DTail: A Flexible Approach to DRAM Refresh Management

Zehan Cui[†‡], Sally A. McKee[§], Zhongbin Zha[†‡], Yungang Bao[†], Mingyu Chen[†]
[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
[‡]University of Chinese Academy of Sciences
[§]Chalmers University of Technology
{cuizehan,zhazhongbin,baoyg,cmy}@ict.ac.cn mckee@chalmers.se

ABSTRACT

DRAM cells must be refreshed (or rewritten) periodically to maintain data integrity, and as DRAM density grows, so does the refresh time and energy. Not all data need to be refreshed with the same frequency, though, and thus some refresh operations can safely be delayed. Tracking such information allows the memory controller to reduce refresh costs by judiciously choosing when to refresh different rows.

Solutions that store imprecise information miss opportunities to avoid unnecessary refresh operations, but the storage for tracking complete information scales with memory capacity. We therefore propose a flexible approach to refresh management that tracks complete refresh information within the DRAM itself, where it incurs negligible storage costs (0.006% of total capacity) and can be managed easily in hardware or software. Completely tracking multiple types of refresh information (e.g., row retention time and data validity) maximizes refresh reduction and lets us choose the most effective refresh schemes. Our evaluations show that our approach saves 25-82% of the total DRAM energy over prior refresh-reduction mechanisms.

1. INTRODUCTION

Main memory systems are commonly composed of cheap, dense Dynamic Random Access Memory (DRAM) devices. DRAM devices store data as charges on cell capacitors, and these charges leak over time. Data thus must be periodically refreshed — read and rewritten — to maintain integrity.

The storage cells are arranged in banks of rectangular arrays indexed by row (wordline) and column (bitline). DRAM accesses *open* a row by loading its entire contents into a bank of sense amplifiers (or a *row buffer*) from which data may be read or written. When data in a different row are needed, the memory controller *closes* the row by writing the contents back to the storage array and precharging the row buffer. Refresh simply opens and closes rows without servicing intervening accesses. We find that, on average, refresh operations on current 4Gb DRAMs impose performance and energy overheads of 7.2% and 18.9%, respectively.

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597663.

Since the length of the DRAM wordlines is constrained by the energy required to drive the bitlines and to open and close the row, memory designers increase density by adding more rows to each bank [13, 14]. This, in turn, requires that more rows be refreshed before data values degrade, causing the refresh time and energy to increase, as well.

DRAM memory systems already account for up to 40% of total system power, which represents up to 60% of the power for each processor-node (CPUs and DRAMs) [18, 3]. For exascale systems, DRAM is predicted to consume about 75% of processor-node power budgets [4]. Within four generations, almost half the memory's contribution to system power will come from refresh instead of useful accesses [20]. This means that even small refresh reductions will be more significant as memory power grows to dominate system power.

A smart memory controller can avoid unnecessary refreshes based on certain information: rows that maintain data integrity longer can be refreshed less often, and rows without meaningful data need not to be refreshed. Since completely tracking such information — termed *refresh data* (RD) — incurs significant costs, previous smarter-refresh approaches compromise by storing imprecise refresh data [20, 2], which reduces their opportunities to optimize refresh. In addition, the ways in which these prior approaches track refresh data make it difficult to leverage multiple types of RD information. For instance, Bloom filters [20] work well for tracking the long tail distribution of retention time variation, but they are impractical for tracking data validity.

We aim to build a smart-refresh memory system that:

1. stores refresh data with negligible cost,
2. tracks multiple types of refresh data,
3. coordinates the memory controller and all levels of software stack, and
4. performs necessary refreshes efficiently.

To this end, we propose *DTail*[1], a flexible, low-overhead DRAM refresh management scheme. The key idea of DTail is to store refresh data in the DRAM itself. Doing so greatly reduces the storage cost because DRAM capacity is much larger and cheaper than either SRAM or registers. Since DRAM data can be accessed with normal memory instructions, the memory controller can collaborate with all levels of the software stack to track and leverage the RD. The row-by-row behavior of refresh makes simple prefetching and caching mechanisms effective for masking access latencies. We further leverage the RD to dynamically select either automatic refresh or explicitly controlled refresh, whichever is likely to perform better. Our contributions are:

[1]We choose this mnemonic because we try to "dovetail" different types of refresh data to create a strong, stable, smart refresh mechanism.

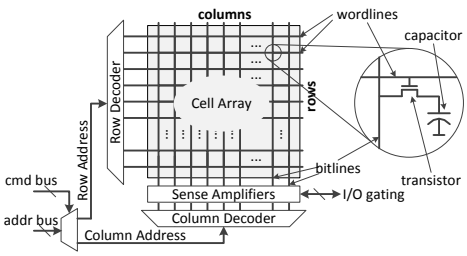
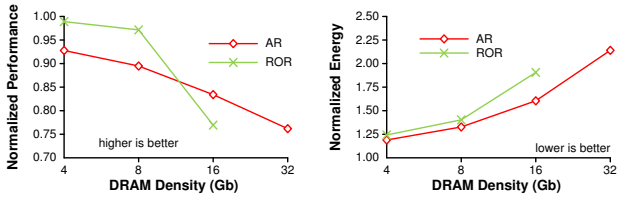


Figure 1: Bank Structure



(a) Performance **(b) Energy**

Figure 2: Performance and energy overheads for using auto refresh (AR) and RAS-only refresh (ROR) for various DRAM densities. Results are normalized to an ideal system that omits refresh entirely.

- We propose DTail, which can completely track multiple types of refresh data and coordinate with all levels of the software stack. DTail introduces negligible storage cost for refresh data — 0.006% of memory capacity — and maximizes refresh reduction;
- We propose to dynamically select among different refresh implementations by predicting the most effective way to perform necessary refreshes;
- We find that by tracking complete refresh data, DTail respectively saves 23.3% and 9.05-41.7% (depending on memory utilization) DRAM energy over two prior mechanisms; and
- We find that by tracking and combining multiple types of refresh data, DTail respectively saves between 25.9-40.0% and 24.6-81.9% (depending on memory utilization) of the DRAM energy over two prior mechanisms.

2. BACKGROUND AND MOTIVATION

We briefly outline DRAM organization and operation. Figure 1 shows the typical organization of a DRAM bank. Each bank comprises a two-dimensional array of cells, sense amplifiers, row and column decoders, and peripheral circuits. To access a bit in the array, the memory controller (MC) sends a row address strobe (RAS)[2] that loads the row into the row buffer (sense amplifiers). The memory controller then sends one or more column address strobes (CASs) to access specific bits. When the memory controller finishes accessing a row, it sends a precharge (PRE) command to write the values back to the storage array and prepare the row buffer for the next access. Multiple banks, which can be accessed in parallel, are organized into ranks, and channels connect one or more ranks to the MC.

2.1 Refresh Operation

To simplify hardware design, refresh operations are traditionally performed for all rows within a period short enough to guarantee data integrity for the most leaky cells[3]. Instead of refreshing all rows sequentially in a burst, refresh operations are usually staggered across the period to avoid blocking normal memory accesses. Each operation may affect one or more rows, depending on the implementation.

JEDEC makes *auto refresh* (AR) a standard for DDRx SDRAMs. The DRAM chip internally maintains a row address counter (RAC) pointing to the next row to refresh. The MC issues a refresh command every $tREFI$ cycles, at which point all DRAM banks simultaneously refresh a number of rows — making the rank unavailable for $tRFC$ cycles — and increase their RAC accordingly. The number of rows to refresh at a time depends on density.

In *RAS-only refresh* (ROR) [24], the MC issues a row address strobe to load the specified row into the row buffer. The row is eventually written back to the DRAM array by a subsequent precharge command. In this case the MC maintains its own RAC. The controller periodically increments the RAC and issues a row address strobe. Refresh proceeds one row at a time, but the MC chooses which refreshes to schedule when. Note that the granularity of auto refresh is much larger than that of RAS-only refresh.

2.2 Performance and Energy Overheads

Table 1 shows refresh timing parameters (auto refresh) for x8/x16 width devices in the JEDEC DDR4 specification [14] at extended temperature ranges. (Devices of x4 width have twice the number of rows.) The interval between two consecutive auto refresh commands, $tREFI$, remains constant. Thus, the number of rows per refresh command doubles each generation. However, the increase in the refresh delay, $tRFC$, is only 1.3× because modern DRAM banks comprise multiple subarrays [17] that can refresh multiple rows in parallel. The portion of time that a rank is unavailable is 8.97% for 8Gb chips and 16.4% for 32Gb chips.

If refresh blocks too many normal memory accesses, the MSHRs (miss status holding registers) or load/store queue slots fill, preventing the issue of other memory access instructions. Delaying critical loads needed by instructions in flight quickly stalls the pipeline. Figure 2 shows performance and energy overheads for a system using auto refresh or RAS-only refresh compared to a system that omits refresh entirely: these overheads grow as density increases, e.g., auto refresh incurs 23.8% performance overhead and 114% energy overhead versus a 32Gb refresh-less system.

Figure 2 omits data for RAS-only refresh at 32Gb because at this density it fails to refresh all rows in our evaluated DRAM organizations. Table 1 shows that 512 rows need to be refreshed every 3.9μs for 32Gb DRAMs, which requires a

Table 1: JEDEC DDR4 Refresh Timing Parameters of x8/x16 Devices at Extended Temperature Range

Density	#rows per refresh command	$tREFI$	$tRFC$	$tRFC/tREFI$
2Gb	16×2[a]	3.9μs	160ns	4.10%
4Gb	16×4	3.9μs	260ns	6.67%
8Gb	16×8	3.9μs	350ns	8.97%
16Gb	16×16	3.9μs	480ns[b]	12.3%
32Gb	16×32	3.9μs	640ns[b]	16.4%

[a] DDR4 has 16 banks, with multiple rows per bank refreshed each time.
[b] We use the projection from Mukundan et al. [27] for 16Gb and 32Gb.

[2]The name RAS is now commonly replaced by ACT for *activate*, but we choose RAS for historical consistency.

[3]For DDRx SDRAM, the period is respectively 64ms and 32ms at normal (0-85 °C) and extended (85-95 °C) temperature ranges.

RAS command to be issued every six cycles (3.9μs/1.25ns/512). A subsequent precharge command is required to write the row back to the DRAM array, which means the address bus would need to transmit two commands every six cycles for each device. This is not possible in our DRAM organization with four ranks sharing one bus.

2.3 Limitations of RAS-only Refresh

RAS-only refresh allows fine-grained control of selective refreshes, and it outperforms auto refresh for lower density devices (4Gb/8Gb) due to DRAM bank-level parallelism. However, it must adhere to the DRAM timing constraints. Specifically, the MC must observe $tRRD$ (the activate-to-activate command delay) and $tFAW$ (four activate window). The former defines the minimum interval between two consecutive RAS commands to the same DRAM device, and the latter specifies a sliding window during which no more than four RAS commands can be issued to the same DRAM device.

As DRAM density increases, the number of rows that need to be refreshed doubles with each generation (Table 1), which means more RAS-only refreshes must be issued. Constrained by $tRRD$ and $tFAW$, RAS-only refresh hurts performance and energy efficiency at higher DRAM densities. Since auto refresh can operate on multiple rows simultaneously, it becomes relatively more efficient at higher densities. Figure 2 shows that auto refresh exhibits higher performance and energy efficiency at a 16Gb density. This suggests that it may be better to dynamically choose between auto refresh and RAS-only refresh, depending on the number and location of rows needing to be refreshed.

3. RELATED WORK

Memory systems incorporating intelligent refresh must decide: 1) when to schedule refresh operations with respect to other memory commands, and 2) which rows to refresh. We briefly survey approaches to making the former decision before treating solutions to the latter in more detail.

3.1 Deciding When to Schedule Refreshes

Refresh schedulers can be classified according to how they schedule refresh operations [35] and whether they schedule regular accesses around [27] or within them (for multi-row refresh granularities) [28]. Stuecheli et al. propose *Elastic Refresh* [35] to dynamically fit the refresh period to the currently executing workload. To prevent the MC queue from stalling useful accesses when filled with commands to a bank being refreshed, Mukundan et al. [27] propose Dynamic Command Expansion (DCE) and Preemptive Command Drain (PCD). The former delays commands to banks under refresh, and the latter proactively schedules commands to banks about to undergo refresh. Nair et al. [28] call for pausable refresh operations that allow regular accesses to proceed with less delay.

3.2 Deciding What Not to Refresh

Intelligent refresh schemes can also be classified according to the kinds of refresh data on which they base their scheduling decisions: cell retention time (**R**), error tolerance of the data (**T**), access recency (**A**), and row validity (**V**), i.e., whether the OS has allocated the physical pages containing those cells. We use this **RTAV** RD taxonomy to survey the rich prior work in DRAM refresh reduction, a summary of which is shown in Table 2.

Retention. Retention time refers to the period during which cells hold valid values as charge gradually leaks. Process variation [10][16] causes the retention time of DRAM cells to vary across the chip. Note that approaches that exploit **R** information are insensitive to system workloads and global memory usage, which makes them attractive components for a refresh-optimized memory subsystem.

In hardware, **R**-based approaches can employ multi-period schemes to refresh cells with long retention times less frequently, as in the Variable Refresh Architecture (VRA) of Ohsawa et al. [29] and the Retention-Aware Intelligent DRAM Refresh (RAIDR) of Liu et al. [20]. VRA stores each row's expected refresh period in registers inside the DRAM. RAIDR exploits the fact that very few rows need a high refresh rate, and it tracks these inside the MC.

In software, the OS can increase the refresh period of the device by only allocating addresses that map to cells with sufficent retention. The Retention-Aware Placement in DRAM (RAPID) of Venkatesan et al. [36] and Refresh Incessantly but Occasionally (RIO) of Baek et al. [2] are two such **R**-based solutions that lower the device refresh rate by isolating physical page frames that require frequent refresh.

Tolerance. Applications like games, media processing, machine learning, and unstructured information analysis tolerate errors in portions of their data and still produce acceptably accurate results. Approximate computation [31, 8] exploits this approximate data to realize tradeoffs among performance, energy, and accuracy. Cells containing such error-tolerant **T** data need not be refreshed as often as those containing critical data. How many cells fall into this category depends on application characteristics.

Liu et al. [22] partition DRAM banks into critical and non-critical regions and extend the self-refresh time to refresh non-critical regions less frequently. Their solution, Flikker, targets smartphones, which keep DRAM in self-refresh mode when (frequently) idle, but a similar technique could be applied to auto refresh in operating mode. If workload characteristics change such that more data become critical, the DRAM must be repartitioned. This partitioning is coarse-grained (e.g., it requires regions to be 1/4, 1/2, or 3/4 the capacity), which simplifies hardware and reduces area overhead but misses opportunities for finer optimization.

Access Recency. DRAM accesses imply refresh operations, and so a subsequent refresh to the same row can be postponed. The number of rows affected depends on how many different rows are accessed within the maximum refresh period, which may be few for many workloads. Ghosh et al. [9] propose Smart Refresh, which divides the refresh period into phases and maintains a per-row timeout counter in the MC. The MC decrements the counters each phase and issues a RAS-only refresh when a counter hits zero.

Emma et al. [7] create smarter refresh policies for embedded DRAM caches: ECC provides error-tolerance and time stamps guide scheduling selective refreshes. Agrawal et al. [1] similarly target eDRAM caches with Refrint, employing eager writeback for seldom used lines in addition to tracking access recency. Since the number of rows in eDRAM is limited, overheads for tracking **A** information are much more tolerable than they are for main memory.

Validity. If the OS has not allocated the page frame, its data are meaningless, and refreshes to it are wasteful. Sev-

Table 2: Summary of Intelligent Refresh Approaches

Information	Technique	Information Tracked		Implementation
		acquisition	storage	
R (retention time)	T. Ohsawa et al. [29] (VRA)	profiling	DRAM	DRAM skips refresh
	J. Liu et al. [20] (RAIDR)		MC	MC skips refresh
	R. Venkatesan et al. [36] (RAPID)		OS	OS deletes pages
	S. Baek [2] (RIO)			
T (tolerance to data errors)	S. Liu et al. [22] (Flikker)	programer annotations	page number	DRAM is split into two refresh regions
A (access recency)	M. Ghosh and H. Lee [9] (smart)	MC tracking	MC	MC skips refresh
	P. Emma et al. [7]	cache controller tracking	cache controller	cache controller skips refresh
	A. Agrawal et al. [1] (refrint)			
V (validity of row data)	T. Ohsawa et al. [29] (SRA)	OS tracking	DRAM	DRAM skips refresh
	C. Isen and L. John [12] (ESKIMO)			
	S. Baek et al. [2] (PARIS)		MC	MC skips refresh

eral software approaches thus attempt to trigger refreshes only for rows with valid data. The effectiveness of schemes using such **V** refresh information is sensitive to the total memory usage of the system.

In addition to VRA, Ohsawa et al. [29] propose a Selective Refresh Architecture (SRA) that uses an **A** bit per row to decide whether to refresh it. This scheme modifies the ISA so that the compiler, OS, or MC can prevent refreshes to dead data. In their combined hardware/software ESKIMO approach, Isen and John [12] adapt SRA to track data significance (e.g., the values of uninitialized data in a newly allocated memory region are insignificant). The OS maintains information on allocation and deallocation of virtual addresses. Baek et al. [2] extract physical memory usage information from the OS instead of monitoring virtual addresses. Their Placement-Aware Refresh In Situ (PARIS) improves SRA by maintaining RD bits in the memory controller. Storage overheads can be reduced by tracking **V** bits for larger row granularities, but this imprecision increases unnecessary refreshes.

3.3 Implementation Issues

Not only do intelligent refresh schemes differ with respect to what refresh information they track — they differ in how they acquire that data, where they store them, and how they use them to decide when to skip refreshes (see Table 2).

The main drawback of most of these methods is that RD storage costs are proportional to memory capacity, potentially adding megabytes of SRAM to the cache controller or memory controller [9, 2] (as shown in Figure 3) or adding up to 20% area overhead to DRAM dies [29]. The economics of manufacturing commodity DRAMs may hinder adoption of such solutions.

In general, **R**-based approaches have the potential to perform well regardless of system load or application behavior, but their implementation poses challenges. **R**-based software solutions fragment memory, which can hurt performance and limit adoptability, since modern operating systems use physical superpages for the kernel, framebuffer, device drivers, and some application data regions. RIO [2] thus restricts deleted pages to be fewer than 0.1% of all page frames, which in turn limits its impact. Current **R**-based hardware solutions (ours included) statically measure retention distributions, but basing refresh-period lengths on offline information may hurt reliability because retention time fluctuates with time and temperature [37, 30, 19]. Obvious solutions are to somehow monitor retention times dynamically or to adopt overly conservative refresh periods (reducing the benefit of these optimizations). **T**, **A**, and **V** approaches are more reliable, but less broadly effective, depending on application behavior. **A** approaches grow less effective as DRAM density increases, since relatively few rows are likely to be accessed during any refresh period. Our study therefore does not address the use of **A** information.

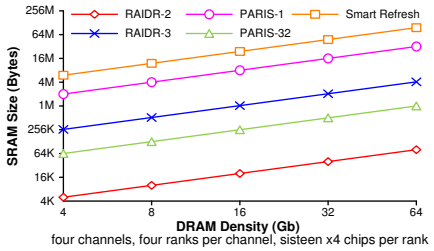


Figure 3: Comparative Storage Costs (instances of approaches from Table 2). RAIDR-n maintains n bloom filters; PARIS-m represents m rows per bit.

4. DTAIL OVERVIEW

Here we describe DTail, a framework that supports multiple refresh reduction techniques. We can mitigate the storage costs of refresh data by acting on two observations:

- only a small part of the RD collection is useful at any refresh decision point, and those points occur infrequently compared to normal memory accesses; and
- RD accesses exhibit high spatial locality, since most refresh schemes rely on a row address counter (RAC) to sequentially cycle through rows of a DRAM bank.

Based on the first observation, we choose not to store RD information within specialized registers or buffers within the memory controller because paying such high overheads for infrequently used data wastes area and power. We instead store the data within the DRAM itself, where, based on the second observation, we employ prefetching to hide the latency of accessing slower storage.

Figure 4 depicts the logical building blocks of DTail: the software and hardware negotiate a contiguous physical memory space in which to store RD entries for all rows. The refresh data must be acquired through software or hardware and then written to the RD space in main memory. As always, the memory controller periodically generates refresh commands, but here it uses the RAC of a potential refresh operation to access the RD entry for that row (already prefetched into the memory controller) to compute whether the potential refresh can be squashed. DTail can make use of any of the **RTV** information types discussed above, either alone or in combination.

Placing refresh information in DRAM avoids the need for a separate device or new interface, and it facilitates hardware/software coordination by allowing the memory con-

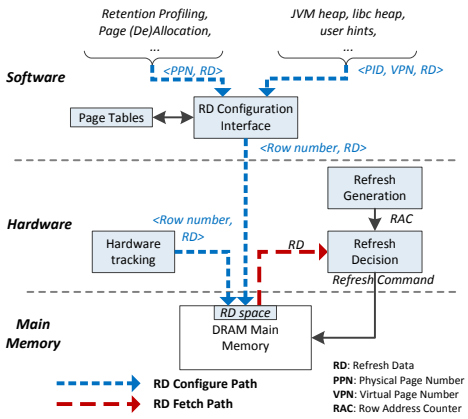


Figure 4: Building Blocks of DTail

troller, OS, compiler, and even the user to participate in refresh management.

The refresh decision logic in Figure 4 can take advantage of prefetched RD to predict the gain of different refresh methods and make the most efficient choice[4].

5. DESIGN DETAILS

In this section, we discuss the design of DTail, including changes to the OS, MC, and DRAM devices.

5.1 Maintaining Refresh Data

5.1.1 Refresh Data Acquisition

Prior work discusses how to acquire the **RTV** refresh data. For example, gathering retention time information requires profiling to determine which DRAM cells hold their data at several discrete refresh rates [2, 19]. The programmer must identify error-tolerant data to guide OS page allocation [22]. Given this requirement, we do not consider data error-tolerance in the current study.

Note that the refresh data may exist in any layer of the software stack. For example, an application (e.g., memcached or JVM) may allocate large chunks of memory at startup and then self-manage it (possibly with garbage collection). The OS may consider a page valid even if it is not currently assigned to an object. Figure 5 shows JVM heap usage for several Java applications (profiled every five seconds). Total execution times range from 50 seconds to 300 seconds, and each application's time is normalized to its total execution time. On average, the "true" valid data only account for 34.6-58.3% of the heap size.

5.1.2 Configuration Interface

DTail provides an easy RD configuration interface to store aquired refresh data of different types from various layers. Refresh data acquired by the OS (e.g., from page allocation/deallocation) can be tagged by the physical page number (PPN). This makes it straightforward to configure the RD space in memory, given the address mapping between rows and pages. If refresh data are acquired from upper layers (e.g., the JVM GC or user hints), a system call passes the process ID (PID) and virtual page number (VPN) to the OS. The OS indexes the page table to translate the tuple <PID, VPN> into the PPN. In case the virtual-to-physical translation is not yet established[5], we add a few bits to the page table entry (PTE) to temporarily store the RD while the OS maps the page.

[4]We do not address global scheduling of memory operations here, but DTail supports intelligent scheduling schemes.

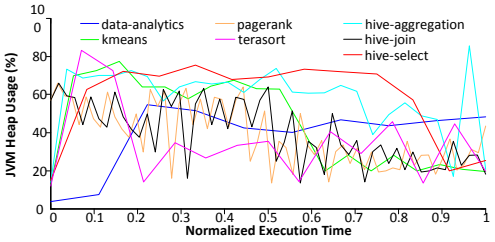


Figure 5: JVM Heap Usage

5.1.3 Refresh Data Storage

The RD entries for all rows can be stored in simple tables. Retention and validity information can be represented by the *expected refresh period* and a *valid bit*, respectively. Representing data error-tolerance as the expected refresh period simplifies the refresh decision logic; critical data require the normal period, but non-critical data are assigned relaxed refresh periods according to their error-tolerance levels.

Since both the expected refresh period and validity are seldom updated (at system boot time and upon events such as page allocation and garbage collection), we merge them, as in Table 3. These RD tables should be stored in contiguous physical memory that the OS allocates at boot time and for which it configures the MC (with physical addresses and lengths). This simplifies MC circuitry: simple state machines can control accesses to these dense structures.

Table 3: Refresh Data (RD) Format

RD (four bits)				expected refresh period
0	X	X	X	no refresh
1	0	0	0	64ms
1	0	0	1	2×64ms
1	0	1	0	4×64ms
	...	...		...
1	1	1	1	128×64ms

5.2 Deciding When to Refresh

The DTail hardware uses information in the RD tables to support different refresh reduction techniques: retention-aware multi-period refresh, validity-aware selective refresh, and tolerance-aware multi-period refresh. The MC generates both auto refresh and RAS-only refresh synchronously, and DTail dynamically selects which refresh method to use based on gain prediction. Figure 6 shows the decision process.

5.2.1 Per-Row Refresh Decisions

Since each RD entry is four bits, each RD access (typically 64 bytes) fetches multiple entries and buffers them in a FIFO within the MC. Simple next-line prefetching helps mask lookup latency.

When a RAS-only refresh command is generated, the refresh decision logic decides whether it will be issued based on the RD entry. If the refresh is not squashed, the expected refresh period (assumed here to be $N \times 64ms$) indicates how often a row must be refreshed. The decision logic in Figure 6 ensures that a refresh occurs once every N periods. The hash function avoids "refresh bursts" by distributing refresh oper-

[5]Linux uses demand paging, which allocates a physical page only when the virtual page is accessed.

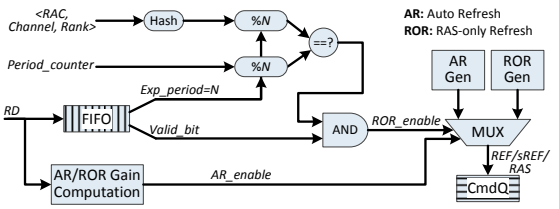


Figure 6: DTail Decision Process

ations for different rows over N periods and by distributing refreshes across the channels, ranks, and banks.

5.2.2 Refresh Choice

DTail can choose between RAS-only refresh and auto refresh. Table 1 shows that multiple rows across all banks — termed a *super-row* to simplify further dicussion — are refreshed on each auto refresh command. DTail predicts the most efficient way to refresh a super-row, issuing either one auto refresh command (refreshing all rows) or multiple RAS-only refresh commands (refreshing rows selectively). As per Section 2.3, auto refresh is more efficient when many rows need to be refreshed in high density DRAMs.

Whenever an RD entry is fetched into the MC, the DTail logic uses it to predict which type of refresh will generate the least overhead. Upon reaching the boundary of a super-row, the gain computation block in Figure 6 computes whether auto refresh should be enabled to refresh the current super-row, and it clears its internal state to compute the choice for the next super-row. If RAS-only refresh is indicated, the per-row RD entries are used to further decide which rows need refreshing. The FIFO depth is equal to the size of a super-row, so FIFO entries will not be consumed until the choice of refresh mode has been made.

This gain computation function can be arbitrarily complex. Our initial implementation checks whether the number of RAS-only refreshes exceeds a given threshold, and if so, it switches to auto refresh. Deriving a smarter algorithm that considers workload characteristics like memory intensiveness and bank-level parallelism is part of ongoing work.

5.2.3 Refresh Issue

If the gain computation indicates that auto refresh is better, a REF command is issued. Otherwise, RAS-only refreshes are used. Each DRAM maintains its own row address counter (RAC), so simply omitting the REF command leaves the RAC in DRAM pointing to the current row.

Here we propose a new refresh command — silent refresh, or $sREF$ — to be added to the DDRx protocol. Upon receiving an $sREF$, the DRAMs increase their RACs, but do not perform the refresh. This synchronizes the RAC values between the MC and the DRAMs. For RAS-only refresh, an unnecessary refresh can be eliminated by omitting the issue of the RAS command.

5.3 Overhead Analysis

DTail incurs storage overheads for the tracked refresh data and fetch-latency and traffic overheads for accessing the data.

Storage Overheads. DTail stores refresh data within the DRAM, which is much cheaper than SRAM or special-purpose registers. For RAS-only refresh, we keep one entry per row in the RD table. In our investigations, we assume four bits suffice to record **RTV** refresh data information for each 8KB DRAM row (1KB per chip and eight chips per rank). The capacity overhead is thus 4bits/8KB = 0.006%.

Fetch Overheads. The small size and high locality of RD table entries trigger minimal fetch traffic. For example, each memory read typically accesses 64 bytes containing 128 RD table entries. For a four-rank system using 4Gb devices at extended temperature ranges, 256 RD table entries are required (for all rows in a super-row per rank) every $tREFI$, which results in two memory reads every 3.9μs. These RD reads account for 0.77% of the total memory accesses, on average. As density increases, the traffic overhead may grow to 5.61% at 32Gb density. (Average traffic overhead is 3.51% for all but our least memory-intensive workloads.) Nonetheless, evaluation results show that DTail still performs better and saves more energy than prior mechanisms that access RD in SRAMs inside the MC. Sending more reads for each prefetch further reduces the fetch overhead by exploiting row buffer locality to save RAS operations.

Other Runtime Overheads. RD acquisition incurs a few other runtime overheads. For instance, the overhead of paging amounts to checking the status of several (OS) pages in the DRAM row and updating the row's RD, which involves several (but not a great many) memory accesses. For upper-layer RD collection mechanisms, the overheads are a system call to do virtual-to-physical mapping and then several memory accesses to update the RD in DRAM. (We have the kernel do the RD update for security.) The execution overhead depends on how frequently such events happen, and here we assume them to be infrequent. Nonetheless, runtime overheads must be managed properly. For instance, JVM divides the heap into young, old, and permanent generations. Tracking RD for the young generation would require frequent (kernel) address translations: an obvious optimization is to track RD only for the other generations. Addressing the young generation is ongoing work.

6. EVALUATION METHODOLOGY

We briefly describe our tools and workloads.

6.1 Simulation Setup

We evaluate our design space by running traces on the USIMM cycle-accurate memory simulator [5]. The original processor model considers only the reorder buffer and issue/retire widths. To generate reasonable access rates, we augment USIMM to model the load/store queue, multiple load/store ports, a three-level cache hierarchy (including MSHRs), and dependences between memory instructions.

We use Pin [23] to generate memory instruction traces with data dependence hints. We capture the read-after-write (RAW) relationships between non-memory instructions and pass the dependences through the CPU registers to generate memory instruction dependences (similarly to Zsim [32]). We omit write-after-read, write-after-write, and control dependences, and instead assume that ideal register renaming and branch prediction can fully resolve them. The original USIMM memory model uses DDR3 SDRAM. We minimally modify it to model the new features of the JEDEC DDR4 specification (such as bank groups).

Table 4 details the system configuration in our evaluations. We choose a 4:1 core-to-channel ratio, a likely configuration in contemporary high-end and future systems. Table 5 shows timing parameters from the JEDEC DDR4 specification, and Table 1 shows refresh-related parameters.

Table 4: System Configuration

Parameter	Detail
Number of cores	4
Core frequency	3.2GHz
ROB size	128
Issue/retire width	4/2
Pipeline depth	10
Load/store queue size	48/32
Load/store ports	2/1
L1 private Dcache	64KB per core, 4-cycle
L2 private cache	256KB per core, 10-cycle
L3 shared cache	4MB, 40-cycle
MSHR	8 entries per core
Read/write queue size	32/24
Scheduling policy	FC-FRFS, open page
Address mapping	row:rank:bank:column:offset
DRAM frequency	800MHz (DDR4-1600)
DRAM device width	x8
DRAM device density	4Gb/8Gb/16Gb/32Gb
Channels	1
DIMMs per channel	2
Ranks per DIMM	2
Banks per rank	16
Rows per bank	32768/65536/131072/262144
Columns per bank	8192

We enhance the Micron power model [25] to derive energy numbers. The model reflects the activating power supply (VPP) and Pseudo-Open Drain (POD) termination [33] of JEDEC DDR4. Table 6 shows the IDD/IPP currents of a 4Gb DDR4 DRAM device according to the Micron document [26]. We project the IDD/IPP currents of higher density devices: all IDD/IPP currents except IDD5B/IPP5B (burst refresh current) remain the same, assuming that technology scaling compensates for the costs of increasing device density and that the current consumed by auto-refresh increases 1.3× as device density doubles. ($tRFC$ grows much less than 2×, which means more rows are refreshed in the same amount of time, which increases current.)

Each simulation models a 1024ms execution (about 3.28 billion cycles at 3.2GHz). We adopt a constant-threshold-based low-power mode management method [6] that incurs less than 2% performance overhead and reduces background power by 35%. We report results at extended temperature ranges since the DRAM operating temperature is usually greater than 85 °C in servers and data centers [21].

6.2 Workloads

To create multiprogrammed workloads from the SPEC CPU2006 benchmark suite [11], we first classify the 29 applications according to memory intensity. We consider an application memory-intensive if the last-level cache misses per kilo instructions (MPKI) is over five. We randomly generate twelve four-program workloads for five intensity categories (based on the number of memory-intensive benchmarks in the mix), creating 60 workloads in total.

To evaluate retention-aware refresh, we extract the retention distribution for a 50nm technology node from Kim and Lee [16] and randomize the physical position of weak cells [2]. Table 7 shows the retention distribution for our evaluated DRAM system using 4Gb devices.

To evaluate validity-aware refresh, we allocate dummy pages in the simulator to model five memory capacity utilization levels. By allocating the desired memory on a real server with 32GB DRAM and scanning the Linux kernel's *mem_map* structure, we obtain the distribution of allocated pages for a 32GB capacity, and on this we base our derivations of distributions for other capacities.

Table 5: DDR4 Timing Parameters

Timing parameter	Value[a]	Timing parameter	Value
$tRCD$	11	$tWTR\_S$	2
tRP	11	$tWTR\_L$	6
$tCAS$	11	$tRTP$	6
tRC	39	$tCCD\_S$	4
$tRAS$	28	$tCCD\_L$	5
$tRRD\_S$	4	$tCWD$	5
$tRRD\_L$	5	$tRTRS$	2
$tFAW$	20	tXP	5
tWR	12	$tBURST$	4

[a] All timing parameters are in DRAM cycles.

Table 6: DDR4 Power Parameters of 4Gb Devices

Power parameter	IDD	IPP
Supply voltage (VDD/VPP)	1.2V	2.5V
One bank active-precharge current (IDD0/IPP0)	40mA	3mA
Precharge standby current (IDD2N/IPP2N)	30mA	1.8mA
Precharge power-down current (IDD2P/IPP2P)	17mA	1.8mA
Active standby current (IDD3N/IPP3N)	37mA	1.8mA
Active power-down current (IDD3P/IPP3P)	25mA	1.8mA
Burst read current (IDD4R/IPP4R)	125mA	1.8mA
Burst write current (IDD4W/IPP4W)	120mA	1.8mA
Burst refresh current (IDD5B/IPP5B)	110mA	15mA

Table 7: Retention Time Distribution of Evaluated DRAM System

Retention time	Probability of cells	Number of rows @ 4Gb
64-128ms	2.93E-10	40
128-256ms	7.78E-09	1069
256-512ms	1.53E-06	200078
512-1024ms	1.91E-05	1353119
1024-2048ms	3.42E-04	542846
>2048ms	˜1.00	0

6.3 Metrics

We use *Weighted Speedup* (WS) [34] to measure the system performance of multi-programmed workloads. We normalize each benchmark IPC to that of the solo execution IPC and calculate weighted speedup as the sum of all normalized benchmark IPCs. Since we run each simulation a fixed number of cycles, higher performing mechanisms execute more memory instructions and thus expend more energy. For a fair comparison, we use *Energy Per Access* (EPA) to report DRAM system power.

7. RESULTS

We compare the following mechanisms:

- RL: refresh-less, in which no refresh is performed.
- AR: the auto refresh method.
- ROR: the RAS-only refresh method.
- RAIDR: the Retention-Aware Intelligent DRAM Refresh method proposed by Liu et al. [20]. We track retention information in two Bloom filters (at reasonable storage costs — see Figure 3).
- PARIS: the Placement-Aware Refresh in Situ method proposed by Baek et al. [2], which skips refreshes for rows with invalid data. We use one bit per 32 rows (64 pages), which results in acceptable storage costs and an average performance overhead of 7.95%.[6]
- DTail-R: DTail with retention information.
- DTail-V: DTail with validity information.
- DTail-RV: DTail with combined retention and validity information.

[6] Defined as missed opportunities for refresh reduction. Representing more rows per bit reduces storage costs but increases the performance overhead to 14.8% for 64 rows and 28.2% for 128 rows, on average.

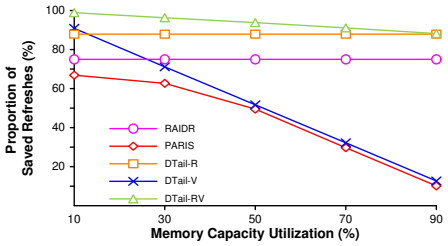
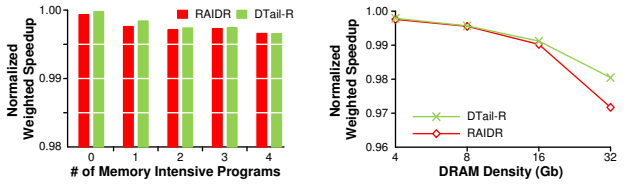
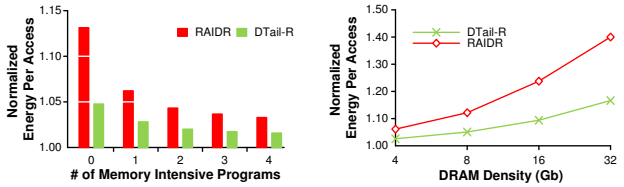


Figure 7: Refresh Reduction of Various Methods



(a) 4Gb (b) All Densities

Figure 8: Performance Comparison between RAIDR and DTail-R (higher is better)



(a) 4Gb (b) All Densities

Figure 9: Power Comparison between RAIDR and DTail-R (lower is better)

All performance (*weighted speedup*) and power (*energy per access*) results in this section are normalized to those of a refresh-less (RL) system. Recall that Figure 2 shows results for AR and ROR.

7.1 Overview of Refresh Reductions

Figure 7 gives an overview of the number of refreshes saved (compared to a system that always refreshes all rows) by various methods for 4Gb DRAMs under different memory utilization rates. Both RAIDR and DTail-R are based on the process variation of the DRAMs themselves, and thus their behaviors are independent of the utilization rate. RAIDR eliminates 75.0% of total refreshes, and DTail-R eliminates 87.9%. The additional 12.9% comes from DTail-R's storing complete per-row refresh data.

The number of refreshes saved by both PARIS and DTail-V decreases when more memory rows store valid data. DTail-V avoids more refreshes than PARIS, again because it stores complete per-row refresh data. The advantage of DTail-V is more significant at low memory utilization, e.g., an additional 24.0% and 8.45% of refreshes are saved at 10% and 30% memory utilization, respectively. Note that the default Linux buddy page allocator preserves a certain amount of continuity among allocated physical pages, and thus using one bit to represent a small number of pages (rows) results in modest overhead at high memory utilization. However, some advanced page allocation mechanisms [15] may break the continuity, increasing the overhead of PARIS.

By taking advantage of both retention and validity information, DTail-RV saves the most refreshes, from 88.2% at high memory utilization to 98.9% at low memory utilization.

7.2 Retention-Aware Methods

We compare DTail-R with RAIDR [20] to evaluate the retention-aware methods. Figure 8 and Figure 9 show the normalized weighted speedups and energy per access of both methods at 4Gb together with projections for higher densities. Results in the projected figures are averaged across all five categories of memory-intensiveness.

Figure 8(a) shows that RAIDR and DTail-R perform similarly to the ideal refresh-less configuration at 4Gb. RAIDR has already cut down 75% of the refreshes, and the performance impact of the remaining refreshes is negligible, especially when RAS-only refresh can utilize the DRAM bank-level parallelism. DTail-R saves an additional 12.9% of the refreshes, which makes its performance advantage grow as DRAM density increases (Figure 8(b)).

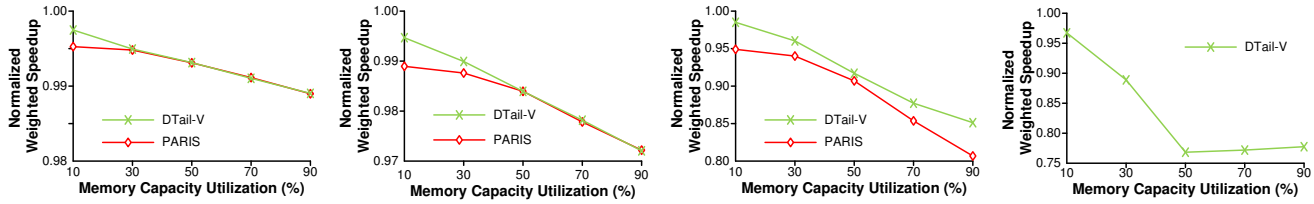
Although the "small" number of refreshes saved by DTail-R compared to RAIDR has little performance impact, the benefit in energy savings is apparent. Figure 9(a) shows that DTail-R consistently saves more energy: compared to RAIDR, DTail-R reduces the energy overhead by an average of 3.54% (up to 8.53%) at 4Gb. Figure 9(b) shows that the benefit grows with DRAM density: from 3.54% energy savings at 4Gb to 23.3% at 32Gb. DTail-R saves about half the refreshes that RAIDR does, but the energy reduction is a bit more than half. For example, at 32Gb the energy overheads of RAIDR and DTail-R compared to a refresh-less system are 40.0% and 16.7%, respectively. This additional energy savings comes from two factors: first, more refreshes force the DRAMs to exit the low power state for low-intensity workloads, increasing background power; second, refresh may unnecessarily precharge open rows that will be reused in the near future, increasing the number of RAS operations and the corresponding energy consumption.

7.3 Validity-Aware Methods

To evaluate the validity-aware methods, we compare DTail-V with PARIS [2]. Figure 10 and Figure 11 show the normalized weighted speedup and energy per access of the two methods at different memory utilizations for various densities. Results are again averaged across all five memory-intensity categories (60 workloads). PARIS fails to work at 32Gb because there is insufficient time to issue all the commands for RAS-only refresh.

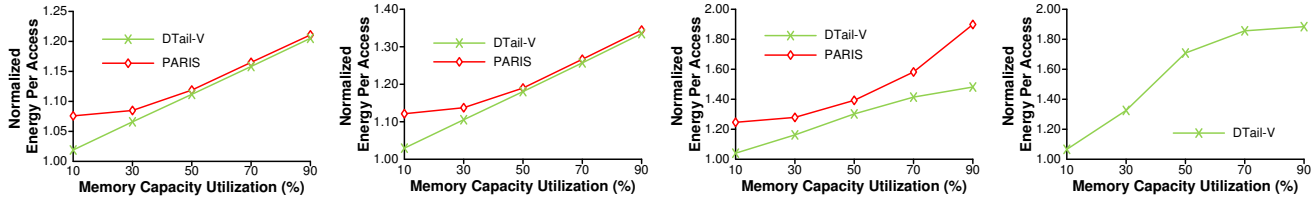
Performance decreases and energy consumption increases as the portion of valid DRAM rows grows. For example, Figure 10(c) and Figure 11(c) show that at the 16Gb density the normalized weighted speedup decreases by 14.2% and energy per access increases by 65.2% for PARIS when we go from 10% to 90% memory utilization.

Figure 7 shows that DTail-V saves more refreshes when memory utilization is low (10-30%) because it stores complete refresh data. This contributes to higher performance (Figure 10) and lower energy consumption (Figure 11) at low memory utilization. However, Figure 10(c) and Figure 11(c) show that DTail-V also performs better at high memory utilizations for higher density DRAMs. For example, with 90% memory utilization at 16Gb, DTail-V improves performance by 4.44% and saves energy by 41.7% compared to PARIS. This is because DTail-V dynamically detects the system page placement behavior and chooses between auto



(a) 4Gb (b) 8Gb (c) 16Gb (d) 32Gb

Figure 10: Performance Comparison between PARIS and DTail-V for Various Densities (higher is better)



(a) 4Gb (b) 8Gb (c) 16Gb (d) 32Gb

Figure 11: Power Comparison between PARIS and DTail-V for Various Densities (lower is better)

refresh and RAS-only refresh. DTail-V's lack of performance improvement at 50%, 70%, and 90% utilization for 32Gb density is due to our heuristic switching mechanism.

7.4 Combining R and V Information

DTail is easy to reconfigure to adopt both retention and validity information. We call this implementation DTail-RV. Figure 12 and Figure 13 show the performance and energy benefits of DTail-RV at different memory utilizations for various densities. We compare to RAIDR [20] for retention-awareness and to PARIS [2] for validity-awareness. Results are averaged across all five memory-intensivity categories (60 workloads). Recall that PARIS fails to work for 32Gb.

Figure 12 shows that RAIDR and DTail-RV perform close to an ideal refresh-less system. DTail-RV is slightly better than RAIDR, especially at higher densities.

Figure 13 shows that DTail-RV can save significantly more energy than RAIDR and PARIS. Compared to RAIDR and PARIS, DTail-RV reduces energy by 15.8-23.8% and 24.6-81.9% (depending on memory utilization), respectively, for 16Gb DRAMs. For 32Gb DRAMs, the energy savings of DTail-RV relative to RAIDR ranges from 40.0% at 10% memory utilization to 25.9% at 90% memory utilization.

8. CONCLUSION

We have introduced DTail, a low-overhead DRAM refresh-management scheme that stores refresh information within the DRAM itself. This small, intuitive innovation delivers high payoff. The capacity overhead is negligible compared to growing DRAM capacities, and the latency of relatively infrequent accesses is easily masked by prefetching.

Technology trends make frameworks like DTail that store refresh data at negligible cost increasingly attractive. Consider emerging high-density 3D-Stacked DRAM technologies whose (many) more rows greatly increase the amount of refresh data to track. Furthermore, 3D-stacked technologies face thermal challenges that increase leakage, which in turn increases the required refresh frequency (e.g., an 8ms refresh period at 95-115 °C). This raises the performance and energy overheads of performing refresh.

Finally, storing the data in regular memory makes it accessible to all levels of the software stack: making it easier for hardware and software to collaborate to minimize refresh performance and energy overheads opens up an interesting design space of hybrid refresh approaches.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported by the National Natural Science Foundation of China (NSFC) under grant number 61221062, 61272132, and 61331008, the National Basic Research Program of China (973 Program) under grant number 2011CB302502, the Strategic Priority Research Program of the Chinese Academy of Sciences under grant number XDA06010401, and Huawei Research Program under grant number YBCB2011030. Yungang Bao is partially supported by a CCF-Intel Young Faculty Research Program (YFRP) grant.

10. REFERENCES

[1] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas. Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies. In *High-Performance Computer Architecture*, pages 400–411, Feb 2013.

[2] S. Baek, S. Cho, and R. Melhem. Refresh now and then. *IEEE Transactions on Computers*, 2013.

[3] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

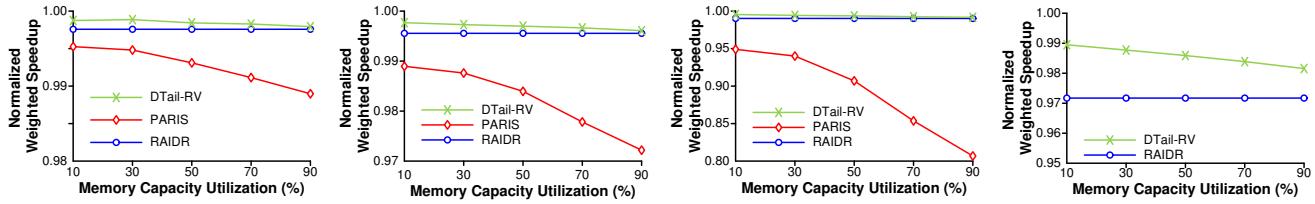
[4] S. Borkar. The exascale challenge. In *Keynote of Parallel Architectures and Compilation Techniques*, Sep 2011.

[5] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the utah simulated memory module. *University of Utah, Techincal Report UUCS-12-002*, 2012.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin. DRAM energy management using software and hardware directed power mode control. In *High-Performance Computer Architecture*, pages 159–169, Jan 2001.

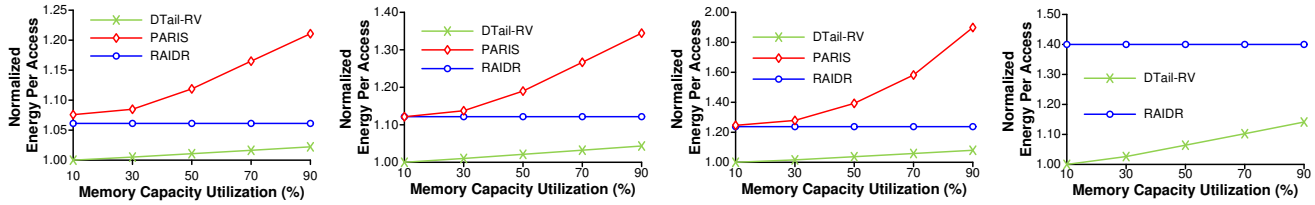
[7] P. Emma, W. Reohr, and M. Meterelliyoz. Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications. *IEEE Micro*, 28(6):47–56, 2008.

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Architectural Support for Programming Languages and Operating Systems*, pages 301–312, Mar 2012.



(a) 4Gb (b) 8Gb (c) 16Gb (d) 32Gb

Figure 12: Performance Comparison between RAIDR, PARIS and DTail-RV for Various Densities (higher is better)



(a) 4Gb (b) 8Gb (c) 16Gb (d) 32Gb

Figure 13: Power Comparison between RAIDR, PARIS and DTail-RV for Various Densities (lower is better)

[9] M. Ghosh and H. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked drams. In *MICRO*, pages 134–145, Dec 2007.

[10] T. Hamamoto, S. Sugiura, and S. Sawada. On the retention time distribution of dynamic random access memory (DRAM). *IEEE Transactions on Electron Devices*, 45(6):1300–1309, 1998.

[11] J. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[12] C. Isen and L. John. ESKIMO: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *MICRO*, pages 337–346, Dec 2009.

[13] JEDEC. JESD79-3E: DDR3 SDRAM specification, 2010.

[14] JEDEC. JESD79-4: DDR4 SDRAM specification, 2012.

[15] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[16] K. Kim and J. Lee. A new investigation of data retention time in truly nanoscaled DRAMs. *IEEE Electron Device Letters*, 30(8):846–848, 2009.

[17] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A case for exploiting subarray-level parallelism SALP in DRAM. In *International Symposium on Computer Architecture*, pages 368–379, Jun 2012.

[18] C. Lefurgy, K. Rajamani, F. Rawson, M. Kistler, and T. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.

[19] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms. In *International Symposium on Computer Architecture*, pages 60–71, Jun 2013.

[20] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In *International Symposium on Computer Architecture*, pages 1–12, Jun 2012.

[21] S. Liu, B. Leung, A. Neckar, S. Memik, G. Memik, and N. Hardavellas. Hardware/software techniques for DRAM thermal management. In *High-Performance Computer Architecture*, pages 515–525, Feb 2011.

[22] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *Architectural Support for Programming Languages and Operating Systems*, pages 213–224, Mar 2011.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Jun 2005.

[24] Micron. TN-04-30: Various methods of DRAM refresh, 1999.

[25] Micron. TN-41-01: Calculating memory system power for DDR3, 2007.

[26] Micron. DRAM memory in high-speed digital designs. `http://www.home.agilent.com/upload/cmc_upload/All/5Micron.pdf`, 2013.

[27] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *International Symposium on Computer Architecture*, pages 48–59, Jun 2013.

[28] P. Nair, C. Chou, and M. Qureshi. A case for refresh pausing in DRAM memory systems. In *High-Performance Computer Architecture*, pages 627–638, Feb 2013.

[29] T. Ohsawa, K. Kai, and K. Murakami. Optimizing the DRAM refresh count for merged DRAM/logic LSIs. In *International Symposium on Low Power Electronics and Design*, pages 82–87, Aug 1998.

[30] P. Restle, J. Park, and B. Lloyd. DRAM variable retention time. In *International Electron Devices Meeting*, pages 807–810, Dec 1992.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation*, pages 164–174, Jun 2011.

[32] D. Sanchez and C. Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *International Symposium on Computer Architecture*, pages 475–486, Jun 2013.

[33] J.-H. Shin, S.-I. Kim, Y.-M. Ahn, Y.-K. Han, and S.-J. Seo. Methodology on power estimation of memory modules with pseudo-open drain and center-tab termination type termination schemes. In *Open Server Summit*, Nov 2011.

[34] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, 35(11):234–244, 2000.

[35] J. Stuecheli and D. Kaseridis. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *MICRO*, pages 375–384, Dec 2010.

[36] R. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM. In *High-Performance Computer Architecture*, pages 155–165, Feb 2006.

[37] D. Yaney, C. Lu, R. Kohler, M. Kelly, and J. Nelson. A meta-stable leakage phenomenon in DRAM charge storage—variable hold time. In *International Electron Devices Meeting*, pages 336–339, Dec 1987.