Refresh Now and Then

Seungjae Baek, *Member, IEEE,* Sangyeun Cho, *Senior Member, IEEE,* and Rami Melhem, *Fellow, IEEE*

Abstract—DRAM stores information in electric charge. Because DRAM cells lose stored charge over time due to leakage, they have to be "refreshed" in a periodic manner to retain the stored information. This refresh activity is a source of increased energy consumption as the DRAM density grows. It also incurs non-trivial performance loss due to the unavailability of memory arrays during refresh. This paper first presents a comprehensive measurement based characterization study of the cell-level data retention behavior of modern low-power DRAM chips. 99.7% of the cells could retain the stored information for longer than 1 second at a high temperature. This average cell retention behavior strongly indicates that we can deeply reduce the energy and performance penalty of DRAM refreshing with proper system support. The second part of this paper, accordingly, develops two practical techniques to reduce the frequency of DRAM refresh operations by excluding a few leaky memory cells from use and by skipping refreshing of unused DRAM regions. We have implemented the proposed techniques completely in the Linux OS for experimentation, and measured performance improvement of up to 17.2% with the refresh operation reduction of 93.8% on smartphone like low-power platforms.

Index Terms—SDRAM, refresh operation, power consumption, performance improvement.

1 Introduction

DRAM is commonly used in a computer system's main memory. Its low bit cost, excellent manufacturability and high performance have been unmatched by other memory types. DRAM cells have a simple structure with only one access transistor and one capacitor, allowing for high bit density and continued scaling with each new technology generation. In DRAM, information ('1' or '0') is stored as charge in the capacitor of a bit cell. To write '1' to a bit cell, charge is injected to the cell. To write '0', charge in the corresponding cell is drained.[1] Retrieval of the stored information from a bit cell is done by sensing the amount of charge shared between the cell and the pre-charged bit line.

The charge in a DRAM cell decays slowly over time via mechanisms like junction leakage and gate-induced drain leakage [1], [2]. The problem is that, if the charge in a cell decreases below a certain level, the previously stored information is lost (e.g., '1' was written and '0' is read). Accordingly, in order to retain the stored information, each DRAM cell has to be "refreshed" periodically. The process of refreshing is similar to a regular read operation. All stored data in a particular DRAM row (comprised of all bit cells tapping a common wordline) are sensed and fed back into the bit cells before the row is "closed" (to make the memory array ready for subsequent accesses). Because all DRAM cells have to be refreshed periodically, this process is repeated for all DRAM rows in sequence over and over again. Refresh period, the time interval within which we must walk through all rows once, is typically 32 ms or 64 ms for modern low-power DRAM implementations [3], [4].

The mandatory refresh activities come at both performance and energy costs. First, DRAM becomes unavailable when it is being refreshed, blocking any read or write operation during the refresh. Stuecheli et al. report the average performance degradation of over 10% due to refresh when running integer benchmarks [5]. The performance hit increases with the growing DRAM density because more rows have to be refreshed within the refresh period. Second, the proportion of DRAM energy consumption in a system can grow rapidly (40% or more), close to or even exceeding that of the CPU [6]. The impact of DRAM refresh is as large as over 25% of the DRAM energy consumption [7]. Notably, in systems where standby periods dominate the usage (e.g., smartphones), DRAM refreshing is a major consumer of the system energy [8].

Much prior work explored techniques to reduce the frequency and overhead of DRAM refreshing. For example, Flikker [9] slows down refresh operation for DRAM regions that store error-tolerant data. RAPID [10] allocates page frames with a long retention time first and aggressively extends the DRAM refresh period to match the shortest retention time of currently allocated pages. ESKIMO [11] exposes memory allocation and deallocation events to track DRAM regions that are not currently utilized and selectively skips refreshing. While these techniques were shown to be effective, they require programmer annotations [9], a non-trivial overhaul of OS-level memory management [10] and ISA modification [11].

In this work, we propose, implement and evaluate two software-oriented techniques to significantly reduce DRAM refresh activities without such requirements. Our first technique, *RIO* (Refresh Incessantly but Occasionally), utilizes

- *S. Baek and R. Melhem are with the Department of Computer Science, University of Pittsburgh, 210 S. Bouquet, Pittsburgh, PA 15260. E-mail: {baeksj, cho, melhem}@cs.pitt.edu*
- *S. Cho is with the Memory Division of Samsung Electronics Co. while on leave of absence from the University of Pittsburgh.*

1. Depending on the cell polarity assignment, the charged state may correspond to the logic value of '0'.

the page level data retention profile information to "delete" a very small number of weak pages (having relatively short retention time) and push the DRAM refresh rate to realize the near-maximum energy savings and performance improvement. Our second technique, *PARIS* (Placement-Aware Refresh In Situ), makes use of the free memory pool information within the OS to identify unused memory regions and skip refreshing for those regions. These techniques are completely transparent to the programmers and the applications, and do not require disruptive modifications to the DRAM controller and off-the-shelf DRAM. We make the following fundamental and applied contributions in this paper:

• **Characterizing the data retention time of modern low-power DRAM.** We comprehensively measure and characterize the data retention times of LPDDR (low-power double data rate) and LPDDR2 DRAM cells on a real hardware platform that resembles a high-end smartphone. The studied DRAM cells are found to retain the stored information for longer than 256 ms, eight times the specified refresh period of 32 ms at a normal temperature (45°C). Moreover, the majority of the DRAM cells (99.7%) show a retention time that is longer than 1 second, even at a high temperature (85°C). Cells showing short retention times are distributed randomly. We also find that there are a tiny fraction of DRAM cells that are relatively weak, motivating us to develop RIO; by identifying and excluding the pages that contain these weak cells, RIO can safely push the nominal refresh period by 2× to 16×.

• **Measuring the impact of DRAM refresh period.** We measure and characterize the impact of DRAM refresh on performance and power consumption as a function of refresh period. When the nominal (32 ms) and an 8× extended refresh period (256 ms) were used, the performance differentials were up to 8.9% and the refresh frequency is cut by 87.5%. We also found that practically, extending the refresh period beyond 256 ms gives very little additional benefit. At a high temperature (85°C and higher), the DRAM chips must be refreshed at 8 ms according to the LPDDR/LPDDR2 specification, and the performance differential between this period and 64 ms (8× the required refresh period) widens to 17.2% (maximum).

• **Design, implementation and evaluation of RIO and PARIS.** Based on the above empirical study, we motivate RIO and PARIS, as well as their specific design strategies. Core ideas of RIO and PARIS are simple enough for realization in common OS'es. To prove their concepts, we fully implement both RIO and PARIS in a Linux kernel and evaluate them on the low-power hardware platforms we used in the characterization study. Furthermore, our system implementation includes a completely flexible software DRAM refresher called *pseudo-refresher* that runs on a separate processor core of the studied hardware platform. Pseudo-refresher allows us to study the impact of PARIS without a real hardware. Employing RIO and PARIS, we measured the refresh reduction of 93.8% under medium memory usage (50%).

In the remainder of this paper, we will first discuss how the DRAM refresh interface has evolved and review prior related work in Section 2. Section 3 will then present a detailed characterization study of DRAM cell retention behavior. Based on the findings, Section 4 will introduce RIO and PARIS and describe in detail how we implement them in the Linux kernel. We evaluate RIO and PARIS on a real hardware platform in Section 5. Finally, Section 6 will conclude.

2 Background

2.1 Evolution of DRAM refresh interface

DRAM's commoditization has been promoted primarily by standards [14]. Because refreshing is essential for DRAM, the interface for it—how DRAM must be refreshed—has been an integral part of all DRAM standards.

For early DRAM designs like EDO-DRAM (e.g., Micron MT4LC4M4E8 [15]), the DRAM controller dictated a specific row address to refresh as part of the refresh command (sometimes called "RAS-only" refresh or "ROR"). This operation is similar to a read operation, but with no data transfer. In newer DRAM standards like synchronous DRAM (SDRAM), "auto-refresh" has been introduced and used. With auto-refresh, the DRAM controller no longer needs to specify the row address when it issues refresh; the DRAM has an internal counter that is incremented with each refresh command, and the row pointed to by the counter (in all available internal banks) is "automatically" selected and refreshed.

The "self refresh" mode is provided in virtually all DRAM designs to suppress static power consumption when a DRAM is not actively accessed (e.g., the system is in an idle mode). Once a DRAM enters the self refresh mode, the majority of its interface pins and internal circuitry are shut down. With no explicit refresh commands from the DRAM controller, the self refresh circuit in the DRAM chip initiates refresh actions.

Recent low-power DRAM standards like LPDDR and LPDDR2 [14] have additional provisions for optimizing refresh behavior of a DRAM chip. First of all, *partial array self refresh* (PASR) allows the system to define the partial DRAM capacity (e.g., $1/2$, $1/4$, $1/8$, $1/16$) that must be refreshed in the self refresh mode. Moreover, as the DRAM chip is "isolated" in the self refresh mode, LPDDR/LPDDR2 DRAM designs incorporate a temperature sensor so that the self refresh circuit can automatically tune the refresh rate based on periodic temperature sensing.[2] At a high temperature, the self refresh circuit refreshes the DRAM arrays more frequently (e.g., 4× at 85°C or higher). In turn, at a low temperature, a slower refresh rate could be used (than the nominal rate). Finally, certain LPDDR2 DRAMs have a separate auto-refresh command for all banks or a single bank. The single bank auto-refresh incurs smaller peak current during refresh operation.

2. DRAM cell leakage mechanisms are sensitive to temperature. Typically, cell leakage current grows super-linearly with the increase in temperature. In an active mode, the DRAM controller has to dynamically change the rate of auto-refresh commands according to the temperature.

2.2 Prior work on reducing DRAM refresh activities

Existing work on reducing the frequency and overheads of DRAM refreshing is motivated by three observations. First, DRAM cell retention time distribution is normal [2], [10], [16], i.e., most DRAM cells have a longer retention time than weak cells on the left tail. This observation has inspired much prior work [9], [10], [13], [17], [18]. Second, it is unnecessary to refresh a DRAM row if it holds no useful data [11], [19]. Lastly, regular DRAM accesses make some refresh operation redundant [7], [20]. For example, if a row is read by the CPU, the same row does not need immediate refreshing. Let us discuss in detail only the most recent related work to ours.

The "RAPID" work by Venkatesan et al. [10] builds on the first two observations. RAPID-1 deletes a small number (1%) of page frames containing weak cells and extends the refresh period (to 3 seconds at room temperature and 1 second at 70°C). RAPID-2 and RAPID-3 classify page frames into sorted bins based on their retention time and allocate page frames from the last non-empty bin. They set the DRAM refresh period to match the shortest retention time of all allocated bins.

The observation on which both RAPID-1 and RIO proposed in this work build on is essentially identical and not new but shares the spirit of the predated work [17], [18]. However, this paper makes a major contribution that a real system can be built based on carefully made design decisions. The RAPID-1 work is not implemented in a real system, and hence did not identify real design problems such as how to determine margins, what are ingredients for fast testing and how to determine cells to discard. Also, an important implication of our work is that RAPID-2 and RAPID-3 are neither realizable (will be discussed in Section 4.1.2), nor warranted (will be discussed in Section 3.2). Our close examination of modern LPDDR/LPDDR2 chips guides us to make very different design trade-offs. Importantly, RIO deletes far fewer pages (less than 0.1% vs. 1% of RAPID-1) to avoid excessive DRAM fragmentation and pushes the refresh period just enough (e.g., 256 ms) to obtain near-maximum benefits. PARIS is aware of a system's DRAM usage like RAPID-2 and 3. However, PARIS does not require binning of all page frames and does not alter the refresh period according to DRAM usage changes, significantly reducing the related design complexities. Both RIO and PARIS have been implemented in Linux and are evaluated on a real platform.

The "RAIDR" work by Liu et al. [13] also builds on the first observation. Like RAPID-2/-3, RAIDR classifies page frames into a few sorted bins based on their retention time. Unlike RAPID-2/-3 that maintain a single refresh period across all DRAM rows at a given time, however, RAIDR applies different refresh periods to DRAM rows that belong to different bins. RAIDR represents the bin information space-efficiently using Bloom filters and implements multiple refresh rates with clever use of several counters. In comparison, RIO simply takes the weakest page frames offline in the OS and increases the refresh period, achieving near-maximum reduction of refresh overheads while maintaining just one refresh period. RAIDR is orthogonal to PARIS and can be used together.

Isen and John utilize the second observation in their ESKIMO scheme [11]. They add to the CPU architecture two new instructions to track memory allocation and deallocation events in user applications. ESKIMO assumes there is a flag per DRAM row to denote whether the row has to be refreshed or not [19].

PARIS proposed in this paper aims at eliminating refresh actions targeting unused DRAM rows like ESKIMO. However, PARIS is very different from ESKIMO in terms of the overall approach and the actual design. While the ESKIMO paper presents a conceptual design, it does not discuss in detail the possible complexities of implementation. Adding new CPU instructions to provide user-level memory usage information to a hardware structure within a CPU, is not trivial. For example, the instructions specify an address range in the virtual address space; the address range then has to be translated into a set of physical pages (before further translated down to DRAM rows), which involves accessing the translation look-aside buffer (TLB) potentially multiple times. Furthermore, events like page swapping and page sharing in a system require saving and retrieving virtual page level information across the events, leading to serious complexities to the design of hardware and the OS. In comparison, PARIS directly and conveniently extracts global physical memory usage information from within the OS and requires no intrusive changes to the CPU and applications. The global memory usage information of PARIS, including the usage of the OS and hardware accelerators, gives a precise snapshot of the DRAM usage. Even if ESKIMO offers more tight view of memory usage in the virtual address space, it is the consecutive physical pages that matters when we identify free or used DRAM rows because DRAM rows are typically a multiple of page frames. The gains with fine-grained information in the virtual address space are believed to be less than substantial to justify the high complexities of implementation.

Flikker builds on the first observation and a principle similar to the second observation ("it is unnecessary to refresh a DRAM row fast if the data it holds is not critical") [9]. It partitions the DRAM space into "normal refresh area" and "low refresh area". The low refresh area is for storing non-critical, error-tolerable data like mpeg movie. As such, Flikker requires modifications to both user applications (especially the data retention model at the programming language level) and the DRAM design. RIO and PARIS do not expose the low-level DRAM refresh management tasks to the programmer by hiding them within the OS and do not depend on special support from commodity DRAMs.

Smart Refresh [7], according to the third observation, skips refreshing a DRAM row if it was accessed recently. Refresh skipping is effective when the DRAM is actively utilized. Smart Refresh is orthogonal to RIO and PARIS

and they can be employed together.

3 Limits of DRAM Refreshing

In this section, we present a detailed characterization study of cell retention behavior for samples of two low-power DRAM types, LPDDR and LPDDR2.

3.1 Approach and experimental setup

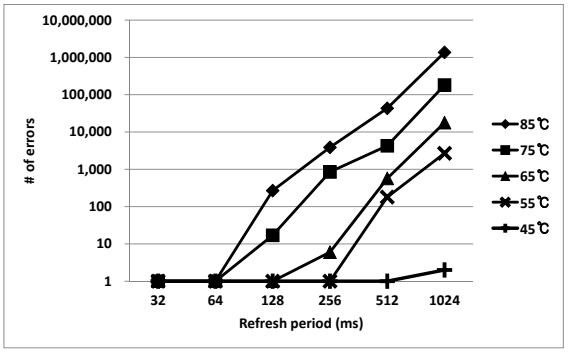
We use a measurement method to collect retention times of DRAM cells, similar to [10], [16]. We employ two different hardware platforms in this study: The first platform, BeagleBoard [21] ("Beagle" hereafter) has a DM3730 CPU with a single 1 GHz ARM Cortex-A8 core, a Micron 512 MB LPDDR DRAM (package-on-packaged (POP'ed) on the CPU, running at 200 MHz) and a variety of peripheral devices like NIC, USB, HDMI and SD card. The second platform is Pandaboard [22] ("Panda") and is comprised of an OMAP4430 CPU with dual 1 GHz ARM Cortex-A9 cores, an Elpida 1 GB LPDDR2 DRAM (four 2 Gb parts POP'ed on OMAP4430, running at 400 MHz) and peripherals. These platforms are fairly comparable to recent smartphone releases with a single or dual-core processors. We have ported and run Linux kernel 2.6.32 on both Beagle and Panda. To measure power consumption, we use a Rigol DM3058 digital multimeter.

Next, we describe the experimental procedure. After booting up the platforms, we run a test code from an on-chip SRAM. The code first initializes DRAM and test it according to the following steps:

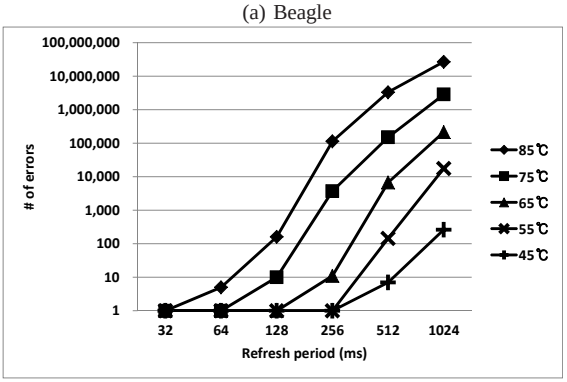
1. Set DRAM refresh period to T_{ref}.
2. Write value (val_w) into the whole DRAM space.
3. Wait for T_{wait}.
4. Read out value (val_r) from DRAM at all addresses.
5. If ($val_w \neq val_r$) for an address, print the address.

The focus of this study is to identify and characterize the weak cells that determine the DRAM's refresh period, rather than obtaining the full retention time distribution of the examined DRAM chips (there are such studies [2], [10], [16]). Accordingly, we use 32 ms (the nominal period) to 1,024 ms (the maximum value allowed by the DRAM controller) for T_{ref}. We use two different values for val_w, `0x00000000` and `0xffffffff`, because a DRAM chip may use two polarity assignments (charged state corresponds to the logic value '0' or '1'). T_{wait} is set to $\max(10 \text{ seconds}, 10 \times T_{ref})$ to ensure that weak cells have sufficient time to be manifested.

As DRAM cell retention times are sensitive to ambient temperature, we repeat the above test at different ambient temperatures—45°C (labeled "Normal"), 55°C, 65°C, 75°C and 85°C (labeled "High") using an in-house chamber. It is important to consider a wide range of temperature, especially with the low-power DRAM. This is because mobile devices are exposed to widely varying operating conditions (e.g., a smartphone on the dashboard of a car in Texas on a sunny summer day). Moreover, low-power DRAMs are typically packaged on top of the CPU chip, and mobile devices have limited cooling capacity.



(a) Beagle



(b) Panda

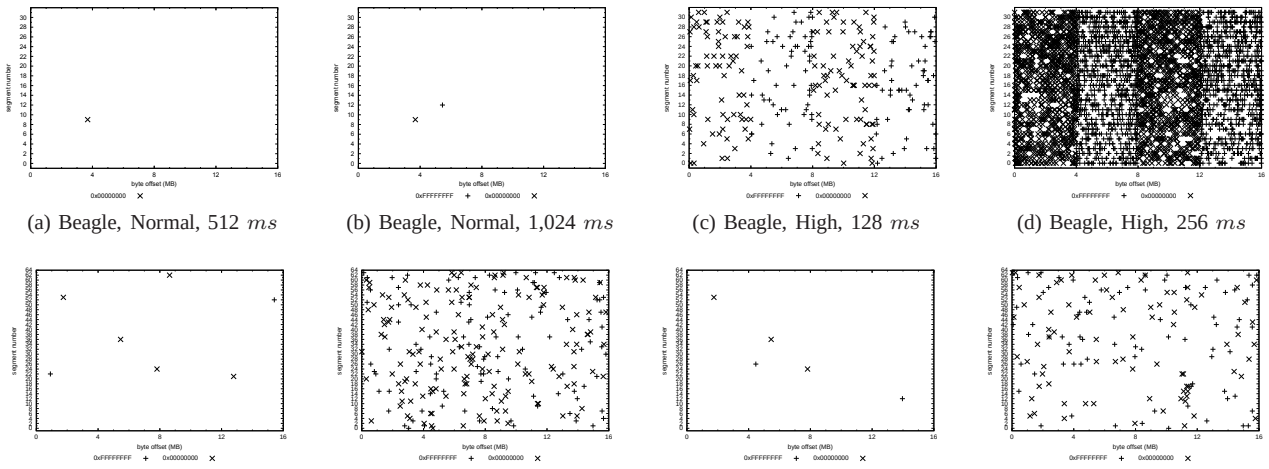
Fig. 1: Number of cell error occurrences vs. refresh period.

3.2 Results

Retention errors as a function of refresh period. Figure 1 shows how many DRAM cell errors occur under five different ambient temperatures as we vary the refresh period. In our result, a cell error manifested at refresh period T_{ref} implies that the cell has a retention time between $(T_{ref}/2 + \epsilon)$ and T_{ref} where ϵ is a small value. The same result is tabulated in Table 1 with additional details. We make several observations from the result.

First of all, the DRAM cells in the studied chips are very sensitive to ambient temperature. We saw only 2 and 264 errors on Beagle and Panda each at $T_{ref} = 1,024\ ms$ and 45°C but the error count climbs rapidly to over one million and ten millions at 85°C. Once errors start to occur, *they multiply by over 10× when we double the refresh period*. We believe that the large variation of error counts against temperature change presents the most important practical issue to consider for any DRAM refresh reduction scheme. Clearly, *DRAM refresh period must be adapted quickly to the ambient temperature change*. Moreover, *for safe system operation, we need to systematically reserve a sufficiently large margin between the target refresh period and the weakest DRAM cell at all temperatures of operation*.

Second, there are only a small number of cells that have a retention time shorter than 512 ms at 45°C to 65°C. For example, on Beagle, there are less than 10 errors at 256 ms. Even at 85°C, there are no errors on Beagle until 64 ms and as few as 5 on Panda. We identify both an opportunity and a limitation from the result: *One may mask off only a few cells to substantially extend the refresh period*. However, *one must not extend the refresh period too much*, say, to



(a) Beagle, Normal, 512 ms (b) Beagle, Normal, 1,024 ms (c) Beagle, High, 128 ms (d) Beagle, High, 256 ms

(e) Panda, Normal, 512 ms (f) Panda, Normal, 1,024 ms (g) Panda, High, 64 ms (h) Panda, High, 128 ms

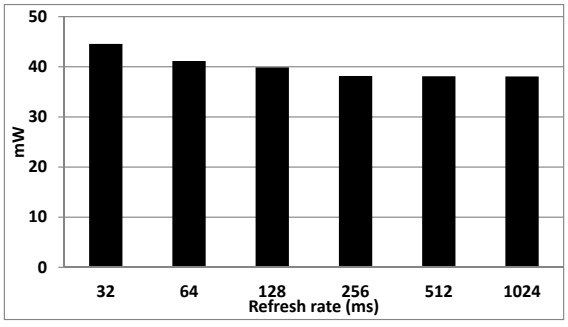
Fig. 2: Spatial distributions of cell errors.

512 ms, especially at an elevated temperature.

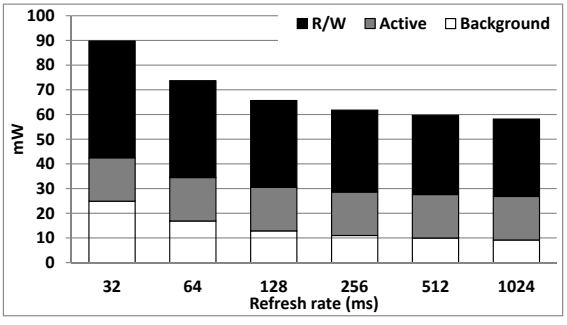
Lastly, Panda was much more sensitive to temperature than Beagle. The relative number of errors (against the total number of cells) at 1,024 ms was an order of magnitude higher on Panda. There is a large difference in the overall cell-level retention behavior across chips (we also saw non-trivial variation between different Panda boards) suggests that *we must test systems individually for safe and effective adaptive DRAM refresh reduction*.

Spatial distributions of weak cells. Figure 2 presents two snapshots of DRAM cell error locations (in the physical address space) on Beagle and Panda at Normal (45°C) and High (85°C). In the plot, a '+' mark is an error detected when val_w = `0x00000000`, while a '×' mark corresponds to an error detected when val_w = `0xffffffff`. The first snapshot is taken at a refresh period value when we start to see errors (e.g., 512 ms on Beagle at Normal) and the second snapshot is taken at twice the refresh period (e.g., 1,024 ms).

We make interesting observations from the result. First, *cell errors are distributed randomly in the address space*. The errors do not appear to be spatially clustered on both Beagle and Panda. This observation offers significant insight into *how the choice of weak cell masking granularity leads to different amounts of wasted space in memory*. Second, the location of the first detected error was different at different temperatures on a chip. For example, on Panda, there are errors that are found at 64 ms and High (Figure 2e) that do not appear at 512 ms and Normal (Figure 2g), and vice versa. Our result strongly suggests that *DRAM cells respond to temperature variation differently* and *data from DRAM cell retention time analysis at a specific temperature may not apply to other temperatures*. Still, all cell level retention behavior instances were repeatedly and stably observed under the same condition. Finally, interestingly, the cell polarity assignment pattern was visible and regular on Beagle (boundaries were at $(4\,\text{MB}\times i)$ and $(4\,\text{MB}\times(i+1)-1)$). On Panda, however, our initial inspection failed to identify visible patterns. In any case, the reason for different polarity assignment is believed to be driven by circuit- and layout-level optimization and does not affect design trade-offs for RIO and PARIS.



(a) Measured result on Beagle



(b) Estimated result with Micron Power Calculator

Fig. 3: Power consumption vs. refresh period.

Power consumption as a function of refresh period. We measure and compare the power consumption of Beagle at different refresh periods. We only report Beagle's result because it provides external pins for reliable power measurements near the power management chip while Panda doesn't.

Figure 3a plots our measurement result. We measured the power consumption of the entire Beagle board as we vary the refresh period. Measuring DRAM power separately is hard on Beagle because the DRAM chip is within the processor chip package. As expected, the system power de-

TABLE 1: Number of weak cells and number of bits/bytes/page frames (in %) to cover all weak cells for each (refresh period, temperature). Shaded configurations have to list more than 0.1% of all page frames to cover all weak cells.

(a) Beagle

	32 ms	64 ms	128 ms	256 ms	512 ms	1,024 ms
45°C	# errors % bits % bytes % pages	0	0	0	1 2.32E-8% 1.86E-9% 7.62E-4%	2 4.65E-8% 3.72E-7% 1.52E-3%
55°C	0	0	0	0	180 4.19E-6% 3.35E-5% 1.37E-1%	2,683 6.24E-5% 4.99E-4% 1.74%
65°C	0	0	0	6 1.39E-7% 1.11E-6% 4.57E-3%	570 1.32E-5% 1.06E-4% 4.34E-1%	17,827 4.15E-4% 3.32E-3% 10.89%
75°C	0	0	17 3.95E-7% 3.16E-6% 1.29E-2%	851 1.98E-5% 1.58E-4% 6.49E-1%	4,276 9.95E-5% 7.96E-4% 2.82%	181,290 4.22E-3% 3.37E-2% 68.40%
85°C	0	0	268 6.23E-6% 4.99E-5% 2.04E-1%	3,878 9.02E-5% 7.22E-4% 2.96%	43,235 1.01E-3% 8.05E-3% 24.31%	1,366,408 3.18E-2% 2.54E-1% 99.98%

(b) Panda

	32 ms	64 ms	128 ms	256 ms	512 ms	1,024 ms
45°C	# errors % bits % bytes % pages	0	0	0	7 8.14E-8% 6.51E-7% 2.67E-3%	264 3.07E-6% 2.45E-5% 1.01E-1%
55°C	0	0	0	0	144 1.67E-6% 1.34E-5% 5.34E-2%	17,707 2.06E-4% 1.64E-3% 6.19%
65°C	0	0	0	11 1.28E-7% 1.02E-6 4.19E-3%	6,736 7.84E-5% 6.27E-4% 2.36%	216,544 2.52E-3% 2.01E-2% 56.41%
75°C	0	0	10 1.16E-7% 9.31E-7% 3.81E-3%	3,730 4.34E-5% 3.47E-4% 1.31%	150,334 1.75E-3% 1.40E-2% 41.88%	2,862,698 3.33E-2% 2.66E-1% 99.99%
85°C	0	5 5.82E-8% 4.65E-7% 1.90E-3%	161 1.87E-6% 1.49E-5% 6.14E-2%	114,183 1.32E-3% 1.06E-2% 33.15%	3,337,103 4.88E-2% 3.20E-1% 99.99%	27,022,708 3.14E-1% 2.51% 100%

creases as we increase the refresh period. Power difference was measurable until 256 ms but the difference became insignificant after that, implying that the power contribution by the refresh operation itself becomes negligibly small when the refresh period reaches 256 ms. The power difference between 256 ms and 1,024 ms was less than 1% of the difference between 32 ms and 256 ms. We obtain independent results showing a similar trend using Micron Power Calculator [23], presented in Figure 3b. In the plot, refresh power is captured in the "background power", the bottom stack of each bar. Again, this result shows that additional power savings are diminishing rapidly as we increase the refresh period past 256 ms.

Our results reveal two insights about the benefit of reducing DRAM refresh operations. First, the power savings with aggressive DRAM refresh reduction has an upper bound (e.g., 7 to 15 mW per 512 MB in the above study) and *extending the refresh period to 256 ms will achieve 90% or more of the maximum power savings*. This observation implies that techniques to very aggressively extend a DRAM refresh period (to several seconds) at non-trivial implementation complexities like RAPID-2/-3 [10] and Flikker [9] are hard to justify in a practical sense.

Second, in turn, the relative power savings of refresh reduction will be high in DRAM's self refresh mode because there is no memory access in this mode and most external and internal circuits are shut down (i.e., the two upper stacks are goine in Figure 3b). Hence, the information we collect and use in RIO and PARIS will be extremely useful to control self refresh power. Pursuing this direction requires major changes in the DRAM interface, however, and remains a future work.

Putting it all together—cost of masking weak cells. Our findings corroborate previous studies [2], [10], [16]—*average DRAM cells have much longer retention times than the nominal refresh period and there are a few cells having much shorter retention times*. Clearly, the result motivates a better-than-worst design approach. In order to summarize our findings and motivate specific design strategies taken by RIO, let us consider the cost of masking weak cells to achieve a long refresh period. Table 1 presents the number of weak cells detected under different conditions and the amount of bits, bytes and page frames required to cover the weak cells on Beagle and Panda.

Under the worst condition of 1,024 ms and 85°C, the weak cell population corresponds to 0.0318% (Beagle) and 0.314% (Panda) of all DRAM cells. As was pointed out, this ratio is small. However, masking them using an OS-manageable granularity of page frame leaves little usable memory space on both platforms. Weak cells' spatial distribution was shown to be random, implying that *the cost of weak cell masking increases (almost) linearly with weak cell occurrences*. RIO has to make a right trade-off between achievable refresh period and wasted resources to mask weak cells.

In fact, we find that a more pressing issue is *memory fragmentation* that occurs when we mask off weak cells. The random nature of weak cell location implies that *it becomes exponentially harder to find consecutive memory space when more weak cells are introduced*, critically hurting the feasibility of weak cell masking from the software design perspective. We will discuss this issue in further detail in Section 4.1.2.

4 Refresh Now and Then in RIO and PARIS

4.1 RIO

RIO (Refresh Incessantly but Occasionally) *logically deletes* page frames that have a weak DRAM cell ("weak page frames") to push the refresh period to a larger value than required at a given temperature. A simple way of deleting weak page frames is to exclude them permanently when booting the system. The main trade-off in RIO is the number of page frames deleted (wasted resources) and the increase in the refresh period (improvement in energy and

TABLE 2: Safe refresh period at different temperatures with RIO.

(a) Beagle

Temperature	Maximum refresh period	
	without RIO	with RIO
(0) ∼55°C	32 ms	512 ms
56 ∼65°C	32 ms	128 ms
66 ∼75°C	32 ms	128 ms
76 ∼85°C	16 ms	64 ms
85 ∼(90°C)	8 ms	32 ms

(b) Panda

Temperature	Maximum refresh period	
	without RIO	with RIO
(0) ∼55°C	32 ms	256 ms
56 ∼65°C	32 ms	256 ms
66 ∼75°C	32 ms	128 ms
76 ∼85°C	16 ms	64 ms
85 ∼ (90°C)	8 ms	64 ms

performance). For example, Table 1 suggests that if RIO deletes 17,707 page frames in Panda (6.19% of all page frames), it may push the refresh period to 512 ms or more at 55°C. However, according to Figure 3, additional gain in energy and performance is diminishing beyond 256 ms. Clearly, pushing the refresh period beyond 256 ms (at high cost) is not cost-effective. Therefore, *RIO statically deletes a very small fixed set of weak page frames to achieve near maximum performance and energy gains*. Also, *RIO dynamically adapts refresh period to ambient temperature by exploiting on-chip temperature sensors*. In what follows, we will discuss how we test a DRAM to identify weak cells, how RIO utilizes the weak cell information to adjust the refresh period at run time and how we implemented RIO in the Linux OS.

4.1.1 Identifying weak cells

Our limit study revealed that testing DRAM cells at a single temperature is not sufficient for RIO to reliably operate. However, invoking DRAM testing at different temperatures is hard once a system has been deployed. Therefore, practically, a system DRAM must be tested before the system leaves the factory, but after the DRAM is soldered to the system board. This is the most feasible scenario since: (1) it is costly for the DRAM manufacturer to communicate the individual cell retention behavior information to the OS that will run on the system, even if the manufacturer collects such information; (2) DRAM cell retention behavior changes due to heat applied during soldering; and (3) all new systems have to go through software based factory testing anyway (e.g., checking component functionality (including DRAM)).

The actual DRAM test process for RIO is similar to the test procedure described in Section 3.1. For practical reasons, it is important to reduce the test time while not compromising the precision of weak cell identification. Since we focus only on the weak cells that determine the achievable refresh period, our test time is significantly shorter than a method that considers the whole span of refresh periods. For example, we were able to reduce the test time to ∼20 seconds via program optimizations on Beagle (i.e., 512 MB DRAM) for a given refresh rate and temperature, which multiplies to 120 seconds if we sweep through six periods (e.g., Table 1). By comparison, testing a 16 MB DRAM for the whole retention time distribution span took more than 22.3 minutes [10]. The actual required test time is longer because test has to repeat at different temperatures.

Because RIO does not "over-extend" the refresh period, there are opportunities to reduce the test time. For example, among the 30 configurations in Table 1, six configurations that bud from the "worst" configuration (1,024 ms and 85°C) as well as the last column (1,024 ms) do not need to be tested. Four configurations from the top in the first column do not need to be considered (for retention test) according to the LPDDR/LPDDR2 specification. Other configurations could be visited systematically to find relevant "boundaries" (e.g., using binary search). While designing a full-fledged factory testing process is beyond the scope of this paper, the common patterns revealed in our study will help determine an order of visitation that minimizes test time.

4.1.2 Translating raw DRAM test information

The proposed DRAM test identifies weak cells—their location and error manifestation condition (i.e., refresh period and temperature). For use in RIO, the raw information is translated into a set of temperature-refresh period mappings that prescribe the maximum refresh period RIO may use at a given temperature. The mappings are then written to a persistent storage like flash memory on the system for RIO to pick up. Table 2 presents an example temperature–refresh period mapping for Beagle and Panda, derived from Table 1.

The key question in this step is, *how do we determine the maximum refresh period target for each temperature?* We have already identified a (weak) constraint that going further than 256 ms gives diminishing returns. There are two other critical constraints we must consider before addressing the above question adequately. The first one is *safety*—we must not risk losing data by extending the refresh period too aggressively. The second one is *feasibility*—we must not delete too many page frames that will leave the physical memory excessively fragmented. Let us explain each separately.

Safety. Manufacturers design DRAM cells to retain information much longer than the nominal, recommended refresh period of, say, 32 ms. This practice guarantees that there is a sufficiently large "guard band" between weakest cells' retention time and the nominal refresh period in spite of process variation. Table 1 shows that there is at least a 2× gap at 75°C (64 ms vs. 32 ms). At 85°C, we start to have weak cells at 64 ms, and more will likely be manifested at a (slightly) higher temperature, even at a short refresh period of 32 ms. Not surprisingly, LPDDR/LPDDR2 specifications require that a DRAM chip be refreshed at 8 ms at 85°C or higher [14]. Based on the observation, we sustain that RIO must honor *a guard band*

of at least 2× for safety which is the same gap maintained by the manufacturer. For example, with no deletion of page frames, RIO may refresh Beagle at 128 ms at below 55°C. With deletion, RIO can extend refresh period further, as Table 2 suggests, which was constructed assuming the 2× guard band.

Feasibility. Our limit study showed that a small number of weak cells may correspond to a large amount of wasted page frames. Because RIO masks errors at page frame granularity, *how many page frames do we delete?* is more relevant than *how many bits?* To obtain good feasibility, we determine that RIO may delete *fewer than 0.1% of all page frames*. This may sound extremely conservative; however, consider the probability of having a 4 MB contiguous memory space (i.e., the largest memory chunk required in Linux) from a "strong" page frame picked up randomly. This probability is *well less than 1%* even if we delete only 0.5% of all page frames ($0.995^{1,024}$ assuming a page frame is 4 KB), because weak cells are distributed randomly in the memory space. Importantly, many low-level software like OS kernel and device drivers require a contiguous physical memory space (e.g., kernel image, network buffer). Indeed, we experienced frequent system crashes during and after boot if we delete more than 0.1% of page frames. To summarize, RIO determines the maximum refresh period at each temperature based on a small number of page frames it deletes (under the feasibility constraint) and after accounting for at least a 2× guard band.

4.1.3 Implementation

We have implemented RIO in the Linux OS for evaluation. Thanks to the simple design choices made for RIO, we expect the RIO algorithms and our implementation strategies to be easily adapted to other OS'es with virtual memory support. Table 3 lists the Linux source files that have been modified and summarizes our modifications. We modified only two files to be able to adjust the default refresh period and another file to permanently delete weak page frames at boot time. In our current implementation, DRAM test data are organized in a format similar to Table 1.

RIO (and all other DRAM refresh reduction optimizations) must dynamically adjust to changes in ambient temperature. It is worth discussing two practical design issues with regard to temperature adaptation: *accurate and timely temperature sensing* and *safe changing of refresh period at run time*.

Temperature sensing is rather straightforward in today's platforms. For example, we have at least two temperature sensing sources on Beagle and Panda: a DRAM-internal sensor (LPDDR2) and a sensor inside the CPU chip. The LPDDR2 sensor can be read through a special DRAM command. The on-CPU sensor is mapped to a memory address. RIO may obtain temperature when a temperature sensor triggers an interrupt. Periodic polling also works well for RIO because the meaningful delta of temperature change is typically in the hundreds of millisecond. Because not all available temperature sensors throw an interrupt, our current implementation uses polling. The reading involves only five hardware register read/write operations.

TABLE 3: List of modified files in Linux (Beagle).

Modified Files	Descriptions
`arch/arm/mach-omap2/board-omap3beagle.c`, `sdram-micron-mt46h32m32lf-6.h`	• Adjust refresh period
`arch/arm/mm/mmu.c`	• Delete page frames that include a weak cell using $\_\_alloc\_bootmem\_low$ function • Create a kernel thread for periodic temperature sensing

Efficient temperature sensing facilitates timely refresh period adaptation by detecting the direction and the speed of temperature changes. We track temperature changes by keeping past sixteen temperature sensor readings. When switching to a different refresh period (e.g., according to Table 2), RIO must exercise conservative policies to determine the triggering point. For example, to switch to a longer refresh period (i.e., temperature is cooling), RIO may wait until the temperature is well within the new, low temperature range. On the other hand, to switch to a shorter refresh period (i.e., temperature is rising), RIO could change the refresh period early, prior to reaching the upper boundary of the current range. The above policies may keep the refresh period unnecessarily small for some time. However, it is important to refresh all (weak) cells frequently enough at a small region of temperature near each boundary of temperature ranges.[3] Conservative reaction to temperature change also helps avoid potentially frequent shifts of the refresh period at the boundaries of temperature ranges.

4.2 PARIS

PARIS (Placement-Aware Refresh In Situ) *logically excludes* DRAM rows that do not currently store useful data from being refreshed. PARIS identifies these "empty DRAM rows" using the OS level information on page frame usage with no programmer and application intervention. With the help of PARIS, a platform *selectively refreshes* DRAM rows on demand, achieving a substantial reduction of refresh activities when DRAM capacity is not fully utilized. We will first discuss *how we extract empty DRAM row information from Linux*. We will then describe *a desirable architectural support* for PARIS and explain *how we emulate the support in software*. Figure 4 is an overview of PARIS, depicting how the OS interacts with the DRAM controller and the DRAM.

4.2.1 Extracting empty DRAM row information in Linux

In order to extract empty DRAM row information, we must first obtain information about page frame usage. There are at least two feasible methods to get the page frame usage information in Linux. The first method relies on the

3. To guarantee reliable data retention, for example, DRAM specifications require that all DRAM rows be refreshed before entering and after exiting from the self refresh mode.

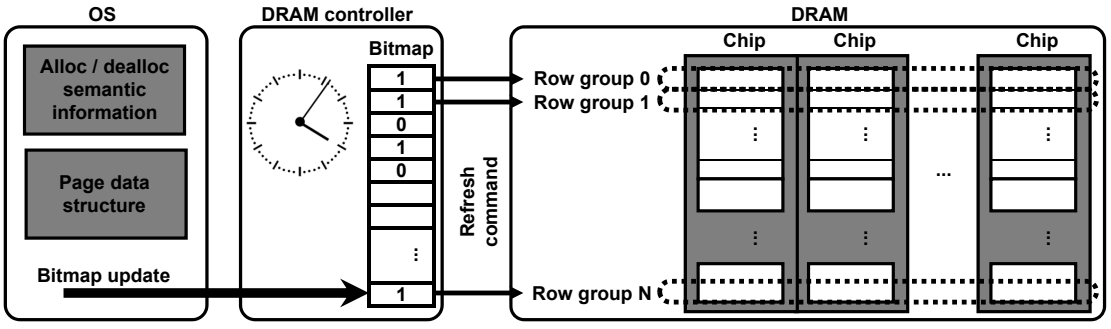


Fig. 4: Overview of PARIS.

information in a static bitmap data structure. Linux manages physical memory resources in page frame unit (typically 4 KB). At boot time, we create a bitmap where each bit corresponds to a page frame in the system. Note that Linux does not manage bitmap for each page frame since it adopts a *Lazy Buddy* algorithm. Then, with minimum modification, we can make Linux set or clear the bitmap when Linux allocates or deallocates page frames using $alloc\_pages()$ and $free\_pages()$, which are the lowest-level memory management interfaces of the Linux *Buddy* allocator. Using this bitmap, we can decide which page frames should be refreshed.

The second method dynamically gathers page frame status information from the Linux *page* data structure. The *page* data structure is at a specific DRAM address and stores page frame status information in a simple array called *mem_map*. Each *page* structure has *flags*, which reflects the current status of the corresponding page frame. Linux updates *flags* dynamically whenever the corresponding page frame status is changed. We can identify currently unused page frames by scanning *flags* of each *page* structure in *mem_map*. We have implemented both methods and evaluated their overheads. Because page frame usage status changes incrementally on specific events (i.e., allocation and deallocation), both methods are feasible with negligible overheads.

Once page frame usage information is collected, the information must be translated into DRAM row level information. How this translation must be done depends on how page frames are mapped to specific DRAM addresses and varies from platform to platform. Once the mapping is known, the translation is straightforward.

4.2.2 Architectural support for selective refreshing

For PARIS, the DRAM controller has to provide a mechanism for the OS to convey the empty row information. A simple bitmap (Figure 4) is sufficient to express which *row groups* to refresh or not. A row group is 2^M DRAM rows ($M \geq 0$) that is assigned a single *full/empty bit* in the bitmap. A DRAM row may span multiple chips (tapping the same chip select signal and receiving data in an interleaved manner). Hence, a single F/E bit covers potentially large capacity (multiple DRAM rows). For example, if we lay out two x32 LPDDR2 chips in parallel (like in Panda), the minimum capacity covered by a single F/E bit is 4 KB.

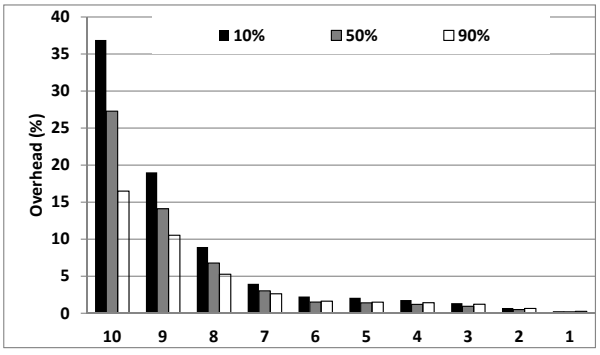


Fig. 5: Refresh overhead vs. M (under memory usage of 10%, 50% and 90%).

In general, a F/E bit covers $\max(\text{page frame size}, 2^M \times \text{DRAM row size})$. Because refresh actions are required per DRAM row and not physical page frame, M determines the bitmap size (=DRAM capacity/($2^M \times$DRAM row size)).

The bitmap size grows with the increase in the DRAM capacity. In Panda (1 GB DRAM), the bitmap could be as large as 1 GB/4 KB=256 Kbits, assuming single-bank refresh and $M = 0$. With more common all-bank refresh, we need 256 Kbits/8 banks=32 Kbits or 4 KB.[4] The bitmap size can be further reduced by increasing M (i.e., row group size). However, because a larger M implies more DRAM rows per bit and more chances for unnecessary refresh on empty rows, we studied how M affects the number of refresh operations.

Figure 5 presents the result. We counted additional refresh operations when $M > 0$ and compute their relative contribution to the base case of $M = 0$ under DRAM memory usage of 10%, 50% and 90%. We artificially allocate memory in a dummy Linux kernel module using *kmalloc* to reach a desired memory usage condition before gathering memory status information from the buddy allocator. The result shows that M can be increased to 7 while limiting the overhead to below 5%. Overhead was less than 4% when memory usage was 10%. Hence, when $M = 7$, the PARIS bitmap size on Panda is only 32 Kbits/2^7 or 256 bits.

The PARIS DRAM controller uses a timer to catch inter-refresh interval (=refresh period/number of rows) like conventional DRAM controllers. However, the PARIS DRAM controller issues a refresh command for a row selectively

4. Both the platforms used in this work support only all-bank refresh.

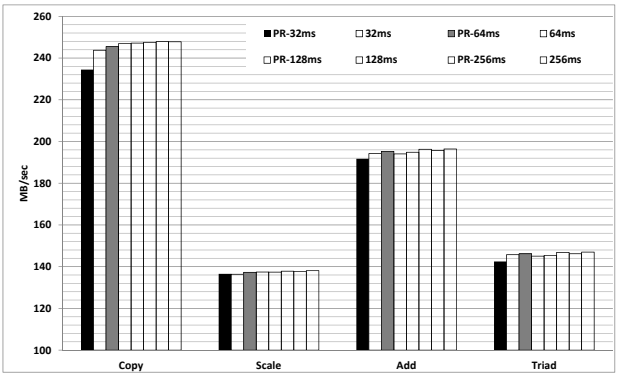


Fig. 6: Pseudo-refresh calibration result.

(i.e., using a RAS-only refresh (ROR) command), only when the corresponding F/E bit has been set to 1. While not required by PARIS, a preferred DRAM design supports ROR. We believe that the required architectural support for PARIS is simple and realizable.

4.2.3 Implementation

To prove the concept of PARIS and the proposed architectural support through experimentation without actual hardware, we design and implement *pseudo-refresher* (PR), a completely flexible software DRAM refresher. PR is implemented as a Linux loadable kernel module. To obtain execution efficiency and predictability, PR is run as a real-time kernel thread on a dedicated processor core (Panda). As a result, PR is not interrupted or scheduled out. After initialization, PR is in an infinite loop, each iteration of which walks three steps: *page frame status check*, *refresh command issue* and *refresh period wait*.[5] When PR runs, *we turn off the hardware DRAM refresher* in the DRAM controller.

The first step of PR iteration identifies empty DRAM rows (Section 4.2.1). The second step issues refresh commands to non-empty DRAM rows. In order to emulate selective refreshing, PR actually reads target non-empty rows. Hence, when PR is in charge of DRAM refreshing, simply counting the number of DRAM reads issued by PR allows us to accurately estimate the number of refresh actions. To ensure a read reaches DRAM, we flush the corresponding cache line first. Finally, PR waits for the current refresh period to expire using a platform's high-resolution hardware timer.

Despite our efficient implementation, PR (essentially a co-scheduled thread) may incur unwanted perturbation during experiments. As we run PR on a dedicated processor core, the perturbation is caused primarily by cache flush/fill operations and contention in the datapath to DRAM. In order to quantify the degree of perturbation, we calibrated PR by comparing the performance of stream-oriented benchmarks (whose details are in Section 5.1) under PR and the hardware refresher.

Figure 6 reports four pairs of experiments per benchmark: PR vs. hardware refresher when the refresh period is 32 ms, 64 ms, 128 ms and 256 ms. The obtained result is intuitive; *performance is lower with PR due to its overhead*. What is important is *how lower*: The performance gap between PR and the hardware refresher becomes smaller as we increase the refresh period, to the point that the gap is nearly unnoticeable at 128 ms and beyond. Our calibration study shows that program performance under PR is affected little if the refresh period is fairly long. Because PARIS is effective only when the system's memory utilization is low (i.e., the number of non-empty DRAM rows is small), the overhead of PR will likely remain very limited in that case. It is worth noting that the use of PR leads to conservative evaluation of PARIS and does not put PARIS in an advantageous position compared with a hardware refresher.

5. PR can emulate arbitrary algorithms for ROR, e.g., RAIDR [13]. We limit the use of PR to studying PARIS in this work.

5 Experimental Evaluation

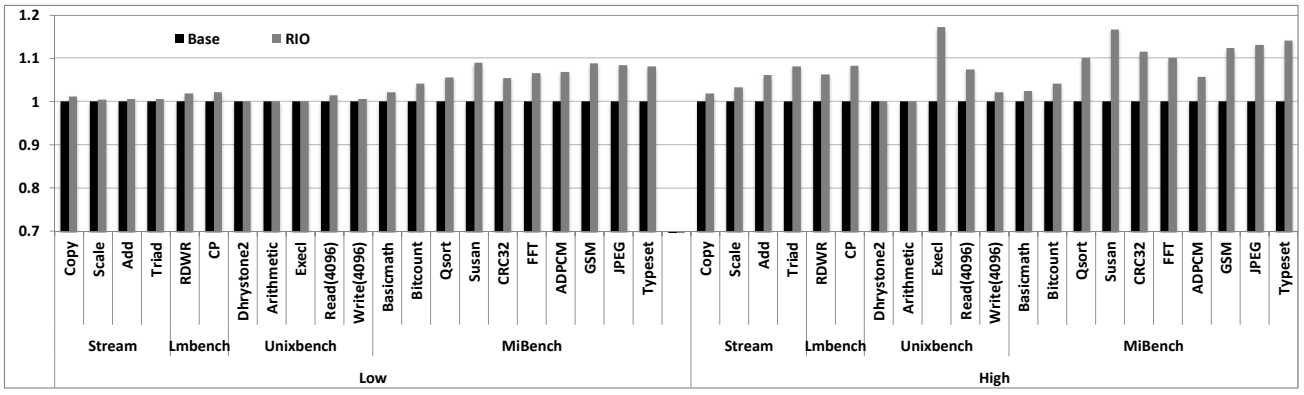
5.1 Setup and benchmark

To evaluate RIO and PARIS, we employ both Beagle and Panda. For intuitive discussions, we choose two temperatures for experiments: Normal (45°C) and High (85°C). We use benchmarks drawn from the following four suites:

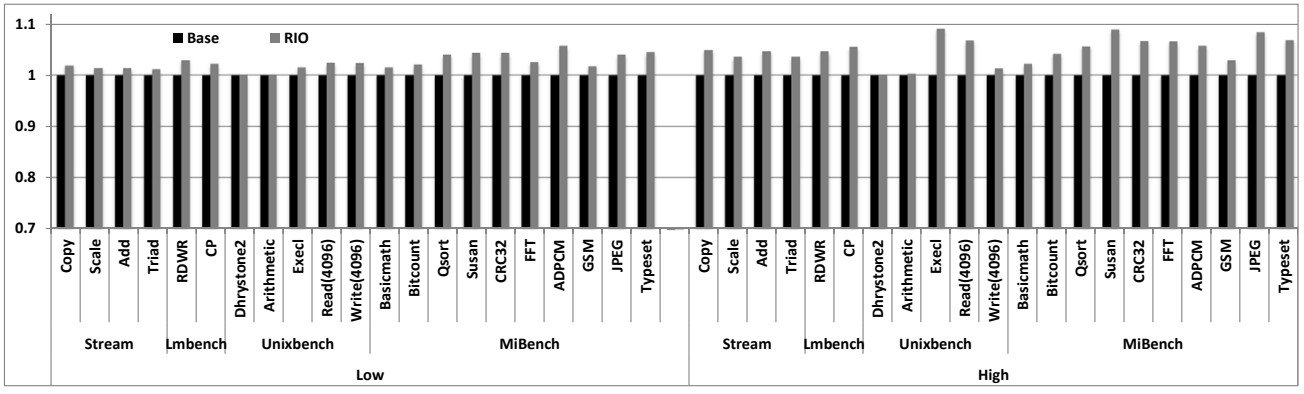
- **Stream** [24]: A synthetic benchmark used for measuring sustainable memory bandwidth. Stream executes four tests, Copy, Scale, Add and Triad, and reports result in MB/sec.
- **Lmbench** [25]: A simple bandwidth and latency benchmark designed to compare Unix systems. Lmbench executes two tests, RDWR and CP, and reports results in benchmark-specific score.
- **Unixbench** [27]: A general-purpose benchmark designed to evaluate the performance of Unix systems. Unixbench integrates CPU, memory and file I/O tests as well as system behavior under various loads. It reports the number of loops performed within a given time. We report five results: "Dhrystone2", "Arithmetic (double)", "Execl throughput", "File read (4 KB buffer size)" and "File write (4 KB buffer size)".
- **MiBench** [26]: A set of commercially representative embedded programs for benchmarking purposes. These benchmarks are divided into six suites (Automotive, Industrial, Consumer, Office, Automation, Networking, Security and Telecommunications) with each suite targeting a specific area of the embedded market.

Benchmark programs were run on an unloaded target platform five times. Before running a benchmark program, we always put the system into a known, clean state by rebooting it. We checked the system integrity by encoding MP3 files and decoding them again (and repeating this process), before performing performance measurement. When reporting results, we use an average value of execution times or benchmark-specific scores.

For performance studies, we compare three configurations, "RIO", "RIO+PARIS" (two techniques used together), and a baseline configuration ("Base") that follows the refresh period specified by the examined DRAM. Base refreshes DRAM at 32 ms at Normal and 8 ms at High. RIO uses Table 2 to determine the refresh period.



(a) Beagle



(b) Panda

Fig. 7: Performance of RIO (normalized to Base) at Normal and High.

5.2 Results

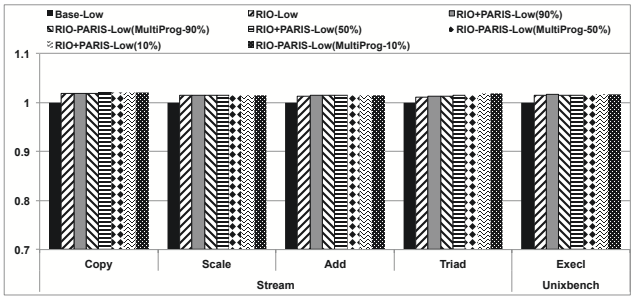
Performance improvement with RIO. Figure 7 presents the relative performance of RIO against Base. Performance improvement was measurable for many programs, especially at High. The maximum performance gain was 17.2% for Execl on Beagle, obtained at High. Execl saw the highest performance improvement on Panda as well. Larger performance gains at High are plausible because High forces the use of faster refresh rates, and as a result, more frequent DRAM access conflicts occur. Still, interestingly, the amount of gain varies across benchmarks and across platforms. For example, performance gain with RIO was larger on Beagle at High. Dhrystone2 and Arithmetic are insensitive to refresh reduction at all, because they are compute-intensive throughout their execution. Application benchmarks (Execl and MiBench) gained more performance with refresh reduction than simple microbenchmarks (Stream and Lmbench) that have predictable, sequential accesses because those show memory intensive behavior during the executions.

DRAM access conflict caused by refreshing may incur two types of performance penalties. First, while DRAM undergoes refreshing, DRAM is unavailable and the effective DRAM bandwidth is decreased. Subsequent DRAM accesses are held back by the DRAM controller, increasing the latency seen by the accesses. Second, data in the per-bank row buffers are lost and subsequent DRAM accesses that would otherwise hit in row buffers may have to re-open the banks.

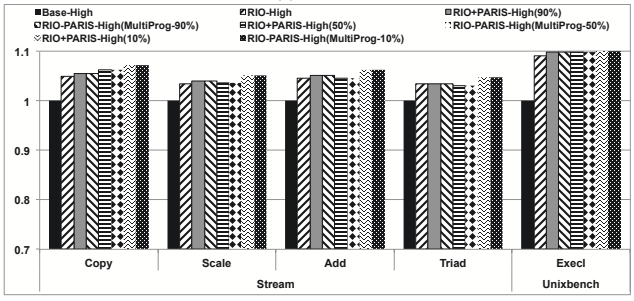
It is believed that architectural differences between Beagle and Panda are responsible for some variation. Beagle has an ARM Cortex-A8 processor with 16 KB L1 caches and a 256 KB L2 cache. By comparison, Panda has more capable Cortex-A9 processors with 32 KB L1 caches and a 1 MB L2 cache. Furthermore, Beagle's maximum DRAM bandwidth is only half that of Panda. The smaller caching capacity of Beagle increases its performance dependence on DRAM bandwidth and access latency, especially at High. At Normal, bandwidth competition becomes less severe, and hence, the effect of row buffer misses may become more pronounced.

Other architectural artifacts are believed to create the behavioral distinction between memory-intensive microbenchmarks (Stream and Lmbench) and application benchmarks (Execl and MiBench). Stream and Lmbench programs have long, yet fairly predictable read and write accesses on array data. Because they are memory-intensive, they could have been sensitive to DRAM bandwidth loss due to refresh actions. However, because of their data streaming nature, there are frequent dirty block evictions from the cache and the write buffer remains the performance bottleneck and dilutes their performance dependence on the DRAM bandwidth.

Performance gain across all benchmarks was 3.3% at Normal and 6.9% at High on average. Individual performance gain was 0.7% (Stream), 2.0% (Lmbench), 0.7%



(a) Normal



(b) High

Fig. 8: Performance of Stream and Unixbench (Execl) with Base and RIO+PARIS on Panda.

(Unixbench, excluding Dhrystone2 and Arithmetic) and 6.5% (MiBench) at Normal and 4.9%, 7.3%, 8.9% and 10% at High on Beagle. On Panda, gain was 1.5% (Stream), 2.5% (Lmbench) 2.1% (Unixbench, excluding Dhrystone2 and Arithmetic) and 3.5% (MiBench) at Normal and 4.2%, 5.0%, 5.7% and 5.8% at High, respectively. We are encouraged by our result, evidencing that *extending refresh period can translate into non-trivial performance gains*.

Performance of RIO+PARIS. Figure 8 compares the performance of RIO and RIO+PARIS against Base when running the Stream benchmark (most insensitive programs to RIO) and Execl (most sensitive). For PARIS, we consider three memory usage scenarios—low (roughly 10%), medium (50%) and high (90%) memory utilization. To adjust memory utilization, we run a dummy process that reserves a desired amount of DRAM space before experimenting, denoted as 10%, 50% and 90% in the Figure. From the point of view of the OS, this dummy process effectively mimics heavy memory pressure of real workload because OSes use demand-paging mechanisms in page unit. However, we additionally use another method to adjust memory utilization to emulate more realistic environment. We run multiple numbers of Lmbench benchmark, whose memory working set size is 120MB, and STREAM benchmark, whose memory working set size is 32MB, at the same time. For example, we run one Lmbench benchmark for the 10% of memory utilization, four Lmbench benchmarks and one STREAM benchmark for the 50% of memory utilization, and seven Lmbench benchmarks and one STREAM benchmark for the 90% of memory utilization, which are denoted as MultiProg-10%, MultiProg-50% and MultiProg-90%, specifically. Because PARIS requires running the pseudo-refresher, we use only Panda in this section.

Overall, our result shows that PARIS can increase the application performance beyond what RIO achieves alone, and the performance gain increases with the decrease in memory usage (90% down to 10%). However, the additional gain of PARIS was rather marginal, especially at Normal. This is because RIO reduces the refresh overhead to the level that leaves little room for further performance improvement. At Normal, additional performance gains with PARIS (RIO+PARIS vs. RIO) under low memory utilization (10%) were: Copy (0.18%), Scale (0.11%), Add(0.15%), Triad (0.81%) and Execl (0.18%). At High, additional performance gains with RIO+PARIS under low memory utilization (10%) were: Copy (2.12%), Scale (1.61%), Add (1.56%), Triad (1.25%) and Execl (0.86%).

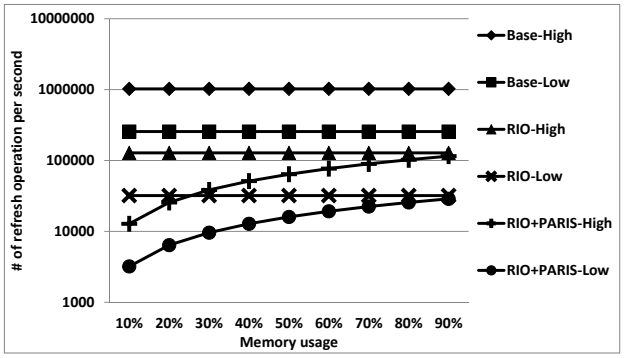


Fig. 9: Number of refresh operations per second.

Refresh reduction with RIO and PARIS. Finally, Figure 9 plots the number of refresh operations per second demanded by each examined scheme. The plot captures the ultimate effectiveness of RIO and PARIS; by increasing the refresh period and excluding empty DRAM rows for refreshing, they reduce the refresh operations required in unit time.

At Normal, RIO reduces refresh operations by about an order of magnitude, achieving a 87.5% reduction (RIO-Low vs. Base-Low). PARIS extends the gap further; RIO+PARIS achieves a reduction of nearly two orders of magnitude under a low memory utilization (10%) and a reduction of 93.8% under a medium memory usage (50%) on average (RIO+PARIS-Low vs. Base-Low). Similarly, at High, RIO and RIO+PARIS achieve a reduction of 8× and up to 80× against Base under a memory usage of 40% or less. The plot also shows that gains with PARIS diminish with the increase in memory usage and by 90% of usage RIO+PARIS converges to RIO.

6 Conclusions

This paper closely examined modern LPDDRx DRAM's cell-level retention behavior using real, on-board measurements. Our examination motivates RIO and PARIS to deeply reduce refresh operations. We implement both schemes in the Linux OS and evaluate them on smartphone-like platforms. As such, our key design strategies and decisions were made based on practical system design perspectives. We summarize the system modifications and overheads incurred by RIO and PARIS, along with other schemes, in Table 4. While all listed schemes require

TABLE 4: Refresh reduction schemes. *nPages*: # page frames. *nRows*: # DRAM rows. *nRGroups*: # row groups (PARIS). Schemes are classified into two groups; two schemes from different groups can be used together, e.g., RAPID+Flikker, RIO+ESKIMO, RAIDR+PARIS.

Scheme	Key Modifications				Storage Overhead (for 4 GB DRAM)
	CPU	DRAM ctrl.	DRAM	Software (complexity)	
RAPID-1 [10]	-	-	-	OS (very light)	-
RAPID-2/-3 [10]	-	-	-	OS (heavy)	4 MB (*nPages* × 4B)
RAIDR [13]	-	Bloom filter	ROR cmd.	OS (very light)	∼1 KB (Bloom filter)
RIO [this work]	-	-	-	OS (very light)	-
ESKIMO [11]	New instr.	Bitmap	ROR cmd.	Library, OS (heavy)	472 KB (*nRows* bits + tracking data)
Flikker [9]	-	-	Two ref. regions	App, compiler, library, OS (heavy)	-
PARIS [this work]	-	Bitmap	ROR cmd.	OS (very light)	1 KB (*nRGroups* bits)

changes to the OS, no prior work realized them in a real OS and experimented on a real platform. This paper, based on real implementation, makes the following contributions:

• Our measurement study led to findings and insights about feasibility and limits of DRAM refresh reduction techniques. Weak cells are manifested when the refresh rate is slowed by 2× to 16×. They occur randomly in space. We also found that pushing the refresh period to more than 256 ms brings little additional benefits. In fact, our result strongly suggests that extending the refresh period to more than 512 ms is simply not feasible because of physical memory fragmentation.

• We propose, design and evaluate RIO. We determined that at least a 2× guard band (between the chosen refresh period and the weakest cell's retention time) is necessary to safely employ RIO, for the DRAM chips studied in this paper. We delete fewer than 0.1% of all page frames to extend refresh period by 2× to 16×. Performance improvement with RIO was measurable and up to 17.2% and 4.5% on average.

• We propose, design and evaluate PARIS. We discuss how PARIS can be efficiently implemented in Linux and proposes simple architectural support to selectively refresh DRAM rows. We emulate the proposed hardware using a flexible software DRAM refresher that runs alongside a benchmark under study. PARIS, when combined with RIO, was shown to reduce refresh operations by as much as 93.8% on average.

We envision a couple of directions for future research. First, studies the impact of RIO and PARIS on the performance of hardware accelerators like HD video codec will be interests. Typically, accelerators have substantially higher DRAM bandwidth requirements than equivalent software. Second, exploring new hardware and software techniques to reduce refresh overheads in DRAM's self refresh mode could be worthwhile. The partial array self refresh interface has been implemented in LPDDR/LPDDR2 DRAMs; however, the interface is rarely used in practice due to lack of software support. Based on our experience with RIO and PARIS, a new interface to flexibly represent memory usage appears feasible.

ACKNOWLEDGMENT

This work was supported in part by the National Research Foundation of Korea Grant funded by the Korean Government (NRF-2011-220-D00098), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012R1A6A3A03040447) and the US National Science Foundation (CCF-1064976 and CNS-1012070).

REFERENCES

[1] K. Saino, S. Horiba, S. Uchiyama, Y. Takaishi, M. Takenaka, T. Uchida, Y. Takada, K. Koyama, H. Miyake, and C. Hu, "Impact of Gate-Induced Drain Leakage Current on The Tail Distribution of DRAM Data Retention Time," in *International Electron Devices Meeting (IEDM), Technical Digest*, pp. 837 –840, 2000.

[2] S. S. Takeshi Hamamoto and S. Sawada, "On The Retention Time Distribution of Dynamic Random Access Memory (DRAM)," *IEEE Transactions on Electron Devices*, vol. 45, no. 6, pp. 1300–1309, Jun 1998.

[3] "Micron 8Gb Mobile LPDDR2 SDRAM Datasheet." http://www.micron.com/products/dram/mobile_lpdram.html

[4] "Samsung 512Mb SDRAM Datasheet." http://www.samsung.com/global/business/semiconductor/products/dram/Products_DRAM.html

[5] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*, pp. 375–384, 2010.

[6] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed., Morgan and Claypool Publishers, 2009.

[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 *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*, pp. 134–145, 2007.

[8] M. A. Viredaz, M. A. Viredaz, D. A. Wallach, and D. A. Wallach, "Power Evaluation of a Handheld Computer: A Case Study," Compaq Western Research Laboratory, Tech. Rep., 2001.

[9] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*, pp. 213–224, 2011.

[10] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID):Software Methods for Quasi-Non-Volatile DRAM," in *In Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture (HPCA '06)*, pp. 155–165, 2006.

[11] C. Isen and L. John, "ESKIMO: Energy Savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*, pp. 337–346, 2009.

[12] J. Kim and M. C. Papaefthymiou, "Dynamic memory design for low data-retention power," in *Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*, 2000, pp. 207–216.

[13] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," in *Proceedings of the 39th annual international symposium on computer architecture (ISCA '12)*, 2012.
[14] "JEDEC." http://www.jedec.org
[15] "Micron MT4LC4M4E8." http://download.micron.com/pdf/datasheets/dram/d47b.pdf
[16] V. Bhalodia, "Scale DRAM Subsystem Power Analysis," Master's Thesis, Massachusetts Institute of Technology, 2005.
[17] J. Kim and M. C. Papaefthymiou, "Dynamic Memory Design for Low Data-Retention Power," in *Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation (PATMOS '00)*, pp. 207–216, 2000.
[18] C. H. Louis Lu, F. Gerd, S. Hohenkirchen, and G. Oliver Weinfurtner, "Dynamic DRAM Refresh Rate Adjustment based on Cell Leakage Monitoring," United States Patent, US 6,483,764, Sep. 26 2002.
[19] T. Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs," in *Proceedings of the 1998 international symposium on Low power electronics and design (ISLPED '98)*, pp. 82–87, 1998.
[20] S. P. Song, "Method and System for Selective DRAM Refresh to Reduce Power Consumption," United States Patent, US 6,094,705, Jul. 25 2000.
[21] "Beagle board." http://beagleboard.org
[22] "Panda board." http://pandaboard.org
[23] "Micron Power Calculator." http://www.micron.com/support/dram/power_calc.html
[24] "STREAM." http://www.cs.virginia.edu/stream
[25] "Lmbench." http://www.bitmover.com/lmbench
[26] "Mibench." www.eecs.umich.edu/mibench/
[27] "Unixbench." http://code.google.com/p/byte-unixbench

Seungjae Baek received his BS, MS and PhD in computer engineering from Dankook University in 2005, 2007, and 2010, respectively. He joined Peromnii Inc. in 2010, as a start-up member, and contributed to the definition and development of instant booting technique. He has been a post-doctoral research associate at the University of Pittsburgh (Pitt) since 2011. Before joining Pitt, he focused on system software issues that arise when Flash memory and novel memory technologies such as PCM, FeRAM and MRAM are deployed. After joining Pitt, his work expanded to the DRAM management scheme for reliability and performance enhancement. In particular, he is interested in file system, storage device, and operating system itself. He is a member of the IEEE and the ACM.

Sangyeun Cho received his B.S. degree in Computer Engineering from Seoul National University in 1994 and a Ph.D. in Computer Science from the University of Minnesota in 2002. In 1999, he joined Samsung Electronics Co., where he was a lead architect of CalmRISC-32, a 32-bit microprocessor core, and designed its caches, DMA, and stream buffers. Since 2004, he has been with the Computer Science Department at the University of Pittsburgh, where he is an Associate Professor. His research interests include computer architecture and embedded systems, with a focus on performance, power, and reliability of the memory and storage hierarchy.

Rami Melhem received a B.E. in Electrical Engineering from Cairo University in 1976, an M.A. degree in Mathematics and an M.S. degree in Computer Science from the University of Pittsburgh in 1981, and a Ph.D. degree in Computer Science from the University of Pittsburgh in 1983. He was an Assistant Professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a Professor in the Computer Science Department which he chaired from 2000 to 2009. His research interests include Power Management, Real-Time and Fault-Tolerant Systems, Optical Networks, High Performance Computing and Parallel Computer Architectures. Dr. Melhem served and is serving on program committees of numerous conferences and workshops and on the editorial boards of the IEEE Transactions on Computers (1991-1996, 2011-), the IEEE Transactions on Parallel and Distributed systems (1998-2002), the Computer Architecture Letters (2001-2010), the Journal of Parallel and Distributed Computing (20032011) and The Journal of Sustainable Computing, Informatics and Systems (2010 -). Dr. Melhem is a fellow of IEEE and a member of the ACM.