ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality

Hasan Hassan[†*], Gennady Pekhimenko[†], Nandita Vijaykumar[†]
Vivek Seshadri[†], Donghyuk Lee[†], Oguz Ergin[*], Onur Mutlu[†]

[†]*Carnegie Mellon University* [*]*TOBB University of Economics & Technology*

ABSTRACT

DRAM latency continues to be a critical bottleneck for system performance. In this work, we develop a low-cost mechanism, called ChargeCache*, that enables faster access to recently-accessed rows in DRAM, with no modifications to DRAM chips. Our mechanism is based on the key observation that a recently-accessed row has more charge and thus the following access to the same row can be performed faster. To exploit this observation, we propose to track the addresses of recently-accessed rows in a table in the memory controller. If a later DRAM request hits in that table, the memory controller uses lower timing parameters, leading to reduced DRAM latency. Row addresses are removed from the table after a specified duration to ensure rows that have leaked too much charge are not accessed with lower latency. We evaluate ChargeCache on a wide variety of workloads and show that it provides significant performance and energy benefits for both single-core and multi-core systems.*

1. Introduction

DRAM technology is commonly used as the main memory of modern computer systems. This is because DRAM is at a more favorable point in the trade-off spectrum of density (cost-per-bit) and access latency compared to other technologies like SRAM or flash. However, commodity DRAM devices are heavily optimized to maximize cost-per-bit. In fact, the latency of commodity DRAM has not reduced significantly in the past decade [49, 66].

To mitigate the negative effects of long DRAM access latency, existing systems rely on several major approaches. First, they employ large on-chip caches to exploit the temporal and spatial locality of memory accesses. However, cache capacity is limited by chip area. Even caches as large as tens of megabytes may not be effective for some applications due to very large working sets and memory access characteristics that are not amenable to caching [36, 54, 70, 74, 76]. Second, systems employ aggressive prefetching techniques to preload data from memory before it is needed [5, 13, 90]. However, prefetching is inefficient for many irregular access patterns and it increases the bandwidth requirements and interference in the memory system [21, 23, 24, 44]. Third, systems employ multithreading [86, 95]. However, this approach increases contention in the memory system [17, 22, 59, 64] and does not aid single-thread performance [37, 94]. Fourth, systems exploit memory level parallelism [16, 28, 62, 64, 65]. The DRAM architecture provides various levels of parallelism that can be exploited to simultaneously process multiple memory requests generated by modern processor architectures [46, 65, 71, 96]. While prior works [18, 35, 46, 64, 69] proposed techniques to better utilize the available parallelism, the benefits of these techniques are limited due to 1) address dependencies among instructions in the programs [6, 25, 61], and 2) resource conflicts in the memory subsystem [42, 78]. Unfortunately, *none* of these four approaches *fundamentally* reduce memory latency at its source and the DRAM latency continues to be a performance bottleneck in many systems.

The latency of DRAM is heavily dependent on the design of the DRAM chip architecture, specifically the length of a wire called *bitline.* A DRAM chip consists of millions of DRAM cells. Each cell is composed of a transistor-capacitor pair. To access data from a cell, DRAM uses a component called *sense amplifier.* Each cell is connected to a sense amplifier using a *bitline.* To amortize the large cost of the sense amplifier, hundreds of DRAM cells are connected to the same bitline [49]. Longer bitlines lead to increase in resistance and parasitic capacitance on the path between the DRAM cell and the sense amplifier. As a result, longer bitlines result in higher DRAM access latency [48, 49, 88].

One simple approach to reduce DRAM latency is to use shorter bitlines. In fact, some specialized DRAM chips [29, 56, 80] offer lower latency by using shorter bitlines compared to commodity DRAM chips. Unfortunately, such chips come at a significantly higher cost as they reduce the overall density of the device because they require more sense amplifiers, which occupy significant area [49]. Therefore, such specialized chips are usually not desirable for systems that require high memory capacity [14]. Prior works have proposed several heterogeneous DRAM architectures (e.g., segmented bitlines [49], asymmetric bank organizations [88]) that divide DRAM into two regions: one with low latency, and another with slightly higher latency. Such schemes propose to map frequently accessed data to the low-latency region, thereby achieving lower average memory access latency. However, such schemes require 1) non-negligible changes to the cost-sensitive DRAM design, and 2) mechanisms to identify, map, and migrate frequently-accessed data to low-latency regions. As a result, even though they reduce the latency for some portions of the DRAM chip, they may be difficult to adopt.

Our goal in this work is to design a mechanism to reduce the average DRAM access latency without modifying the existing DRAM chips. We achieve this goal by exploiting two major observations we make in this paper.

Observation 1. We find that, due to DRAM bank conflicts [42, 78], many applications tend to access rows that were recently closed (i.e., closed within a very short time interval). We refer to this form of temporal locality where certain rows are closed and opened again frequently as *Row Level Temporal Locality (RLTL)*. An important outcome of this observation is that a DRAM row remains in a *highly-charged* state when accessed for the second time within a short interval after the prior access. This is because accessing the DRAM row inherently replenishes the charge within the DRAM cells (just like a refresh operation does) [12, 27, 51, 52, 67, 85].

Observation 2. The amount of charge in DRAM cells determines the required latency for a DRAM access. If the amount of charge in the cell is low, the sense amplifier completes its operation in longer time. Therefore, DRAM access latency increases. A DRAM cell loses its charge over time and the charge is replenished by a refresh operation or an access to the row. The access latency of a cell whose charge has been replenished recently can thus be significantly lower than the access latency of a cell that has less charge.

We propose a new mechanism, called *ChargeCache*, that reduces average DRAM access latency by exploiting these two observations. The **key idea** is to track the addresses of recently-accessed (i.e., highly-charged) DRAM rows and serve accesses to such rows with lower latency. Based on our observation that workloads typically exhibit significant Row-Level Temporal Locality (see Section 3), our experimental results on multi-programmed applications show that, on average, ChargeCache can reduce the latency of 67% of all DRAM row activations.

The operation of ChargeCache is straightforward. The memory controller maintains a small table that contains the addresses of a set of recently-accessed DRAM rows. When a row is evicted from the row-buffer, the address of that row, which contains highly-charged cells due to its recent access, is inserted into the table. Before accessing a new row, the memory controller checks the table to determine if the row address is present in the table. If so, the row is accessed with low latency. Otherwise, the row is accessed with normal latency. As cells leak charge over time, ChargeCache requires a mechanism to periodically invalidate entries from the table such that only highly-charged rows remain in it. Section 4 describes the implementation of ChargeCache in detail.

Our evaluations show that ChargeCache significantly improves performance over commodity DRAM for a variety of workloads. For 8-core workloads, ChargeCache improves average workload performance by 8.6% with a hardware cost of only 5.4KB and by 10.6% with a hardware cost of 43KB. As ChargeCache can only *reduce* the latency of certain accesses, it does *not* degrade performance compared to commodity DRAM. Moreover, ChargeCache can be combined with other DRAM architectures that offer low latency (e.g., [12, 15, 42, 48, 49, 68, 81, 82, 88]) to provide even higher performance. Our estimates show that the hardware area overhead of ChargeCache is only 0.24% of a 4MB cache. Our mechanism requires no changes to DRAM chips or the DRAM interface. Section 6 describes our experimental results.

We make the following **contributions**.

- We observe that, due to bank conflicts, many applications exhibit a form of locality where recently-closed DRAM rows are accessed frequently. We refer to this as *Row Level Temporal Locality (RLTL)*.
- We propose an efficient mechanism, ChargeCache, which exploits *RLTL* to reduce the average DRAM access latency by requiring changes *only* to the memory controller. ChargeCache maintains a table of recently-accessed row addresses and lowers the latency of the subsequent accesses that hit in this table within a short time interval.
- We comprehensively evaluate the performance, energy efficiency, and area overhead of ChargeCache. Our experiments show that ChargeCache significantly improves performance and energy efficiency across a wide variety of systems and workloads with negligible hardware overhead.

2. Background on DRAM

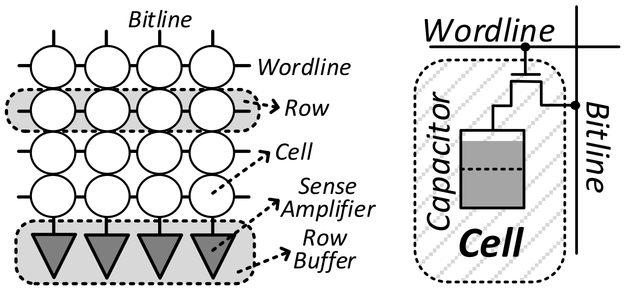
DRAM-based main memory consists of a hierarchy of structures. At the top level, each processor is connected to one or more memory channels. Each *channel* has its own command, address, and data buses. Multiple memory *modules* can be plugged into a single channel. Each module contains many DRAM *chips*. To enable high bandwidth, chips are grouped together to form a *rank*. Each chip (and rank) consists of multiple *banks* that operate mostly independently.

2.1. Accessing Data from a Bank

Logically, each DRAM bank can be viewed as *rows* of DRAM cells connected to a structure called *row buffer*. Each row of DRAM cells contains multiple *columns*. To access data from a specific row and column of a bank, three steps are required. The first step is called *row activation*. This step copies the data from a row of DRAM cells to the row buffer. This process is triggered by issuing an activate (*ACT*) command to the bank with the corresponding row address. Once the row is activated, data can be accessed using the *READ* or *WRITE* command with the corresponding column address. While activated, multiple columns can be accessed from the row. The last step is to prepare the bank for a subsequent access to a different row. This step, called *precharging*, is triggered by the precharge (*PRE*) command.

2.2. Physical Organization of a Bank

Although a DRAM bank can be logically viewed as a single large 2D array of DRAM cells, physically, each DRAM bank consists of multiple subarrays [42]. Each subarray contains multiple rows (typically 512 or 1024 rows) with its own local row buffer. Figure 1a pictorially shows the organization of a subarray. As shown in the figure, the row buffer contains an array of components called *sense amplifiers*. Each cell is connected to the corresponding sense amplifier using a wire called *bitline*. As shown in the figure, multiple cells share the same sense amplifier and bitline. Cells of the same row share a *wordline*, which controls the connection between the cells of that row and the corresponding bitlines. Figure 1b shows the connection between the DRAM cell, the wordline, and the bitline. As shown, each cell consists of a *capacitor*, which stores data in terms of charge, and an *access transistor* that acts as a switch between the capacitor and the bitline. The transistor itself is controlled by the corresponding wordline.



(a) DRAM Subarray **(b)** DRAM Cell

Figure 1: DRAM Subarray and Cell

In this work, we exploit the analog nature of data transfer from the DRAM cell to the sense amplifier to reduce the latency of accessing *recently-accessed* cells. Therefore, in the following section, we describe the operation of a single DRAM cell in more detail.

2.3. DRAM Cell Operation

Figure 2 shows the different sub-steps involved in transferring the data from a DRAM cell to the sense amplifier and their mapping to DRAM commands. Each sub-step takes some time, thereby imposing some constraints on when the memory controller can issue different commands. These constraints are referred to as *timing parameters*. The figure also shows the major timing parameters that govern regular DRAM operation.

In the initial *precharged* state ❶, the bitline is precharged to a voltage level of $V_{dd}/2$. The wordline is lowered (i.e., at 0V) and hence, the bitline is *not* connected to the capacitor. An access to the cell is triggered by the *ACT* command to the corresponding row. This command first raises the wordline (to voltage level V_h), thereby connecting the capacitor to the bitline. Since the capacitor (in this example) is at a higher voltage level than the bitline, charge flows from the capacitor to the bitline, thereby raising the voltage level on the bitline to $V_{dd}/2+\delta$ ❷. This phase is called *charge sharing*. After the charge sharing phase, the sense amplifier is enabled and it detects the deviation on the bitline, and amplifies the deviation. This process, known as *sense amplification*, drives the bitline and the cell to the voltage level corresponding to the original state of the cell (V_{dd} in this example). Once the sense amplification has sufficiently progressed ❸, the memory controller can issue a *READ* or *WRITE* command to access the data from the cell. The time taken by the cell to reach this state ❸ after the *ACT* command is specified by the timing constraint *tRCD*. Once the sense amplification process is complete ❹, the bitline and the cell are both at a voltage level of V_{dd}. In other words, the original charge level of the cell is fully-*restored*. The time taken for the cell to reach this state ❹ after the *ACT* is specified by the timing constraint *tRAS*. In this state, the bitline can be precharged using the *PRE* command to prepare it for accessing a different row. This process first lowers the wordline, thereby disconnecting the cell from the bitline. It then precharges the bitline to a voltage level of $V_{dd}/2$ ❺. The time taken for the precharge operation is specified by the timing constraint *tRP*.

2.4. DRAM Charge Leakage and Refresh

As DRAM cells are not ideal, they leak charge after the precharge operation [51, 52]. This is represented in state ❻ of Figure 2. As described in the previous section, an access to a DRAM cell fully restores the charge on the cell (see states ❹ and ❺). However, if a cell is not accessed for a sufficiently long time, it may lose too much charge that its last cell state may be flipped. To avoid such cases, DRAM cells are periodically refreshed by the memory controller using the refresh (*REF*) command. The interval at which DRAM cells should be refreshed by the controller is referred to as the *refresh interval*.

2.5. Initial Charge vs. Activation Latency

The amount of charge a DRAM cell contains affects the latency of the cell activation process [48, 85]. If the initial charge on the cell is low (as shown in state ❻), then the perturbation caused by the cell on the bitline voltage is also

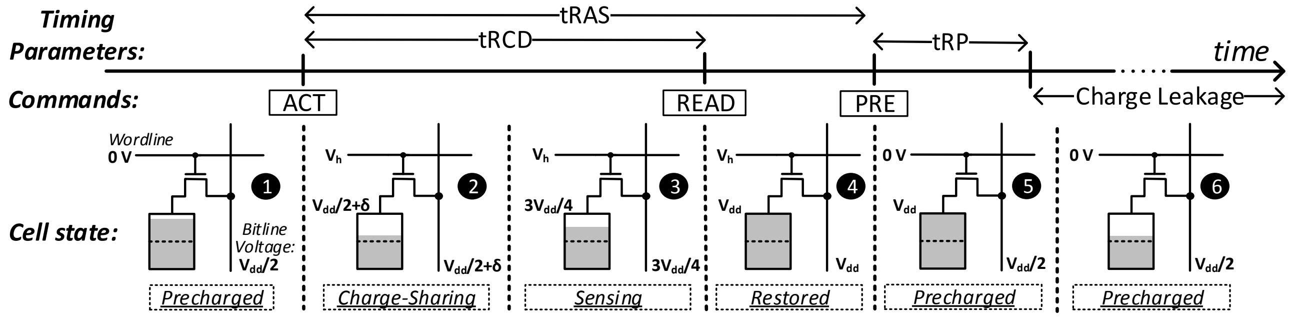


Figure 2: Commands used to read data from DRAM and timing parameters associated with them

low. As a result, the sense amplifier takes longer to reach states ❸ and ❹. This is the case if the cell is accessed *long after* it is refreshed. In fact, the values for *tRCD* and *tRAS* are determined based on the *worst case*, when the cell is accessed *just before* the end of its refresh interval, to guarantee correct operation of *all* cells.

In contrast, if the initial charge on the cell is high (close to full, as in state ❶), then the perturbation caused by the cell on the bitline voltage is also high. This is the case if the cell is accessed *soon after* it is refreshed. As a result, the cell takes shorter time to reach states ❸ and ❹, enabling an opportunity to reduce both *tRCD* and *tRAS*.

3. Motivation

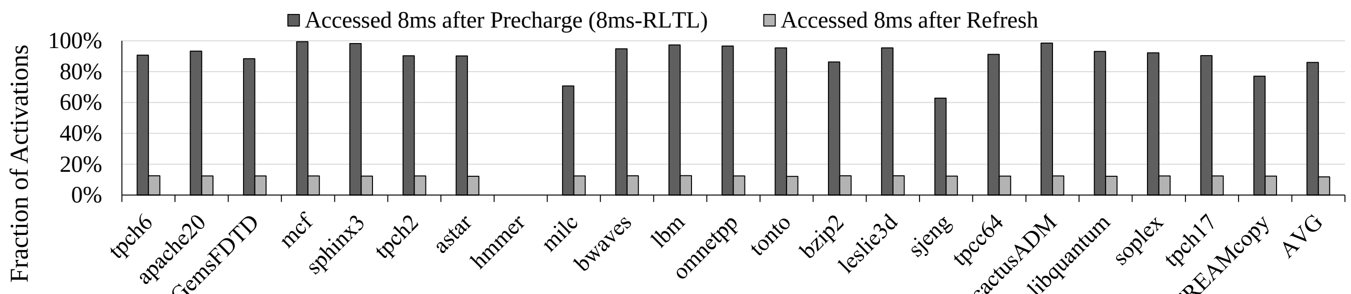
The key takeaway from DRAM operation that we exploit in this work is the fact that *cells closer to the fully-charged state can be accessed with lower activation latency* (i.e., lower *tRCD* and *tRAS*) than standard DRAM specification. A recent work [85] exploits this observation to access rows that were recently recharged via a *refresh* operation with lower latency. Specifically, when a row needs to be activated, the memory controller determines when the row was last refreshed. If the row was refreshed recently (e.g., within $8ms$), the controller uses a lower *tRCD* and *tRAS* for the activation.

However, this refresh-based approach for lowering latency has two shortcomings. First, with the standard refresh mechanism, the refresh schedule used by the memory controller has no correlation with the memory access characteristics of the application. Therefore, depending on the point when the program begins execution, a particular row activation, due to a memory access initiated by the program, may or may not be to a recently-refreshed row. Therefore, a mechanism that reduces latency to recently-refreshed rows cannot provide consistent performance improvement. Second, if we use only the time from the last refresh to identify rows that can be accessed with low latency (i.e., highly-charged rows), we find that only 12% of all memory accesses benefit from low latency (see Figure 3). However, as we show next, a much greater number of rows can actually be accessed with low latency.

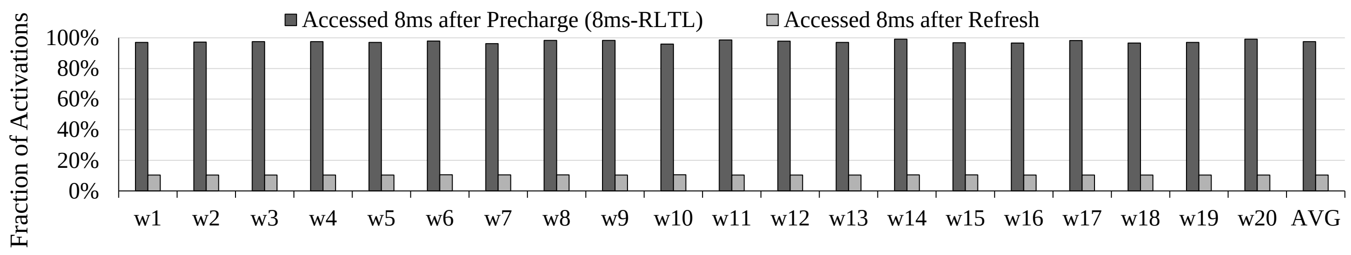
As we described in Section 2.3, an access to a row fully recovers the charge of its cells. Therefore, if a row is *activated twice in a short interval*, the second activate can be served with lower latency as the cells of that row would still be highly charged. We refer to this notion of row activation *locality* as *Row-Level Temporal Locality* (RLTL). We define *t-RLTL* of an application for a given time interval *t* as the fraction of row activations in which the activation occurs within the time interval *t* after a previous precharge to the same row. (Recall that, a row starts leaking charge only after the precharge operation as shown in Section 2.4).

To this end, we would like to understand what fraction of rows exhibit RLTL, and thus can be accessed with low latency after a precharge operation to the row due to program behavior versus what fraction of rows are accessed soon after a refresh to the row and thus can be accessed with low latency due to a recent preceding refresh. Figure 3a compares the fraction of row activations of that happen within $8ms$ after the corresponding row is refreshed to the $8ms$-RLTL of various applications. As shown in the figure, with the exception of *hmmer*[1], the $8ms$-RLTL (86% on average) is significantly higher than the fraction of row activations within $8ms$ after the refresh of the row (12% on average). Figure 3b plots the corresponding values on an 8-core system that executes 20 multiprogrammed workloads, with randomly

[1] *hmmer* effectively uses the on-chip cache hierarchy. Therefore, we do not observe any requests to the main memory.

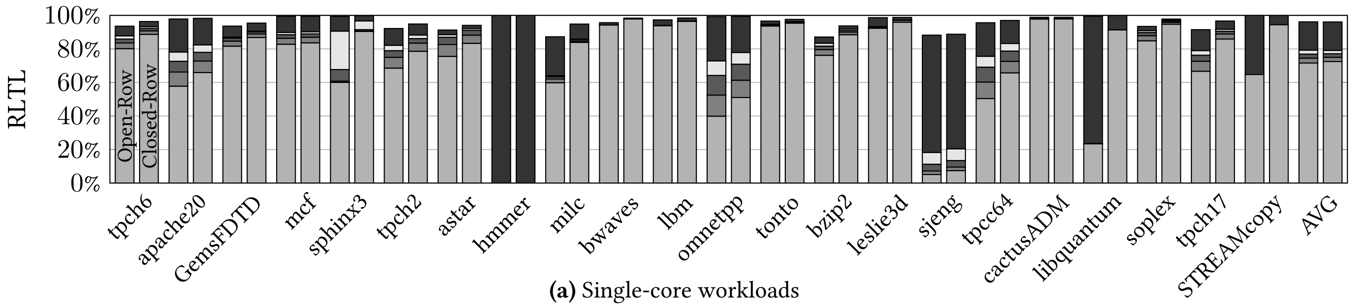


(a) Single-core workloads

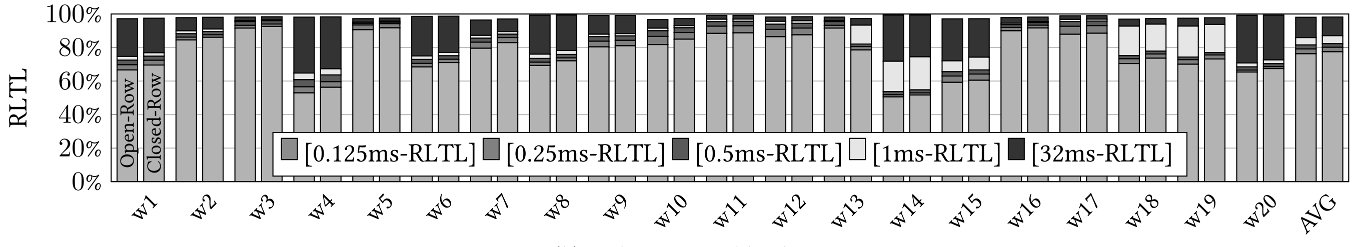


(b) Eight-core workloads

Figure 3: Fraction of row activations that happen 8ms after precharge (8ms-RLTL) or refresh of the row.



(a) Single-core workloads



(b) Eight-core workloads

Figure 4: RLTL for various time intervals (after the precharge of a row).

chosen applications for each workload. As shown, the fraction of row activations within $8ms$ after refresh is almost the same as that of the single-core workloads. This is because the refresh schedule has no correlation with the application access pattern. On the other hand, the $8ms$-RLTL for the 8-core workloads is much higher than that of the single-core workloads. This is because, in multi-core systems, the exacerbated bank-level contention [41, 46, 60, 63, 64, 99] results in row conflicts, which in turn results in rows getting closed and activated within shorter time intervals, leading to a high RLTL.

Figure 4 shows the RLTL for different single-core and 8-core workloads with five different time intervals (from $0.125ms$ to $32ms$) as a stacked bar and two different DRAM row management policies, namely, open-row and closed-row [4, 40]. For each workload, the first bar represents the results for the open-row policy, and the second bar represents the results for the closed-row policy. The open-row policy prioritizes row-buffer hits by keeping the row open until a request to another row is scheduled (bank conflict). In contrast, the closed-row policy proactively closes the active row after servicing all row-hit requests in the request buffer.

For single-core workloads (Figure 4a), regardless of the row-buffer policy, even the average $0.125ms$-RLTL is 66%. In other words, 66% of all the row activations occur within $0.125ms$ after the row was previously precharged. For 8-core workloads (Figure 4b), due to the additional bank conflicts, the average $0.125ms$-RLTL is 77%, significantly higher than that for the single-core workloads. Similar to the single-core workloads, the row-buffer policy does not have a significant impact on the RLTL for the 8-core workloads.

Key Observation and Our Goal. We observe that *many* applications exhibit high row-level temporal locality. In other words, for many applications, a significant fraction of the row activations occur within a small interval after the corresponding rows are precharged. As a result, such row activations can be served with lower activation latency than specified by the DRAM standard. **Our goal** in this work is to exploit this observation to reduce the effective DRAM access latency by tracking recently-accessed DRAM rows in the memory controller and reducing the latency for their next access(es). To this end, we propose an efficient mechanism, ChargeCache, which we describe in the next section.

4. ChargeCache

ChargeCache is based on three observations: 1) rows that are highly-charged can be accessed with lower activation latency, 2) activating a row refreshes the charge on the cells of that row and the cells start leaking only after the following precharge command, and 3) many applications exhibit high row-level temporal locality, i.e., recently-activated rows are more likely to be activated again. Based on these observations, ChargeCache tracks rows that are recently activated, and serves future activates to such rows with lower latency by lowering the DRAM timing parameters for such activations.

4.1. High-level Overview

At a high level, ChargeCache adds a small table (or cache) to the memory controller that tracks the addresses of recently-accessed DRAM rows, i.e., highly-charged rows. ChargeCache performs three operations. First, when a precharge command is issued to a bank, ChargeCache inserts the address of the row that was activated in the corresponding bank to the table (Section 4.2.1). Second, when an activate command is issued, ChargeCache checks if the corresponding row address is present in the table. If the address is not present, then ChargeCache uses the standard DRAM timing parameters to issue subsequent commands to the bank. However, if the address of the activated row is present in the table, ChargeCache employs reduced timing parameters for subsequent commands to that bank (Section 4.2.2). Third, ChargeCache invalidates entries from the table to ensure that rows corresponding to valid entries can indeed be accessed with lower access latency (Section 4.2.3).

We named the mechanism *ChargeCache* as it provides a *cache*-like benefit, i.e., latency reduction based on a locality property (i.e., RLTL), and does so by taking advantage of the *charge* level stored in a recently-activated row. The mechanism could potentially be used with current and emerging DRAM-based memories where the stored charge level leads to different access latencies. We explain how ChargeCache can be applied to other DRAM standards in Section 7.2.

In the following section, we describe the different components and operation of ChargeCache in more detail. In Section 4.3, we present the results of our SPICE simulation that analyzes the potential latency reduction that can be obtained using ChargeCache.

4.2. Detailed Design

ChargeCache adds two main components to the memory controller. Figure 5 highlights these components. The first component is a tag-only cache that stores the addresses of a subset of highly-charged DRAM rows. We call this cache the *Highly-Charged Row Address Cache* (HCRAC). We organize HCRAC as a set-associative structure similar to the processor caches. The second component is a set of two counters that ChargeCache uses to invalidate entries from the HCRAC that can potentially point to rows that are no longer highly-charged. As described in the previous section, there are three specific operations with respect to ChargeCache: 1) insert, 2) lookup, and 3) invalidate. We now describe these operations in more detail.

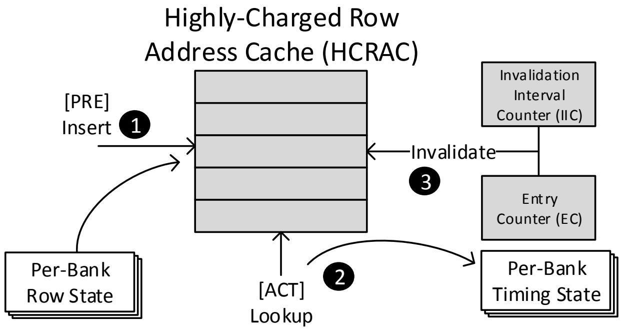


Figure 5: Components of the ChargeCache Mechanism

4.2.1. Inserting Rows into HCRAC. When a *PRE* command is issued to a bank, ChargeCache inserts the address of the row that was activated in the corresponding bank into the HCRAC ❶. Although the *PRE* command itself is associated only with the bank address, the memory controller has to maintain the address of the row that is activated in each bank (if any row is activated) so that it can issue appropriate commands when a bank receives a memory request. ChargeCache obtains the necessary row address information directly from the memory controller. Some DRAM interfaces [57] allow the memory controller to precharge all banks with a single command. In such cases, ChargeCache inserts the addresses of the activated rows across *all* the banks into the HCRAC.

Just like any other cache, HCRAC contains a limited number of entries. As a result, when a new row address is inserted, ChargeCache may have to evict an already valid entry from the HCRAC. While such evictions can potentially result in wasted opportunity to reduce DRAM latency for some row activations, our evaluations show that even with a small HCRAC (e.g., 128-entries), ChargeCache can provide significant performance improvement (see Section 6).

4.2.2. Employing Lowered DRAM Timing Constraints. To employ lower latency for highly-charged rows, the memory controller maintains two sets of timing constraints, one for regular DRAM rows, and another for highly-charged DRAM rows. While we evaluate the potential reduction in timing constraints that can be enabled by ChargeCache, we expect the lowered timing constraints for highly-charged rows to be part of the standard DRAM specification.

On each *ACT* command, ChargeCache looks up the corresponding row address in the HCRAC ❷. Upon a hit, ChargeCache employs lower *tRCD* and *tRAS* for the subsequent *READ*/*WRITE* and *PRE* operations, respectively. Upon a miss, ChargeCache employs the default timing constraints for the subsequent commands.

4.2.3. Invalidating Stale Rows from HCRAC. Unlike conventional caches, where an entry can stay valid as long as it is not explicitly evicted, entries in HCRAC have to be invalidated after a specific time interval. This is because as DRAM cells continuously leak charge, a highly-charged row will no longer be highly-charged after a specific time interval.

One simple way to invalidate stale entries would be to use a clock to track time and associate each entry with an expiration time. Upon a hit in the HCRAC, ChargeCache can check if the entry is past the expiration time to determine which set of timing parameters to use for the corresponding row. However, this scheme increases the storage cost and complexity of implementing ChargeCache.

We propose a simpler, periodic invalidation scheme that is similar to how the memory controller issues refresh commands [52]. Our mechanism uses two counters, namely, the *Invalidation Interval Counter* (IIC) and the *Entry Counter* (EC). We assume that the HCRAC contains k entries and the number of processor cycles for which a DRAM row stays highly-charged after a precharge is C. IIC cyclically counts up to C/k, and EC cyclically counts up to k. Initially, both IIC and EC are initialized to zero. IIC is incremented every cycle. Whenever IIC reaches C/k, 1) the entry in the HCRAC pointed to by EC is invalidated, 2) EC is incremented, and 3) IIC is cleared. Whenever EC reaches k, it is cleared. This mechanism invalidates every entry in the HCRAC once every C processor cycles. Therefore, it ensures that any valid entry in the HCRAC indeed corresponds to a highly-charged row. While our mechanism can prematurely invalidate an entry, our evaluations show that the loss in performance benefit due to such premature evictions is negligible.

4.3. Reduction in DRAM Timing Parameters

We evaluate the potential reduction in *tRCD* and *tRAS* for ChargeCache using circuit-level SPICE simulations. We implement the DRAM sense amplifier circuit using $55nm$ DDR3 model parameters [77] and PTM low-power transistor models [72, 100]. Figure 6 plots the variation in bitline voltage level during cell activation for different initial charge amounts of the cell.

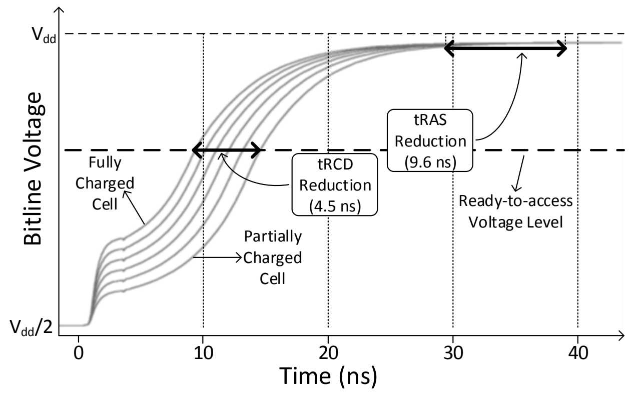


Figure 6: Effect of initial cell charge on bitline voltage.

Depending on the initial charge (i.e., voltage level) of the cell, the bitline voltage increases at different speeds. When the cell is *fully-charged*, the sense amplifier is able to drive the bitline voltage to the *ready-to-access voltage level* in only $10ns$. However, a partially-charged cell (i.e., one that has not been accessed for $64ms$) brings the bitline voltage up slower. Specifically, the bitline connected to such a partially-charged cell reaches the ready-to-access voltage level in $14.5ns$. Since DRAM timing parameters are dictated by this worst-case partially-charged state right before the refresh interval, we can achieve $4.5ns$ reduction in *tRCD* for a fully-charged cell. Similarly, the charge of the cell capacitor is restored at different times depending on the initial voltage of the cell. For a fully-charged cell, this results in $9.6ns$ reduction in *tRAS*.

In practice, we expect the DRAM manufacturers to identify the lowered timing constraints for different caching durations. Today, DRAM manufacturers test each DRAM chip to determine if it meets the timing specifications. Similarly, we expect the manufacturers would also test each chip to determine if it meets the ChargeCache timing constraints.

Caching duration (i.e., how long a row address stays in ChargeCache) provides a trade-off between ChargeCache hit-rate and the DRAM access latency reduction. A longer *caching duration* leads to a longer *Invalidation Interval*. Thus, a row address stays a longer time in ChargeCache. This creates an opportunity to increase ChargeCache hit-rate. On the other hand, with a longer *caching duration*, the amount of charge that remains in DRAM cells at the end of the duration decreases. Consequently, the room for reducing *tRCD* and *tRAS* shrinks. As Figure 3 indicates a very high *RLTL* even with a $0.125ms$ duration, we believe sacrificing ChargeCache hit-rate for DRAM access latency is a reasonable design choice. Therefore, we assume a $1ms$ *caching duration* and a corresponding 4/8 cycle reduction in *tRCD*/*tRAS* (determined using SPICE simulations) for a DRAM bus clocked at 800 MHz frequency. To support our design decision, we also analyze the effect of various *caching durations* in Section 6.4.2.

5. Methodology

To evaluate the performance of ChargeCache, we use a cycle-accurate DRAM simulator, Ramulator [1, 43], in CPU-trace-driven mode. CPU traces are collected using a Pintool [55]. Table 1 lists the configuration of the evaluated systems. We implement the HCRAC similarly to a 2-way associative cache that uses the LRU policy.

Table 1: Simulated system configuration

Processor	1-8 cores, 4GHz clock frequency, 3-wide issue, 8 MSHRs/core, 128-entry instruction window
Last-level Cache	64B cache-line, 16-way associative, 4MB cache size
Memory Controller	64-entry read/write request queues, FR-FCFS scheduling policy [79, 101], open/closed row policy [40, 41] for single/multi core
DRAM	DDR3-1600 [57], 800MHz bus frequency, 1/2 channels, 1 rank/channel, 8 banks/rank, 64K rows/bank, 8KB row-buffer size, tRCD/tRAS 11/28 cycles
ChargeCache	128-entry (672 bytes)/core, 2-way associativity, LRU replacement policy, $1ms$ *caching duration*, tRCD/tRAS reduction 4/8 cycles

For area, power, and energy measurements, we modify McPAT [50] to implement ChargeCache using $22nm$ process technology. We also use DRAMPower [10] to obtain power/energy results of the off-chip main memory subsystem. We feed DRAMPower with DRAM command traces obtained from our simulations using Ramulator.

We run 22 workloads from SPEC CPU2006 [89], TPC [3] and STREAM [2] benchmark suites. We use SimPoint [31] to obtain traces from representative phases of each application. For single-core evaluations, unless stated otherwise, we run each workload for 1 billion instructions. For multi-core evaluations, we use 20 multi-programmed workloads by assigning a randomly-chosen application to each core. We evaluate each configuration with its best performing row-buffer management policy. Specifically, we use the open-row policy for single-core and closed-row policy for multi-core configurations. We simulate the benchmarks until each core executes at least 1 billion instructions. For both single and multi-core configurations, we first warm up the caches and ChargeCache by fast-forwarding 200 million cycles.

We measure performance improvement for single-core workloads using the Intructions per Cycle (IPC) metric. We measure multi-core performance using the weighted speedup [87] metric. Prior work has shown that weighted speedup is a measure of system throughput [26].

6. Evaluation

We experimentally evaluate the following mechanisms: 1) ChargeCache, 2) NUAT [85], which accesses *only* rows that are *recently-refreshed* at lower latency than the DRAM standard, 3) ChargeCache + NUAT, which is a combination of ChargeCache and NUAT [85] mechanisms, and 4) Low-Latency DRAM (LL-DRAM) [29], which is an idealized comparison point where we assume *all rows* in DRAM can be accessed with low latency, compared to our baseline DDR3-1600 [57] memory, at any time, regardless of when they are accessed or refreshed.

We primarily use a 128-entry ChargeCache, which provides an effective trade-off between performance and hardware overhead. We analyze sensitivity to ChargeCache capacity in Section 6.4.1. We evaluate LL-DRAM to show the upper limit of performance improvement that can be achieved by reducing *tRCD* and *tRAS*. LL-DRAM uses, for all DRAM accesses, the same reduced values for these timing parameters as we use for ChargeCache hits. In other words, LL-DRAM is the same as ChargeCache with a 100% hit rate.

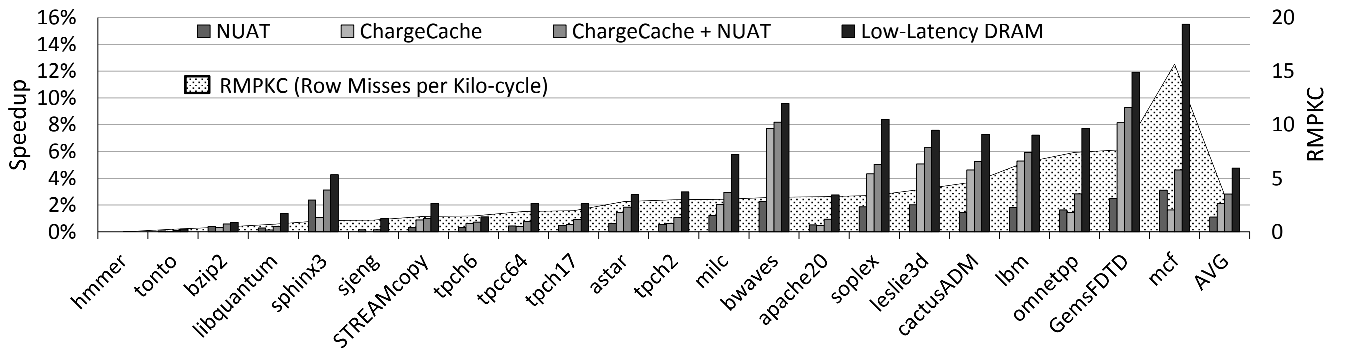
We compare the performance of our mechanism against the most closely related previous work, NUAT [85], and also show the benefit of using both ChargeCache and NUAT together. The key idea of NUAT is to access *recently-refreshed* rows at low latency, because these rows are already highly-charged. Thus, NUAT does not usually access rows that are recently-*accessed* at low latency, and hence it does not exploit existing RLTL (Row-Level Temporal Locality) present in many applications. As we show in Section 3, the fraction of activations that are to rows that are recently-accessed by the application is much higher than the fraction of activations that are to rows that are recently-refreshed. In other words, many workloads have very high RLTL, which is not exploited by NUAT. As a result, we expect ChargeCache to significantly outperform NUAT since it can reduce DRAM latency for a much greater fraction of DRAM accesses than NUAT. To quantitatively prove our expectation that ChargeCache should widely outperform NUAT, we implement NUAT in Ramulator using the default 5PB configuration used in [85]. Note that NUAT bins the rows into different latency categories based on how recently they were refreshed. For instance, NUAT accesses rows that were refreshed between $0 - 6ms$ ago with different *tRCD* and *tRAS* parameters than rows that were refreshed between $6 - 16ms$ ago. We determined the different timing parameters of different NUAT bins using SPICE simulations. Although ChargeCache can implement a similar approach to NUAT by using multiple *caching durations*, our *RLTL* results motivate a single *caching duration* since a row is typically accessed within $1ms$ (as shown in Section 3). A row that hits in ChargeCache is always accessed with reduced timings (Section 4.3).

6.1. Impact on Performance

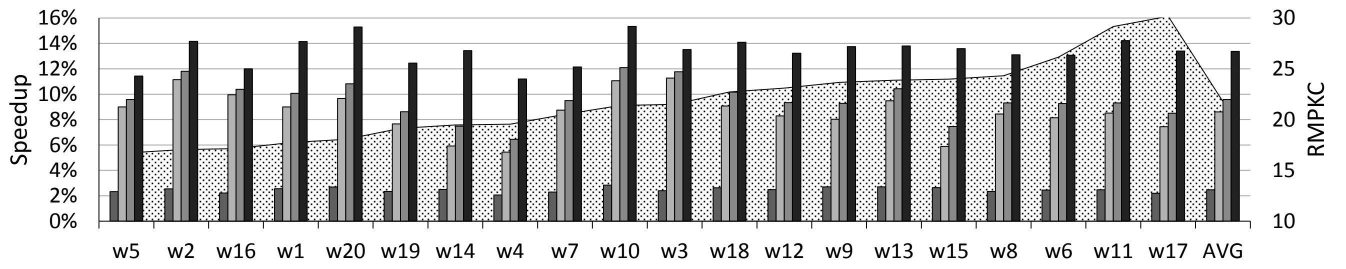
Figure 7 shows the performance of single-core and eight-core workloads. The figure also includes the number of row misses per kilo-cycles (RMPKC) to show row activation intensity, which provides insight into the RLTL of the workload.

Single-core. Figure 7a shows the performance improvement over the baseline system for single-core workloads. These workloads are sorted in ascending order of RMPKC. ChargeCache achieves up to 9.3% (an average of 2.1%) speedup. Our mechanism outperforms NUAT and achieves a speedup close to LL-DRAM with a few exceptions. Applications that have a wide gap in performance between ChargeCache and LL-DRAM (such as *mcf, omnetpp*) access a large number of DRAM rows and exhibit high row-reuse distance [38]. A high row-reuse distance indicates that there is large number of accesses to other rows between two accesses to the same row. Due to this reason, ChargeCache *cannot* retain the addresses of highly-charged rows until the next access to that row. Increasing the number of ChargeCache entries or employing cache management policies aware of reuse distance or thrashing [19, 74, 84] may improve the performance of ChargeCache for such applications. We leave the evaluation of these methods for future work. We conclude that ChargeCache significantly reduces execution time for most high-RMPKC workloads and outperforms NUAT for all but few workloads.

Eight-core. Figure 7b shows the speedup on eight-core multiprogrammed workloads. On average, ChargeCache and NUAT improve performance by 8.6% and 2.5%, respectively. Employing ChargeCache in combination with NUAT achieves a 9.6% speedup, which is only 3.8% less than the improvement obtained using LL-DRAM. Although the multiprogrammed workloads are composed of the *same* applications as in single-core evaluations, we observe much higher performance improvements among eight-core workloads. The reason is twofold. First, since multiple cores share a limited capacity LLC, simultaneously running applications compete for the LLC. Thus, individual applications access main memory more often, which leads to higher RMPKC. This makes the workload performance more sensitive to main memory latency [8, 33, 42]. Second, the memory controllers receive memory requests from multiple simultaneously-running applications to a limited number of memory banks. Such requests are likely to target different rows since they use separate memory regions and these regions map to separate rows. Therefore, applications running concurrently exacerbate the



(a) Single-core workloads



(b) Eight-core workloads

Figure 7: Speedup with ChargeCache, NUAT and Low-Latency DRAM for single-core and eight-core workloads.

bank-conflict rate and increase the number of row activations that hit in ChargeCache.

Overall, ChargeCache improves performance by up to 8.1% (11.3%) and 2.1% (8.6%) on average for single-core (eight-core) workloads. It outperforms NUAT for most of the applications and using NUAT in combination with ChargeCache improves the performance slightly further.

6.2. Impact on DRAM Energy

ChargeCache incurs negligible area and power overheads (Section 6.3). Because it reduces execution time with negligible overhead, it leads to significant energy savings. Even though ChargeCache increases the energy efficiency of the entire system, we quantitatively evaluate the energy savings only for the DRAM subsystem since Ramulator [43] does not have a detailed CPU model. Figure 8 shows the average and maximum DRAM energy savings for single-core and eight-core workloads. ChargeCache reduces energy consumption by up to 6.9% (14.1%) and on average 1.8% (7.9%) for single-core (eight-core) workloads. We conclude that ChargeCache is effective at improving the energy efficiency of the DRAM subsystem, as well as the entire system.

6.3. Area and Power Consumption Overhead

HCRAC (Highly-Charged Row Address Cache) is the most area/power demanding component of ChargeCache. The overhead of *EC* and *IIC* is negligible since they are just two simple counters. As we replicate ChargeCache on a per-core and per-memory channel basis, the total area and power overhead ChargeCache introduces depends on the number of

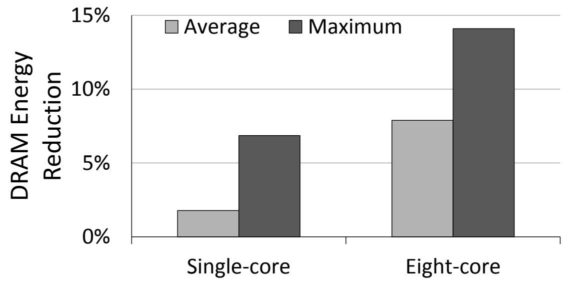


Figure 8: DRAM energy reduction of ChargeCache.

cores and memory channels.[2] The total storage requirement is given by Equation 1, where *C* are *MC* are the number of cores and memory channels, respectively. *LRUbits* depends on ChargeCache associativity. *EntrySize* is calculated using Equation 2, where *R*, *B*, and *Ro* are the number of ranks, banks, and rows in DRAM, respectively.

$$Storage_{bits} = C * MC * Entries * (EntrySize_{bits} + LRU_{bits}) \quad (1)$$

$$EntrySize_{bits} = log_2(R) + log_2(B) + log_2(Ro) + 1 \quad (2)$$

Area. Our eight-core configuration has two memory channels. This introduces a total of 5376 bytes in storage requirement for a 128-entry ChargeCache, corresponding to an area of $0.022\ \text{mm}^2$. This overhead is only 0.24% of the 4MB LLC.

Power Consumption. ChargeCache is accessed on every *activate* and *precharge* command issued by the memory controller. On an *activate* command, ChargeCache is searched for the corresponding row address. On a *precharge* command,

[2]Note that sharing ChargeCache across cores can result in even lower overheads. We leave the exploration of such designs to future work.

the address of the precharged row is inserted into ChargeCache. ChargeCache entries are periodically invalidated to ensure they do not exceed a specified *caching duration*. These three operations increase dynamic power consumption in the memory controller, and the ChargeCache storage increases static power consumption. Our analysis indicates that ChargeCache consumes 0.149 mW on average. This is only 0.23% of the average power consumption of the entire 4MB LLC. Note that we include the effect of this additional power consumption in our DRAM energy evaluations in Section 6.2. We conclude that ChargeCache incurs almost negligible chip area and power consumption overheads.

6.4. Sensitivity Studies

ChargeCache performance depends mainly on two variables: *HCRAC capacity* and *caching duration*. We observed that associativity has a negligible effect on ChargeCache performance. In our experiments, increasing the associativity of HCRAC from two to full-associativity improved the hit rate by only 2%. We analyze the hit rate and performance impact of *capacity* and *caching duration* in more detail.

6.4.1. ChargeCache Capacity. Figure 9 shows the average hit rate versus capacity of ChargeCache for single-core and eight-core systems. The horizontal dashed lines indicate the maximum hit rate achievable with an unlimited-capacity ChargeCache. We observe that 128 entries is a sweet spot between hit rate and storage overhead. Such a configuration yields 38% and 66% hit rate for single-core and eight-core systems, respectively. The storage requirement for a 128-entry ChargeCache is only 672 bytes per core assuming our two-channel main memory (see Section 6.3).

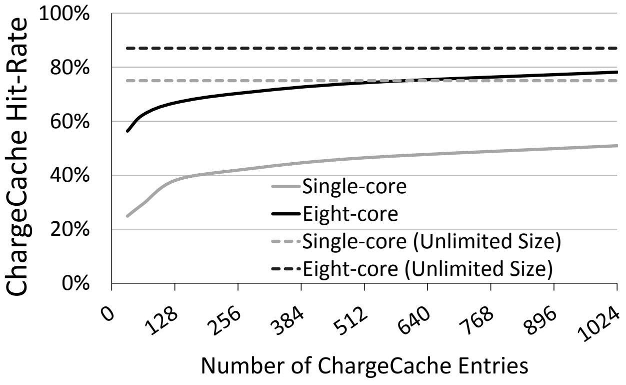


Figure 9: ChargeCache hit rate for single-core and eight-core systems at 1ms *caching duration*.

Figure 10 shows the speedup with various ChargeCache capacities. Larger capacities provide higher performance thanks to the higher ChargeCache hit rate. However, they also incur higher hardware overhead. For a 128-entry capacity (672 bytes per-core), ChargeCache provides 8.8% performance improvement, and for a 1024-entry capacity (5376 bytes per-core) it provides 10.6% performance improvement. We conclude that ChargeCache is effective at various sizes, but its benefits start to diminish at higher capacities.

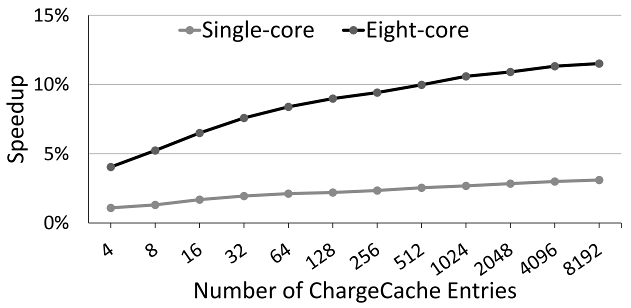


Figure 10: Speedup versus ChargeCache capacity.

6.4.2. Caching Duration. Increasing the *caching duration* may improve the hit rate by decreasing the number of invalidated entries. We evaluate several *caching durations* to determine the duration value that provides favorable performance. For each *caching duration*, Table 2 shows the *tRCD* and *tRAS* values which we obtain from our circuit-level SPICE simulations. We also provide the default timing parameters used as a baseline in the first row of the table.

Table 2: tRCD and tRAS for different *caching durations* (determined via SPICE simulations)

Caching Duration (ms)	tRCD (ns)	tRAS (ns)
N/A (Baseline)	13.75	35
1	8	22
4	9	24
16	11	28

Figure 11 shows how ChargeCache speedup and ChargeCache hit rate vary with different *caching durations*. We make two observations. First, increasing the *caching duration* negatively affects the performance improvement of ChargeCache. This is because a longer caching duration leads to lower reductions in *tRCD* and *tRAS* (as Table 2 shows), thereby reducing the benefit of a ChargeCache hit. Second, ChargeCache hit rate increases slightly (by about 2%) for the single-core system but remains almost constant for the eight-core system when *caching duration* increases. The latter is due to the large number of bank conflicts in the 8-core system, as we explained in Section 3. With many bank conflicts, the aggregate number of precharge commands is high and ChargeCache evicts entries very frequently even with a $1ms$ *caching duration*. Thus, a longer *caching duration* does not have much effect on hit rate.

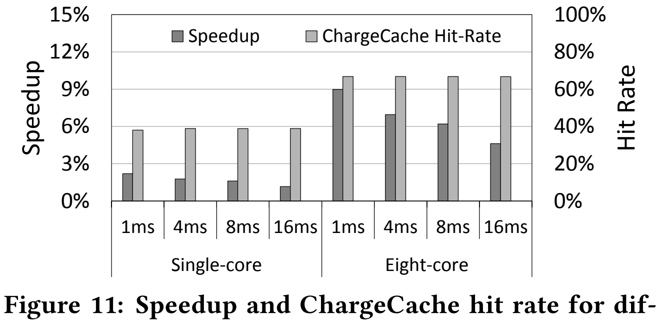


Figure 11: Speedup and ChargeCache hit rate for different *caching durations*

We conclude that, with a longer *caching duration*, the improvement in ChargeCache hit rate does not make up for the loss in the reduction of the timing parameters. We conclude that ChargeCache is effective for various *caching durations*, yet the empirically best *caching duration* is $1ms$, which leads to the highest performance improvement.

7. Discussion

7.1. Temperature Independence

Charge leakage rate of DRAM cells approximately doubles for every 10°C increase in the temperature [39, 48, 51, 58, 75]. This observation can be exploited to lower the DRAM latency when operating at low temperatures. A previous study, Adaptive Latency DRAM (AL-DRAM) [48], proposes a mechanism to improve system performance by reducing the DRAM timing parameters at low operating temperature. It is based on the premise that DRAM typically does not operate at temperatures close to the worst-case temperature (85° C) even when it is heavily accessed. However, new 3D-stacked DRAM technologies such as HMC [32], HBM [34], WideIO [20] may operate at significantly higher temperatures due to tight integration of multiple stack layers [7, 47, 73]. Therefore, *dynamic latency scaling* techniques such as AL-DRAM may be less useful in these scenarios.

ChargeCache is *not* based on the charge difference that occurs due to temperature dependence. Rather, we exploit the high level of charge in recently-precharged rows to reduce timing parameters during later accesses to such rows. After conducting tests to determine the reduction in timing parameters (for ChargeCache hits) at *worst-case* temperatures, we find that these timing parameters can be reduced independently of the operating temperature. Dynamic latency scaling can still be used in conjunction with ChargeCache at low temperatures to reduce the access latency even further.

7.2. Applicability to Other DRAM Standards

Although we evaluate only DDR3-based main memory within the scope of this paper, implementing ChargeCache for other DRAM standards is straightforward. In theory, ChargeCache is applicable to any memory technology where cells are volatile (leak charge over time). However, the memory interface can prevent the implementation of ChargeCache entirely in the memory controller. For example, RL-DRAM [56] is a DRAM type incompatible with ChargeCache. In RL-DRAM, read and write operations are directly handled by *READ* and *WRITE* commands without explicitly activating and precharging DRAM rows. Hence, the RL-DRAM memory controller does not have control over the activation delay of the rows and the timing parameters *tRCD* and *tRAS* do not exist.

However, ChargeCache can be used with to a large set of specifications derived from DDR (DDRx, GDDRx, LPDDRx, etc.) in a manner similar to the mechanism described in this work, without modifying the DRAM architecture at all. All of these memories require *ACT* and *PRE* commands to explicitly open and close DRAM rows. Using ChargeCache with 3D-stacked memories [47, 53] such as WideIO [20], HBM [34] and HMC [32] is also straightforward. The difference is that the DRAM controller, and hence ChargeCache, may be implemented in the logic layer of the 3D-stacked memory chip instead of the processor chip.

8. Related Work

To our knowledge, this paper is the first to (*i*) show that applications typically exhibit significant *Row-level Temporal Locality (RLTL)* and (*ii*) exploit this locality to improve system performance by reducing the latency of requests to recently-accessed rows.

We have already qualitatively and quantitatively (in Sections 3 and 6) compared ChargeCache to NUAT [85], which reduces access latency to *only* recently-refreshed rows. We have shown that ChargeCache can provide significantly higher average latency reduction than NUAT because RLTL is usually high, whereas the fraction of accesses to rows that are recently-refreshed is typically low.

Other previous works have proposed techniques to reduce performance degradation caused by long DRAM latencies. They focused on 1) enhancing the DRAM, 2) exploiting variations in manufacturing process and operating conditions, 3) developing several memory scheduling policies. We briefly summarize how ChargeCache differs from these works.

Enhancing DRAM Architecture. Lee at al. propose Tiered-Latency DRAM (TL-DRAM) [49] which divides each subarray into near and far segments using isolation transistors. With TL-DRAM, the memory controller accesses the near segment with lower latency since the isolation transistor reduces bitline capacitance in that segment. Our mechanism could be implemented on top of TL-DRAM to reduce the access latency for both the near and far segment. Kim et al. unlock parallelism among subarrays at low cost with SALP [42]. The goal of SALP is to reduce DRAM latency by providing more parallelism to reduce the impact of bank conflicts. O et al [68] propose a DRAM architecture where sense amplifiers are decoupled from bitlines to mitigate precharge latency. Choi et al [15] propose to utilize multiple DRAM cells to store a single bit when sufficient DRAM capacity is available. By using multiple cells, they reduce activation, precharge and refresh latencies. Other works [11, 12, 30, 81–83, 88, 98] also propose new DRAM architectures to lower DRAM latency.

Unlike ChargeCache, all these works require changes to the DRAM architecture itself. The approaches taken by these works are largely orthogonal and ChargeCache could be implemented together with any of these mechanisms to further improve the DRAM latency.

Exploiting Process and Operating Condition Variations. Recent studies [9, 48] proposed methods to reduce the safety margins of the DRAM timing parameters when operating conditions are appropriate (i.e., not worst-case). Unlike these works, ChargeCache is largely independent of operating conditions like temperature, as discussed in Section 7.1, and is orthogonal to these latency reduction mechanisms.

Memory Request Scheduling Policies. Memory request scheduling policies (e.g., [40, 41, 45, 63, 64, 79, 91–93, 97]) reduce the average DRAM access latency by improving DRAM parallelism, row-buffer locality and fairness in especially multi-core systems. ChargeCache can be employed in conjunction with the scheduling policy that best suits the application and the underlying architecture.

9. Conclusion

We introduce ChargeCache, a new, low-overhead mechanism that dynamically reduces the DRAM timing parameters for recently-accessed DRAM rows. ChargeCache exploits two key observations that we demonstrate in this work: 1) a recently-accessed DRAM row has cells with high amount of charge and thus can be accessed faster, 2) many applications repeatedly access rows that are recently-accessed.

Our extensive evaluations of ChargeCache on both single-core and multi-core systems show that it provides significant performance benefit and DRAM energy reduction at very modest hardware overhead. ChargeCache requires no modifications to the existing DRAM chips and occupies only a small area on the memory controller.

We conclude that ChargeCache is a simple yet efficient mechanism to dynamically reduce DRAM latency, which significantly improves both the performance and energy efficiency of modern systems.

Acknowledgments

We thank the reviewers and the SAFARI group members for their feedback. We acknowledge the generous support of Google, Intel, Nvidia, Samsung, and VMware. This work is supported in part by NSF grants 1212962, 1320531, and 1409723, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

[1] "Ramulator (source code)," https://github.com/CMU-SAFARI/ramulator.
[2] "STREAM Benchmark," http://www.streambench.org/.
[3] "TPC," http://www.tpc.org/.
[4] M. Awasthi *et al.*, "Prediction based DRAM row-buffer management in the many-core era," in *PACT*, 2011.
[5] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *ICS*, 1991.
[6] M. Bekerman *et al.*, "Correlated load-address predictors," in *ISCA*, 1999.
[7] B. Black *et al.*, "Die stacking (3D) microarchitecture." in *MICRO*, 2006.
[8] D. Chandra *et al.*, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA*, 2005.
[9] K. Chandrasekar *et al.*, "Exploiting expendable process-margins in DRAMs for run-time performance optimization," in *DATE*, 2014.
[10] K. Chandrasekar *et al.*, "Towards variation-aware system-level power estimation of DRAMs: an empirical approach," in *DAC*, 2013.
[11] K. K.-W. Chang *et al.*, "Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM," in *HPCA*, 2016.
[12] K. K.-W. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014.
[13] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM JRD*, 1997.
[14] N. Chatterjee *et al.*, "Leveraging heterogeneity in DRAM main memories to accelerate critical word access," in *MICRO*, 2012.
[15] J. Choi *et al.*, "Multiple clone row DRAM: a low latency and area optimized DRAM," in *ISCA*, 2015.
[16] Y. Chou *et al.*, "Microarchitecture optimizations for exploiting memory-level parallelism," in *ISCA*, 2004.
[17] R. Das *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA*, 2013.
[18] W. Ding *et al.*, "Compiler support for optimizing memory bank-level parallelism." in *MICRO*, 2014.
[19] N. Duong *et al.*, "Improving cache management policies using dynamic reuse distances," in *MICRO*, 2012.
[20] D. Dutoit *et al.*, "A 0.9 pJ/bit, 12.8 GByte/s WideIO memory interface in a 3D-IC NoC-based MPSoC," in *VLSIT*, 2013.
[21] E. Ebrahimi *et al.*, "Prefetch-aware shared resource management for multi-core systems," in *ISCA*, 2011.
[22] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.
[23] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO*, 2009.
[24] E. Ebrahimi *et al.*, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, 2009.
[25] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM JRD*, 1993.
[26] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[27] 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.
[28] A. Glew, "MLP yes! ILP no," in *ASPLOS WACI*, 1998.
[29] GSI, "Low latency DRAMs," http://www.gsitechnology.com.
[30] N. D. Gulur *et al.*, "Multiple sub-row buffers in DRAM: unlocking performance and energy improvement opportunities," in *ICS*, 2012.
[31] G. Hamerly *et al.*, "Simpoint 3.0: Faster and more flexible program phase analysis," *JILP*, 2005.
[32] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 2.0," Tech. Rep., November 2014.
[33] R. Iyer *et al.*, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, 2007.
[34] JEDEC, "High bandwidth memory (HBM) DRAM," 2013.
[35] M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems." in *HPCA*, 2012.
[36] D. Jevdjic *et al.*, "Unison cache: A scalable and effective die-stacked DRAM cache," in *MICRO*, 2014.
[37] J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.
[38] M. Kandemir *et al.*, "Memory row reuse distance and its role in optimizing application performance," in *SIGMETRICS*, 2015.
[39] S. Khan *et al.*, "The efficacy of error mitigation techniques for DRAM retention failures: a comparative experimental study," in *SIGMETRICS*, 2014.
[40] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
[41] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
[42] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (SALP) in DRAM." in *ISCA*, 2012.

[43] Y. Kim *et al.*, "Ramulator: A fast and extensible DRAM simulator," in *CAL*, 2015.
[44] C. J. Lee *et al.*, "Prefetch-aware DRAM controllers," in *MICRO*, 2008.
[45] C. J. Lee *et al.*, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," *UT-Austin, HPS, Tech. Report*, 2010.
[46] C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *MICRO*, 2009.
[47] D. Lee *et al.*, "Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost," *TACO*, 2016.
[48] D. Lee *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *HPCA*, 2015.
[49] D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture." in *HPCA*, 2013.
[50] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[51] J. Liu *et al.*, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *ISCA*, 2013.
[52] J. Liu *et al.*, "RAIDR: Retention-aware intelligent DRAM refresh," in *ISCA*, 2012.
[53] G. H. Loh, "3D-stacked memory architectures for multi-core processors." in *ISCA*, 2008.
[54] P. Lotfi-Kamran *et al.*, "Scale-out processors," in *ISCA*, 2012.
[55] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[56] Micron, "RLDRAM 2 and 3 specifications," http://www.micron.com/products/dram/rldram-memory.
[57] Micron Technology, "4Gb: x4, x8, x16 DDR3 SDRAM," 2011.
[58] Y. Mori *et al.*, "The origin of variable retention time in DRAM," in *IEDM*, 2005.
[59] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.
[60] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.
[61] O. Mutlu *et al.*, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *MICRO*, 2005.
[62] O. Mutlu *et al.*, "Techniques for efficient processing in runahead execution engines," in *ISCA*, 2005.
[63] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.
[64] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems." in *ISCA*, 2008.
[65] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.
[66] O. Mutlu and L. Subramanian, "Research problems and opportunities in memory systems," *SUPERFRI*, 2014.
[67] P. Nair *et al.*, "A case for refresh pausing in DRAM memory systems," in *HPCA*, 2013.
[68] S. O *et al.*, "Row-buffer decoupling: a case for low-latency DRAM microarchitecture," in *ISCA*, 2014.
[69] V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *MICRO*, 1999.
[70] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ISCA*, 1994.
[71] Y. N. Patt *et al.*, "HPS, a new microarchitecture: rationale and introduction," in *MICRO*, 1985.
[72] PTM, "Predictive technology model," http://ptm.asu.edu/.
[73] K. Puttaswamy and G. H. Loh, "Thermal analysis of a 3d die-stacked high-performance microprocessor." in *GLSVLSI*, 2006.
[74] M. K. Qureshi *et al.*, "Adaptive insertion policies for high performance caching," in *ISCA*, 2007.
[75] M. K. Qureshi *et al.*, "AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems," in *DSN*, 2015.
[76] M. K. Qureshi *et al.*, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *HPCA*, 2007.
[77] Rambus, "DRAM power model (2010)," http://www.rambus.com/energy.
[78] B. R. Rau, "Pseudo-randomly interleaved memory," in *ISCA*, 1991.
[79] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.
[80] Y. Sato *et al.*, "Fast cycle RAM (FCRAM); a 20-ns random row access, pipe-lined operating DRAM," in *VLSI Circuits*, 1998.
[81] V. Seshadri *et al.*, "Fast bulk bitwise AND and OR in DRAM," in *CAL*, 2015.
[82] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.
[83] V. Seshadri *et al.*, "Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses," in *MICRO*, 2015.
[84] V. Seshadri *et al.*, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *PACT*, 2012.
[85] W. Shin *et al.*, "NUAT: A non-uniform access time memory controller." in *HPCA*, 2014.
[86] B. J. Smith, "A pipelined, shared resource MIMD computer," in *ICPP*, 1978.
[87] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *ASPLOS*, 2000.
[88] Y. H. Son *et al.*, "Reducing memory access latency with asymmetric DRAM bank organizations," in *ISCA*, 2013.
[89] SPEC CPU2006, "Standard Performance Evaluation Corporation," http://www.spec.org/cpu2006.
[90] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, 2007.
[91] L. Subramanian *et al.*, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
[92] L. Subramanian *et al.*, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*, 2015.
[93] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.
[94] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.
[95] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Fall Joint Computer Conference*, 1964.
[96] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM JRD*, 1967.
[97] H. Usui *et al.*, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *TACO*, 2016.
[98] T. Zhang *et al.*, "Half-DRAM: a high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation," in *ISCA*, 2014.
[99] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.
[100] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *IEEE TED*, 2006.
[101] 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," 1997, US Patent 5,630,096.