

RICE UNIVERSITY

# Reducing DRAM Row Activations with Eager Writeback

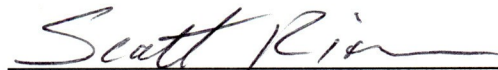
by

**Myeongjae Jeon**

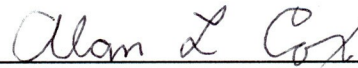
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

**Master of Science**

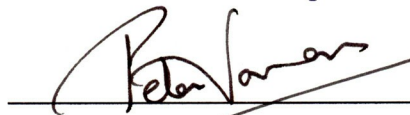
APPROVED, THESIS COMMITTEE:



Scott Rixner, Chair  
Associate Professor of Computer Science  
and Electrical and Computer Engineering



Alan L. Cox  
Associate Professor of Computer Science  
and Electrical and Computer Engineering



Peter J. Varman  
Professor of Electrical and Computer  
Engineering and Computer Science

Houston, Texas

May, 2012

## ABSTRACT

### Reducing DRAM Row Activations with Eager Writeback

by

Myeongjae Jeon

This thesis describes and evaluates a new approach to optimizing DRAM performance and energy consumption that is based on eagerly writing dirty cache lines to DRAM. Under this approach, dirty cache lines that have not been recently accessed are eagerly written to DRAM when the corresponding row has been activated by an ordinary access, such as a read. This approach enables clustering of reads and writes that target the same row, resulting in a significant reduction in row activations. Specifically, for 29 applications, it reduces the number of DRAM row activations by an average of 38% and a maximum of 81%. The results from a full system simulator show that for the 29 applications, 11 have performance improvements between 10% and 20%, and 9 have improvements in excess of 20%. Furthermore, 10 consume between 10% and 20% less DRAM energy, and 10 have energy consumption reductions in excess of 20%.

## Acknowledgments

I would like to first thank my advisors, Prof. Scott Rixner and Prof. Alan L. Cox, for their guidance and support on my research. Their suggestions and directions over the past year have had a major influence on all aspects of my research and helped me develop professionalism. I would also like to thank Prof. Peter Varman for his helpful advice and valuable comments on this thesis. Additionally, I am grateful to Conglong Li for his contribution to this project with several discussions and suggestions on the design.

I would like to thank our group members including Kaushik Kumar Ram, Thomas Barr, Brent Stephens, and Mehul Chadha for their feedback and advice at the practice talk. Also, I am grateful to my friends in CS including Xu Liu, Shams Imam, Dragos Dumitru Sbirlea, and Wei-Cheng Xiao for helping me prepare a successful defense. Prof. Jan Hewitt, throughout ENGI 600 class, helped shape this thesis. Finally, I would like to thank my parents for their unwavering support throughout the years of my study.

---

# Contents

---

Abstract	ii
List of Illustrations	vi
List of Tables	viii
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Contributions . . . . .	3
1.3 Organization . . . . .	5
<b>2 DRAM Data Access</b>	<b>6</b>
2.1 Energy and Delay . . . . .	6
2.2 Timing Constraints . . . . .	9
<b>3 Related Work</b>	<b>12</b>
3.1 Eager Writeback Schemes . . . . .	12
3.2 DRAM Access Clustering . . . . .	14
3.3 Hardware Approaches to DRAM Energy Savings . . . . .	15
3.4 Software Approaches to DRAM Energy Savings . . . . .	17
<b>4 Workload Analysis</b>	<b>19</b>
4.1 Effects of Cache Line Writebacks . . . . .	20
4.1.1 Frequency of DRAM Writes . . . . .	20
4.1.2 Frequency of Row Activations . . . . .	20
4.2 Effects of Row Activations . . . . .	21
4.3 Conclusions . . . . .	23

<b>5</b>	<b>DRAM Energy and Performance Optimizations</b>	<b>26</b>
5.1	Background: Eager Writeback . . . . .	26
5.2	Row Activation Reduction . . . . .	27
5.2.1	Illustrative Example . . . . .	28
5.3	Architecture Design . . . . .	30
5.3.1	Cache/Memory System Coordination . . . . .	33
5.4	Memory Access Scheduling . . . . .	35
5.5	Conclusions . . . . .	39
<b>6</b>	<b>Evaluation</b>	<b>40</b>
6.1	Full-System Simulator . . . . .	40
6.2	Applications . . . . .	42
6.3	Simulation Results . . . . .	42
6.3.1	Comparison to The Baseline . . . . .	43
6.3.2	Varying the Number of Cache Lines Written Back . . . . .	46
6.3.3	Evaluation with SWAS and Cancellation Disabled . . . . .	49
6.3.4	XOR-based Address Mapping . . . . .	50
6.3.5	Overhead . . . . .	53
6.4	Conclusions . . . . .	55
<b>7</b>	<b>Conclusions</b>	<b>59</b>
7.1	Future work . . . . .	60
	<b>Bibliography</b>	<b>62</b>

---

## Illustrations

---

2.1	Modern DRAM Structure [1]. Row address is issued by row activate or precharge command, and column address is issued by read or write command. . . . .	7
2.2	Row Activation Timing. . . . .	10
4.1	Read/Write Operations per Activated Row. <i>libquantum</i> Read (19.13), <i>libquantum</i> Write (14.91), <i>libquantum</i> No Write (48.89), and <i>lbn</i> No Write (29.72) are cut off due to the space. . . . .	22
5.1	Reshaped Memory Access Pattern with Eager Writeback. . . . .	29
5.2	A coordinated Cache/Memory System. . . . .	34
5.3	Scheduling Eager Writeback Operations. See Figure 5.1 for request type and latency information. . . . .	37
6.1	Performance Improvement and Energy Savings for 29 Applications (Normalized to The Results of No Eager WB). The larger number means the more optimized. . . . .	45
6.2	Performance Improvement with Different Cache Lines for a Eager WB (denoted as Eager WB-N for using N LRU-side lines for a speculative writeback issue). . . . .	47
6.3	Energy Savings with Different Cache Lines for a Eager WB (denoted as Eager WB-N for using N LRU-side lines for a speculative writeback issue). . . . .	48
6.4	Comparison of Performance Improvement with Results Observed With SWAS and Cancellation disabled. . . . .	50

6.5	Comparison of Energy Savings with Results Observed With SWAS and Cancellation disabled. . . . .	51
6.6	Comparison of Performance Improvement with XOR-Based Memory Mapping (XOR) and with Eager WB on XOR (Eager WB-XOR) . . . . .	52
6.7	Comparison of Energy Savings with XOR-Based Memory Mapping (XOR) and with Eager WB on XOR (Eager WB-XOR) . . . . .	53
6.8	Decomposition of DRAM Accesses by Access Type. No Eager WB, Eager WB-1, and Eager WB-2 correspond to the three stacked bars (in order from left to right) for each application. The fractions in all of the values are relative to the total number of DRAM accesses in No Eager WB.	54

---

## Tables

---

2.1	Micron DDR3-1600 Delay and Energy Costs [2]. . . . .	8
4.1	Decomposition of DRAM Accesses, DRAM Activations, and Row Buffer Hit Count by Access Type. SPEC applications are sorted in decreasing order of WR in DRAM Accesses. . . . .	25
5.1	Fraction of Writes to Dirty Cache Lines with Respect to the LRU-MRU Position. . . . .	32
6.1	Processor and DRAM System Parameters. . . . .	41
6.2	Decomposition of Row Activations. The SPEC applications are presented in the same order as in Table 4.1, which is based on the fraction of DRAM accesses that are writes. . . . .	57
6.3	Average Row Buffer Hit Count for No Eager and Eager WB. The Improved indicates the the number that the results in Eager WB are normalized to the results in No Eager WB. . . . .	58



# CHAPTER 1

---

## Introduction

---

### 1.1 Introduction

DRAM has become as a performance bottleneck in many computing systems because its speed, pin count, and pin bandwidth have all increased very slowly [3, 4]. DRAM performance becomes an even more critical issue in modern systems with the widespread use of data-centric applications. These applications, including scientific applications and databases, have much larger memory footprints than ever before and thus dramatically increase the demand for memory bandwidth. Since the bandwidth provided by DRAM is a very limited resource and is not scalable, it is often under extremely heavy contention that degrades overall system performance considerably.

The performance is not all that matters in the design of DRAM systems. DRAM has become one of the largest energy consumers in modern server systems. Therefore, the DRAM power budget is now comparable to or even exceeds the CPU power budget. It has been shown that DRAM in a commercial server consumes 25–45% of total system-wide power [5, 6, 7]. For example, DRAM in Google’s datacenters spends up to 30% of the total system power, which is similar in amount to CPU power consumption, which is 33% of the total [6]. In a next-generation high-end POWER7 server, power consumed in DRAM system is expected to grow up to 46% of the total [7]. This is the largest portion in the

breakdown of system power with respect to system components, such as processors and DRAM. With this trend, DRAM energy efficiency has emerged as the first-order design constraint of the server design.

In designing energy efficient DRAM, the dynamic power consumption has become a much more important concern than the static power consumption. DRAM dynamic power consumption grows proportionally to dynamic activities experienced in DRAM, such as the rate of DRAM accesses, while DRAM static power consumption is solely determined by its capacity. Therefore, the amount of DRAM power varies depending on its dynamic activities. For example, Micron DDR3-1600 1GB DRAM with no access consumes 1.6 watts of power, but under the maximum access rate it can consume up to 15.17 watts [8]. Managing the dynamic power becomes a greater issue because DRAM access rate has rapidly been increasing with the popular use of data-centric applications.

To optimize DRAM performance and dynamic energy consumption, the primary focus should be placed on improving the row-level locality of access and reducing the number of row activations. Every DRAM operation requires a chunk of data to access (called *a row*) to be placed in DRAM buffer area (called *row buffer*) before the read/write access is scheduled. This process, *row activation*, is recognized as the most expensive DRAM operation in terms of energy and delay. The row buffer (or a row) is typically 4–16KB in size, and so it stores 64–256 adjacent cache lines at a time. Thus, if the DRAM system performs as many read and write accesses per activated row as possible, then row activations are correspondingly reduced. Rows otherwise would be activated so frequently that energy and delay are significantly wasted.

In modern systems, there are typically only 1–3 cache line accesses per row activation. This thesis analyzes DRAM activities and shows that cache line writebacks are the root cause of the lack of row utilization. Specifically, since last level caches evict dirty cache

lines without taking the locality of data written back to DRAM into account, 1) writebacks often require row activations, and 2) they often cause future read accesses to re-activate rows that would have already been in the row buffer if the writeback has not occurred. This thesis addresses these two problems by placing an emphasis on the achieved row locality in DRAM when performing line writebacks.

Recent efforts have been made to optimize these DRAM inefficiencies by clustering cache line writebacks that target the same row [9, 10]. These approaches have been successful at maximizing the number of write accesses when a row is activated for a writeback operation. However, these approaches still have cache line writebacks that exhibit no locality of access with the surrounding DRAM read operations. Therefore, the aforementioned problems are never eliminated entirely. To eliminate the problems entirely, the best way is to cluster writebacks with both read and writes that all target to the same row. This clustering can maximize overall energy savings and performance because of the overall high number of DRAM accesses per row activation.

## 1.2 Contributions

In this thesis, we first present analysis results that show how applications exercise the memory system in practice. We delved into how activated rows are accessed for each access type, *i.e.* cache line read or writeback, and found that writebacks are a problem. Specifically, we found that rows are activated by writebacks too frequently and that these writebacks are a main reason that makes the activated rows insufficiently accessed. Applications may exhibit good data access locality that would result in much reuse in the row buffer by reads. However, writebacks often interfere with that locality, reducing the number of accesses per row activation. Eliminating this interference is a key to decreasing the number of row activations in DRAM.

Second, we present a technique for dealing with problems observed in the analysis. We propose a novel scheme that significantly reduces the number of row activations through clustering of read and write accesses that target the same row. The proposed scheme contributes to addressing inefficiencies existing in the current memory system. Specifically, it transforms destructive and expensive writebacks that accompany DRAM row activations into very efficient ones that require no such row activations. Moreover, it enables data access locality exhibited in an application to be kept in the row buffer, without being interfered with by writebacks. As a result, the proposed scheme can significantly optimize DRAM performance and energy consumption.

There have been attempts to cluster a specific type of memory accesses, *e.g.* writeback, for improving the low-level locality of access. However, there have been no attempts to cluster read and write accesses to the same row. This clustering of reads and writes has several advantages. First, in most cases, it enables DRAM writes to be effectively related with DRAM reads in terms of hits on the row buffer. Second, even applications that do not show enough data access locality can benefit from it. Finally, it can offer better scalability in supporting multi-core systems by activating rows much less frequently, thereby reducing competition for the row buffer.

Last, this thesis explores the design space of the eager writeback scheme that is used to improve the row-level locality of access. The scheme can be aggressive by processing a burst of eager writebacks without being stopped, or can be conservative by clearing away those writebacks from the memory controller when an ordinary, non-eager access appears. Also, it can be aggressive by eagerly writing back all dirty cache lines of a row, or can be conservative by writing back only a part of them. We explore these configurations for the scheme and present the trade-offs involved with energy consumption and performance for each configuration.

### **1.3 Organization**

This thesis is organized as follows. Chapter 2 provides relevant background on DRAM access characteristics in terms of performance and energy. Chapter 3 discusses related work. Chapter 4 explains analysis results that motivate our eager writeback scheme. Chapter 5 describes the design of the scheme. Chapter 6 presents experimental results under a variety of different configurations. Finally, we summarize this thesis and present future work in Chapter 7.

## CHAPTER 2

---

### DRAM Data Access

---

Modern DRAM devices are organized into banks, rows, and columns, as shown in Figure 2.1. This structure leads to non-uniform energy and latency of access [11]. Therefore, the memory controller must carefully schedule memory accesses in order to minimize energy consumption and access latency. This chapter explains DRAM access characteristics in detail and highlights the importance of intelligently managing row activations to build high-performance and energy-efficient systems.

#### 2.1 Energy and Delay

As Figure 2.1 shows, a DRAM device is itself partitioned into 4–16 banks. Each bank can process memory requests independently of other banks. The memory controller utilizes this *bank-level parallelism* to maximize DRAM bandwidth by overlapping the processing of memory requests across different banks. However, memory cells within a bank cannot be accessed directly. Instead, data may only be read or written from the *row buffer*. Each bank has a dedicated row buffer that can hold one row from the bank at a time. A row is transferred from the memory array to the row buffer by a *row activate* command. After an arbitrary number of read and write commands which transfer columns within the row buffer, the *precharge* command restores the row back to the memory array and prepares the bank for another row activation.

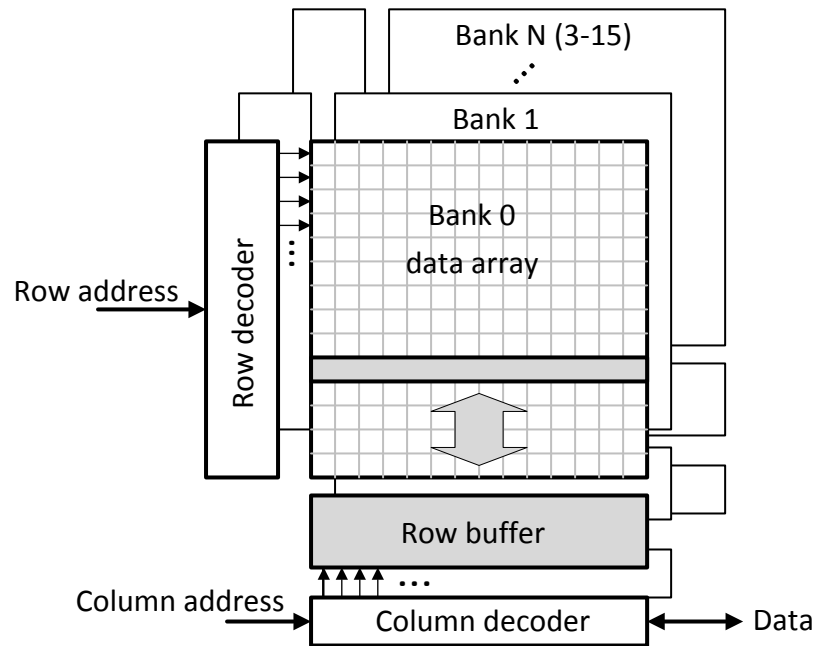


Figure 2.1 : Modern DRAM Structure [1]. Row address is issued by row activate or precharge command, and column address is issued by read or write command.

Multiple DRAM devices are tied together across a rank to create a wider data interface. Within a rank, the address lines are tied together to create the illusion of a single, wider DRAM. This leads to rows of 4–16KB across the rank. A modern DIMM consists of 1–4 ranks of DRAM devices.

Modern DRAM timing is quite complex and there are many constraints placed upon when DRAM commands (precharge, activate, read/write) can be issued. These constraints ensure both that data can traverse the long internal wires of the DRAM device and that the power dissipation does not exceed the limits of the device.

The precharge and activate commands are far more expensive, both in terms of energy and delay, than the read and write operations to a column within the row buffer. Intuitively, this makes sense, as the column operations simply read a few bytes out of a buffer, whereas the precharge and activate commands must drive thousands of long wires across an entire

Operation	Latency		Energy
	Cycles	ns	nJ
Precharge	10	12.5	3.90
Activate	10	12.5	
Column read	14	17.5	1.44
Column write	12	15.0	1.44

Table 2.1 : Micron DDR3-1600 Delay and Energy Costs [2].

DRAM bank.

To illustrate the delay and energy costs of a modern DRAM, Table 2.1 shows the timing and energy costs of the various DRAM operations for a Micron DDR3-1600 1 Gb 8-bit wide DRAM device [2]. Recall that before performing a column operation, the appropriate bank must first be precharged and then the appropriate row within that bank must be activated in order to transfer it to the row buffer. So, to read a single burst of data (64 bits in this example) would take  $10 + 10 + 14 = 34$  memory cycles, or  $42.5$  ns. Note that the read access occupies the data bus for only the last four of the 14 cycles. Once a row has been activated, however, read operations can be pipelined, yielding 64 data bits every 4 memory cycles (due to the 8-bit wide double data rate bus). As such DRAM devices are aggregated into a rank, as discussed above, each read would occur simultaneously across 8 DRAM devices to yield a 64-byte cache line for each read operation. This clearly illustrates that there is an enormous advantage to reading multiple cache lines per row, to achieve the peak bandwidth of the DRAM of one cache line every 4 cycles, instead of one cache line every 34 cycles.

Write operations are similarly expensive, with a potential 32 memory cycle delay. They



also can be pipelined with each other at a rate of one burst every 4 memory cycles. However, read and write operations cannot be easily interleaved. When transitioning from reading to writing, there is a two cycle delay in the Micron DRAM to allow the bus drivers to turn around. The situation is worse when transitioning from writing to reading, as the logic for dealing with the row buffer requires these operations to be isolated. This leads to a 20 memory cycle delay from the completion of a write to the completion of the read (6 empty memory cycles plus the full 14 cycle latency of a column read).

While these latencies are all quite high, in practice the full latencies can be partially hidden for two reasons. First, precharge operations can be overlapped with read operations so that you can precharge the bank as the last column read is proceeding. Second, precharge and activate operations can occur in parallel to other banks while the row buffer from one bank is being read or written. However, due to power considerations, there are limits to the amount of bank parallelism that can be exploited.

## 2.2 Timing Constraints

As Table 2.1 shows, the combination of a precharge and an activate command for a bank consumes  $3.9 \text{ nJ}$ . Not only is this more than twice the energy of a read or write access, these row operations can occur in parallel for multiple banks. If there were no limitations, all eight banks within a DRAM could be activated simultaneously, which would result in instantaneous power dissipation of 1.25 W per DRAM device, without even transferring any data across the pins of the device. In a memory system with 8 DIMMs, each with 8 DRAM devices, that would consume 80 W.

To limit the power dissipation of the DRAM, modern devices limit the rate at which row activations can occur. The three key DRAM timing constraints that help limit power consumption are the row cycle time ( $t_{RC}$ ), the activate to activate delay ( $t_{RRD}$ ), and the

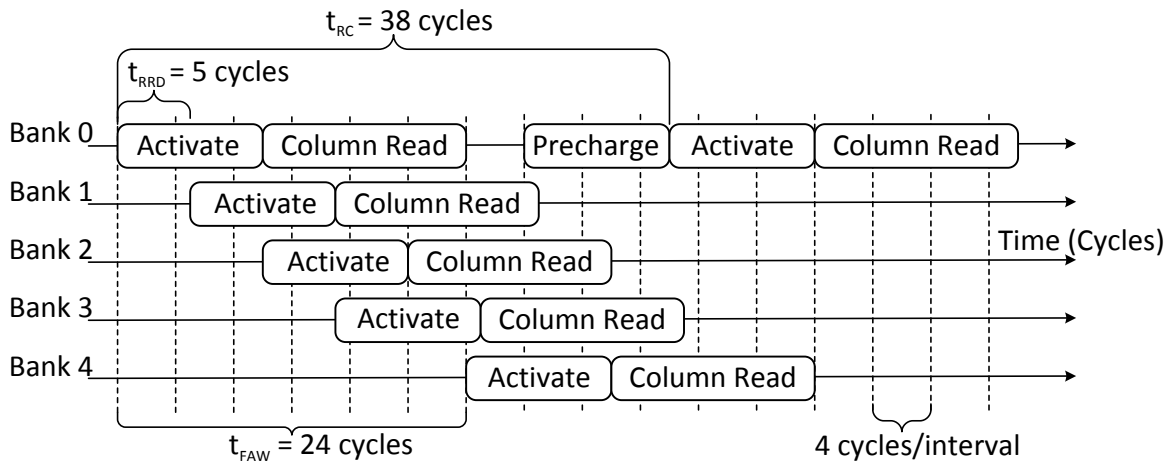


Figure 2.2 : Row Activation Timing.

four-bank activation window ( $t_{FAW}$ ). Figure 2.2 illustrates these timing constraints. The row cycle time is the minimum time interval between successive row activations in the same bank. The activate to activate delay is the minimum time interval between successive row activations to different banks in the same rank. The four-bank activation window is a time window in which no more than four row activations are allowed in the same rank. These timing constraints all prevent activations from occurring too frequently, but they also limit bandwidth utilization if there are few reads/writes to the activated row.

For the Micron device,  $t_{RC}$  is 38 memory cycles,  $t_{FAW}$  is 24 memory cycles, and  $t_{RRD}$  is 5 memory cycles. Under these constraints, it would take 24 memory cycles to issue five consecutive activation commands to different banks, as shown in Figure 2.2. If the first activation is issued on memory cycle 0, the subsequent four activations would be issued on cycles 5, 10, 15, and 24. The first four activations are limited only by  $t_{RRD}$ , whereas the last one is limited by  $t_{FAW}$ . Furthermore, another row activation (after precharging) could not be issued to the first bank until cycle 38 because of  $t_{RC}$ . These constraints make it even more critical to perform multiple column accesses per row activation, as they further delay

row activations in a busy memory system.

Due to the latency, energy, and timing constraints of row activations, it is important to activate rows judiciously and to perform as many read and write accesses per activated row as possible. In order to mitigate the cost of row activation, there have been several proposals to either reduce the row size of the DRAM [12, 13, 14, 15] or to improve the locality of access within a row [16].

---

## **Related Work**

---

The technique for read and write clustering presented in this thesis is built upon eager writeback. Eager writeback was recently used to find more writebacks that can cause row hits [9, 10]. In our work, writebacks are clustered with both reads and writes to maximize row hit operations. Classic DRAM access clustering techniques can alleviate row buffer conflicts by grouping some accesses together. However, they do not consider energy consumption. Work on saving memory energy consumption can be divided into hardware-based approaches and software-based approaches. Hardware innovations recently proposed have mostly focused on architecting DRAM with smaller row buffer size to lower dynamic energy consumption by activating fewer bits. This, however, sometimes sacrifices system performance. The software-based approaches are restricted to an optimization that exploits low power modes of DRAM for saving background energy consumption. Our work is complementary to these approaches.

### **3.1 Eager Writeback Schemes**

Lee *et al.* [17] first proposed *eager writeback* to optimize systems that experience a significant number of cache misses while fetching streaming data. In writeback caches, dirty cache lines are evicted and written to DRAM at the same time that the data is being fetched into the cache. This causes the memory bus congestion that impedes the delivery of data to

the cache, thereby negatively affecting an application's forward progress. Eager writeback distributes the competing traffic by writing dirty cache lines to DRAM before the lines must be evicted from the cache. The concept of eager writeback has recently been used to address a different problem, *i.e.* improving row-level access locality.

*Virtual Write Queue* [9] is a recent proposal that exploits eager writeback for increasing row-level access locality. This technique coordinates DRAM access scheduling with cache writeback to increase the opportunities to find dirty cache lines in the last-level cache that can hit on the activated row. While write operations in the DRAM write queue are being scheduled, other dirty cache lines that are mapped to the same row as the scheduled ones are searched in the last-level cache and immediately transferred to DRAM. Eager writeback is therefore exploited in *Virtual Write Queue* to increase row-level locality of DRAM writes.

At the same time that *Virtual Write Queue* was introduced, a similar technique, called *DRAM-Aware Writeback* [10], was proposed. It also exploits eager writeback to cluster writebacks. *DRAM-Aware Writeback* instead monitors the writeback of dirty cache line entering into the processor's write buffer and finds other dirty cache lines in the last-level cache that are mapped to the same row as the evicted one. Doing so can populate the write buffer with many row hit eager writebacks. For a trigger of the flushing of eager writebacks to the DRAM, *DRAM-Aware Writeback* utilizes *drain-when-full* write buffer management – the drain-when-full services all writes in the write buffer at once when it is full.

*Virtual Write Queue* and *DRAM-Aware Writeback* enable writebacks to be consolidated in order to maximize the number of column writes when a row is activated for a writeback operation. However, both of these schemes have two limitations which have been addressed in this thesis. First, they only cluster writebacks with other writebacks. In contrast, our proposed scheme clusters writebacks with both reads and writes. Clustering writebacks with both reads and writes is critical in order to maximize overall energy savings and per-

formance because of the overall low number of column operations per row activation (as discussed in Section 4.2). Second, these prior approaches do not allow eager writebacks to be cancelled. We cancel eager writebacks when new conflicting read operations arrive, thereby ensuring that subsequent read operations are not delayed. Without cancellation, eager writebacks can delay non-eager reads, leading to decreased performance.

### 3.2 DRAM Access Clustering

Some efforts have been made to try to avoid access interference within banks. The XOR-based address mapping [18] scheme avoids row buffer conflicts between reads and cache writebacks under page interleaving by generating bank addresses pseudo-randomly. This technique effectively redirects to another bank writes that would conflict in a bank. We simulate this address mapping scheme in Section 6.3.4 and compare it with our eager writeback scheme. A write buffer with read bypass [19] could also alleviate row buffer conflicts by postponing writebacks. This potentially would allow consecutive reads to be grouped together.

Unlike the technique presented in this thesis, these schemes do not eliminate write interference but rather attempt to reduce it, and neither scheme reduces the number of row activations caused by writes. However, because the XOR-based address mapping is an address translation optimization, combining it with our eager writeback scheme achieves better results than either one alone, as discussed in Section 6.3.4.

There have been several studies, such as [20, 1, 21, 22, 23, 24, 25, 26], that present techniques to reschedule memory accesses to improve overall system performance. These works introduce a variety of scheduling mechanisms and algorithms to increase performance. However, they do not consider energy consumption and are orthogonal to our read and write clustering technique.

### 3.3 Hardware Approaches to DRAM Energy Savings

Sudan *et al.* [16] recently observed that a large number of accesses within heavily accessed OS pages happen to small, contiguous chunks of cache lines, and they therefore proposed the co-location of the chunks from different OS pages in a row buffer to improve its utilization. This technique is effective if applications expose spatial locality of access in rows. Therefore, some of the applications discussed in Section 4.2, such as *sjeng* and *gems*, which do not inherently have good spatial locality, will not benefit from this technique at all.

Energy and latency could further be reduced by utilizing smaller rows within the DRAM. Rixner [12] and Cooper-Balis *et al.* [27] presented memory controller policies that can make effective use of commercial DRAM architectures that support the use of subsets of rows to further reduce the average latency of the DRAM. These types of proposals have recently been revisited in the context of multi-core systems where access streams are mixed to compose more interference in the row-buffers. Udipi *et al.* [15] proposed a re-design of DRAM to activate subsets of a row and to keep inactive subarrays in low-power sleep modes. Others have proposed a *rank subsetting* that enables the smaller number of devices to be involved for a memory access [13] [14]. A comprehensive analysis for the effectiveness of rank subsetting on the performance, energy efficiency, and reliability is presented in [28]. All of these techniques are orthogonal to our read and write clustering strategy and we would expect the ideas to be complementary.

Taking advantage of low-power modes has also been focused in the memory controller. Hur *et al.* [29] extended the Adaptive History-Based Scheduler (AHB) with power consumption to increase the average idle duration of each rank, thus increasing the utility of the power-down unit. Huang *et al.* [30] identified memory access traffic as often random when the OS arbitrarily maps virtual pages to physical pages. They thus proposed

a way to reshape such random memory traffic to produce longer idle periods by which more aggressive power-saving mode is actively utilized. These two optimizations focus on optimizing DRAM static energy consumption while our work focuses on optimization DRAM dynamic energy consumption. We would expect both types of techniques to be complementary and to be easily combined.

Since re-activating main memory from a low-power mode is costly, it is crucial to mask the resynchronization time associated with re-activating memory modules. Pisharath *et al.* [31] proposed on-chip memory module buffer that reduces the overhead incurred due to frequent resynchronization of memory modules. Floyd *et al.* [32] designed a queue-driven policy in IBM POWER6 on-chip memory controller for power-down exploitation. Under the policy, applications of the power-down mode rarely see loss of performance, yet the system can reduce DRAM power consumption significantly.

Memory is also a major power consumer in mobile devices [33, 34]. Mobile devices can be either in active mode or in standby (or suspended) mode, and in standby mode, the refresh operation is dominant consumer of DRAM power. Liu *et al.* [35] designed Flicker, a software/hardware coordinated technique for lowering the refresh rate of the part of memory containing non-critical data to save power; an example of non-critical data is frame data in a video processing application. Since the application's reliability is not sensitive to errors in the non-critical data, making marginal disruptions in the data is acceptable. Isen and John [36] proposed ESKIMO, a hardware mechanism to reduce the refresh power of unused part of memory. In ESKIMO, the hardware exploits information about application's memory allocation patterns that are exposed by memory allocator. Finally, DRAM refresh operations can be reduced by employing a time-out counter for each DRAM row. Ghosh and Lee [37] utilized the counter to identify the DRAM row that was recently read or written to by the processor and thus that does not need to be refreshed by periodic re-



fresh operation. I would expect these techniques can be used in combination with our eager writeback technique to save power in both active mode and suspend mode of mobile devices.

### 3.4 Software Approaches to DRAM Energy Savings

At a much higher semantic level, there have been approaches to save memory power/energy consumption in the context of OS or VMM hypervisor. In these approaches, memory management [38, 39, 40, 41, 42], IO processing [43], runtime library [44], compiler [45, 46], or process/VM scheduling [47, 48, 49] of OS or VMM hypervisor is optimized to be aware of memory power characteristics. Then, based on this, low power modes that can be applied to each memory unit (i.e. a DRAM device or a rank) are exploited.

Huang *et al.* [39] optimized OS memory management to reduce energy consumption by putting more memory into low power modes. They specifically exploited page aggregation and migration techniques by which the average number of active memory devices used by a process can be minimized. More memory devices that are idle therefore can take advantage of low power modes to consume less power. Lee *et al.* [41] additionally took buffer caches of the OS into account in power management to save more energy consumption of memory. These proposals are for per-process power management where all memory ranks that belong to the running process are powered on and kept on during the execution. Bi *et al.* [42] addressed this limitation by managing memory power when file-I/O system calls in the OS are invoked. They exploited the delay for the system calls to hide the delay incurred by the memory power mode transition so that performance degradation is minimized.

Pandey *et al.* [43] characterized the effect of DMA accesses on memory energy and identified that significant energy is wasted due to the mismatch between memory and I/O bus bandwidths. Due to the mismatch, memory is often idle but active (in power consump-

tion) during DMA transfers. They therefore proposed DMA-aware techniques for memory energy management that maximize the utilization of memory devices by increasing the level of concurrency between multiple DMA transfers from different I/O buses to the same memory device.

Scheduling also has a dramatic impact on memory energy consumption. Merkel *et al.* [48] found that scheduling for avoiding resource contention is crucial both in terms of performance and energy efficiency. Accordingly, they designed a scheduling policy that avoids memory contention by sorting the processors' runqueues by memory intensity, and a frequency heuristic based on memory intensity. Jang *et al.* [49] proposed memory-aware virtual machine scheduling for virtualized servers running on multicore systems. Under the proposed scheduling, memory ranks that belong to virtual machines that run on the cores overlap as many others as possible so that more ranks are put into low power modes.

All of these techniques are intended to save DRAM background (static) energy consumption through, for example, transition to low power modes. However, they do not improve row-level locality of accesses that determines DRAM dynamic energy consumption, and therefore are orthogonal to our read and write clustering strategy.

---

## **Workload Analysis**

---

Given the impact of row buffer utilization on both delay and energy discussed in the previous section, it is important to understand how applications exercise the memory system. In modern systems, there are typically only 1–3 column accesses per row activation. This leads to poor DRAM performance and energy consumption. This chapter explores in detail the root cause of this lack of utilization.

This chapter clarifies the problems with cache line writebacks. These writebacks interfere with the read locality found in every application that we studied and themselves lead to significant performance and energy problems. Table 4.1 highlights these problems. The data in this table was collected using the same experimental methodology that will be described in Chapter 6. To make sure that results are not influenced by the lack of bank-level parallelism, we varied the degree of the parallelism by having a configuration of more memory ranks; we observed similar results.

This chapter is organized as follows. Section 4.1 characterizes frequencies of DRAM accesses and row activations for 29 applications. Section 4.2 continues to analyze the utilization of activated rows and discusses the cause of insufficient accesses to the row buffer that almost all applications have shown. Section 4.3 summarizes this chapter’s conclusions.

## 4.1 Effects of Cache Line Writebacks

### 4.1.1 Frequency of DRAM Writes

Whenever a cache line is first accessed, it must be read from the DRAM. Therefore, both read and write cache misses trigger DRAM read accesses. Given that all modern systems employ write-back caches, DRAM writes occur only when a dirty cache line is evicted. Intuitively, this should occur far less often than DRAM reads.

The first column of Table 4.1 shows the number and fraction of DRAM accesses that are reads and writes for 29 applications. For 20 of these applications, writes account for more than 20% of total DRAM accesses. Writes account for more than 40% of the DRAM accesses for some of these applications (*libquantum*, *hmmmer*, *lbm*, *astar*, *sjeng*, and *apriori*). Therefore, DRAM writes are not actually all that infrequent.

### 4.1.2 Frequency of Row Activations

Modern cache replacement policies attempt to achieve the highest hit rate in the cache possible. When performing eviction, they select a cache line that is not likely to be accessed later in the set, *e.g.* the least recently accessed one in the LRU scheme. They do not typically consider the locality of data written back to the next level in the memory hierarchy. In particular, last level caches place no emphasis on the achieved row locality in the DRAM, which can greatly influence overall performance. Therefore, writes caused by evictions frequently exhibit no locality of access with the surrounding read operations. Furthermore, they are most likely unrelated to other surrounding write operations, as well.

The second column of Table 4.1 shows the number and fraction of row activations caused by read and write operations. Except for 4 applications (*libquantum*, *tonto*, *h264ref*, and *namd*), more than 70% of the write operations cause row activations. For example,

*bzip2* has 1,064,285 total writes, of which 865,982 (81.4%) cause row activations. This leads to the situation where row activations caused by writes account for more than 30% of all row activations in 19 of the applications. In 6 applications, writes account for more than 50% of all row activations.

## 4.2 Effects of Row Activations

More important than the frequency of row activations caused by cache writebacks is the damage that these row activations cause. The third column in Table 4.1 shows the number of read/write operations that occur per activated row. As the table shows, for most applications there are 1–3 read/write operations per activated row, with an average of 3.13 read operations and 1.91 write operations. Worse, there are fewer than 2 write operations per activated row for 25 of the applications. Furthermore, for every application except *gems* and *bwave* there are more read operations per activated row than write operations.

Surprisingly, the trends for rows that are activated by read operations is not much better than for rows that are activated by writes. Except for *libquantum*, which is an outlier, the most reads per activated row is just 5.64 for *namd*, and most applications perform less than 3 reads per activated row.

There are two reasons for the very low hit rates on activated rows. First, not all applications have good spatial locality in the row buffer. These experiments map the physical address space across ranks and banks at a row granularity, which should provide the maximum opportunity for such row locality. Therefore, purely sequential access would yield numerous read operations per activated row, but this is not happening because the cache hierarchy acts as a filter over the memory access stream which yields less predictable access. Second, even when applications do have good spatial locality, cache evictions often interfere with that locality, further reducing the number of read accesses per row activation.

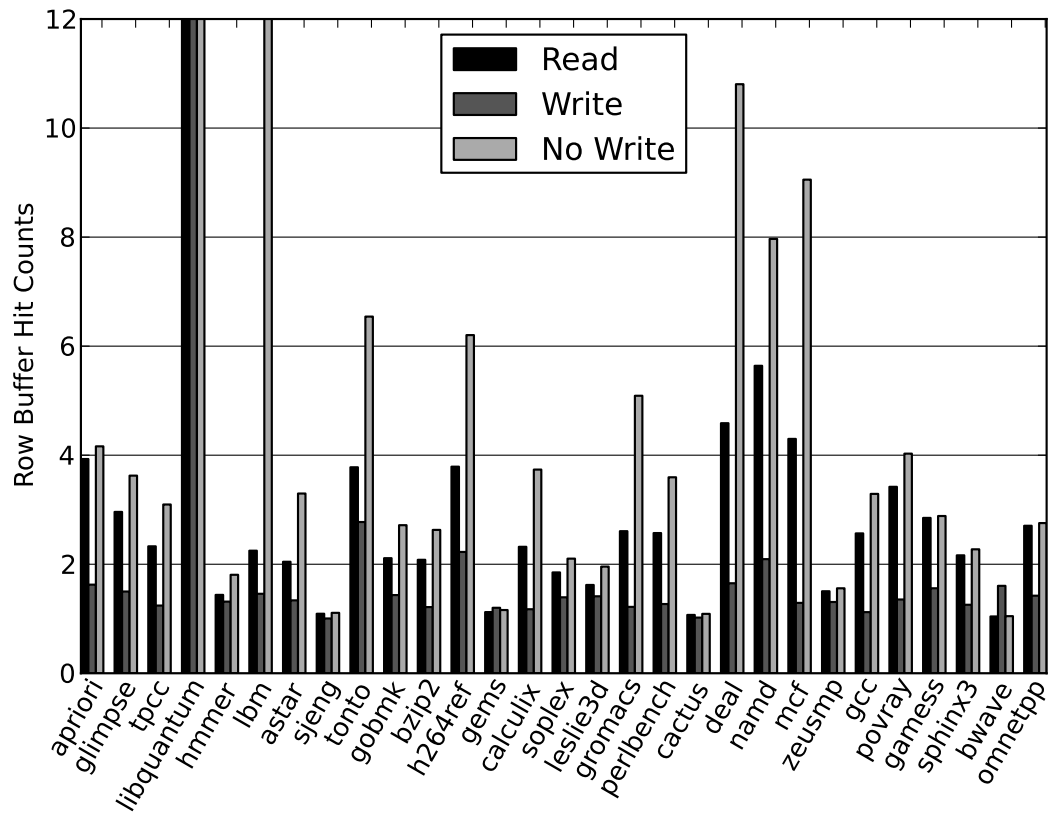


Figure 4.1 : Read/Write Operations per Activated Row. *libquantum* Read (19.13), *libquantum* Write (14.91), *libquantum* No Write (48.89), and *lbm* No Write (29.72) are cut off due to the space.

We will refer to this phenomenon as *write interference* throughout the paper. This write interference increases the number of row activations considerably.

Figure 4.1 shows the number of read/write operations that occur per row buffer activation. The first two bars (labeled “read” and “write”) are repeated from Table 4.1. The third bar, labeled “no write”, shows what would happen if all of the write operations were discarded. This indicates the amount of row locality there is in the reference stream, and implicitly shows the damage done by write interference. While all applications show at

least modest increases in the number of reads per activated row, many do not increase by all that much. However, the number of reads per activated row does increase markedly for eight of the applications (*libquantum*, *lbm*, *tonto*, *h264ref*, *gromacs*, *deal*, *namd*, and *mcf*). Therefore, eliminating write interference will have a positive effect for all applications and will have a significant impact on some applications.

One might think that simply prioritizing row hit reads (*i.e.*, column-first scheduling) would suffice to avoid write interference when row hit reads and row miss writes compete for access to the DRAM. However, this is not always the case. A memory controller will choose the best issuable command among pending DRAM commands on each memory cycle. So, if the next row hit read arrives even a cycle late, then the memory controller may already have begun the precharge/activate cycle for the row miss write. This subtle timing occurs often in practice.

Furthermore, some efforts have been made to avoid access interference within banks. Examples include the XOR-based address mapping [18] scheme and a write buffer with read bypass [19]. As already discussed in Section 3.2, both schemes do not eliminate write interference but rather attempt to reduce it. Moreover, neither scheme can relate writes to surrounding reads in terms of locality of access.

### 4.3 Conclusions

In modern systems, there are typically only 1–3 column accesses per row activation. This leads to poor DRAM performance and energy consumption. The root cause of this lack of utilization is cache line writebacks. Specifically, they occur more often than might be expected, they often require row activations, and they often cause future read accesses to re-activate rows that would have already been in the row buffer if the write had not occurred. The first two phenomena lead to significant performance and energy problems

of the DRAM. The situation is worse with the third phenomenon, though, because even when applications exhibit good spatial locality of access, writebacks often interfere with that locality, reducing the number of accesses per row activation.



	DRAM Accesses		Row Activations		Row buffer hit count		
	RD	WR	RD	WR	RD	WR	Avg.
libquantum	6396537 (0.50)	6346650 (0.50)	334285 (0.44)	425596 (0.56)	19.13	14.91	16.77
hammer	1557443 (0.52)	1446100 (0.48)	1082032 (0.50)	1101266 (0.50)	1.44	1.31	1.38
lbm	14573106 (0.58)	10669136 (0.42)	6481510 (0.47)	7317800 (0.53)	2.25	1.46	1.83
astar	3689208 (0.60)	2458745 (0.40)	1801606 (0.49)	1841565 (0.51)	2.05	1.34	1.69
sjeng	210523 (0.60)	139795 (0.40)	192496 (0.58)	139369 (0.42)	1.09	1.01	1.06
tonto	106241 (0.61)	68368 (0.39)	26786 (0.50)	26469 (0.50)	3.78	2.77	3.28
gobmk	684619 (0.68)	323681 (0.32)	318330 (0.58)	234455 (0.42)	2.11	1.43	1.82
bzip2	2317657 (0.69)	1064285 (0.31)	1119656 (0.56)	865982 (0.44)	2.08	1.21	1.70
h264ref	686468 (0.70)	300904 (0.30)	180844 (0.57)	135736 (0.43)	3.79	2.23	3.12
gems	10811870 (0.70)	4550271 (0.30)	9616973 (0.72)	3804887 (0.28)	1.12	1.20	1.14
calculix	23210 (0.73)	8611 (0.27)	9970 (0.57)	7405 (0.43)	2.32	1.17	1.83
soplex	8677229 (0.73)	3207230 (0.27)	4680762 (0.67)	2315124 (0.33)	1.85	1.39	1.70
leslie3d	5869350 (0.77)	1756647 (0.23)	3626551 (0.75)	1239772 (0.25)	1.62	1.41	1.57
gromacs	495834 (0.78)	140818 (0.22)	189728 (0.62)	116692 (0.38)	2.61	1.22	2.08
perlbench	1828075 (0.78)	511830 (0.22)	706975 (0.63)	410769 (0.37)	2.57	1.27	2.09
cactus	2702913 (0.78)	761070 (0.22)	2527874 (0.77)	742433 (0.23)	1.07	1.02	1.06
deal	892999 (0.79)	231881 (0.21)	193273 (0.57)	144297 (0.43)	4.59	1.65	3.33
namd	90830 (0.82)	20106 (0.18)	16089 (0.62)	9665 (0.38)	5.64	2.09	4.31
mcf	30894021 (0.84)	5990861 (0.16)	7184316 (0.61)	4646697 (0.39)	4.30	1.29	3.12
zeusmp	1543721 (0.84)	283265 (0.16)	1021625 (0.82)	221578 (0.18)	1.50	1.31	1.47
gcc	97740 (0.86)	15796 (0.14)	37888 (0.72)	14570 (0.28)	2.57	1.12	2.16
povray	7975 (0.89)	975 (0.11)	2322 (0.75)	758 (0.25)	3.42	1.35	2.91
gamess	7480 (0.92)	607 (0.08)	2539 (0.82)	552 (0.18)	2.85	1.56	2.62
sphinx3	6674087 (0.95)	346501 (0.05)	3083259 (0.92)	278787 (0.08)	2.16	1.26	2.09
bwave	31388285 (0.95)	1545320 (0.05)	30119315 (0.97)	963946 (0.03)	1.04	1.60	1.06
omnetpp	6901520 (0.97)	207057 (0.03)	2549077 (0.94)	148554 (0.06)	2.71	1.42	2.64
apriori	13068580 (0.51)	12574560 (0.49)	3144676 (0.28)	8172633 (0.72)	3.93	1.63	2.27
glimpse	464982 (0.75)	151102 (0.25)	155383 (0.60)	104167 (0.40)	2.96	1.50	2.37
tpcc	4576886 (0.77)	1339750 (0.23)	1955964 (0.64)	1096572 (0.36)	2.33	1.24	1.94

Table 4.1 : Decomposition of DRAM Accesses, DRAM Activations, and Row Buffer Hit Count by Access Type. SPEC applications are sorted in decreasing order of WR in DRAM Accesses.

---

## **DRAM Energy and Performance Optimizations**

---

### **5.1 Background: Eager Writeback**

In certain applications that process huge incoming data streams, such as 3D graphics and multimedia, the overall performance tends to be bounded by the efficiency in processing incoming stream data. However, in these applications, the writeback traffic often competes for limited memory bandwidth with traffic for demand reads, delaying the delivery of the stream data to the cache. Specifically, in these applications, data fetches into the cache cause many conflict misses and have dirty cache lines evicted to the next level of memory hierarchy, while consecutive reads of a data stream are in progress. This competition often makes it difficult for the applications with enormous data streams to fully utilize the available memory bandwidth for reads. For this problem, unfortunately, even a large write buffer does not help a lot.

Eager writeback was first proposed to reduce the likelihood of writing dirty cache lines that impede data fetches into the cache [17]. The key approach is writing dirty cache lines to DRAM before they are evicted by cache replacement, especially when the bus is idle. When the cache lines are eagerly written back, their dirty bits are cleared. If these cache lines are re-written before eviction and thus marked dirty again, data writes to DRAM are seen more frequently. However, this will never impact the correctness of architectural

state because it is nothing but writing “premature” data. For the best triggering of eager writebacks, only cache lines that have been dirty and reach the LRU (Least Recently Used) state can be considered as prime candidates for writing back because they rarely get dirty again before being evicted.

Eager writeback is an effective way that decouples and distributes the two competing traffics by selectively writing some dirty cache lines to DRAM prior to writeback, but later than write-through. Therefore, it can be considered as a compromise between write-through and writeback policies. In the scheme, “any” of dirty cache lines in the cache can be eagerly written to DRAM if the bus is “idle”. These conditions, however, are not taken into account if eager writeback is used for improving row-level access locality.

## 5.2 Row Activation Reduction

We utilize eager writeback as a way to reduce the number of row activations. The basic idea is to write back dirty cache lines ahead of time when their associated row has been activated by other ordinary, non-eager access. This will enable better clustering of accesses, both read and write, that target the same row. By increasing the clustering of reads and writes together, there will be fewer row activations and less write interference.

In order to preserve cache behavior, lines that are speculatively written back are not actually evicted from the cache. Therefore, the overall cache hit rate should remain largely unaffected (minor timing variations due to the rescheduling of DRAM accesses could cause minor changes). If those cache lines do not subsequently become dirty again, then their later eviction will not cause a write to the DRAM, thereby reducing both row activations due to writes and write interference.

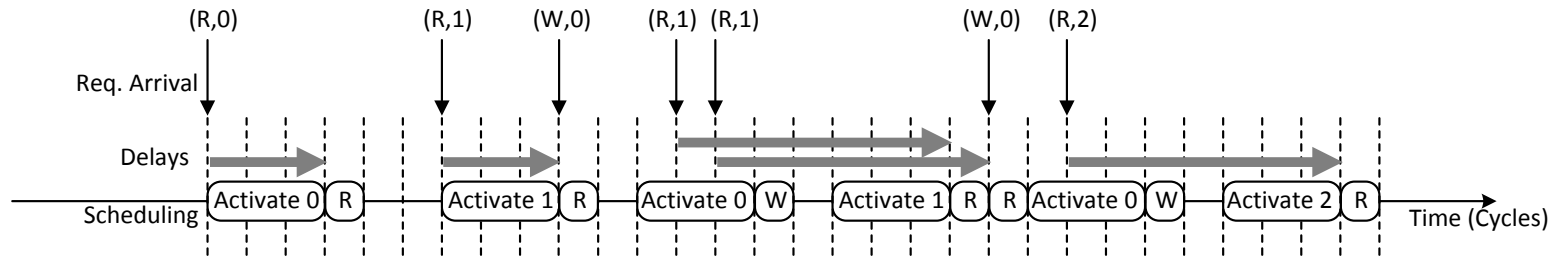
This strategy improves the performance of both reads and writes in the DRAM by effectively clustering read and write accesses to the same row. This increases the overall

number of DRAM accesses per activated row, and consequently results in both improved performance and reduced energy consumption in the memory system, as will be discussed in Section 6.3.

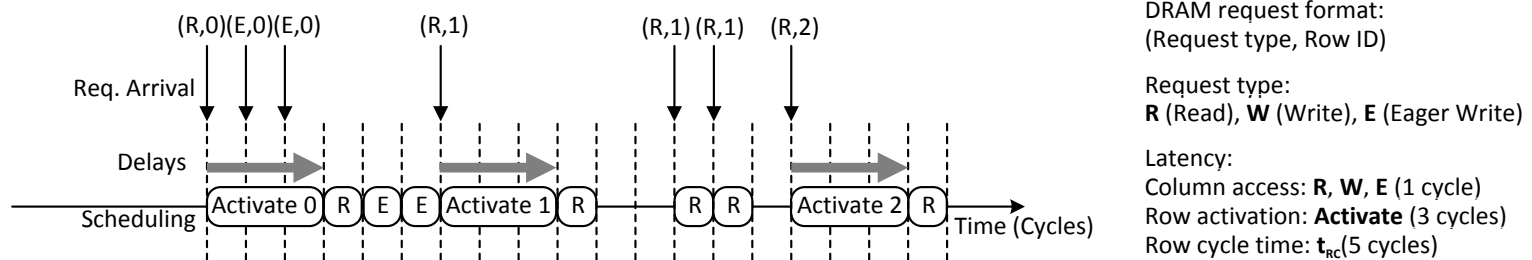
### 5.2.1 Illustrative Example

Figure 5.1 shows how the memory access pattern in a conventional DRAM system is reshaped when using our proposed eager writeback strategy. Figure 5.1(a) shows the impact of writes to the DRAM. The two writebacks to row 0 cause interference in the DRAM, thereby delaying several read accesses and reducing the number of accesses per row activation. The first writeback causes a problem because the cache line eviction occurs before the subsequent read to the currently open row. As the memory controller cannot know that such a read will arrive, it activates the row needed for the writeback operation. The second writeback causes a problem because the cache line eviction causes a row activation which delays the activation of a different row for a subsequent read operation. Note that in both cases, a row is activated to perform a single write operation. The situation is even worse in reality because the timing constraints of the DRAM (specifically  $t_{RC}$ ) force additional delay before the activation for the following read operation can occur.

Given that row 0 was already activated to satisfy the first read operation, eager writeback can be used to speculatively perform the two writeback operations, as shown in Figure 5.1(b). In this case, the two eager writebacks occur speculatively while row 0 is active, and then do not later interfere with any of the following read operations. This results in better access clustering and DRAM efficiency. There are three accesses for each of the first two row activations. In the first case, one read and two writes are clustered, and in the second case, three reads are clustered. This clustering also mitigates the delays caused by the timing constraints of the DRAM, because fewer row activations need to be performed



(a) Memory access pattern without eager writeback



DRAM request format:  
(Request type, Row ID)

Request type:  
**R** (Read), **W** (Write), **E** (Eager Write)

Latency:  
Column access: **R, W, E** (1 cycle)  
Row activation: **Activate** (3 cycles)  
Row cycle time:  $t_{rc}$  (5 cycles)

(b) Memory access pattern with eager writeback

Figure 5.1 : Reshaped Memory Access Pattern with Eager Writeback.

to satisfy the stream of memory references.

In this example, the DRAM timings are simplified to make the DRAM writeback problem easier to understand. A precharge/activate is assumed to take 3 memory cycles and a column access is assumed to take a single cycle. The row cycle time ( $t_{RC}$ ) is 5 cycles, so two read or write operations are needed to avoid additional delays. Further, there is no delay when switching between read and write operations. Under these simplified timings, our eager writeback strategy results in fewer row activations, increased DRAM bandwidth, and decreased average read latency. In practice, with realistic DRAM timings, the reference streams will be more complex and eager writebacks will not always eliminate the subsequent writeback when the cache line is later evicted. However, the savings are very real, as the results will later show.

### 5.3 Architecture Design

Eager writebacks are initiated as soon as the memory controller begins to activate a row. The memory controller then cooperates with the last-level cache controller to find dirty cache lines belonging to the activated row. Since this does not affect the LRU status of the cache lines, it does not compromise any benefit that comes from cache replacement optimizations.

The primary architectural modification to support our eager writeback strategy is that the last level cache controller must be able to find all dirty cache lines that are from currently activated rows in the external memory system. The memory controller can easily transmit the row address to the last level cache whenever it activates a row. The cache must then send the appropriate speculative writes back to the memory controller.

In order for this to work, the cache hit logic must be modified. Each row activation is likely to trigger only a small number of eager writebacks. So, the cache controller must

access the cache a handful of times using the row address to find dirty cache lines. A cache line will be selected for eager writeback if it, (1) matches the given row address, (2) is dirty, (3) is in a particular LRU position, and (4) has not already been speculatively written back. In order to satisfy condition (1), the cache hit logic must be modified to enable matching only on the tag bits that correspond to the DRAM row address. This is a minor modification that requires the hit logic to support two modes, normal full tag matches and partial row matches. Conditions (2) and (3) can easily be checked by reusing the cache line eviction logic. In most cases, it is sufficient to only perform eager writebacks on cache lines in the LRU position, as the results will later show.

Unless it is guaranteed that the memory controller will always perform the speculative writeback, condition (4) requires that each cache line need the addition of a single bit, called the “eager” bit. This bit would normally be cleared for all cache lines. Once a cache line is selected for eager writeback, the “eager” bit would be set. This would prevent the same cache line from being returned twice from a partial row match.

Using the above modifications, when the last level cache controller receives a row address from the memory controller, it can simply access the cache repeatedly with the row address until there is a miss or the memory controller indicates it is going to activate a different row. These lines can be forwarded to the memory controller for speculative writeback as they are found in the cache. As a modern memory system has multiple channels and banks, the last level cache controller may need to interleave accesses for several active rows.

The “eager” bit can be used to ensure that the cache state is always correct. As a speculative write is not guaranteed to be performed, as will be described later in this chapter, the dirty bit for a cache line cannot necessarily be cleared until the memory controller confirms that the line has been written. If another write occurs to that cache line while the “eager” bit

	apriori	tpcc	glimpse	soplex	lbm	gems	mcf	hmmmer	gcc
1st MRU	99.14	98.39	99.75	99.08	79.89	97.9	98.69	50.91	99.95
2nd MRU	0.51	0.92	0.17	0.62	18.18	0.57	1.14	22.61	0.05
3rd MRU	0.19	0.39	0.06	0.24	1.92	1.51	0.15	26.44	0.00
LRU	0.16	0.30	0.02	0.06	0.01	0.02	0.02	0.04	0.00

Table 5.1 : Fraction of Writes to Dirty Cache Lines with Respect to the LRU-MRU Position.

is set, it should be cleared. When the confirmation comes back from the memory controller that the speculative writeback has been performed, if the “eager” bit is still set, then both the “eager” and dirty bits will be cleared for the cache line. If instead, the “eager” bit is no longer set, that indicates there has been new data written to the cache line, so the dirty bit is not cleared.

If a speculative writeback is guaranteed to be performed by the memory controller, then the “eager” bit is not necessary and the dirty bit can be cleared as soon as the eager writeback is initiated. However, there is a significant advantage to allowing the memory controller to “cancel” speculative writebacks in order to activate other rows to perform pending read accesses.

As noted before, the dirty cache lines selected for eager writeback can be limited to some number of the least recently used ways in the cache. The most recently used cache lines in a set are likely to be written again, negating advantages gained from eager writeback. Lee *et al.* [17] have shown this trend by analyzing the ratio of the number of times a dirty cache line in the LRU (or MRU) state is written to. The analysis results show that dirty cache lines that reach the LRU state are rarely written again. We observed a similar, but more obvious trend over the applications we used. Table 5.1 presents the results observed



in 9 representative applications.

### 5.3.1 Cache/Memory System Coordination

Figure 5.2 shows a system design for our eager writeback scheme. Eager writeback is initiated by a row activation, which is to be detected by Row Activation Detector (RAD) in the memory controller. RAD informs the last level cache controller the ID of the activated row to trigger eager writeback of dirty cache lines associated to the row. In our design, the physical address of the row is used as the ID for simplicity. The bus used for delivering the row ID to the cache controller is of the same bit width as the memory address bus width. For page interleaving, however, low order bits corresponding to row size can be ignored and taken off from the row ID to reduce the bus width.

Upon the receipt of a row address at the cache, Eager Writeback Manager (EWM) creates a new entry in its Eager Writeback Queue (EWQ). By managing EWQ, EWM keeps tracking concurrent eager writebacks in progress at multiple DRAM banks. EWQ is a FIFO queue that is dynamically growable, and each queue entry contains the address of cache line to examine for eager writeback for an activated row. A new entry is initialized with the row address, *i.e.* the address of the first cache line of the row, and if it is scheduled, then the entry is updated to the address of the next cache line. The size of EWQ is determined by the number of DRAM banks given that there is at most one activated row per bank. For example, assuming a 8-bank 8-rank DRAM system, EWQ contains at most 64 entries. This is a small overhead in hardware.

While eager writebacks for multiple banks are served, EWM needs to interleave accesses for several active rows to fully exploit bank-level parallelism. This can simply be implemented by scheduling EWQ entries in a round robin fashion. After pulling an entry at the head of the queue, EWM coordinates with Eager Writeback Arbiter (EWA) to find

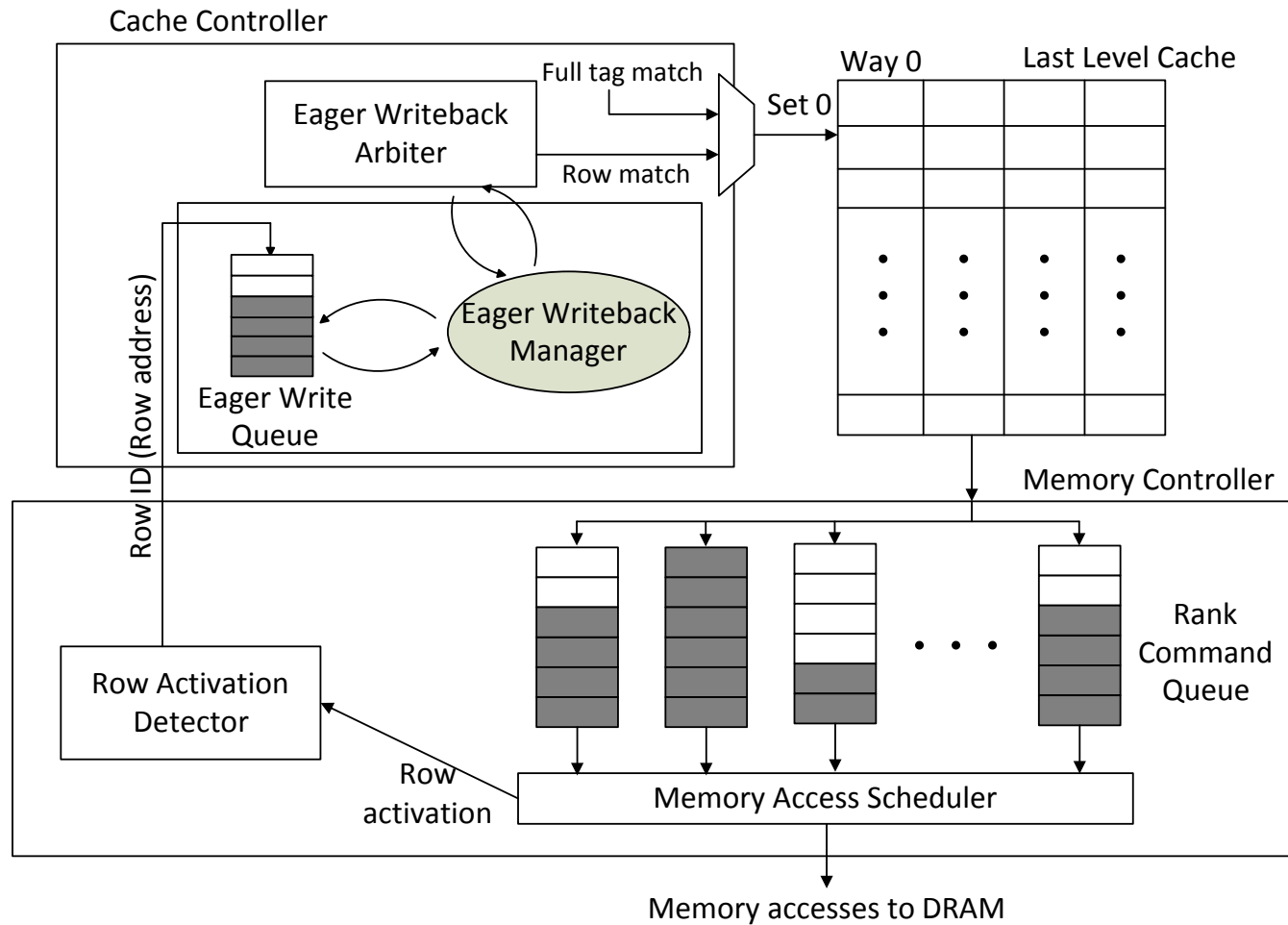


Figure 5.2 : A coordinated Cache/Memory System.

the cache line to eagerly write back. EWA monitors cache bus utilization to find idle cycles and, if found, it processes the request. Once completed, EWM will generate the next cache line to process and, if the cache line is still mapped to the same row, EWM will put it into the tail of the queue.

Note that EWM needs to be fully aware of the address mapping policy of the DRAM because data stored in a row is not always recognized as contiguous in the physical address space. Consider an address mapping where adjacent cache line accesses are striped across different channels so that streaming bandwidth can be sustained across multiple channels. Under the mapping, contiguous cache lines in the row are not contiguous in the physical address range. In this case, generating the physical address for a certain cache line in a row must consider bit fields encoded for channel selection to retrieve the exact address of the next cache line in the row. Because the address mapping is known at system boot time, this address generation is not a significant constraint.

Note also that a dirty cache line may be evicted while its associated speculative eager writeback is still in the memory controller. The memory controller should detect this situation and remove the eager writeback in favor of the actual eviction. Of course, this is a performance optimization that will not affect correctness.

## **5.4 Memory Access Scheduling**

The proposed eager writeback technique often results in a memory access pattern in which an initial read access to a row is followed by a burst of other reads and writes to the same row. When there are a lot of dirty cache lines, this access pattern may keep the row open for quite a long time, delaying pending reads to other rows in the same bank. Therefore, there is a trade-off between the value of reducing write interference by performing many eager writebacks and the value of favoring pending read accesses in order to minimize their

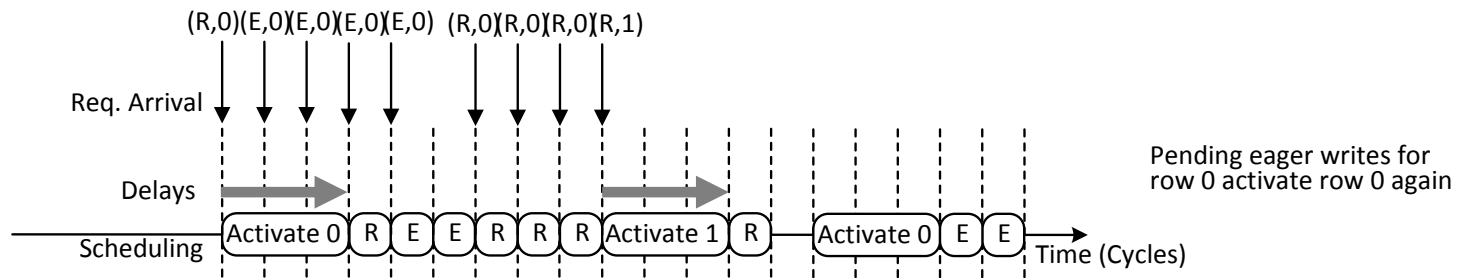
latency.

It is difficult to evaluate this trade-off, as the resulting performance in either case depends on several factors, including the degree of write interference, the update rate to cache lines that have previously been speculatively written back, and the average number of writes that delay pending reads. For some benchmarks, such as the *apriori* data mining benchmark, performing eager writeback in this manner can degrade performance because too many reads accesses are delayed for too long. Note, however, that even in this case, energy is still saved because of the large reduction in row activations.

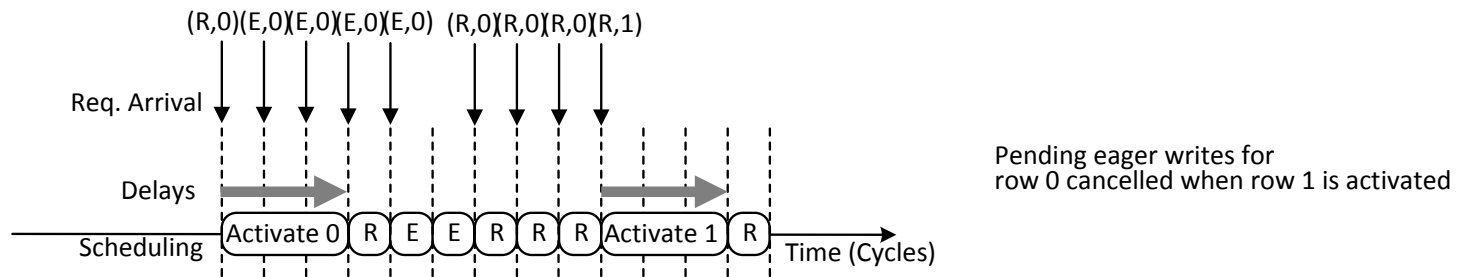
To address the problem of delayed read accesses, while maintaining the benefits of speculative eager writeback, speculative writebacks must be handled specially by the memory controller and it must be possible to cancel them in certain situations. In combination, this addresses the delayed read access problem, thereby improving the value of the speculative eager writeback technique.

Speculative writeback-aware scheduling (SWAS) differentiates between a write caused by a normal cache eviction and a write caused by a speculative writeback. The baseline memory access scheduler is based on a column-first scheduling policy [1]. Effectively, this means that the scheduler favors column accesses first, row activations second, and precharge operations third. This maximizes the number of column accesses per row activation. With SWAS, a column access for a speculative writeback is not given first priority with other column accesses, but rather is given fourth priority behind all other DRAM command types. This ensures that speculative writebacks will only be performed if there are no other useful non-speculative operations that can be performed on a particular memory cycle.

If the memory controller activates another row while speculative writebacks remain for the previous row, those speculative writebacks would trigger another row activation later.



(a) Memory access pattern with only SWAS



(b) Memory access pattern with SWAS and cancellation

Figure 5.3 : Scheduling Eager Writeback Operations. See Figure 5.1 for request type and latency information.

While that may still be useful, it is often a better choice to *cancel* all speculative writebacks to a row once that bank is precharged. That way, they will either be re-issued speculatively if the original row is ever re-activated by a read operation or they will simply occur when the cache lines are finally evicted, as if no eager writeback had been attempted.

Figure 5.3 shows how the SWAS/cancellation coordinated system allows eager writeback to better cluster reads and writes without unnecessarily delaying subsequent non-speculative operations. In Figure 5.3(a), the activation of row 0 triggers four eager writeback operations. When read operations later arrive, they push back the eager writeback operations. As the figure shows, row 0 must be re-activated later in order to satisfy all of the eager writebacks. Figure 5.3(b) shows that the delays caused by eager writebacks can be eliminated simply by cancelling them. Note that once cancelled, these writebacks will never be scheduled, so row 0 will not need to be re-activated for them. Eventually, the lines will be evicted from the cache and have to be written back, though. Or they might be eagerly written back in the future if row 0 is activated for another reason before the lines are evicted.

Note that the situation shown in Figure 5.3(a) is not necessarily bad. It potentially saves energy, as the number of column operations to row 0 is higher. The trade-off between additional accesses to row 0 and the latency of the read to row 1 is difficult to evaluate in a situation like this. The value depends on what happens to the writes later. The effectiveness of cancellation is mainly determined by how many useful writes are processed before being canceled—here “useful” writes are those not having further updates to their cache lines and thus not generating “redundant” DRAM writes by either repeated eager writeback or cache eviction. If eager writes are organized in an ordered way so that more useful writes come first and less useful writes come next, cancellation may be more effective.

## 5.5 Conclusions

DRAM performance and energy consumption in modern systems are significantly impacted by the efficiency of managing the row buffer. Unfortunately, applications cannot make an effective use of data in the row buffer because of the lack of the low-level access locality. An analysis in the previous chapter showed that cache line writebacks are the root cause of this lack of utilization. This chapter has presented a new approach to improving the low-level access locality that is based on clustering writebacks with reads that will hit in the same row. We used eager writeback for this optimization. While this thesis focuses a simple DRAM configuration that is composed of single memory channel and single rank in a DIMM, the results should apply generally to other configurations.

Since Modern DRAM timing is quite complex and there are many constraints of the DRAM, this chapter has extensively explored several dimensions in the design of our eager writeback to avoid potential performance degradation. Our eager writeback scheme incorporates two mechanisms, Speculative Writeback-Aware Scheduling (SWAS) and Cancellation, to prevent eager writebacks from adversely affecting a program's execution time. These two optimizations make certain that the scheme does no harm. In addition, the scheme incorporates a less aggressive configuration that writes back one or two least recently used cache lines from the set. Examining these cache lines is sufficient to achieve overall good results without incurring much overhead.

---

## Evaluation

---

### 6.1 Full-System Simulator

We use a full-system simulator to evaluate eager writeback. This simulator is based on HP Labs' COTSon [50], which we have extended to simulate a cycle-accurate DRAM system using DRAMSim2 [51]. COTSon uses AMD's SimNow [52] to emulate an x86-64 processor and to capture all instructions that are executed for the operating system and user-level processes. For each instruction, the opcode, the registers accessed, and the instruction and data addresses accessed are fed to the COTSon processor module for detailed simulation. This simulation models an out-of-order processor using a 256-entry re-order buffer (ROB), in-order issue and out-of-order execution, comprehensive register- and memory-dependency checking, and blocking reads. The simulator does not model implementation-dependent parameters, like the bandwidth between pipeline stages and the number of outstanding cache misses, that do not directly affect our evaluation of eager writeback. Table 6.1 shows the important parameters of the simulated system.

The timing model in COTSon is modified so that a cache hit at any level of the memory hierarchy adds zero latency to an instruction's execution. This allows us to focus on the impact of the DRAM system. Last-level cache misses are sent to DRAMSim2, which accurately models all characteristics of the external memory system, including the state



Processor ISA	AMD Family 10h
Re-Order-Buffer	256
Pipeline	2.4 GHz, in-order issue, out-of-order execution, out-of-order write
L1 caches	16 KB Inst/16 KB Data, 2-way, 64 Bytes line size, write-through, no write-allocate
L2 cache	512 KB, 4-way, 64 Bytes line size, write-back, write-allocate
Memory Configuration	2 DIMMs/channel, 1 rank/DIMM, 8 devices/rank, 8-bit output/device, 64 bit channel
DRAM Device Parameters	Micron MT41J128M8 DDR3-1600 [2] Timing Parameters $t_{RCD}-t_{RP}-t_{CL} = 10-10-10$ (12.5 ns) 8banks/device, 32768 rows/bank, 512 columns/row, 4 KB row buffer
Total DRAM Capacity	1 GBit/device $\times$ 8 devices/rank $\times$ 2 ranks = 2 GB

Table 6.1 : Processor and DRAM System Parameters.

of all channels, ranks, banks, and rows, in a cycle-accurate manner. The modified timing model in COTSon ensures that an instruction only blocks when either the reorder buffer is full or there is a memory dependence. As the external memory system is cycle-accurate, these memory dependencies accurately model the delays that would be incurred by DRAM latency.

All configurations use column-first scheduling, single-channel address mapping, page interleaving, and open page mode for DRAM (with 1Gb x8 devices). A single channel with one rank per DIMM is simulated for simplicity. As discussed in Section 5.4, memory access scheduling is performed using a column-first scheduler [1] modified to give speculative eager writebacks low priority. Detailed timing and power parameters for the memory devices are based upon the Micron MT41J128M8 DDR3-1600 datasheet [2]. Energy and power are calculated using Micron’s power calculation methodology, assuming x8 DRAM devices [53].

## 6.2 Applications

We use a wide variety of applications, including Glimpse [54], TPC-C, and a large number of applications from the MineBench [55], SPEC CFP, and SPEC INT benchmark suites [56]. Glimpse is a text indexing and search application. We use a collection of program source and text files, whose total size is 2.3 GB, as the input to Glimpse. TPC-C is a distributed, on-line transaction processing (OLTP) benchmark specification. We use an open-source implementation of TPC-C, TPCC-UVa [57]. MineBench is a data mining benchmark suite. We use *apriori*, an association rule mining application, on a large dataset. For each of these applications, we simulate 0.5 billion instructions. To eliminate the initialization phase from our simulations, we apply appropriate fast-forwarding to each application. We have also performed some simulations with a larger number of instructions and they have shown similar results.

## 6.3 Simulation Results

The evaluation of our eager writeback scheme proceeds as follows. We first evaluate a representative eager writeback configuration that performs SWAS and cancellation and that examines two LRU-side cache lines within a set. This configuration is referred to as *Eager WB* throughout this section. We then evaluate the different features of our scheme individually in order to understand their impact on the overall results. Specifically, we evaluate the effects of varying the number of cache lines that are considered for eager writeback and of disabling SWAS and cancellation. Finally, we look at the effects of combining eager writeback with a state-of-the-art DRAM address interleaving optimization, XOR-based address mapping [18]. Unless stated otherwise, the performance and energy consumption results for each application are normalized to the results for the baseline system, *No Eager WB*,

that does not perform eager writeback.

### 6.3.1 Comparison to The Baseline

Since a single write access has high possibility of generating a row activation, as discussed in Section 4.1, its effectiveness largely depends on the fraction of write accesses. If more write accesses are issued in running an application, more row activations can be reduced through eager writebacks. Moreover, if so, the damage done by write interference can be more reduced.

Table 6.2 presents a breakdown of the row activations with No Eager WB and Eager WB for each of the applications. Row activations are broken down into three categories: DRAM read (RD), write interference (RD Interfered), and DRAM write (WR). Row activations in both RD Interfered and WR categories represent potentially destructive row activations and can possibly be eliminated by our eager writeback scheme.

The fractions in all of the columns are relative to the total number of row activations in No Eager WB. For example, 17% of all activations in *libquantum* are due to DRAM reads. In Eager WB, row activations due to DRAM reads increase to 18% of the number of activations in No Eager WB. However, the total number of row activations with Eager WB is only 42% of the total number of activations with No Eager WB. Therefore, the lower the “Total”, the more effective our eager writeback scheme is at reducing row activations.

With Eager WB, the total number of row activations is reduced by an average of 38% and a maximum of 81%. Moreover, the total number of row activations is never increased for any of these applications. Specifically, the number of row activations in the RD Interfered category is reduced from 14% to 2% of the total with No Eager WB. With Eager WB, the average number of row activations by DRAM writes, which is the sum of WR and Eager WR, accounts for only 10% of the average number of all row activations.

This is a significant reduction from the 36% observed with No Eager WB. However, some applications, such as *sjeng*, *gamess*, *bwave*, and *omnetpp*, show no more than a 10% reduction in the total number of row activations. Except for *sjeng*, these are the applications that perform the fewest DRAM writes.

The remaining destructive row activations in Eager WB (from RD Interfered and WR) occur largely because of cancellation. When higher priority read accesses arrive, pending eager writebacks are cancelled and therefore occur later when the dirty cache line is evicted, often causing row activations at that point.

Table 6.3 presents the average number of accesses to an activated row with No Eager and Eager WB for each of the applications in the first two columns. The third column (Improved) shows the number that the results in Eager WB are normalized to the results in No Eager WB. This shows the relative improvement in utilizing activated rows. For example, an activated row in *lbm* was accessed an average of 1.83 times with No Eager WB. With Eager WB, it increases to 9.58, meaning that there is 5.23 times of improvement in accessing data in the activated row. Therefore, the higher the Improved, the more effective our eager writeback scheme is at clustering reads and writes to the same row. With Eager WB, 9 applications improve accesses to the row more than twice.

There are two factors that limit this improvement in Eager WB. First, Cancellation itself is a mechanism that limits performing many accesses per row activation. Second, the physical address is mapped into DRAM banks following the direct-mapped scheme, and the degree of bank-level parallelism in the memory is relatively small. So, there are still frequent conflicts between DRAM accesses within the bank that interchange rows in the row buffer frequently. We observed that to promote accesses on the row, releasing strict constraints placed upon Cancellation works better than having more bank-level parallelism.

Because the reduction in row activations is considerable, many of the applications

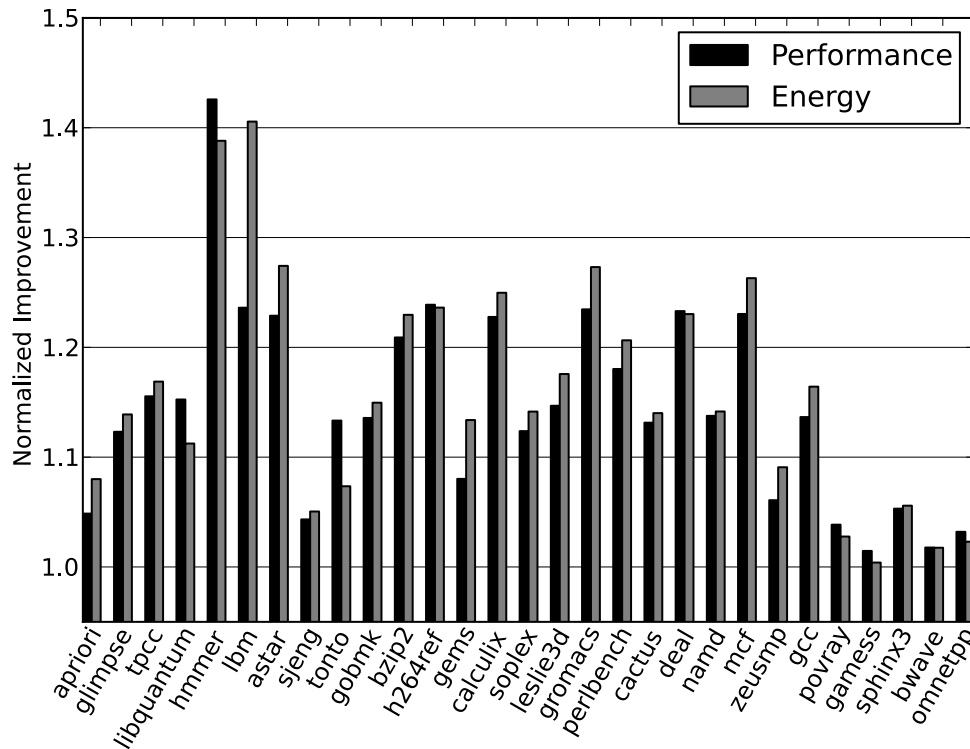


Figure 6.1 : Performance Improvement and Energy Savings for 29 Applications (Normalized to The Results of No Eager WB). The larger number means the more optimized.

achieve significant performance improvements and energy savings, as illustrated in Figure 6.1. Specifically, out of the 29 total applications, 11 have overall performance improvements between 10% and 20%, and 9 have improvements in excess of 20%. Moreover, 10 consume between 10% and 20% less DRAM energy, and 10 have energy consumption reductions in excess of 20%. One interesting, also expected, trend is that applications for which row activations are significantly reduced tend to gain remarkable benefits. For example, *lbm*, *mcf*, *hmmer*, *gromacs*, *deal*, *calculix*, and *h264ref*, for which the reduction in row activations is larger than 50%, achieve 20% or more improvement in both performance and energy consumption. Conversely, applications with minor reductions in row activations

(e.g. *sjeng*, *gamess*, *bwave*, and *omnetpp*) achieve performance improvements and energy reductions of no more than 5%.

### 6.3.2 Varying the Number of Cache Lines Written Back

Determining appropriate cache lines used for eager writeback is critical. If many useless cache lines are included, resources in DRAM, such as bandwidth and cycles, will significantly be wasted—these are eager cache line writebacks where the cache line is written to again before it is evicted, and thus the cache line must be written again to DRAM. On the other hand, if too few useful cache lines are included, it is difficult to expect to achieve substantial improvements. One simple metric that can be used to control these properties is the position in the set-associative cache [17] where cache lines are located. As discussed in Section 5.3, dirty cache lines located nearer to the least recently used position are do not tend to be written again. Therefore, these cache lines are more useful than the cache lines that are most recently used. But, note that not all cache lines at the most recently used are useless.

Figure 6.2 and 6.3 show the effect on performance and energy consumption, respectively, of varying the number of least recently used positions in a cache set that may be eagerly written back. In these figures, Eager WB-N denotes a configuration that may write back the N least recently used cache lines from each set. Thus, in these figures, the configuration that we evaluated in Section 6.3.1, Eager WB, is renamed Eager WB-2.

As compared to Eager WB-2, more aggressive configurations, such as Eager WB-4, that may write back more cache lines from a set achieve relatively worse overall results. While for some applications the more aggressive configurations achieve slight improvements, for many other applications they achieve significantly worse results. The explanation for this has two aspects. First, with the more aggressive configurations, *useless* cache line write-

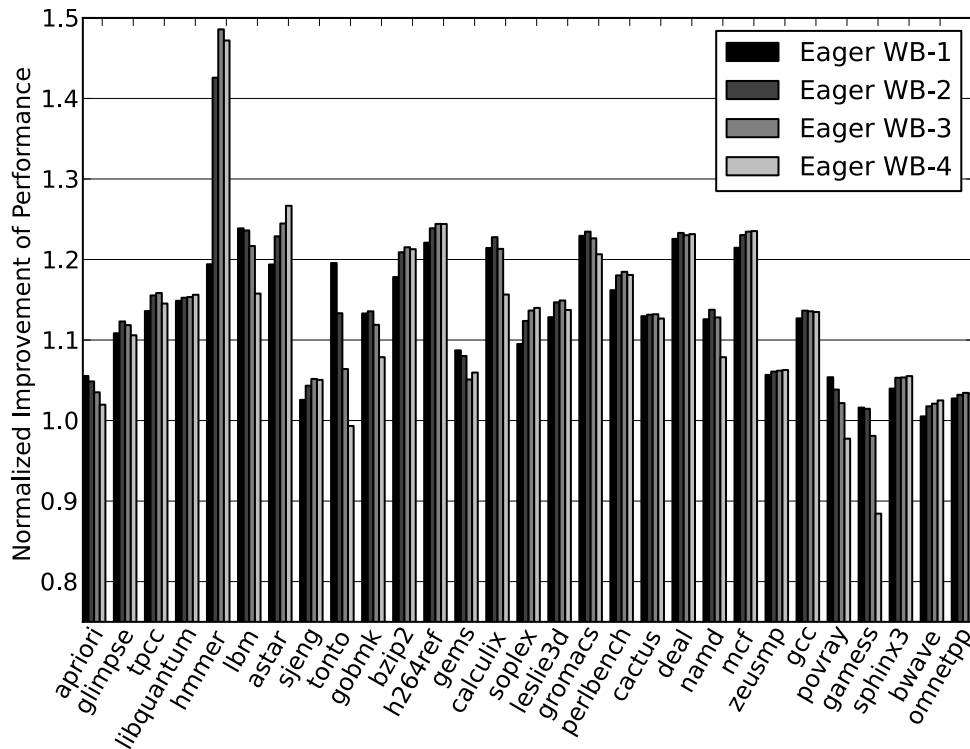


Figure 6.2 : Performance Improvement with Different Cache Lines for a Eager WB (denoted as Eager WB-N for using N LRU-side lines for a speculative writeback issue).

backs are more likely to occur. Second, cancellation is more likely to occur because of the larger number of cache lines that the more aggressive configurations are writing back. Thus, many useful cache line writebacks are cancelled, leading to later DRAM writes when the cache lines are evicted.

The situation in Eager WB-4 will be much better if eager writebacks for a row are issued in an ordered way that *useful* cache lines are written back before *useless* ones. These eager writebacks are issued in a way that those dirty cache lines in the most LRU position are written first, those in the second most LRU position are written next, and so on. In this approach, eager writebacks that are not likely to be written again will be scheduled earlier.

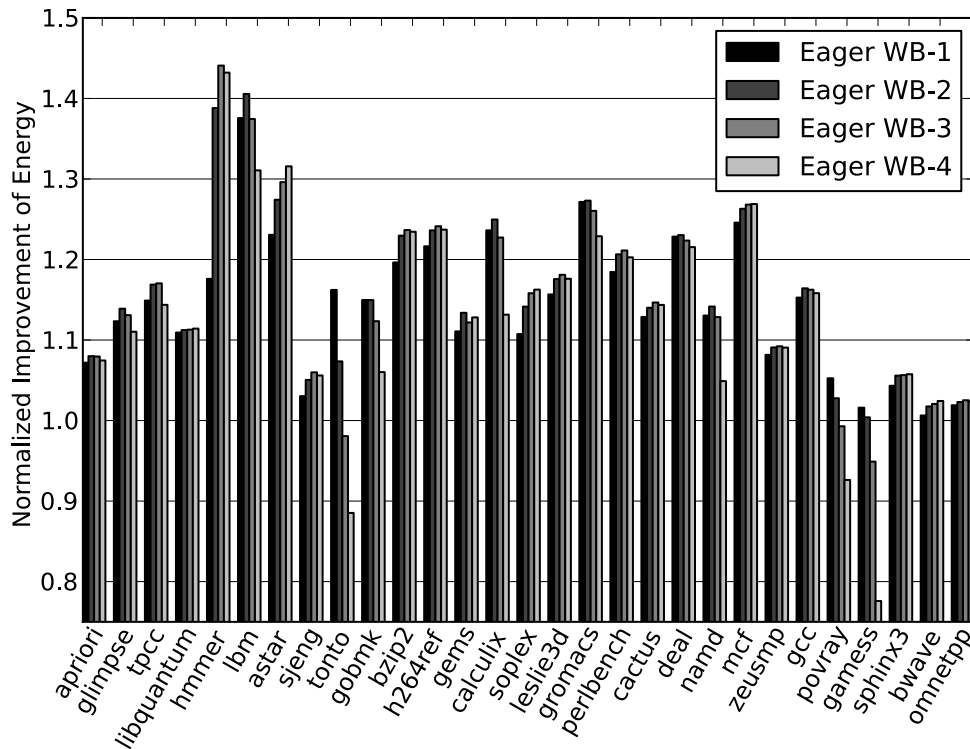


Figure 6.3 : Energy Savings with Different Cache Lines for a Eager WB (denoted as Eager WB-N for using N LRU-side lines for a speculative writeback issue).

Therefore, eager writebacks that are scheduled until the cancellation tend to be more *useful* compared to the unordered issues. This will result in having less cache lines that must be written again to DRAM because of more processing of useful cache line writebacks. We expect that this approach will at least perform as good as other good configurations, such as Eager WB-2.

Surprisingly, the least aggressive configuration, Eager WB-1, achieves overall results that are almost as good as Eager WB-2. In fact, *games*, *povray*, and *tonto* achieve the most energy savings and performance improvement with Eager WB-1. For these applications, as compared to the others, data writes are more likely to hit in the cache, cache lines are less



likely to be evicted, and DRAM writebacks are relatively infrequent. So, more aggressive configurations simply result in more useless cache line writebacks.

### 6.3.3 Evaluation with SWAS and Cancellation Disabled

Using SWAS and Cancellation allows read accesses to be scheduled as early as possible. However, without these optimizations, clustering of reads and writes is maximized, as all eager writebacks will always complete, keeping rows open for a longer period of time and performing more accesses per row activation. In contrast, the use of SWAS and Cancellation reduces the average read latency, but potentially incurs additional row activations. These optimizations are complementary and perform much better together than alone.

Figure 6.4 and 6.5 evaluate the trade-offs with these optimizations. “SWAS-Can-Dis” shows the performance improvements and energy savings relative to No Eager WB when the SWAS and Cancellation optimizations are disabled. In contrast, “SWAS-Can-En” shows the improvements over No Eager WB when the optimizations are enabled. As the figure shows, our eager writeback scheme provides improvements both with and without SWAS and Cancellation for all benchmarks except *apriori*. *Apriori* under SWAS-Can-Dis performs 25% more writes than the baseline due to useless cache line writebacks. It also has a high row-buffer hit count ( $> 6$ ). Together these result in the situation that reads are often delayed by earlier speculative writes, some of which are also useless writes. Therefore, these two optimizations are necessary to make sure that our eager writeback scheme “does no harm”.

Enabling only Cancellation works very similarly to SWAS-Can-Dis, meaning that Cancellation itself has no significant effects. Enabling only SWAS, however, has a potential problem that switches scheduling of eager writebacks and incoming reads back and forth, causing repeated row activations. In the worst case,  $2N$  row activations can occur with  $N$

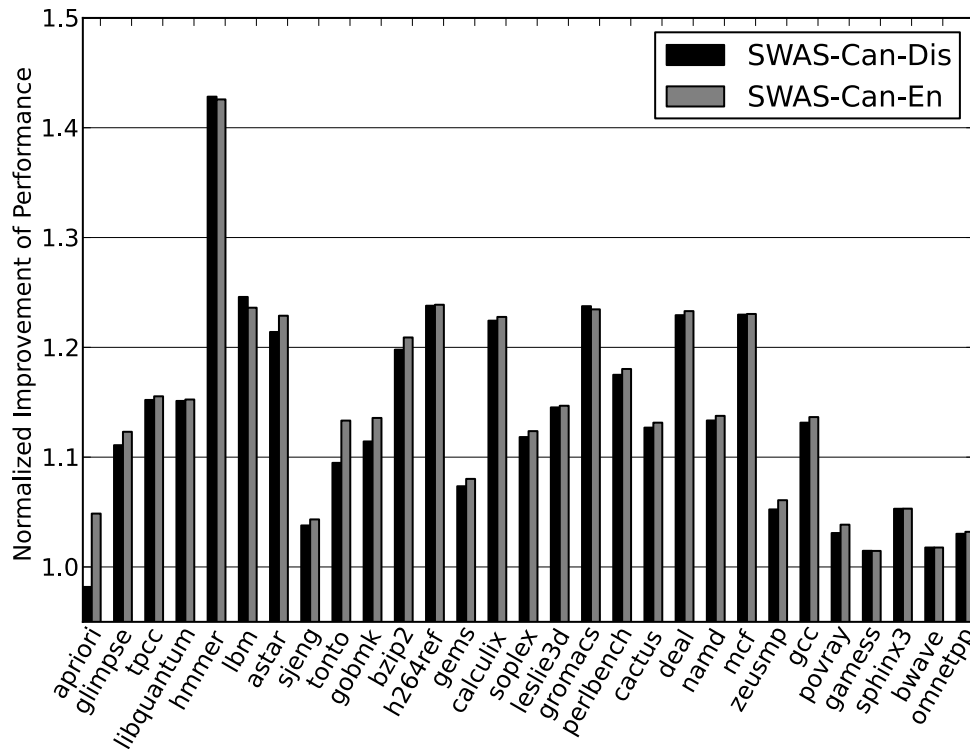


Figure 6.4 : Comparison of Performance Improvement with Results Observed With SWAS and Cancellation disabled.

eager writebacks. This is observed in *apriori* and results in much more energy consumption than SWAS-Can-Dis. Because using SWAS makes reads not significantly delayed though, the performance with only SWAS enabled is very similar to SWAS-Can-En.

#### 6.3.4 XOR-based Address Mapping

XOR-based address mapping, often called *Bank Swizzle Mode*, generates the memory bank index by XOR-ing two portions of the memory address bits. It was proposed as a way of reducing row buffer conflicts between a cache line fill and a cache line eviction within the same set [18]. Specifically, it reduces the likelihood that the fill and the eviction access the

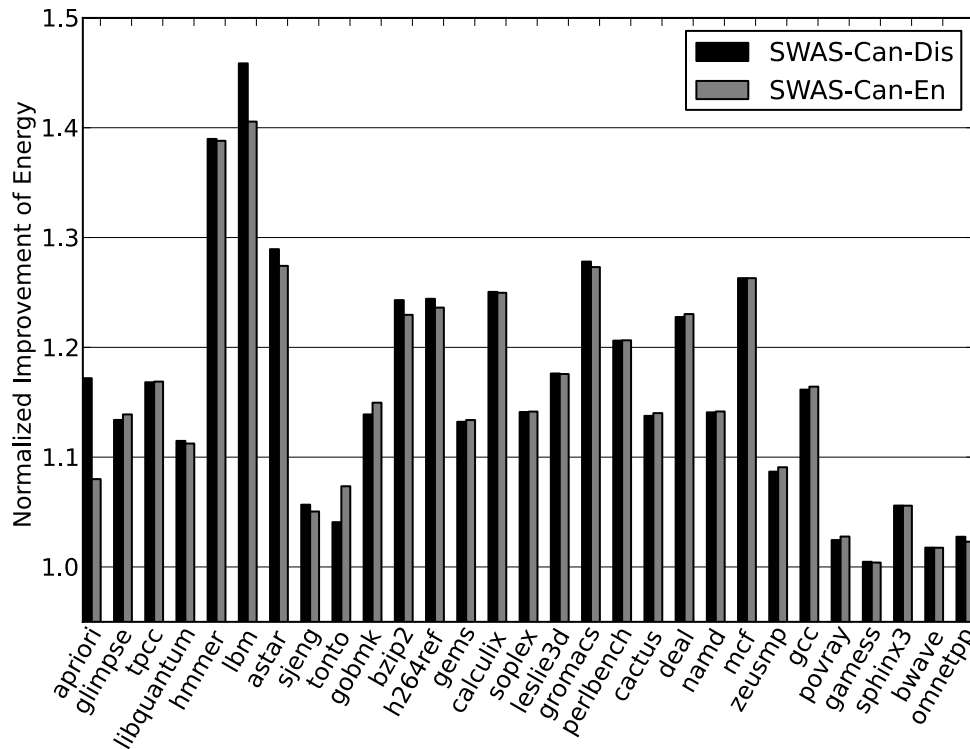


Figure 6.5 : Comparison of Energy Savings with Results Observed With SWAS and Cancellation disabled.

same bank. Hence, it reduces the likelihood of write interference if the system typically has some idle memory banks. XOR-based mapping is supported by many AMD and Intel processors, and can be enabled via the BIOS interface [58].

Figures 6.6 and 6.7 compare the performance and energy savings, respectively, for three configurations: No Eager WB with XOR-based mapping (XOR), Eager WB, and Eager WB with XOR-based mapping (Eager WB-XOR). As always, the results shown in the figure are normalized to the results for No Eager WB. By itself, XOR increases performance and reduces energy consumption considerably. These applications are single-threaded. Consequently, there are often idle banks that XOR can take advantage of. Nonetheless,

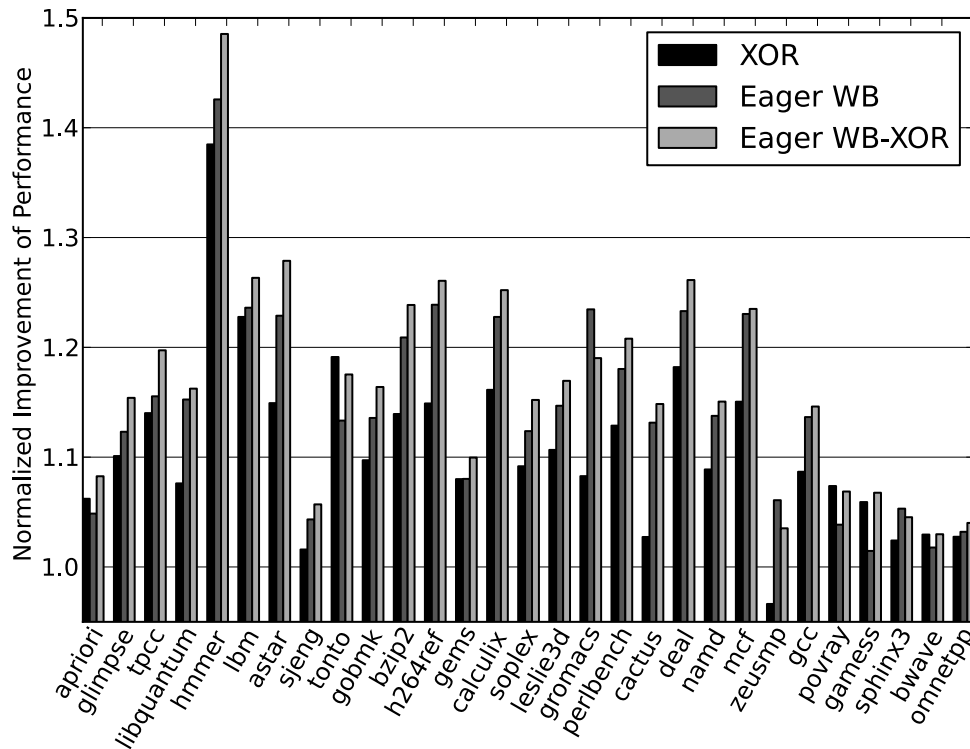


Figure 6.6 : Comparison of Performance Improvement with XOR-Based Memory Mapping (XOR) and with Eager WB on XOR (Eager WB-XOR)

Eager WB usually achieves better results than XOR. Moreover, the combination of eager writeback and the XOR-based mapping, Eager WB-XOR, is usually the best of the three configurations. Like Eager WB, Eager WB-XOR always achieves better results than the baseline, No Eager WB. Moreover, Eager WB-XOR achieves better results than XOR for all but 2 of the 29 applications, *tonto* and *povray*. For those two applications Eager WB-XOR suffers from frequent switching from eager writes to reads across different banks in a rank under our priority scheme.

While XOR-based mapping may reduce the likelihood of write interference, eager writeback actually addresses its fundamental cause, poor row-level locality of access.

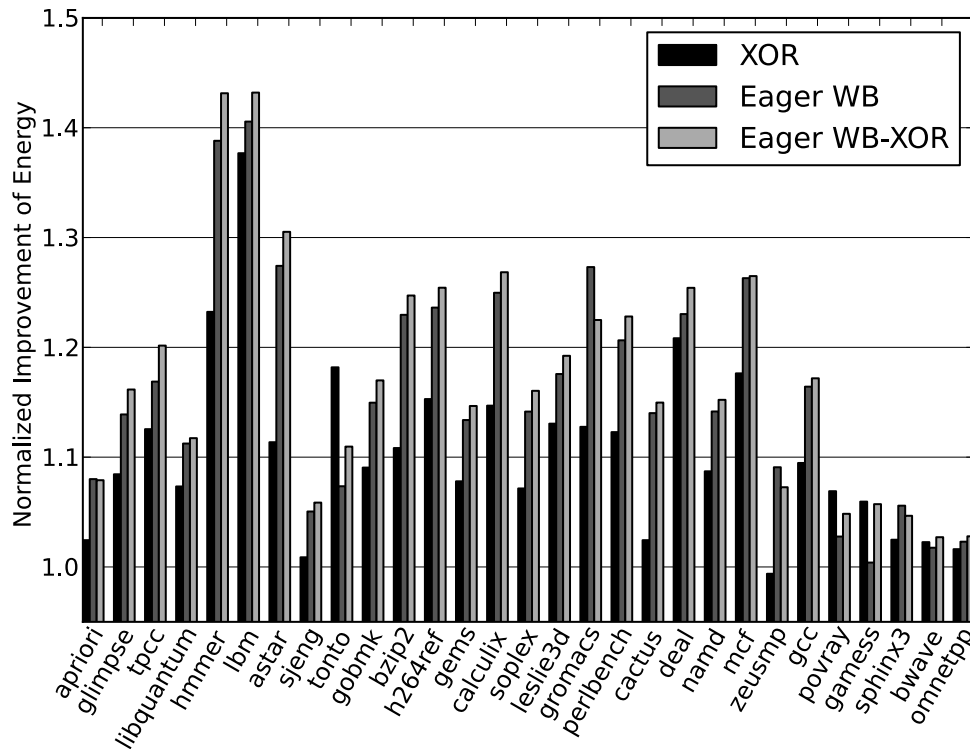


Figure 6.7 : Comparison of Energy Savings with XOR-Based Memory Mapping (XOR) and with Eager WB on XOR (Eager WB-XOR)

Whereas XOR-based mapping simply redirects accesses to different banks, eager write-back aggressively reorders accesses to increase row-level locality. Furthermore, the eager writeback approach is not dependent on the existence of idle banks to increase performance and reduce energy consumption.

### 6.3.5 Overhead

Figure 6.8 presents a decomposition of the DRAM accesses with No Eager WB, Eager WB-1, and Eager WB-2 for each of the applications, which correspond to the first, second, and third stacked bar, respectively, for the application. DRAM accesses are broken down

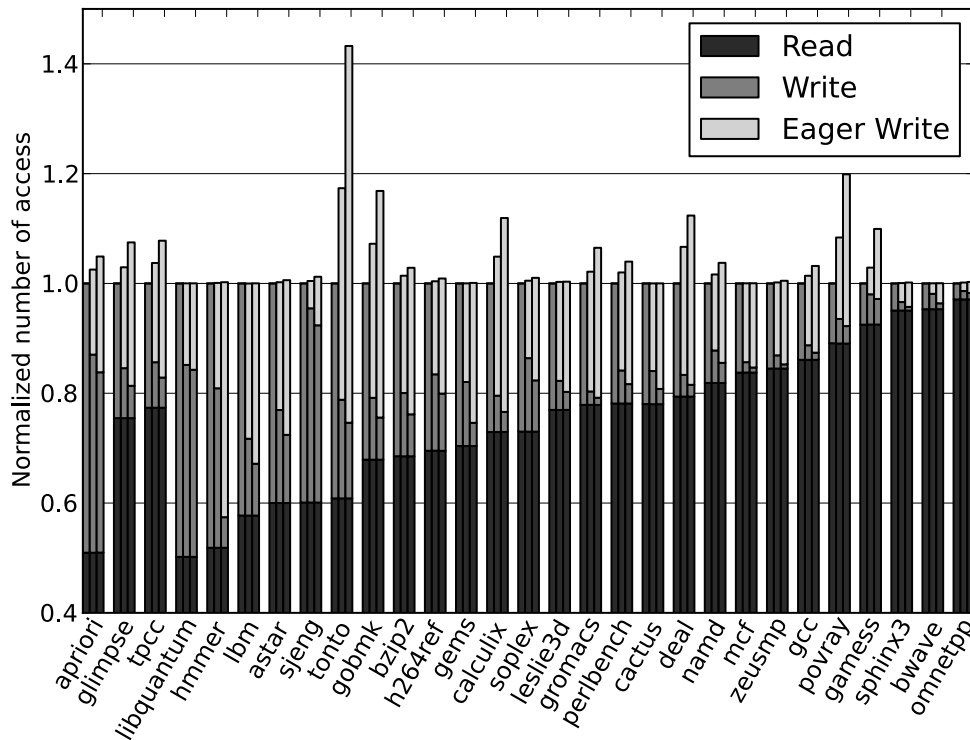


Figure 6.8 : Decomposition of DRAM Accesses by Access Type. No Eager WB, Eager WB-1, and Eager WB-2 correspond to the three stacked bars (in order from left to right) for each application. The fractions in all of the values are relative to the total number of DRAM accesses in No Eager WB.

into three categories: DRAM read (*Read*), DRAM write (*Write*), and eager writeback (*Eager Write*). In the figure, the fractions in all of the values are relative to the total number of DRAM accesses in No Eager WB. For example, 51.9% and 48.1% of all DRAM accesses in *hmmer* are due to reads and writes, respectively, with No Eager WB. In Eager WB-2, DRAM writes decrease to 5.5% of the number of DRAM accesses in No Eager WB, and instead eager writebacks occupy 42.8%. The fraction of DRAM reads remains the same.

This figure specifically illustrates two trends in eager writebacks. First, it shows how effectively DRAM writes are transformed to eager writebacks as moved to Eager WB-2. The lower the “Write” in Eager WB-1 or Eager WB-2, the more effective our eager writeback scheme is at doing so. Second, it shows the overhead of using eager writebacks such that more memory traffic is required with Eager WB-1 or Eager WB-2 due to writing cache lines that will be re-written later. For example, since the fraction of DRAM reads remains unchanged with eager writebacks, there is only a marginal increase in the memory traffic by 0.2% in *hammer*.

The figure shows that our eager writeback scheme is very effective in transforming normal, destructive DRAM writes into eager writebacks for both configurations. Moreover, it shows that our scheme effectively prevents the memory traffic from increasing too much. In fact, if more cache lines from a set are written back, the overhead tends to increase. Therefore, for some applications, such as *tonto*, *gobmk*, *calculix*, *deal*, and *povray*, our eager writeback scheme increases the memory traffic more than 10%. However, for most of the applications, our eager writeback scheme incurs acceptable overhead. Especially, more than half of the applications increase the memory traffic less than 1.0%.

## 6.4 Conclusions

A variety of applications running on a full-system simulator are used to evaluate the effects of our eager writeback scheme on performance and energy consumption. In short, the simulations show that our eager writeback scheme reduces the number of DRAM row activations by an average of 38% and a maximum of 81%. Moreover, our eager writeback scheme produces compelling performance improvements and energy consumption reductions. Out of 29 applications, 11 have overall performance improvements between 10% and 20%, and 9 have improvements in excess of 20%. Furthermore, 10 consume between

10% and 20% less DRAM energy, and 10 have energy consumption reductions in excess of 20%.

In addition, this chapter examines the effects of combining our eager writeback scheme with a well-known DRAM address interleaving optimization, XOR-based address mapping. This optimization is also called *Bank Swizzle Mode*. XOR-based address mapping was originally proposed as a way of reducing row buffer conflicts between a cache line fill and a cache line eviction within the same set [18]. The results show that our eager writeback scheme and XOR-based address mapping generally complement each other. Usually, their combination achieves better results than either one alone.



	No Eager WB			Eager WB			Total
	RD	RD Interfered	WR	RD	RD Interfered	WR	
libquantum	130816 (0.17)	203469 (0.27)	425596 (0.56)	136350 (0.18)	16108 (0.02)	169750 (0.22)	(0.42)
hammer	862322 (0.39)	219710 (0.10)	1101266 (0.50)	870460 (0.40)	5688 (0.00)	151525 (0.07)	(0.47)
lbm	577917 (0.04)	5903593 (0.43)	7317800 (0.53)	532237 (0.04)	971420 (0.07)	1133063 (0.08)	(0.19)
astar	1118966 (0.31)	682640 (0.19)	1841565 (0.51)	1144581 (0.31)	182995 (0.05)	515661 (0.14)	(0.51)
sjeng	189913 (0.57)	2583 (0.01)	139369 (0.42)	189592 (0.57)	2478 (0.01)	112807 (0.34)	(0.92)
tonto	16249 (0.31)	10537 (0.20)	26469 (0.50)	15294 (0.29)	2024 (0.04)	7439 (0.14)	(0.46)
gobmk	252021 (0.46)	66309 (0.12)	234455 (0.42)	251913 (0.46)	17286 (0.03)	68098 (0.12)	(0.61)
bzip2	880371 (0.44)	239285 (0.12)	865982 (0.44)	881132 (0.44)	53769 (0.03)	209472 (0.11)	(0.58)
h264ref	110662 (0.35)	70182 (0.22)	135736 (0.43)	111708 (0.35)	10287 (0.03)	29492 (0.09)	(0.48)
gems	9320385 (0.69)	296588 (0.02)	3804887 (0.28)	9324255 (0.69)	44177 (0.00)	307186 (0.02)	(0.72)
calculix	6214 (0.36)	3756 (0.22)	7405 (0.43)	6270 (0.36)	538 (0.03)	1057 (0.06)	(0.45)
soplex	4126784 (0.59)	553978 (0.08)	2315124 (0.33)	4127887 (0.59)	171281 (0.02)	836009 (0.12)	(0.73)
leslei3d	3000215 (0.62)	626336 (0.13)	1239772 (0.25)	2986233 (0.61)	38073 (0.01)	169415 (0.03)	(0.66)
gromacs	97400 (0.32)	92328 (0.30)	116692 (0.38)	98886 (0.32)	5178 (0.02)	7704 (0.03)	(0.36)
perlbench	508525 (0.45)	198450 (0.18)	410769 (0.37)	510178 (0.46)	32320 (0.03)	70545 (0.06)	(0.55)
cactus	2479443 (0.76)	48431 (0.01)	742433 (0.23)	2473102 (0.76)	3357 (0.00)	95701 (0.03)	(0.79)
deal	82642 (0.24)	110631 (0.33)	144297 (0.43)	82739 (0.25)	12448 (0.04)	17596 (0.05)	(0.33)
namd	11407 (0.44)	4682 (0.18)	9665 (0.38)	11404 (0.44)	727 (0.03)	2274 (0.09)	(0.56)
mcf	3413153 (0.29)	3771163 (0.32)	4646697 (0.39)	3432441 (0.29)	208079 (0.02)	285611 (0.02)	(0.33)
zeusmp	990150 (0.80)	31475 (0.03)	221578 (0.18)	990734 (0.80)	3232 (0.00)	12371 (0.01)	(0.81)
gcc	29709 (0.57)	8179 (0.16)	14570 (0.28)	29773 (0.57)	722 (0.01)	1417 (0.03)	(0.61)
povray	1983 (0.64)	339 (0.11)	758 (0.25)	1986 (0.64)	112 (0.04)	244 (0.08)	(0.76)
gamess	2379 (0.77)	160 (0.05)	552 (0.18)	2446 (0.79)	91 (0.03)	348 (0.11)	(0.93)
sphinx3	2935051 (0.87)	148208 (0.04)	278787 (0.08)	2936898 (0.87)	17585 (0.01)	41168 (0.01)	(0.89)
bwave	29960493 (0.96)	158822 (0.01)	963946 (0.03)	29962832 (0.96)	10976 (0.00)	322769 (0.01)	(0.97)
omnetpp	2505014 (0.93)	44063 (0.02)	148554 (0.06)	2514700 (0.93)	8438 (0.00)	68579 (0.03)	(0.96)
apriori	2914963 (0.26)	229713 (0.02)	8172633 (0.72)	2975646 (0.26)	185554 (0.02)	5751747 (0.51)	(0.79)
glimpse	128272 (0.49)	27111 (0.10)	104167 (0.40)	128160 (0.49)	8421 (0.03)	29723 (0.11)	(0.64)
tpcc	1478498 (0.48)	477466 (0.16)	1096572 (0.36)	1479989 (0.48)	129609 (0.04)	298001 (0.10)	(0.62)
Avg.	(0.50)	(0.14)	(0.36)	(0.50)	(0.02)	(0.10)	(0.62)

Table 6.2 : Decomposition of Row Activations. The SPEC applications are presented in the same order as in Table 4.1, which is based on the fraction of DRAM accesses that are writes.

	No Eager WB	Eager WB	Improved
libquantum	16.77	39.56	2.36
hammer	1.38	2.93	2.13
lbm	1.83	9.58	5.23
astar	1.69	3.36	1.99
sjeng	1.06	1.16	1.10
tonto	3.28	10.11	3.08
gobmk	1.82	3.49	1.91
bzip2	1.70	3.04	1.78
h264ref	3.12	6.58	2.11
gems	1.14	1.59	1.39
calculix	1.83	4.53	2.47
soplex	1.70	2.34	1.38
leslei3d	1.57	2.40	1.53
gromacs	2.08	6.07	2.92
perlbench	2.09	3.97	1.90
cactus	1.06	1.35	1.27
deal	3.33	11.21	3.36
namd	4.31	7.99	1.86
mcf	3.12	9.40	3.01
zeusmp	1.47	1.82	1.24
gcc	2.16	3.67	1.70
povray	2.91	4.58	1.57
gams	2.62	3.08	1.18
sphinx3	2.09	2.35	1.12
bwave	1.06	1.09	1.03
omnetpp	2.64	2.75	1.04
apriori	2.27	3.02	1.33
glimpse	2.37	3.98	1.68
tpcc	1.94	3.34	1.72

Table 6.3 : Average Row Buffer Hit Count for No Eager and Eager WB. The Improved indicates the the number that the results in Eager WB are normalized to the results in No Eager WB.

## **Conclusions**

---

This thesis introduces a new way in which to use eager writeback in order to optimize DRAM performance and energy efficiency. Under this scheme, dirty cache lines are written to DRAM ahead of time when their associated row has been activated. This enables better clustering of accesses, both read and write, that target the same row. The proposed scheme increases the row-level locality of access and reduces the number of row activations, increasing bandwidth and reducing energy.

The proposed scheme reduces row activations by an average of 36% for 29 applications. For the applications with more than 15% DRAM writes, row activations are reduced by 69%. This reduction results in significant improvements in both performance and energy. Specifically, 20 applications achieve a 10% improvement in both performance and energy consumption.

This thesis evaluates the proposed eager writeback scheme along several dimensions. In most cases, it is sufficient to perform eager writebacks only for cache lines in the LRU position. Speculatively writing back cache lines in more recently used positions results in marginal benefits. Therefore, the technique is practical, as it can share cache line eviction logic. In addition, the use of a modified scheduling algorithm and allowing eager writebacks to be canceled ensures that our proposed techniques do not degrade performance or energy consumption.

## 7.1 Future work

The wide use of multi-core processors requires main memory to have steadily rising bus frequency to provide higher data bandwidth. Moreover, data in the row buffer is accessed much fewer, as memory access streams that multi-core processors simultaneously generate have made main memory randomly accessed [28, 16]. The dynamic energy consumption of main memory in this trend significantly increases. Furthermore, the situation will be worse as systems continue to evolve to embed more cores in the design. To this extent, more work should be done to qualitatively and quantitatively verify the suitability of using the scheme proposed in this thesis on multi-core systems.

The primary concern with the use in multi-core systems is the effectiveness of SWAS and Cancellation. In these systems, almost all memory banks are busy for processing requests coming from the multiple cores. SWAS and Cancellation are not likely to be effective in this case because there will be considerably fewer usable idle cycles during which eager writebacks can be scheduled and processed. This means that performance improvement and energy savings with the direct use of the proposed scheme on multi-core systems can be not as good as results presented in this thesis. Thus, the proposed scheme would need to be qualitatively and quantitatively evaluated, analyzed and improved in more detail to be efficiently working in single-, multi-, or even many-core systems.

Managing useless writes should also be taken into account in current computer systems. This thesis has shown that it is sufficient to only perform eager writebacks on cache lines in the LRU position. However, ignoring the waste of memory bandwidth that is caused by the processing of useless writes can be problematic in such systems that the available memory bandwidth is scarce and under heavy use (*i.e.* many-core systems). Further research is needed to determine which dirty cache lines will be and not be generating useless writes if

eagerly written. This is an issue of identifying premature and mature cache lines, similar to that identifying dead cache lines [59, 60]. We plan to analyze the behavior of writes to cache lines to understand the characteristics of those cache lines.

---

## Bibliography

---

- [1] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” in *ISCA '00*, pp. 128–138, 2000.
- [2] Micron Technology Inc., “Micron MT41J128M8 DDR3-1600,” 2010. [http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb\\_DDR3\\_SDRAM.pdf](http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf).
- [3] R. Ho, K. W. Mai, and M. A. Horowitz, “The Future of Wires,” in *Proceedings of the IEEE*, pp. 490–504, 2001.
- [4] ITRS. International Technology Roadmap for Semiconductors, 2007 Edition. <http://www.itrs.net/Links/2007ITRS/Home2007.htm>.
- [5] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, “Energy Management for Commercial Servers,” *IEEE Computer*, vol. 36, pp. 39–48, 2003.
- [6] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st ed., 2009.
- [7] M. S. Ware, K. Rajamani, M. S. Floyd, B. Brock, J. C. Rubio, F. L. Rawson III, and J. B. Carter, “Architecting for power management: The IBM POWER7<sup>TM</sup> approach,” in *HPCA '10*, pp. 1–11, 2010.
- [8] DDR3 SDRAM System-Power Calculator. [http://www.micron.com/support/dram/~//media/Documents/Products/Power%](http://www.micron.com/support/dram/~//media/Documents/Products/Power%20Calculator)

20Calculator/4300DDR3\_Power\_Calc.ashx.

- [9] 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*, pp. 72–82, 2010.
- [10] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, “DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems,” Tech. Rep. TR-HPS-2010-002, The University of Texas at Austin, April 2010.
- [11] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.
- [12] S. Rixner, “Memory Controller Optimizations for Web Servers,” in *MICRO 37*, pp. 355–366, 2004.
- [13] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu, “Mini-rank: Adaptive DRAM architecture for improving memory power efficiency,” in *MICRO 41*, pp. 210–221, 2008.
- [14] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, “Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs,” *IEEE Comput. Archit. Lett.*, vol. 8, pp. 5–8, January 2009.
- [15] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, “Rethinking DRAM design and organization for energy-constrained multi-cores,” in *ISCA '10*, pp. 175–186, 2010.
- [16] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, “Micro-pages: increasing DRAM efficiency with locality-aware data placement,” in

- ASPLOS '10*, pp. 219–230, 2010.
- [17] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, “Eager writeback - a technique for improving bandwidth utilization,” in *MICRO 33*, pp. 11–21, 2000.
- [18] Z. Zhang, Z. Zhu, and X. Zhang, “A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality,” in *MICRO 33*, pp. 32–41, 2000.
- [19] T.-F. Chen and J.-L. Baer, “Reducing memory latency via non-blocking and prefetching caches,” in *ASPLOS-V*, pp. 51–61, 1992.
- [20] S. McKee, W. Wulf, J. Aylor, R. Klenke, M. Salinas, S. Hong, and D. Weikle, “Dynamic Access Ordering for Streamed Computations,” *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1255–1271, 2000.
- [21] Z. Zhu, Z. Zhang, and X. Zhang, “Fine-grain Priority Scheduling on Multi-channel Memory Systems,” in *HPCA '02*, pp. 107–, 2002.
- [22] I. Hur and C. Lin, “Adaptive History-based Memory Schedulers,” in *MICRO 37*, pp. 343–354, 2004.
- [23] Z. Zhu and Z. Zhang, “A Performance Comparison of DRAM Memory System Optimizations for SMT Processors,” in *HPCA '05*, pp. 213–224, 2005.
- [24] I. Hur and C. Lin, “Memory Scheduling for Modern Microprocessors,” *ACM Transactions on Computer Systems*, vol. 25, no. 4, 2007.
- [25] O. Mutlu and T. Moscibroda, “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,” in *MICRO 40*, 2007.
- [26] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems,” in *ISCA '08*, pp. 63–74, 2008.



- [27] E. Cooper-Balis and B. Jacob, “Fine-grained activation for power reduction in dram,” *IEEE Micro*, vol. 30, pp. 34–47, May 2010.
- [28] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, “Future scaling of processor-memory interfaces,” in *SC '09*, pp. 42:1–42:12, 2009.
- [29] I. Hur and C. Lin, “A comprehensive approach to DRAM power management,” in *HPCA-14*, pp. 305–316, 2008.
- [30] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller, “Improving energy efficiency by making DRAM less randomly accessed,” in *ISLPED '05*, pp. 393–398, 2005.
- [31] J. Pisharath and A. Choudhary, “An integrated approach to reducing power dissipation in memory hierarchies,” in *CASES '02*, pp. 88–97, 2002.
- [32] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware, “System power management support in the IBM POWER6 microprocessor,” *IBM J. Res. Dev.*, vol. 51, pp. 733–746, 2007.
- [33] J. Flinn, K. I. Farkas, and J. Anderson, “Power and energy characterization of the Itsy pocket computer (version 1.5),” Tech. Rep. TN-56, Compaq Western Research Laboratory, February 2000.
- [34] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *USENIX ATC '10*, pp. 21–21, 2010.
- [35] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Flicker: saving DRAM refresh-power through critical data partitioning,” in *ASPLOS '11*, pp. 213–224, 2011.
- [36] C. Isen and L. John, “ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem,” in *MICRO 42*, pp. 337–346,

2009.

- [37] M. Ghosh and H.-H. S. Lee, “Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs,” in *MICRO 40*, pp. 134–145, 2007.
- [38] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, “Power aware page allocation,” in *ASPLOS-IX*, pp. 105–116, 2000.
- [39] H. Huang, P. Pillai, and K. G. Shin, “Design and implementation of power-aware virtual memory,” in *USENIX ATC '03*, pp. 5–5, 2003.
- [40] H. Huang, K. G. Shin, C. Lefurgy, K. Rajamani, T. Keller, E. Hensbergen, and F. Rawson, “Software-hardware cooperative power management for main memory,” in *PACS'04*, pp. 61–77, 2005.
- [41] M. Lee, E. Seo, J. Lee, and J.-s. Kim, “PABC: Power-Aware Buffer Cache Management for Low Power Consumption,” *IEEE Transactions on Computers*, vol. 56, pp. 488–501, 2007.
- [42] M. Bi, R. Duan, and C. Gniady, “Delay-Hiding energy management mechanisms for DRAM,” in *16th International Conference on High-Performance Computer Architecture*, HPCA-16, pp. 1–10, 2010.
- [43] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini, “DMA-aware memory energy management,” in *12th International Symposium on High-Performance Computer Architecture*, HPCA-12, pp. 133–144, 2006.
- [44] V. D. L. Luz, M. Kandemir, and I. Kolcu, “Automatic data migration for reducing energy consumption in multi-bank memory systems,” in *DAC '02*, pp. 213–218, 2002.

- [45] 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*, pp. 159–, 2001.
- [46] O. Ozturk and M. Kandemir, “Data Replication in Banked DRAMs for Reducing Energy Consumption,” in *ISQED '06*, pp. 551–556, 2006.
- [47] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Scheduler-based DRAM energy management,” in *DAC '02*, pp. 697–702, 2002.
- [48] A. Merkel and F. Bellosa, “Memory-aware scheduling for energy efficiency on multi-core processors,” in *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pp. 1–1, 2008.
- [49] J.-W. Jang, M. Jeon, H.-S. Kim, H. Jo, J.-S. Kim, and S. Maeng, “Energy reduction in consolidated servers through memory-aware virtual machine scheduling,” *IEEE Transactions on Computers*, vol. 60, pp. 552–564, 2011.
- [50] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, “COTSon: infrastructure for full system simulation,” *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, 2009.
- [51] DRAMSim2: A Detailed Memory-System Simulation Framework. <http://www.ece.umd.edu/dramsim/>.
- [52] AMD SimNow Simulator. <http://developer.amd.com/tools/simnow/Pages/default.aspx>.
- [53] “Calculating Memory System Power for DDR3,” Tech. Rep. TN-41-01, Micron Technology Inc., 2007.

- [54] U. Manber and S. Wu, “GLIMPSE: a tool to search through entire file systems,” in *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 23–32, 1994.
- [55] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, “MineBench: A Benchmark Suite for Data Mining Workloads,” in *2006 IISWC*, pp. 182–188, 2006.
- [56] J. L. Henning, “SPEC CPU2006 Benchmark Descriptions,” in *Proceedings of ACM SIGARCH Computer Architecture News*, 2005.
- [57] TPCC-UVa: A free, open-source implementation of the TPC-C Benchmark. <http://www.infor.uva.es/~diego/tpcc-uva.html>.
- [58] AMD, “BIOS and Kernel Developer’s Guide for AMD Athlon 64 and AMD Opteron Processors.” [http://support.amd.com/us/Processor\\_TechDocs/26094.pdf](http://support.amd.com/us/Processor_TechDocs/26094.pdf).
- [59] A.-C. Lai, C. Fide, and B. Falsafi, “Dead-block prediction & dead-block correlating prefetchers,” in *ISCA ’01*, pp. 144–154, 2001.
- [60] M. Kharbutli and Y. Solihin, “Counter-based cache replacement and bypassing algorithms,” *IEEE Transactions on Computers*, vol. 57, pp. 433–447, April 2008.