The Dissertation Committee for Jeffrey Adam Stuecheli
certifies that this is the approved version of the following dissertation:

Mitigating DRAM Complexities Through Coordinated Scheduling Policies

Committee:

Lizy John, Supervisor

Earl Swartzlander

Tony Ambler

Mattan Erez

Lixin Zhang

Mitigating DRAM Complexities Through Coordinated Scheduling Policies

by

Jeffrey Adam Stuecheli, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

Dedicated to my wife Erica and daughter Cadence.

Acknowledgments

I wish to thank the multitudes of people who helped me in the lengthy and difficult process of completing a Ph.D. while maintaining my full time position at IBM. That said, working on the industry leading IBM POWER architecture team over this time provided me with many real problems to solve, very much an advantage. This advantage was reciprocated into my work, in that the combined research community provided enumerable solutions.

I also thank my advisor Lizy John and our research group LCA. Lizy tolerated the low research productivity periods that correspond with high demands at IBM. She also motivated me to continue, when the end seemed hard to reach. LCA also provided an excellent ecosystem in order to conduct my work. Specifically, LCA member Dimitris Kaseridis was invaluable in both enabling the simulation infrastructure knowhow, combined with his expertise in the field.

Members of the IBM team were also a key component in completing this work. The specific members of my development organization contributed to the work include, the full list is too long to name, William Starke, Steve Dodson, Warren Maule, and Balaram Sinharoy. IBM research also proved a tremendous resource. Hillery Hunter's command of memory design and keen insights were exceptionally helpful. In addition David Daly, KH Kim, Moin Qureshi, Vijayalakshmi Srinivasan. Ram Rajamoni, and John Carter helped at

various stages.

Most of all I would like to thank my family and friends, who most certainly missed my company as I was spending all my free time down at the ACES building.

Mitigating DRAM Complexities Through Coordinated Scheduling Policies

Publication No. _____

Jeffrey Adam Stuecheli, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Lizy John

Contemporary DRAM systems have maintained impressive scaling by managing a careful balance between performance, power, and storage density. In achieving these goals, a significant sacrifice has been made in DRAM's operational complexity. To realize good performance, systems must properly manage the significant number of structural and timing restrictions of the DRAM devices. DRAM's efficient use is further complicated in many-core systems where the memory interface has to be shared among multiple cores/threads competing for memory bandwidth.

In computer architecture, caches have primarily been viewed as a means to hide memory latency from the CPU. Cache policies have focused on anticipating the CPU's data needs, and are mostly oblivious to the main memory. This work demonstrates that the era of many-core architectures has created new main memory bottlenecks, and mandates a new approach: coordination of cache policy with main memory characteristics.

Using the cache for memory optimization purposes dramatically expands the memory controller's visibility of processor behavior, at low implementation overhead. Through memory-centric modification of existing policies, such as scheduled writebacks, this work demonstrates that performance-limiting effects of highly-threaded architectures combined with complex DRAM operation can be overcome. This work shows that an awareness of the physical main memory layout and by focusing on writes, both read and write average latency can be shortened, memory power reduced, and overall system performance improved.

The use of the "Page-Mode" feature of DRAM devices can mitigate many DRAM constraints. Current open-page policies attempt to garner the highest level of page hits. In an effort to achieve this, such greedy schemes map sequential address sequences to a single DRAM resource. This non-uniform resource usage pattern introduces high levels of conflict when multiple workloads in a many-core system map to the same set of resources.

This work presents a scheme that provides a careful balance between the benefits (increased performance and decreased power), and the detractors (unfairness) of page-mode accesses. In the proposed Minimalist approach, the system targets "just enough" page-mode accesses to garner page-mode benefits, avoiding system unfairness. This is accomplished with the use of a fair memory hashing scheme to control the maximum number of page mode hits.

High density memory is becoming ever 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 work 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 industry standard DRAM memory specifications. The proposed mechanisms are shown to mitigate much of the penalties seen with dense DRAM devices.

In summary this work presents a significant improvement in the ability to exploit the capabilities of high density, high frequency, DRAM devices in a many-core environment. This is accomplished though coordination of previously disparate system components, exploiting integration of such components into highly integrated system designs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is now well-understood that in the nanometer era, technology scaling will continue to provide transistor density improvements, but that power density and performance improvements will slow. In response, processor designers now target chip-level throughput (instead of raw single-core performance), packing increasing numbers of cores and threads onto a chip. In 2000, virtually all server processors were single-core, single-thread; today systems contain 16 threads in the Intel Nehalem EX [10], 32 threads in the IBM POWER7 [23], and 128 threads in the Sun Rainbow Falls [10].

The processor-memory interface has been particularly challenged by this many-core trend. Technology scaling provides roughly 2x the number of transistors per lithography generation, so when core or thread counts more than double per generation, the result is generally a decrease in the available cache size per core and/or thread. This is very evident in the Rainbow Falls design, where 128 threads share a 6 MB cache. From first principles, a drop in on-chip cache size will result in higher miss rates and higher memory bandwidth demands. Single socket memory demands have thus been rapidly increasing, not only due to core and thread counts, but also from the transition to throughput-type designs, which provide fewer cache

bits per thread.

These many-core architectures struggle not only to provide sufficient main memory bandwidth per core/thread, but also to schedule high bus utilization compared to single threaded designs. Server processors generally have one or two main memory controllers per chip, meaning that many cores share a single controller and a memory controller simultaneously sees requests from different work streams. In this context, locality is easily lost, and it becomes difficult to find and schedule spatially sequential accesses. Inefficient scheduling results in performance reductions and consumes unnecessary energy.

Most servers currently use JEDEC (Joint Electron Device Engineering Council) standardized Double-Data-Rate (DDR) memory [22], so a fairly accurate understanding of memory bandwidth scaling can be obtained by looking at DDR trends. In terms of raw IO (Input/Output) speeds, DDR has continued to improve, with peak speeds doubling each generation (400Mbps DDR, 800Mbps DDR2, 1600Mbps DDR3). IO frequencies continue to scale, but other key parameters, such as reading a memory cell or turning a bus around from a write to a read operation, are not scaling at comparable rates. At higher signaling rates, the electrical integrity of bus interconnects becomes much more difficult to maintain – both within the DRAM chips and across the main memory path to/from the processor. This results in a complex set of timing parameters which dictate that gaps be inserted when the access stream transitions from a write to a read or vice-versa, significantly degrading effective memory bandwidth. This problem has worsened with each memory generation. For example, tWRT, the Write-to-Read Turnaround

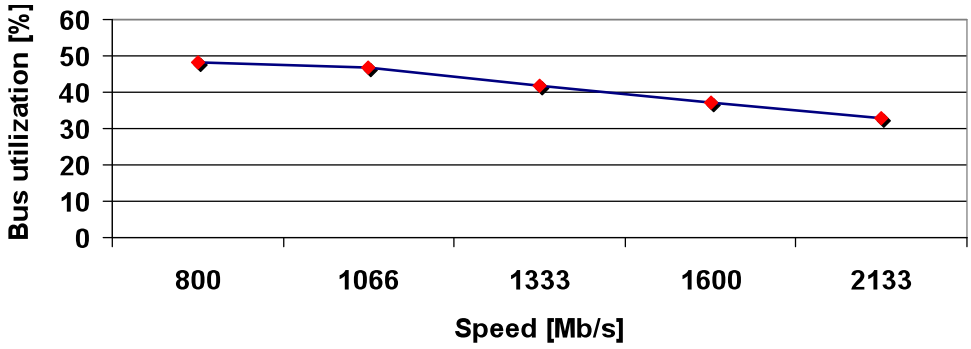


Figure 1.1: DDR3 single rank bus utilization efficiency, limited by DRAM parameters (tRC, tRRD, tFAW), and bus turnaround time (tWRT) [22]

delay, has stayed within a range of 5-10ns for DDR through DDR3. Therefore, as DRAM IO frequencies increase, the number of cycles wasted between each access grows. Figure 1.1, calculated from JEDEC parameters, shows the dramatic impact of these mandatory timing delays on effective bandwidth: even with perfect scheduling, utilization of a single memory rank can be as low as 25-40% for high-frequency DDR. Clearly, this trend cannot continue – in the many-core era, computer architects must find ways to improve not only raw memory bandwidth, but also memory bandwidth efficiency.

While DRAM devices output only 16-64 bits per request (depending on the DRAM type and burst settings), internally, the devices operate on much larger, 1KB pages (also referred to as *rows*). Since the read latency and power overhead of the DRAM cell array access have already been paid, accessing multiple columns of that page decreases both the latency and power of subsequent accesses. These successive accesses are said to be performed in *page mode* and the memory

requests that are serviced by an already opened page loaded in the row buffer are characterized as *page hits*.

Due to the reductions in both latency and energy consumption possible with page mode, techniques to aggressively target page mode operations are often used. There are downsides however, which must be addressed. Leaving a specific page open produces a higher access latency to other rows in the same bank (*page conflict*). In addition, certain scheduling algorithms such as the *First-Ready, First-Come-First-Served* (FR-FCFS) [51] give higher priority to page hit operations, which can result in unfairness for non page hit operations. Therefore, although row buffer hits are useful, they must be used in moderation.

In addition to limits on bandwidth scaling, the act of refreshing the DRAM cells has become more invasive. Each DRAM cell requires a refresh every 64ms. DDR DRAM chips implement internal refresh control logic, such that the memory controller must send refresh commands at a specified rate such that all rows in the DRAM are refreshed in this 64ms time. Traditionally this rate was determined by dividing the 64ms by the number of rows in the dram. As DRAM density double with each generation, the number of rows also doubles. As such, using this traditional method, the rate at which refresh command must be sent would need to double with each generation. In order to reduce the magnitude of the refresh penalty, DRAM vendors have designed refresh commands such that multiple rows are refreshed in one command. While this does reduce the refresh penalty, there is a cost in that the time required to execute a refresh increases with each generation. This is shown in Table 5.1. The parameter tRFC defines the time for refresh

Table 1.1: Refresh parameters as density increases [22]

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

commands to complete while tREFI defines the interval between refresh commands. Data is included for both 85C and 95C operational points. Note that 95C is common operating point in dense server environments such as the systems that this work targets.

1.1 Objective

These complexities in achieving high performance in memory systems require advancements in scheduling policies. This dissertation seeks to achieve these advancements through innovative coordination of previously isolated system components combined with more sophisticated policies in previously ignored aspects of memory scheduler design.

Traditional memory controller designs are implemented in relative isolation with respect to the devices they service. Requests to read and write memory are scheduled for execution in the DRAM through internal request queues and priority

logic. Internally generated requests such as refresh are also generated and executed. This work explores the improved system execution speed and energy usage of memory controller policies that coordinates the processing of the read, write, and refresh commands both within and beyond the memory controller functional unit.

Design philosophy of the Coordinated controller: Develop mechanisms to increase both direct and perceived control over the operation set presented to the memory controller scheduler, enabling more efficient scheduling.

1.2 Overall Mechanism Description

The following sections describe the various micro-architectural components and mechanisms that comprise the overall Coordinated Memory Scheduler. The overall system is shown in Figure 1.2. In this system, several system components communicate with the memory scheduler to comprise the Coordinated design. These components are,

1. **Minimalist open-page policy**: The management of the DRAM row buffer is an important aspect of the memory system. Row buffer hit induced starvation has been shown to be a significant problem in multi core systems [45]. This work utilizes a minimalist open-page policy, where "just enough" page mode hits are produced, while avoiding row buffer hit induced starvation.

2. **Virtual Write Queue [58]**: Enables direct control of cache write traffic based off memory traffic.

3. **Elastic Refresh Queue [57]**: A structure to stage and control execution of

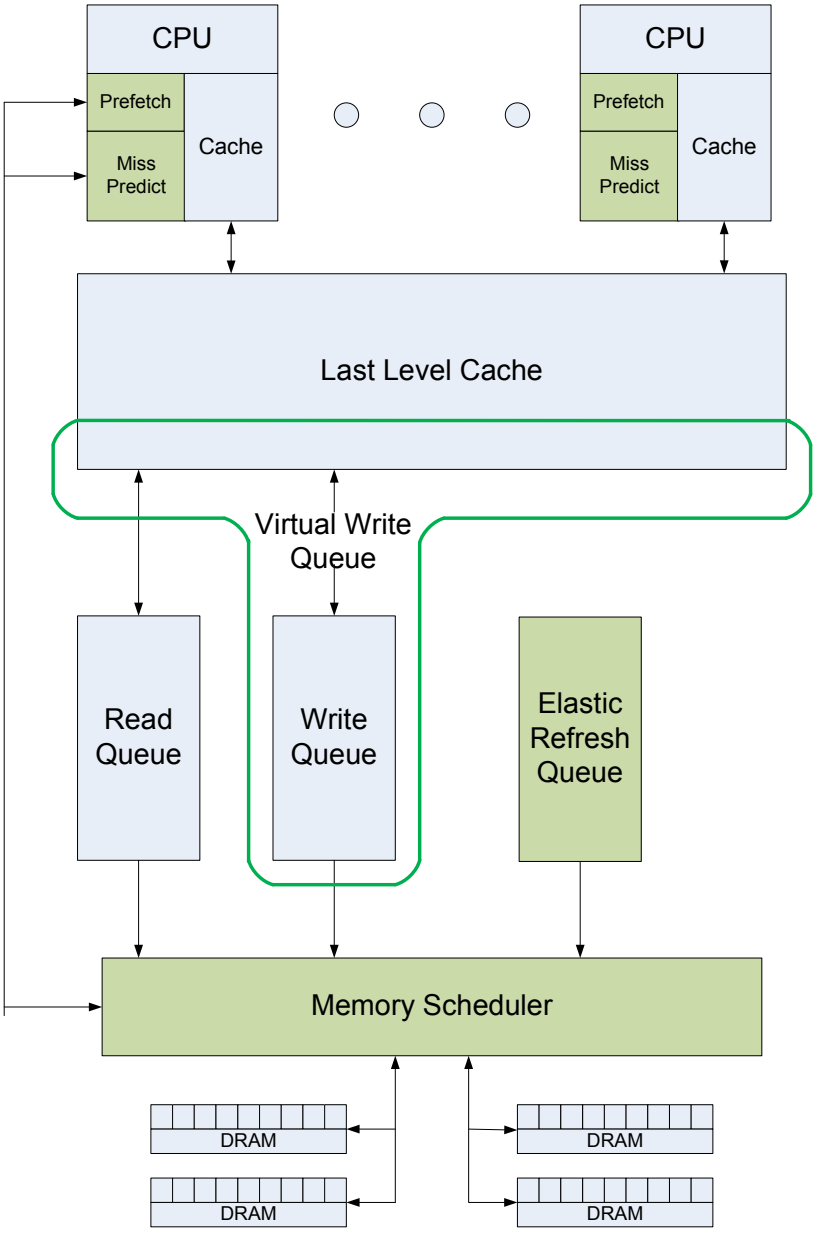


Figure 1.2: Overall Coordinated System

refresh requests, and ensure DRAM parameters are not violated.

1.3 Thesis Statement

The complexity of high frequency and density DRAM combined with the need to support many execution threads of modern multi-threaded chip multi-processors can be mitigated by coordinating the CPU, cache, and memory controller policies around these constraints.

1.4 Contributions

To achieve this goal, this dissertation makes several contributions. These are listed below.

1. Minimalist open-page: Identify the memory controller should target a smaller number of page hits through the memory hash function. This prevents row-buffer induced starvation.

2. Virtual Write Queue: Propose a coordinated system between the last level cache and the memory write scheduler, increasing the efficiency of write scheduling, increasing overall bandwidth and reducing read latency increases due to write conflicts.

3. Elastic Refresh: Identify refresh induced latency penalties in high-density DRAM, and propose predictive mechanisms that significantly mitigate the penalty.

1.5 Organization

The structure of this work is as follows. Chapter 2 contains background of DRAM systems. Chapter 3 describes the experimental methodology utilized in this work. This is followed by a survey of related work in Chapter 4. Chapter 5 contain a detailed analysis of the most important emerging DRAM complicities and system behaviors. This drives to motivation for coordinated scheduling policies. Chapter 6 covers DRAM ”page mode” in light of these system constraints, proposing a Minimalist open-page policy. Chapter 7 is a detailed description of the Virtual Write Queue, which coordinates the last level cache and memory write scheduler. Completing the design components, the Elastic Refresh mechanism is covered in Chapter 8, which mitigates refresh penalties in high-density memory. Following each design description, analysis sections covers both high level and detailed simulations of the proposed policies.

Chapter 2

Main Memory Background and Terminology

The system structure assumed for this work is shown in Figure 2.1. To maximize memory bandwidth and memory capacity, server processors have multiple *memory channels* per chip. Each channel is connected to one or more *DIMMs* (memory cards), each containing numerous *DRAM chips*. These DRAM chips are arranged logically into one or more *ranks*. Within a rank, each DRAM chip provides just 4-8 bits of data per data cycle, and a rank of 8-16 DRAM chips works in unison to produce eight bytes per data cycle. The DRAM *burst-length* (BL) specifies an automated number of data beats that are sent out in response to a single command, commonly 8 data beats, to provide 64 Bytes of data. From the time of applying an address to the DRAM chips, it takes about 24ns (96 processor clocks at 4GHz) for the first cycle of data, but subsequent data appear at high frequency, closer to 2-3 processor clocks.

Internally, DRAM chips are partitioned into *banks* that can be accessed independently, and banks are partitioned into *pages*. DRAM *page mode* provides the opportunity to read or write multiple locations within the same DRAM page more efficiently than accessing a new page. Page mode accesses require that the memory controller find requests with adjacent memory addresses, but are executed

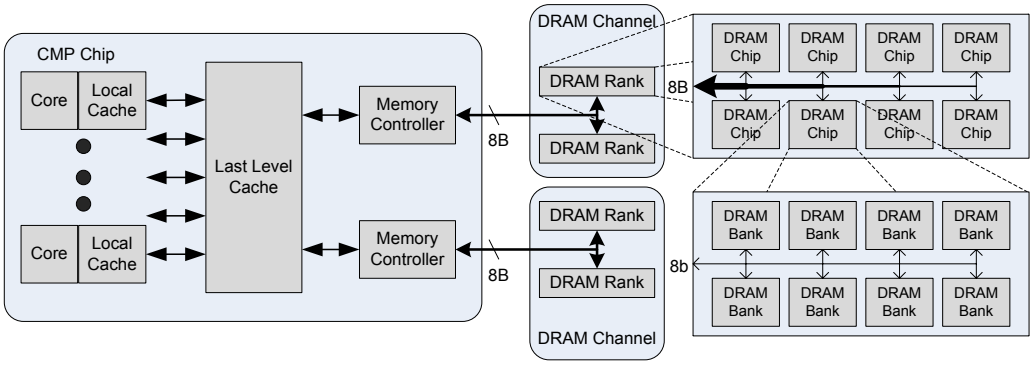


Figure 2.1: Baseline CMP and memory system

with fewer timing constraints, and incur lower per-access power than non-page mode accesses.

DRAM chips are optimized for cost, meaning that technology, cell, array, and periphery decisions are made with a high priority on bit-density. This results in devices and circuits which are slower than standard logic, and chips that are more sensitive to noise and voltage drops. A complex set of timing constraints has been developed to mitigate each of these factors for standardized DRAMs, such as outlined in the JEDEC DDR3 standard [22]. These timing constraints result in dead times before and after each random access; the processor memory controller's job is to hide these performance-limiting gaps through exploitation of parallelism.

Chapter 3

Experimental Methodology

Evaluation of computer systems present a complex set of constraints. Moore's law is a double edged sword. Each new generation of machines enables greater compute capability for simulation based analysis methods. However, each future system becomes more complex, forcing system evaluation to be inherently extrapolation based. While rapid prototyping based on configurable arrays is growing in popularity [5], the flexibility and speed of general purpose system based simulation of designs is still the leading evaluation system. While simulation methods are commonly used, care must be taken to address the difficulties in simulating enough detail, yet managing simulation runtime overhead. Beyond model representativeness, the workload simulated must be properly reduced from the full complete program execution.

This dissertation uses a combination of a full-system timing/functional simulator SIMICS [38], along with the detailed processor, interconnect, and memory system microarchitecture simulator GEMS [39], to evaluate the proposed memory subsystem enhancements. Evaluation of the proposed structures is primarily based on cycle simulation of various Spec CPU 2006 benchmark configurations [12]. This was augmented with analysis of complex commercial

workloads using bus traces gathered from machines running complex workloads, of the scale which is unfeasible in simulation. In addition smaller less detailed simulations were used when the number of experiments would be prohibitive and unnecessary.

A summary of the primary evaluation tool is provided in the following subsections. This is followed by descriptions of the evaluated benchmarks, and the evaluation metrics utilized.

3.1 GEMS on Simics Simulator

This work utilizes *Simics* from Virtutech [38] as a fully system timing simulator. Simics is a very powerful full system simulator that is able to boot and run unmodified operating systems and their applications on the target simulated machine. Simics is a system-level instruction set (ISA) simulator that is able to accurately simulate the functionality of each ISA instruction in the system. Simics is an architectural simulator only, detailed micro-architecture timings of the instruction execution must be modeled outside Simics. The configuration used in this work (Simics 3.0) simulates an existing SPARC-v9 SMP processor, the UltraSPARC III+, along with the necessary memory modules, motherboard chipset, network card and hard disk to allow booting and running a full, unmodified version of Solaris 10 operating system using an SMP kernel. An overview of the Simic's architecture is illustrated in Figure 3.1. In summary, Simics provides the necessary infrastructure to simulate a full, real machine at the functional level while the detailed cycle-accurate, timing simulation of a realistic processor and memory

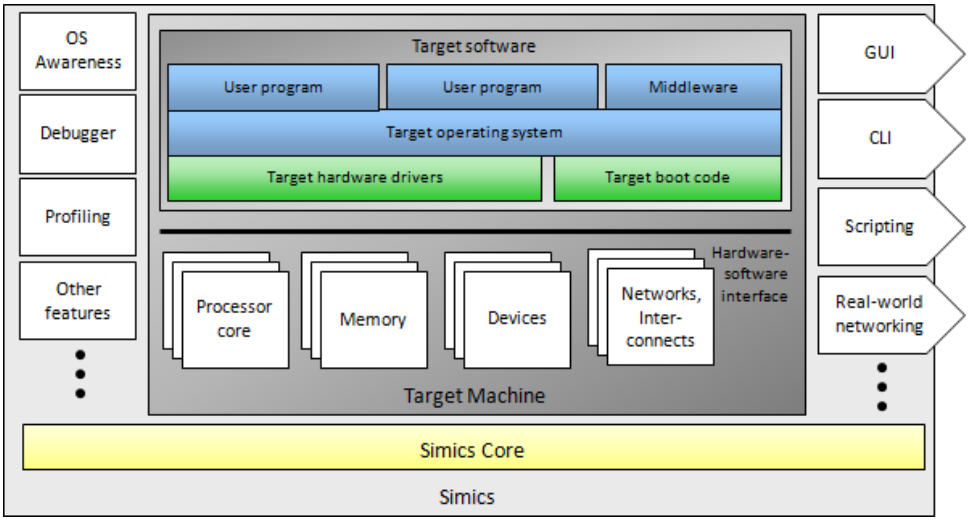


Figure 3.1: Simics functional simulator architecture [38]

hierarchy is left to GEMS as an external module added to Simics API hooks.

3.2 Detailed Microarchitecture Simulator

As a detailed microarchitecture simulator this dissertation uses the multi-facets general execution-driven multiprocessor simulator (GEMS) toolset from the University of Wisconsin [39]. As described in the previous section, GEMS in combined with Simics to provide a detailed, cycle-accurate simulator by decoupling simulation functionality and timing. Simics provides a robust environment to boot an unmodified OS along with the functional simulator. GEMS timing modules interact with Simics to determine when Simics should execute an instruction. However, what the result of the execution of the instruction is ultimately dependent on Simics. Therefore, the two tools operate in a lock-step mode. Even though,

GEMS decouples functional simulation and timing simulation, the functional simulator is still affected by the timing simulator, allowing the system to capture timing-dependent effects.

The basic architecture of GEMS is illustrated in Figure 3.2. Its basic functionality is divided between two basic components: *Ruby* and *Opal*. Ruby is the most important component and is the basic timing simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect, memory controllers, and banks of main memory. Ruby combines hard-coded timing simulation for components that are largely independent of the cache coherence protocol (e.g., the interconnection network) with the ability to specify the protocol-dependent components (e.g., cache controllers, coherence protocol) in a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence). When Ruby is used stand-alone it simulates a simplified in-order processor for every core in the system. To simulate more advanced cores, the additional *Opal* module has to be used along with *Ruby* in *Simics*. Opal models a SPARC ISA, out-of-order, superscalar, deeply-pipelined processor core. Opal is configured by default to use a two-level gshare branch predictor, MIPS R10000 style register renaming, dynamic instruction issue, multiple execution units, and a load/store queue to allow for out-of-order memory operations and memory bypassing. Because Opal runs ahead of the Simics functional processor, it models all wrong path effects of instructions that are not eventually retired. Opal implements an aggressive implementation of sequential consistency, allowing memory operations to occur out-of-order and detecting possible memory ordering

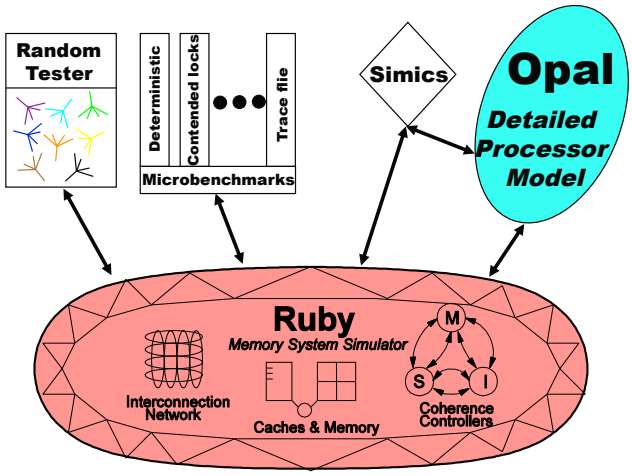


Figure 3.2: GEMS detailed, full-system, microarchitecture simulator overview [39]

violations as necessary.

Overall, the combination of Ruby, Opal and Simics allows a very detailed and accurate simulation of almost any CMP and memory-subsystem configuration. Some more advance features that are missing from a typical contemporary core, like cache-line prefetchers and banked DNUCA-like last-level caches were added on top of Ruby. More details about modifications and additions in the baseline simulation tool-set is describe in each individual chapter.

3.3 Bus Trace Simulation

Further evaluation was performed through a set of simulations using three diverse commercial workloads. Due to the complexity and size of the commercial workloads, a detailed evaluation using a cycle-accurate, full system simulator is prohibitively expensive. A large contributor to this complexity is the configuration

of the runtime environment. For example, full-size database systems contain massive disk arrays, large "workload drivers", large working sets, and high numbers of execution threads. As a solution, commercial workloads were evaluated using bus trace based cache simulations. These bus traces contain all primary cache misses generated as benchmarks run at full speed. This enabled evaluation of the detailed address relationships, and how these translated into specific memory resources.

These bus traces were collected on an IBM POWER5 [55] system using an IBM internal-use-only toolset. The tracing system operates by observing cache miss operations on the system bus interface. In order to decrease the filtering of requests observed, the cache is configured into direct mapped mode, with only one eight the full capacity. While the peak benchmark score is reduced with this smaller cache, the system is still well behaved (*i.e.*, no software timeouts), and provides realistic operation. Each operation is packed into full cache line records, that are written to a dedicated in-memory buffer. Each record contains the operation type, device issuing the request, coherence state, and timestamp information. As traffic can potentially overrun the tracing hardware, the system includes support to indicate any lost records, while maintaining proper timestamps.

The bus traces were evaluated using basic cache protocol simulation written in the C programming language. The simulator did contain basic timing information, that was utilized to estimate the behavior of the structures in the cache and memory system. This analysis was essentially a relatively low cost method to extend the evaluation into an application space not possible with detailed

simulations methods.

3.4 Benchmarks Suites

This work utilizes various combinations and configurations of the SPEC CPU2006 benchmark suite for detailed simulation, while full scale commercial workloads drive the bus traced analysis. This section contains descriptions of these workloads.

3.4.1 SPEC CPU2006

To evaluate the performance of a single core configuration and multiprogrammed workloads set, SPEC CPU2006 [12] was used. SPEC CPU2006 is the most popular set of benchmarks used in the evaluation of the CPU performance and memory of a modern server computer systems. The 29 benchmarks within the integer and floating point suites contain a broad mix of workloads, ranging from CPU intensive workloads to memory bandwidth bound applications. The workload is limited in the exclusion of system IO traffic, and true multi-threaded workloads. Workloads that stress these systems aspects are left for future work, as optimization of the system without these additional complexities still provides a very significant scope for this work.

3.4.2 Commercial Workloads

Traces were collected on three commercial workload. The first workload is an *On-Line Transaction Processing* (OLTP) workload driven by hundreds of

simulated individual clients generating remote transactions against a large database. The second workload represents a typical *Enterprise Resource Planning* (ERP) workload. As with the OLTP workload, the ERP workload was driven by simulated users who sent remote queries and updates to the database. Finally, the third workload is SPECjbb2005 benchmark [12] that targets the performance of a three-tier client/server system with emphasis on the middle tier.

Chapter 4

Related Work

This research relates to prior work in the field through a range of perspectives. Each of the following sections presents the most relevant work in each area.

Multi-threaded aware DRAM schedulers: Multi-threaded aware proposals have primarily addressed mechanisms to reorder requests received by the memory controller. Work on *Fair Queuing Memory Systems* by Nesbit, *et al.* [48] utilizes *fair queuing* network principles on the memory controller to provide fairness and throughput improvements. *Parallelism-Aware Batch Scheduling* [47] alleviates the problem by maintaining groups of requests from the various threads in the system and issuing them in batches to the controller. A suite of work has considered minimizing prefetch impacts on overall memory subsystem performance, some of which consider prioritizing prefetch according to page mode opportunity [35] [33].

System and DRAM interaction: Proposals to improve interactions between the DRAM controller and other system components have been proposed in the following areas. The *Eager Writeback* technique [32] addresses breaking the connection between cache fills and evictions, but the approach has minimal communication between the last-level cache and the memory controller and thus

misses performance and power opportunities which arise through knowledge of logical-to-physical address mapping and troublesome memory timing constraints. *SDRAM-aware* scheduling from Jang, *et al.* [21] addresses management of the on-chip network such that requests are ordered with regard to memory efficiency.

DRAM write-read turnaround: For specialized applications, DRAMs have been offered with separated read and write IOs, allowing for high bus utilization, even for mixed read/write access streams [1]. Borkenhagen, *et al.* [4], describe the problem of DRAM *write*-to-*read* turnaround delay, and recognize the need for cache/memory controller interaction to most effectively alleviate its performance effect. Borkenhagen, *et al.* propose a *read predict* signal, which provides the memory controller early notice that a read may soon arrive at the memory controller. If *read predict* is asserted, the memory controller will not issue pending writes, to avoid incurring a *write*-to-*read* turnaround delay which would delay the read.

Row Buffer Access Priority: Rixner, *et al.* [51] first described the *First-Ready First-Come-First-Serve* scheduling policy that prioritizes row-hit requests over other requests in the memory controller queue. This proposal utilizes a combination of a column centric DRAM mapping scheme combined with FR-FCFS policy. This approach was shown to create starvation and throughput deficiencies when applied to multi-threaded systems as described by Moscibroda *et al.* [45]. Prior work attempts to mitigate these problems through memory requests scheduling priority. Solutions that bias row buffer hits such as FR-FCFS [51] map to Scenario 1. Mutlu, *et al.* based their "Stall Time Fair Memory" (STFM)

scheduler [46] on the observation that giving priority to requests with opened pages can lead to significant introduction of unfairness in the system. As a solution their proposed a scheme that identifies threads that are stalled for a significant amount of time and prioritize them over requests to open-pages. On the average case, STFM will operate similarly to FR-FCFS mapping to Scenario 1. The "Micro-pages" proposal from Sudan, *et al.* [59] describe a scheme that uses smaller than typical OS page sizes in an effort to co-locate multiple frequently used pages in the same DRAM row buffer. Their solutions includes software and hardware migration mechanisms to move data in such small pages.

The *Adaptive per-Thread Least-Attained-Service* memory scheduler (ATLAS) [26] proposal, that tracks attained service over longer intervals of time, would follow Scenario 2, where the low bandwidth workload B would heavily penalize workload A. Following the same logic, the *Parallelism-aware Batcher Scheduler* (PA-BS) [47] ranks lower the applications with larger overall number of requests stored in every "batch" formed in the memory queue. Since streaming workloads inherently have on average a large number of requests in the memory queue, they are scheduled with lower priority and therefore would also follow follow Scenario 2. The most recent work, the *Thread Cluster Memory Scheduler* (TCM) [27] extends the general concept of the ATLAS approach. In TCM, unfriendly workloads with high row-buffer locality, that utilize a single DRAM bank for an extended period of time, are given less priority in the system, such that they interfere less frequently with the other workloads.

Prefetch Scheduling: Lee, *et al.* [31] propose a *Prefetch-Aware* controller

priority, where processors with a history of wasted prefetch requests are given lower priority. In the proposed approach, prefetch confidence and latency criticality are generated for each request, based on state of the prefetch stream combined with a history of stream behavior. With this more precise per request information, more accurate decisions can be made. Lin, *et al.* [34] proposed a memory hierarchy that coordinated the operation of the existing prefetch engine with the memory controller policy to improve bus utilization and throughput. In their hierarchy, the prefetch engine issues requests that are spatially close to recent demand misses in L2 with the memory controller sending the requests to memory only when the memory bus is idle. Their prefetcher relies on a column-centric address hash which introduces unfairness in the system that is not addressed in the proposal.

DRAM power reduction through page-mode writes: Several approaches have been proposed for DRAM power management, however most leverage memory sleep states, rather than exploiting page mode power savings. [24] considers the power cost of opening and closing pages, and [36] proposes a *Page Hit Aware Write Buffer* (PHA-WB), a 64-entry structure residing between the memory controller and DRAM, which holds onto writes until their target page is opened by a read. The PHA-WB, however, was evaluated for a writethrough cache, for which memory-level access locality will be much more apparent than in a writeback structure. Writeback caches provide significant improvements in available memory bandwidth and power consumption, so are thus a more realistic baseline for many-core server-class systems.

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 [9] work, the authors propose taking advantage of the inherent refresh that occurs through existing `read` and `write` operations when ranks are precharged. In ESKIMO [18], 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. 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 [61]. 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" [2], 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 [6]. 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 [8], 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 was found to be quite poor.

Chapter 5

Characterization of Memory System Behavior

In order to motivate the work, the following section presents the particular challenges associated with DRAMs systems. Analysis identified these characteristics as the most important factors in addressing memory performance:

5.1 Bus turnaround penalty

As described in the introduction, memory IO frequencies have been improving (resulting in raw bandwidth increases), but timing constraints related to signaling and electrical integrity have, for the most part, remained constant. This is especially well illustrated by tWRT, which defines the minimum delay from the completion of a write to the initiation of a read at the same DRAM rank. tWRT is 7.5ns on DDR3 devices – at 4GHz, this is 30 CPU cycles – a very significant penalty for any reads issued after a write.

Figure 5.1 conceptually illustrates DDR3 write command timings. Successive writes can be issued back-to-back, realizing 100% data bus utilization (Figure 5.1.a). On the other hand, when a read follows a write to the same device, the read must not only wait for the write's completion, but also for the bus turnaround time to elapse. This adds noticeable latency to the read operation

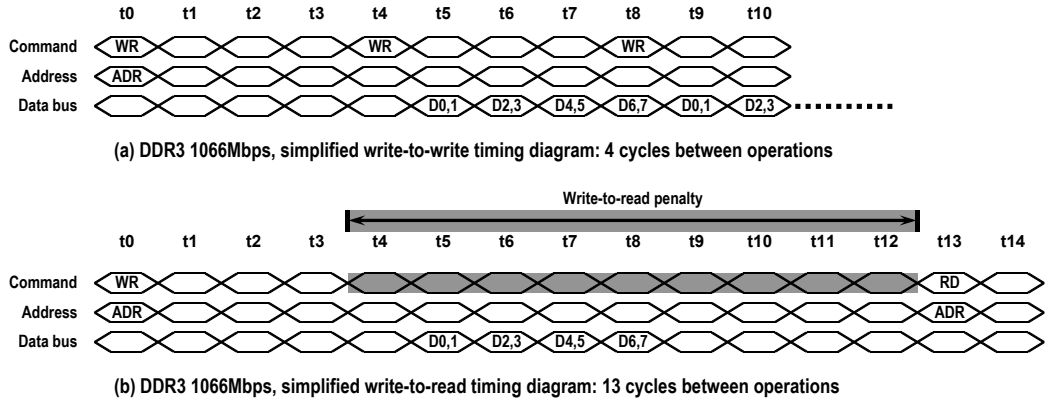


Figure 5.1: Write-to-read turnaround noticeably worsens command latency and databus utilization. tWPST [22] further worsens turnaround, but has been removed for simplicity

(Figure 5.1.b), and results in dismal data bus usage: 31%. For server-class 1066Mbps DDR3, the extra nine DRAM cycles between a write and read amount to a ≈66 cycle read penalty at 4GHz.

Memory scheduling efficiency is thus heavily influenced by the mixing of read and write operations. Timing gaps are required when read and write operations are intermingled in a continuous memory access sequence, but with many application streams and limited queuing resources, these turnarounds are generally difficult for the memory controller to avoid. As shown in Figure 5.2, memory data bus utilization can be greatly improved by significantly increasing the number of consecutive read/write operations the memory controller issues before switching the operation type on the bus. For example, if the scheduler can manage 32 reads/writes per scheduling block, utilization grows to 94%. Throughout the

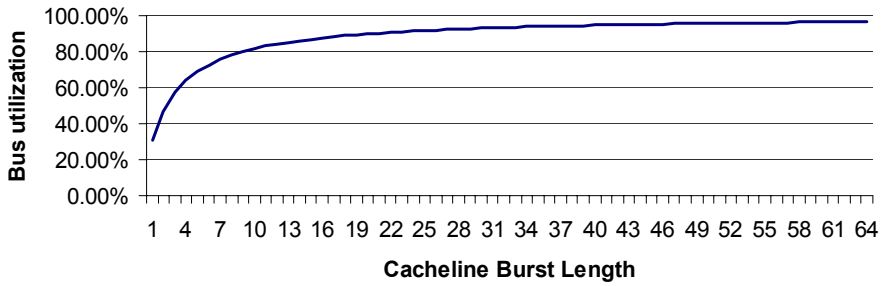


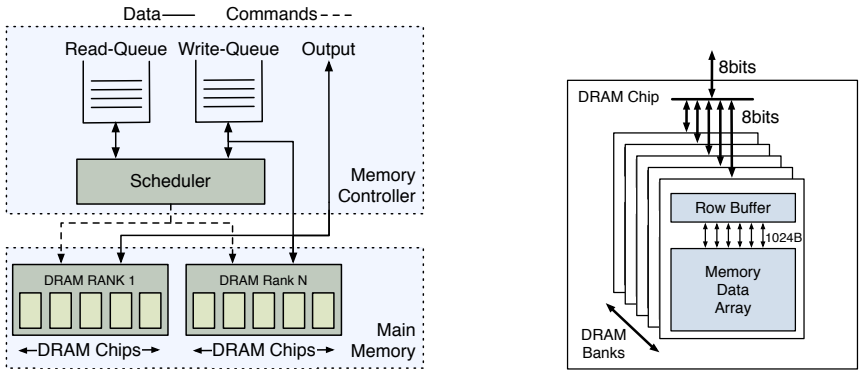
Figure 5.2: Bus utilization based on cache burst lengths

text, streams of consecutive reads or writes are referred to as a *cacheline burst* to memory.

5.2 Page Mode

A typical memory controller organization is shown in Figure 5.3(a). A significant component of this work proposes optimizations to improve the access latency and increase the sustained efficiency of the IO interface connecting main memory with the processors. This is accomplish through policy improvements to the memory scheduler and how operations are mapped to the DRAM devices.

While DRAM devices output only 16-64 bits per request (depending on the DRAM type and burst settings), internally, the devices operate on much larger, 1KB pages (also referred to as *rows*). As shown in Figure 5.3(b), each DRAM array access causes all 1KB of a page to be read into an internal array called *Row Buffer*, followed by a "*column*" access to the requested sub-block of data. Since the read latency and power overhead of the DRAM cell array access have already been paid,



(a) Memory controller and organization (b) DRAM Bank structure

Figure 5.3: Memory Organization

accessing multiple columns of that page decreases both the latency and power of subsequent accesses. These successive accesses are said to be performed in *page mode* and the memory requests that are serviced by an already opened page loaded in the row buffer are characterized as *page hits*.

Due to the reductions in both latency and energy consumption possible with page mode, techniques to aggressively target page mode operations are often used. There are downsides however, which must be addressed. Leaving a specific page open produces a higher access latency to other rows in the same bank (*page conflict*). In addition, certain scheduling algorithms such as the *First-Ready, First-Come-First-Served* (FR-FCFS) [51] give higher priority to page hit operations, which can result in unfairness for non page hit operations. Therefore, although row buffer hits are useful, they must be used in moderation.

The potential benefits (and caveats) of page mode are the following:

1. **Latency Effects:** A read access to an idle DDRx DRAM has a latency of 25ns. An access to an already opened page reduces this latency in half to 12.5ns. Conversely, the accesses to different rows in the DRAM bank can result in increased latency. Overall, increases in latency are caused by two mechanisms in the DRAM. Firstly, if a *row* is left open in an effort to service page hits, to service a request to another page incurs a delay of 12.5ns to close the current page followed by the latency to open and access the new page. Secondly, DRAM devices specify a minimum delay between back-to-back activations of two different rows within the same bank. This delay, known as *tRC* parameter, has remained approximately 50ns across the most recent DDRx DRAM devices. In a system that attempts to exploit page mode accesses, the overall effect on loaded memory latency and program execution speed due to the combination of these properties can significantly increase the observable latency. In addition, the DRAM device access is only a component of the processor's observed access latency to memory. For example, in the Intel I7 processor design the DRAM contributes 25ns of the total latency of 65ns memory latency [44]. Thus while a page hit halves the DRAM device latency, the observed latency reduction is only 20%. To summarize, the relative capacity and latency of the various levels in a modern memory hierarchy are shown in Figure 5.4. Note the small latency and capacity of the row buffer, which limits any benefit to spatial locality on capacity based cache misses.

2. **Bank Utilization:** The utilization of the DRAM banks can be a critical parameter in achieving high scheduling efficiency. If bank utilization is high,

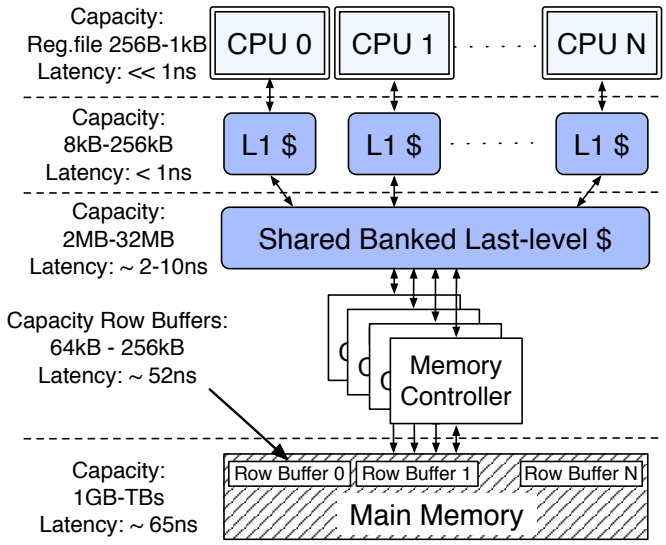


Figure 5.4: Capacities and latencies of memory

the probability that a new request will conflict with a busy bank is greater. As the time to activate and precharge the array overshadows data bus transfer time, having available banks is often more critical than having available data bus slots. Increasing the data transfered with each DRAM activate, through page mode, amortizes the expensive DRAM bank access, reducing utilization. Figure 5.5 shows the bank utilization of a DDR3 1333 MHz system, with two devices (*ranks*) sharing a data bus at 60% bus utilization. A closed-page policy, with one access per activate would produce an unreasonably high bank utilization of 62%. However, the utilization drops off quickly as the accesses per activate increases. For example four accesses per activate reduces the bank utilization to 16%, greatly reducing the probability that a new request will be delayed behind a busy bank.

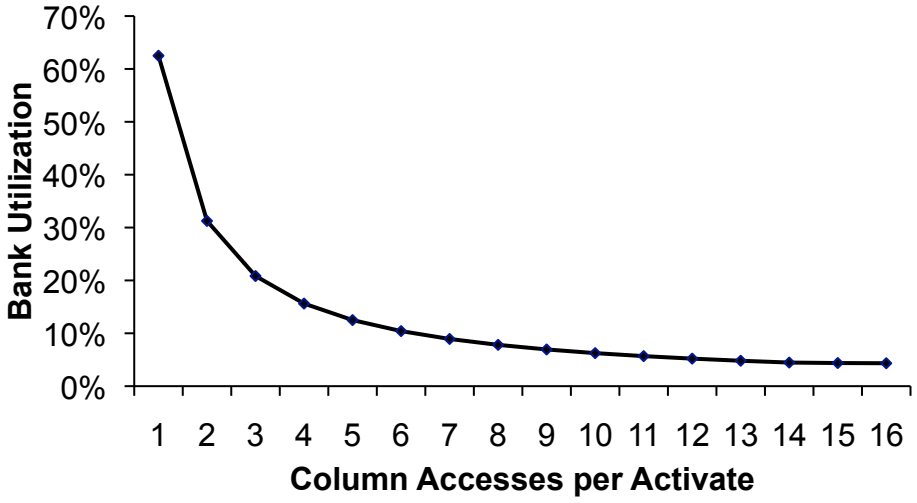


Figure 5.5: Improvements for increased access per activation: Bank utilization for a 2-Rank 1333 Mhz system at 60% data bus utilization

3. **Power Reduction:** Page mode accesses reduce DRAM power consumption by amortizing the activation power associated with reading the DRAM cells data and storing them into the row buffer. Figure 5.6 shows the DRAM power consumption of a 2GBit DDR3 1333MHz DRAM as the number of row accesses increases. As the power corollary of Amdahl's law predicts, since page mode only reduces the page activation power component, DRAM power quickly becomes dominated by the data transfer and background (not proportional to bandwidth) power components.

4. **Other DRAM Complexities:** Beyond the first order effects described above, more subtle DRAM timing rules can have significant effects of DRAM utilization, especially as the data transfer clock rates increase in every DRAM generation. As many DRAM parameters do not scale with frequency increases

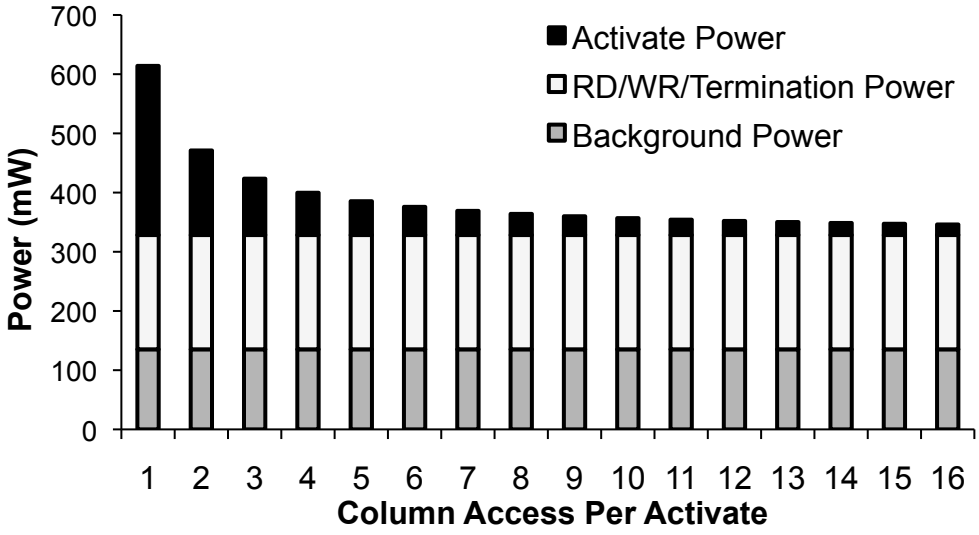


Figure 5.6: Improvements for increased access per activation: DRAM power for each 2GBit DDR3 1333 Mhz at 40% read, 20% write utilization [42]

due to either constant circuit delays and/or available device power. One example is the TFAW parameter. TFAW specifies the maximum number of activations in a rolling time window in order to limit peak instantaneous current delivery to the device. In a 1333MHz DDR3, the TFAW parameter specifies a maximum of four activations every 30ns. A transfer of a 64 byte cache block requires 3ns, thus for a single transfer per activation TFAW limits peak utilization to 80% ($6ns * 4/30ns$). However, with only two accesses per activation, TFAW has no effect ($12ns * 4/30ns > 1$). The same trend is observed across several other DRAM parameters, where a single access per activation results in efficiency degradation, while a small number of accesses alleviates the restriction.

In summary, a relatively small number of accesses to a page is very effective

in taking advantage of DRAM page mode for both scheduling and power efficiency. For example, at four row accesses per activation, power and bank utilization are 80% of their ideal values. Latency effects are more complex, as scheduling policies to increase page hits also increase bank conflicts, making raw latency reductions difficult to achieve.

5.2.1 Write Page mode opportunities

Page mode DRAM access greatly improves both memory utilization and power characteristics, but the optimization possibilities for read and write operations are significantly different. Reads are visible as the program (or a prefetch engine) generates them; this should enable spatial locality in reference sequences to be executed in page mode. In contrast, write operations (in the common writeback cache policy) are generated as older cache lines are evicted to make room for newly allocated lines. As such, spatial locality at eviction time can be obscured through variation in set usage between allocation and eviction, as shown in Figure 5.7.

Figure 5.7 shows the total number of page mode writes possible for various commercial workloads, for a range of memory controller write queue sizes. The workloads and simulation environment for this characterization data are described in Section 7.8. For practical write queue sizes, such as 32 entries, there is essentially no page mode opportunity (approximately one write possible per page activate). That stated, a large amount of spatial locality is contained within the various cache levels of the system, but today's CPU-centric caches do not give the memory controller visibility into this locality. Significantly larger memory

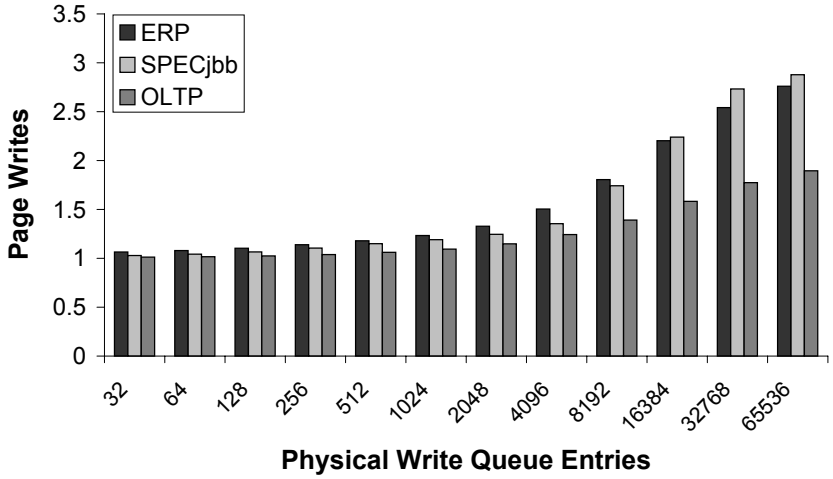


Figure 5.7: Characteristics of commercial workloads: Page writes per activate *vs.* number of Physical Write Queue entries

controller write queues, though impractical, could provide the needed visibility and enable significant page mode opportunities.

Note, the commercial workloads shown here reflect lower amounts of potential write page hits compared to the SPEC CPU 2006 workloads. For example, the OLTP workload has limited write page-mode opportunities even for very large write queue sizes. With SPEC CPU workloads, single threaded version with 32 write queue entires were effective in finding many page mode operations. To achieve these write page levels with multiple processors, the sizes needed to scale linearly. For example an eight core system achieved single core hit rates with a 256 entry write queue structure. As the write queue lookup is associative, growing the stricture to even 256 entires would require some kind of multi-level structure, which adds significant complexity.

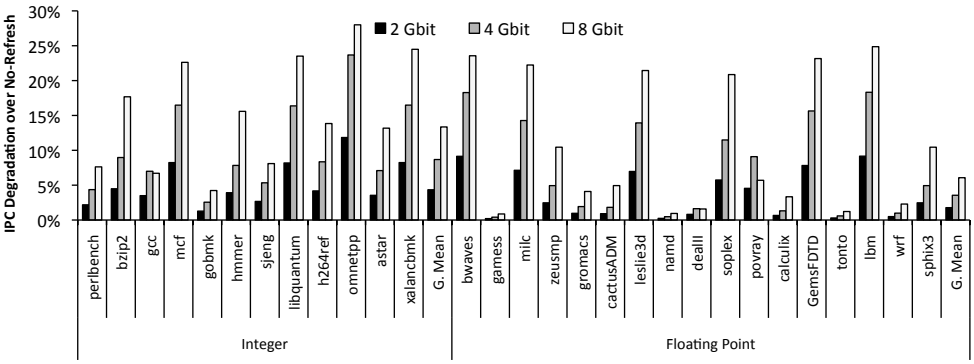


Figure 5.8: Refresh performance penalty for emerging DRAM sizes (four-core). See Section 8.6.1 for a description of the modeled architecture.

5.3 Refresh Penalty

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 [53], and would be anticipated to appear on the 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 [12] in Figure 5.8, 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 [20].

5.3.1 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 [16,

43]. 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 [41]. 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 [20]. Ideally, DRAM devices would simply refresh more bits with each operation, but this would over-tax the available current delivery. The length of time of this delay is the parameter `tRFC` (ReFresh Cycle time). Table 5.1 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.

5.3.2 Refresh Cycle Time Beyond JEDEC DDR3

Table 5.1 contains the JEDEC DDR3 `tRFC` values. Projections to future values are difficult due to the seeming discontinuity between the trend lines shown in Figure 5.9. 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.

Table 5.1: Refresh parameters as density increases [22]

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

5.4 Bursty behavior

Most programs exhibit bursty behavior. At the last-level cache, this results in phases when a large number of load misses must be serviced. Common cache allocation/eviction policies compound this effect, as bursts of cache fills create

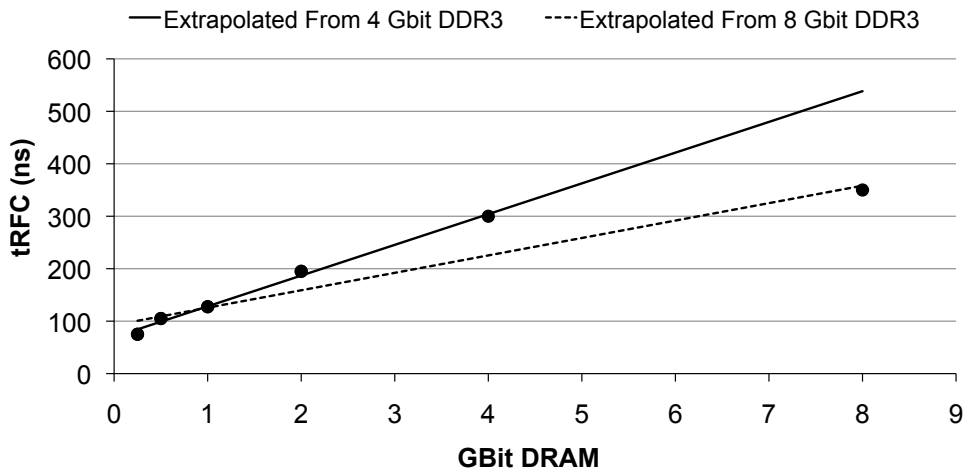


Figure 5.9: tRFC Across DDR3 Generations

bursts of forced writebacks, thus clustering bursts of reads with bursts of writes.

Figure 5.10 shows the distribution of time between main memory operations for three workloads. The workloads and analysis are discussed in more detail in Section 7.8. Note that 20-40% of all memory requests occur with less than ten cycles delay after the previous memory operation, while the median can be in the hundreds of cycles and many requests take significantly longer.

Over-committed multi-threaded systems have always experienced some degree of cache thrashing, as workloads evict one another's data, and threads re-warm their local cache. However, this is no longer a problem seen only in large-scale SMP systems. Instead, this phenomenon is inherent to today's virtualized computing environments as disparate partitions share a physical CPU. If write operations can be executed early, bursts of read operations execute with lower

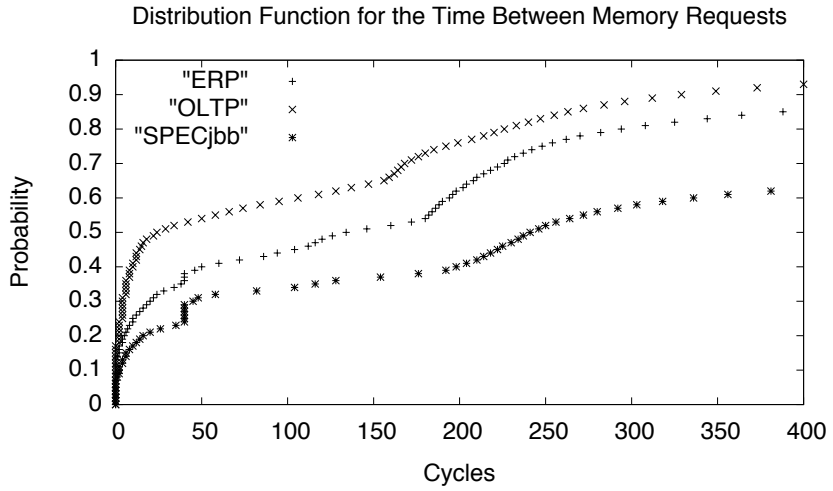


Figure 5.10: Characteristics of commercial workloads: distribution of time between memory requests

combined bandwidth demands. Ideally, the memory controller will always have write operations ready to be sent to idle DRAM resources.

5.5 Balanced Designs and Importance of Cache Capacity and Memory Bandwidth

An efficient design requires each of the structures in the system to be appropriately balanced. If certain components are over-sized, these resources are wasted while undersizing structures create bottlenecks. As analyzed in "Scaling the bandwidth wall" [52], the balance between number of cores, cache capacity, and memory bandwidth must scale appropriately into future generations. In practice, a system will service workloads with a wide range of cache capacity requirements and memory bandwidth pressure. Therefore, a well designed system will saturate

the memory interface on a reasonable subset of workloads. However, this leaves an important fraction of workloads where the bandwidth usage is significant, yet not saturated.

This insight has important implications into systems design. Specifically, the analysis indicated that the fullness of the read input queues of the memory controller is relatively low for many workload combinations. The only cases where significant queue depths were observed was in cases of high bandwidth workloads executing together, saturating the memory interface. Due to the low fullness of the memory structures in these cases, policies which employ reordering of requests in the memory controller as the primary control point are inherently ineffective for workloads where the memory interface is below saturation. This is a significant motivator for this work.

Chapter 6

Minimalist Open-page Policy

As management of the DRAM row buffer is fundamental aspect of the memory system design, this chapter covers the baseline policy assumed in the subsequent sections. Row buffer hits enable potentially lower acces latency and power consumption combined with reduced DRAM resource utilization. Yet, row buffer hit induced starvation has been shown to be a significant problem in multi core systems [45]. This work utilizes a Minimalist open-page policy, where "just enough" page mode hits are produced, while avoiding row buffer hit induced starvation. This chapter covers the positive and negative aspect of page mode hits, and how a Minimalist policy balances between these properties.

6.1 Row Buffer Locality in Modern Processors

This section describes observations as to the characterization of page mode access as seen in current workstation/server class designs. Contemporary CMP processor designs have evolved to impressive systems on a chip. Many high performance processors (eight in current leading edge designs) are backed by large last-level caches containing up to 32 MB of capacity [23]. A typical memory hierarchy that includes the DRAM row buffer is shown in Figure 5.4. As a large last-

level cache filters requests to memory, row buffers inherently exploit only spacial locality. Temporal locality within the program results in hits to the much larger last-level cache.

Access patterns with high levels of spatial locality, that miss in the large last level cache, are often very predictable. In general, speculative execution and more specifically modern prefetch algorithms can be exploited to generate memory requests with spatial locality in dense access sequences. Consequently, the relatively small latency benefit of page mode is hidden.

6.2 Bank and Row Buffer Locality Interplay With Address Mapping

Commonly used open-page address mapping schemes put all column bits directly above the cache block to capture the greatest possible number of page hits [26, 34, 47, 51]. As identified by Moscibroda, *et al.* [45], this hashing can produce interference between the applications sharing the same DRAM devices, resulting in significant performance loss. The primary problem identified in that work is due to the FR-FCFS [51] policy where page hits have a higher priority than requests with lower page affinity. Beyond fairness, schemes that map long sequential address sequences to the same row, suffer from low bank-level parallelism (BLP). If many workloads with low BLP share a memory controller, it becomes inherently more difficult to interleave the requests, as requests from two workloads mapping to the same DRAM bank will either produce a large number of bank conflicts, or one of them has to stall, waiting for all of the other workload's

request to complete, significantly increasing it's access latency.

Figure 6.1 illustrates an example where workload A generates a long sequential access sequence, while workload B issues a single operation mapping to the same DRAM. With a standard open-page policy mapping, both requests map to the same DRAM bank. With this mapping, there are two scheduling options, shown in Scenarios 1 and 2. The system can give workload A higher priority until all page hits are completed, significantly increasing the latency of the workload B request (Scenario 1, Figure 6.1). Conversely, workload A can be interrupted, resulting in very inefficient activate to activate commands conflict for request A4 (Scenario 2, Figure 6.1), mainly due to the time to load the new page in the row buffer and the unavoidable *tRC* timing requirement between back-to-back activations of a page in a bank. Neither of theses solutions optimize fairness and throughput. This proposal adapts the memory hash to convert workloads with high row buffer locality (RBL), into workloads with high bank-level parallelism. This is shown in Scenario 3 of Figure 6.1, where sequential memory accesses are executed as reading four cache blocks from each row buffer, followed by switching to the next memory bank. With this mapping, operation B can be serviced without degrading the traffic to workload A.

6.3 Proposed Minimalist Open-page Mode Scheme

As analyzed in the motivation section, the *Minimalist Open-page* scheme is based on the observation that most of the page-mode gains can be realized with a relatively small number of page accesses for every page activation. In

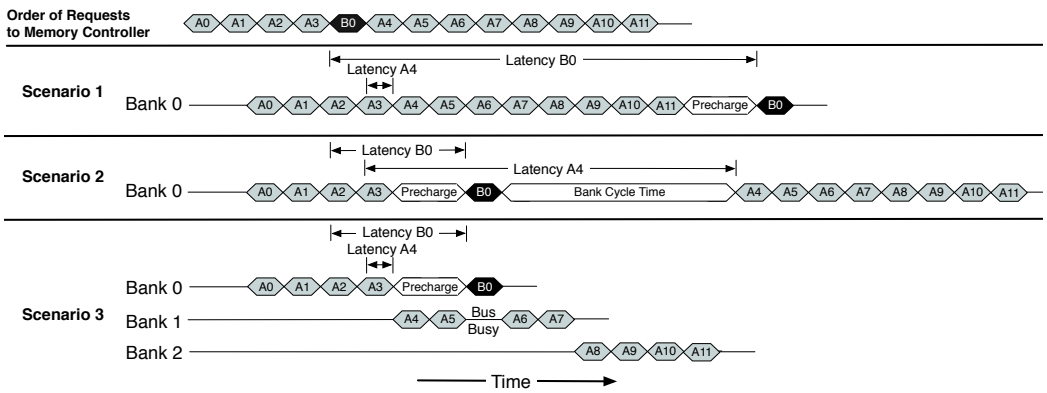


Figure 6.1: Row buffer policy examples

addition, address hashing schemes that map sequential regions of memory to a single DRAM page result in poor performance due to high latency conflict cases. The Minimalist policy defines a target number of page hits that enable a careful balance between the benefits (increased performance and decreased power), and the detractors (resource conflicts and starvation) of page-mode accesses. With this scheme several system improvements are accomplished. Firstly, through the address mapping scheme, fairness is provided. This alleviates the memory request priority scheme requirements compared to prior approaches that must address row-buffer starvation. This work proposes the scheduler can focus its priority policy on memory request criticality, which is important in achieving high system throughput and fairness.

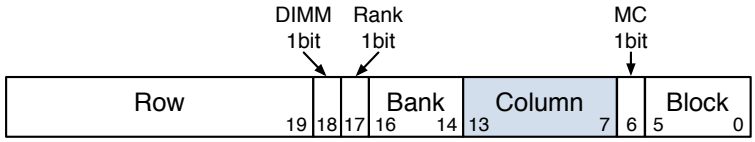
As described in Section 6.1, most of the memory operations with "page-mode" opportunities are the results of memory accesses generated through prefetch operations. Therefore, the prefetch engine in the processor is used to provide request meta-data information which directs the scheme's page-mode accesses and

request priority scheme. In effect, this enables the prefetch generated page-mode access to be done reliably, with back to back scheduling on the memory bus.

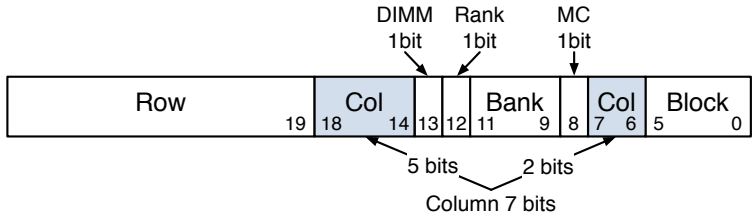
In the remainder of this section the Minimalist policy is described in detail. First, the fair address mapping scheme that enables bank-level parallelism with the necessary amount of row-buffer locality is described. This is followed by the prefetch hardware engine, as this component provides prefetch request priorities and prefetch-directed page mode operation. Finally, the scheduling scheme for assigning priorities and issuing the memory request to the main memory is covered.

6.4 DRAM Address Mapping Scheme

The differences between a typical mapping and the one used in this proposal are summarized in Figure 6.2. The basic difference is that the *Row Column* access bits that are used to select the row buffer columns are split in two places. The first 2 LSB bits (Least Significant Bits) are located right after the *Block* bits to allow the sequential access of up to 4 consequent cache lines in the same page. The rest of the MSB (most significant bits) column bits (five bits in this case since 128 overall cache lines stored in every row buffer) are located just before the *Row* bits. Not shown in the figure for clarity, higher order address bits are XOR-ed with the bank bits shown in the figure to produce the actual bank selection bits. This reduces row buffer conflicts as described by Zhang, *et al.* [64]. The above combination of bits selection allows workloads, especially streaming, to distribute their accesses to multiple DRAM banks; improving bank-level parallelism and avoiding over-utilization of a small number of banks that leads to thread starvation and priority



(a) Typical Mapping



(b) Proposed Mapping

Figure 6.2: System Address Mappings to DRAM Address - Example system in figure has 2 memory controllers (MC), 2 Ranks per DIMM, 2 DIMMs per Channel, 8 Banks per Rank and 64B Cache-lines.

inversion in multi-core environments.

6.4.1 DRAM Page Closure (Precharge) Policy

In general, the Minimalist policy does not speculatively leave DRAM pages open. If a multi-line prefetch request is being processed, the page is closed with an auto-precharge sent with the read command (In DDRX the auto-precharge bit indicates to close the page after the data are accessed [19]) . This saves the command bandwidth of an explicit precharge command. For read and single line prefetch operations, the page is left open based on the following principle. The *tRC* DRAM parameter specifies the minimum time between activations to a DRAM bank. The *tRC* is relativity long at 50ns compared to the precharge delay of 12.5ns.

Therefore, closing a bank after a single access does not allow reactivation of the bank until the *tRC* delay expires. With this insight, pages are speculatively left open for the *tRC* window, as this provides for a "free" open page interval.

6.5 Evaluation

The scheme was evaluated through a simulated an 8 core CMP system using the Simics functional model [38] extended with the GEMS toolset [39]. The system was configured with an aggressive out-of-order processor model from GEMS combined with a detailed memory subsystem. In addition, the default memory controller was augmented to simulate a DDR3 1333MHz DRAM using the appropriate memory controller policy for each experiment. Table 6.1 summarizes the full-system simulation parameters used in the study.

The evaluation utilized a set of multi-programmed workload mixes from the SPEC cpu2006 suite [12]. 27 randomly selected 8-core mixes spanning from low bus utilization levels to saturation were selected. This was accomplished with summations of the single core bandwidth requirements of a large number of randomly selected workloads. 27 sets were then selected by choosing the total bandwidth target to span from 15% to 300% of the available peak memory bandwidth. The sets are ordered from lower to higher bus utilization. In addition, the workloads were divided in *low*, *medium*, and *high* sets with nine workloads each. The bandwidth threshold between low and medium is 35%, while the medium to high is 70% (the point where the system enters bus saturation). For the evaluation, each experiment was executed in Simics to its most representative phase; the next

Table 6.1: Full-system detailed simulation parameters

Core Characteristics	**Clock Frequency**	**Pipeline**	**Reorder Buffer /Scheduler**	**Branch Predictor**
	4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries	Direct YAGS / indirect 256 entries
Prefetcher	H/W stride n with dynamic depth, 32 streams / core			
Memory Subsystem	**L1 Data & Inst. Cache**	**L2 Cache**	**Outstanding Requests**	**Memory Latency**
	64 KB, 2-way associative, 3 cycles access time, 64B blocks	16 MB, 8 ways associative, 12 cycles bank access, 64B blocks	16 Requests per Core	65ns
	Controller Organization	**DRAM**	**Controller Resources**	**Memory Bandwidth**
	2 Memory Controllers 2 Ranks per Controller 8 DRAM chips per Rank	DDR3 1333MHz 8-8-8	32 Read Queue & 32 Write Queue Entries	21.333 GB/s

100M instructions to warm up the caches and memory controller structures; and then simulated until the slower benchmark completes 100M instructions. Only the statistics gathered for the representative 100M instruction phase after the warming up period were used for speedup calculations.

For each experiment a speed-up estimation was calculated using the weighted throughput, $\sum(IPC_i/IPC_{i,FR-FCFS})$, where $IPC_{i,FR-FCFS}$ is the IPC of the i-th application measured in the FR-FCFS baseline system using an open-page policy memory controller. In addition, to estimate the execution fairness of every proposal, the harmonic mean of weighted speedup, $Fairness = N/\sum(IPC_{i,alone}/IPC_i)$, where $IPC_{i,alone}$ is the IPC of the *i*-th application when it was executed standalone, was utilized. This method was previously suggested by Luo, *et al.* [37].

The *Minimalist Open-page* scheme is compared against three of the most representative memory controller policies proposed in the past: a) *Parallelism-aware Batcher Scheduler* (PAR-BS) [47], b) *Adaptive per-Thread Least-Attained-Service* memory scheduler (ATLAS) [26], and c) *First-Ready, First-Come-First-Served* (FR-FCFS) [51] with open-page policy.

6.5.1 Throughput

The analysis compares the speedups PABS, ATLAS, and Minimalist to FC-FCFS. The results are shown in Figure 6.3. For the lowest bandwidth workloads (1 to 4) no improvements are observed over FR-FCFS. As the memory utilization increases, some reasonable gains are seen on workloads 5 and 7. For the medium bandwidth workloads, significant gains for Minimalist are observed over all other

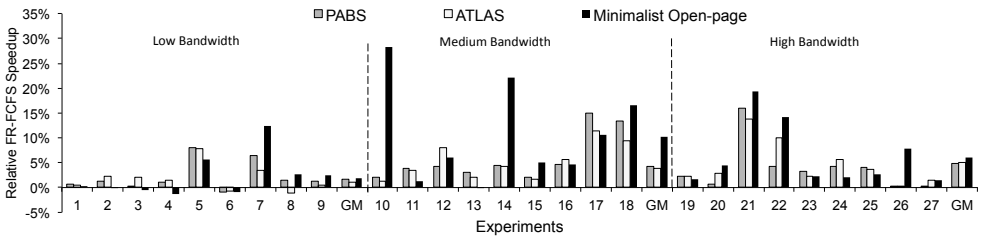


Figure 6.3: Speedup of PABS, ATLAS, and Minimalist relative to FR-FCFS

policies. For example, in workload set 10, Minimalist achieves more than 28% gains over all other policies. These gains are mainly due to the memory streaming workload `libquantum` as Table 6.2 shows. `Libquantum` is disruptive in systems with typical open-page mapping, where low bank level parallelism is observed. Minimalist does well in this case, due to the inherently fair memory hash, where `libquantum` does not ”park” on a specific memory bank for many sequential requests. In addition, both PABS and ATLAS are effective only when the memory queues contain reasonable numbers of operations. This does not occur unless bandwidth is saturated.

For high bandwidth workloads, such as experiment sets 18, 21 and 22 from Table 6.2, significant gains are seen compared to the ATLAS and PAR-BS policies. However, the overall geometric mean of the high bandwidth cases exhibits a slightly lower speedup. The larger amount of memory queuing enables the PABS and ATLAS policies to be effective. As the Minimalist hash solves row-buffer induced starvation without utilizing the memory scheduler priority, further enhancements to the priority scheme beyond FR-FCFS are possible. These optimizations are left for future work.

Table 6.2: Randomly selected workloads for 8-core SPEC cpu2006 workload sets

Exp. #	**Workload Sets (Core-0 → Core-7)**
1	xalancbmk, xalancbmk, omnetpp, soplex, lbm, omnetpp, wrf, zeusmp
2	bzip2, omnetpp, astar, libquantum, xalancbmk, soplex, sjeng, sjeng
3	gcc, leslie3d, zeusmp, sjeng, zeusmp, libquantum, mcf, gcc
4	gcc, namd, lbm, namd, soplex, lbm, tonto, milc
5	wrf, omnetpp, leslie3d, gamess, xalancbmk, tonto, lbm, xalancbmk
6	omnetpp, namd, GemsFDTD, leslie3d, calculix, GemsFDTD, bzip2, wrf
7	omnetpp, mcf, cactusADM, xalancbmk, mcf, omnetpp, GemsFDTD, gamess
8	gcc, omnetpp, omnetpp, xalancbmk, dealII, xalancbmk, cactusADM, libquantum
9	lbm, gamess, xalancbmk, sphinx3, mcf, soplex, omnetpp, omnetpp
10	gcc, soplex, tonto, soplex, leslie3d, libquantum, namd, astar
11	namd, xalancbmk, leslie3d, soplex, dealII, tonto, sphinx3, mcf
12	cactusADM, libquantum, libquantum, milc, gamess, mcf, omnetpp, soplex
13	omnetpp, namd, soplex, libquantum, h264ref, astar, lbm, lbm
14	soplex, xalancbmk, lbm, milc, omnetpp, perlbench, mcf, milc
15	libquantum, mcf, soplex, gromacs, omnetpp, xalancbmk, omnetpp, bwaves
16	xalancbmk, libquantum, lbm, gamess, omnetpp, mcf, xalancbmk, namd
17	hmmer, sphinx3, xalancbmk, cactusADM, libquantum, xalancbmk, zeusmp, GemsFDTD
18	GemsFDTD, wrf, gromacs, lbm, lbm, sphinx3, cactusADM, mcf
19	libquantum, astar, libquantum, sphinx3, xalancbmk, sphinx3, wrf, h264ref
20	bzip2, calculix, soplex, milc, lbm, xalancbmk, libquantum, namd
21	lbm, libquantum, lbm, mcf, cactusADM, lbm, xalancbmk, perlbench
22	leslie3d, sphinx3, xalancbmk, bzip2, h264ref, leslie3d, GemsFDTD, gobmk
23	sphinx3, sjeng, sphinx3, xalancbmk, leslie3d, mcf, soplex, xalancbmk
24	perlbench, soplex, lbm, lbm, xalancbmk, milc, libquantum, calculix
25	bwaves, leslie3d, omnetpp, xalancbmk, soplex, mcf, leslie3d, GemsFDTD
26	bwaves, libquantum, xalancbmk, namd, libquantum, libquantum, omnetpp, GemsFDTD
27	GemsFDTD, perlbench, lbm, astar, libquantum, xalancbmk, libquantum, zeusmp

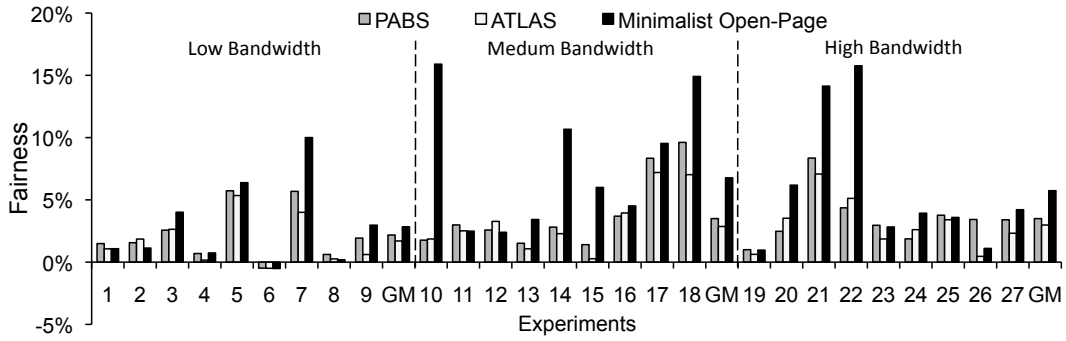


Figure 6.4: Fairness, compared to FR-FCFS

6.5.2 Fairness

Figure 6.4 shows the fairness improvement of all schemes relative to the FR-FCFS baseline system using the harmonic mean of weighted speedup. It is important to note that the throughput gains Minimalist achieves are accompanied with improvements in the fairness. This is expected as the throughput gains are a result of alleviating unresolveable conflict cases associated with row buffer starvation. Essentially, Minimalist matches the throughput gains in cases without row buffer conflicts while significantly improving cases where row buffer conflicts exist.

As explained by Kim [26], ATLAS is less fair than PABS, since ATLAS targets throughput over fairness (interestingly similar throughput was observed for both algorithms in these experiments). Minimalist improves fairness up to 15% with an overall improvement of 6.4%, 2.8% and 2.1% over the medium bus utilization workload sets, and 5.74%, 3.48%, 2.87% over the high bus utilization workload sets for FCFR, PABS and ATLAS, respectively.

Chapter 7

The Virtual Write Queue

In computer architecture, caches have primarily been viewed as a means to hide memory latency from the CPU. Cache policies have focused on anticipating the CPU's data needs, and are mostly oblivious to the main memory. This chapter demonstrates that the era of many-core architectures has created new main memory bottlenecks, and mandates a new approach.

This chapter proposes the Virtual Write Queue, which utilizes the cache for memory optimization purposes, dramatically expanding the memory controller's visibility of processor behavior at low implementation overhead. Through memory-centric modification of existing policies, such as scheduled writebacks, this work demonstrates that performance-limiting effects of highly-threaded architectures can be overcome. Through an awareness of the physical main memory layout and by focusing on writes, both read and write average latency can be shortened, memory power reduced, and overall system performance improved.

Typical memory controllers contain 10's of queued write operations [55]. Queue structures are costly in terms of area and power, and the size of the write queue is a critical parameter for overall memory subsystem performance. The effective size of the write queue can be greatly increased using the lower LRU

members of the last-level cache function as a very large write queue e.g., 64k effective entries for the lower one-quarter of a 16 MB last-level cache). This LRU section of the last-level cache is referred to as the *Virtual Write Queue*, in that the usage of this region is overload and re-purposed as a much larger effective queue structure. Analysis is shown that a coordinated cache/memory policy based on direct management from the *Virtual Write Queue* mitigates DRAM challenges, increasing the performance of the memory subsystem. Specifically:

1. **Bus turnaround penalty avoidance**: Through a *Scheduled writeback* policy, the system efficiently drain more pending write operations, minimizing the probability of interleaved read and write operations, which incur costly scheduling penalties ($\approx$66 processor cycles @ 4GHz).

2. **Gathering page mode opportunities**: Directed cache lookups, to a broad region of the LRU space, enable harvesting of additional writes to be executed in page mode at higher performance and lower power.

3. **Burst leveling**: With the *Virtual Write Queue*, the ability to buffer $\approx$1000x more write operations than in a standard memory controller enables significantly greater leveling of memory traffic bursts.

Figure 7.1 shows the *Virtual Write Queue*, which logically consists of some LRU fraction of the last-level cache (1/4 in this example); the *Physical Write Queue* in the memory controller; and the added coordinating control logic and structures. This structure is referred to as virtual because no additional queueing structures

are added. The overall area cost is small: ≈0.3% overhead over a typical cache directory implementation.

Scheduled Writeback: Traditional writeback cache policies initiate memory writes only when a cache fill replaces an LRU cacheline. This is referred to as *forced writeback*. There are two problems with this policy. First, writes are only sent to the memory controller at the time of cache fills, so idle DRAM cycles cannot be filled with writes. This is addressed with *Eager Writeback* [32], where cachelines are sent to the memory controller when it appears to be idle (an empty write queue is detected). The second problem deals with the mapping of write locations to DRAM resources. Since the memory controller is aware of each DRAM's state, it would ideally decide which writeback operation can be executed most efficiently. With *Eager Writeback*, this selection is made by the cache, without knowledge of what would be best for DRAM scheduling. *Scheduled Writeback* are introduced to solve this problem. With *Scheduled Writeback*, the memory controller can direct the cache to transfer lines that map to specific DRAM resources.

As shown in Figure 7.2, the primary microarchitectural addition over traditional designs is the Cache Cleaner. The Cache Cleaner orchestrates *Scheduled Writebacks* from the last-level cache to the *Physical Write Queue*. While the cache and the *Physical Write Queue* are structurally equivalent to traditional designs (and thus hardware overhead is minimal), the logical behavior is significantly altered. This is reflected in the distribution of *dirty* lines in the system. Specifically, dirty lines have been cleaned from the lower section of the cache, and the *Physical Write Queue* is maintained at a higher level of fullness, with an ideal mix of commands

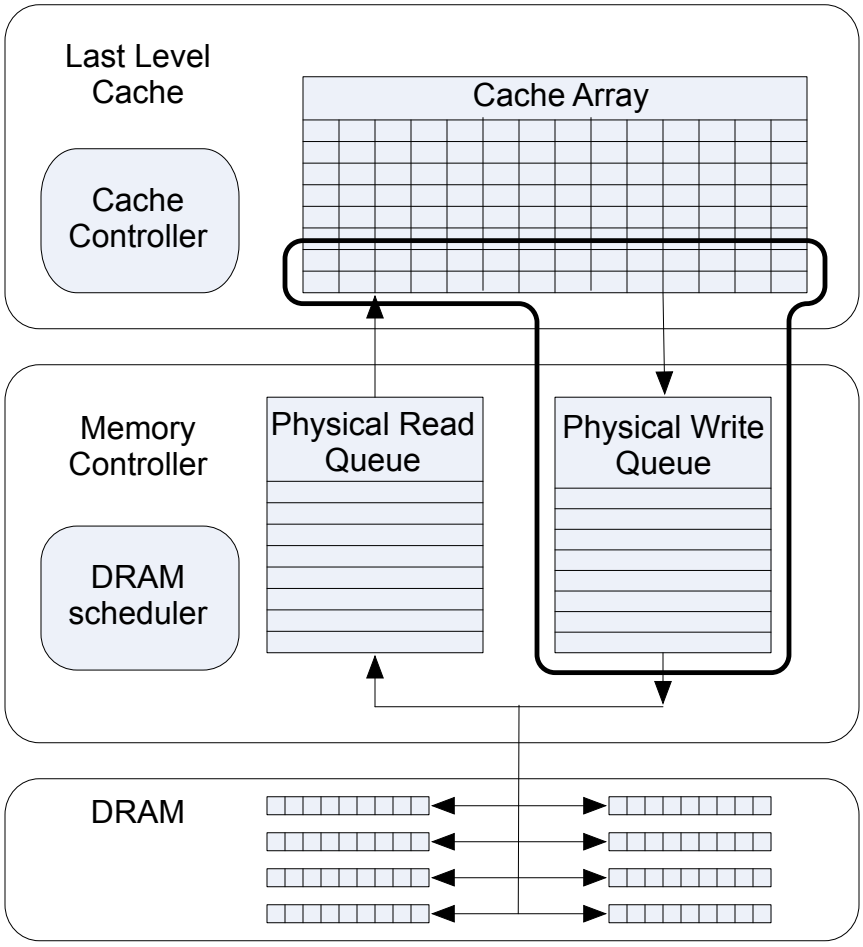


Figure 7.1: Memory controller and DRAM system with typical system with separate read and write queues. Proposed *Virtual Write Queue* outlined.

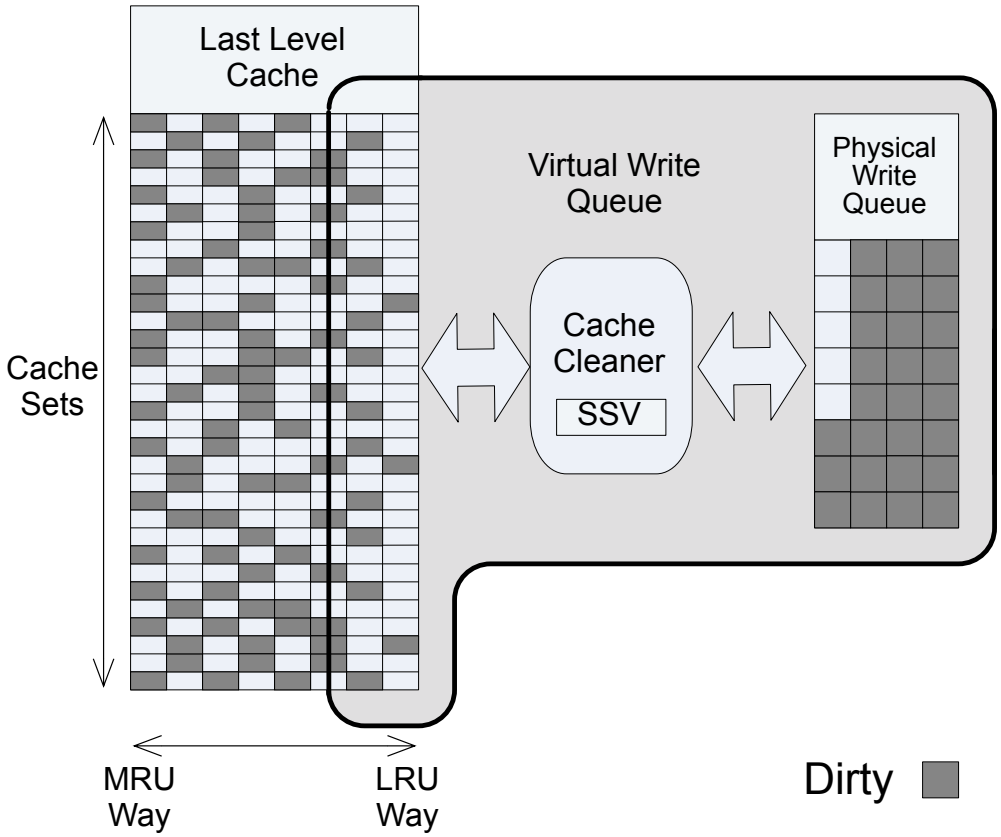


Figure 7.2: The proposed *Virtual Write Queue* details.

with respect to scheduling DRAM accesses. In addition, typical memory controllers decide write operation priority based on only the *Physical Write Queue*, whereas this system uses the much larger *Virtual Write Queue*. The *Physical Write Queue* becomes a directly managed staging buffer, of the now much larger window of write visibility.

7.1 High Level Description of Virtual Write Queue Policies

At steady state, the *Physical Write Queue* is filled to a defined ≈full level with a mix of operations to all DRAM resources. This level is chosen to keep the queue as full as possible, while retaining the capacity to receive cache writebacks. *Scheduled Writebacks* can vary in length, depending on the number of eligible lines found in the same DRAM page; the write queue must maintain capacity to absorb these operations.

The DRAM scheduler executes write operations based on the conditions of the DRAM devices, read queue operations, and the current write priority. Write priority is determined dynamically depending on the fullness of the *Virtual Write Queue*. As write operations are executed to DRAM, the fullness of the *Physical Write Queue* is decreased, so the Cache Cleaner refills the *Physical Write Queue* to the target level. The Cache Cleaner will search the last-level cache *Virtual Write Queue* region for write operations to the desired DRAM resource. This DRAM resource is chosen in two ways: 1) If the memory controller attempts a burst of operations to a specific rank, an operation mapping to that rank will be sent; 2) alternately, if no burst is in progress, the *Physical Write Queue* will be rebalanced by choosing the rank with the fewest operations pending. This maintenance of an even mix of operations to various DRAM resources enables opportunistic write execution, in that a write is always available to any DRAM resource that becomes idle.

As part of the Cache Cleaner function, additional writes are harvested which map to the same DRAM page as the write selected by the Cache Cleaner for

writeback. This is accomplished via queries to cache sets which map to the same DRAM page (Section 7.3). In the evaluated system, groups are defined from cache sets based on the cache and DRAM address translation algorithms. In the evaluation it was found that groups of four cache sets are the ideal size.

Upon completion of the *Scheduled Writebacks*, the *Physical Write Queue* once again contains an ideal mix of operations to be scheduled. While a linear sequence is described, in practice, the structure can concurrently operate on all steps, accommodating periods of high utilization.

7.2 Physical Write Queue Allocation

As previously described, a large barrier to efficient utilization of a main store (whether DRAM or future technologies) is the transition between read and write operations. In addition to write-to-read turnaround, alternating between different ranks on the same bus can introduce wasted bus cycles. To have good efficiency, DRAM banks must additionally be managed such that operations to the same bank, but to different pages, are avoided. These characteristics motivate creation of long bursts of reads or writes to ranks, while avoiding bank conflicts. The *Physical Write Queue* allocation scheme addresses the formation of write bursts.

A key aspect of the *Virtual Write Queue* solution is its two-level design. Since writes can only be *executed* from the *Physical Write Queue*, the *Scheduled Writebacks* must expose parallelism of the *Virtual Write Queue* into the *Physical Write Queue* to achieve the highest value from last-level cache buffering. This is accomplished by maintaining the best possible mix of operations in the *Physical*

Write Queue, given what is visible in the entire *Virtual Write Queue* structure.

This is accomplished in two ways. First, the capability is created to opportunistically execute *write operations* to any rank. In this way, the scheduler can react to a temporarily idle rank with a burst of writes at any moment. Extending this idea, several writes are maintained to each rank which can be executed without idle cycles. Ideally, many writes are maintained to the same DRAM pages. When it is not possible to maintain accesses targeting the same rank and page, operations are selected to the same rank, but a different bank. For a write burst to a rank that is longer than what can be stored in the *Physical Write Queue*, *Scheduled Writebacks* are directly generated in concert with execution of writes, such that the cache writeback latency is overlapped with the *Physical Write Queue* transfers to memory. Once the initial latency of the first cache writeback has passed, the cache has the required bandwidth to maintain a busy DRAM bus.

An example timing diagram for this *Virtual Write Queue* function is shown in Figure 7.3. In this example, the *Physical Write Queue* initially contains four cachelines which map to a target rank (cachelines 0 to 3 in the first column). At t_0, the scheduler initiates an eight cacheline write burst. While the initial four cachelines of data are available in the write buffer, the remaining lines must be transferred from the last-level cache using the *Scheduled Writeback* mechanism. In this case, a request is made at t_1 to the Cleaner logic coincident with the initiation of the writes to memory. To maintain back-to-back transfers on the DRAM bus, the Cache Cleaner must be able to provide data within the delay of the transfer of all of the data blocks in the *Physical Write Queue* for the given rank to main

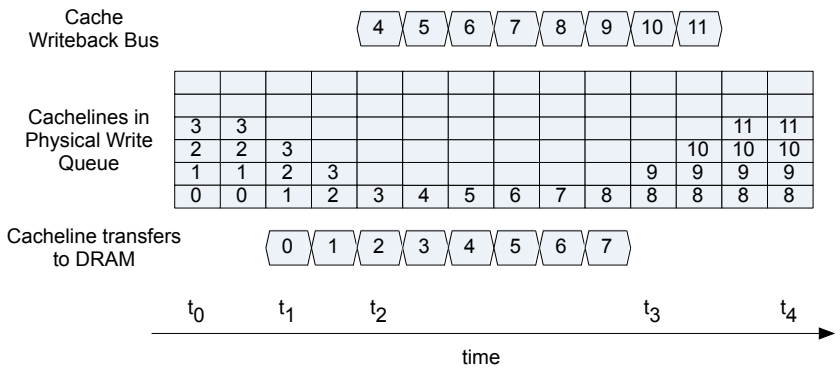


Figure 7.3: Virtual Write Queue timing diagram of operation

memory. Note, the Cleaner latency essentially determines the required number of cachelines required in the PWQ to maintain a burst transfer. For example, in the system using DDR3 1066 memory with a burst length (BL) of 8, each cacheline requires ≈8ns to be transferred (*i.e.*, ≈32 CPU cycles). Thus the Cleaner must be able to provide a cache line within 32ns, assuming four cachelines stored in the *Physical Write Queue*. The design analysis shows that this is easily achieved for typical last-level cache latencies (10 ns measured on the Intel I7 965 [14]). In the example of Figure 7.3, it is shown that the first writeback data, cacheline 4, arrives at time t_2. At this point, the *Physical Write Queue* has been depleted of lines 0-3, and data is streamed from the last-level cache. As the eight-line write burst completes at time t_3, the remaining lines from the last-level cache transfer are used to refill the *Physical Write Queue*. At time t_4, the physical queue is once again full, and ready to execute another write burst.

7.3 The Cache Cleaner

To qualify as an efficient implementation, the scheme must (1) not interfere with the mainline cache controller; (2) be power-efficient; and (3) be timely. Specifically, the *Cache Cleaner* (shown in Figure 7.2) must not affect hit rates or cause excessive access to cache directory and data arrays; it must avoid excess reads of cache directory arrays for power efficiency reasons; and cache lines to be cleaned must be located in a timely manner. The Cleaner uses a Set State Vector to accomplish these goals.

7.4 The Set State Vector (SSV)

While the cache lines in the *Virtual Write Queue* are contained within the state information of the cache directory itself, direct access is problematic. Specifically, cache directory structures are optimized for efficient CPU-side interaction. As such, directories are organized such that the cache tags, state bits, and LRU array are read together in a cache access.

This CPU-centric directory organization conflicts with the access requirements of the Cache Cleaner. The Cache Cleaner would like to efficiently search across many sets, in search of *dirty* cache lines to be scheduled for writeback to the DRAM. This efficient search is enabled with the addition of the *Set State Vector* (SSV). As shown in Figure 7.4, each entry in the SSV represents the set state from the perspective of the *Virtual Write Queue*. The SSV is stored in a dense SRAM, where a single read access contains information across many sets. Each SSV entry

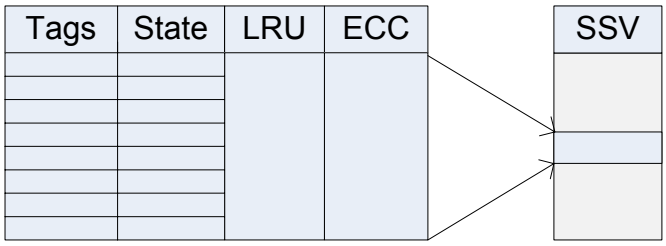


Figure 7.4: Set State Vectors (SSVs): directory set map to SSV entries

contains the *dirty data criticality* of each set. For the system, sets are defined with *dirty* data in the oldest $1/4$ of the cache as critical. Evaluation of multiple critical indications in the SSV (2bit: empty, low, medium, high) but found a single bit to be adequate (future work).

7.5 Cleaner SSV Traversal

Adjacent entries in the SSV are not necessarily adjacent sets in the cache. The dense packing is based upon the mapping of the system address into the physical mapping on the DRAM's channel/rank/bank resources. Adjacent entries in the SSV all map to the same channel/rank/bank resource.

A mapping example is shown in Figure 7.5. In this example a *closed-*

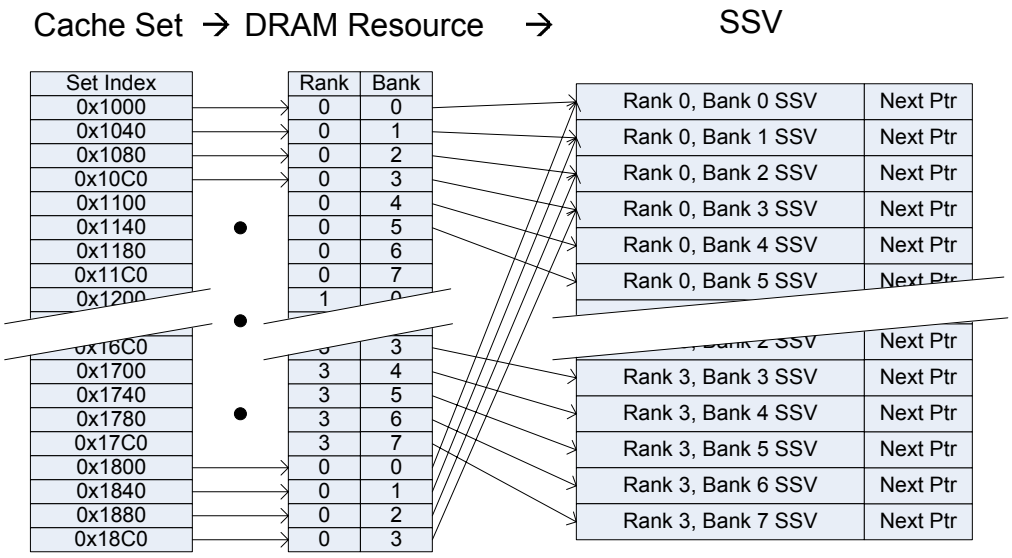


Figure 7.5: Set State Vectors (SSVs): mapping of cache sets to SSVs

page mapping for four ranks, each with 8 banks is used. In this case, every 32nd cache line maps to the same SSV. In general the mapping logic must be configured to match the DRAM mapping function; this is not a significant constraint, since mappings are known at system boot time. The scheme requires all bits that feed the DRAM mapping function to be contained within the set selection bits of the cache. This enables not only the SSV mapping function, but also *page mode harvesting*. This restriction does not produce significant negative effects, since all bits above the system page size are effectively random, and large last-level cache sizes have several higher order bits available for more sophisticated mapping functions (which avoid power of two conflicts that are common in simple lower-order bit mappings) [62].

The SSV is then subdivided into regions for each channel/rank/bank configured in the system. The Cache Cleaner maintains a working pointer for each of these configured regions. As the Cache Cleaner receives writeback requests from the memory controller, the associated working pointer reads a section of the SSV (with the matching *Next Ptr* in Figure 7.5). A set is selected, which will determine the specific set with which a writeback request will be generated. This request is sent to the cache controller to initiate the actual cache cleaning operation.

7.6 Write Page Mode Harvester Logic

The cache eviction mechanism is augmented to query adjacent lines in the directory, such that groups of requests within the same memory page can be detected and sent as a group for batch execution to the DRAM. When a line is pulled out of the cache array, the associated sets of the cache that contain possible page mode addresses are searched. If the corresponding page mode addresses are found, these will be sent as a group to the memory controller, to be processed as a burst page-write command sequence. In the evaluation it was found that three look-ups associated with a block of four cachelines provided significant gains without excessive directory queries.

While the design was evaluated used a static search criteria, dynamic policies are possible. Since the static proposal always searches three locations for page hits, power is wasted when page hits are infrequent. For workloads with greater opportunities, three lookups may not exploit all of the page hits that are possible. The primary difficulty in a dynamic proposal is detecting when to attempt

larger searches. It is certainly known when a search was unsuccessful, but knowing in which situations to search a larger region is more difficult. A sampled approach may be effective. This area could be expanded in future work.

7.7 Prevention of Extra Memory Writebacks

Since the system speculatively writes dirty data back to memory, there is some chance that extra memory write traffic is introduced. Specifically, if a store occurs to the data after it is cleaned, the cleaning operation is wasted. As a solution to this problem lines, state information is added to each line in the cache to indicate the line has been cleaned. If a cleaned line receives a store, this indicates an extra write was produced. As the line is listed as previously cleaned, future cleans to the line are inhibited.

In an effort to reduce storage overhead to track this marker state, additional cache states are added to the unused state encodes of MOESI to indicate a line was once dirty, but has been cleaned. Lines in *Cleaned* states are excluded from being cleaned a second time. A complete extension to the MOESI protocol would require *Cleaned* version of all four valid states. This presents additional overhead in that the total number of states would reach nine. Since MOESI systems require three state bits of encoding, three unused state encodes are available. To avoid the overhead of adding a fourth state bit, the *Shared Cleaned* state was excluded, maintaining the same state overhead as standard MOESI. The justification for the exclusion of *Shared Cleaned* is best explained through the state transitions shown in Table 7.1. In the table, the *Cleaned* states are represented with a lowercase *c*, e.g. *Modified*

Cleaned as M_c.

Table 7.1: Extra coherence protocol transitions introduced to prevent extra memory writebacks

#	Initial State	Event	Next State	Comment
1	M	Clean	E_c	Scheduled Writeback
2	S	Store	M	M If no O_c in system
3	S	Store	M_c	M_c If O_c in system
4	M_c	Snooped Read	O_c	
5	O_c	Store	M_c	
6	O_c	Snooped Read	O_c	
7	E_c	Store	M_c	
8	O	Clean	S	Disallowed due to lack of S_c
9	E_c	Snooped Read	S	Loss of cleaned Information

There are two cases of potential transitions into the *Shared Cleaned* state (S_c). In row 8 of Table 7.1, the case of an *Owned* (O) line that if cleaned would transition to *Shared Cleaned* is shown. In the simulations *Owned* lines are not cleaned thus this case is avoided. In row 9 the case of a line in *Exclusive Cleaned* state (E_c) where a read operation is snooped is shown. Here the *Exclusive Cleaned* line must transition to an *Shared* state. Since the *Cleaned* modifier is not required for coherence, the traditional *Shared* state is used in this case. In the analysis no degradation was observed due to this policy.

7.7.1 Overall Overhead Analysis of Virtual Write Queue

The actual storage overhead of the proposed *Virtual Write Queue* is limited to the overhead for the SSVs. The rest of the scheme primarily reuses existing

structures in the last-level cache and memory controller of a typical CMP system. All of the remaining structures added to the last-level cache controller and memory controller are negligible in size compared to the storage required for the SSVs. The overhead of the SSV can be evaluated by comparing it to the cache directory. For a 16MB 8-way associative cache used in the analysis, each cache set requires 346 bits (see Figure 7.4: $8 * 32$bits Tag bits, $8 * 3$bits State Bits, 28 LRU bits and 38 ECC bits). Since the proposal adds only one SSV bit per cache set, the overhead is approximately $1/346 \approx 0.3\%$ (4Kbytes of storage for the SSV compared to 1384Kbytes for the cache directory).

7.8 Results

To evaluate the effectiveness of the proposed *Virtual Write Queue*, Simics [38] extended with the GEMS toolset [39] was used to simulate cycle-accurate out-of-order processors and a detailed memory subsystem. The toolset was configured to simulate an 8-core SPARCv9 CMP with 8GB of main memory. The memory subsystem model includes an inter-core last-level cache network that uses a directory-based MOESI cache coherence protocol along with a detailed memory controller. Both the last-level cache and the memory controller were augmented to support a baseline memory controller and the proposed *Virtual Write Queue*. The baseline implementation simulates a *First-Ready, First-Come-First-Served* (FR_FCFS) [51] memory controller with the addition of *Eager writebacks* [32], referred to as *FR_FCFS+EW* in the evaluation. The SPEC CPU2006 Rate scientific benchmark suite [12] was used, compiled to the SPARC ISA with full optimizations

Table 7.2: Core and memory-subsystem parameters used for cycle-accurate simulations

Core Characteristics	**Clock Frequency**	**Pipeline**	**Reorder Buffer /Scheduler**	**Branch Predictor**
	4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries	Direct YAGS / indirect 256 entries
Memory Subsystem	**L1 Data & Inst. Cache**	**L2 Cache**	**Outstanding Requests**	**Memory Bandwidth**
	64 KB, 2-way associative, 3 cycles access time, 64 Bytes block size, LRU	16 MB, 8 ways associative, 12 cycles bank access, 64 Bytes block size, LRU	16 Requests per Core	16.6 GB/s
	Controller Organization	**DRAM**	**Controller Resources**	**Virtual Write Queue**
	2 Memory Controllers 1, 2, and 4 Ranks per Controller 8 DRAM chips per Rank	8GB DDR3-1066 7-7-7	32 Read Queue & 32 Write Queue Entries	2 LRU ways 4096 High & 4064 Low Watermark

(peak flags). Each benchmark is fast-forwarded for 4 billion instructions to reach its execution steady-state phase. The next 100M instructions are then simulated to warm up the last-level cache and the memory controller structures; followed by a final 100M instructions used in the evaluation. Table 7.2 includes the basic system parameters.

7.8.1 System Throughput Speedup Analysis

Cycle-accurate simulations of SPEC CPU2006 Rate showed that the *Virtual Write Queue* enables significant throughput gains for workloads with high memory bandwidth requirements. These gains over the baseline FR_FCFS+EW [51][32] system is shown in Figure 7.6. Speedups for three memory bus configurations (1, 2,

and 4 DRAM ranks per channel) are shown. SPEC workloads not listed in the figure did not show any measurable change in performance. As expected, workloads with high memory utilization showed benefits due to reduction in bus penalties by forming long back-to-back write bursts. The largest speedup is observed on the single rank system. In this case, the controller does not have other ranks to which to send requests, and the "write-to-read-same-rank" penalty is incurred at every bus turnaround. For the 2-rank system, smaller gains are shown, since the baseline system is able to schedule around "write-to-read-same-rank" penalties in many cases. In that case, delays due to rank-to-rank transitions become more important. The performance of the 4-rank system is very close to that of the 2-rank since for this case the controller incurs fewer "write-to-read-same-rank" penalties, but generates more frequent rank-to-rank transitions. Overall, the *Virtual Write Queue* achieved average improvements of 10.9% in throughput when configured with 1 rank, while for the cases of 2 and 4 ranks the IPC improvements were found to be 6.4% and 6.7%, respectively.

7.8.2 Page Mode Analysis

Using the *Virtual Write Queue* significant increases the amount of page mode write operations executed over the baseline FR_FCFS+EW [51][32] system. The full-system simulation shows an average of 3.2 write accesses per page – this contributes to throughput system gains (as observed in Section 5.1) and memory power reduction. While the throughout gains are only observed in high bandwidth workloads, the power reductions are more universal. In Figure 7.7 the memory

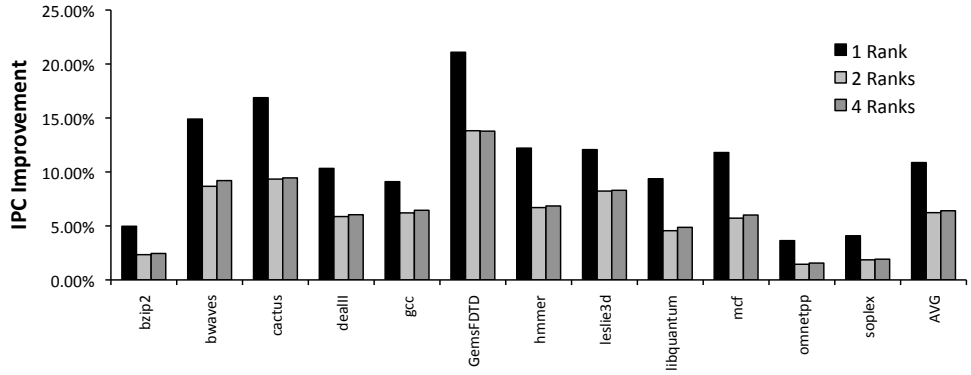


Figure 7.6: IPC improvements of *Virtual Write Queue* over prior work (FR_FCFS + Eager) [51][32]

power reduction for each workload is estimated using the Micron power estimator [42]. Overall, an average DRAM power reduction of 8.7% is observed. As shown in [50], main memory can be a significant portion of total system power. For *stream*, Rajamani, *et al.* indicate that memory power can be 48% of high performance system power, even after accounting for supply losses. The 11-15% power saved for half of the workloads through the page mode write scheduling of *Virtual Write Queue* would thus result in a 5-7% system-level savings.

To further evaluate the page mode behavior of the *Virtual Write Queue* a set of simulations were performed using three diverse commercial workloads. Due to the complexity and size of the commercial workloads, a detailed evaluation using a cycle-accurate, full system simulator is prohibitively expensive. As a solution, for commercial workloads trace-based cache simulations were utilized. The first workload is an *On-Line Transaction Processing* (OLTP) workload driven by hundreds of simulated individual clients generating remote transactions against

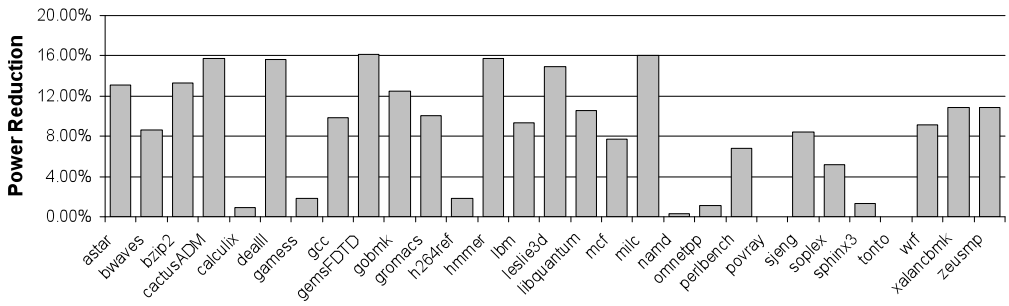


Figure 7.7: DRAM power reductions achieved by *Virtual Write Queue* for SPEC CPU2006 Rate

a large database. The second workload represents a typical *Enterprise Resource Planning* (ERP) workload. As with the OLTP workload, the ERP workload was driven by simulated users who sent remote queries and updates to the database. Finally, the third workload is SPECjbb2005 benchmark [12] that targets the performance of a three-tier client/server system with emphasis on the middle tier. Trace-based simulations were performed using an in-house cache simulator augmented to model the *Virtual Write Queue*.

The cache simulator was augmented to monitor the total page mode writes that were possible when varying the number of cache ways that were allocated to the *Virtual Write Queue*. The simulation results are shown in Figure 7.8. Whenever a dirty line was evicted from the cache, the last N ways of the cache were checked for other dirty lines that mapped to the same DRAM page. As expected, there is a steady increase in page mode opportunity as the number of ways considered is increased, with significant increases in 2 ways, and diminishing returns after considering more than 4 ways, or half the cache. The increased performance

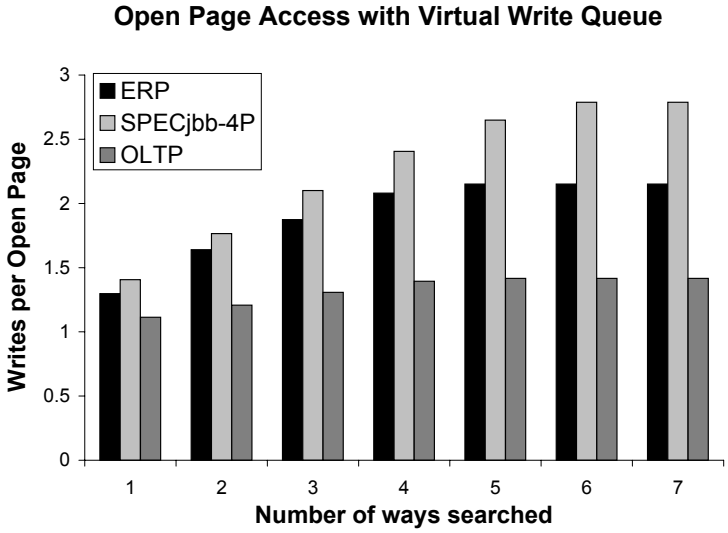


Figure 7.8: The *Virtual Write Queue* exploits the LRU ways of the last-level cache to virtually expand the write queue.

from considering more ways of the set must be balanced against the overhead of additional writes introduced by the cleaning. Though not shown, it is worth noting the implied difference in performance results across the three workloads. The OLTP workload sees limited benefit, while the ERP and SPECjbb see much larger opportunities.

7.8.3 Prevention of Extra Memory Writeback Analysis

As described in Section 7.7, the *Virtual Write Queue* mechanisms has the potential to generate additional writeback traffic for cases where a cleaned line is modified after its speculative writeback to the memory. In this section the magnitude of the problem and the avoidance mechanism presented in Section 7.7

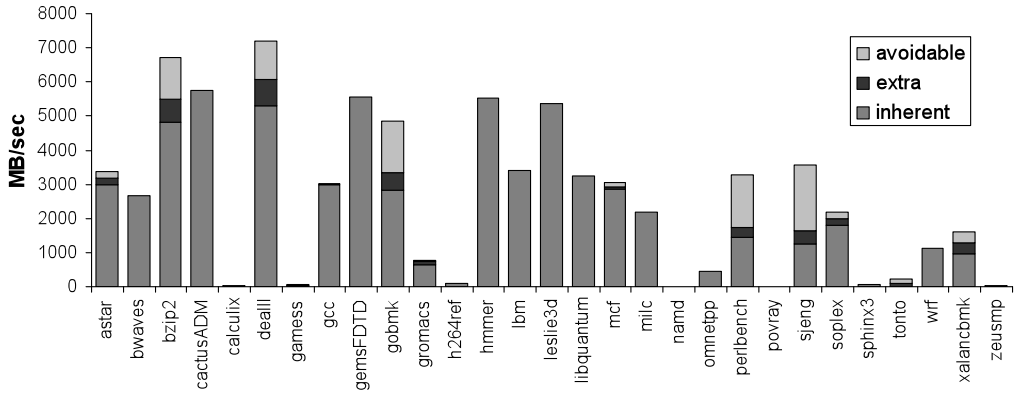


Figure 7.9: Extra Writeback avoidance for SPEC CPU2006

are evaluated. Through cycle-accurate simulations, the writebacks to memory are classified into the following categories: a) *Inherent*: normal bandwidth that is not created by speculative writebacks, b) *Extra*: bandwidth created through speculative writeback that is *not* removed though the cache state enhancements, and c) *Avoidable*: Speculative bandwidth that is avoided with the addition of the proposed cache state enhancements. In Figure 7.9 the write memory bandwidth is shown, in MB/sec, as was collected from the simulations. As shown, certain workloads, such as `sjeng`, see significant extra bandwidth that is eliminated using the cache state enhancements of Section 7.7. Note, the power estimates in Section 7.8.2 assume these cache state enhancements. In all cases, the power savings achieved at the DRAM using page mode offset the increase from extra traffic (since most of the extra traffic contains no bank activates). In addition, a limited evaluation of commercial workloads was performed to gauge how problematic this behavior is. Again, there are significant differences between the workloads. Using the

lower 2-ways of LRU in the *Virtual Write Queue*, SPECjbb showed 1% increase in writebacks compared to an increase of 6% for ERP and 9% for OLTP.

7.8.4 Summary

This chapter proposes a novel approach towards *coordinating last-level cache and DRAM policies* by expanding the memory controller's visibility and providing awareness of the physical main memory layout. The work demonstrates that the *Virtual Write Queue* scheme is able to achieve significant raw system throughput improvements (10.9%) and power consumption reductions (8.7%) with very low hardware overhead (≈0.3%). Overall, the *Virtual Write Queue* demonstrates that co-optimizations of multiple system components enables low-cost, high-yield improvements over traditional, isolated to each resource, designing approaches.

Chapter 8

Elastic Refresh

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 chapter 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. The overall scheme is titled 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.

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 [22]. In this work the term *postponed* is used 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 chapter.

8.1 Baseline Refresh Scheduling

Figure 8.1 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. The analysis in Section 8.6.1 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.

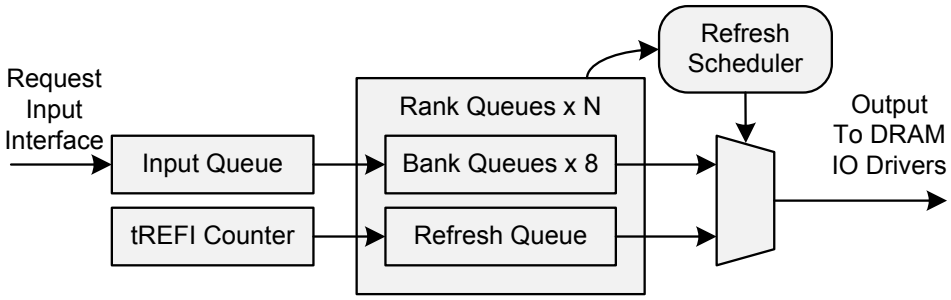


Figure 8.1: Refresh Control Logic

8.2 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, current policies are examined, referring to the memory controller logic which decides when to issue `refresh` commands as the *refresh scheduler* (shown in Figure 8.1).

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 [63] and GEMS [39]. In addition, even work dealing in sophisticated operation schedulers have employed this method [17]. This work refers to this common policy as *Demand Refresh* (DR).

In a more sophisticated policy that exploits the ability to postpone refresh commands [60], `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. This policy is referred to as *Defer Until Empty* (DUE).

8.3 Examples of Where Typical Approaches Break Down

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

8.3.1 Low Memory-Level-Parallelism Workloads

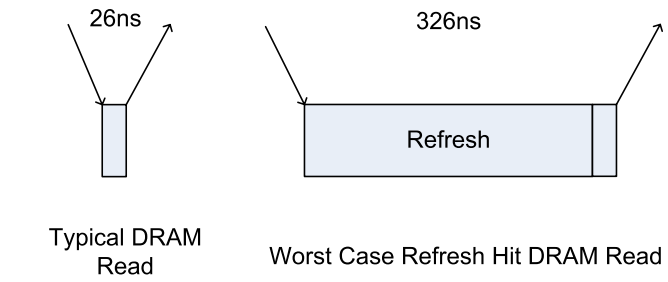
Traditional approaches behave poorly while running low-memory-level-parallelism (low-MLP) workloads. In low-MLP workloads, memory utilization is often quite light, but each reference to memory is critical to the workload's execution progress. A classic example of such an algorithm/workload is the traversal 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.

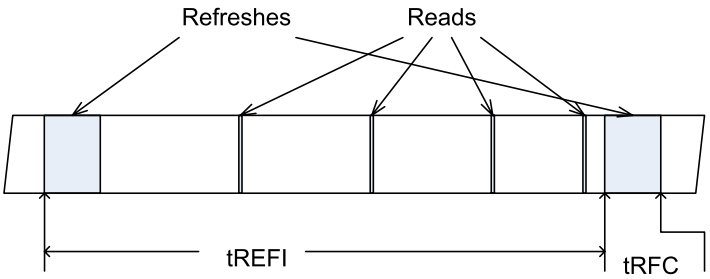
The magnitude of this effect is significantly larger than expected when only considering the fraction of time the DRAM is executing the refresh. In Figure 8.2(a), the magnitude of the delay experienced by a read received just after a 300ns refresh operation of 4Gbit DRAM is graphical shown, compared to the typical closed page access latency of 26ns to accomplish a typical read operation. In Figure 8.2(b), the fraction of time the DRAM bus is executing refresh operations over a tREFI interval is shown. 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 8.2 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 8.2, 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



(a)



(b)

Figure 8.2: Refresh Latency Penalty Example

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 8.2 illustrates, this latency penalty can be very significant. For example, a modern processor might achieve a baseline memory latency of ∼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 [56]. 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.

Table 8.1: 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

8.3.2 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. The analysis indicates that traditional refresh deferral solutions reach significant backlogs only in 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.

8.4 Refresh Beyond DDRx SDRAM

In addition to the emerging `tRFC` penalties 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. Examples include Phase-change-Memory (PCM), Resistive-random-access-memory (RRAM), and Spin-transfer-random-access-memory (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 ($\leq 95°C$) [28]. 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.

Also shown in Table 8.2 are the first order refresh overhead penalties across DRAM types. The 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 10% tax in the densest ddr3 technology. Potentially more invasive is the latency overhead. This best case latency overhead 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 in general the scheduler would delay a refresh if a read operation was queued, this represents the case where the read is unlucky. In this case, the latency penalty is on average half the tRFC time. The rate at which this latency event occurs matches the bandwidth overhead calculation. As shown, this latency penalty can be very

significant. For example, a modern processor achieves an unloaded latency of $\tilde{5}$0ns. For 8Gb DRAM, the penalty of 24.7ns represents a 50% latency increase due to refresh.

Table 8.2: Refresh penalty as density increases

DRAM type	tRFC	bw overhead (85c)	latency overhead (85c)	bw overhead (95c)	latency overhead (95c)
256Mb	75ns	1.0%	0.4ns	1.9%	0.7ns
512Mb	105ns	1.3%	0.7ns	2.7%	1.4ns
1Gb	127.5ns	1.6%	1.0ns	3.3%	2.1ns
2Gb	195ns	2.5%	2.4ns	5.0%	4.9ns
4Gb(est)	292.5ns	3.8%	5.5ns	7.5%	11.0ns
8Gb(est)	438.75ns	5.6%	12.3ns	11.3%	24.7ns

8.5 Elastic Refresh Scheduling

The behavior observed in current refresh scheduling algorithms is addressed 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, predictive mechanisms are used to decrease the probability of a `read` or `write`'s collision with a recently issued `refresh` operation.

The *Elastic Refresh* algorithm proposed 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 8.1). In the proposed method, the `refresh` command is delayed an additional period of time for the rank to be idle. 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`. This idea is extended 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.

8.5.1 Idle Delay Function

The idle delay is expressed 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 8.3. Note, in this proposal, the parameters of the IDF are dynamically adjusted based on the workload characteristics. Three regions of delay characteristics are defined:

1. *Constant*: Analysis of simulations showed that 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.

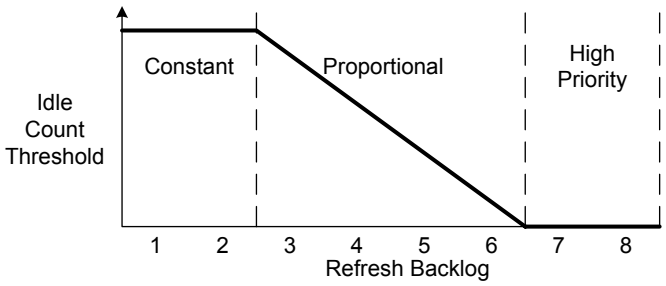


Figure 8.3: Idle Delay Function (IDF)

2. *Proportional*: This region represents the area where the postponed refresh count approaches the maximum allowed value where the scheduler transitions 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 count of eight, the `refresh` will be sent before any other commands, as soon as the DRAM bus parameters allow.

8.5.2 Idle Delay Function Control

As the optimal characteristics of the idle delay function are workload-dependent, a set of parameters is defined to configure the delay equation. These are listed in Table 8.3. The Max Delay and Proportional Slope parameters are

Table 8.3: 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

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.

8.5.2.1 Max Delay Control

Analysis 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, the average delay of all idle periods was found to be an effective value across a range of workloads. As such, a circuit was devised 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

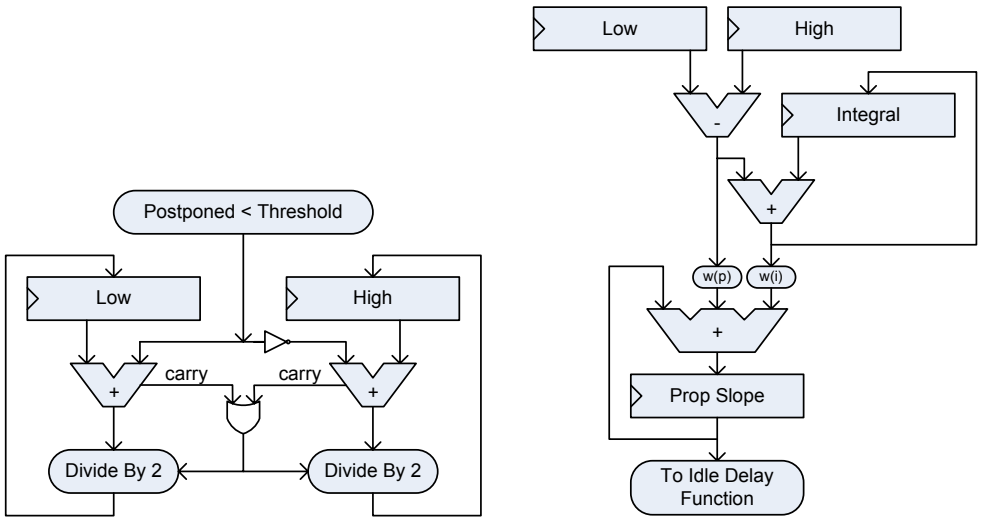
the accumulator. The average is calculated every $2^{10} = 1024$ idle intervals with a simple shift-left of 10 bits. If the accumulator overflows, a maximum average value of 1024 is used.

8.5.2.2 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. A postponed count threshold of four was used in this system, reflecting the midpoint of the deferral range.

The hardware structure to implement this function is shown in Figure 8.4. 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 8.4(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 High and Low counters. The value is applied to a Proportional Integral (PI) (shown in Figure 8.4(b)) control circuit to update the Proportional Slope parameter for the subsequent interval. Not shown in Figure 8.4 is the reset of the High and Low counters after each adjustment interval.

For this analysis, the following parameters were used, which were deter-



(a) Low and High counter update logic (b) Proportional Slope control circuit

Figure 8.4: Proportional Slope Control Circuit

mined 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).

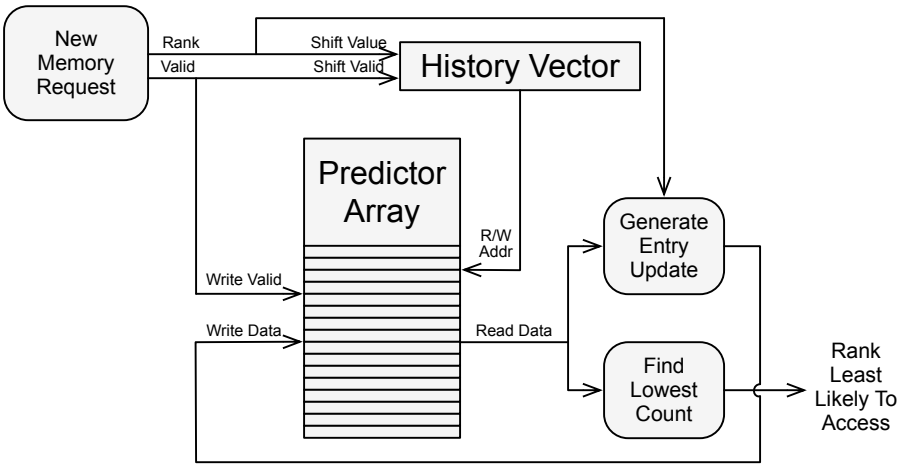
8.5.3 Cache Read Miss Prediction

The accuracy with which `refresh` operations are scheduled to avoid memory `read` operations can be enhanced with the usage of history-based prediction structures. A predictor can be used to detect situations where the probability of a request to a given rank is low. This prediction can be used to adjust the idle count to a lower value, biasing a `refresh` to that rank.

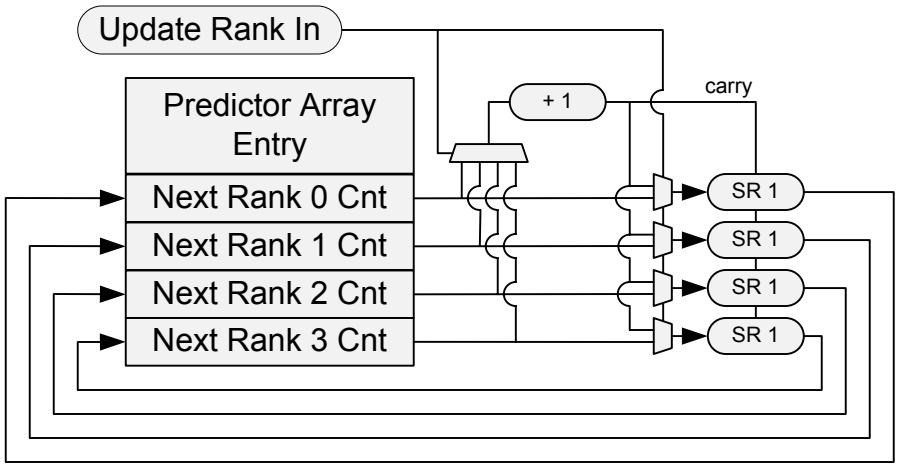
A per-core history based-approach was found to be effective. For each CPU on the CMP, a history is maintained for each of the memory ranks accessed on the previous N cache misses. These history vectors are used to index a prediction array. Each entry in the prediction array contains a relative count amongst the memory ranks, which is used to predict the least likely rank to be accessed next by a given core. A history vector per CPU was utilized to capture the behavior of an executing thread. If each CPU executed multiple threads concurrently, a history vector for each thread within the CPU is suggested.

8.5.3.1 Prediction Structure

The primary structures and logical flow of the predictor are shown in Figure 8.5(a): the History Vector, Predictor Array, and the surrounding combinatorial logic. Figure 8.5(b) shows the contents of each entry of the Predictor Array. Prediction of the next rank to be accessed is based on the "path" of rank accesses leading up to an access. For each path, the history of which rank was accessed next is stored in a predictor array. This predictor array is indexed with the history vector.



(a)



(b)

Figure 8.5: Rank Access Prediction

8.5.3.2 Next Rank Probability

Logically, the predictor maintains a Next Rank access count for each inbound path. Considering each History Vector value as a node in a Markov model, the probability of each outbound link is estimated using the recent history of prior traversals. To model this behavior, a counter for each outbound link (Next Rank Cnt in Figure 8.5(b)) is utilized. Whenever a new request is seen, the Next Rank Cnt for the rank of the new request is updated in the node referenced by the current History Vector. As the width of the Next Rank counters drive the storage requirements of the predictor, the design uses counters as small as possible. With these small counters, the rollover behavior of each counter is critical to the design. Elastic Refresh employs the following mechanism: as any counter generates a carry (*i.e.*, would typically wrap back to zero), all counters are divided by 2 (shift-right by 1). To select the rank with the lowest probability, the combinatorial block "Find Lowest Count" in Figure 8.5(a) determines the lowest counter value (in a tie, an arbitrary rank is chosen).

8.5.3.3 Predictor Capacity

The width of the History Vector has a direct effect on the size of the predictor array. This size equates to,

$$Number\_Ranks * Next\_Rank\_Cnt\_Width * 2^{History\_Length * log_2(Number\_Ranks)}$$

A predictor table with a modest history of four requests would require a 2^8 = 256 entry table. With 4-bit next-rank counters (representing the four ranks in the baseline system), each entry is 16 bits in size, giving a total of 4096 bits of storage.

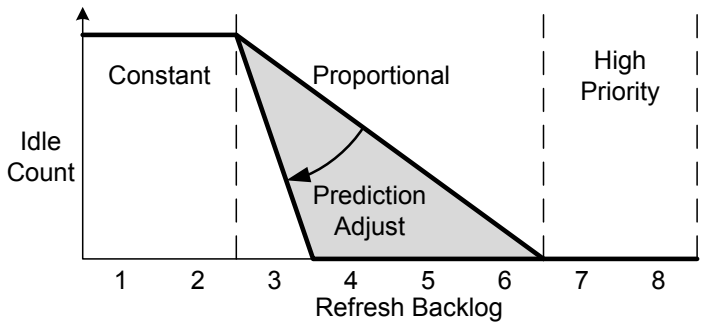


Figure 8.6: Idle Delay Function Enhanced With Explicit Prediction

Section 8.6.2 contains details on the accuracy of the predictor relative to various geometries.

8.5.3.4 Predictor Result Integration into Rank Idle Delays

This mechanism integrates the next rank predictor by decreasing the required idle delay for ranks which are not expected to be accessed next. The idle delay function modification is shown in Figure 8.6. This is estimated through two criteria. First, the rank with the lowest transition probability is selected. Secondly, the transition probability must be below a threshold which was determined experimentally.

8.5.4 Elastic Refresh Queue Overhead

Table 8.4 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 (FD) and the additional hardware to dynamically tune

Table 8.4: 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 8.4 to exploit full deferral capability
Predictive Delay	History-based next rank prediction	Prediction confidence

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 counter. 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.

8.6 Evaluation

8.6.1 Simulation Configuration

To evaluate the proposed Elastic Refresh policies, the GEMS toolset [39] built on top of the Simics [38] functional simulator was utilized. GEMS provides

a cycle-based out-of-order processor model. This was enhanced 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) [51] memory controller that supports two separate baseline refresh policies: a) *Demand Refresh* (DR) and b) *Defer Until Empty* (DUE) (see Section 8.2) along with the proposed Elastic Refresh policies. Table 8.5 includes the basic system parameters.

For the memory refresh parameters, a configuration representing what `tRFC` could be in the 16Gbit DRAM time-frame was evaluated. 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 5.3.2). Based on this, a value of 550ns for `tRFC` was selected. For `tREFI`, the 95°C interval of 3.9μs was utilized, as this reflects usage in dense server environments, where CMP systems and large memory configurations are common [16, 43].

The SPEC CPU2006 benchmark suite [12] 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

Table 8.5: 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

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 in the homogeneous SPEC Rate benchmarks. In any case, the total IPC across all cores was measured in the interval in which core 0 executed 100M instructions.

8.6.2 Cache Read Miss Rank Predictor

The predictor structure outlined in Section 8.5.3 was evaluated across a range of possible configurations to determine the ideal geometry. The cache miss

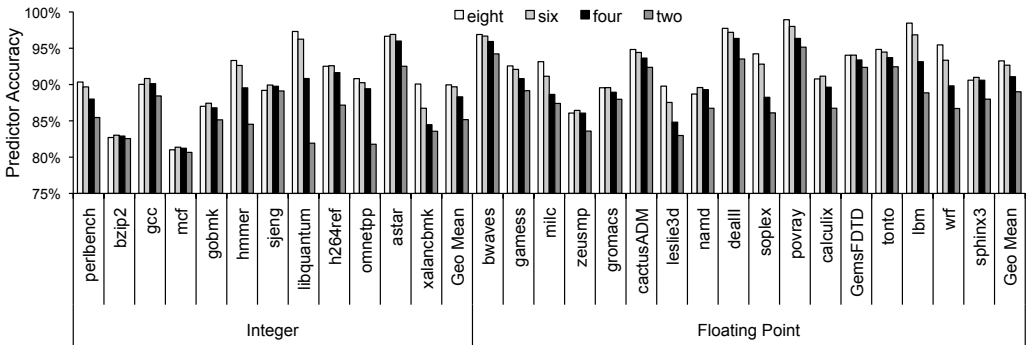


Figure 8.7: Prediction rate for a range of history vector sizes.

reference stream produced through single core executions of the SPEC CPU 2006 benchmark suite was used. This stand-alone evaluation enabled targeted evaluation of the predictor accuracy across many configuration options. For this analysis a four-rank system was simulated. The prediction accuracy is defined as the fraction of time when the actual next rank requested is different than the predicted rank. As such, prediction results are best compared against the 75% result achieved with a random guess. With this analysis a reasonable predictor structure was found, which was then integrated into the refresh scheduler used in the detailed simulations. The following parameters were explored in the analysis:

1. **History Vector Length**: Simulations for History Vector lengths of the previous eight, six, four, and two requests were run. The prediction accuracy is shown in Figure 8.7. In general, this shows that longer histories are beneficial, but this must be balanced with the reality that each additional history bit grows the predictor array by a factor of two.

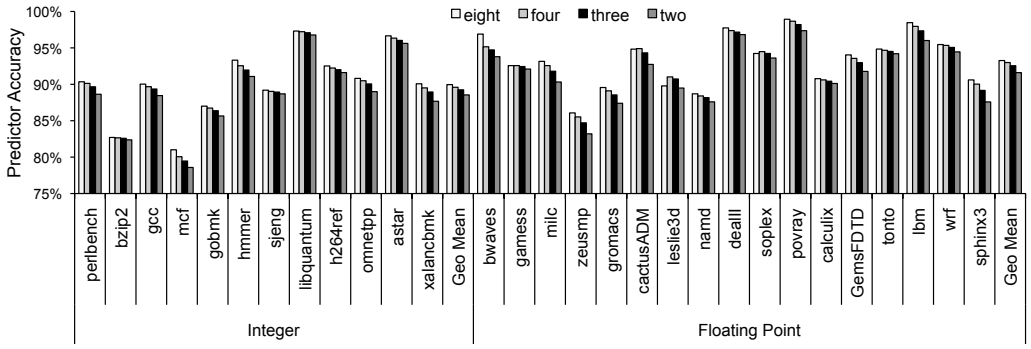


Figure 8.8: Prediction rate for a range of predictor entry counter sizes.

2. **Next Rank Probability Counter**: Simulations for the Next Rank counter sizes of 8, 4, 3, and 2 bits were also run. The results are shown in Figure 8.8. These results show relatively gradual degradation across the range of sizes.

3. **Predictor Array Size Vs. Accuracy**: To cover the range of predictor options, all the combinations of History Vector and Next Rank Counter sizes listed above were simulated. Figure 8.9 contains a plot of the average accuracy for the integer and floating point benchmark suites, sorted with respect to the overall number of bits contained in the prediction array. A number of sizes that would be prohibitive to implement in hardware were included, in order to gage the limits of the overall scheme. Note that the "Bits" data series increases beyond the scale of the plot. Based on this analysis a predictor with a History Vector of four request, and 4-bit Next Rank counters was selected for the detailed analysis. This represents a 4k bit structure, which is reasonable given the potential performance improvements.

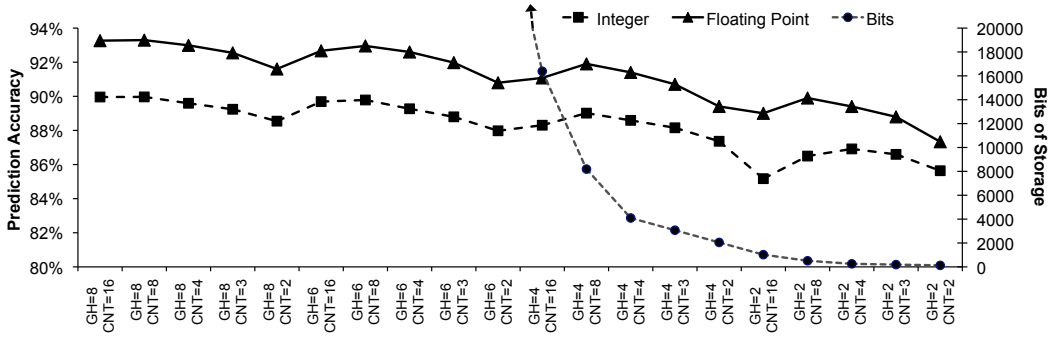


Figure 8.9: Prediction rate for a range of history vector sizes.

8.6.3 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 [12] results are shown in Figure 8.10; four core SPEC Rate [12] results in Figure 8.11; and eight core SPEC Rate results in Figure 8.12. In general, the most significant throughput gains are observed 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 indicates that the refresh problem is more tied to latency penalties rather than simply bandwidth overhead.

8.6.3.1 Fixed Delay Results

For the Fixed Delay runs static values were selected for each of the parameters that seemed to be effective for most workloads (an exhaustive search

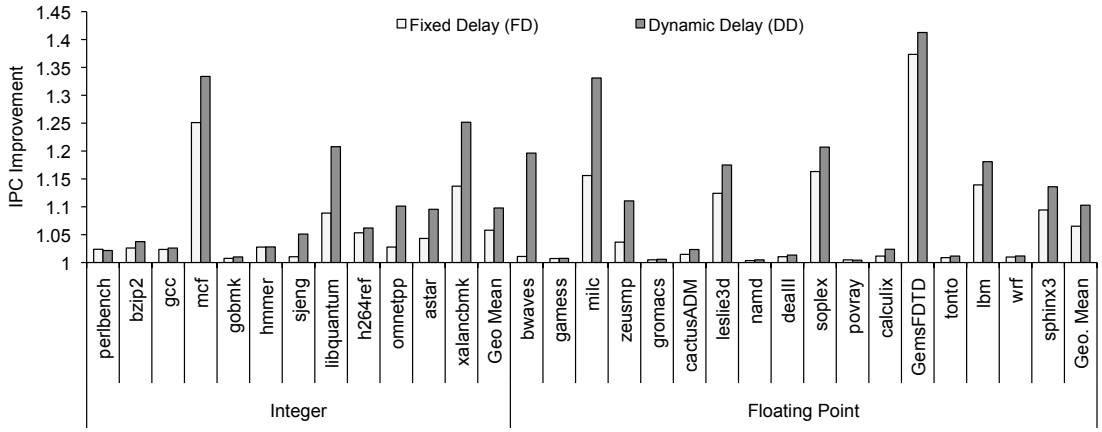


Figure 8.10: IPC improvement of proposed refresh policy techniques over baseline refresh policy on 1 core

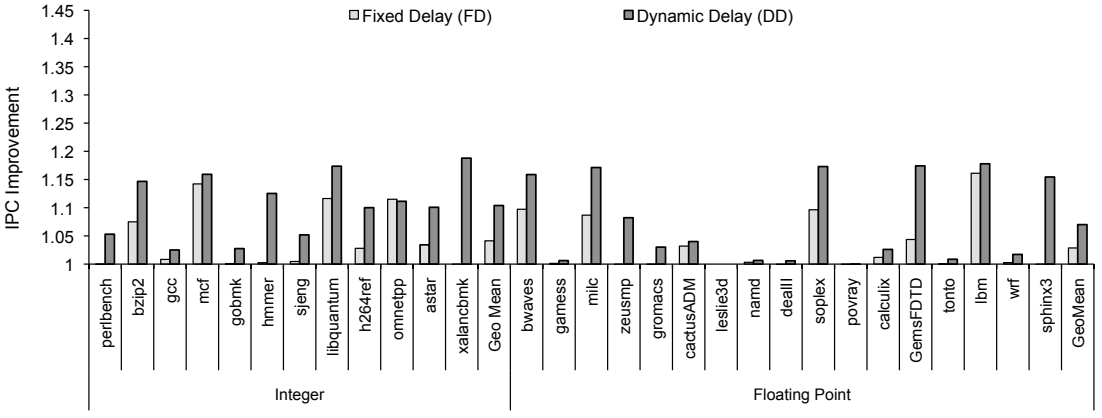


Figure 8.11: Relative IPC improvement of proposed refresh policy techniques over baseline refresh policy on 4 cores

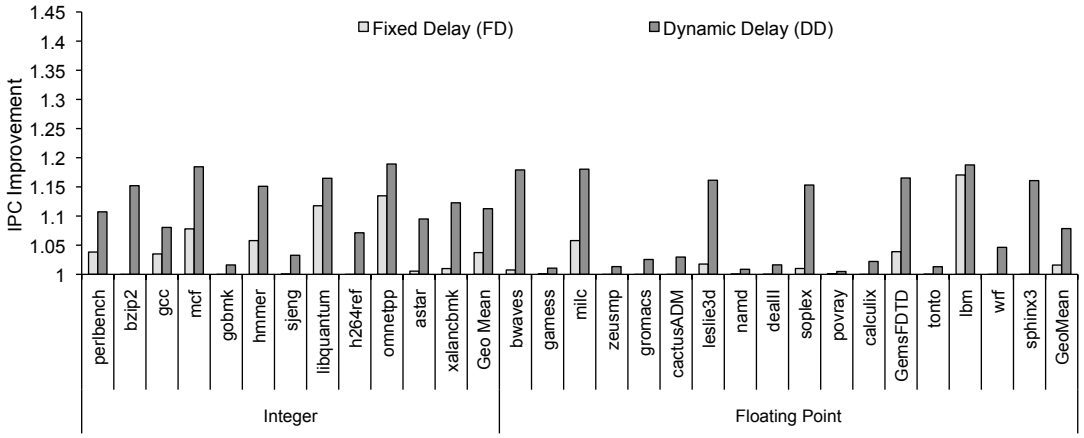


Figure 8.12: Relative IPC improvement of proposed refresh policy techniques over baseline refresh policy on 8 cores

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, performance improvements of Integer (5.9%, 4.1%, 3.7%) and Floating-Point (6.5%, 2.8%, 1.6%) across one, four, and eight CPUs were observed. 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.

8.6.3.2 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.

8.6.3.3 Prediction Results

The inclusion of the Explicit Miss predictor was most significant for a subset of the single core Speed [12] benchmark runs. The most notable improvements were seen on `libquantum`, `astar`, `bwaves`, `milc`, `soplex`, `GemsFDTD`, and `lbm` benchmarks. The largest improvement of these workloads was 16% observed on `libquantum`, with an average improvement across the suite of (12%,10%,10%) for one, four, and eight CPUs. The improvements were generally found on the memory intensive benchmarks, with the notable exceptions of `mcf` and `omnetpp`. To help in understanding the access patterns of these workloads, plots of the memory accesses over time were created. In this analysis, very irregular references for these two workloads were found. Tracing this back to the source code, the traversal of large tree like structures in `mcf` was found to be the culprit. In `omnetpp` the source of the irregular access was the usage of a priority heap, where timestamps of events

are stored and sorted as part of the event simulation (`omnetpp` is a network event simulation).

Initially, any throughput increases were not expected with usage of the Read Miss prediction circuit for multi-core environments. Intuition predicts that the superposition of the prediction for which rank each core will not visit next will result in no ranks that are not expected to have a reference. While this was true for most of the workloads, there were a handful of exceptions. One notable exception is the `libquantum` benchmark, a $\sim$13% throughput increase is observed for multi-core rates runs. In this workload, the program consumes a large (32 MB) vector sequentially. As such, the next rank prediction is $\sim$100% correct. The interesting behavior observed in the simulations is that the copies "sync" up in the traversal of memory. That is, all the CPUs to be generating cache misses to the same ranks in unison. This synchronization was due to the `refresh` commands themselves, as all of the CPUs would be stalled waiting for read data behind a `refresh` operation. When the `refresh` of the rank completed, all CPUs would be in the same location of the repeated rank traversal order. As all the CPUs maintain the same long lived traversal order, they would stay in sync, thus enabling an accurate prediction of which rank to refresh.

8.6.4 Summary

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.

Chapter 9

Conclusions

As many-core based computer systems demand higher performance memory systems, operational complexity is introduced. As this operational complexity is rooted in the fundamental behavior of DRAM devices, mechanisms to mitigate these complexities is of high importance. This work presents a set of low overhead system enhancement that are able to substantially improve memory performance in light of these system behaviors. In many senses, this work focuses on non-critical operations such that critical reads can be executed with lower latency.

9.1 Summary

This work describes three contributions towards mitigating DDRx DRAM complexities for many threaded systems. While the target of this research is standard DDRx SDRAM, the concepts can be extended into other memory devices and coordinated policies in general.

The concept of Minimalist Open-page policy is introduced. This counter-intuitive policy policy challenges the traditional viewpoint that page-mode should be exploited as much as possible [35]. The reduction in page-mode prevents row-buffer induced starvation identified by Moscibroda et. al [45], without more

complex priority schemes. This enables the priority schemes to serve other optimizations.

The Virtual Write Queue demonstrates cross unit optimizations within the now integrated components of CMP systems. The work targets efficient operations of writes through more optimal burst of write operations. This largely mitigates the effectively longer write to read device turn around penalty of higher frequency DDRx DRAM. In addition, page mode writes are made possible in shared writeback caches. These page mode writes increase utilization while decreasing energy consumption. These improvements are realized with very little hardware overhead.

This work also identifies the rapidly growing penalty of refresh in DRAM memory as device density increases. Elastic refresh scheduling is introduced, which greatly reduces the penalty of these refresh operations. As research in memory refresh scheduling is limited, this work represents a first step in attempting to scale DRAM to even higher densities.

9.2 Future Work

While the combined contributions of this work make significant steps in increased memory performance, several opportunities for future work exist. These are organized specific enhancements for each of the three major contribution areas, followed by more general future research directions.

9.2.1 Enhancements to the VWQ

Several design details of the Virtual Write Queue could potentially be extended. Specifically, a simple threshold based on VWQ fullness was used to switch to high priority writes. While this worked well for SPEC CPU 2006, other workloads could potentially benefit from more sophisticated write priority mechanisms. Along similar lines, the page mode harvester executed a simple search of the three adjacent potential page mode hits. A mechanism that abandoned searching for workloads without any hits could save directory power and bandwidth. Conversely, searches beyond three lines are possible. Note, the group of four hashing of the minimalist policy does not prevent greater searches, as the harvest logic can skip over the lower order bank bits. For long streams this could be effective.

Beyond these policy enhancements, future work could be address scaling the design beyond the single chip, shared cache evaluated in this work. Work could also investigate more complex cache hierarchies.

9.2.2 Refresh Enhancements

While the Basic Elastic policies provide very good gains for very low area overheads, the more complex prediction methods are less effective. More sophisticated prediction may be possible, which could enable greater gains. In addition, the work focuses primarily on low to medium bandwidth workloads. Mechanisms targeting higher bandwidth workloads are possible. In addition, work investigating the limits of density based around refresh penalties could be useful.

At these limits, changes to the basic mechanism may be more palatable.

9.2.3 Future Memory Devices

As memory beyond DDR3 is released, new bottlenecks are inevitable. Evaluation of the DDR4 standard should be an area of focus. Interestingly, this work would suggest smaller DRAM row-buffers should be utilized. The power saving from a future DRAM device with this attribute would be a useful extension of this work.

In addition to DDRx memory, other memory technologies such as Phase-change memory (PCM) are becoming more highly studied. An extension of this work directed towards non-DDRx SDRAM memory could provide important insights.

Bibliography

[1] "Exploring the RLDRAM II feature set," Micron Technologies, Inc., Tech. Rep. TN-49-02.

[2] "Making DRAM refresh predictable," in *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ser. ECRTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 145–154.

[3] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09, 2009, pp. 290–301.

[4] J. M. Borkenhagen, B. T. Vanderpool, and L. D. Whitley, "Read prediction algorithm to provide low latency reads with SDRAM cache," US Patent 6801982, 2004."

[5] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.16

[6] 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, pp. 159–169.

[7] J. Doweck, "Inside intel core microarchitecture and smart memory access," 2006, http://www.influentmotion.com/Server White Paper.pdf.

[8] 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.

[9] 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.

[10] P. Glaskowsky, "High-end server chips breaking records," August 2009.

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

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

[13] Z. Hongzhong, L. Jiang, Z. Zhao, E. Gorbatov, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency,"

in *41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 210 –221.

[14] J. Hruska, "Nehalem by the numbers: The Ars review." [Online]. Available: http://arstechnica.com/hardware/reviews/2008/11/nehalem-launch-review.ars/3

[15] I. Hur and C. Lin, "Feedback mechanisms for improving probabilistic memory prefetching," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009, pp. 443 –454.

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

[17] 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.

[18] 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*, 2009, pp. 337–346.

[19] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[20] 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.

[21] W. Jang and D. Z. Pan, "An SDRAM-aware router for networks-on-chip," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 800–805. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630117

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

[23] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "POWER 7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, pp. 7–15, March 2010.

[24] N.-Y. Ker and C.-H. Chen, "An effective SDRAM power mode management scheme for performance and energy sensitive embedded systems," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '03. New York, NY, USA: ACM, 2003, pp. 515–518. [Online]. Available: http://doi.acm.org/10.1145/1119772.1119879

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

[26] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1 –12.

[27] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 65–76.

[28] 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.

[29] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, 1981, pp. 81–87.

[30] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639 –662, 2007.

[31] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM controllers," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 200–209.

[32] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback - a technique for improving bandwidth utilization," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO

33. New York, NY, USA: ACM, 2000, pp. 11–21. [Online]. Available: http://doi.acm.org/10.1145/360128.360132

[33] J. Lin, H. Zheng, Z. Zhu, Z. Zhang, and H. David, "DRAM-level prefetching for fully-buffered DIMM: Design, performance and power saving," *IEEE International Symposium on Performance Analysis of Systems and Software*, vol. 0, pp. 94–104, 2007.

[34] W. Lin, S. Reinhardt, and D. Burger, "Designing a modern memory hierarchy with hardware prefetching," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1202 –1218, Nov. 2001.

[35] W.-f. Lin, "Reducing dram latencies with an integrated memory hierarchy design," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 301–. [Online]. Available: http://portal.acm.org/citation.cfm?id=580550.876450

[36] S. Liu, S. O. Memik, Y. Zhang, and G. Memik, "A power and temperature aware dram architecture," in *Proceedings of the 45th annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 878–883. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391691

[37] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in SMT processors," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001.

[38] 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.

[39] 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.

[40] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

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

[42] Micron Technologies, Inc, "DDR3 SDRAM system-power calculator, revision 0.1," 2007.

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

[44] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA:

IEEE Computer Society, 2009, pp. 261–270. [Online]. Available: http://portal.acm.org/citation.cfm?id=1636712.1637764

[45] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007, pp. 18:1–18:18.

[46] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 146–160.

[47] ——, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–74.

[48] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, 2006, pp. 208–222.

[49] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of DRAM bandwidth in multicore processors," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07, 2007, pp. 245–258.

[50] K. Rajamani, C. Lefurgy, S. Ghiasi, J. Rubio, H. Hanson, and T. Keller, "Power management solutions for computer systems and datacenters," in *Proceeding of the 13th international symposium on Low power electronics and design*, ser. ISLPED '08. New York, NY, USA: ACM, 2008, pp. 135–136. [Online]. Available: http://doi.acm.org/10.1145/1393921.1393956

[51] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/339647.339668

[52] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for cmp scaling," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09, 2009, pp. 371–382.

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

[54] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 285–294.

[55] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, pp. 505–521, July 2005.

[56] 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.

[57] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 375–384. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.22

[58] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: coordinating DRAM and last-level cache policies," in *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 72–82.

[59] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 219–230.

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

[61] 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*, 2004, pp. 206–207.

[62] M. Valero, T. Lang, and E. Ayguadé, "Conflict-free access of vectors with power-of-two strides," in *Proceedings of the 6th international conference on Supercomputing*, ser. ICS '92. New York, NY, USA: ACM, 1992, pp. 149–156. [Online]. Available: http://doi.acm.org/10.1145/143369.143403

[63] 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.

[64] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33, 2000, pp. 32–41.