total area of

these tables to be 0.034 mm^2. This overhead is only 0.11% of the area consumed by the 16 MB last-level cache.

The timer table is also the predominant source of power consumption in CAL. Timer table entry insertions, lookups, and updates are the major operations that increase dynamic power consumption, and the table dissipates static power. We analyze the timer table activity for our applications, accounting for all of the major table operations, and find that the table consumes 0.202 mW on average. This is only 0.08% of the average power consumed by the last-level cache. We include this additional power consumption in our system energy evaluations in Section 6.2.

Aside from the timer table, we need to introduce a mechanism that allows CAL to reduce the activation and restoration latencies. One simple way to do so is by introducing new commands to the DDR interface, which perform shortened activation and restoration. These new commands can make use of the undefined encodings that are reserved for future use in the DDR4 specification [26]. Because the unused encodings are already accounted for in the specification, no additional bits are needed for the command bus when we introduce our new commands.

We conclude that CAL incurs small chip area, power consumption, and DRAM interface change overheads.

6.4. Effect of Timer Table Capacity

We perform a sensitivity study on how the number of entries in the timer table affects performance. Figure 12 shows the speedup of *CAL* over *Base* as we vary the size of each core's timer table from 16 to 1024 entries, along with the speedup for a table with infinite capacity. We make two observations from the figure.

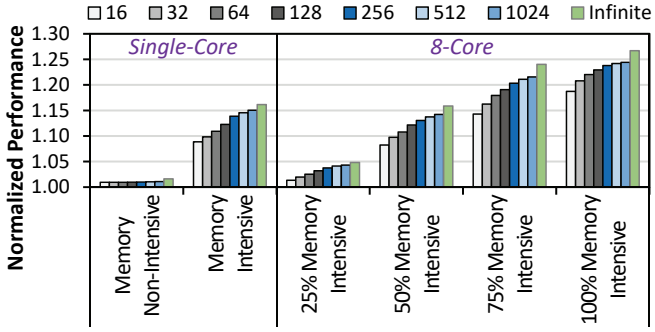


Figure 12: Speedup for different timer table capacities.

First, we observe that as the timer table capacity increases, CAL's performance improvement also increases. A larger number of entries reduces the number of evictions, and has the potential to provide more opportunities for CAL to reduce the activation and restoration latencies for rows that would otherwise be evicted.

Second, CAL's performance improvement tapers off at larger timer table sizes. For example, increasing the table size from 256 to 1024 entries improves the performance by less than 2% for 8-core workloads where 100% of the applications are memory intensive, even though these applications benefit most from CAL. The larger number of entries requires an additional 23 kB of storage overhead, but the higher overhead yields diminishing returns, making the additional storage less cost-effective. Even an infinite timer table improves performance by only 3% on average across all of our applications over a 1024-entry table.

We conclude that a larger number of timer table entries provides higher performance, but that the performance improvements diminish at larger table capacities. We implement 256-entry per-core timer tables to achieve a balance between performance improvement and storage overhead.

6.5. Effect of Different Restoration Levels

The charge restoration level can affect CAL performance because it affects the trade-off between t_{RCD} and t_{RAS}/t_{WR} reductions. Recall from Section 3 that a lower restoration level can achieve larger t_{RAS} and t_{WR} reductions, at the expense of reducing the amount by which CAL can reduce t_{RCD}. We choose $0.85V_{DD}$ as the optimal trade-off point in this paper.

We examine the performance effect of different restoration levels. Figure 13 compares the performance speedup of CAL with restoration levels that are lower (*L-Level1*, *L-Level2*) and higher (*H-Level1*, *H-Level2*) than the optimal restoration level (*Opt-Level*) used in this paper. *L-Level1* and *L-Level2* are $0.83V_{DD}$ and $0.81V_{DD}$, respectively. *H-Level1* and *H-Level2* are $0.87V_{DD}$ and $0.91V_{DD}$, respectively.

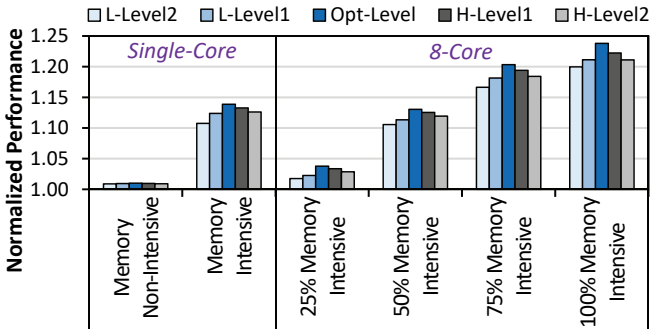


Figure 13: Speedup for different restoration levels.

As shown in the figure, *Opt-Level* outperforms all other choices for all types of workloads. We conclude that the charge restoration level of a DRAM cell plays an important role in balancing the performance benefits from restoration latency reduction and activation latency reduction.

6.6. Effect of DRAM Row Management Policy

All of the results presented so far are on a system that uses a closed-row policy for DRAM. The closed-row policy closes a DRAM row if there are no more pending requests in the memory controller request buffer that target the opened row. Conversely, an open-row policy keeps a row open until a request to a different row is scheduled by the memory controller. The row policy can affect the opportunities available for CAL to reduce the activation and restoration times. Therefore, we evaluate the sensitivity of CAL to three row management policies: (1) closed-row; (2) open-row; and (3) *minimalist open-page* [30], which keeps a row open for a certain period of time (i.e., the minimum delay of back-to-back activations to two different rows within the same bank), unless there is a pending request to the open row.

Figure 14 shows the speedup of *CAL* over *Base* with the three row policies. We observe that *CAL* provides signifi-

cant speedups for all three row policies. The speedup of *CAL* is lower for open-row and minimalist-open page than for closed-row by 6.5% on average for our 8-core workloads as the open-row and minimalist open-page policies reduce the number of row buffer conflicts, which results in fewer activation and restoration operations. However, there are still many opportunities for CAL to reduce the latency even with the open-row and minimalist open-page policies. Overall, we conclude that CAL is effective with different row management policies.

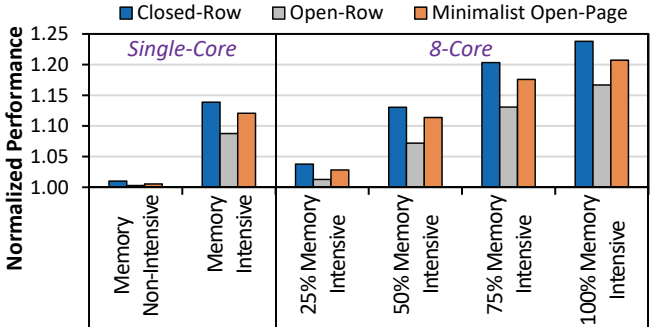


Figure 14: Speedup for different row management policies.

6.7. Effect of Address Mapping Policy

We choose a commonly-used *channel interleaving* address mapping scheme in this paper, which maps the physical address into the tuple (*row_ID, rank_ID, bankgroup_ID, bank_ID, channel_ID, column_ID, cache_offset*), where the *cache_offset* corresponds to the least significant bits of the address. To analyze the performance effect of different address mapping schemes, we implement a permutation-based mapping scheme (PAM) [88], which XORs the lower k bits of the *row_ID* with the original *bank_ID* to produce a new *bank_ID*, where k is the length of *bank_ID* in bits. In doing so, PAM randomizes the *bank_ID* to reduce the number of row buffer conflicts that occur. We evaluate the sensitivity of CAL to the address mapping policy, for both PAM and an *offline address mapping scheme* (OAM) [20]. OAM is an *idealized* address mapping policy that analyzes the entropy of each bit in the address *offline* to determine the address mapping in a way that is supposed to minimize bank conflicts.

Figure 15 shows the speedup of *CAL* over *Base* for the three address mapping policies. We observe that PAM has only a small impact on the speedup that *CAL* achieves, while OAM has a slightly larger impact, where the speedup of *CAL* under OAM is about 3.8% lower than that under channel interleaving, on average for our 8-core, 100% memory-intensive workloads. However, CAL still provides significant performance improvements over OAM mapping, which may not be feasible in many systems where offline address mapping is *not* possible. We conclude that CAL effectively improves performance under different address mapping schemes.

6.8. Effect of High Temperature

So far, we assume a 64 ms refresh interval, which maintains data integrity for cells at normal temperatures. At higher temperatures, the refresh interval reduces [50, 51]. We evaluate the sensitivity of CAL to the refresh interval, for intervals of

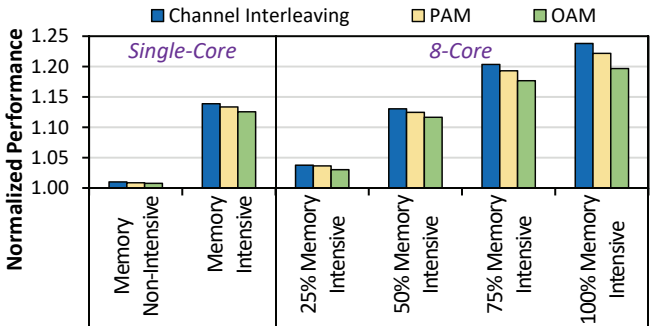


Figure 15: Speedup for different address mapping schemes.

64 ms, 32 ms, and 16 ms. For smaller refresh intervals, we linearly scale the maximum access-to-access interval for which CAL applies partial restoration, to account for the increased rate of charge decay at higher temperatures. For example, for a 32 ms refresh interval, partial restoration is applied only when the access-to-access interval is less than 8 ms.

Figure 16 shows the speedup of *CAL* over *Base* for the three refresh intervals. We observe that the speedup decreases as the refresh interval becomes smaller. This is because when we decrease the maximum time considered to be a *small* access-to-access interval, a row with such an interval is less likely to have a similarly small access-to-access interval in the future, reducing the accuracy of CAL's access-to-access interval type predictor. Even then, CAL is still effective at reduced refresh intervals. We can further improve the performance of CAL by using more sophisticated prediction methods, but we leave this for future work.

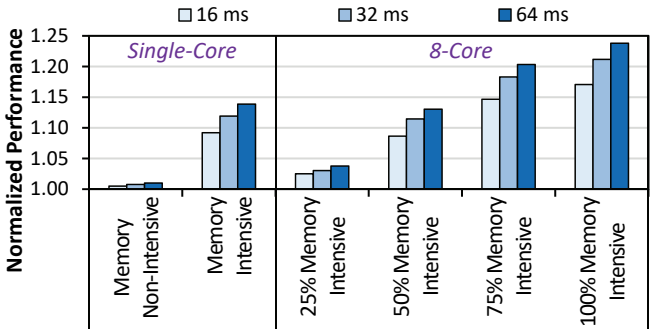


Figure 16: Speedup for different refresh intervals.

Note that for a high refresh rate, the cost of refresh itself becomes more significant [12, 22, 32, 51], and it becomes more promising to extend CAL to support *partial refresh* [15], in which the restoration level, and thus the cost, of refresh can be reduced by shortening the refresh latency for a row that will be activated soon after the refresh. We leave this extension for future work. We conclude that CAL is robust to different refresh intervals.

7. Related Work

To our knowledge, this is the first work to (1) enable partial restoration for rows with small access-to-access intervals, and (2) cooperatively exploit both restoration latency reduction for soon-to-be-reactivated cells and activation latency reduction for cells with a high charge level. We already quan-

titatively demonstrate that CAL outperforms two state-of-the-art mechanisms for partial restoration [87] and activation latency reduction [24], and their combinations, in Section 6. In this section, we describe other related works in (1) high charge level exploitation, (2) characterization-based DRAM optimizations, and (3) DRAM architecture modifications and memory scheduling.

High Charge Level Exploitation. Similar to ChargeCache [24], NUAT [73] employs reduced DRAM timing parameters for *recently-refreshed* DRAM rows. NUAT is orthogonal to, and can be combined with, CAL, as the two mechanisms target different opportunities for latency reduction. Smart Refresh [22] eliminates unnecessary refresh operations for recently-accessed rows, because these rows have cells with high charge levels. Tovletoglou et al. [82] reduce the number of refresh operations by scheduling memory accesses that can fully restore the charge level of a row in place of a refresh operation. In contrast, CAL reduces the latency of the *restoration* operation, and is complementary to the mechanisms of [22] and [82]. We can combine CAL with these mechanisms to achieve further performance improvements.

Characterization-Based DRAM Optimizations. Several studies [7, 10, 11, 15, 21, 31, 33, 34, 44, 45, 62, 64, 68] experimentally investigate various reliability, data retention, and latency characteristics of modern DRAM chips to provide opportunities for lower DRAM latency and energy consumption. CAL achieves latency reduction independent of operating conditions and cell characteristics, and can be combined with mechanisms that are proposed by these characterization-based studies to further improve performance.

DRAM Architecture Modifications and Memory Scheduling. Tiered-Latency DRAM (TL-DRAM) [46] divides each bitline into two segments, and enables faster access to the segment closer to the sense amplifiers. LISA [13] uses isolation transistors to provide wide inter-subarray connections within a bank, which can be used to quickly cache the most frequently accessed rows into a fast subarray, and to reduce the precharge latency. Lee et al. [48] propose a mechanism that allows write requests to memory to activate only part of a row. Other works [12, 23, 35, 43, 47, 52, 69, 71, 75] propose optimized DRAM architectures that lower the DRAM latency or energy consumption. Efficient memory scheduling mechanisms [4, 37, 38, 59, 60, 77, 78, 79, 80, 81, 84] can also reduce DRAM latency by making better use of the existing DRAM resources. The mechanisms in these works are largely orthogonal to CAL, and CAL can be implemented together with these works to further improve DRAM performance and energy efficiency.

8. Conclusion

We propose CAL, a novel, low-overhead mechanism that (1) performs partial restoration on both soon-to-be-refreshed *and* soon-to-be-reactivated rows, and (2) mitigates the negative effect of partial restoration on activation latency reduction for highly-charged DRAM rows. CAL consists of two key components. First, with very high accuracy, CAL predicts *when* a row will be reactivated in the future. Second, CAL uses the future reactivation time, the next refresh time, and the current charge level of a row to reduce *both* the activation and restoration latencies for the row in a way that provides the largest decrease in overall DRAM access latency. We implement CAL fully within the memory controller, *without* any changes to the DRAM module. We find that CAL improves the average performance of an 8-core system by 14.7%, and reduces the average DRAM energy consumption by 11.3%, across a wide range of workloads. We show that CAL outperforms two state-of-the-art mechanisms for DRAM restoration and activation latency reduction, and their combinations. We conclude that CAL is an effective mechanism to significantly reduce the DRAM access latency, which is a major bottleneck in modern computing systems.

Acknowledgments

This research started at ETH Zürich, during Yaohua Wang's stay as a postdoctoral researcher in SAFARI. We thank the anonymous reviewers, SAFARI group members for the feedback and the stimulating research environment, and our industrial partners, especially Alibaba, Google, Huawei, Intel, Microsoft, and VMware, for their generous support. This research was partially supported by the Semiconductor Research Corporation. Lois Orosa was supported by FAPESP fellowship 2016/18929-4.

References

[1] A. Agrawal, A. Ansari, and J. Torrellas, "Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-Chip eDRAM Modules," in *HPCA*, 2014.
[2] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *ISPASS*, 2005.
[3] Arizona State Univ., NIMO Group, "Predictive Technology Model," http://ptm.asu.edu/, 2010.
[4] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[5] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions," in *ISCA*, 2015.
[6] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[7] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
[8] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "Towards Variation-Aware System-Level Power Estimation of DRAMs: An Empirical Approach," in *DAC*, 2013.
[9] K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon Univ., 2017.
[10] K. Chang, A. G. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
[11] K. K. Chang, A. Kashyap, H. Hassan, , S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[12] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[13] K. K. Chang, P. J. Nair, , D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[14] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
[15] A. Das, H. Hassan, and O. Mutlu, "VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency," in *DAC*, 2017.
[16] F. M. David, J. C. Carlyle, and R. H. Campbell., "Context Switch Overheads for Linux on ARM Platforms," in *ExpCS*, 2007.
[17] J. Dean and L. A. Barroso, "The Tail at Scale," *CACM*, 2013.
[18] L. Eeckhout and S. Eyerman, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[19] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II Video: Expediting the Next Generation of Video Systems Research," *Microprocess. Microsyst.*, 2009.

[20] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "DReAM: Dynamic Re-Arrangement of Address Mapping to Improve the Performance of DRAMs," in *MEMSYS*, 2016.

[21] S. Ghose, A. G. Yağlıkçı, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.

[22] 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.

[23] N. D. Gulur, R. Manikantan, M. Mehendale, and R. Govindarajan, "Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities," in *ICS*, 2012.

[24] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, , and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.

[25] JEDEC Solid State Technology Assn., "JESD79-3F: DDR3 SDRAM Standard," 2012.

[26] JEDEC Solid State Technology Assn., "JESD79-4B: DDR4 SDRAM Standard," 2017.

[27] Journal of Instruction-Level Parallelism, "Memory Scheduling Championship," http://www.cs.utah.edu/~rajeev/jwac12/, 2012.

[28] A. B. Kahng, B. Lin, and S. Nath, "Explicit Modeling of Control and Data for Improved NoC Router Estimation," in *DAC*, 2012.

[29] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.

[30] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era," in *MICRO*, 2011.

[31] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.

[32] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.

[33] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.

[34] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.

[35] Y. Kim, "Architectural Techniques to Enhance DRAM Scaling," Ph.D. dissertation, Carnegie Mellon Univ., 2015.

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[37] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[38] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[39] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, 2015.

[41] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Texas at Austin, HPS Research Group, Tech. Rep. TR-HPS-2010-002, 2010.

[42] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon Univ., 2016.

[43] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.

[44] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.

[45] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.

[46] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[47] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.

[48] Y. Lee, H. Kim, S. Hong, and S. Kim, "Partial Row Activation for Low-Power DRAM System," in *HPCA*, 2017.

[49] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009.

[50] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[51] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[52] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[54] S. A. McKee, "Reflections on the Memory Wall," in *CF*, 2004.

[55] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.

[56] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[57] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," in *SUPERFRI*, 2015.

[58] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.

[59] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[60] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[61] S. O, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.

[62] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.

[63] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "PinPoints: Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation," in *MICRO*, 2004.

[64] M. K. Qureshi, D.-H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.

[65] Rambus Inc., "DRAM Power Model," http://www.rambus.com/energy/, 2016.

[66] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.

[67] Y. Ro, H. Cho, E. Lee, D. Jung, Y. H. Son, J. H. Ahn, and J. W. Lee, "SOUP-N-SALAD: Allocation-Oblivious Access Latency Reduction with Asymmetric DRAM Microarchitectures," in *HPCA*, 2017.

[68] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS*, 2009.

[69] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.

[70] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[71] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

[72] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "SimPoint: Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.

[73] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.

[74] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[75] Y. H. Son, S. O, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.

[76] Standard Performance Evaluation Corp., "SPEC CPU® 2006," http://www.spec.org/cpu2006/, 2006.

[77] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.

[78] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *IEEE TPDS*, 2016.

[79] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[80] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[81] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement," in *ASPLOS*, 2010.

[82] K. Tovletoglou, D. S. Nikolopoulos, and G. Karakonstantis, "Relaxing DRAM Refresh Rate Through Access Pattern Scheduling: A Case Study on Stencil-Based Algorithms," in *IOLTS*, 2017.

[83] Transaction Processing Performance Council, "TPC Benchmarks," http://www.tpc.org/.

[84] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, 2016.

[85] M. V. Wilkes, "The Memory Gap and the Future of High Performance Memories," *CAN*, 2001.

[86] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation," in *ISCA*, 2014.

[87] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang, "Restore Truncation for Performance Improvement in Future DRAM Systems," in *HPCA*, 2016.

[88] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," in *MICRO*, 2000.

[89] W. 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.