Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory

Jeffrey Stuecheli[§‡], Dimitris Kaseridis[§], Hillery C. Hunter[*] and Lizy K. John[§]
[§]Electrical & Computer Engineering Department
The University of Texas at Austin, Austin, TX, USA
Email: {stuechel, kaseridi, ljohn}@ece.utexas.edu
[‡]IBM Corp., Austin, TX, USA
[*]IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA
Email: hhunter@us.ibm.com

***Abstract*—High density memory is becoming more important as many execution streams are consolidated onto single chip many-core processors. DRAM is ubiquitous as a main memory technology, but while DRAM's per-chip density and frequency continue to scale, the time required to refresh its dynamic cells has grown at an alarming rate. This paper shows how currently-employed methods to schedule refresh operations are ineffective in mitigating the significant performance degradation caused by longer refresh times. Current approaches are deficient – they do not effectively exploit the flexibility of DRAMs to postpone refresh operations. This work proposes dynamically reconfigurable predictive mechanisms that exploit the full dynamic range allowed in the JEDEC DDRx SDRAM specifications. The proposed mechanisms are shown to mitigate much of the penalties seen with dense DRAM devices. We refer to the overall scheme as Elastic Refresh, in that the refresh policy is stretched to fit the currently executing workload, such that the maximum benefit of the DRAM flexibility is realized.**

We extend the GEMS on SIMICS tool-set to include Elastic Refresh. Simulations show the proposed solution provides a ∼10% average performance improvement over existing techniques across the entire SPEC CPU suite, and up to a 41% improvement for certain workloads.

I. Introduction

Since its invention in the 1970's, the dynamic memory cell has become an indispensable part of modern computer systems. From smartphones to mainframes, these simple one-transistor-one-capacitor structures provide data storage which is fast (as compared to disk) and dense (as compared to on-processor SRAM memory). In the server space, JEDEC-standardized DDR3 DRAMs are currently prevalent, and DDR4 is forecast to emerge within the next several years [1].

The frequency and power "walls" of silicon logic technology scaling have been broadly discussed in recent literature, and processor designs have been accordingly retargeted for throughput, ushering in the many-core era. In contrast, DRAM, which is manufactured with a different set of technology steps, and which runs at lower frequencies than high-performance logic, has continued to scale in several ways, providing roughly 2x bit-density every 2 years, and stretching to twice its current frequency within the next 3-4 years [2]. For example, the specified frequencies for DDR, DDR2, DDR3, and DDR4 are 200–400 MHz, 400–1066 MHz, 800–1600 MHz, and 1600–3200 MHz, respectively. Despite this optimism, examining bit-density and frequency does not tell the full story; one must also consider the overhead of *maintaining* these cells' stored values. While DRAM has not been declared to have met a "scaling wall," manufacturers are continually challenged to find new materials and processes to create hi-capacity capacitors, small/fast/low-leakage access transistors, and robust means of supplying power in commodity system environments. Each of these challenges, along with the total number of bits per DRAM chip, directly impact the specification of DRAM *refresh*, the process by which cells' values are kept readable.

In this paper, we identify a troublesome trend in DRAM refresh characteristics and show how refresh impacts performance, especially for many-core processors. We propose a new approach to refresh scheduling and provide dynamic, configurable algorithms which more effectively address the "refresh wall" than approaches commonly employed in today's memory controllers.

II. Motivation

In order to retain the contents of dynamic memory, `refresh` operations must be periodically issued. JEDEC-standard DRAMs maintain an internal counter which designates the next segment of the chip to be refreshed, and the processor memory controller simply issues an address-less `refresh` command. As more bits have been added to each DRAM chip, changes have occurred in two key JEDEC parameters—`tREFI` and `tRFC`—which specify the interval at which `refresh` commands must be sent to each DRAM and the amount of time that each `refresh` ties-up the DRAM interface, respectively.

Most prior work on memory controller scheduling algorithms has assumed that refresh operations are simply sent whenever the `tREFI`-dictated "refresh timer" expires. This is a sufficient assumption for historical systems, where refresh overhead is relatively low, *i.e.* refresh completes quickly, and does not block `read` and `write` commands for very long. However, for the 4Gb DRAM chips which have been recently demonstrated [3], and would be anticipated to appear on the

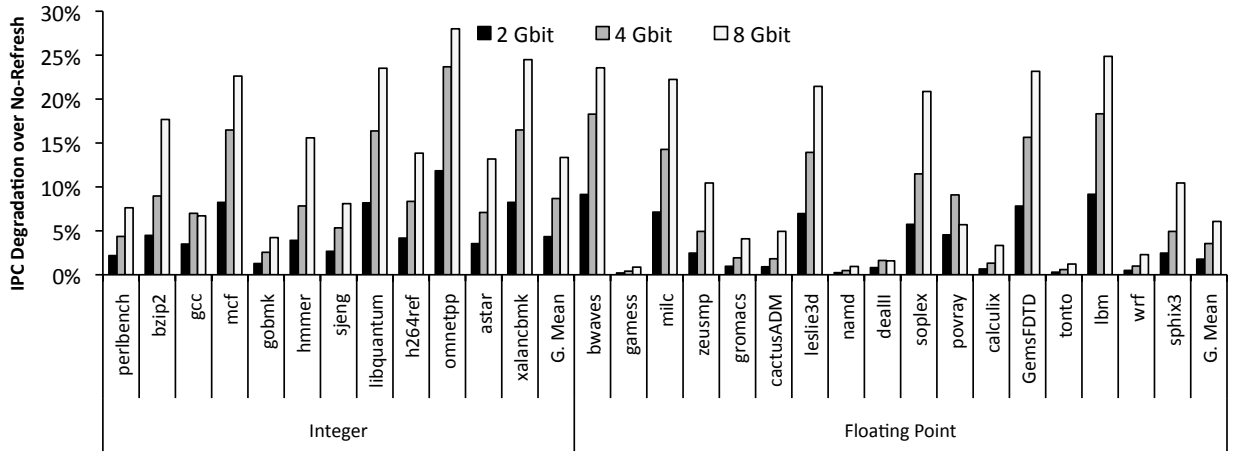


Fig. 1. Refresh performance penalty for emerging DRAM sizes (four-core). See Section V-A for a description of the modeled architecture.

mass market soon, a `refresh` command takes a very long time to complete (300ns). The net effect is a measurable increase in *effective memory latency*, as reads and writes are forced to stall while `refresh` operations complete in the DRAM. The baseline performance impact of 2Gb, 4Gb, and 8Gb chips is shown across the Spec2006 benchmark suite [17] in Figure 1, normalized to application performance when run without DRAM `refresh` commands. This penalty grows from negligible to quite severe: up to 30% for memory latency sensitive workloads with a geometric mean of 13% for integer and 6% for floating point. As denser memory chips come to market, this problem will only become worse [4].

A. DRAM Refresh Requirements and Thermal Environment of Modern Servers

The temperature at which a device is operated significantly impacts its leakage. For DRAM cells, which consist of a storage capacitor gated by an access transistor, their ability to retain charge is directly related to leakage through the transistor, and thus to temperature. While processors have hit a power-related "frequency wall," and have stopped scaling their clock rates, DRAMs have continued to be offered at faster speeds, resulting in increased DRAM power dissipation. At the same time, server designs have become increasingly dense (*e.g.*, the popularity of blade form-factors), and so main memory is increasingly thermally-challenged. The baseline server DRAM operating temperature range is 0°C – 85°C, but the JEDEC standard now includes an *extended temperature range* of (85°C – 95°C), and this has become the common realm of server operation [5], [6]. In this extended range, DRAM retention time is specified to be one-half that of the standard thermal environment.

In the standard thermal range, each DRAM cell requires a refresh every 64ms. As the memory controller issues `refresh` operations, the DRAM's internal refresh control logic sequentially steps through all addresses, ensuring that all rows in the DRAM are refreshed within this 64ms interval. The rate at which the memory controller must issue `refresh`es was initially determined by dividing 64ms by the number of rows in the DRAM. This value, referred to as `tREFI` (REFresh Interval), was specified to be 7.8μs for 256M DDR2 DRAM. As DRAM density doubles every several lithography generations, the number of rows also doubles. As such, using this traditional method, the rate at which `refresh` commands must be sent would need to double with each generation.

Instead, in order to reduce the volume of `refresh` traffic, DRAM vendors have designed their devices such that multiple rows are refreshed with one command [7]. While this does reduce the command bandwidth, the time required to execute a `refresh` increases with each generation, as more bits are handled in response to each `refresh` command [4]. Ideally, DRAM devices would simply refresh more bits with each operation, but this would over-tax the current delivery available. The length of time of this delay is the parameter `tRFC` (ReFresh Cycle time). Table I shows the worsening of `tRFC` as DRAMs become more dense, along with the impact of temperature on `tREFI`. Note that initially the increase in `tRFC` was significantly less than 2x (*i.e.*, 512Mb to 1Gb). This was possible due to constant-time aspects of refresh such as decoding the command and initiating the engine.

TABLE I
REFRESH PARAMETERS AS DENSITY INCREASES [8]

DRAM type	tRFC	tREFI@85°C	tREFI@95°C
512Mb	90ns	7.8μs	3.9μs
1Gb	110ns	7.8μs	3.9μs
2Gb	160ns	7.8μs	3.9μs
4Gb	300ns	7.8μs	3.9μs
8Gb	350ns	7.8μs	3.9μs

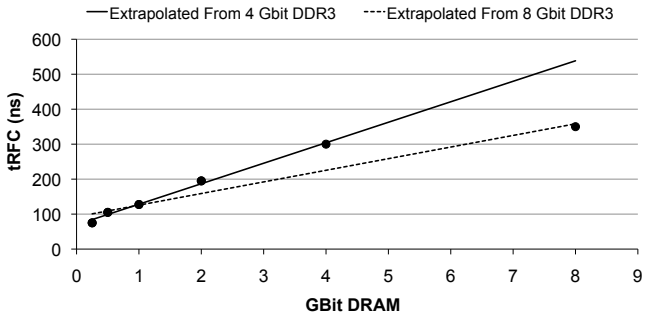


Fig. 2. tRFC Across DDR3 Generations

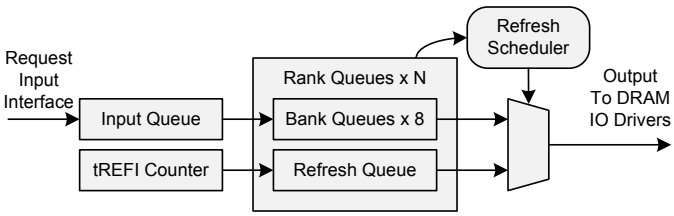


Fig. 3. Baseline Memory Controller

B. Refresh Cycle Time Beyond JEDEC DDR3

Table I contains the JEDEC DDR3 `tRFC` values. Projections to future values are difficult due to the seeming discontinuity between the trend lines shown in Figure 2. The linear regression from 512Mbit–4 Gbit would project 550 ns for 8 Gbit. The actual DDR3 JEDEC value is specified at 350 ns. There is debate in the DRAM community as to what `tRFC` values will be required for even higher density DDR4 memory, especially as new materials must be used to scale DRAM to higher densities and lower lithographies.

III. Background

A. Effective Memory Latency

Baseline memory latency is generally quoted as the time from a `load`'s issue until data is returned (from DRAMs in main memory) to the load/store unit. While average or cold-start metrics are sometimes used, "memory latency" is more commonly an optimistic/lower-bound value, and actual memory commands can be delayed by many architectural and system-level factors. This brings about the concept of *effective memory latency*, which is an average `load` service time from memory, and is impacted by collisions at, and delays in, queues, buses, DRAM banks, and other physical resources. These factors can be significant, and must be included for accurate performance projection [9]. In this paper, we focus on the emerging impact of `refresh` on effective memory latency.

Memory-Level Parallelism (MLP) is the degree to which computation can continue on a processor, despite delays as data is fetched from memory. For workloads with low MLP, memory latency becomes a significant contributor to overall performance, and we later demonstrate that such workloads require new approaches to refresh scheduling, in order to avoid detrimental effects of future DRAM refresh durations.

B. Deferral of Refresh Operations

The JEDEC DDRx standards allow flexibility in the spacing of `refresh` operations. Delaying a specific command for small numbers of `tREFI` periods does not result in loss of data, assuming the overall average refresh rate is maintained (*i.e.*, all bits of the DRAM are touched within their retention time). For this reason, commodity DRAMs allow deferral of some number of `refresh` operations, presuming that the memory controller then "catches up" when the maximum deferral count is reached. For the current DDR3 standard, this maximum `refresh` deferral count is eight [8]. In this work we use the term *postponed* to describe the number of `tREFI` intervals across which a `refresh` operation was deferred. Exploiting this *elasticity* in the scheduling of `refresh` operations is the key focus of this work.

C. Baseline Memory Controller

Figure 3 shows the queue structure of the memory controller used in this work. The `read` and `write` operations accepted by the controller from the CPUs (via the cache controller) are first placed in the Input Queue. Operations are moved to the appropriate Bank Queue as space is available. In our analysis in Section V-A, we specified 32 entries for each of these queues. The memory controller must also execute `refresh` operations; these are created as the `tREFI` counter expires, and stored in the Refresh Queue until they are executed. Selection between the various operations in the Bank Queues and the Refresh Queue is managed by the overall memory scheduler, of which only the Refresh Scheduler is shown. The Refresh Scheduler is explicitly shown, as the focus of this work explores the policy and priority with which `refresh` operations are intersperse with `read` and `write` requests.

D. Typical Approach to Refresh Scheduling

As previously suggested, most memory controllers have paid little attention to the scheduling of `refresh` commands, as the penalties have not warranted the complexity of a sophisticated algorithm. In this section, we examine current policies, referring to the memory controller logic which decides when to issue `refresh` commands as the *refresh scheduler* (shown in Figure 3).

The most straight-forward refresh scheduling algorithm simply forces a `refresh` operation to be sent as soon as the `tREFI` interval expires. This approach is commonplace due to the simplicity of the required hardware control logic. Historically, `tRFC` penalties were low enough to not warrant additional complexity. This algorithm can be found in readily-available memory simulators, such as DRAMsim [10] and GEMS [11]. In addition, even work dealing in sophisticated operation schedulers have employed this method [12]. This paper refers to this common policy as *Demand Refresh* (DR).

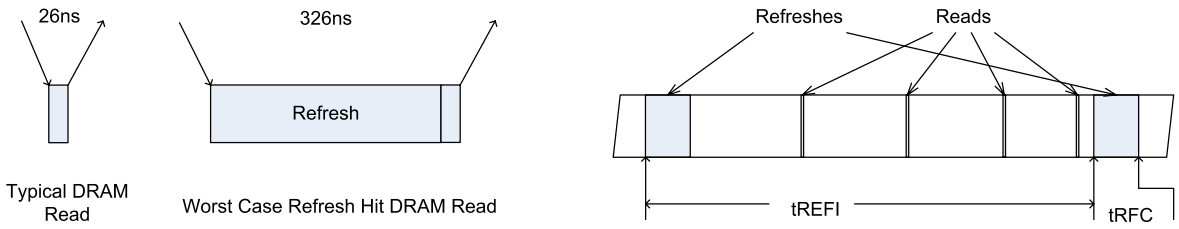


Fig. 4. Refresh Latency Penalty Example

In a more sophisticated policy that exploits the ability to postpone refresh commands [13], `refresh` operations are treated as low priority (never chosen over `read` or `write` traffic) until the postponed count reaches seven `refresh` operations. At this point, refreshes become higher priority than all other operations, to ensure the maximum deferral limit (eight) is not reached. Deferral-based designs do enable bursts of operations to proceed without refresh penalties, but as described in the next section, they fall short of isolating refresh penalties in several important scenarios. We refer to this policy as *Defer Until Empty* (DUE).

E. Examples of Where Typical Approaches Break Down

In the following sections, we describe several examples in which current refresh scheduling approaches fail to isolate `refresh` operations, including cases where ample idle DRAM cycles are available.

1) Low-MLP Workloads: Traditional approaches behave poorly while running low-MLP workloads. In low-MLP workloads, memory utilization is often quite light, but each reference to memory is critical to the workload's execution progress. A classic example of such an algorithm/workload is the traversal of pointer-based large data structures. For these applications, each execution thread generates only one miss to memory at a time. As such, there are many periods of time where the memory controller Bank Queues are empty. In these cases, the refresh scheduler will often execute refreshes immediately when the `tREFI` counter expires. The problem is that even though the scheduler is often empty, memory traffic is still present. This, combined with the very long refresh completion delay of high-density DRAMs (300 ns+), results in large penalties for operations received by the memory controller in the interval after the `refresh` was scheduled.

The magnitude of this effect is significantly larger than expected when only considering the fraction of time the DRAM is executing the `refresh`. In Figure 4(a), we graphically show the magnitude of the delay experienced by a `read` received just after a 300ns `refresh` operation of 4Gbit DRAM, compared to the typical closed page access latency of 26ns to accomplish a typical read operation. In Figure 4(b), we show graphically the fraction of time the DRAM bus is executing `refresh` operations over a `tREFI` interval. DRAM `read` operations are shown to give a scale of the relative bus busy time. This disproportionate busy time drives the very significant latency penalties.

Table II shows the first-order refresh-associated performance penalties across DRAM types. Bandwidth overhead is calculated by taking the refresh time (`tRFC`) over the refresh interval (`tREFI`). This gives the fraction of time that a DRAM chip is off-line from mainline traffic to execute `refresh` operations. This grows to over a 9% bandwidth tax in the densest DDR3 technology.

The latency overhead of refresh is more disruptive. To illustrate this, the first-order latency overhead, as shown in the fourth and sixth columns of Table II, is calculated assuming an idle system. In an idle system, a `read` request would incur a latency penalty if the DRAM scheduler had recently sent a refresh request to the needed DRAM device. Note that, in general, the scheduler would delay a `refresh` if a `read` operation was queued; the values shown represent the case where the `read` is unlucky. In this case, the latency penalty is on average one-half the `tRFC` time. The rate at which this higher effective `read` latency event occurs is indicated by the bandwidth overhead calculation. As Table II illustrates, this latency penalty can be very significant. For example, a modern processor might achieve a baseline memory latency of $\sim$50ns. For 8Gb DRAM, the penalty of 15.7ns represents a 31% memory latency increase due to refresh. Beyond the sheer magnitude of a 31% latency penalty, the cost in performance is higher in modern, speculative, out-of-order processors than the average latencies implies [9]. While in general memory latency can be hidden through hardware features such as prefetch and out-of-order execution, the reach of such mechanisms is limited by total hardware capacity. As such, designing for high latency events requires much larger structures than needed when latency is more uniform.

2) Medium to High Utilization Workloads: The general problem of refresh penalties due to scheduler inefficiencies also applies to workloads with high DRAM bus utilization. While the refresh timer may expire when the operation queues are not empty, in many cases the memory controller becomes idle for at least some period of time relatively soon compared to the `tREFI` interval. Though the bus may be idle, new operations could arrive shortly after the `refresh` is sent, incurring the large refresh penalty. Current designs do nothing to judge how long the controller will be empty, and are ineffective at avoiding these penalties. Our analysis indicates that traditional refresh deferral solutions reach significant backlogs only in

TABLE II
REFRESH PENALTY AS DENSITY INCREASES

DDR3 DRAM capacity	tRFC	bandwidth overhead (85°C)	latency overhead (85°C)	bandwidth overhead (95°C)	latency overhead (95°C)
512Mb	90ns	1.3%	0.7ns	2.7%	1.4ns
1Gb	110ns	1.6%	1.0ns	3.3%	2.1ns
2Gb	160ns	2.5%	2.4ns	5.0%	4.9ns
4Gb	300ns	3.8%	5.8ns	7.7%	11.5ns
8Gb	350ns	4.5%	7.9ns	9%	15.7ns

workloads with saturated memory buses. In these cases, the refresh scheduler is constantly forcing `refresh` operations, since there are never free intervals to hide the refresh.

F. Refresh Beyond DDRx SDRAM

In addition to the emerging `tRFC` penalties we have identified for dense commodity DRAM, there has been much interest in non-DRAM memory technologies which may come to market in the next 10 years (such as PCM, RRAM, and STT-RAM). Many recent works have assumed a primary advantage of these technologies is their non-volatility. While these are indeed "non-volatile" technologies at traditional Flash temperatures ($\leq$ 55°C), several of these suffer from accelerated *drift effects* at temperatures in the range of server main memory ($\leq$ 95°C) [14]. Drift causes a change in the memory cell's resistance value. While drift may be manageable in the initial single-bit-per-cell PCM implementations which are currently on the market, dense multi-level cell PCM relies on storing and sensing finer resistance granularities, and drift will become more of an issue. Dense, multi-bit implementations which are currently envisioned for hybrid and tiered memory systems, are thus likely to require a refresh-like command to combat drift in high-temperature server environments. The length of such an operation may be similar to these technologies' write/programming times (much longer than DRAM, generally). For one leading emerging memory contender, phase-change memory, its write time could result in a drift-compensating `tRFC` easily 3x that currently specified for DRAMs. From the above, it is clear that simple refresh scheduling mechanisms will not be sufficient for future memory.

IV. ELASTIC REFRESH SCHEDULING

We address the behavior observed in current refresh scheduling algorithms by decreasing the aggressiveness with which `refresh` operations are scheduled. In being less aggressive, the proposed mechanisms more effectively exploit the available refresh deferral dynamic range. This is accomplished by waiting to issue a `refresh` command, even when the bus is idle. At the most fundamental level, we use predictive mechanisms that decrease the probability of a `read` or `write`'s collision with a recently issued `refresh` operation.

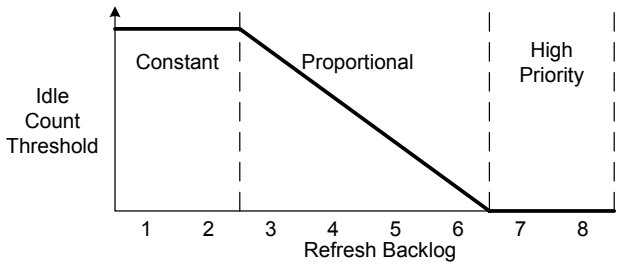


Fig. 5. Idle Delay Function (IDF)

The *Elastic Refresh* algorithm we propose differs from the best existing approach (DUE) in the mechanism used to issue low priority `refresh` operations. Current mechanisms consider low priority `refresh` operations eligible to be sent when all Bank Queues for a rank are empty (structure in Figure 3). In our method we wait an additional period of time for the rank to be idle before issuing the `refresh` command. The usage of this additional delay, effectively lowering refresh priority further, exploits typical system behavior where memory operations arrive in bursts. Using this assumption, as the time since a prior operation increases, the probability of receiving future memory operations decreases. This reduces the likelihood that a new operation will collide with an executing `refresh`. We extend this idea with the following observation: at low postponed `refresh` counts, the prediction can aggressively choose to not send an operation. As the postponed `refresh` count increases, this bias is reduced by decreasing the idle delay period.

A. Idle Delay Function

We can express the idle delay as a function of the `refresh` postponed count. The general form of this function, referred to as the *Idle Delay Function* (IDF), is shown in Figure 5. Note, in our proposal, the parameters of the IDF are dynamically adjusted based on the workload characteristics. We define three regions of delay characteristics:

1) *Constant*: In our analysis, we found many workloads have a characteristic idle delay period, where the probability of receiving a future command in the `tRFC` interval is very low. The constant region effectively sets the maximum IDF at this value.
2) *Proportional*: This region represents the area where the postponed refresh count approaches the maximum allowed value, and we must begin to more aggressive issuing of `refresh` operations. The slope of the proportional region is tuned such that the full dynamic range of postponed operations is exploited.
3) *High Priority*: As the number of postponed requests approaches the maximum, the delay strategy must be abandoned, as the `refresh` must be issued within one additional `tREFI` interval. From this perspective, the High Priority region has two phases, both with an idle delay of zero. At a count of seven, the scheduler will send the `refresh` as the bank queue becomes empty. At a

TABLE III
IDLE DELAY FUNCTION PARAMETERS

Parameter	Units	Description
Max Delay	Memory Clocks	Sets the delay in the constant region
Proportional Slope	Memory Clocks Postponed Step	Sets slope of the proportional region
High Priority Pivot	Postponed Step	Point where the idle delay goes to zero

count of eight, the `refresh` will be sent before any other commands, as soon as the DRAM bus parameters allow.

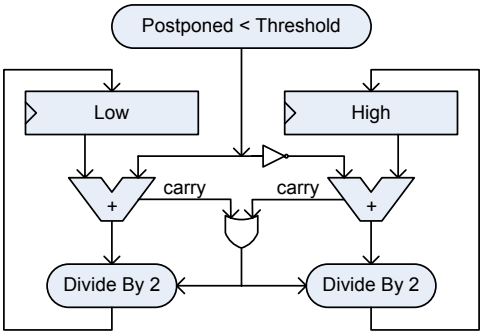
B. Idle Delay Function Control

As the optimal characteristics of the idle delay function are workload-dependent, we must define a set of parameters to configure the delay equation. These are listed in Table III. The Max Delay and Proportional Slope parameters are determined with the use of two hardware structures that profile the references. The High Priority Pivot (the transition from Proportional to High Priority) is fixed at seven postponed refreshes, as this was effective to prevent forcing High Priority unnecessarily.

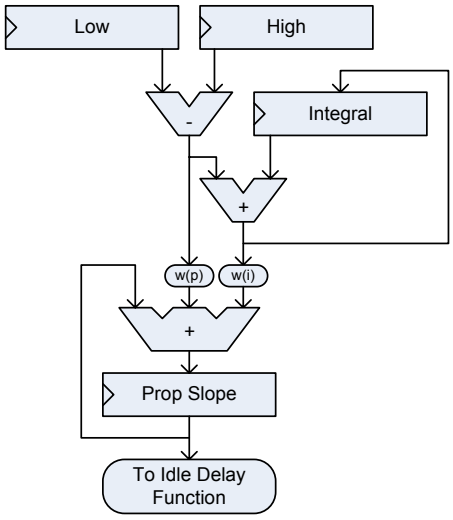
1) Max Delay Control: We found that delays greater than some threshold were counter-productive in exploiting the full dynamic range of the DRAM postponed refresh capability. Through manual exploration of a range of delays, we found the average delay of all idle periods was an effective value across a range of workloads. As such, we devised a circuit to estimate the average delay value. This is accomplished without the logic complexity of a true integer divide circuit. The circuit maintains a 20-bit accumulator and a 10-bit counter. As every idle interval ends, the counter increments by one, while the number of idle cycles in the interval are added to the accumulator. The average is calculated every $2^{10} = 1024$ idle intervals with a simple shift-left of 10 bits. If the accumulator overflows, a maximum average value of 1024 is used.

2) Proportional Slope Control: The goal of the proportional region is to dynamically center the distribution of `refresh` operations in the postponed spectrum. This is accomplished by tracking the relative frequency of `refresh` operations across a postponed pivot point. This postponed point is the target average `refresh` execution point. We used a postponed count threshold of four in our system, reflecting the midpoint of the deferral range.

The hardware structure to implement this function is shown in Figure 6. The structure maintains two counters containing the frequency of operations that fall on the low and high sides of the pivot threshold. When either of the counters overflow, all related counters (the *Low* and *High* counters of Figure 6(a), in this case) are divided in half by right-shifting each register by one. The scheme operates over profiling intervals, which are followed by adjustments at the end of each interval. At each adjustment interval, the logic subtracts the values of the



(a) Low and High counter update logic



(b) Proportional Slope control circuit

Fig. 6. Proportional Slope Control Circuit

High and Low counters. The value is applied to a Proportional Integral (PI) (shown in Figure 6(b)) control circuit to update the Proportional Slope parameter for the subsequent interval. Not shown in Figure 6 is the reset of the High and Low counters after each adjustment interval.

For our analysis, we use the following parameters which were determined to be effective through simulation analysis. The High, Low, and Integral counters are 16 bits in width. A relatively short adjustment interval of 128k memory clocks is used, since the profiling structure has a fairly small amount of state and stabilizes quickly. The Proportional Slope value is a 7-bit register, which represents the slope of the proportional region (units of decrease in delay cycles per postponed step). The $w(p)$ and $w(i)$ weighting functions of the PI controller use simple power-of-two division accomplished by truncating the value to largest 5-bit value (shifting off up to 11 leading zeros).

C. Elastic Refresh Queue Overhead

Table IV shows a summary of all the components of the Elastic Refresh scheduler. The overhead of the Elastic Refresh Queue can be divided into the basic static control mechanism

TABLE IV
REFRESH SCHEDULING MECHANISMS

Name	Description	Dynamic Control
Fixed Delay	Sets the maximum delay value of the Idle Delay Function	Detection of average delay of workload
Proportional Delay	Idle Delay which scales based on number of deferred `refresh` operations	Adjust with PI based control of Figure 6 to exploit full deferral capability

(FD) and the additional hardware to dynamically tune the parameters (DD). For the FD system, each memory rank requires a 10 bit idle counter. In addition, the max delay, proportional slope, and high priority pivot parameters require 10, 7, and 3 bit registers. In total this overhead is negligible (an 8 rank memory control would gain 100 register bits). The hardware to dynamically adjust the *Max Delay* parameter requires the addition of a 20 bit wide, 10 bit input accumulator and a 10 bit counters. The *Proportional Slope* logic consist of two 16 bit *High/Low* counters, a 16 bit *Integral* accumulator, and a 7 bit two input accumulator for the *Proportional Slope* term generation. All of these components are negligible compared to the size of a typical memory controller which would contain this logic.

V. EVALUATION

A. Simulation Methodology

To evaluate the proposed Elastic Refresh policies, we utilized the Gems toolset [11], built on top of the Simics [15] functional simulator. Gems provides a cycle-accurate out-of-order processor model along with a detailed memory subsystem. Gems was configured to simulate from 1 to 8 aggressive out-of-order cores. The memory subsystem model uses a directory-based MOESI cache coherence protocol and a detailed memory controller. The Gems default memory controller was augmented to simulate a *First-Ready, First-Come-First-Served* (FR_FCFS) [16] memory controller that supports two separate baseline refresh policies: a) *Demand Refresh* (DR) and b) *Defer Until Empty* (DUE) (see Section III-D) along with the proposed Elastic Refresh policies. Table V includes the basic system parameters.

For the memory refresh parameters, we evaluated a configuration representing what `tRFC` could be in the 16Gbit DRAM time-frame. The exact value of `tRFC` is difficult to narrow down due to the irregularities between DDR3 values for 4 GBit and 8 GBit devices (described in Section II-B). Based on this, we chose a value of 550ns for `tRFC`. For `tREFI`, we selected the 95°C interval of 3.9μs, as this reflects usage in dense server environments, where CMP systems and large memory configurations are common [5], [6].

The SPEC CPU2006 benchmark suite [17] was compiled to the SPARC ISA with full optimizations (peak flags). To estimate representative average behavior, for each experiment eight segments of 100M instructions were simulated, selected evenly along the whole execution of the benchmark. To do so, each benchmark was fast-forwarded to the beginning of each segment; the next 100M instructions were used to warm up the last-level cache and memory controller structures; and finally the following 100M instructions were used to evaluate the Elastic Refresh policies. The performance of each experiment is estimated based on the average behavior along the eight 100M instructions segments. In simulations involving multiple cores, each processor's instruction count can drift, though this effect is extremely small amounts in the homogeneous SPEC Rate benchmarks. In any case, we measured the total IPC across all cores in the interval in which core 0 executed 100M instructions.

TABLE V
CORE AND MEMORY-SUBSYSTEM PARAMETERS USED FOR CYCLE-ACCURATE SIMULATIONS

CPU	**Frequency**	**Pipeline**	**Branch Predictor**
	4 GHz	30 stages / 4-wide fetch / decode	Direct YAGS / indirect 256 entries
Memory	**L1 Data & Inst. Cache**	**L2 Cache**	**Memory Bandwidth**
	64 KB, 2-way associative, 3 cycles access time, 64 Bytes block size, LRU	8 MB, 8 ways associative, 12 cycles bank access, 64 Bytes block size, LRU	21.33 GB/s
	DRAM	**Controller Organization**	**Controller Queue Sizes**
	8GB DDR3-1333 8-8-8	2 Memory Controllers 2 Ranks per Controller 8 DRAM chips per Rank	32 Read Queue & 32 Write Queue Entries

B. Performance of Refresh Mitigation Policies

The net performance benefit of the Elastic Refresh scheme are analyzed in this section. All results are relative to the best known algorithm DUE. Single core SPEC Speed [17] results are shown in Figure 7; four core SPEC Rate [17] results in Figure 8; and eight core SPEC Rate results in Figure 9. In general, we observe the most significant throughput gains on workloads that exhibit high levels of memory traffic. Interestingly, these workloads include the classic high memory bandwidth workloads `libquantum` and `bwaves`, but also include more moderate bandwidth workloads that exhibit low MLP, such as `omnetpp` and `xalancbmk`. This reflects the refresh problem is more tied to latency penalties rather than simply bandwidth overhead.

1) Fixed Delay Results: For the Fixed Delay runs we selected static values for each of the parameters that seemed to be effective for most workloads (an exhaustive search would be prohibitive, considering the number of simulation cycles required). These values were a Constant region value of 400 memory clocks and a Proportional Slope value of 40 memory clocks per deferral. On average, we observed performance

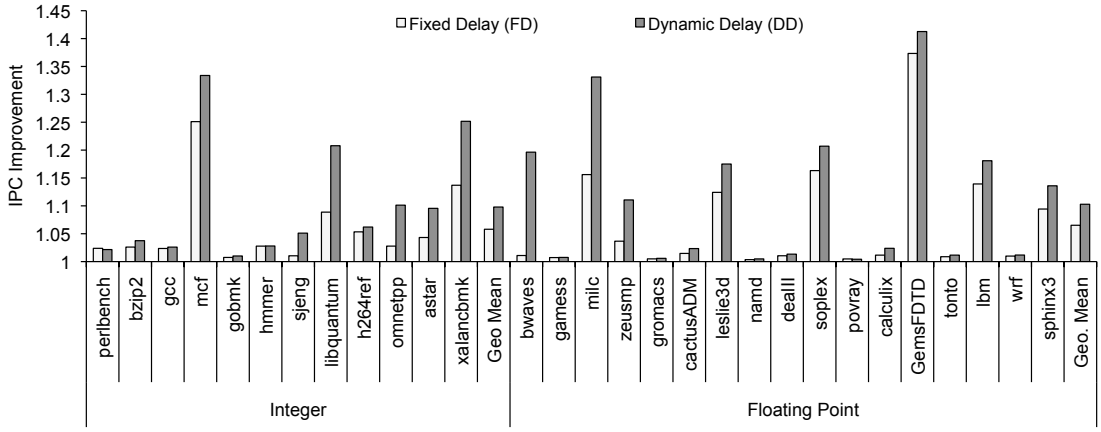


Fig. 7. IPC improvement of proposed refresh policy techniques over baseline refresh policy on 1 core

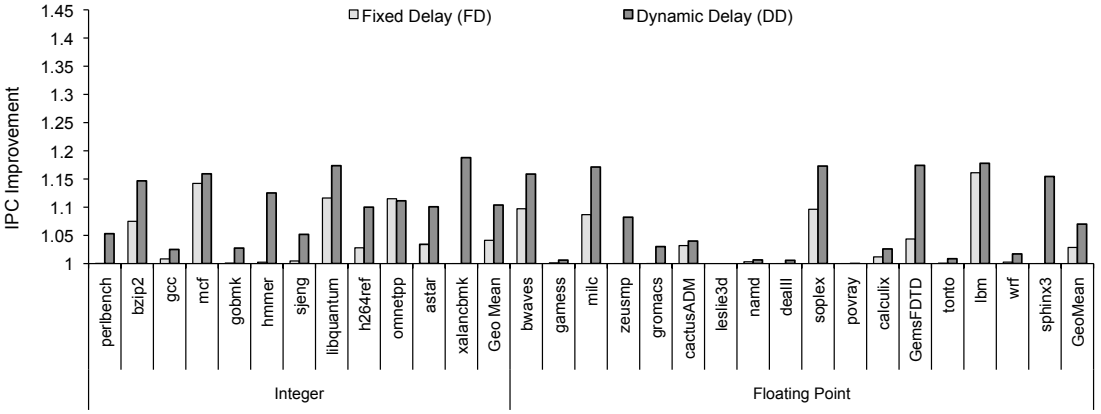


Fig. 8. Relative IPC improvement of proposed refresh policy techniques over baseline refresh policy on 4 cores

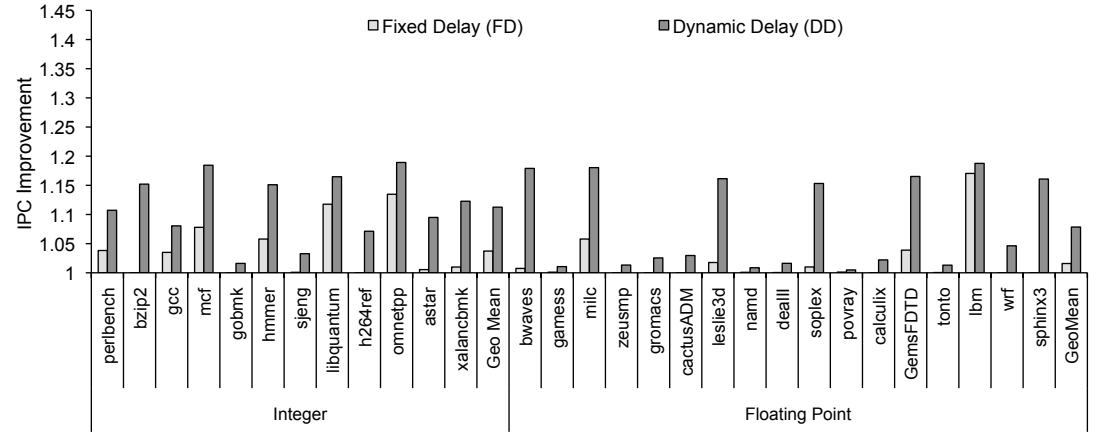


Fig. 9. Relative IPC improvement of proposed refresh policy techniques over baseline refresh policy on 8 cores

improvements of Integer (5.9%, 4.1%, 3.7%) and Floating-Point (6.5%, 2.8%, 1.6%) across one, four, and eight CPUs. These improvements are quite significant given the very simple mechanism and extremely low logic required. That said, as the delay intervals present in high bandwidth workloads (more pervasive as the core count is increased) are inherently shorter, a a static setting simply cannot work across all cases. Note the most effective static settings favored lower bandwidth workloads as the improvements were larger in these cases. This biased the selection of the static parameters for the single core runs.

2) Dynamic Delay Results: The Dynamic Delay results show greater gains across the different workloads and system sizes with improvements of Integer (9.8%, 10.3%, 11.2%) and Floating-Point (10.2%, 7.0%, 7.9%) across one, four, and eight CPUs simulations. As expected, the improvements for high bandwidth single core workloads such as `libquatum`, `bwaves`, and `milc` are significant with Dynamic Delay. The improvement using dynamic parameters is very significant in the 8 core simulations, increasing the meager 3% fixed delay to a 9% gain. These results are particularly impressive considering the trivial logic area overhead of the mechanisms.

VI. Related Work

Avoiding Refresh: One option to help reduce refresh penalties is to avoid sending some fraction of the operations that are determined to be unneeded. In the Smart Refresh work, the authors propose taking advantage of the inherent refresh that occurs through existing `read` and `write` operations when ranks are precharged [18]. In ESKIMO, methods are proposed to utilize semantic knowledge, such as "deleted" dynamically allocated memory, to avoid refreshing memory regions which the program is no longer using [19]. While both of these refresh avoidance techniques are potentially quite useful, they are incompatible with existing commodity DRAM devices. In addition, the significant design changes required would be difficult and timely to negotiate through JEDEC committees, and proprietary DRAM designs, such as Rambus DRAM, have been challenging to bring to market.

Hiding Refresh: It is straight-forward to envision a DRAM architected such that `read` and `write` commands may be completed in other sections of the memory at the same time as `refresh` is taking place elsewhere in the bit-arrays or banks. Indeed, such *concurrent refresh* schemes have been implemented outside the commodity server DRAM space [20]. However, for commodity DRAMs, this approach has not been taken, due to the high current draw of a `refresh` operation, and the added design and system expense that might be required to support multiple simultaneous operations, from a power supply/noise perspective.

As irregular memory latency can be detrimental to real time systems, memory refresh prevents dynamic memory adoption in many embedded application spaces. In "Making Refresh Predictable" [21], the program itself can specify when `refresh` operations can be sent, thus avoiding penalties. Extending this idea to the more general server computation space may be possible, but the irregularity and complexity of multi-programmed system operation increases the difficulties of deploying this solution compared to more explicitly controlled real time systems.

Memory Request Prediction: The concept of predicting future memory references has been proposed as a method to decide when to enter latency-penalizing lower power DRAM states [22]. The fundamental difference between the prediction for low power states as compared to refresh is centered in the functional requirement of refresh (to prevent loss of memory data). As such, the urgency aspect of refresh, which drove the dynamic nature of the prediction in this work is very different from this prior work. Another important difference between refresh and powerdown scheduling policies is highlighted by Fan et al. in [23], which demonstrates lower DRAM power if idle-time predictors are ignored, and memory is put in low power states as soon as possible. With powerdown, it is beneficial to drive to the lower power state as often as possible, whereas refresh must be driven at a specific rate. Fan's observations about powerdown essentially reflect the traditional Demand Refresh scheduling policy, which we found to be quite poor.

VII. Conclusions

This work has shown that Elastic Refresh mechanisms are effective in mitigating much of the increasing penalty of DRAM refresh, providing a $\sim$10% average performance improvement across the SPEC CPU suite on one, four, and eight core simulations. These gains were achieved using very low overhead mechanisms, that are easily incorporated into existing memory schedulers, and are effective on commodity JEDEC DDRx SDRAM memory devices.

The relatively large gains compared to the very small logic overhead highlight the importance of the memory interface in multi-core designs, and particularly "background" operations such as memory refresh. As memory technologies become more complex, operations beyond typical reads and writes will become more important. These future memories include both future DDRx memories (and more complex 3D packagings), but also non-DRAM memories such as PCM, RRAM, and STT-RAM.

Acknowledgements

The authors would like to thank Steve Dodson, Warren Maule, Kyu-Hyoun Kim, and the anonymous reviewers for their suggestions that helped improve the quality of this paper. The authors acknowledge the use of the Archer infrastructure for their simulations. This work is sponsored in part by the National Science Foundation under award 0702694 and CRI collaborative awards: 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884, 0751091, and by IBM. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM.

REFERENCES

[1] D. Graham-Smith, "IDF: DDR3 won't catch up with DDR2 during 2009," in *PC Pro*, Aug. 2008.

[2] K. Kilbuck, "Main memory technology direction," in *Microsoft WinHEC*, 2007.

[3] Samsung Corp., "Samsung develops world's highest density DRAM chip (low-power 4gb DDR3)," January 2009, press Release.

[4] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[5] B. L. Jacob, "DRAM Refresh is Becoming Expensive in Both Power and Time," in *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[6] Influent Corp., "Reducing server power consumption by 20% with pulsed air cooling," Jun. 2009, http://www.influentmotion.com/Server White Paper.pdf.

[7] L. Minas and B. Ellison, "The problem of power consumption in servers," in *Intel Press Report*, 2009.

[8] Micron, "TN-47-16 Designing for High-Density DDR2 Memory Introduction," 2005.

[9] JEDEC Committee JC-42.3, "JESD79-3D," Sep. 2009.

[10] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer, "CMP Memory Modeling: How much does accuracy matter?" in *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation*, June 2009.

[11] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: A memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, 2005.

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet: A general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, 2005.

[13] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A reinforcement learning approach," *Computer Architecture, International Symposium on*, vol. 0, pp. 39–50, 2008.

[14] Texas Instruments, "TMS320DM647/DM648 DSP DDR2 Memory Controller User's Guide, Literature Number: SPRUEK5A," 2007.

[15] S. Kostylev and T. Lowrey, "Drift of programmed resistance in electrical phase change memories," in *Proceedings of the European Phase Change and Ovonics Symposium*, 2008.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 128–138.

[18] M. Ghosh and H. S. Lee, "Smart Refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 134–145.

[19] C. Isen and L. John, "ESKIMO: Energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 337–346.

[20] K. Toshiaki, P. Paul, H. David, K. Hoki, J. Golz, F. Gregory, R. Raj, G. John, R. Norman, C. Alberto, W. Matt, and I. Subramanian, "An 800 MHz embedded DRAM with a concurrent refresh mode," in *IEEE ISSCC Digest of Technical Papers*, Feb. 2004, pp. 206–207.

[21] B. Bhat and F. Mueller, "Making DRAM refresh predictable," *Real-Time Systems, Euromicro Conference on*, vol. 0, pp. 145–154, 2010.

[22] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "DRAM Energy Management Using Software and Hardware Directed Power Mode Control," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, p. 159.

[23] X. Fan, C. Ellis, and A. Lebeck, "Memory controller policies for dram power management," in *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001, pp. 129–134.