RAIDR: Retention-Aware Intelligent DRAM Refresh

Jamie Liu   Ben Jaiyen   Richard Veras   Onur Mutlu
Carnegie Mellon University
{jamiel,bjaiyen,rveras,onur}@cmu.edu

Abstract

Dynamic random-access memory (DRAM) is the building block of modern main memory systems. DRAM cells must be periodically refreshed to prevent loss of data. These refresh operations waste energy and degrade system performance by interfering with memory accesses. The negative effects of DRAM refresh increase as DRAM device capacity increases. Existing DRAM devices refresh all cells at a rate determined by the leakiest cell in the device. However, most DRAM cells can retain data for significantly longer. Therefore, many of these refreshes are unnecessary.

In this paper, we propose RAIDR (Retention-Aware Intelligent DRAM Refresh), a low-cost mechanism that can identify and skip unnecessary refreshes using knowledge of cell retention times. Our key idea is to group DRAM rows into retention time bins and apply a different refresh rate to each bin. As a result, rows containing leaky cells are refreshed as frequently as normal, while most rows are refreshed less frequently. RAIDR uses Bloom filters to efficiently implement retention time bins. RAIDR requires no modification to DRAM and minimal modification to the memory controller. In an 8-core system with 32 GB DRAM, RAIDR achieves a 74.6% refresh reduction, an average DRAM power reduction of 16.1%, and an average system performance improvement of 8.6% over existing systems, at a modest storage overhead of 1.25 KB in the memory controller. RAIDR's benefits are robust to variation in DRAM system configuration, and increase as memory capacity increases.

1. Introduction

Modern main memory is composed of dynamic random-access memory (DRAM) cells. A DRAM cell stores data as charge on a capacitor. Over time, this charge leaks, causing the stored data to be lost. To prevent this, data stored in DRAM must be periodically read out and rewritten, a process called *refreshing*. DRAM refresh operations waste energy and also degrade performance by delaying memory requests. These problems are expected to worsen as DRAM scales to higher densities.

Previous work has attacked the problems caused by DRAM refresh from both hardware and software angles. Some hardware-only approaches have proposed modifying DRAM devices to refresh DRAM cells at different rates [19, 20, 37, 52], but these incur 5–20% area overheads on the DRAM die [20, 37] and are therefore difficult to implement given the cost-sensitive DRAM market. Other hardware-only approaches have proposed modifying memory controllers, either to avoid unnecessary refreshes [7] or decrease refresh

rate and tolerate retention errors using error-correcting codes (ECC) [5, 17, 51], but these suffer from significant storage or bandwidth overheads. Hardware-software cooperative techniques have been proposed to decrease refresh rate and allow retention errors only in unused [11, 50] or non-critical [26] regions of memory, but these substantially complicate the operating system while still requiring significant hardware support.

In this paper, our goal is to minimize the number of refresh operations performed without significantly increasing hardware or software complexity and without making modifications to DRAM chips. We exploit the observation that only a small number of weak DRAM cells require the conservative minimum refresh interval of 64 ms that is common in current DRAM standards. For example, Figure 1 shows that in a 32 GB DRAM system, fewer than 1000 cells (out of over 10^{11}) require a refresh interval shorter than 256 ms, which is four times the minimum refresh interval. Therefore, refreshing most DRAM cells at a low rate, while selectively refreshing weak cells at a higher rate, can result in a significant decrease in refresh overhead. To this end, we propose *Retention-Aware Intelligent DRAM Refresh (RAIDR)*. RAIDR groups DRAM rows into retention time bins based on the refresh rate they require to retain data. Rows in each bin are refreshed at a different rate, so that rows are only refreshed frequently if they require a high refresh rate. RAIDR stores retention time bins in the memory controller, avoiding the need to modify DRAM devices. Retention time bins are stored using Bloom filters [2]. This allows for low storage overhead and ensures that bins never overflow, yielding correct operation regardless of variation in DRAM system capacity or in retention time distribution between DRAM chips.

Our experimental results show that a configuration of RAIDR with only two retention time bins is able to reduce DRAM system power by 16.1% while improving system performance by 8.6% in a 32 GB DRAM system at a modest storage overhead of 1.25 KB in the memory controller. We compare our mechanism to previous mechanisms that reduce refresh overhead and show that RAIDR results in the highest energy savings and performance gains.

Our contributions are as follows:

- We propose a low-cost mechanism that exploits inter-cell variation in retention time in order to decrease refresh rate. In a configuration with only two retention time bins, RAIDR achieves a 74.6% refresh reduction with no modifications to DRAM and only 1.25 KB storage overhead in a 32 GB memory controller.
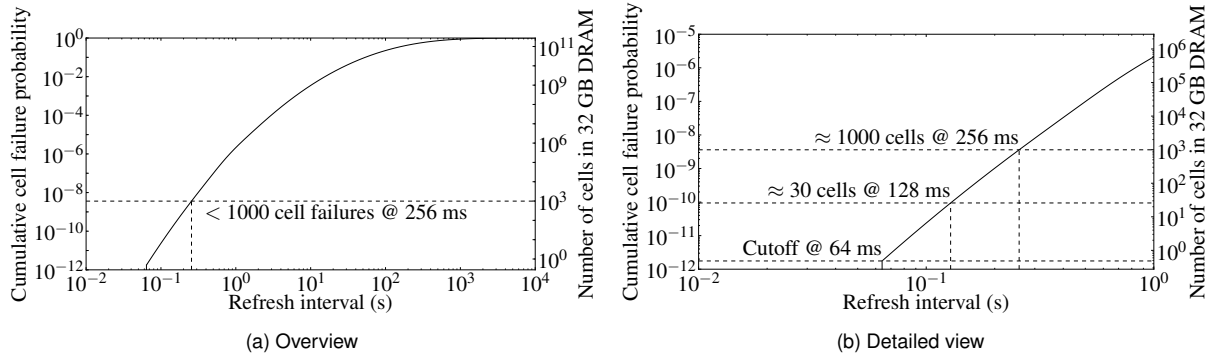
(a) Overview   (b) Detailed view

Figure 1: DRAM cell retention time distribution in a 60 nm process (based on data from [21])

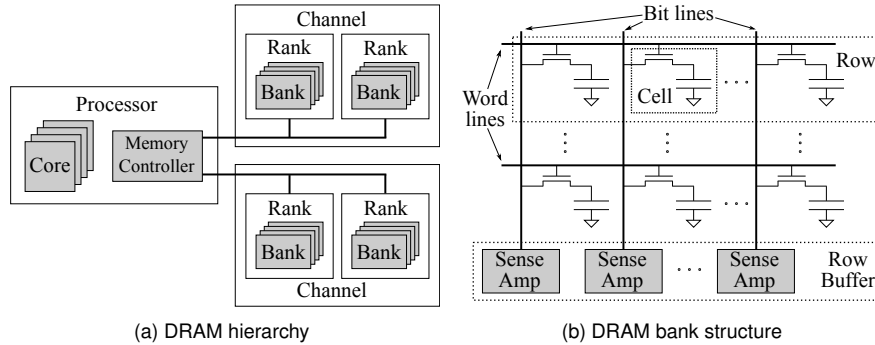(a) DRAM hierarchy   (b) DRAM bank structure

Figure 2: DRAM system organization

- We show that RAIDR is configurable, allowing a system designer to balance implementation overhead and refresh reduction. We show that RAIDR scales effectively to projected future systems, offering increasing performance and energy benefits as DRAM devices scale in density.

2. Background and Motivation

2.1. DRAM Organization and Operation

We present a brief outline of the organization and operation of a modern DRAM main memory system. Physical structures such as the DIMM, chip, and sub-array are abstracted by the logical structures of rank and bank for clarity where possible. More details can be found in [18].

A modern DRAM main memory system is organized hierarchically as shown in Figure 2a. The highest level of the hierarchy is the channel. Each channel has command, address, and data buses that are independent from those of other channels, allowing for fully concurrent access between channels. A channel contains one or more ranks. Each rank corresponds to an independent set of DRAM devices. Hence, all ranks in a channel can operate in parallel, although this *rank-level parallelism* is constrained by the shared channel bandwidth. Within each rank is one or more banks. Each bank corresponds to a distinct DRAM cell array. As such, all banks in a rank can operate in parallel, although this *bank-level parallelism* is constrained both by the shared channel bandwidth as well as by resources that are shared between banks on each DRAM device, such as device power.

Each DRAM bank consists of a two-dimensional array of DRAM cells, as shown in Figure 2b. A DRAM cell consists of a capacitor and an access transistor. Each access transistor connects a capacitor to a wire called a *bitline* and is controlled by a wire called a *wordline*. Cells sharing a wordline form a *row*. Each bank also contains a row of sense amplifiers, where each sense amplifier is connected to a single bitline. This row of sense amplifiers is called the bank's *row buffer*.

Data is represented by charge on a DRAM cell capacitor. In order to access data in DRAM, the row containing the data must first be *opened* (or *activated*) to place the data on the bitlines. To open a row, all bitlines must previously be precharged to $V_{DD}/2$. The row's wordline is enabled, connecting all capacitors in that row to their respective bitlines. This causes charge to flow from the capacitor to the bitline (if the capacitor is charged to V_{DD}) or vice versa (if the capacitor is at 0 V). In either case, the sense amplifier connected to that bitline detects the voltage change and amplifies it, driving the bitline fully to either V_{DD} or 0 V. Data in the open row can then be read or written by sensing or driving the voltage on the appropriate bitlines.

Successive accesses to the same row, called *row hits*, can be serviced without opening a new row. Accesses to different rows in the same bank, called *row misses*, require a different row to be opened. Since all rows in the bank share the same bitlines, only one row can be open at a time. To close a row, the row's word line is disabled, disconnecting the capacitors from the bitlines, and the bitlines are *precharged* to $V_{DD}/2$ so that another row can be opened. Opening a row requires driving the row's wordline as well as all of the bitlines; due to the high parasitic capacitance of each wire, opening a row is expensive both in latency and in power. Therefore, row
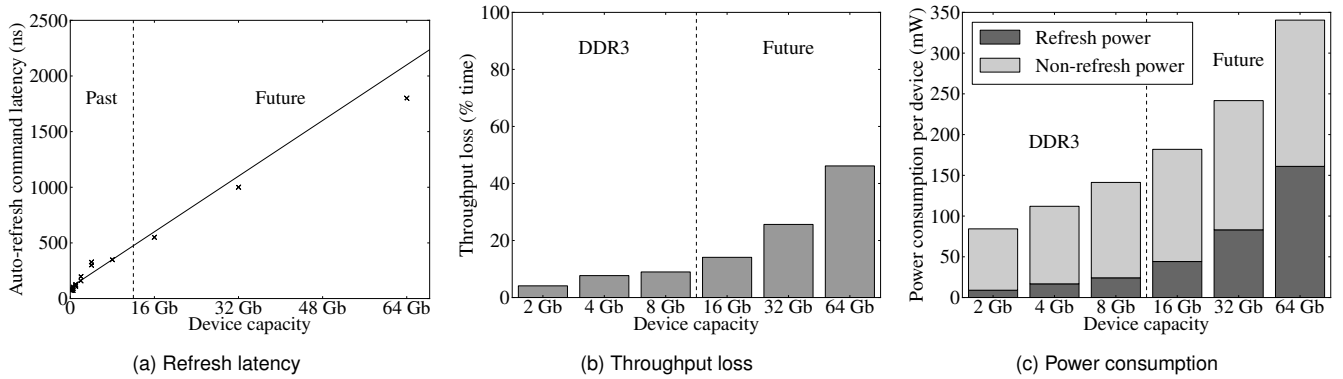
Figure 3: Adverse effects of refresh in contemporary and future DRAM devices

hits are serviced with both lower latency and lower energy consumption than row misses.

The capacity of a DRAM device is the number of rows in the device times the number of bits per row. Increasing the number of bits per row increases the latency and power consumption of opening a row due to longer wordlines and the increased number of bitlines driven per activation [18]. Hence, the size of each row has remained limited to between 1 KB and 2 KB for several DRAM generations, while the number of rows per device has scaled linearly with DRAM device capacity [13, 14, 15].

2.2. DRAM Refresh

DRAM cells lose data because capacitors leak charge over time. In order to preserve data integrity, the charge on each capacitor must be periodically restored or *refreshed*. When a row is opened, sense amplifiers drive each bit line fully to either V_{DD} or 0 V. This causes the opened row's cell capacitors to be fully charged to V_{DD} or discharged to 0 V as well. Hence, a row is refreshed by opening it.[1] The refresh interval (time between refreshes for a given cell) has remained constant at 64 ms for several DRAM generations [13, 14, 15, 18].

In typical modern DRAM systems, the memory controller periodically issues an *auto-refresh* command to the DRAM.[2] The DRAM chip then chooses which rows to refresh using an internal counter, and refreshes a number of rows based on the device capacity. During *normal temperature* operation (below 85 °C), the average time between auto-refresh commands (called t_{REFI}) is 7.8 μs [15]. In the *extended temperature* range (between 85 °C and 95 °C), the temperature range in which dense server environments operate [10] and 3D-stacked DRAMs are expected to operate [1], the time between auto-refresh commands is halved to 3.9 μs [15]. An auto-refresh operation occupies all banks on the rank simultaneously (preventing the rank from servicing any requests) for a length of time t_{RFC}, where t_{RFC} depends on the number of rows being refreshed.[3] Previous DRAM generations

---

[1] After the refresh operation, it is of course necessary to precharge the bank before another row can be opened to service requests.

[2] Auto-refresh is sometimes called CAS-before-RAS refresh [30].

[3] Some devices support *per-bank* refresh commands, which refresh several rows at a single bank [16], allowing for bank-level parallelism at a rank during refreshes. However, this feature is not available in most DRAM devices.

also allowed the memory controller to perform refreshes by opening rows one-by-one (called RAS-only refresh [30]), but this method has been deprecated due to the additional power required to send row addresses on the bus.

Refresh operations negatively impact both performance and energy efficiency. Refresh operations degrade performance in three ways:

1. *Loss of bank-level parallelism:* A DRAM bank cannot service requests whenever it is refreshing, which results in decreased memory system throughput.

2. *Increased memory access latency:* Any accesses to a DRAM bank that is refreshing must wait for the refresh latency t_{RFC}, which is on the order of 300 ns in contemporary DRAM [15].

3. *Decreased row hit rate:* A refresh operation causes all open rows at a rank to be closed, which causes a large number of row misses after each refresh operation, leading to reduced memory throughput and increased memory latency.

Refresh operations also degrade energy efficiency, both by consuming significant amounts of energy (since opening a row is a high power operation) and by reducing memory system performance (as increased execution time results in increased static energy consumption). The power cost of refresh operations also limits the extent to which refresh operations can be parallelized to overlap their latencies, exacerbating the performance problem.

All of these problems are expected to worsen as DRAM device capacity increases. We estimate refresh latency by linearly extrapolating t_{RFC} from its value in previous and current DRAM generations, as shown in Figure 3a. Note that even with conservative estimates to account for future innovations in DRAM technology, the refresh operation latency exceeds 1 μs by the 32 Gb density node, because power constraints force refresh latency to increase approximately linearly with DRAM density. Next, we estimate throughput loss from refresh operations by observing that it is equal to the time spent refreshing per refresh command (t_{RFC}) divided by the time interval between refresh commands (t_{REFI}). This estimated throughput loss (in extended-temperature operation) is shown in Figure 3b. Throughput loss caused by refreshing quickly becomes untenable, reaching nearly 50% at the 64 Gb density node. Finally, to estimate refresh energy consumption, we
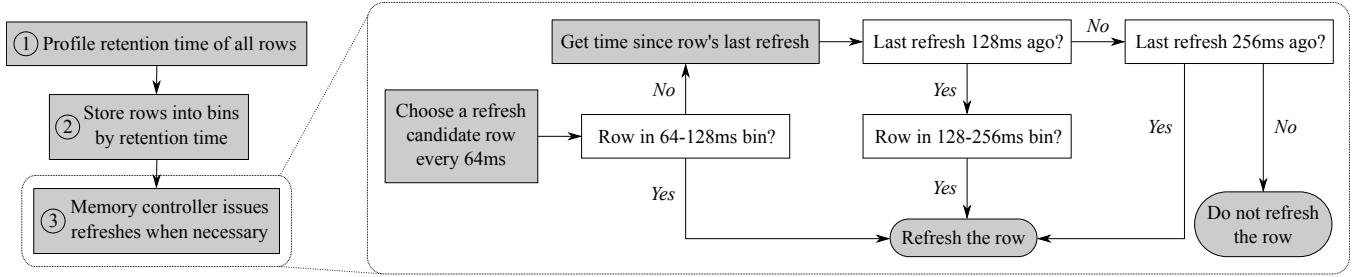
Figure 4: RAIDR operation

apply the power evaluation methodology described in [31], extrapolating from previous and current DRAM devices, as shown in Figure 3c. Refresh power rapidly becomes the dominant component of DRAM power, since as DRAM scales in density, other components of DRAM power increase slowly or not at all.[4] Hence, DRAM refresh poses a clear scaling challenge due to both performance and energy considerations.

2.3. DRAM Retention Time Distribution

The time before a DRAM cell loses data depends on the leakage current for that cell's capacitor, which varies between cells within a device. This gives each DRAM cell a characteristic *retention time*. Previous studies have shown that DRAM cell retention time can be modeled by categorizing cells as either *normal* or *leaky*. Retention time within each category follows a log-normal distribution [8, 21, 25]. The overall retention time distribution is therefore as shown in Figure 1 [21].[5]

The DRAM refresh interval is set by the DRAM cell with the lowest retention time. However, the vast majority of cells can tolerate a much longer refresh interval. Figure 1b shows that in a 32 GB DRAM system, on average only $\approx$ 30 cells cannot tolerate a refresh interval that is twice as long, and only $\approx 10^3$ cells cannot tolerate a refresh interval four times longer. For the vast majority of the 10^{11} cells in the system, the refresh interval of 64 ms represents a significant waste of energy and time.

Our goal in this paper is to design a mechanism to minimize this waste. By refreshing only rows containing low-retention cells at the maximum refresh rate, while decreasing the refresh rate for other rows, we aim to significantly reduce the number of refresh operations performed.

3. Retention-Aware Intelligent DRAM Refresh

3.1. Overview

A conceptual overview of our mechanism is shown in Figure 4. We define a *row's retention time* as the minimum retention time across all cells in that row. A set of bins is added to the memory controller, each associated with a range of retention times. Each bin contains all of the rows whose retention time falls

into that bin's range. The shortest retention time covered by a given bin is the *bin's refresh interval*. The shortest retention time that is not covered by any bins is the *new default refresh interval*. In the example shown in Figure 4, there are 2 bins. One bin contains all rows with retention time between 64 and 128 ms; its bin refresh interval is 64 ms. The other bin contains all rows with retention time between 128 and 256 ms; its bin refresh interval is 128 ms. The new default refresh interval is set to 256 ms. The number of bins is an implementation choice that we will investigate in Section 6.5.

A retention time profiling step determines each row's retention time (① in Figure 4). For each row, if the row's retention time is less than the new default refresh interval, the memory controller inserts it into the appropriate bin ②. During system operation ③, the memory controller ensures that each row is chosen as a refresh candidate every 64 ms. Whenever a row is chosen as a refresh candidate, the memory controller checks each bin to determine the row's retention time. If the row appears in a bin, the memory controller issues a refresh operation for the row if the bin's refresh interval has elapsed since the row was last refreshed. Otherwise, the memory controller issues a refresh operation for the row if the default refresh interval has elapsed since the row was last refreshed. Since each row is refreshed at an interval that is equal to or shorter than its measured retention time, data integrity is guaranteed.

Our idea consists of three key components: (1) retention time profiling, (2) storing rows into retention time bins, and (3) issuing refreshes to rows when necessary. We discuss how to implement each of these components in turn in order to design an efficient implementation of our mechanism.

3.2. Retention Time Profiling

Measuring row retention times requires measuring the retention time of each cell in the row. The straightforward method of conducting these measurements is to write a small number of static patterns (such as "all 1s" or "all 0s"), turning off refreshes, and observing when the first bit changes [50].[6]

Before the row retention times for a system are collected, the memory controller performs refreshes using the baseline auto-refresh mechanism. After the row retention times for a system have been measured, the results can be saved in a file by the operating system. During future boot-ups, the results can be

---

[4]DRAM static power dissipation is dominated by leakage in periphery such as I/O ports, which does not usually scale with density. Outside of refresh operations, DRAM dynamic power consumption is dominated by activation power and I/O power. Activation power is limited by activation latency, which has remained roughly constant, while I/O power is limited by bus frequency, which scales much more slowly than device capacity [12].

[5]Note that the curve is truncated on the left at 64 ms because a cell with retention time less than 64 ms results in the die being discarded.

[6]Circuit-level crosstalk effects cause retention times to vary depending on the values stored in nearby bits, and the values that cause the worst-case retention time depend on the DRAM bit array architecture of a particular device [36, 25]. We leave further analysis of this problem to future work.
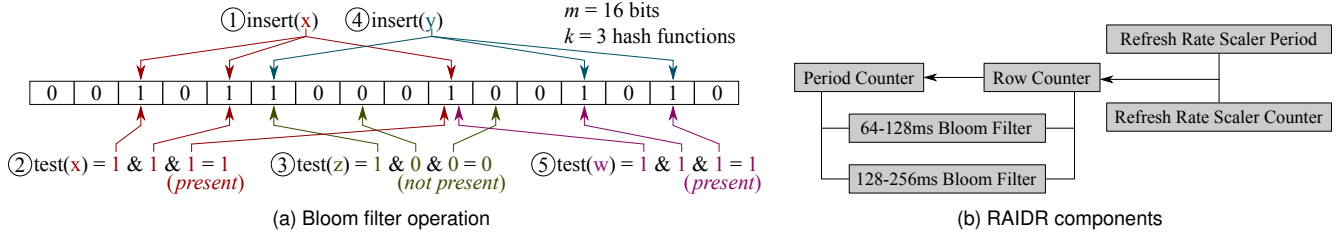
Figure 5: RAIDR implementation details

3.3. Storing Retention Time Bins: Bloom Filters

restored into the memory controller without requiring further profiling, since retention time does not change significantly over a DRAM cell's lifetime [8].[7]

The memory controller must store the set of rows in each bin. A naive approach to storing retention time bins would use a table of rows for each bin. However, the exact number of rows in each bin will vary depending on the amount of DRAM in the system, as well as due to retention time variation between DRAM chips (especially between chips from different manufacturing processes). If a table's capacity is inadequate to store all of the rows that fall into a bin, this implementation no longer provides correctness (because a row not in the table could be refreshed less frequently than needed) and the memory controller must fall back to refreshing all rows at the maximum refresh rate. Therefore, tables must be sized conservatively (i.e. assuming a large number of rows with short retention times), leading to large hardware cost for table storage.

To overcome these difficulties, we propose the use of Bloom filters [2] to implement retention time bins. A Bloom filter is a structure that provides a compact way of representing set membership and can be implemented efficiently in hardware [4, 28].

A Bloom filter consists of a bit array of length m and k distinct hash functions that map each element to positions in the array. Figure 5a shows an example Bloom filter with a bit array of length $m = 16$ and $k = 3$ hash functions. All bits in the bit array are initially set to 0. To insert an element into the Bloom filter, the element is hashed by all k hash functions, and all of the bits in the corresponding positions are set to 1 (① in Figure 5a). To test if an element is in the Bloom filter, the element is hashed by all k hash functions. If all of the bits at the corresponding bit positions are 1, the element is declared to be *present* in the set ②. If any of the corresponding bits are 0, the element is declared to be *not present* in the set ③. An element can never be removed from a Bloom filter. Many different elements may map to the same bit, so inserting other elements ④ may lead to a *false positive*, where an element is incorrectly declared to be present in the set even though it was never inserted into the Bloom filter ⑤. However, because bits are never reset to 0, an element can never be incorrectly declared to be not present in the set; that is, a *false negative* can never occur. A Bloom filter is therefore

a highly storage-efficient set representation in situations where the possibility of false positives and the inability to remove elements are acceptable. We observe that the problem of storing retention time bins is such a situation. Furthermore, unlike the previously discussed table implementation, a Bloom filter can contain any number of elements; the probability of a false positive gradually increases with the number of elements inserted into the Bloom filter, but false negatives will never occur. In the context of our mechanism, this means that rows may be refreshed more frequently than necessary, but a row is never refreshed less frequently than necessary, so data integrity is guaranteed.

The Bloom filter parameters m and k can be optimally chosen based on expected capacity and desired false positive probability [23]. The particular hash functions used to index the Bloom filter are an implementation choice. However, the effectiveness of our mechanism is largely insensitive to the choice of hash function, since weak cells are already distributed randomly throughout DRAM [8]. The results presented in Section 6 use a hash function based on the *xorshift* pseudo-random number generator [29], which in our evaluation is comparable in effectiveness to H_3 hash functions that can be easily implemented in hardware [3, 40].

3.4. Performing Refresh Operations

During operation, the memory controller periodically chooses a candidate row to be considered for refreshing, decides if it should be refreshed, and then issues the refresh operation if necessary. We discuss how to implement each of these in turn.

Selecting A Refresh Candidate Row We choose all refresh intervals to be multiples of 64 ms, so that the problem of choosing rows as refresh candidates simply requires that each row is selected as a refresh candidate every 64 ms. This is implemented with a *row counter* that counts through every row address sequentially. The rate at which the row counter increments is chosen such that it rolls over every 64 ms.

If the row counter were to select every row in a given bank consecutively as a refresh candidate, it would be possible for accesses to that bank to become starved, since refreshes are prioritized over accesses for correctness. To avoid this, consecutive refresh candidates from the row counter are striped across banks. For example, if the system contains 8 banks, then every 8th refresh candidate is at the same bank.

Determining Time Since Last Refresh Determining if a row needs to be refreshed requires determining how many 64 ms intervals have elapsed since its last refresh. To simplify this problem, we choose all refresh intervals to be power-of-2 mul-

---

[7]Retention time *is* significantly affected by temperature. We will discuss how temperature variation is handled in Section 3.5.

tiples of 64 ms. We then add a second counter, called the *period counter*, which increments whenever the row counter resets. The period counter counts to the default refresh interval divided by 64 ms, and then rolls over. For example, if the default refresh interval is 256 ms = 4 × 64 ms, the period counter is 2 bits and counts from 0 to 3.

The least significant bit of the period counter is 0 with period 128 ms, the 2 least significant bits of the period counter are 00 with period 256 ms, etc. Therefore, a straightforward method of using the period counter in our two-bin example would be to probe the 64 ms–128 ms bin regardless of the value of the period counter (at a period of 64 ms), only probe the 128 ms–256 ms bin when the period counter's LSB is 0 (at a period of 128 ms), and refresh all rows when the period counter is 00 (at a period of 256 ms). While this results in correct operation, this may lead to an undesirable "bursting" pattern of refreshes, in which every row is refreshed in certain 64 ms periods while other periods have very few refreshes. This may have an adverse effect on performance. In order to distribute refreshes more evenly in time, the LSBs of the row counter are compared to the LSBs of the period counter. For example, a row with LSB 0 that must be refreshed every 128 ms is refreshed when the LSB of the period counter is 0, while a row with LSB 1 with the same requirement is refreshed when the LSB of the period counter is 1.

Issuing Refreshes In order to refresh a specific row, the memory controller simply activates that row, essentially performing a RAS-only refresh (as described in Section 2.2). Although RAS-only refresh is deprecated due to the power consumed by issuing row addresses over the DRAM address bus, we account for this additional power consumption in our evaluations and show that the energy saved by RAIDR outweighs it.

3.5. Tolerating Temperature Variation: Refresh Rate Scaling

Increasing operational temperature causes DRAM retention time to decrease. For instance, the DDR3 specification requires a doubled refresh rate for DRAM being operated in the extended temperature range of 85 °C to 95 °C [15]. However, change in retention time as a function of temperature is predictable and consistent across all affected cells [8]. We leverage this property to implement a *refresh rate scaling* mechanism to compensate for changes in temperature, by allowing the refresh rate for all cells to be adjusted by a multiplicative factor. This rate scaling mechanism resembles the temperature-compensated self-refresh feature available in some mobile DRAMs (e.g. [32]), but is applicable to any DRAM system.

The refresh rate scaling mechanism consists of two parts. First, when a row's retention time is determined, the measured time is converted to the retention time at some reference temperature T_{REF} based on the current device temperature. This temperature-compensated retention time is used to determine which bin the row belongs to. Second, the row counter is modified so that it only increments whenever a third counter, called the *refresh rate scaler*, rolls over. The refresh rate scaler increments at a constant frequency, but has a programmable period chosen based on the temperature. At T_{REF}, the rate scaler's

period is set such that the row counter rolls over every 64 ms. At higher temperatures, the memory controller decreases the rate scaler's period such that the row counter increments and rolls over more frequently. This increases the refresh rate for all rows by a constant factor, maintaining correctness.

The reference temperature and the bit length of the refresh rate scaler are implementation choices. In the simplest implementation, $T_{REF} = 85$ °C and the refresh rate scaler is 1 bit, with the refresh rate doubling above 85 °C. This is equivalent to how temperature variation is handled in existing systems, as discussed in Section 2.2. However, a rate scaler with more than 1 bit allows more fine-grained control of the refresh interval than is normally available to the memory controller.

3.6. Summary

Figure 5b summarizes the major components that RAIDR adds to the memory controller. In total, RAIDR requires (1) three counters, (2) bit arrays to store the Bloom filters, and (3) hash functions to index the Bloom filters. The counters are relatively short; the longest counter, the row counter, is limited in length to the longest row address supported by the memory controller, which in current systems is on the order of 24 bits. The majority of RAIDR's hardware overhead is in the Bloom filters, which we discuss in Section 6.3. The logic required by RAIDR lies off the critical path of execution, since the frequency of refreshes is much smaller than a processor's clock frequency, and refreshes are generated in parallel with the memory controller's normal functionality.

3.7. Applicability to eDRAM and 3D-Stacked DRAM

So far, we have discussed RAIDR only in the context of a memory controller for a conventional DRAM system. In this section, we briefly discuss RAIDR's applicability to two relatively new types of DRAM systems, 3D die-stacked DRAMs and embedded DRAM (eDRAM).

In the context of DRAM, 3D die-stacking has been proposed to improve memory latency and bandwidth by stacking DRAM dies on processor logic dies [1, 39], as well as to improve DRAM performance and efficiency by stacking DRAM dies onto a sophisticated controller die [9]. While 3D stacking may allow for increased throughput and bank-parallelism, this does not alleviate refresh overhead; as discussed in Section 2.2, the rate at which refresh operations can be performed is limited by their power consumption, which 3D die stacking does not circumvent. Furthermore, DRAM integrated in a 3D stack will operate at temperatures over 90 °C [1], leading to reduced retention times (as discussed in Section 3.5) and exacerbating the problems caused by DRAM refresh. Therefore, refresh is likely to be of significant concern in a 3D die-stacked DRAM.

eDRAM is now increasingly integrated onto processor dies in order to implement on-chip caches that are much more dense than traditional SRAM arrays, e.g. [43]. Refresh power is the dominant power component in an eDRAM [51], because although eDRAM follows the same retention time distribution (featuring normal and leaky cells) described in Section 2.3, retention times are approximately three orders of magnitude smaller [24].

RAIDR is applicable to both 3D die-stacked DRAM and eDRAM systems, and is synergistic with several characteristics of both. In a 3D die-stacked or eDRAM system, the controller logic is permanently fused to the DRAM. Hence, the attached DRAM can be retention-profiled once, and the results stored permanently in the memory controller, since the DRAM system will never change. In such a design, the Bloom filters could be implemented using laser- or electrically-programmable fuses or ROMs. Furthermore, if the logic die and DRAM reside on the same chip, then the power overhead of RAS-only refreshes decreases, improving RAIDR's efficiency and allowing it to reduce idle power more effectively. Finally, in the context of 3D die-stacked DRAM, the large logic die area may allow more flexibility in choosing more aggressive configurations for RAIDR that result in greater power savings, as discussed in Section 6.5. Therefore, we believe that RAIDR's potential applications to 3D die-stacked DRAM and eDRAM systems are quite promising.

4. Related Work

To our knowledge, RAIDR is the first work to propose a low-cost memory controller modification that reduces DRAM refresh operations by exploiting variability in DRAM cell retention times. In this section, we discuss prior work that has aimed to reduce the negative effects of DRAM refresh.

4.1. Modifications to DRAM Devices

Kim and Papaefthymiou [19, 20] propose to modify DRAM devices to allow them to be refreshed on a finer block-based granularity with refresh intervals varying between blocks. In addition, their proposal adds redundancy within each block to further decrease refresh intervals. Their modifications impose a DRAM die area overhead on the order of 5%. Yanagisawa [52] and Ohsawa et al. [37] propose storing the retention time of each row in registers in DRAM devices and varying refresh rates based on this stored data. Ohsawa et al. [37] estimate that the required modifications impose a DRAM die area overhead between 7% and 20%. [37] additionally proposes modifications to DRAM, called Selective Refresh Architecture (SRA), to allow software to mark DRAM rows as unused, preventing them from being refreshed. This latter mechanism carries a DRAM die area overhead of 5% and is orthogonal to RAIDR. All of these proposals are potentially unattractive since DRAM die area overhead results in an increase in the cost per DRAM bit. RAIDR avoids this cost since it does not modify DRAM.

Emma et al. [6] propose to suppress refreshes and mark data in DRAM as invalid if the data is older than the refresh interval. While this may be suitable in systems where DRAM is used as a cache, allowing arbitrary data in DRAM to become invalid is not suitable for conventional DRAM systems.

Song [45] proposes to associate each DRAM row with a *referenced bit* that is set whenever a row is accessed. When a row becomes a refresh candidate, if its referenced bit is set, its referenced bit is cleared and the refresh is skipped. This exploits the fact that opening a row causes it to be refreshed. Patel et al. [38] note that DRAM retention errors are unidirec-

tional (since charge only leaks off of a capacitor and not onto it), and propose to deactivate refresh operations for clusters of cells containing non-leaking values. These mechanisms are orthogonal to RAIDR.

4.2. Modifications to Memory Controllers

Katayama et al. [17] propose to decrease refresh rate and tolerate the resulting retention errors using ECC. Emma et al. [5] propose a similar idea in the context of eDRAM caches. Both schemes impose a storage overhead of 12.5%. Wilkerson et al. [51] propose an ECC scheme for eDRAM caches with 2% storage overhead. However, their mechanism depends on having long (1 KB) ECC code words. This means that reading any part of the code word (such as a single 64-byte cache line) requires reading the entire 1 KB code word, which would introduce significant bandwidth overhead in a conventional DRAM context.

Ghosh and Lee [7] exploit the same observation as Song [45]. Their Smart Refresh proposal maintains a timeout counter for each row that is reset when the row is accessed or refreshed, and refreshes a row only when its counter expires. Hence accesses to a row cause its refresh to be skipped. Smart Refresh is unable to reduce idle power, requires very high storage overhead (a 3-bit counter for every row in a 32 GB system requires up to 1.5 MB of storage), and requires workloads with large working sets to be effective (since its effectiveness depends on a large number of rows being activated and therefore not requiring refreshes). In addition, their mechanism is orthogonal to ours.

The DDR3 DRAM specification allows for some flexibility in refresh scheduling by allowing up to 8 consecutive refresh commands to be postponed or issued in advance. Stuecheli et al. [47] attempt to predict when the DRAM will remain idle for an extended period of time and schedule refresh operations during these idle periods, in order to reduce the interference caused by refresh operations and thus mitigate their performance impact. However, refresh energy is not substantially affected, since the number of refresh operations is not decreased. In addition, their proposed idle period prediction mechanism is orthogonal to our mechanism.

4.3. Modifications to Software

Venkatesan et al. [50] propose to modify the operating system so that it preferentially allocates data to rows with higher retention times, and refreshes the DRAM only at the lowest refresh interval of all allocated pages. Their mechanism's effectiveness decreases as memory capacity utilization increases. Furthermore, moving refresh management into the operating system can substantially complicate the OS, since it must perform hard-deadline scheduling in order to guarantee that DRAM refresh is handled in a timely manner.

Isen et al. [11] propose modifications to the ISA to enable memory allocation libraries to make use of Ohsawa et al.'s SRA proposal [37], discussed previously in Section 4.1. [11] builds directly on SRA, which is orthogonal to RAIDR, so [11] is orthogonal to RAIDR as well.

Table 1: Evaluated system configuration

Component	Specifications
Processor	8-core, 4 GHz, 3-wide issue, 128-entry instruction window, 16 MSHRs per core
Per-core cache	512 KB, 16-way, 64 B cache line size
Memory controller	FR-FCFS scheduling [41, 54], line-interleaved mapping, open-page policy
DRAM organization	32 GB, 2 channels, 4 ranks/channel, 8 banks/rank, 64K rows/bank, 8 KB rows
DRAM device	64x Micron MT41J512M8RA-15E (DDR3-1333) [33]

Table 2: Bloom filter properties

Retention range	Bloom filter size m	Number of hash functions k	Rows in bin	False positive probability
64 ms – 128 ms	256 B	10	28	$1.16 \cdot 10^{-9}$
128 ms – 256 ms	1 KB	6	978	0.0179

Liu et al. [26] propose Flikker, in which programmers designate data as non-critical, and non-critical data is refreshed at a much lower rate, allowing retention errors to occur. Flikker requires substantial programmer effort to identify non-critical data, and is complementary to RAIDR.

5. Evaluation Methodology

To evaluate our mechanism, we use an in-house x86 simulator with a cycle-accurate DRAM timing model validated against DRAMsim2 [42], driven by a frontend based on Pin [27]. Benchmarks are drawn from SPEC CPU2006 [46] and TPC-C and TPC-H [49]. Each simulation is run for 1.024 billion cycles, corresponding to 256 ms given our 4 GHz clock frequency.[8] DRAM system power was calculated using the methodology described in [31]. DRAM device power parameters are taken from [33], while I/O termination power parameters are taken from [53].

Except where otherwise noted, our system configuration is as shown in Table 1. DRAM retention distribution parameters correspond to the 60 nm technology data provided in [21]. A set of retention times was generated using these parameters, from which Bloom filter parameters were chosen as shown in Table 2, under the constraint that all Bloom filters were required to have power-of-2 size to simplify hash function implementation. We then generated a second set of retention times using the same parameters and performed all of our evaluations using this second data set.

For our main evaluations, we classify each benchmark as memory-intensive or non-memory-intensive based on its last-level cache misses per 1000 instructions (MPKI). Benchmarks with MPKI > 5 are memory-intensive, while benchmarks with MPKI < 5 are non-memory-intensive. We construct 5 different categories of workloads based on the fraction of memory-intensive benchmarks in each workload (0%, 25%, 50%, 75%, 100%). We randomly generate 32 multiprogrammed 8-core workloads for each category.

We report system performance using the commonly-used *weighted speedup* metric [44], where each application's instructions per cycle (IPC) is normalized to its IPC when run-

ning alone on the same system on the baseline auto-refresh configuration at the same temperature, and the weighted speedup of a workload is the sum of normalized IPCs for all applications in the workload.

We perform each simulation for a fixed number of cycles rather than a fixed number of instructions, since refresh timing is based on wall time. However, higher-performing mechanisms execute more instructions and therefore generate more memory accesses, which causes their total DRAM energy consumption to be inflated. In order to achieve a fair comparison, we report DRAM system power as *energy per memory access serviced*.

6. Results

We compare RAIDR to the following mechanisms:

- The auto-refresh baseline discussed in Section 2.2, in which the memory controller periodically issues auto-refresh commands, and each DRAM chip refreshes several rows per command,[9] as is implemented in existing systems [15].
- A "distributed" refresh scheme, in which the memory controller performs the same number of refreshes as in the baseline, but does so by refreshing one row at a time using RAS-only refreshes. This improves performance by allowing the memory controller to make use of bank-level parallelism while refresh operations are in progress, and by decreasing the latency of each refresh operation. However, it potentially increases energy consumption due to the energy cost of sending row addresses with RAS-only refreshes, as explained in Section 2.2.
- Smart Refresh [7], as described in Section 4.2. Smart Refresh also uses RAS-only refreshes, since it also requires control of refresh operations on a per-row granularity.
- An ideal scheme that performs no refreshes. While this is infeasible in practice, some ECC-based schemes may decrease refresh rate sufficiently to approximate it, though these come with significant overheads that may negate the benefits of eliminating refreshes, as discussed in Section 4.2.

For each refresh mechanism, we evaluate both the normal temperature range (for which a 64 ms refresh interval is prescribed) and the extended temperature range (where all retention times and refresh intervals are halved).

---

[8]The pattern of refreshes repeats on a period of 32, 64, 128, or 256 ms, depending on refresh mechanism and temperature. Hence, 256 ms always corresponds to an integer number of "refresh cycles", which is sufficient to evaluate the impact of refresh.

---

[9]In our evaluated system, each auto-refresh command causes 64 rows to be refreshed.

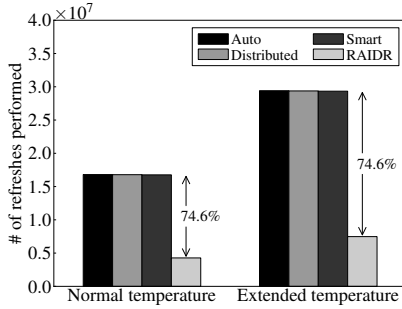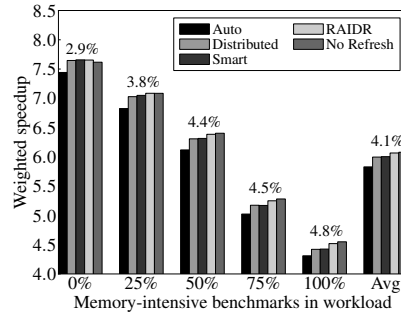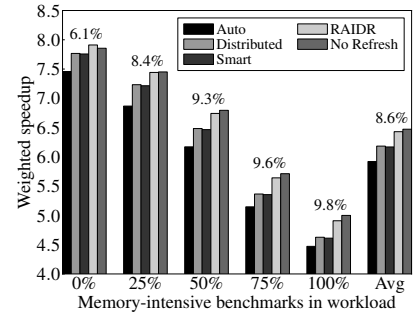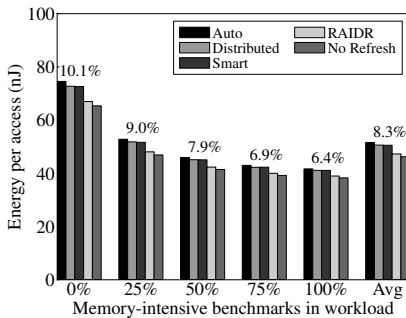Figure 6: Number of refreshes
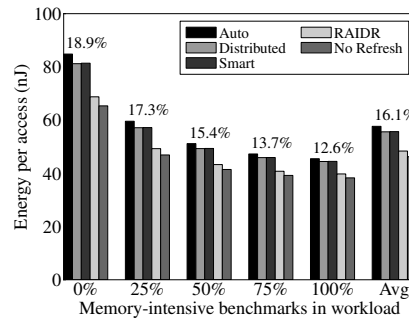

(a) Normal temperature range
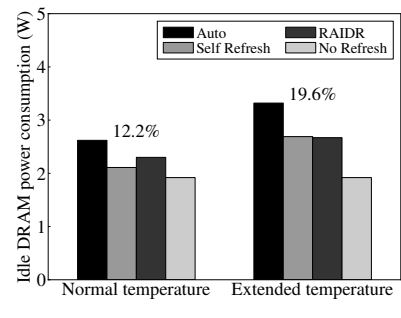
(b) Extended temperature range

Figure 7: Effect of refresh mechanism on performance (RAIDR improvement over auto-refresh in percent)

(a) Normal temperature range

(b) Extended temperature range

(c) Idle power consumption

Figure 8: Effect of refresh mechanism on energy consumption (RAIDR improvement over auto-refresh in percent)

6.1. Refresh Reduction

Figure 6 shows the number of refreshes performed by each mechanism.[10] A mechanism that refreshes each row every 256 ms instead of every 64 ms would reduce refreshes by 75% compared to the auto-refresh baseline. RAIDR provides a 74.6% refresh reduction, indicating that the number of refreshes performed more frequently than every 256 ms (including both rows requiring more frequent refreshes and rows that are refreshed more frequently due to false positives in the Bloom filters) is very low. The distributed refresh mechanism performs the same number of refreshes as the auto-refresh baseline. Smart Refresh does not substantially reduce the number of refreshes since the working sets of our workloads are small compared to the size of DRAM, and Smart Refresh can only eliminate refreshes to accessed rows.

6.2. Performance Analysis

Figure 7 compares the system performance of each refresh mechanism as memory intensity varies. RAIDR consistently provides significant performance gains in both the normal and extended temperature ranges, averaging a 4.1% (8.6%) improvement over auto-refresh.[11] Part of this performance improvement is a result of distributing refreshes, for the rea-

---

[10]For these results, we do not categorize workloads by memory intensity because the number of refreshes is identical in all cases for all mechanisms except for Smart Refresh, and very similar in all workloads for Smart Refresh. The no-refresh mechanism is omitted because it performs zero refreshes.

[11]This result, and further results, are given as "normal temperature (extended temperature)".

sons described in Section 6. However, RAIDR averages 1.2% (4.0%) performance improvement over distributed refresh, since reducing the number of refreshes reduces interference beyond what is possible through distributing refreshes alone. RAIDR's performance gains over auto-refresh increase with increasing memory intensity, to an average of 4.8% (9.8%) for workloads in the 100% memory intensity category. This is because increased memory intensity means there are a larger number of memory requests, so more requests encounter interference from refreshes.

Surprisingly, RAIDR outperforms the no-refresh system at low memory intensities. This unintuitive result occurs because while the common FR-FCFS memory scheduling policy maximizes memory throughput, it does not necessarily maximize system performance; applications with high row hit rates can starve applications with low row hit rates [34, 35]. However, refresh operations force rows to be closed, disrupting sequences of row hits and guaranteeing that the oldest memory request in the memory controller's request queue will be serviced. This alleviates starvation, thus providing better fairness. At low memory intensities, this fairness improvement outweighs the throughput and latency penalties caused by RAIDR's relatively infrequent refreshes.

6.3. Energy Analysis

We model the Bloom filters as a 1.25 KB direct-mapped cache with 64-bit line size, for ease of analysis using CACTI [48]. According to CACTI 5.3, for a 45 nm technology, such a cache requires 0.013 mm^2 area, consumes 0.98 mW standby leakage

power, and requires 3.05 pJ energy per access. We include this power consumption in our evaluations.

Figure 8 compares the energy per access for each refresh mechanism as memory intensity varies. RAIDR decreases energy per access by 8.3% (16.1%) on average compared to the auto-refresh baseline, and comes within 2.2% (4.6%) of the energy per access for no-refresh ideal. Despite the additional energy consumed by transmitting row addresses on the bus for RAS-only refresh in all mechanisms except for the baseline, all refresh mechanisms result in a net energy per access decrease compared to the auto-refresh baseline because the improvements in performance reduce the average static energy per memory access. The relative improvement for all mechanisms, including RAIDR, decreases asymptotically as memory intensity increases, since increased memory intensity results in increased DRAM dynamic power consumption, reducing the fraction of DRAM energy consumed by refresh.[12] Nevertheless, even for workloads in the 100% memory intensity category, RAIDR provides a 6.4% (12.6%) energy efficiency improvement over the baseline.

6.4. Idle Power Consumption

We compare three refresh mechanisms for situations where the memory system is idle (receives no requests).

- In the auto-refresh mechanism employed while idle, the DRAM is put in its lowest-power power-down mode [15], where all banks are closed and the DRAM's internal delay-locked loop (DLL) is turned off. In order to perform refreshes, the DRAM is woken up, an auto-refresh command is issued, and the DRAM is returned to the power-down mode when the refresh completes.
- In the self-refresh mechanism, the DRAM is put in its self-refresh mode [15], where the DRAM manages refreshes internally without any input from the memory controller.
- In RAIDR, the DRAM is put in its lowest-power power-down mode (as in the auto-refresh mechanism used while idle), except that the DRAM is woken up for RAIDR row refreshes rather than auto-refresh commands.

We do not examine an "idle distributed refresh" mechanism, since performance is not a concern during idle periods, and distributing refreshes would simply increase how frequently the DRAM would be woken up and waste energy transmitting row addresses. We also do not examine Smart Refresh, as it does not reduce idle power, as discussed in Section 4.2.

Figure 8c shows the system power consumption for each mechanism, as well as the no-refresh case for reference. Using RAIDR during long idle periods results in the lowest DRAM power usage in the extended temperature range (a 19.6% improvement over auto-refresh). The self-refresh mechanism has lower power consumption in the normal temperature range. This is for two reasons. First, in the self-refresh mechanism, no communication needs to occur between the memory controller

and the DRAM, saving I/O power. Second, in self-refresh, the DRAM internal clocking logic is disabled, reducing power consumption significantly. However, for the latter reason, when a DRAM device is woken up from self-refresh, there is a 512-cycle latency (768 ns in DDR3-1333) before any data can be read [15]. In contrast, a DRAM device waking up from the lowest-power power-down mode only incurs a 24 ns latency before data can be read [15]. This significant latency difference may make RAIDR the preferable refresh mechanism during idle periods in many systems. In addition, as refresh overhead increases (due to increased DRAM density or temperature), the energy saved by RAIDR due to fewer refreshes begins to outweigh the energy saved by self-refresh, as shown by RAIDR's lower power consumption in the extended temperature range. This suggests that RAIDR may become strictly better than self-refresh as DRAM devices increase in density.

6.5. Design Space Exploration

The number of bins and the size of the Bloom filters used to represent them are an implementation choice. We examined a variety of Bloom filter configurations, and found that in general RAIDR's performance effects were not sensitive to the configuration chosen. However, RAIDR's energy savings are affected by the configuration, since the chosen configuration affects how many refreshes are performed. Figure 9a shows how the number of refreshes RAIDR performs varies with the configurations shown in Table 3. The number of bins has the greatest effect on refresh reduction, since this determines the default refresh interval. The number of refreshes asymptotically decreases as the number of bits used to store each bin increases, since this reduces the false positive rate of the Bloom filters. As DRAM device capacities increase, it is likely worth using a larger number of bins to keep performance and energy degradation under control.

6.6. Scalability

The impact of refreshes is expected to continue to increase as DRAM device capacity increases. We evaluate how RAIDR scales with DRAM device capacity. We assume throughout that the amount of space allocated to RAIDR's Bloom filters scales linearly with the size of DRAM.[13] For these results we only evaluated the 32 workloads with 50% memory-intensive benchmarks, as this scenario of balanced memory-intensive and non-memory-intensive benchmarks is likely to be common in future systems [22]. We also focus on the extended-temperature range. Refresh times are assumed to scale approximately linearly with device density, as detailed in Section 2.2.

Figure 9b shows the effect of device capacity scaling on performance. As device capacity increases from 4 Gb to 64 Gb, the auto-refresh system's performance degrades by 63.7%, while RAIDR's performance degrades by 30.8%. At the 64 Gb device capacity, RAIDR's performance is 107.9% higher than the auto-refresh baseline. Figure 9c shows a sim-

---

[12]However, note that although we only evaluate the energy efficiency of the DRAM, the energy efficiency of the entire system also improves due to improved performance, and this energy efficiency gain increases with increased memory intensity since RAIDR's performance gains increase with increased memory intensity, as shown in Section 6.2.

[13]This seems to be a reasonable assumption; at the 64 Gb density, this would correspond to an overhead of only 20 KB to manage a 512 GB DRAM system.

Table 3: Tested RAIDR configurations

Key	Description	Storage Overhead
Auto	Auto-refresh	N/A
RAIDR	Default RAIDR: 2 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$)	1.25 KB
1 bin (1)	1 bin (64–128 ms, $m = 512$)	64 B
1 bin (2)	1 bin (64–128 ms, $m = 1024$)	128 B
2 bins (1)	2 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 2048$)	512 B
2 bins (2)	2 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 4096$)	768 B
2 bins (3)	2 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 16384$)	2.25 KB
2 bins (4)	2 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 32768$)	4.25 KB
3 bins (1)	3 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$; 256–512 ms, $m = 32768$)	5.25 KB
3 bins (2)	3 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$; 256–512 ms, $m = 65536$)	9.25 KB
3 bins (3)	3 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$; 256–512 ms, $m = 131072$)	17.25 KB
3 bins (4)	3 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$; 256–512 ms, $m = 262144$)	33.25 KB
3 bins (5)	3 bins (64–128 ms, $m = 2048$; 128–256 ms, $m = 8192$; 256–512 ms, $m = 524288$)	65.25 KB

(a) RAIDR configurations   (b) Performance scaling   (c) Energy scaling
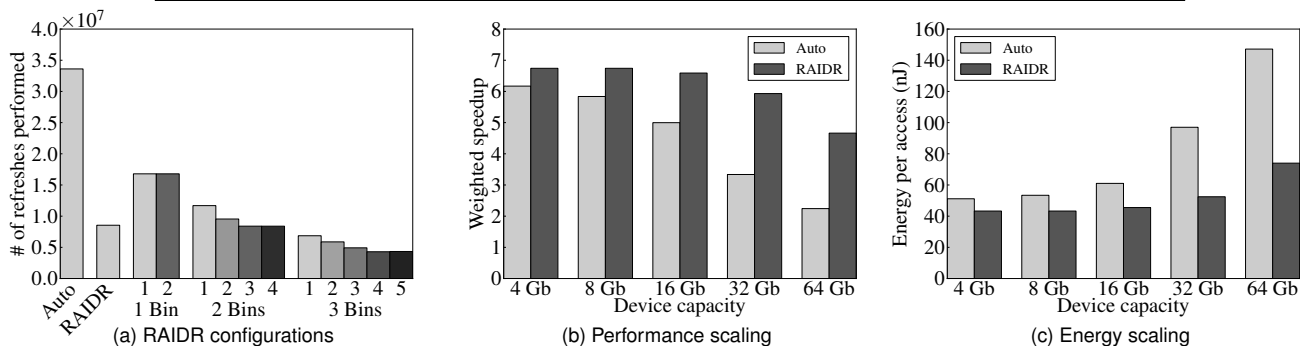
Figure 9: RAIDR sensitivity studies

ilar trend for the effect of device capacity scaling on energy. As device capacity scales from 4 Gb to 64 Gb, the auto-refresh system's access energy increases by 187.6%, while RAIDR's access energy increases by 71.0%. At the 64 Gb device capacity, RAIDR's access energy savings over the auto-refresh baseline is 49.7%. These results indicate that RAIDR scales well to future DRAM densities in terms of both energy and performance.

Although these densities may seem farfetched, these results are potentially immediately relevant to 3D die-stacked DRAMs. As discussed in Section 3.7, a 3D die-stacked DRAM is likely to operate in the extended temperature range, and its ability to parallelize refreshes to hide refresh overhead is limited by shared chip power. Therefore, a DRAM chip composed of multiple stacked dies is likely to suffer from the same throughput, latency, and energy problems caused by refresh as a single DRAM die with the same capacity operating at high temperatures. As a result, RAIDR may be applicable to 3D die-stacked DRAM devices in the near future.

6.7. Retention Error Sensitivity

As mentioned in Section 2.3, a DRAM cell's retention time is largely dependent on whether it is normal or leaky. Variations between DRAM manufacturing processes may affect the number of leaky cells in a device. We swept the fraction of leaky cells from 10^{-6} to 10^{-5}. Even with an order of magnitude increase in the number of leaky cells, RAIDR's performance improvement decreases by only 0.1%, and energy savings decreases by only 0.7%.

6.8. Future Trends in Retention Time Distribution

Kim and Lee [21] show that as DRAM scales to smaller technology nodes, both the normal and leaky parts of the retention time distribution will narrow, as shown in Figure 10. Since this would lead to a *decrease* in the proportion of very weak cells in an array, RAIDR should remain effective. To confirm this, we generated a set of retention times corresponding to the distribution in Figure 10b and confirmed that RAIDR's performance improvement and energy savings changed negligibly (i.e. by less than 0.1%).

7. Conclusion

We presented Retention-Aware Intelligent DRAM Refresh (RAIDR), a low-cost modification to the memory controller that reduces the energy and performance impact of DRAM refresh. RAIDR groups rows into bins depending on their required refresh rate, and applies a different refresh rate to each bin, decreasing the refresh rate for most rows while ensuring that rows with low retention times do not lose data. To our knowledge, RAIDR is the first work to propose a *low-cost* memory controller modification that reduces DRAM refresh operations by exploiting variability in DRAM cell retention times.

Our experimental evaluations show that RAIDR is effective in improving system performance and energy efficiency with modest overhead in the memory controller. RAIDR's flexible configurability makes it potentially applicable to a variety of systems, and its benefits increase as DRAM capacity increases.
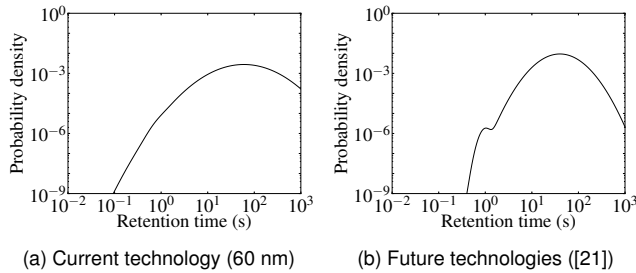
Figure 10: Trend in retention time distribution

We conclude that RAIDR can effectively mitigate the overhead of refresh operations in current and future DRAM systems.

Acknowledgments

References

[1] B. Black *et al.*, "Die stacking (3D) microarchitecture," in *MICRO-39*, 2006.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

[3] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *STOC-9*, 1977.

[4] Y. Chen, A. Kumar, and J. Xu, "A new design of Bloom filter for packet inspection speedup," in *GLOBECOM*, 2007.

[5] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications," *IEEE Micro*, 2008.

[6] P. G. Emma, W. R. Reohr, and L.-K. Wang, "Restore tracking system for DRAM," U.S. patent number 6389505, 2002.

[7] 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-40*, 2007.

[8] T. Hamamoto, S. Sugiura, and S. Sawada, "On the retention time distribution of dynamic random access memory (DRAM)," *IEEE Transactions on Electron Devices*, 1998.

[9] Hybrid Memory Cube Consortium, "Hybrid Memory Cube," 2011. Available: http://www.hybridmemorycube.org/

[10] Influent Corp., "Reducing server power consumption by 20% with pulsed air jet cooling," White paper, 2009.

[11] C. Isen and L. K. John, "ESKIMO: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *MICRO-42*, 2009.

[12] ITRS, "International Technology Roadmap for Semiconductors," 2010.

[13] JEDEC, "DDR SDRAM Specification," 2008.

[14] JEDEC, "DDR2 SDRAM Specification," 2009.

[15] JEDEC, "DDR3 SDRAM Specification," 2010.

[16] JEDEC, "LPDDR2 SDRAM Specification," 2010.

[17] Y. Katayama *et al.*, "Fault-tolerant refresh power reduction of DRAMs for quasi-nonvolatile data retention," in *DFT-14*, 1999.

[18] B. Keeth *et al.*, *DRAM Circuit Design: Fundamental and High-Speed Topics*. Wiley-Interscience, 2008.

[19] J. Kim and M. C. Papaefthymiou, "Dynamic memory design for low data-retention power," in *PATMOS-10*, 2000.

[20] J. Kim and M. C. Papaefthymiou, "Block-based multiperiod dynamic memory design for low data-retention power," *IEEE Transactions on VLSI Systems*, 2003.

[21] K. Kim and J. Lee, "A new investigation of data retention time in truly nanoscaled DRAMs," *IEEE Electron Device Letters*, 2009.

[22] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA-16*, 2010.

[23] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Addison-Wesley, 1998, vol. 3.

[24] W. Kong *et al.*, "Analysis of retention time distribution of embedded DRAM — a new method to characterize across-chip threshold voltage variation," in *ITC*, 2008.

[25] Y. Li *et al.*, "DRAM yield analysis and optimization by a statistical design approach," *IEEE Transactions on Circuits and Systems*, 2011.

[26] S. Liu *et al.*, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *ASPLOS-16*, 2011.

[27] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[28] M. J. Lyons and D. Brooks, "The design of a Bloom filter hardware accelerator for ultra low power systems," in *ISLPED-14*, 2009.

[29] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, 2003.

[30] Micron Technology, "Various methods of DRAM refresh," 1999.

[31] Micron Technology, "Calculating memory system power for DDR3," 2007.

[32] Micron Technology, "Power-saving features of mobile LPDRAM," 2009.

[33] Micron Technology, "4Gb: x4, x8, x16 DDR3 SDRAM," 2011.

[34] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.

[35] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO-40*, 2007.

[36] Y. Nakagome *et al.*, "The impact of data-line interference noise on DRAM scaling," *IEEE Journal of Solid-State Circuits*, 1988.

[37] T. Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM refresh count for merged DRAM/logic LSIs," in *ISLPED*, 1998.

[38] K. Patel *et al.*, "Energy-efficient value based selective refresh for embedded DRAMs," *Journal of Low Power Electronics*, 2006.

[39] L. A. Polka *et al.*, "Package technology to address the memory bandwidth challenge for tera-scale computing," *Intel Technology Journal*, 2007.

[40] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, 1997.

[41] S. Rixner *et al.*, "Memory access scheduling," in *ISCA-27*, 2000.

[42] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, 2011.

[43] B. Sinharoy *et al.*, "IBM POWER7 multicore server processor," *IBM Journal of Research and Development*, 2011.

[44] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS-9*, 2000.

[45] S. P. Song, "Method and system for selective DRAM refresh to reduce power consumption," U.S. patent number 6094705, 2000.

[46] Standard Performance Evaluation Corporation, "SPEC CPU2006," 2006. Available: http://www.spec.org/cpu2006/

[47] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO-43*, 2010.

[48] S. Thoziyoor *et al.*, "CACTI 5.1," HP Laboratories, Tech. Rep., 2008.

[49] Transaction Processing Performance Council, "TPC," 2011. Available: http://www.tpc.org/

[50] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *HPCA-12*, 2006.

[51] C. Wilkerson *et al.*, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ISCA-37*, 2010.

[52] K. Yanagisawa, "Semiconductor memory," U.S. patent number 4736344, 1988.

[53] H. Zheng *et al.*, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *MICRO-41*, 2008.

[54] W. K. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," U.S. patent number 5630096, 1997.