1-1-2006

The design of a memory controller for DDR SDRAM

Gary Anthony Thorpe
Ryerson University

Follow this and additional works at: <http://digitalcommons.ryerson.ca/dissertations>

Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Thorpe, Gary Anthony, "The design of a memory controller for DDR SDRAM" (2006). *Theses and dissertations*. Paper 424.

This Thesis is brought to you for free and open access by Digital Commons @ Ryerson. It has been accepted for inclusion in Theses and dissertations by an authorized administrator of Digital Commons @ Ryerson. For more information, please contact bcameron@ryerson.ca.

THE DESIGN OF A MEMORY CONTROLLER FOR
DDR SDRAM

by

Gary Anthony Thorpe

B.Sc. Computer Engineering

Melbourne, Florida, USA, 2001

A thesis

presented to Ryerson University

in partial fulfilment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2006

©Gary Anthony Thorpe 2006

UMI Number: EC53793

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform EC53793
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

—

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

—

THE DESIGN OF A MEMORY CONTROLLER FOR DDR SDRAM

Master of Applied Science, 2006

Gary Anthony Thorpe

Electrical and Computer Engineering

Ryerson University

Abstract

Memory system performance is an important factor in determining overall system performance. The design of key components of the memory system, such as the memory controller, becomes more important as memory performance becomes a limiting factor in high performance computing.

This work focuses on the design of a unit which sends control signals to Double Data Rate Synchronous DRAM (DDR SDRAM). The design is based on established concepts such as access reordering. A novel, adaptive page policy based on a machine learning algorithm has been developed in this work and evaluated with traditional page policies.

This work illustrates some of the design trade-offs in a memory controller and the performance of the designs when using real application address traces. The results show that access reordering improves the performance of DDR SDRAM compared to in-order scheduling (up to 50% improvement) and that scheduling multiple requests can result in latency hiding. The dynamic page policy approximates the best static page policy in most cases.

Acknowledgements

I would like to thank my supervisor, Dr. Mekhiel, for his support and advice on the topic of this thesis and his feedback on the production of this work. I also thank the members of the exam committee for taking the time to read and evaluate this work and provide feedback.

I would like to thank my family and friends for their concern and for their support of and interest in my efforts to complete my degree. It has been appreciated.

Finally, I would like to thank the Department of Electrical and Computer Engineering and Ryerson University for their financial support of my studies and for the help of staff in making my time at Ryerson University more productive.

Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Contents	v
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Current Trends	1
1.2 Motivation	2
1.3 Organization	4
2 Background Information and Previous Work	5
2.1 The Memory Hierarchy	6
2.1.1 Cache Performance	6
2.1.2 Tolerating Latency	8
2.2 DRAM	10
2.2.1 DDR SDRAM	11
2.2.2 Other DRAM Architectures	14

2.3	Main Memory and System Performance	15
2.4	Memory Controller Design	15
2.5	Machine Learning	17
2.6	Summary	19
3	Design of the Memory Controller	21
3.1	HDL Specification	22
3.2	Design Hierarchy	23
3.2.1	Configuration Interface	25
3.2.2	Host Input Interface	28
3.2.3	Control Unit	34
3.2.4	Data Transfer Unit	42
3.2.5	Host Output Interface	43
3.3	Summary	44
4	Dynamic Page Policy	45
4.1	Page Rate Definitions	45
4.2	Page Rates and DDR SDRAM Performance	46
4.3	Reinforcement Learning Algorithms	47
4.4	Controlling Page Policy	49
4.4.1	State-Action Table	52
4.4.2	Generalization: Function Approximation	52
4.4.3	Learning and Access Reordering	55
4.4.4	Synthesizeable Implementation	55
4.5	On-line Learning and Control	57
4.5.1	Table-Based Example	57
4.5.2	Function Approximation Example	61
4.6	Summary	66
5	Experimental Methodology	67
5.1	Traces	69

5.2	Simulation Tools	70
5.3	VHDL Instrumentation	70
5.4	Verification of Control Sequences	71
5.5	Simulation Parameters	72
5.6	Synthesis	73
5.7	Form of Results	73
6	Experimental Results	76
6.1	Distribution of Operations Across Banks	77
6.1.1	Overall Distributions	77
6.1.2	Bank Parallelism	81
6.2	Page Rates	85
6.2.1	Page Hit Rates	85
6.2.2	Page Miss Rates	88
6.2.3	Page Open Rates	91
6.2.4	Affect of Generalization on Page Rates	91
6.3	Average Response Time	93
6.4	Average Latency	94
6.5	Average Bus Utilization	98
6.6	Performance vs. Queue Size	98
6.7	Summary	102
7	Discussion of Results	103
7.1	Addressing Modes	103
7.2	Scheduling Policies	104
7.3	Page Policy	104
7.4	Learning Algorithm Performance	105
7.5	Queue Length and Performance	106
7.6	Summary	107

8	Conclusions	109
8.1	General Observations	109
8.2	Novel Contributions	110
8.3	Future Work	111
8.3.1	Learning Algorithms	111
8.3.2	Bank Parallelism	112
8.3.3	Implementation	113
A	Format of Raw Simulation Results	114
B	Source Code	119
B.1	Modified Binary to ASCII Trace Conversion Tool	119
B.1.1	Example of Converted Trace Data	119
B.1.2	Modified b2a.c	120
B.1.3	Modified byuttr.h	124
	Bibliography	127

List of Tables

3.1	Example of Mapping Addresses to DIMMs	31
3.2	Bank, Row and Column Decoding	33
4.1	Sarsa Parameters for Controlling Page Policy	51
4.2	Control using State-Action Table	59
4.3	Control using Function Approximation	62
5.1	Parameters for DDR400 SDRAM	73
6.1	Average Bank Parallelism: Maximum Queue Size	82
6.2	Average Bank Parallelism vs. Queue Size: Linear	83
6.3	Average Bank Parallelism vs. Queue Size: Custom	84
6.4	Page Hit Rates	86
6.5	Page Hit Rates (α, γ): Eon	87
6.6	Page Miss Rates	90
6.7	Page Miss Rates (α, γ): Eon	90
6.8	Page Open Rates	92
6.9	Page Open Rates (α, γ): Eon	92
6.10	Average Response Times	94
6.11	Average Latencies	97
6.12	Average Bus Utilization	99
6.13	Performance vs. Queue Size: Linear	100
6.14	Performance vs. Queue Size: Custom	101

List of Figures

3.1	Top-Level of Memory Controller	23
3.2	Memory Controller Hierarchy	24
3.3	Configuration Interface	27
3.4	Host Input Interface	29
3.5	Control Unit	34
3.6	State Diagram of Bank Model	36
3.7	State Diagram of Row/DIMM Model	37
3.8	State Diagram of Bus Model	38
3.9	Generic Scheduling Procedure	42
3.10	Data Transfer Unit	43
3.11	Host Output Interface	44
4.1	Sarsa Algorithm	48
4.2	Linear Gradient-Descent Sarsa	54
5.1	Simulation Method	68
5.2	Design Framework	74
6.1	Eon Operation Distributions	78
6.2	Bzip Operation Distributions	78
6.3	GCC Operation Distributions	79
6.4	Parser Operation Distributions	79
6.5	PERL Operation Distributions	80

6.6	TPCC Operation Distributions	80

Chapter 1

Introduction

Modern computer systems usually have multi-level memory hierarchies: one or more levels of cache and main memory are a standard feature for most general purpose machines. The subject of this thesis involves the lowest level of this hierarchy, farthest from the CPU: the main memory system. The study of the main memory subsystem is a very important aspect of evaluating overall computer system performance.

1.1 Current Trends

Due to the inherent limitations of cache subsystems and the widening performance gap between processors and memory, more and more applications are becoming *memory-bound*: further increases in cache sizes, associativity, novel caching algorithms, and advanced out-of-order execution techniques are having less effect on an application's running time. These execution times already (or will eventually) depend primarily on how quickly memory operations can complete.

A system's performance is generally limited by its slowest part. The widening performance gap between processors and memory has made the memory subsystem one of the limiting factors of a general purpose computer

system's performance. This phenomenon has been termed *The Memory Wall* (see Chapter 2). For example, previous work with RAMBUS Dynamic RAM (RDRAM) found that, even with four separate channels of RDRAM (each having a bandwidth of 1.6GB for a total bandwidth of 6.4GB), the SPEC benchmark suite spent on average 57% of the time waiting for the main memory to fulfil requests instead of doing useful computation [8]. Modern memory systems usually have large bandwidths and this is often thought of as being able to compensate for large memory latencies. This may be true for applications where large data streams are used (such as media processing), but as the above example shows, the availability of bandwidth alone cannot help to reduce all memory latencies.

Because memory technologies are unlikely to change drastically in terms of their speed (given that the current technologies offer cheap, high-capacity devices), more focus is usually put on changing the organization and interface of the memory to develop new memory designs. For example, many memory architectures are simply variants of DRAM: extended data out DRAM, synchronous DRAM (SDRAM), double data rate SDRAM (DDR SDRAM), and RDRAM all have similar underlying DRAM-based technology. The major difference among these variants is in how the devices are interfaced to the rest of the system [5].

1.2 Motivation

However, because the performance gap between processors and memory systems continues to grow rapidly, more effort is now being put on the system-level design of DRAM memory systems [17].

There are many techniques used to hide long latencies outside of memory system design. These include multi-threaded processors, out-of-order execution, non-blocking caches, prefetching, branch prediction and speculative execution. However, many of these techniques have reached their limits in

terms of design effort, cost, and power dissipation. Focusing on optimizing already small delays in processors is counterproductive when the limiting factor (memory performance) remains neglected.

Most, if not all, DRAM-based memory subsystems do not have a uniform access time. In fast-page mode DRAM, accesses to the same row have a much lower latency than if a new page (row) has to be opened (see Chapter 2 for more details on DRAM structure and organization). This non-uniform access time provides a challenge and an opportunity to improve memory performance.

One architectural approach to improving the performance of the memory subsystem is to modify the memory controller unit to provide optimizations for this non-uniform access time. For example, memory access scheduling can be implemented in the memory controller to reduce page misses and decrease the average access latency.

However, previous designs were usually targeted at vector processing machines, media processors and generally for situations that are not encountered by the general purpose computer. Instead of designing the memory subsystem for architectures which are less dependent on a high-speed cache (because their performance is driven by bandwidth), this thesis will explore memory controller design for a system supporting a general purpose CPU. One common memory technology used with such CPUs is DDR SDRAM.

The memory controller's design will be based on the specifications of an integrated memory controller found in the Intel 82875P Memory Controller Hub (MCH). This commercial memory controller supports dual channels of DDR SDRAM interfaced to the Intel Pentium 4 processor [16]. While intended to work with the Pentium 4 processor, the memory controller integrated with the MCH has features that are not processor specific.

In this thesis, the design space for memory controller policies will be explored by varying some key parameters associated with memory controller design: page policy and access reordering. In addition, a new approach to

DRAM page policies will be explored and presented.

1.3 Organization

The structure of this thesis is as follows. After this introduction, Chapter 2 will cover background information, theory, and previous work related to memory controller design and the novel approach explored in this work. Chapter 3 will go through the hardware design and algorithms involved with producing a standard memory controller. Chapter 4 will present a novel approach to page policy based on machine learning. Chapter 5 will explain the methodology and tools used in this thesis. Chapter 6 will outline the results obtained from the experimentation carried out; Chapter 7 will provide some analysis and discussion of these results and identify general trends. Finally, Chapter 8 will conclude this report by highlighting the contributions of this work and suggesting potential areas of interest for future research.

Chapter 2

Background Information and Previous Work

The focus of this work is the design of a DDR SDRAM memory controller, based on the capabilities of a memory controller integrated into a commercial design (Intel 82875P). Memory controller design has been explored by various researchers and the performance of the memory system has been an active interest of those designing high performance systems.

Most modern systems have a hierarchical memory system. The reasons for the existence of this hierarchy and the place of DRAM in this organization will be explained in Section 2.1. DDR SDRAM will also be described to relate its architecture to other DRAM variants in Section 2.2. The effect of main memory performance on system performance will be outlined in Section 2.3. Current research activities on memory controllers will be also discussed in Section 2.4. Finally, because the main novel application in this thesis is based on a machine learning algorithm, general information on these algorithms will also be provided in Section 2.5.

The information in this chapter will provide the necessary information to understanding the contributions of this work in relation to the rest of the research on memory controller design.

2.1 The Memory Hierarchy

Most main memory systems are composed of a form of Dynamic RAM (DRAM) technology. DRAM is preferred over other technologies such as Static RAM (SRAM) because of its lower cost, lower power usage, and higher storage density.

While SRAM supports higher speeds, the cost and power of using SRAM for a main memory system would make the system expensive and at the same time offer a lower storage capacity. While DRAM has access times in the tens of nanoseconds, SRAM can be accessed in under ten nanoseconds. In the first level of the cache hierarchy of multi-gigahertz CPUs, SRAM operates at clock periods below $1ns$. However, the price for this speed (more expensive parts, lower storage density, and higher power) is high enough to prevent SRAM from being used to provide most of the storage for the memory system.

Instead, SRAM is often used in high performance cache memories which are found in smaller quantities than the main memory subsystem and higher up in the memory hierarchy, closer to the CPU (level 1 is usually integrated onto the CPU's die). This arrangement maximizes the cost-benefit of the SRAM: the most frequently accessed data are kept in the smaller high speed caches. The main memory system, composed of the more affordable DRAM in higher capacities, is only used to service infrequent cache misses. This arrangement usually works in a general purpose machine because the access patterns of applications result in temporal and spatial locality. This high locality allows the cache to effectively reduce the average memory latency by servicing the majority of the requests.

2.1.1 Cache Performance

For example, consider a system with two levels of cache. Assume these assumptions are true: the level one (L1) access time (T_{L1}) is $1ns$, the L1 hit rate is 95% (miss rate, $M_{L1} = 0.05$), the level two (L2) access time (T_{L2}) is

5ns, the L2 hit rate is 90% (miss rate, $M_{L2} = 0.1$), and the main memory has an average access time (T_M) of 75ns. The average memory latency (T_{AV}) is given by:

$$\begin{aligned} T_{AV} &= T_{L1} + M_{L1} \cdot T_{L2} + M_{L1} \cdot M_{L2} \cdot T_M \\ &= 1 + 0.05 \cdot 5 + 0.05 \cdot 0.1 \cdot 75 \\ &= 1.25ns \end{aligned}$$

The average memory access latency is almost equal to T_{L1} when the caches have these relatively high hit rates (not necessarily true for all workloads).

Assuming higher hit rates ($M_{L1} = 0.025$, $M_{L2} = 0.05$):

$$\begin{aligned} T_{AV} &= 1 + 0.025 \cdot 5 + 0.025 \cdot 0.05 \cdot 75 \\ &= 1.2ns \end{aligned}$$

Notice that halving the miss rates of the caches has not resulted in an equivalently large improvement in the average access latency. This is the inherent limitation of a cache system: even assuming much higher hit rates (which are not necessarily attainable) the system's performance does not scale linearly. The limit of this performance is T_{L1} as the L1 hit rate tends to 100%.

Assuming lower hit rates ($M_{L1} = 0.075$, $M_{L2} = 0.15$):

$$\begin{aligned} T_{AV} &= 1 + 0.075 \cdot 5 + 0.075 \cdot 0.15 \cdot 75 \\ &= 2.2ns \end{aligned}$$

A 50% increase in the miss rates results in almost double the latency (the hit rates remain relatively high at 92.5% for L1 and 85% for L2). Systems that use caches depend on high hit rates in order to get a high overall performance.

In order to minimize costs and maintain high performance, a compromise between using high-speed SRAM and cheaper DRAM has resulted in a standard memory hierarchy for most general purpose machines. The SRAM is used in smaller, faster caches to accelerate the common case (cache hits) while the DRAM is used to provide the main storage capacity and provide

replacement blocks for the faster cache. This arrangement balances the cost of high speed memory with the need for large cheap storage. Most general purpose CPUs rely on the caches (with high hit rates) to maintain their high performance. For workloads with high cache miss rates, performance becomes dominated by the performance of the main memory system (as $M_{L1} \rightarrow 1, M_{L2} \rightarrow 1$):

$$T_{AV} = 1 + 5 + 50$$

If the amount of time used to access the main memory does not decrease, then it will begin to dominate the overall access time (especially with low hit rates in the caches). The scalability of the system’s overall performance is limited due to *Amdahl’s Law* [11]: no matter how much time is spent accessing fast caches, eventually main memory must be accessed and this limits the speedup attainable from improving the caches alone.

2.1.2 Tolerating Latency

Because caches only provide high performance when an application accesses memory in certain patterns, modern CPUs are developed with techniques to hide long memory latencies on cache misses or even the relative long latency of a L1-miss and L2-hit [11]:

Hardware Multi-threading allows the processor to execute multiple threads either simultaneously or using hardware thread switching [24]. This technique allows greater processor utilization (thread-level parallelism becomes instruction-level parallelism) when threads simultaneously execute. It also allows the processor to continue executing another thread if one thread stalls on a cache miss or other delays. This improves the processor’s ability to hide latencies by overlapping them with useful computation.

Out-of-Order (OOO) Execution allows the processor to rearrange the instruction stream (in one thread) so that instructions that do not de-

pend on the results of stalled instructions (such as a load that depends on data that is not in the cache) can proceed without waiting. This results in the processor continuing to execute instructions during periods when an in-order processor would have completely stalled. The window of instructions that has to be maintained for OOO execution cannot be scaled indefinitely.

Non-blocking Caches continue to supply cache hits while servicing cache misses. This allows other data to be accessed during the time when the main memory system is fetching the replacement cache block. Multiple misses can also be supported, but this requires the memory system to service multiple outstanding requests (see Chapter 3 for a simple example of such an interface).

Prefetching (hardware or software based) is an attempt to anticipate what memory accesses will occur and fetch the relevant cache blocks in advance from main memory. This leverages the spare bandwidth usually available in a main memory system. Prefetching can hinder overall performance (it can waste bandwidth on blocks that are not used).

Speculation can occur within the processor itself as well as in the memory hierarchy (prefetching). Speculative execution allows instructions to be executed without being committed: the results are discarded if the wrong path was chosen. Speculation allows the processor to continue executing instructions for longer when stalls occur. A speculative processor can use most of its resources executing instructions that are never used (potentially inefficient).

Branch Prediction is used to predict where program branches will lead so that the processor can fetch and issue those instructions. This technique is used to maintain high processor throughput by requesting instructions (and possibly data) in advance of their use. When branch

prediction is not combined with speculation, instructions are not executed until the prediction is known to have been true or false.

The fact that these and other techniques exist and are widely used implies that hiding long latencies outside of the processor is an important means of attaining a high performance system. It also suggests that these long latencies are beginning to dominate the overall performance of general purpose systems.

Now that some basic information on the memory hierarchy has been provided, the components of most general purpose memory systems, DRAM-based technologies, will be discussed.

2.2 DRAM

SRAM uses a synchronous latched protocol to match its high speed operation. All the locations in an SRAM module require the same amount of time to be accessed. The address bits are not multiplexed and each access requires only one operation.

In contrast, the original DRAM used an asynchronous protocol which was suited for its slower operation. The address space for a DRAM module is divided into rows and columns: the address bits are multiplexed (to reduce the cost) to first access the row and then access the column. The row is first precharged and then the read or write is sent to the relevant column. Addresses and data are transferred using strobes. DRAM must also be periodically refreshed to preserve the data contents on the modules.

In fast page mode (FPM) DRAM, the row can be left *open* to allow successive accesses to columns in the same row to occur quickly. This is more efficient than having to send the same row address over the multiplexed lines for each access to precharge the row. The basic idea of a non-uniform access time in all current DRAM-based memories is based on this underlying attribute of DRAM.

DRAM is relatively slow, but because of its high capacity, large amounts can be integrated into the same component. To compensate for the low speed of operation, several devices can be connected in parallel. When used in such a parallel configuration, the individual speed of each device is unchanged but more data is moved at the same time resulting in higher bandwidth. This improvement is easy to accomplish, cost-effective and is a popular means of improving the performance of a DRAM-based memory system. This high bandwidth allows DRAM to achieve high throughput even though the latency is still high.

This scheme is usually replicated outside the integrated components by arranging the components into banks. By using interleaved banks or by operating the banks independently, further bandwidth improvements can be achieved for low component and design costs.

Because improving the bandwidth is much simpler than improving the latency, some generations of DRAM technologies are now marketed and distinguished not by their access latencies but by their bandwidths. Using DDR SDRAM speed grades as an example, DDR266 has a maximum bandwidth of $266MB/s$ and DDR333 has a maximum bandwidth of $333MB/s$. While it is true that DDR266 has a $133MHz$ clock and DDR333 has a $167MHz$ clock, the command latency has not necessarily improved by the same proportion that the bandwidth has increased. For some operations, the command latency of DDR333 is unchanged compared to DDR266 [18]. Only the data bandwidth is guaranteed to have improved.

2.2.1 DDR SDRAM

Synchronous DRAM (SDRAM) uses a clock signal to synchronize address commands to the memory module. The clock signal is also used to synchronize data transfers in and out of the module. While the interface is now synchronous, the address lines are still multiplexed to lower costs. Like FPM DRAM, SDRAM can allow pages (analogous to rows) to be kept open so

that several accesses to the same page can occur more quickly without the need to precharge the page. SDRAM also transfers data in bursts which closely matches the need to service cache misses by transferring entire blocks of data. The elimination of the slower asynchronous strobes and the burst capability lead to higher performance memory systems when using SDRAM.

DDR SDRAM is a DRAM architecture with a synchronous interface and support for data bursts like SDRAM. The major difference between SDRAM and DDR SDRAM is that DDR uses both rising and falling edges of a strobe signal (synchronized with the clock signal) to transfer data. This doubles the maximum data transfer rate of the memory. Again, the focus for a DRAM-based system is the bandwidth since the latency is harder to improve.

Internally, SDRAM and DDR SDRAM are multi-banked architectures like other recent DRAM-based memory systems. They allow some simple programmability which eases the memory controller from some of the overhead of managing the banks of DRAM and allows higher speed operation by allowing bursts of data in transfers. For example, both technologies use registers during the initialization phase to set various attributes such as the burst size for memory transfers.

A DDR SDRAM memory chip has several signals [18]:

CK, $\overline{CK}$ Differential clock inputs. All address and control input signals are sampled on the crossing of the positive edge of CK and negative edge of CK. Output (read) data is referenced to the crossings of CK and CK (both directions of crossing).

CKE Clock Enable: CKE HIGH activates and CKE LOW deactivates internal clock signals, device input buffers and output drivers.

DQ Bidirectional data bus: 16 (x16) or 8 bits (x8) wide.

DQS DQS is a bidirectional signal (for x16, UDQS and LDQS) driven by the host on writes and driven by the DDR SDRAM on reads. Data is

edge-aligned for writes and level-aligned for reads. DQS is synchronized with the clock signal during data transfers.

A_0 – A_{13} Address lines which carry multiplexed row and column addresses. A_{10} is used to control auto-precharge when sending the column address for a read or write operation.

BA_0 , BA_1 Bank address lines (each device has four banks).

$\overline{CS}$ Chip Select: commands are masked when $\overline{CS}$ is registered high.

$\overline{RAS}$, $\overline{CAS}$, $\overline{WE}$ Row Address Strobe, Column Address Strobe, and Write Enable are used (with Chip Select) to send commands to the DDR SDRAM. The names are derived from traditional names for DRAM signals, but are used to encode commands such as bank activation, read, write, precharge, and refresh.

DM Data Mask is used to mask data during writes (UDM and LDM for x16).

DDR SDRAM has several timing parameters. The parameters of interest in this thesis are described below and related to operations in the same bank except where noted [18]:

tCK, Clock Period Period of the clock signal supplied to the DDR SDRAM.

tDQS Time between issue of a write command and the first latching transition.

tRAS Delay between bank activation (bank opening) and precharge (bank closing) commands.

tRC Auto-refresh command cycle time: time for auto-refresh command to complete.

tMRD Mode Register set command cycle time: time for mode register set command to complete.

tRFC Delay between an auto-refresh command and bank activation command.

tRCD Delay between bank activation and read or write command.

tRP Precharge command cycle time: time for precharge command to complete.

tRRD Delay between activating bank A and bank B in the same device.

tWR Write Recovery Time: period after a write command has been issued to a bank during which the bank cannot accept a precharge command.

tWTR Write to Read Time: delay after sending a write command before it becomes possible to send a read command to the same device without truncating data.

These parameters are used in the memory controller design for internal models to control the sequencing of the operations, especially the bank model (see Chapter 3).

2.2.2 Other DRAM Architectures

Other modern DRAM architectures include extended data out (EDO) DRAM, RAMBUS DRAM (RDRAM), video DRAM (VDRAM) and other variants. Like DDR SDRAM, they are all based on the same DRAM technology. For example, all of the DRAM architectures require periodic refreshing. For SDRAM and RDRAM, refreshing takes the forms of synchronous commands instead of the strobed commands used in the original DRAM, but it still must be carried out.

2.3 Main Memory and System Performance

The importance of main memory performance is illustrated by *The Memory Wall*, a trend observed in computer system performance that suggests system performance will eventually be solely determined by the time it takes to access the main memory subsystem, even with perfect caching [19].

The Memory Wall shows that the performance of a hypothetical system with an infinitely large cache and a perfect replacement algorithm will still be limited by the speed of the main memory system, which agrees with Amdahl's Law (the slowest part of a system determines the system's overall performance).

As a result, current systems are already limited by their memory systems. In database benchmarks, some CPUs are spending most of their time (about three out of four cycles) waiting on the memory subsystem [17]. A simulated processor, using parameters comparable to many modern CPUs, in a system with a high-performance multi-channel RDRAM memory system, was found to have spent over half of its cycles waiting for L2 cache misses (that is, accessing the main memory system) when executing the SPEC2000 benchmark [8]. The performance gap continues to grow yearly: the processor speeds are improving at a faster rate than the memory latencies [20].

The DRAM portion of the memory system has again become a bottleneck which cannot be compensated for by using smaller amounts of faster SRAM. However, no cheap alternative to DRAM has been developed that has the required high operating speeds. Modern memory systems will continue to use DRAM for the foreseeable future.

2.4 Memory Controller Design

Because of the Memory Wall, adding more cache or increasing the speed of the caches will not scale the performance of the system indefinitely. This means that performance of the DRAM portion of the memory system is the

limiting factor for future systems. Because of this, there has been much work in the field of memory controller design for DRAM-based systems.

A memory controller designed for RDRAM combined hardware prefetching during idle periods with bank addressing schemes to increase performance [8]. The results from that work indicated that because of the similarity of the underlying technologies, modern DRAM-based memory systems have the same pitfall of stalling the processor for relatively long periods whenever they are accessed.

A controller implementing memory access scheduling was designed for the Imagine stream processor [7]. The Imagine processor was a media/vector processor that did not use caching because caches are usually ineffective for large data streams. The memory controller was able to increase throughput and performance of the processor on several benchmarks: applications traces showed an improvement of up to 93% when using aggressive scheduling compared to performing operations in-order.

Adaptive methods have also been applied to the scheduling problem for the Power5 general purpose processor: the access history observed by a memory controller was used to select read or writes operations by dynamically choosing between history-based arbiters [12]. In contrast, this work uses a fixed scheduling scheme and instead uses the memory operation history to predict the page policy.

In addition to scheduling policies, bank mapping policies have also been pursued. Various schemes have been developed to interleave bank accesses in such a way as to reduce row-buffer conflicts [26, 27]. In addition to using novel address decoding schemes, the same researchers also found that scheduling writes differently from reads can be beneficial to performance [26, 27].

Memory controllers can also implement additional address translation schemes in order to dynamically give applications the ability to control how their data is accessed and cached: the Impulse memory controller provides an interface for software to remap physical pages and do page colouring in

hardware while also performing prefetching [6].

This thesis will study controller design for a typical system involving a general purpose CPU: one with a cache hierarchy and with memory accesses showing more temporal locality than would be found in a streaming processor. There is one study of DDR SDRAM and the IA32 system for both the IA32 in-order bus and the Itanium out-of-order bus [20]. However, that study was based on a purely software simulation written in C++: the work in this thesis is based on a VHDL model. A VHDL model can be a part of a synthesizable hardware design as well as a simulation platform.

2.5 Machine Learning

Machine learning is an area of study in Artificial Intelligence (AI). Techniques of AI research, such as neural networks, are used widely in pattern-recognition systems, fuzzy logic applications, and applications for which there exists available data to train the network to perform a fixed task. For tasks where there is no training data available, many neural algorithms are usually not the best choices for machine learning [9].

Because memory access patterns are not generally known beforehand in a general purpose machine, an algorithm that is going to be applied to the design of a memory controller cannot rely on training vectors. Similarly, assumptions about the distribution of access patterns cannot be justified for a machine that could be put to a variety of uses. In order to work within a memory controller, a machine learning algorithm must be able to accomplish goals that, while definable, will not be static.

The chosen learning algorithm in this thesis comes from a family of algorithms devised to solve reinforcement learning problems. In reinforcement learning, an *agent* (the entity that is learning and potentially making decisions based on what it has learned), interacts with its *environment*. The agent chooses actions based on situations presented by its environment. The

environment also provides *rewards*: the agent's actions result in changes to the environment which can either be interpreted positively or negatively. Rewards are accumulated over time as the experience of learning. When performing control functions, the agent uses this experience to choose its actions according to a *policy*, π , which maps states to actions.

The goal of the agent is to maximize the rewards it receives over time. This basic model of learning allows reinforcement learning algorithms to perform adaptive prediction and control. Feedback from the environment is used to determine the effectiveness of the choices made by the agent and possibly improve future choices. More complete descriptions of this family of algorithms are available [23].

There are several reasons to choose a Reinforcement Learning algorithm:

- Suited for following moving targets (on-line learning or control)
- Some implementations are simple and incremental
- Proven to work: maze navigation, circuit partitioning
- Control and Prediction possible
- May not require environmental models or assumptions

The other methods have many of these advantages, but also disadvantages depending on the method. For each method, there are some weaknesses:

Neural networks Neural networks are suited for supervised learning tasks where the target is known and fixed, for example, in function approximation. Learning can be very slow and requires prior knowledge of correct results.

Statistical methods Statistical Methods such as Bayesian learning require prior knowledge of probability distributions or assumptions about those distributions. Bayesian learning also requires large amounts of computation.

Reinforcement learning has been researched to solve various problems in control, pattern recognition, and shortest-path graph problems (dynamic programming such as the Bellman-Ford shortest-paths algorithm). Many reinforcement learning algorithms have several beneficial features.

These include the freedom of not having to model the system's behaviour or make assumptions beforehand. The algorithms can find optimal solutions through trial-and-error searching. Because memory access patterns are hard to predict or model for an arbitrary application, it would seem to be a problem for which a reinforcement learning algorithm is suited. However, reinforcement learning algorithms can also utilize models of the environment (called *planning*).

Another advantage of many of the algorithms is that they can be implemented incrementally: learning proceeds in small simple steps which accumulate over time. This makes them easy to develop and simplifies an implementation.

The algorithms can also be used *on-line*—that is, learning can proceed while the task is being performed and off-line training is unnecessary. This flexibility allows the algorithms to adapt to changing patterns, but other AI techniques such as neural networks are more suited for fixed functions like image classification or function generation. For the particular problem of predicting memory access patterns, which change depending on the application, a reinforcement learning algorithm is a better choice than other popular methods.

2.6 Summary

The memory hierarchy of many modern, general purpose machines is organized to allow the use of relatively cheap, high capacity DRAM. The low speed of the DRAM-based memory is compensated for by using high-speed SRAM caches. However, the *Memory Wall* points out that the use of caches

cannot result in indefinite scalability. Modern processors are already being limited by the speed of the main memory system and have advanced features to compensate for long external delays.

The architecture of DRAM devices means that the traditional view of memory having a uniform access time no longer holds. Optimizations are possible: memory accesses within different regions of the memory system can have differing latencies depending on the order in which they are received. This reality has led to the exploration of access reordering and other techniques to allow memory operations to be carried out with minimum latency.

Machine learning has also been introduced with a focus on algorithms that can learn the best actions through trial-and-error learning without prior assumptions. In order to be able to work with memory access patterns that are constantly changing over time, such algorithms are preferable to ones which require off-line training and models of either the environment or probabilities.

The next chapter will describe the overall design of the memory controller, which will be based on the DDR SDRAM specifications and previous research related to memory controller design.

Chapter 3

Design of the Memory Controller

The specifications for the memory controller are based on the Intel 82875P MCH memory controller: an integrated system chip-set with a memory controller supporting DDR SDRAM [18] with a 64-bit data bus and clock speeds of $133MHz$, $167MHz$, or, $200MHz$. This commercial product supports two channels of DDR SDRAM and has a maximum memory size of $4GB$ ($2GB$ per channel).

A single channel was chosen for emulation because it captures most of the controller's operations: with two channels, the controller would behave almost identically. Either the controller would behave as if it had a single channel with more devices or the two channels would operate in lock-step (double the bandwidth, but operating in parallel on the same commands). Each channel supports up to 2 double-sided Dual In-Line Memory Modules (DIMMs) and 16 banks, with 4 banks per device (each side acts like a single DIMM). The technology of the devices that are supported are x8 and x16.

The MCH includes many other functions and interfaces, including the front-side bus (FSB) interface: the FSB interface has an in-order queue (IOQ) that can support up to 12 outstanding transactions. The only functions that

are being reverse engineered in this project involve the memory controller.

The generic requirements for the memory controller are:

- DDR SDRAM speed grades of DDR266, DDR333, and DDR400
- Support up to 12 outstanding transactions
- Maximum clock speed of $200MHz$
- Burst size of $4 \times 64\text{bits} = 32\text{bytes}$

The memory controller described here does not have the full functionality of the MCH. This extra functionality requires extra design and is otherwise unrelated to the focus of this thesis, which is the performance of the memory subsystem. The controller in this thesis is designed with some constraints: it has a fixed burst size (to match the size of a cache block typically fetched from main memory when a miss occurs higher up in the hierarchy of caches) and a simpler interface than the Intel 82875P MCH controller. The burst size was selected to match the specifications of the machine on which the address traces used for simulation were collected.

3.1 HDL Specification

The memory control unit and its sub-components are specified using VHDL. The HDL follows conventions of the 1993 IEEE standard on VHDL. VHDL was chosen over other tools because of the ease of use and features: designs can be expressed based on structure and behaviour and modularity is a natural part of the language. The language is also highly standardized and supported by many simulation and synthesis tools. This allows the controller to be used as part of a larger design that can be implemented using a number of technologies for both ASIC and FPGA designs. It could be integrated with other components such as a cache controller or as part of a System-on-Chip, or implemented as a stand-alone component at the board level.

Figure 3.1: Top-Level of Memory Controller

3.2 Design Hierarchy

The top-level view of the memory controller is shown in Figure 3.1. It shows the interfaces to the host and the DDR SDRAM memory as well as the configuration interface.

Internally, the memory controller is composed of several modules. The modularity of the specification makes it easier to perform design and functional testing tasks on smaller subunits. The overall design hierarchy is shown in Figure 3.2.

Figure 3.2: Memory Controller Hierarchy

There are five modules in the top-level hierarchy:

1. Configuration interface
2. Host input interface
3. Control unit
4. Data transfer unit
5. Host output interface

These units are described in following sections.

3.2.1 Configuration Interface

The operation of the controller is governed by control signals (loosely based on the signals of the 82875P MCH [16]) described below:

MCH Configuration Register (`mch_cfg`) This 16-bit register determines the clock frequencies of the system among other things.

DRAM Row Boundary Registers (`bounds`) These 8-bit signals define the upper boundary address of each DRAM *row* (not to be confused with the row addresses in the DRAM banks). Bit 7 of each signal is reserved and must be set to 0. Bits 6 through 0 are compared against address lines 31:26 (0 is concatenated with the address bits 31:26) to determine which row is being addressed. Each signal indicates how many multiples of 64MB are accessible in total from the lowest DIMM. For example, if the rows corresponding to locations 0 and 1 contain 256MB each and the rest are empty, then $bounds[0] = 01h$ and $bounds[1] = 02h$, with the $bounds[2] = bounds[3] = bounds[1]$. The MCH has 8 signals instead of 4 (for two channels).

DRAM Row Attribute Registers (*attrs*) These signals define the page size of each DRAM row. Each signal has three bits which define the page size for the corresponding DRAM row. In the MCH, each signal would be doubled in size (for two channels).

DRAM Timing Register (*dram_timing*) This 32-bit signal defines timing parameters in terms of clock cycles: the Activate to Precharge delay (*tRAS*), the CAS latency (*tCL*), RAS to CAS latency (*tRCD*, time between a bank activation and a read or write command issued to that bank), and the time between a precharge command and bank activation (*tRP*).

DRAM Controller Mode Register (*ctrl_mode*) A 16-bit signal which specifies the refresh period, and special modes used for initialization at power up. *ctrl_mode*[10:8] selects the refresh period (15.6us, 7.8us, 64us, or 64 clocks[fast refresh mode]). *ctrl_mode*[6:4] selects the mode of operation. This signal is driven externally to allow BIOS control over the initialization sequence.

Addressing mode (*mch_mode*) Defines the controller's addressing mode: *linear*, *dynamic*, or *custom*.

Configuration registers with the values of these signals (with the exception of *ctrl_mode*) are loaded by asserting a single control signal (*config_set*) sampled with the positive edge of the clock. To ensure consistent operation, these registers should be loaded shortly after power on before the initialization procedure is carried out. For simulation, they are set by the test bench according to generic parameters. The configuration interface is illustrated in Figure 3.3.

Figure 3.3: Configuration Interface

3.2.2 Host Input Interface

The controller has an input interface that allows the host to send the controller read and write requests. The interface has the following signals:

input_data 256-bit wide input provides data on write operations

input_addr 32-bit wide input provides the address used in read or write operations

input_tag 4-bit input used to uniquely identify each *split transaction*

input_op single bit input used to indicate read or write operation

input_send single bit input used by host to indicate when a request is being sent

input_ack single bit output used to acknowledge send operations

The data, which is equivalent to the size of a complete burst operation ($4 \times 64b$), is transferred in a single clock cycle. This was implemented to model the interface of a real system, which usually has a much higher bandwidth than the DDR SDRAM either by having a higher clock speed or a wider data bus. To avoid the issue of interfacing different clock domains, the host's data path was made wider than the 64-bit DDR SDRAM data bus.

The input interface provides a simple split transaction interface: the host sends operations to the controller with the necessary address information, any necessary data, the type of operation, and a tag to uniquely identify the operation. The input tag must be unique among all outstanding requests to ensure correct operation of the controller, but can be any arbitrary valid bit pattern.

In this simple split-transaction model, this input interface is coupled with an output interface (see Section 3.2.5) that returns the tag on completion along with any data and the operation bit. This allows the controller to

Figure 3.4: Host Input Interface

continue accepting requests while processing already received requests and enables the possibility of concurrency among the requests in exploiting bank parallelism. The requests received in this unit are stored in the register file. The input interface is shown in Figure 3.4.

3.2.2.1 Register File

The input interface is integrated with a register file. The register file is designed to hold up to 12 requests in order to mimic the number of outstanding requests supported by the MCH. The register file is partitioned into data and address portions.

The data portion of the register file is only used to store the data passed

over the input interface as part of a write operation, since read operations will capture data received from the DDR SDRAM memory modules.

The address portion of the register file stores the decoded address as row, column, and bank addresses. The row and column addresses are to be multiplexed over the DDR SDRAM's address lines as part of the commands and are sent with the bank address. The operation bit is stored along with the tag. Unlike the data portion of the register file, the entries are maintained in the order that they arrive by always inserting entries at one end and shifting the contents down to the other end when entries become available. This allows out-of-order completion while maintaining information on ordering constraints to allow the scheduling method to maintain memory consistency.

3.2.2.2 Address Decoding

The host input unit must decode the addresses being sent from the host before storing them in the register file. Some of the address decoding is based on the schemes defined in the manual for the Intel 82875P MCH [16]. The *linear* and *dynamic* modes are based on the corresponding scheme used by the MCH. The third scheme, *custom* is a modification of the linear scheme to increase bank parallelism.

3.2.2.2.1 Selecting the DIMM The high order address bits (31:26) are used to select which DIMM is being addressed. The value of these address bits is compared with the contents of *bounds* to find the correct row (see Section 3.2.1).

For *linear* and *dynamic* modes, the DIMM is selected by comparing the high order address bits until they are less than the contents of the *bounds*: the selected DIMM will correspond to this entry. In these modes, the address space is a simple concatenation of the DIMM's capacities (this is the default behaviour of the MCH).

For the *custom* mode, the DIMM is selected based on a ratio of the

Address Range	DIMM	
	Linear/Dynamic	Custom
00000000 → 03ffff	0	0
04000000 → 07ffff	0	1
08000000 → 0bffff	0	2
0c000000 → 0fffff	0	3
10000000 → 13ffff	1	0
14000000 → 17ffff	1	1
18000000 → 1bffff	1	2
1c000000 → 1fffff	1	3
20000000 → 23ffff	2	0
24000000 → 27ffff	2	1
28000000 → 2bffff	2	2
2c000000 → 2fffff	2	3
30000000 → 33ffff	3	0
34000000 → 37ffff	3	1
38000000 → 3bffff	3	2
3c000000 → 3fffff	3	3

Table 3.1: Example of Mapping Addresses to DIMMs

capacities of each DIMM. The DIMMs are interleaved by ratio: 4 DIMMs of equal size will be interleaved 1 : 1 : 1 : 1 around 64MB intervals. The decoding for all address modes is illustrated in Table 3.1 for the situation where 4 DIMMs of 256MB each are used. $bounds[0] = 04h, bounds[1] = 08h, bounds[2] = 0ch, bounds[3] = 10h$

From the example, it is clear that for the *custom* mode, the address space is more evenly distributed across the memory modules. One of the requirements of this design, based on the specifications of a commercial memory controller, is that the DIMMs may be installed in unequal capacities and in any order. This makes a fixed interleaving scheme difficult. Research on flexible address mappings has previously been done [25] using look-up tables, which would not work in this case since it is not known beforehand what the

memory configuration will be. The *custom* mode uses a relatively flexible scheme that can perform interleaving in some cases (all of the DIMMs have more than 64MB capacity). If one DIMM is 64MB in size, this scheme becomes equivalent to original concatenated method used in the *linear* and *dynamic* modes.

3.2.2.2.2 Selecting Row, Column and Bank Address The *linear* and *dynamic* modes decode the address in the same way as the corresponding modes of the MCH. The *custom* mode is based on the linear mode, but uses lower-order address bits for the bank address, which should increase the number of banks used in a given sequence of memory accesses. These address decodings are shown for the various DIMM and page sizes (the *page* is the region of memory that can be issued reads and writes when a bank is opened; accesses are on 64-bit boundaries) supported by the memory controller in Table 3.2. Addressing modes in Table 3.2 are abbreviated as Lin (Linear), Dyn (Dynamic), and Cus (Custom).

The *custom* scheme is an intermediate between two extremes. The first extreme is to use the lowest-order address bits to select the largest unit (in this case the DIMM) with higher bits selecting the row then column then bank. The second extreme is to use the lowest-order bits to select the bank with higher bits selecting the column then row then DIMM. The first approach leads to the greatest parallelism but will significantly increase the row conflicts and reduce the page hits. The second approach will maximize page hits but will probably not take advantage of bank parallelism. In a study of DRAM performance for a uniprocessor system (the intended target of this research), it was found that the first approach was better for closed-page policy with a large number of banks [4].

Size MB	Page KB	Mode	B_1B_0	Address: $A_{12...0}$	
				Row	Column
64	4	Lin	$a_{13}a_{12}$	$a_{16}a_{15}a_{14}a_{25...17}$	$a_{11...3}$
		Dyn	$a_{12}a_{18}$	$a_{16}a_{13}a_{14}a_{27,26}a_{23...21}a_{25,24}a_{15}a_{17}$	$a_{11...3}$
		Cus	a_6a_5	$a_{16}a_{15}a_{14}a_{25...17}$	$a_{13...7}a_4a_3$
128	4	Lin	$a_{14}a_{13}$	$a_{26}a_{16}a_{15}a_{25...17}$	$a_{11...3}$
		Dyn	$a_{12}a_{18}$	$a_{26}a_{16}a_{13}a_{14}a_{28,27}a_{23...21}a_{25,24}a_{15}a_{17}$	$a_{11...3}$
		Cus	a_6a_5	$a_{26}a_{16}a_{15}a_{25...17}$	$a_{13...7}a_4a_3$
	8	Lin	$a_{14}a_{13}$	$a_{16}a_{15}a_{26...17}$	$a_{12...3}$
		Dyn	$a_{18}a_{13}$	$a_{16}a_{14}a_{26}a_{28,27}a_{23...21}a_{25,24}a_{15}a_{17}$	$a_{12...3}$
		Cus	a_6a_5	$a_{16}a_{15}a_{26...17}$	$a_{14...7}a_4a_3$
256	8	Lin	$a_{14}a_{13}$	$a_{27}a_{16}a_{15}a_{26...17}$	$a_{12...3}$
		Dyn	$a_{18}a_{13}$	$a_{27}a_{16}a_{14}a_{26...21}a_{29,28}a_{15}a_{17}$	$a_{12...3}$
		Cus	a_6a_5	$a_{27}a_{16}a_{15}a_{26...17}$	$a_{14...7}a_4a_3$
512	16	Lin	$a_{15}a_{14}$	$a_{28}a_{16}a_{27...17}$	$a_{13...3}$
		Dyn	$a_{14}a_{18}$	$a_{28}a_{16}a_{27...21}a_{30,29}a_{15}a_{17}$	$a_{13...3}$
		Cus	a_6a_5	$a_{28}a_{16}a_{27...17}$	$a_{15...7}a_4a_3$

Table 3.2: Bank, Row and Column Decoding

Figure 3.5: Control Unit

3.2.3 Control Unit

The control unit sends signals internally to the host interface unit and data transfer unit to indicate when a read or write command has been issued and externally to the DDR SDRAM in the form of commands such as reads, writes, and bank activations.

The control unit is itself organized into sub-components (see Figure 3.5):

1. Internal Models
2. Refresh Unit
3. Control Arbiter
4. Transaction Execution Unit

3.2.3.1 Internal Models

The operation of the execution units (transaction execution and refresh) must be controlled according to various timing parameters, such as the ones defined in the JEDEC DDR standard [18]. To allow for an easier design problem, some of the external components are modelled inside the controller. These models receive control signals from the other execution units and provide feedback signals indicating their state. The memory controller's operation revolves around this continuous stimulation and feedback.

3.2.3.1.1 Bank Models In a real DDR-based memory system, banks are organized into groups of four: there are four banks per chip in supported DDR SDRAM devices. These chips are combined in groups to supply the 64-bit data bus: 8 devices for x8 chips and 4 devices for x16 chips. These groupings are called *rows*, which should not be confused with the row address in each bank. The chips in a row all act in unison and undergo the same state changes simultaneously, so one bank model can represent the entire row.

The bank module is represented by a VHDL entity. This entity has the usual clock and reset signals and has some of the configuration registers as inputs to be used to determine timing parameters.

Another set of signals allow the execution units to stimulate and gather feedback from the model such as performing precharge operations or determining the bank's status (unknown, idle, or open). The 'unknown' state is the reset or power up state in which the bank must be precharged.

The general design of the bank model is to set timers based on the control signals and update the status flags based on the timer values. For example, the precharge command resets a timer counting down tRP. The bank can be opened when this timer is expires.

In this bank model, reads and writes cannot be interrupted (this allows the unit generating the control signals to be much simpler). A state diagram of this entity is shown in Figure 3.6.

Figure 3.6: State Diagram of Bank Model

Figure 3.7: State Diagram of Row/DIMM Model

3.2.3.1.2 DIMM Models While the banks in the row operate in a lock-step fashion, the rows are further organized into physical devices. For example the first four rows will correspond to the four banks per DDR SDRAM chip with 4 or 8 chips forming each row (DIMM). There are additional timing constraints of the DDR SDRAM chip which have to be modelled in addition to the independent operation of each bank.

For example, when one bank in a device is issued a write command, there is a minimum period during which reads cannot be issued to any other bank in the chip ($tWTR$ period). Such timing constraints are followed by representing the devices in a row with a single model. $tRRD$, the minimum period between opening different banks in the same device, is another timing constraint that is modelled in this entity. A state diagram for this model is shown in Figure 3.7.

3.2.3.1.3 Bus Model The data bus is another entity which has to be modelled to prevent different transaction from overlapping on the bus. The design is a simple finite state machine using the same timer-based approach

Figure 3.8: State Diagram of Bus Model

used in the previous models (see Figure 3.8).

3.2.3.2 Refresh Unit

The refresh unit generates the initialization sequence responses under external control and periodically refreshes all banks simultaneously during normal operation. It uses the outputs of all the models to provide the necessary timing for its actions.

On reset, the unit enters the initialization state. In this state, the commands sent to the DDR SDRAM are controlled completely by an external source which is defined in the 82875P manual [16] as the BIOS. The BIOS is responsible for the power-on sequence and controls the refresh unit by setting the value of DRC (see Section 3.2.1). The BIOS signals the end of the

power-on sequence by setting the value of this register to '111' which tells the unit to resume normal operation.

The normal operation of the unit is to reset the timer and decrement it: when the timer reaches zero, all banks are precharged and then an auto refresh command is sent to all banks. The timer's reset value is equal to the refresh period minus the time it will take to precharge all banks and go into the idle state.

3.2.3.3 Control Arbiter

When the refresh timer is equal to zero, the output of the transaction execution unit to the DDR SDRAM is inhibited and output instead comes from the refresh unit. This is done by the Control Arbiter which switches between the output of refresh and transaction execution units when the refresh timer reaches zero.

3.2.3.4 Transaction Execution Unit

The transaction execution unit, like the refresh unit, uses the outputs of all the models to provide the necessary timing for its actions and provides further commands for the models to simulate. Only this unit and the refresh unit drive the SDRAM bus unit interface's address bus signals.

The transaction execution unit is implemented by using multiple VHDL architectures, one for each implementation. The behaviour of each architecture varies according to the Page Policy, Scheduling Policy, and Bank Arbitration.

3.2.3.4.1 Page Policy There are three possible implementation of the page policy: closed page policy, open page policy, and a mixed page policy (combination of open and closed page policies).

Closed page policy is implemented by using auto-precharge commands when performing reads or writes: the page is closed at the end of the oper-

ation. New operations must always open the bank after an auto-precharge read/write command.

Open page policy is implemented using ordinary read and write commands without auto-precharge, so the bank is left in the "open" state.

Mixed page policy selectively applies the auto-precharge to some commands. It is possible to implement a mixed policy using static or dynamic methods.

3.2.3.4.2 Scheduling Policy There are two scheduling (inter-bank arbitration) policies used in the memory controller: First-In-First-Out (FIFO), and First-Ready (FR).

For the FIFO policy, the actual read and write commands must be issued in-order, which is the simplest way of ensuring memory consistency. However, other commands (such as precharge) may occur out-of-order, which allows some command latency hiding.

The FR policy removes the restriction that the reads and writes must occur in order. The first command that can complete (the bank is ready for the operation and the enough time is left before a refresh cycle) is executed. Commands are chosen from the set of memory accesses that have been chosen by intra-bank arbitration (see below, referred to as "bank arbitration" onwards).

3.2.3.4.3 Bank Arbitration There are two forms of bank arbitration: FIFO and Page-Hit.

FIFO arbitration uses the in-order arrival of requests to assign banks. If requests (with relevant row address) $A_1(2000)$, $A_2(500)$, and $A_3(600)$, are all queued for the same bank, they will be assigned the bank in that order regardless of the bank's state (open or closed). This ensure that the contents of memory will not differ from the intended order.

Page-Hit (PH) arbitration assigns banks by giving priority to those transactions which have page hits. If two requests have a page hit (or they both

have page misses), FIFO is used, which ensures consistency. If one request is a hit and another is a miss, then the requests *must* be for different addresses and are safe to reorder.

PH is intended to maximize page hits. For example, suppose the following memory operations are queued for the same bank : $A_1(1000)$, $A_2(2000)$, $A_3(1000)$, $A_4(500)$, $A_5(2000)$, $A_6(500)$. Assuming the bank is in the closed state initially, the reordered assignment will be: $A_1(1000)$, $A_3(1000)$, $A_2(2000)$, $A_5(2000)$, $A_4(500)$, $A_6(500)$. Note that accesses to the same rows occur in the same order as FIFO, but row accesses are clustered together.

When the bank is in the initial closed state, the requests are all page opens and have the same priority, so FIFO is used. $A_1(1000)$ is the first access and is the first one chosen. When row 1000 has been activated, all requests for that row address are prioritized over page misses. $A_3(1000)$ is the second request chosen. When all the requests using row 1000 are processed, the remaining requests are all page misses and have the same priority: FIFO is again used with the remaining accesses: $A_2(2000)$, $A_4(500)$, $A_5(2000)$, $A_6(500)$. The next access is $A_2(2000)$, which activates row 2000. After A_2 , the page is open and the active row is 2000 and $A_5(2000)$ now has the highest priority. After no more page hits can be taken advantage of for row 2000, FIFO is again used because A_4, A_6 are both page misses and the same priority. $A_4(500)$ is the next operation chosen. Only one operation remains, so $A_6(500)$ is chosen.

This simplified example assumes no accesses arrive in the queue for this bank once this initial set has arrived. If more accesses for row 1000 had arrived before A_3 had finished, then A_2 would have been preempted because these new accesses would have had a higher priority. To prevent starvation, a maximum of 12 such preemptions are allowed before FIFO is used to force progress of the preempted operations (this parameter is independent of the queue size).

```

While true:
    If requests are in register file:
        Perform bank arbitration (FIFO or PH) using bank's status
        Choose highest priority, ready request (FIFO or FR)
        If refresh timer will not expire:
            Send commands to DDR SDRAM/internal models
        Else
            Send NOP to DDR SDRAM
    Else
        Send NOP to DDR SDRAM
End

```

Figure 3.9: Generic Scheduling Procedure

3.2.3.4.4 Generic Scheduling Algorithm All of the controller implementations in this thesis have the same basic scheduling algorithm (shown in Figure 3.9) that is tailored to implement the different policies.

The transaction execution unit has several standard implementation, based on combinations of these policies: OFIFO (open page, FIFO scheduling and bank arbitration), CFIFO (closed page, FIFO scheduling and bank arbitration), CPHFR (mixed page policy: bank is closed when no more reordering is possible, first-ready scheduling, PH bank arbitration), and OPHFR (bank is left in open state when no more reordering is possible, first-ready scheduling and PH bank arbitration).

3.2.4 Data Transfer Unit

This unit is directed by the control unit to send and receive data from the DDR SDRAM. Commands are latched with any data (for write operation) and transfers occur in-order (the control unit decides the order of operations). Data read from the DDR SDRAM is buffered and returned to the host output

Figure 3.10: Data Transfer Unit

interface. All operations are sent with the original operation bit and tag bit-field to the host output interface when their data transfer phase is complete.

3.2.5 Host Output Interface

The unit (see Figure 3.11) receives commands from the data transfer unit indicating when reads and writes that have been issued to the DDR SDRAM are complete. The unit buffer up to two requests to allow the controller to store the status of a request while the host is reading the status of the previous request. The operation bit and tag and stored in this unit are received by the host so that the host can know when an outstanding transaction has been completed (its data has been transferred to or from the main memory). For

Figure 3.11: Host Output Interface

reads, the data output port will have the data read from the DDR SDRAM to be returned to the host.

3.3 Summary

The basic architecture describe in this chapter is the same used for the new controller designs described later. The major difference between the controller implementations is in the Transaction execution unit: all other units remain the same and the overall architecture is unchanged.

The variants described here are examples of memory access scheduling and page policies for standard memory controllers. In the next chapter, a new approach will be discussed for memory controller design.

Chapter 4

Dynamic Page Policy

In this chapter, the standard memory controller will be augmented with a machine learning algorithm in order to implement a dynamic page policy. Such a memory controller will be able to dynamically adjust the page policy to better match the memory access patterns as they occur.

Before discussing the optimization proposed in this work, it is necessary to highlight some of the performance issues in a DDR SDRAM memory system. First, some basic terminology needs to be established for *page rates*.

4.1 Page Rate Definitions

There are three classifications of memory operations as they occur in a DDR SDRAM memory system: page hits, page misses, and page opens.

A *page hit* occurs when an operation is issued to a bank which is already open with the needed row already activated. An operation that experiences a page hit has the smallest possible latency as only the read or write command has to be sent to the DDR SDRAM (only a single command).

A *page miss* occurs when an operation is issued to a bank which is already open but has a different row active from the one needed by the operation. In this case, the operation has the maximum latency: the bank must first be

precharged (to close the page), reopened with the correct row activated, and finally the read or write command can be sent to the DDR SDRAM (three commands in total).

A *page open* incurs an intermediate latency for operations that are issued to closed banks: the bank must be opened to activate the needed row and then the read or write command can be issued (two commands in total).

The relative proportions of these operation classes will affect the performance of the memory system because of the non-uniformity of the memory system and the resulting differing latencies.

4.2 Page Rates and DDR SDRAM Performance

To maximize the controller's performance, the stream of requests should have as high a page hit rate as is possible. If page hits are not possible, page opens (intermediate latency) are better than page misses (maximum latency) so it is better to close pages by precharging them when no more page hits can be exploited.

However, this simple performance model is only applicable when there is no overlapping of commands. Consider the decision of whether or not to use the auto-precharge version of a read or write command. Without auto-precharge, the page is left open: a subsequent page miss will need to precharge the bank, reopen the bank and then issue the next read or write command to the bank. If auto-precharge is used, then this page miss will instead be a page open and only two commands will have to be sent to the DDR SDRAM.

The latency of the auto-precharge command is the same as the latency of a normal read or write command followed by a precharge command issued at the earliest possible time without losing any data in the burst [18]. If enough advance notice is given to the controller (for example, if requests are

queued), the extra latency of a page miss compared to a page open can be completely hidden by having the controller issue the precharge command at the earliest possible time, which will result in the same overall latency as using auto-precharge.

Using auto-precharge does not decrease the minimum possible latency if the following operation to the bank is a page miss. However, using auto-precharge results in lower address bus bandwidth requirements (only two commands need to be sent instead of three) and possibly lower power requirements (this aspect is not explored in this work). Using auto-precharge can allow more commands to be sent to the memory system concurrently which results in more parallelism and command overlapping.

The subject of this chapter is to describe a general method for choosing an optimal strategy of open page or closed page (using the auto-precharge feature) based on the previous behaviour of memory accesses. The chosen method is based on a reinforcement learning algorithm.

4.3 Reinforcement Learning Algorithms

Reinforcement learning algorithms include Dynamic Programming, Monte Carlo methods and *Temporal Difference* (TD) learning [23]. Dynamic programming requires that the environment be perfectly modelled as a Markov Decision Process (MDP) and requires large amounts of calculations to evaluate the state set of the MDP. Monte Carlo methods and TD learning were devised to solve the same problems as dynamic programming using much less computation and without using a perfect model. All of these algorithms use *value functions* to evaluate states or state-action pairs.

Monte Carlo methods require several iterations (an entire *episode*) before the learning can be applied but TD methods provide immediate responses. Because of the fast response time, TD algorithms were explored for this thesis. TD algorithms consider temporally successive states and actions for

---

```

Initialization: set  $Q(s, a)$  arbitrarily
For each episode:
    Initialize state,  $s$ 
    Choose action,  $a$ , using policy,  $\pi$ , derived from  $Q(s, a)$ 
    For each step of episode:
        Take action  $a$ ; observe reward,  $r$ ; and next state,  $s'$ 
        Choose next action,  $a'$ , using  $\pi$  derived from  $Q(s', a')$ 
        Update  $Q(s, a)$ 
             $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
        Update present state and action
             $s \leftarrow s', a \leftarrow a'$ 
    Until  $s$  is terminal
End

```

Figure 4.1: Sarsa Algorithm

learning.

A control algorithm, which selects the optimum policy, was of main interest: an algorithm was needed to select the auto-precharge functionality when needed. A previously developed control algorithm is *Sarsa* [23]. Sarsa is an *on-policy* TD control algorithm. It is on-policy because the actions are chosen by a policy which is updated as learning progresses [23]. The algorithm is given in Figure 4.1.

Alpha ($0 \leq \alpha \leq 1$) is the step-size parameter: this controls the learning rate. As $\alpha \rightarrow 0$, the agent learns more slowly. Gamma ($0 \leq \gamma \leq 1$) is the discount rate and determines the present value of future rewards. As $\gamma \rightarrow 1$, the agent takes futures awards into account more strongly. $Q(s, a)$ is the value of selecting action a given state s .

The update of $Q(s, a)$ is done by using the reward, the learning parameters (α and γ), the current value of $Q(s, a)$, and the value of the next state-action pair, $Q(s', a')$.

There are several strategies choosing a from s (known as the *policy*, π). Examples of π include:

Greedy The best action is always selected

$$Q(s_i, a_i) = \max(Q(s, a)) \Rightarrow a' = a_i$$

ϵ -Greedy The best action is usually chosen, but with a small probability, ϵ , a random action is chosen

$$\begin{aligned} p > 1 - \epsilon : & \quad Q(s_i, a_i) = \max(Q(s, a)) \Rightarrow a' = a_i \\ p \leq 1 - \epsilon : & \quad a_j = \text{rand}(a) \Rightarrow a' = a_j \end{aligned}$$

The method of choosing the action should be such that the policies are all eventually tried in order to learn the most desirable policy: this is known as General Policy Iteration (GPI) [23]. The policy choice can improve learning performance, but a policy which is best in one case is not necessarily the best choice for another problem. There are several other policies besides Greedy and ϵ -Greedy.

4.4 Controlling Page Policy

To implement a TD algorithm for controlling the page policy, the algorithm needs to have certain variables mapped to the problem. The TD algorithm used to control page policy in this thesis is based on *Sarsa*.

The relevant *state* in the problem is the active row in an open bank. Using this information as the basis for the state information in the problem will allow the algorithm to select better states which will directly correspond to the rows in the bank. The algorithm will learn which rows should be kept open and which rows in the bank should be closed once the read or write command has been issued. Thus, the algorithm should allow the controller

to dynamically select the page policy depending on previous outcomes (page hits or misses).

The *action* to be taken by the controller will be whether to leave a bank open by using a normal read or write command or to close the bank by using the auto-precharge read or write command supported by DDR SDRAM. With reordering to maximize page hits, the controller algorithm is based on the CPHFR algorithm described in Chapter 3 with one simple modification: instead of always closing the page when no more page hits are visible, the controller will use the learning algorithm to determine which action is best. For FIFO, the control algorithm is applied to all read or write commands.

The *policy* to be used to select the action will be Greedy (the best action will always be chosen, ties select open page policy). This is the simplest policy to implement (others require a pseudo-random component) although it does not allow as much exploration as other policies. However, because the exploration space is small (only two action are possible in any state), a Greedy policy can still be effective because the rewards fed back into the algorithm will eventually affect the algorithm's choices. The 'best' action is defined as the one which has a more positive $Q(s, a)$: for a given row r , $Q(r, 0) \geq Q(r, 1) \Rightarrow$ open page, $Q(r, 0) < Q(r, 1) \Rightarrow$ closed page.

The *reward* will be positive when the outcome is as expected. If the page is left open and the same row is used on the next bank access, then the reward is positive. If the page has been closed and a different row is used on the next bank access, the reward is also positive because the correct prediction had been made. When the prediction is wrong (the page is left open and the next access is a page miss or the page is closed and the same row is used in the next access), the reward is negative. The reward is used to update $Q(s, a)$: the action (normal or auto-precharge) which accumulates more positive rewards over time will eventually become preferred for that row.

A *learning episode* starts with a reset of the controller. The episode

Parameter	Assignment
s	the active row (13-bits long, zero-filled for smaller page sizes)
a	Column address bit A_{10} : $a = 0 \rightarrow$ normal read/write, $a = 1 \rightarrow$ auto-precharge read/write
$r \in \{+1, -1\}$	reward for {prediction, mis-prediction}
$Q(s, a)$	$3.0 \geq Q(s, a) \geq -3.0$
α	generic, values of 0.125, 0.5 and 0.875
γ	generic, values of 0.125, 0.5 and 0.875
π	Greedy

Table 4.1: Sarsa Parameters for Controlling Page Policy

continues until the controller is reset again. The parameters used in the Sarsa algorithm to control the page policy are in Table 4.1.

The dynamic page policy will use a reinforcement learning algorithm to learn, as the controller accepts requests, the appropriate page policy based on the history of the accesses. The policy will leave the page open for those page/row combinations which have encountered significant positive reinforcement in the past. Pages which have experienced page misses will eventually use a closed page policy. By using trial-and-error searching, the learning algorithm should be able to determine whether and open or closed page policy is a better choice based on previous actions and observed rewards.

The aim of the learning algorithm is to maintain a high page hit rate while converting page misses into page opens using the auto-precharge feature. This will reduce the latency of scheduling concurrent operations and make the performance of the memory controller more resilient if the latencies of page misses cannot be completely hidden. By using an adaptive algorithm, the controller should be able to maintain good performance across different applications and loads without needing manual readjustment.

There are two methods of implementing the *Sarsa* algorithm to control the page policy: using table look-ups and using a function-approximation

method.

4.4.1 State-Action Table

The most straightforward implementation of Sarsa uses a table to store $Q(s, a)$ values. The maximum storage requirement for state and actions, S_{TMAX} , is given by:

$$S_{TMAX} = (S_{Q(s,a)} \times \text{number of states} \times \text{number of actions})$$

Since number of states = number of banks $\times$ number of rows per bank = 16×2^{13} , number of actions = 2, this gives:

$$S_{TMAX} = (S_{Q(s,a)} \times 2^4 \times 2^{13} \times 2) = 2^{18} S_{Q(s,a)}$$

4.4.2 Generalization: Function Approximation

To reduce the amount of storage needed by the algorithm, $Q(s, a)$ can be approximated by a function instead of using a table to store all possible state-action combinations [23, 22]. By using linear gradient-descent-based function approximation, the storage requirements can be substantially reduced.

In linear function approximation, states are represented by feature vectors and the value function is a linear function of a vector of parameters. Corresponding to every state s , there is a vector of features $\vec{\phi}_s = (\phi_1, \phi_2, \dots, \phi_n)$. For this problem, the set of features will be determined for each DDR SDRAM bank (just as there are state-action tables for each bank in the previous implementation). Q is represented as a linear function of the parameter vector $\vec{\theta} = (\theta_1, \theta_2, \dots, \theta_n)$.

One linear method of function approximation is *Tile Coding*, where the state space is exhaustively partitioned into sub-regions called tiles. Each tile is a receptive field for one binary feature. The use of binary features simplifies

the calculations in the learning algorithms as well. Tile Coding was selected for this thesis.

For this problem, the number of features will be n , where n is the maximum number of bits in the row address. The bits of the row address will be combined with the action bit using an exclusive OR operation to form the tile address. This mapping is intended to combine the row (state) information with the action (A_{10}) to obtain the features for the state/action pair. For example, if the current active row is 1000011000101, then

- if the action is to keep the page open, $A_{10} = 0$, then $F(s, a) = 1000011000101$,
- if the action is to close the page open, $A_{10} = 1$, then $F(s, a) = 0111100111010$.

It is possible for multiple state/action pairs to share the same features: if the active row is 0111100111010 and the action is to keep the page open, the features $F(s, a) = 0111100111010$ are the same as the features for the previous example when the page is kept closed.

The use of tile coding reduces the distinctions between different state/action pairs. For this implementation, complete overlap only occurs for row addresses which are the bit-wise compliments of each other and for opposite actions. However, rows which have many of the same bits will partially overlap. In each step, the algorithm will update $\vec{\theta}$ for only one feature vector representing one state-action pair, but this learning will affect all state-action pairs that share similar features and have many bits in the same locations. This generalization makes learning less precise but requires much less storage space. Generalization can also help improve the outcome if rows with similar bits tend to have the same behaviour—learning which occurs for one should also be applied to similar rows.

A version of Sarsa used with function approximation is given in Figure 4.2. The features of the next and previous state/action pairs are used to update

---

```

Initialize  $\vec{\theta}$  arbitrarily
For each episode:
    Initialize state,  $s$ 
     $\forall a \in A(s)$ :
         $\vec{F}_{s,a} \leftarrow \text{features}(s, a)$ 
         $Q_{s,a} = \sum_{i \in \vec{F}_{s,a}} \theta(i) = \vec{F}_{s,a} \cdot \theta(i)^T$ 
    Choose action,  $a$ , using policy,  $\pi$ , derived from  $Q_{sa}$ 
    For each step of episode:
         $\vec{F}_{s,a} \leftarrow \text{features}(s, a)$ 
        Take action  $a$ ; observe reward  $r$ , next state  $s'$ 
         $\delta \leftarrow r - Q_{sa}$ 
         $\forall a' \in A(s')$ :
             $\vec{F}_{s',a'} \leftarrow \text{features}(s', a')$ 
             $Q_{s',a'} = \sum_{i \in \vec{F}_{s',a'}} \theta(i) = \vec{F}_{s',a'} \cdot \theta(i)^T$ 
        Choose  $a'$  using policy,  $\pi$ , derived from  $Q_{s',a'}$ 
         $\delta \leftarrow \delta + \gamma Q_{s',a'}$ 
         $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \nabla_{\vec{\theta}} Q_{s',a'} = \vec{\theta} + \alpha \delta \vec{F}_{s,a}$ 
         $s \leftarrow s', a \leftarrow a'$ 
    Until  $s$  is terminal
End

```

Figure 4.2: Linear Gradient-Descent Sarsa

the parameter vector which is analogous to using the next and previous table entries to update the state-action table for the table-based implementation.

Delta, δ , is the *temporal difference error*. If $\delta > 0$, then the tendency should be strengthened. If $\delta < 0$, then the tendency should be weakened. The size of parameter vector, $\vec{\theta}$, is equal to the size of the feature vector $\vec{F}_{sa}$. $\vec{F}_{sa} = \nabla_{\vec{\theta}} Q_a$, which is the gradient of the value function.

The maximum storage requirements for state and actions in the modified

Sarsa algorithm is now:

$$\begin{aligned} S_{TMAX} &= (\text{number of banks} \times S_{\theta}) \\ &= (16 \times 13 \times S_{\theta(i)}) \\ &= 208S_{\theta(i)} \end{aligned}$$

The use of the parameter vector for each bank has significantly reduced the amount of state-action information that must be stored.

4.4.3 Learning and Access Reordering

When memory operations are performed in-order, the learning algorithm is used to determine the page policy for each access. However, when accesses are being reordered to maximize the page hit rate, the algorithm only has to make a decision when no more page hits are available.

The algorithms can still use the information provided by the reordered accesses as learning *simulations*. While the algorithm is no longer being used for control, rewards can still be observed and the value of state-action pairs can be reinforced during reordering. The alternative is to only use the algorithm when a decision has to be made when reordering is finished.

For the implementations used in this work, the algorithms always observe rewards, even when reordering is taking place and the algorithm is no longer being used for control. This results in a less complicated specification because there are fewer conditionals: extra information is needed to detect when reordering is being done. This decision may significantly affect learning performance but this can be explored for future work.

4.4.4 Synthesizeable Implementation

The algorithms described above require floating point numbers. Because the algorithms are being implemented using VHDL, there is an issue regarding

the ability of synthesis tools to produce working implementations. Since the default floating point type (*real*) is not synthesizable by most tools, a fixed point approximation using vectors of bits was chosen for use in the algorithms. A pre-existing package, slated to become part of the next VHDL standard (VHDL 2000X), was chosen [10].

A version compatible with the VHDL 1993 standard was used to implement $Q(s, a)$. $Q(s, a)$ is represented using 19 bits: 3 bits (including a sign bit) for the integer portion of the number and 16 bits for the fractional portion of the number. The size of each element of the parameter vector, $\theta(i)$, is equal to the size of each entry $Q(s, a)$ used in the table-based implementation. This gives the following storage costs for the learning algorithm implementations:

Implementation	S_{TMAX} , bits
Table-based	$2^{18} S_{Q(s,a)} = 4980736$
Function Approximation	$208 S_{\theta(i)} = 3952$

This representation was arbitrarily chosen with enough range to permit several rewards to accumulate over time. The sensitivity of the algorithms to the precision of the numbers has not been evaluated. It is possible that a different number representation could affect the results of learning depending on how quickly this fixed point representation saturates and the loss of accuracy in computations. This could be explored in future work. If less precision is needed for comparable performance, then the size of these bit vectors can be reduced.

The new memory controller implementations, based on these numerical representations, are: reinforcement learning with FIFO (RLFIFO), approximated reinforcement learning with FIFO (ARLFIFO), reinforcement Learning with PHFR (RLPHFR), and approximated reinforcement learning with PHFR (ARLPHFR).

4.5 On-line Learning and Control

The learning algorithm, in both table-based and function approximation implementations, is intended to discover the best behaviour and even follow a moving target by continuous adaptation. If the algorithm makes mistakes (such as leaving a page open when it should have been closed), they will be reflected in a change its behaviour. An action which is resulting in negative feedback will eventually not be used in these implementations. Learning continues as long as the controller is in operation, but given enough rewards the number system will saturate (see Section 4.4.4) and learning will effectively stop. One way to illustrate the operation of the algorithm is by using a numerical example for a given sequence of memory accesses.

Because the learning algorithm learns whether to leave a bank open or closed based on the history of memory accesses independently of other banks, only one bank will be considered in this example. A sequence of column addresses will be processed using both algorithms to show the internal state changes and the control choices made as learning progresses. Starting from the initial state (all values in the state-action table or $\vec{\theta}$ are zero), it will be assumed that the following row addresses will be accessed (in the given order) in the bank: 5000, 5000, 5000, 100, 1000, 5000, 2000, and 100.

The actions will be: 0 (leave page open) and 1 (use auto-precharge). The reward will be: 1 for successful prediction, -1 for mis-prediction, 0 for initial step. The learning parameters are: $\alpha = 0.5, \gamma = 0.125$. The updates will follow the equations given in Figure 4.1 and Figure 4.2 for each respective implementation. The examples are given for each implementations in the following Sections (see Table 4.2 and Table 4.3).

4.5.1 Table-Based Example

Table 4.2 details each step of the *Sarsa* algorithm when the sequence of row address accesses is simulated for a bank. Two addresses are off interest: 5000

and 100.

The first access is used as a starting point so the algorithm does not consider it a success or failure and there is no reward. Because $Q(5000, 0) = Q(5000, 1) = 0$ (after initialization, all table entries are zero), the algorithm chooses to leave the page open. Two page hits are then observed and each time the algorithm chooses to leave the page open. This increases the value of $Q(5000, 0)$. One page miss for the row is then observed when 100 becomes the row address being accessed. When address 5000 is encountered again, the previous experience has accumulated: $Q(5000, 0)$ is now only slightly positive, but it is still greater than $Q(5000, 1)$ and so the page is left open (the action bit is '0'). When another page miss is experienced because the next address is 2000, $Q(5000, 0)$ is updated and becomes negative (the reward is -1 because the page was left open and a page miss occurred).

The first occurrence of 100 comes after the second to last access of address 5000. Because $Q(100, 0) = Q(100, 1) = 0$ (nothing has been observed yet), the algorithm leaves the page open because $Q(100, 0) \geq Q(100, 1)$. This is the wrong prediction, because the next address to be accessed is 1000. This results in $Q(100, 0)$ becoming negatively reinforced and its new value is -0.125 . On the next access of address 100, $Q(100, 0) < Q(100, 1)$ so the action becomes '1' and auto-precharge is used to close the bank when the access is completed. If the next access to the bank is for the same row address, this prediction would be wrong and a negative reward would result. If the next access is for a different bank, this prediction would be correct and a positive reward would result.

The sequence of predictions and outcomes is used to update the table continuously so that previous experience is used to make decisions on what future actions to take. The reward cannot be observed until the next row access occurs. This means that when each new address is accessed, the *value* of the last state-action pair is updated. The action is chosen in the same step, but the reward will not be observed until the next row access occurs.

The temporal occurrence of states and actions is also relevant: the value of the next state-action pair is used to update the value of the last state-action pair. This means that the transitions between state-action pairs also has an effect on learning (controlled by the discount rate, γ). Other reinforcement learning algorithms can use these *back-ups* by going further back along the chain of state-action pairs [21, 23].

Table 4.2: Control using State-Action Table

Row	Current State	Algorithm Step
5000	$Q = (0, 0, \dots, 0)$ $a = \text{N/A}$ $s = \text{N/A}$	$s' = 5000$ $Q(5000, 0) \geq Q(5000, 1) \Rightarrow a' = 0$ $r = 0$ (no previous actions) $Q = Q$ (no change)
5000	$Q = (0, 0, \dots, 0)$ $a = 0$ $s = 5000$	$s' = 5000$ $Q(5000, 0) \geq Q(5000, 1) \Rightarrow a' = 0$ $r = 1$ (page open, page hit) $Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$ $Q(5000, 0) = 0 + 0.125(1 + 0.5(0) - 0)$ $Q(5000, 0) = 0.125$
5000	$Q(5000, 0) = 0.125$ $a = 0$ $s = 5000$	$s' = 5000$ $Q(5000, 0) \geq Q(5000, 1) \Rightarrow a' = 0$ $r = 1$ (page open, page hit) $Q(5000, 0) = 0.125 + 0.125(1 + 0.5(0.125) - 0.125)$ $Q(5000, 0) = 0.242$
continued on next page		

Table 4.2: Control using State-Action Table

Row	Current State	Algorithm Step
100	$Q(100, 0) = 0$ $Q(100, 1) = 0$ $Q(5000, 0) = 0.242$ $a = 0$ $s = 5000$	$s' = 100$ $Q(100, 0) \geq Q(100, 1) \Rightarrow a' = 0$ $r = -1$ (page open, page miss) $Q(5000, 0) = 0.242 + 0.125(-1 + 0.5(0) - 0.242)$ $Q(5000, 0) = 0.0869$
1000	$Q(1000, 0) = 0$ $Q(1000, 1) = 0$ $Q(100, 0) = 0$ $a = 0$ $s = 100$	$s' = 1000$ $Q(1000, 0) \geq Q(100, 1) \Rightarrow a' = 0$ $r = -1$ (page open, page miss) $Q(100, 0) = 0 + 0.125(-1 + 0.5(0) - 0)$ $Q(100, 0) = -0.125$
5000	$Q(1000, 0) = 0$ $Q(5000, 0) = 0.0869$ $Q(5000, 1) = 0$ $a = 0$ $s = 1000$	$s' = 5000$ $Q(5000, 0) \geq Q(5000, 1) \Rightarrow a' = 0$ $r = -1$ (page open, page miss) $Q(1000, 0) = 0 + 0.125(-1 + 0.5(0.0869) - 0)$ $Q(1000, 0) = -0.120$
2000	$Q(5000, 0) = 0.0869$ $Q(2000, 0) = 0$ $Q(2000, 1) = 0$ $a = 0$ $s = 5000$	$s' = 2000$ $Q(2000, 0) \geq Q(2000, 1) \Rightarrow a' = 0$ $r = -1$ (page open, page miss) $Q(5000, 0) = 0.0869 + 0.125(-1 + 0.5(0) - 0.0869)$ $Q(5000, 0) = -0.049$
continued on next page		

Table 4.2: Control using State-Action Table

Row	Current State	Algorithm Step
100	$Q(100, 0) = -0.125$ $Q(100, 1) = 0$ $Q(2000, 0) = 0$ $a = 0$ $s = 2000$	$s' = 100$ $Q(100, 0) < Q(100, 1) \Rightarrow a' = 1$ $r = -1$ (page open, page miss) $Q(2000, 0) = 0 + 0.125(-1 + 0.5(0) - 0)$ $Q(2000, 0) = -0.125$

4.5.2 Function Approximation Example

The working of the function approximation version of *Sarsa* is shown for the row address accesses in Table 4.3. For reference, the binary features of the rows are:

Row Address	Binary Features
5000	1001110001000
100	0000001100100
1000	0001111101000
2000	0011111010000

The major difference is that instead of a table, $\vec{\theta}$ is updated with each learning step. Notice that with function approximation, state-action pairs are evaluated based on the accumulated rewards for the relevant features. Based on the features, each element in $\vec{\theta}$ can be updated by several state-action pairs instead of just one.

After the initial consecutive accesses of address 5000, the features for closed page are overall more positive than the features for open page when address 100 follows 5000. This occurs despite more page hits being experienced for address 5000 than misses. A future improvement of the algorithm could involve adding more features so that the features associated with open

page policy are distinct from those for closed page policy. Less interference should improve the algorithm's performance.

As with the table-based implementation, previous experience is used to choose future page policy. However, because of the particular selection of features, the function approximation version of the algorithm tries the closed page policy more often and gets more positive reward compared to the table-based implementation with this particular sequence of address accesses. The overall performance of the algorithm will be explored in Chapter 6: it will be determined how this generalization impacts the algorithm's performance compared to the table-based implementation.

Table 4.3: Control using Function Approximation

Row	Current State	Algorithm Step
5000	$\vec{\theta} = (0, 0, \dots, 0)$ $s = N/A$ $a = N/A$ $F_{s,a} = N/A$	$s' = 5000$ $F_{s',0} = 1001110001000$ $F_{s',1} = 0110001110111$ $Q_{5000,0} = \vec{\theta}^T \cdot F_{5000,0} = 0$ $Q_{5000,1} = \vec{\theta}^T \cdot F_{5000,1} = 0$ $Q_{5000,0} \geq Q_{5000,1} \Rightarrow a' = 0$ $F_{s',a'} = F_{5000,0}$ $r = 0$ (no previous actions) $\vec{\theta} = \vec{\theta}$ (no change)
continued on next page		

Table 4.3: Control using Function Approximation

Row	Current State	Algorithm Step
5000	$\vec{\theta} = (0, 0, \dots, 0)$ $s = 5000$ $a = 0$ $F_{s,a} = 1001110001000$	$s' = 5000$ $F_{s',0} = 1001110001000$ $F_{s',1} = 0110001110111$ $Q_{5000,0} = \vec{\theta}^T \cdot F_{5000,0} = 0$ $Q_{5000,1} = \vec{\theta}^T \cdot F_{5000,1} = 0$ $Q_{5000,0} \geq Q_{5000,1} \Rightarrow a' = 0$ $F_{s',a'} = F_{5000,0}$ $r = 1$ (page open, page hit) $\delta = r - Q_{s,a} + \gamma Q_{s',a'} = 1 - 0 + 0.5(0) = 1$ $\vec{\theta} = \vec{\theta} + \alpha \delta F_{s,a}$ $\vec{\theta} = (0.125, 0, 0, 0.125, 0.125, 0.125, 0, 0, 0, 0.125, 0, 0, 0)$
5000	$\vec{\theta} = (0.125, 0, 0, 0.125, 0.125, 0.125, 0, 0, 0, 0.125, 0, 0, 0)$ $s = 5000$ $a = 0$ $F_{s,a} = 1001110001000$	$s' = 5000$ $F_{s',0} = 1001110001000$ $F_{s',1} = 0110001110111$ $Q_{5000,0} = 0.625$ $Q_{5000,1} = 0$ $Q_{5000,0} \geq Q_{5000,1} \Rightarrow a' = 0$ $F_{s',a'} = F_{5000,0}$ $r = 1$ (page open, page hit) $\delta = 1 - 0.625 + 0.5(0.625) = 0.688$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot 0.688 \cdot F_{5000,0}$ $\vec{\theta} = (0.211, 0, 0, 0.211, 0.211, 0.211, 0, 0, 0, 0.211, 0, 0, 0)$
continued on next page		

Table 4.3: Control using Function Approximation

Row	Current State	Algorithm Step
100	$\vec{\theta} = (0.211, 0, 0, 0.211, 0.211, 0.211, 0, 0, 0, 0.211, 0, 0, 0)$ $s = 5000$ $a = 0$ $F_{s,a} = 1001110001000$	$s' = 100$ $F_{s',0} = 0000001100100$ $F_{s',1} = 1111110011011$ $Q_{100,0} = 0$ $Q_{100,1} = 0.625$ $Q_{100,0} < Q_{100,1} \Rightarrow a' = 1$ $F_{s',a'} = F_{100,1}$ $r = -1$ (page open, page miss) $\delta = -1 - 0.625 + 0.5(0.625) = -1.527$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot -1.527 \cdot F_{100,0}$ $\vec{\theta} = (0.021, 0, 0, 0.021, 0.021, 0.021, 0, 0, 0, 0.021, 0, 0, 0)$
1000	$\vec{\theta} = (0.021, 0, 0, 0.021, 0.021, 0.021, 0, 0, 0, 0.021, 0, 0, 0)$ $s = 100$ $a = 1$ $F_{s,a} = 1111110011011$	$s' = 1000$ $F_{s',0} = 0001111101000$ $F_{s',1} = 1110000010111$ $Q_{1000,0} = 0.084$ $Q_{1000,1} = 0.021$ $Q_{1000,0} \geq Q_{1000,1} \Rightarrow a' = 0$ $F_{s',a'} = F_{1000,0}$ $r = 1$ (page closed, page miss) $\delta = 1 - 0.105 + 0.5(0.084) = 0.937$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot 0.937 \cdot F_{1000,0}$ $\vec{\theta} = (0.138, 0.117, 0.117, 0.138, 0.138, 0.138, 0, 0, 0, 0.117, 0.138, 0, 0.117)$
continued on next page		

Table 4.3: Control using Function Approximation

Row	Current State	Algorithm Step
5000	$\vec{\theta} = (0.138, 0.117, 0.117, 0.138, 0.138, 0, 0, 0.117, 0.138, 0, 0.117, 0.117)$ $s = 1000$ $a = 0$ $F_{s,a} = 0001111101000$	$s' = 5000$ $F_{s',0} = 1001110001000$ $F_{s',1} = 0110001110111$ $Q_{5000,0} = 0.690$ $Q_{5000,1} = 0.585$ $Q_{5000,0} \geq Q_{5000,1} \Rightarrow a' = 0$ $F_{s',a'} = F_{5000,0}$ $r = -1$ (page open, page miss) $\delta = -1 - 0.552 + 0.5(0.690) = -1.212$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot -1.212 \cdot F_{1000,0}$ $\vec{\theta} = (0.138, 0.117, 0.117, -0.013, -0.013, -0.013, -0.151, -0.151, 0.117, -0.013, 0, 0.117, 0.117)$
2000	$\vec{\theta} = (0.138, 0.117, 0.117, -0.013, -0.013, -0.013, -0.151, -0.151, 0.117, -0.013, 0, 0.117, 0.117)$ $s = 5000$ $a = 0$ $F_{s,a} = 1001110001000$	$s' = 2000$ $F_{s',0} = 0011111010000$ $F_{s',1} = 1100000101111$ $Q_{2000,0} = 0.044$ $Q_{2000,1} = 0.325$ $Q_{2000,0} < Q_{2000,1} \Rightarrow a' = 1$ $F_{s',a'} = F_{2000,1}$ $r = -1$ (page open, page miss) $\delta = -1 - 0.086 + 0.5(0.325) = -0.924$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot -0.924 \cdot F_{5000,0}$ $\vec{\theta} = (0.023, 0.117, 0.117, -0.128, -0.128, -0.128, -0.151, -0.151, 0.117, -0.128, 0, 0.117, 0.117)$
continued on next page		

Table 4.3: Control using Function Approximation

Row	Current State	Algorithm Step
100	$\vec{\theta} = (0.023, 0.117, 0.117, -0.128, -0.128, -0.128, -0.151, -0.151, 0.117, -0.128, 0, 0.117, 0.117)$ $s = 2000$ $a = 1$ $F_{s,a} = 1100000101111$	$s' = 100$ $F_{s',0} = 0000001100100$ $F_{s',1} = 1111110011011$ $Q_{100,0} = -0.302$ $Q_{100,1} = 0.096$ $Q_{100,0} < Q_{100,1} \Rightarrow a' = 1$ $F_{s',a'} = F_{100,1}$ $r = 1$ (page closed, page miss) $\delta = 1 - 0.095 + 0.5(0.096) = 0.953$ $\vec{\theta} = \vec{\theta} + 0.125 \cdot 0.953 \cdot F_{2000,0}$ $\vec{\theta} = (0.142, 0.236, 0.117, -0.128, -0.128, -0.128, -0.151, 0.002, 0.117, -0.009, 0.119, 0.236, 0.236)$

4.6 Summary

For both table-based and function approximation implementations, the learning algorithm will observe page hits and misses as the experience for learning. The selection of the learning parameters can affect the learning rates and the performance of the algorithms.

The next chapter will describe how the proposed controller implementations and addressing schemes will be compared and summarizes the various processes involved in simulating these designs.

Chapter 5

Experimental Methodology

The memory controller implementations were evaluated using trace-based simulation. A VHDL test bench was used to read requests from a text file and send them to the controller via the host input interface described in Chapter 3. The test bench records requests as complete when they are signalled on the host output interface, which simulates the overall latency of memory requests to a real host. This process is illustrated in Figure 5.1.

The test bench loads requests as fast as possible without regard for the timing specified in the trace data: the trace is *accelerated*. The timing information in the trace is itself dependent on the average memory access time and using it would not give an accurate picture of how the designs implemented in this thesis actually affect the average memory latency.

It would have been preferable to perform the trace-based simulation using realistic timing, but the inter-dependence of the rate at which memory requests are issued with the average response time for memory requests makes the synthesis of such timing difficult. Therefore, the traces were read into the simulation as quickly as possible (accelerated). Accelerated traces allow the controller to achieve its maximum possible performance because the maximum amount of concurrency is available. Although using accelerated traces can hide some of the downfalls of certain controller implementations,

Figure 5.1: Simulation Method

it can provide a useful indication of the performance limits of the memory controller.

Using accelerated traces also simulates the most likely scenario in future systems where the memory subsystem is the limiting factor in system-level performance. If the higher levels of the memory hierarchy are limited by the memory performance it is likely that the processor will spend much of its time waiting for memory requests to service cache misses. In such a scenario, it is reasonable to assume that the memory system will be kept as busy as possible.

5.1 Traces

Traces were obtained from the Brigham Young University (BYU) Trace Distribution Center [1]. The traces were chosen for the same machine configuration in order for comparability:

- Pentium III: 256KB L2 cache, 32-byte line size, 256-bit cache interface [14, 15]
- 1 GB of memory
- Caches enabled (in order to reflect typical memory access patterns)
- Red Hat Linux 6.2 operating system

The machine configuration matches some of the assumptions made in the design of the memory controller in this thesis in terms of the burst size and the size of the interface.

Most of the traces selected were portions of benchmarks in the SPEC 2000 benchmark suite [2]. The traces used from SPEC 2000 are *256.bzip* (*graphic 7*), compression of graphic file; *252.eon* (*cook*), probabilistic ray-tracing using Cook algorithm; *176.gcc* (*expr*), compiling preprocessed file *expr.i*; *197.parser*, word processing; and *253.perlmbk* (*diffmail*), scripting.

One other trace was included: TPC-C, Postgres 7.1.2., one iteration: an on-line transaction processing (OLTP) database benchmark. Full details of the benchmarks used to generate the traces used can be found at other sources [1, 2, 3]. These traces represent a variety of programs having different access patterns. It was hoped that this diversity would reveal important differences between the memory controller implementations.

Traces were uncompressed and then converted to text format before being fed into the simulator. The VHDL language includes packages to read and write text file but binary files cannot be read without using vendor-specific packages. This would result in a non-portable VHDL test-bench that cannot easily moved to newer and better tools. The conversion tool was a modified version of the trace conversion tool provided by BYU (see Appendix B.1).

5.2 Simulation Tools

Initially, Synopsys tools were used for compiling and simulating the designs. The tools used were: *vhdlan* and *vhdlsim*. Other tools used for debugging included *waves* (a waveform viewer) and *vhdlldb*.

The work was eventually moved to Cadence tools. The tools used included: *ncvhdl* (VHDL compiler), *ncelab* (design elaborator), *ncsim* (design simulator), and *simvision* (an integrated simulation and visualization environment).

5.3 VHDL Instrumentation

In order to collect results from the simulation of traces that contain millions of memory references, extra code was added to the VHDL specifications to perform data collection. This allows one to get an overview of the behaviour of the implementations without having to manually inspect waveforms using the programs included with the simulation tools.

However, this extra HDL should only be used for simulation purposes: it has no place in a real hardware design. The HDL that performs this data collection is conditionally generated based on the value of a generic parameter. This parameter is specified for the top-level design unit and is propagated down to subunits throughout the design as they are instantiated. The instrumentation code is only used when this generic parameter has a certain value. Otherwise, the code is not generated or involved in the operation of the memory controller. The generic value can be set for simulation purposes and disabled for synthesis purposes.

5.4 Verification of Control Sequences

The simulations are used to collect performance metrics, but they can also serve as functional verification platforms. The test bench includes the top level entity of the DDR SDRAM memory controller as a component and it can also conditionally generate VHDL models of DDR SDRAM components arranged in DIMM-like configurations.

These pseudo-DIMMs do not provide backing store. However, they model the behaviour of the physical chips and can provide information on any timing violations which may occur during the simulation. Any violations are provided with the output as failed assertions and indicate the originating unit and the time the violation occurred.

The DDR SDRAM model used was obtained from Micron Technologies [13]. The model used in the simulations was designed for the mt46v32m8 device (x8 device with grades from DDR266 to DDR400).

Originally, the models were instantiated to exactly mimic a real DIMM: a DIMM using x8 devices required 8 models and one using x16 devices required 4 models. However, this configuration led to large simulation running times. Because all the models in the DIMM operate in lock-step (they all receive the same commands at the same times), the operation of the memory controller

can still be verified by using only one model per DIMM. This considerably reduced the simulation overhead and enabled quicker gathering of the results.

Even though this modification improved simulation performance, the memory requirements were still very large ($> 2GB$ in some cases). Therefore, the functional verification was only done for the simulations using the smallest trace (252.eon) and with a queue length of 12. For the simulations involving the learning algorithm, only one combination of (α, γ) was used for verification purposes.

5.5 Simulation Parameters

The simulations were performed using the following parameters:

- Memory speed of DDR400 ($5ns$ clock period)
- $1GB$ of memory using $256MB$ DIMMs (4 DIMMs = 16 banks)
- x8 DDR SDRAM chips
- Refresh period of $64us$ (the maximum period)

Other parameters such as the scheduling policy and page policy were varied to observe differences in the implementations. For the learning policy controllers, the parameters of the machine learning algorithms were also varied for comparison.

For DDR400 SDRAM, there are several timing parameters (see Chapter 2 for details on these parameters) with values determined by the DDR SDRAM specification [18]: the value and a short description of each parameter is provided in Table 5.1. These parameters are used in the VHDL model of the DDR SDRAM for verification purposes and are also internally used by the memory controller. They are based on the worst-case values for the various DDR400 speed grades [18].

Parameter	Time	$\times tCK$	Short Description
tCK	5ns	1	Clock period
tDQS	10ns	2	Write command data delay
tRAS	40ns	8	Active to Precharge command delay
tRC	60ns	12	Active to Active command delay (same bank)
tRRD	10ns	2	Active to Active command delay (different banks)
tMRD	10ns	2	Mode register set command period
tRFC	70ns	14	Auto Refresh command period
tRCD	20ns	4	Active to Read or Write command delay
tRP	20ns	4	Precharge command period
tWR	15ns	3	Write recovery period
tWTR	10ns	2	Internal Write to Read command delay

Table 5.1: Parameters for DDR400 SDRAM

5.6 Synthesis

The overall design framework is shown in Figure 5.2. The VHDL used to produce the simulations, being a hardware description, can also be used to synthesize hardware.

Synthesis was not performed in this project due to time constraints. However, most of the VHDL specification is synthesizable and the project can be used as a basis of a design intended for synthesis or as a functional model for comparison/verification.

5.7 Form of Results

The simulation outputs measurements taken during the simulation when the process has completed. This output summarizes the activities of the controller during the simulation based on the data collected by the components. This allows an overview of the controller's performance without having to manually examine the details of a long waveform (and requires less space).

Figure 5.2: Design Framework

The form of the output is detailed in Appendix A with an example. The complete results of the simulations are used to generate the graphs and tables found in Chapter 6.

Chapter 6

Experimental Results

Simulations were performed for each complete trace and a short summary of the activities of the simulation produced by each simulation run was used to produce the graphs and tables in this Chapter.

Results will be shown for: the average memory latency, the bus utilization, the average bank parallelism, and rates for page hits/misses/opens for the different scheduling algorithms implemented for the memory controller. Results for each trace are shown separately because the applications that are used to produce the traces are not closely related. Also, keeping the results separate allows general trends, which do not depend on the application, to be identified. For a general purpose machine, these trends are as important as the overall average result.

The distribution of the memory requests will also be shown for each trace simulation. The distributions will be shown first because they are independent of the scheduling algorithms and only depend on the addressing mode.

The results from the learning algorithms are shown for constant values of α (0.5) and γ (0.125) unless noted: the effect of these parameters will be shown separately where applicable.

6.1 Distribution of Operations Across Banks

The simulation output includes information about the memory access distribution among the 16 memory banks. This information will be presented in both graphical and tabular forms.

6.1.1 Overall Distributions

The distributions for each trace are shown for all addressing modes in Figures 6.1, 6.2, 6.3, 6.4, 6.5, and 6.6.

Except for *bzip*, the addressing modes have similar overall distributions. The *custom* addressing mode tends to have slightly more even (smooth) distribution of operations both across the DIMMs and among the banks on a single DIMM. In the case of *bzip*, the regions of highest activity are on different memory modules and these regions are also in a different order compared to the *linear* and *dynamic* modes.

A more even usage of banks within a single DIMM under the *custom* mode is most likely due to the use of lower-order bits for the bank address. This is likely to have caused more frequent bank switches. The interleaving of 64MB regions among the DIMMs in the *custom* mode also helps smoothen out the distributions although some banks still see more activity on average than others.

Fixed address mappings usually only benefit applications having certain access patterns. They can reduce the performance of other applications having behaviour that does not match a particular scheme. Using a fixed mapping scheme will not result in universal improvements in performance. The effect of the addressing mode on page hit and miss rates will be shown in Section 6.2.

Figure 6.1: Eon Operation Distributions

Figure 6.2: Bzip Operation Distributions

Figure 6.3: GCC Operation Distributions

Figure 6.4: Parser Operation Distributions

Figure 6.5: PERL Operation Distributions

Figure 6.6: TPCC Operation Distributions

6.1.2 Bank Parallelism

The distribution of operations among the banks is skewed for most traces: only a small number of the 16 banks are used by most of the requests. This implies that the bank parallelism is actually not very high for these application despite 16 banks being available and a maximum of 12 transaction being serviced concurrently.

This is supported by the measurements taken by the instrumentation code for bank parallelism, shown in Table 6.1. These numbers indicate how many banks on average were being used concurrently by the memory controller for each scheduling algorithm. The customized addressing mode results in more banks being used in parallel on average.

The size of the queue will also affect the bank parallelism. The effect of the queue length on bank parallelism was investigated for both *linear* and *custom* addressing modes (the *dynamic* mode was not explored because of its similarity to the *linear* mode). The relationships are shown in Tables 6.2 and 6.3.

The results suggest that the bank parallelism is nearing a limit when the queue's length is 12. When the queue length is 8, the maximum parallelism has already been reached in many cases. The bank parallelism does not scale linearly with the queue size, which suggests that the bank locality is relatively high and that further increases in queue size may not yield a significant increase in bank parallelism.

While the bank parallelism increase more rapidly and reaches a higher value when using the *custom* addressing mode, the overall trend is the same. The amount of concurrent bank accesses is limited for these application traces. This parallelism will not scale linearly with the queue size due to locality in the sequence of operations, even for an addressing scheme which purposefully reduces the bank locality of the requests.

Algorithm	Average Bank Parallelism								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	2	2	5	5	5	6	3	3	5
OFIFO	3	3	5	5	5	6	3	3	5
RLFIFO	3	3	5	5	5	6	3	3	5
ARLFIFO	3	3	5	5	5	6	3	3	5
CPHFR	2	2	4	4	4	5	2	3	4
OPHFR	2	2	4	4	4	5	2	3	4
RLPHFR	2	2	4	4	4	5	2	3	4
ARLPHFR	2	2	4	4	4	5	2	3	4
	perl			tpcc			parser		
CFIFO	3	3	5	3	3	6	3	3	5
OFIFO	3	4	5	3	3	6	3	3	5
RLFIFO	3	4	5	3	3	6	3	3	5
ARLFIFO	3	3	5	3	3	6	3	3	5
CPHFR	3	3	4	3	2	5	3	3	4
OPHFR	3	3	4	2	2	5	3	3	4
RLPHFR	3	3	4	2	2	5	3	3	4
ARLPHFR	3	3	4	2	2	5	3	3	4

Table 6.1: Average Bank Parallelism: Maximum Queue Size

Algorithm	Average Bank Parallelism											
	Queue Sizes: 1 4 8 12											
	eon				bzip2				gcc			
CFIFO	1	2	2	2	1	3	4	5	1	2	2	3
OFIFO	1	2	2	3	1	3	4	5	1	2	2	3
ARLFIFO	1	2	2	3	1	3	4	5	1	2	2	3
RLFIFO	1	2	2	3	1	3	4	5	1	2	2	3
CPHFR	1	2	2	2	1	2	3	4	1	2	2	2
OPHFR	1	2	2	2	1	3	4	4	1	2	2	2
ARLPHFR	1	2	2	2	1	2	3	4	1	2	2	2
RLPHFR	1	2	2	2	1	2	3	4	1	2	2	2
	perl				tpcc				parser			
CFIFO	1	2	3	3	1	2	3	3	1	2	3	3
OFIFO	1	2	3	3	1	2	3	3	1	2	3	3
ARLFIFO	1	2	2	3	1	2	3	3	1	2	3	3
RLFIFO	1	2	3	3	1	2	3	3	1	2	3	3
CPHFR	1	2	2	3	1	2	2	3	1	2	2	3
OPHFR	1	2	2	3	1	2	2	2	1	2	2	3
ARLPHFR	1	2	2	3	1	2	2	2	1	2	2	3
RLPHFR	1	2	2	3	1	2	2	2	1	2	2	3

Table 6.2: Average Bank Parallelism vs. Queue Size: Linear

Algorithm	Average Bank Parallelism Queue Sizes: 1 4 8 12											
	eon				bzip2				gcc			
CFIFO	1	3	4	5	1	3	5	6	1	3	4	5
OFIFO	1	3	4	5	1	3	5	6	1	3	4	5
ARLFIFO	1	3	4	5	1	3	5	6	1	3	4	5
RLFIFO	1	3	4	5	1	3	5	6	1	3	4	5
CPHFR	1	3	4	4	1	3	4	5	1	3	4	4
OPHFR	1	3	4	4	1	3	4	5	1	3	4	4
ARLPHFR	1	3	4	4	1	3	4	5	1	3	4	4
RLPHFR	1	3	4	4	1	3	4	5	1	3	4	4
	perl				tpcc				parser			
CFIFO	1	3	4	5	1	3	5	6	1	3	4	5
OFIFO	1	3	4	5	1	3	6	6	1	3	4	5
ARLFIFO	1	3	4	5	1	3	5	6	1	3	4	5
RLFIFO	1	3	4	5	1	3	6	6	1	3	4	5
CPHFR	1	3	4	4	1	3	5	5	1	3	4	4
OPHFR	1	3	4	4	1	3	4	5	1	3	4	4
ARLPHFR	1	3	4	4	1	3	4	5	1	3	4	4
RLPHFR	1	3	4	4	1	3	4	5	1	3	4	4

Table 6.3: Average Bank Parallelism vs. Queue Size: Custom

6.2 Page Rates

One set of important factors in the performance of a memory controller are the rates of page hits, page misses and page opens for memory operations. These page rates are defined for the first action taken on behalf of a read or write operation targeted at a particular bank. Chapter 4 contains details on page rates and their affect on DDR SDRAM performance.

The following sections will present the results from simulating the various policies and algorithms on the page rates. For hit, miss and open rates, the results for all traces will be presented first with constant parameters for the learning algorithms, followed with results showing how the learning parameters affect the learning algorithm's performance on the shortest address trace, Eon. In order to show the effects of α and γ alone, only one trace was used to clearly demonstrate how the algorithms respond to changes in only these parameters. Also, only the linear addressing mode is used for the same reason when the parameters are varied. All results are shown for the maximum queue size of 12, which allows the best possible performance for the reordering algorithms.

6.2.1 Page Hit Rates

The page hit rates for the algorithms are shown in Table 6.4 for all addressing modes. The results indicate that there are mixed results when comparing linear and dynamic addressing modes with some traces getting higher values for linear and other getting higher page hits rates with dynamic. The custom addressing mode reduces the page hit rate compared to both linear and dynamic modes: the increase in bank parallelism decreases the locality of the references. If less significant bits are used to determine the bank, it is likely that requests which would have previously gone to the same bank are now going to different banks. This will reduce the page hits that could have occurred if operations were allowed to use the same banks.

Algorithm	Page Hit Rate (%)								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OFIFO	62.7	63.1	56.1	16.1	19.3	13.8	41.3	47.2	29.0
RLFIFO	59.1	60.2	51.8	12.3	15.8	10.5	37.4	45.2	23.7
ARLFIFO	44.2	41.1	37.4	8.3	9.9	6.9	18.3	17.2	11.7
CPHFR	68.5	68.7	50.1	19.2	23.0	13.4	61.2	61.8	34.8
OPHFR	71.9	72.0	61.8	24.0	28.4	19.5	65.9	66.8	44.4
RLPHFR	70.3	70.3	58.7	21.8	25.8	16.5	64.0	64.7	41.2
ARLPHFR	70.4	70.4	58.6	21.7	25.9	16.5	64.0	64.7	40.8
	perl			tpcc			parser		
CFIFO	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OFIFO	32.2	36.7	21.9	75.9	72.2	61.0	28.8	36.6	18.8
RLFIFO	24.9	31.7	16.0	73.9	70.4	57.7	21.3	32.9	13.0
ARLFIFO	16.7	15.6	11.4	52.2	43.5	37.9	12.6	12.3	7.7
CPHFR	48.6	49.1	26.6	68.8	69.9	43.9	48.6	49.4	28.7
OPHFR	53.0	53.9	33.8	80.2	78.3	71.9	52.8	54.4	34.2
RLPHFR	51.0	51.9	30.5	78.6	76.8	69.4	50.7	52.3	30.9
ARLPHFR	51.1	51.8	30.8	77.7	75.7	66.7	50.9	52.3	31.6

Table 6.4: Page Hit Rates

There seems to be only a small difference between the linear and dynamic addressing modes. The custom addressing mode increases bank parallelism by sacrificing the number of possible page hits.

The CFIFO algorithm always has a page hit rate of zero since the page is closed after each access. OPHFR has the highest page hit rates for all the traces: if the page is always left open, the maximum possible number of page hits will occur. Reordering also further maximizes the hit rate.

The CPHFR algorithm takes advantage of page hits for requests that are queued and reorders them to maximize page hits. However, when there is only one more page hit for a bank in the queue, the last operation uses an auto-precharge read or write command to close the bank. CPHFR uses the

Parameters (α, γ)	Policy			
	RLFIFO	ARLFIFO	RLPHFR	ARLPHFR
(0.125,0.125)	55.0	41.3	69.9	70.0
(0.125,0.5)	57.0	50.1	70.1	70.1
(0.125,0.875)	57.4	53.5	70.6	70.5
(0.5,0.125)	59.1	44.2	70.3	70.4
(0.5,0.5)	59.1	38.8	70.4	70.4
(0.5,0.875)	58.1	50.9	70.5	70.9
(0.875,0.125)	60.0	52.8	70.5	70.7
(0.875,0.5)	58.7	43.4	70.5	70.7
(0.875,0.875)	60.1	51.8	70.6	71.3

Table 6.5: Page Hit Rates (α, γ): Eon

advance knowledge provided by examining the queue to perform reordering before the page is closed.

The learning algorithms are able to achieve a higher rate of page hits compared to a closed page policy when using reordering. They approximate the page hit rate of the OPHFR policy.

For FIFO, the table-based implementation of the algorithm always gets a higher page hit rate compared to function-approximation version. The learning algorithms also have more of a gap between their performance and the performance of the open page policy: without reordering, there is a larger margin for deviation and the performance of the algorithm becomes more critical.

The effect of (α, γ) on the page hit rate is shown in Table 6.5. The learning parameters have more of an effect on the resulting page hit rate for FIFO compared to PHFR: after reordering is done, there is less room for variation. These results suggest that the approximated learning algorithm behaves similarly to the table-based implementation and is a better approximation when less learning has to be performed, such as when using access reordering.

An increase in either α or γ seems to increase the number of page hits

for the learning algorithms in most cases.

6.2.2 Page Miss Rates

The detailed page miss rates are presented in Table 6.6. A closed page policy algorithm cannot experience page misses because the page is never left open after one set of accesses to the page has completed. In the FIFO case, the set is one memory operation while for PHFR it is all the operations that were reordered to take advantage of the open page. Therefore, the page miss rates for CFIFO and CPHFR are always zero.

OPHFR can never have a page miss rate higher than OFIFO: all page hits that would occur with FIFO will also occur for PHFR. PHFR is also able to create more page hits by reordering memory operations within a small window. Thus, OPHFR has a lower page miss rate than OFIFO when using the same addressing mode.

The learning algorithms are able to reduce the page miss rates for all traces when compared to open page policy (OFIFO or OPHFR). Here, the function approximation implementation performs better than the table-based implementation for OFIFO, but worse for OPHFR.

The *simulation* that occurs with reordering may result in an open page policy being used more often for the ARLPHFR and so page misses are more likely. While *simulation* also occurs for RLPHFR, this algorithm does not use any generalization: page hits for one particular rows will not affect $Q(s, a)$ for similar rows. This suggests that the rows in a bank have page hit patterns that are not very dependent on one another.

Without reordering, the results indicate that RLFIFO uses open page more often than ARLFIFO and so experiences more page hits. With generalization, page misses for one row affect the learning of other similar rows and ARLFIFO is more likely to use closed page policy for a given row when any page miss is experienced. This suggests that the rows have somewhat related miss patterns. The reasons for these results could be:

- good locality leads to a good page hit rate but most likely only for one row (good locality implies that one region is being reused)
- poor locality leads to a high page miss rate for all rows that are accessed (poor locality implies that any region that is used will be unlikely to be reused)

A high page hit rate seems more likely to affect just one row at a time. In contrast, a high page miss rate seems more likely to affect multiple rows (if multiple rows were not being accessed, the miss rate would not be high). These intuitions would explain these results and could be exploited in future memory controller designs. Perhaps less features (more generalization) could be used to learn miss patterns and a more accurate features (less generalization) could be used to learn hit patterns to focus resources where more resolution is needed.

These results show the second ability of the learning algorithm to reduce the miss rate compared to the open page policy. The ideal combination would be to maintain almost the same hit rate while having a much lower miss rate.

The custom addressing mode increases the page miss rate in all cases. Since lost page hits must either become page misses or page opens, this is expected. For an open page policy, most of these lost page hits will become page misses with only a small portion becoming page opens (see Section 6.2.3).

The effect of (α, γ) on the miss rate of the learning algorithm is shown in Table 6.7. An increase in α or γ results in an increase in the page miss rate. This seems to be the expected result: if increasing these parameters increase the page hit rate (by keeping page open more frequently) then it should also increase the likelihood that a page miss can occur. A balance between the page hit and miss rate must be reached because simply increasing the page hit rate and ignoring the increasing page miss rate may result in lowering the algorithms performance (the latency of memory requests). Ideally, a high page hit rate should be combined with a low page miss rate for the best performance.

Algorithm	Page Miss Rate (%)								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OFIFO	36.9	36.5	43.5	83.6	80.3	85.8	58.3	52.4	70.6
RLFIFO	20.6	22.3	22.2	24.5	39.0	26.5	36.3	38.2	35.9
ARLFIFO	11.7	9.9	12.5	18.8	19.3	18.1	17.9	13.3	19.0
CPHFR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OPHFR	27.8	27.8	38.0	75.8	71.4	80.3	33.8	32.9	55.3
RLPHFR	4.8	4.9	7.3	4.5	5.3	7.1	7.2	7.9	11.8
ARLPHFR	12.6	12.7	16.8	21.9	25.6	23.0	16.3	16.4	26.7
	perl			tpcc			parser		
CFIFO	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OFIFO	67.4	62.9	77.6	23.8	27.4	38.7	70.9	63.1	80.8
RLFIFO	29.2	40.0	26.3	11.5	14.4	16.4	33.6	45.3	34.7
ARLFIFO	20.5	15.5	20.7	11.9	14.9	15.2	19.0	16.8	19.2
CPHFR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OPHFR	46.8	45.8	65.9	19.6	21.4	27.9	47.0	45.3	65.6
RLPHFR	6.4	7.2	5.7	5.5	7.1	8.1	7.1	8.0	6.9
ARLPHFR	20.4	20.2	26.2	9.0	11.1	14.1	21.4	21.8	27.5

Table 6.6: Page Miss Rates

Parameters (α, γ)	Policy			
	RLFIFO	ARLFIFO	RLPHFR	ARLPHFR
(0.125,0.125)	11.1	2.8	3.3	7.1
(0.125,0.5)	14.4	6.2	4.1	7.8
(0.125,0.875)	17.5	10.4	11.2	13.1
(0.5,0.125)	20.6	11.7	4.8	12.6
(0.5,0.5)	20.4	8.8	5.7	12.1
(0.5,0.875)	18.3	13.9	9.2	16.3
(0.875,0.125)	22.8	18.9	7.0	15.4
(0.875,0.5)	20.2	15.6	7.6	15.0
(0.875,0.875)	25.1	21.7	9.9	19.9

Table 6.7: Page Miss Rates (α, γ): Eon

6.2.3 Page Open Rates

The final categories of page accesses is shown in Table 6.8. The remainder of the memory operations (those that are not page hits or misses) will occupy this category.

CFIFO always opens pages, so its page open rate is 100%. CPHFR has page hits and the rest of its memory accesses must be page opens. However, OFIFO and OPHFR have very small page open rates. The open page algorithms only have to open pages after refresh cycles occur: during the refresh cycles all the banks are precharged (closed) and an auto-refresh command is sent to all the banks. The majority of accesses are either page hits or page misses for an open page policy.

The page open rate for CPHFR is increased significantly for the custom addressing mode. This is because the reduction in page hits is directly translated into page opens (closed page policies have no page misses). There seems to be only small difference between linear and dynamic addressing modes.

The differences in page opens between closed page and the learning algorithm are either converted to page hits or page misses. The learning algorithm has a lower page open rate than a closed page policy in all cases. Some of these page opens are converted to page hits, increasing the page hit rate, and the rest of the missing page opens become page misses.

The effect of (α, γ) on the open rate of the learning algorithm is shown in Table 6.9. These results show that increasing α or γ decreases the page open rate (because it increases the page hit and miss rate). For FIFO, ARL has a higher open rate. However, ARL has a lower open rate for PHFR.

6.2.4 Affect of Generalization on Page Rates

Because the learning for (s_i, a_i) can affect the learning for (s_j, a_j) , ARL will sometimes pick a closed page policy for rows which should be kept open (seems to happen frequently for FIFO) and will sometimes pick an open page

Algorithm	Page Open Rate (%)								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	100	100	100	100	100	100	100	100	100
OFIFO	0.4	0.4	0.4	0.3	0.3	0.3	0.4	0.4	0.5
RLFIFO	20.2	17.5	26.0	63.2	45.2	63.0	26.3	16.6	40.4
ARLFIFO	44.1	49.0	50.2	72.9	70.8	75.0	63.8	69.5	69.3
CPHFR	31.5	31.3	49.9	80.8	77.0	86.6	38.8	38.2	65.2
OPHFR	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.3
RLPHFR	24.9	24.8	34.0	73.8	68.9	76.4	28.8	27.4	47.0
ARLPHFR	17.0	17.0	24.6	56.4	48.5	60.5	19.7	18.8	32.5
	perl			tpcc			parser		
CFIFO	100	100	100	100	100	100	100	100	100
OFIFO	0.4	0.4	0.5	0.3	0.4	0.4	0.4	0.4	0.4
RLFIFO	45.9	28.3	57.7	14.6	15.2	25.9	45.1	21.8	52.2
ARLFIFO	62.7	68.9	67.9	35.9	41.6	46.9	68.5	70.9	73.1
CPHFR	51.4	50.9	73.4	31.2	30.1	56.1	51.4	50.6	71.3
OPHFR	0.2	0.2	0.3	0.3	0.3	0.3	0.2	0.2	0.2
RLPHFR	42.6	40.9	63.8	15.9	16.1	22.5	42.1	39.7	62.2
ARLPHFR	28.5	28.0	43.0	13.3	13.2	19.2	27.6	25.9	40.9

Table 6.8: Page Open Rates

Parameters (α, γ)	Policy			
	RLFIFO	ARLFIFO	RLPHFR	ARLPHFR
(0.125,0.125)	33.9	55.8	26.8	22.9
(0.125,0.5)	28.6	43.7	25.8	22.2
(0.125,0.875)	25.0	36.2	18.2	16.5
(0.5,0.125)	20.2	44.1	24.9	17.0
(0.5,0.5)	20.4	52.4	23.9	17.5
(0.5,0.875)	23.5	35.3	20.3	12.8
(0.875,0.125)	17.2	28.3	22.5	13.8
(0.875,0.5)	21.1	41.0	21.9	14.3
(0.875,0.875)	14.8	26.4	19.5	8.8

Table 6.9: Page Open Rates (α, γ): Eon

policy for rows which should be closed (PHFR results in higher page hits and makes this scenario more likely). In contrast, learning for each state-action pair is independent of any other pairs for RL, so it does not suffer from this problem.

These results suggest that the rows in a given bank have access patterns which are different enough to affect the performance of the two implementations of the learning algorithm in different ways. RLPHFR does not generalize and has a higher performance (high page hits and low page misses) compared to ARLPHFR. Without reordering, RLFIFO has a higher page hit and miss rates compared to ARLFIFO which makes it harder to automatically classify RLFIFO as being better. The actual performance will be compared in the following Sections.

6.3 Average Response Time

The previous measurements indicate the differences in the detailed behaviour of the scheduling algorithms. While useful for analyzing how these algorithms resemble or differ from one another, they are secondary to the primary concern with memory controller design: performance.

Beginning with this section, the focus now shifts to the overall measurable differences between the algorithms and how they affect throughput. The first metric to be presented will be the average response time.

The instrumentation in the HDL measured the average response time as the time a request first entered the queue of waiting requests to the time which the request is actually issued to the DDR SDRAM (read or write) and completed. This measurement indicates the queueing delay and the service time.

The average response time for the addressing modes are shown in Table 6.10 for the maximum queue length of 12. The average response time for all the PHFR algorithms are tightly bunched. This suggests that page policy

Algorithm	Average Response Time (ns)								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	640	634	376	433	414	370	591	551	366
OFIFO	320	316	275	384	360	358	412	369	344
RLFIFO	332	326	279	391	367	359	427	375	347
ARLFIFO	405	416	303	403	383	364	510	481	358
CPHFR	223	224	186	213	206	204	222	205	194
OPHFR	224	225	189	222	214	215	222	205	199
RLPHFR	223	224	186	214	207	205	222	205	194
ARLPHFR	223	224	187	215	209	207	222	205	195
	perl			tpcc			parser		
CFIFO	557	527	385	480	492	303	554	514	380
OFIFO	421	392	368	249	272	260	433	388	364
RLFIFO	448	408	374	252	275	261	463	401	370
ARLFIFO	483	467	378	330	369	278	498	470	375
CPHFR	226	215	206	183	187	186	220	208	198
OPHFR	228	218	212	186	189	182	223	211	205
RLPHFR	226	216	206	187	190	181	220	209	198
ARLPHFR	227	216	208	186	189	181	221	210	200

Table 6.10: Average Response Times

has a smaller effect on response time than being able to reorder the memory requests.

All the PHFR-based algorithms have a lower response time than the FIFO-based algorithms regardless of page policy.

6.4 Average Latency

The average response time is not the best indication of memory controller performance. All of the implementations studied have the ability to overlap the processing of requests by performing operations concurrently. This allows

the individually long response times to be overlapped. This overlapping improves the throughput of the memory controller and directly affects how many requests can be serviced per unit time.

The average latency measures the average time to process all requests. This will provide a clearer indication of the controller's performance when dealing with multiple requests. The average latency is also directly comparable: it measures how long each policy takes to finish a given trace.

The average latency of the controller policies are shown in Table 6.11 for a maximum queue length of 12. While it may take comparatively longer to respond to a single request, many such requests can be concurrently processed (overlapped) and the time between successive responses is much lower. This is similar in principle to pipelining. Notice that the average response time is roughly equal to the average latency multiplied by the queue length (T_{AVL} is the average latency, T_S is the service (average response) time, and N is the queue length):

$$\begin{aligned} T_R &= T_S + (N - 1)T_{AVL} \\ \text{since } T_S &= T_{AVL} \text{ and } N = 12, \Rightarrow \\ T_R &\approx 12T_{AVL} \end{aligned}$$

The average latency is improved for all of the scheduling algorithms when using the custom addressing mode. The open page policy algorithms have lower average latencies with the custom addressing mode, even though the page hit rates for all the traces are reduced by using this addressing scheme. This suggests that the bank parallelism is a bigger factor in DRAM system performance than the page hit rates. The dynamic addressing mode has a slight advantage over the linear mode.

The PHFR algorithms all perform similarly and achieve almost 50% smaller latencies compared to FIFO policies in many cases. CPHFR has a slight advantage over OPHFR for most cases. This suggests that once the locality in the queue is used to reorder requests, it is unlikely that further

page hits will occur and the page/row should be closed. The learning algorithm implementations have latencies approximating the smaller latency when comparing open and closed page policies. In one case (TPCC, custom addressing mode) the learning algorithms perform better than either static policy.

An open page policy was best for FIFO algorithms. CFIFO saw a greater performance improvement when using the custom addressing mode (almost 50% faster than linear or dynamic modes for some cases) but it still trailed OFIFO's average latency. In the case of FIFO, the learning algorithms always performed better than closed page but were also always slower than the open page policy for the maximum queue length of 12. However, RLFIFO comes close to equalling the performance of OFIFO.

The results also show the adaptability of the learning algorithms. They can approximate either open page or closed page policy when that policy becomes the optimal choice (open page when using FIFO, closed page when using PHFR). For PHFR, the performance difference between open and closed page policies becomes very small. Although the learning algorithms can learn the optimal policy, it only makes a small difference when reordering is possible. However, the usefulness of the algorithms is not only limited to situations where reordering is possible as they work over a variety of conditions. In a general purpose machine with changing workloads, this flexibility may allow a memory controller with a dynamic page policy to maintain performance when conditions which benefit reordering become less favourable. This is because the learning algorithms are used for *prediction* while reordering uses *look-ahead* on the access stream.

For a DDR 400 SDRAM memory subsystem, the clock cycle is $5ns$. The best performing algorithms (PHFR-based) approached an average memory latency of three clock cycles for an accelerated trace. The theoretical limit is two clock cycles: assuming that all command overheads can be hidden and that the number of read-to-write delays (one cycle) and write-to-read delays

Algorithm	Average Latency (ns)								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	50.8	50.3	28.8	33.5	31.9	28.3	46.6	43.2	27.9
OFIFO	24.2	23.8	20.4	29.4	27.5	27.2	31.7	28.1	26.0
RLFIFO	25.2	24.6	20.7	30.0	28.0	27.3	32.9	28.6	26.3
ARLFIFO	31.2	32.1	22.8	31.0	29.3	27.8	39.8	37.4	27.2
CPHFR	16.0	16.1	13.0	15.2	14.6	14.5	15.9	14.4	13.5
OPHFR	16.2	16.2	13.3	15.9	15.3	15.4	15.8	14.5	13.9
RLPHFR	16.1	16.1	12.9	15.2	14.6	14.5	15.8	14.4	13.5
ARLPHFR	16.1	16.2	13.0	15.4	14.8	14.6	15.8	14.4	13.6
	perl			tpcc			parser		
CFIFO	43.8	41.4	29.6	37.5	38.5	22.7	43.5	40.1	29.0
OFIFO	32.5	30.1	28.1	18.2	20.1	19.1	33.4	29.7	27.7
RLFIFO	34.8	31.4	28.6	18.5	20.4	19.3	35.9	30.8	28.2
ARLFIFO	37.7	36.3	28.9	24.9	28.2	20.7	38.8	36.5	28.6
CPHFR	16.3	15.4	14.6	12.7	13.1	12.9	15.7	14.7	13.8
OPHFR	16.4	15.6	15.1	13.0	13.2	12.7	15.9	14.9	14.4
RLPHFR	16.3	15.4	14.6	13.0	13.3	12.6	15.7	14.7	13.8
ARLPHFR	16.3	15.5	14.8	13.0	13.2	12.6	15.7	14.8	14.0

Table 6.11: Average Latencies

(one cycle for unrelated banks to prevent bus contention, otherwise t_{WTR}) is small compared to the number of data cycles, the average latency of a memory request would be equal to the time to transfer a burst of data (for a burst length of four, two cycles are required).

6.5 Average Bus Utilization

The bus utilization should have an inverse relationship with the average latency: the lower the average latency, the higher the bus utilization should reach. The bus utilization for the various policies are shown in Table 6.12.

Even with an accelerated trace, none of the scheduling policies manage to get over 80% bus utilization. The greatest limitation is most likely the lack of sufficient bank parallelism: even a small increase led to an almost universal increase in performance when comparing custom versus linear/dynamic addressing mode. With greater bank parallelism, more latency hiding is possible and more concurrent operations can proceed simultaneously.

The learning algorithms either match or come close to matching the best utilization achieved by either open or closed page policies. The lowest utilizations are achieved by the FIFO policies. The overall trends are similar to those for the average latency.

6.6 Performance vs. Queue Size

The effect of the queue length on the performance of the memory controller is of interest. A smaller queue requires less hardware resources for buffering the data of write operations as split-transactions (data must be buffered until it can be sent to the DDR SDRAM memory). Also, the other circuitry used in the scheduler will be less complex for smaller queues and will likely be able to reach higher speeds.

The relative performances of the algorithms for *linear* and *custom* ad-

Algorithm	Average Bus Utilization								
	linear			dynamic			custom		
	eon			bzip2			gcc		
CFIFO	19.7	19.9	34.7	29.8	31.3	35.4	21.5	23.1	35.9
OFIFO	41.4	41.9	49.1	34.0	36.4	36.7	31.6	35.6	38.4
RLFIFO	39.7	40.6	48.3	33.3	35.7	36.6	30.4	34.9	38.0
ARLFIFO	32.1	31.1	43.9	32.2	34.1	36.0	25.1	26.7	36.8
CPHFR	62.4	62.1	77.0	65.7	68.4	69.2	63.1	69.3	74.2
OPHFR	61.9	61.6	75.4	62.9	65.4	65.0	63.1	69.2	71.9
RLPHFR	62.3	62.0	77.2	65.6	68.3	69.0	63.2	69.5	74.1
ARLPHFR	62.2	61.9	76.8	65.0	67.5	68.3	63.2	69.3	73.4
	perl			tpcc			parser		
CFIFO	22.8	24.2	33.8	26.7	26.0	44.0	23.0	24.9	34.5
OFIFO	30.7	33.3	35.6	54.9	49.7	52.4	29.9	33.7	36.1
RLFIFO	28.8	31.9	35.0	54.2	49.1	51.9	27.9	32.5	35.5
ARLFIFO	26.6	27.5	34.5	40.1	35.5	48.4	25.8	27.4	35.0
CPHFR	61.5	65.0	68.5	78.5	76.6	77.3	63.7	68.1	72.5
OPHFR	60.8	64.1	66.1	77.2	75.6	78.9	63.0	67.0	69.3
RLPHFR	61.4	64.9	68.5	76.7	75.3	79.4	63.8	68.0	72.2
ARLPHFR	61.3	64.7	67.8	77.1	75.7	79.5	63.5	67.7	71.3

Table 6.12: Average Bus Utilization

Algorithm	Average Latency (ns) Queue Sizes: 1 4 8 12											
	eon				bzip2				gcc			
CFIFO	52	50	50	50	38	33	33	33	48	46	46	46
OFIFO	29	24	24	24	43	29	29	29	36	31	31	31
ARLFIFO	34	31	31	31	37	30	31	31	42	39	39	39
RLFIFO	28	25	25	25	37	29	29	30	36	32	32	32
CPHFR	52	18	16	16	38	19	16	15	48	22	17	15
OPHFR	29	18	16	16	43	20	17	15	36	20	17	15
ARLPHFR	34	18	16	16	37	19	16	15	42	21	17	15
RLPHFR	28	18	16	16	37	19	16	15	36	21	17	15
	perl				tpcc				parser			
CFIFO	46	43	43	43	39	37	37	37	46	43	43	43
OFIFO	40	32	32	32	22	18	18	18	41	33	33	33
ARLFIFO	41	37	37	37	27	25	24	24	43	38	38	38
RLFIFO	39	34	34	34	21	18	18	18	41	35	35	35
CPHFR	46	21	17	16	39	14	12	12	46	22	17	15
OPHFR	40	21	17	16	22	14	13	12	41	21	17	15
ARLPHFR	41	21	21	16	27	14	13	12	43	21	17	15
RLPHFR	39	21	17	16	21	14	13	13	41	21	17	15

Table 6.13: Performance vs. Queue Size: Linear

dressing modes are shown in Table 6.13 and Table 6.14 respectively. There are not as many significant digits in these results compared to other tables in this chapter only because they have been formatted this way: all simulations produce results with the same precision. Even without as much precision, the trends in these tables are still discernible.

When the queue size is one, the performance of the PHFR and FIFO variants becomes identical: no reordering is possible in this case, so PHFR behaves identically to FIFO.

The performance of the FIFO algorithms reaches its limits when the queue size is four. Beyond this point, performance remains almost unchanged. This suggests that the maximum latency hiding or parallelism is reached very

Algorithm	Average Latency (ns)											
	Queue Sizes: 1 4 8 12											
	eon				bzip2				gcc			
CFIFO	34	28	28	28	34	28	28	28	34	27	27	27
OFIFO	29	20	20	20	43	27	27	27	38	26	26	26
ARLFIFO	28	22	22	22	35	27	27	27	35	27	27	27
RLFIFO	27	20	20	20	36	27	27	27	34	26	26	26
CPHFR	34	15	13	12	34	18	15	14	34	17	14	13
OPHFR	29	15	13	13	43	19	16	15	38	17	14	13
ARLPHFR	28	14	13	13	35	18	15	14	35	17	14	13
RLPHFR	27	14	13	12	36	18	15	14	34	17	14	13
	perl				tpcc				parser			
CFIFO	35	29	29	29	30	22	22	22	35	28	28	28
OFIFO	41	28	28	28	27	19	19	19	42	27	27	27
ARLFIFO	36	28	28	28	27	20	20	20	36	28	28	28
RLFIFO	36	28	28	28	24	19	19	19	37	28	28	28
CPHFR	35	18	15	14	30	14	13	12	35	18	14	13
OPHFR	41	19	16	15	27	14	12	12	42	18	15	14
ARLPHFR	36	18	15	14	27	14	12	12	36	18	15	14
RLPHFR	36	18	15	14	24	14	12	12	37	18	14	13

Table 6.14: Performance vs. Queue Size: Custom

quickly for a FIFO-based implementation. For a FIFO implementation, a queue size above four will result in a decreasing cost/performance ratio.

The performance of the PHFR algorithms continues to scale sub-linearly with the queue size and reaches higher levels compared to FIFO. However, most of the improvements are reached with a queue length of four. The performance characteristics seem to follow those for bank parallelism in relation to the size of the queue (see Section 6.1.2).

The performances of the policies when the queue size is one also show instances when open page or closed page is a better policy for a given trace or addressing mode. With no queue, accesses are performed one-at-a-time and no latency hiding is possible. Under these circumstances, closing the page as soon as possible using auto-precharge helps reduce the latency of a page miss over using an open page policy because this latency can no longer be hidden. The learning algorithms usually perform equivalently to the best policy under these conditions, with RL being a better approximation than ARL for most traces and for both addressing modes. In some cases with the custom mode, the learning algorithms get faster performance than either open or closed page policy.

6.7 Summary

The results of the trace-based simulations reveal some interesting information on the locality of the references in the traces used as well as how well several algorithms and scheduling policies perform when processing the requests as quickly as possible.

The next chapter will provide more analysis of these results and possible implications of these results on the design of a memory subsystem using DDR SDRAM.

Chapter 7

Discussion of Results

Chapter 6 presented a number of graphs and tables measuring various aspects of controller performance for a number of simulated implementations. In this chapter, the overall picture will be illustrated and important trends identified.

7.1 Addressing Modes

The *linear* and *dynamic* addressing modes showed very little difference in terms of the response time, average latency, or bus utilization of the various algorithms.

The *custom* addressing mode resulted in faster performance in the algorithms (in terms of the response time and average latency of a memory operation) than these addressing modes in all cases. The largest positive effect was on CFIFO. Because it reduces the page hit rate (which will not affect a closed page policy) and increases the bank parallelism, CFIFO was the mostly likely to experience a significant performance improvement using the custom addressing mode.

The results suggest that bank parallelism is an important requirement for a high performance memory system and even a small amount can significantly affect the overall performance. Sufficient bank parallelism can also overcome

lowered page hit rates.

However, Table 6.2 and Table 6.3 show that bank parallelism will not increase indefinitely with queue size and that a limit will eventually be reached if it has not already been reached with a queue size of 12.

7.2 Scheduling Policies

A PHFR-based policy always outperformed its corresponding FIFO policy. However, access reordering requires the controller to be able to queue memory operations. While the queue has a minimal effect on the performance of FIFO (which shows no improvement after the queue reaches a size of four), a larger queue allows PHFR to continue to increase the amount of reordering possible and improve the average latency of servicing the memory requests.

However, the scalability of the reordering is limited as shown in Table 6.13 and Table 6.14. This suggests that reordering is an effective means of improving the memory system performance but it may not provide a long-term path to higher performance levels.

7.3 Page Policy

For FIFO, open page seemed to always be a better choice than closed page when multiple memory operations are queued. When the queue size is greater than one, the extra latency of a page miss can be hidden by the controller (see Section 4.2). In addition, the controller can take advantage of page hits with an open page policy. When the queue size is one, closed page sometimes works better than open page. Even with some page hits, the strategy of closing pages allows page misses to be avoided and reduces the average latency. A queue length of one eliminates the ability to hide the latency of page misses when using the open page policy and shows the distinct advantage of a closed page policy in some situations.

The learning algorithm seems to be able to adapt to the best performing policy. With a queue length of one, the learning algorithms' performance usually matches the better of open or closed page. For the *linear* addressing mode, open page policy is usually faster when the queue length is one (see Table 6.13). For the *custom* addressing mode, closed page policy is usually faster when the queue length is one (see Table 6.14). In each case, the learning algorithms' performance matches the better of the static choices.

The learning algorithms are able to adapt to perform similarly to the better policy regardless of the trace or addressing mode. This adaptability shows that the algorithms can effectively predict the best policy by exploring and using feedback from the environment to update the *value* of each action for a given row address. For systems that do not support split-transactions or multiple outstanding memory operations (effectively a queue length of one), the use of a learning algorithm will allow the controller to adapt its policy to the best choice and increase the overall performance.

With larger queue lengths, closing the page after reordering seems to be the best static choice overall. This suggests that there is much locality in memory requests within a small window and that outside this window the locality is much less. Again, the learning algorithms are able to approach the best static policy through trial-and-error searching without prior knowledge.

7.4 Learning Algorithm Performance

The table-based implementation usually outperformed the function approximation implementation of the learning algorithm (see Section 6.4 and Section 6.5). Without any generalization, the table-based algorithm is able to learn more effectively which suggests that the chosen features for the function approximation implementation are not ideally correlated to the individual access patterns of each row.

The bits of the row address do have some relation to the access patterns

of each row (the approximation is somewhat effective), but this relation is not clear enough for this generalization to be as effective as storing each state-action pair individually. However, the approximation does get good performance compared to the ideal and will require much less resources in a hardware implementation (see Chapter 4). It is possible that other features could result in better performance. Also, other approximation methods exist and could be explored.

The variation of α and γ (see Section 6.2) showed that larger values of these parameters increase the page hit rate but also increase the page miss rate. Because of this trade-off, performance cannot be easily predicted based on the parameters. Generally, smaller values such as 0.125 or 0.5 seemed to work better.

7.5 Queue Length and Performance

Allowing the controller to queue operations can allow more command overlap and latency hiding. The use of a queue allows an open page policy to outperform a close page policy with in-order scheduling (which is a novel result), and allows memory accesses to be reordered to take advantage of open page states (previously demonstrated for DDR SDRAM [7]). The scalability of the queue seems to be limited however (see Section 6.6), which has not been previously explored.

When memory accesses are queued, the controller can precharge banks as early as possible when a page miss will occur. With an open page policy, this gives the same latency as using the auto-precharge command in a closed page policy. The queue gives the open page policy an advantage: not only can the policy take advantage of lower latency page hits, but the latency of page misses can also be partially hidden by issuing the precharge command as early as possible. When queueing memory requests without using access reordering, this work has found that it is always best to use an open page

policy for DDR SDRAM.

The memory accesses can also be reordered when advance knowledge of requests is known by the controller [7, 20] through the use of a queue. This results in a large performance gain compared to performing read and writes in-order. This implementation of access reordering allows a higher page hit rate at the cost of more complex mechanisms for normal operation. When the queue length is one, reordering becomes impossible and the algorithms behave identically to their FIFO counterparts.

The scalability of the queue seems to be limited. With FIFO ordering, there is no benefit to having a queue larger than 4. Because PHFR uses the queue to perform reordering of the memory accesses, it sees a larger performance gain from using a queue. The scalability of the queue is also increased when using reordering to maximize the page hits, but it still does not scale linearly and seems to have reached near its maximum with a queue length of eight.

7.6 Summary

The real-world performance depends on the controller being able to use parallelism and concurrency to hide latencies. The controller must also issue requests as soon as it is able to accept them to keep the queue as full as possible to maximize latency hiding. If FIFO is being used, queueing has little or no benefit for performance. The queue also does not scale the performance linearly with its size.

The learning algorithms are able to learn the best policy without any prior knowledge. In situations where the open page policy cannot hide the latency of page misses and a closed page policy would be preferable, the learning algorithm can adapt itself to use the closed page policy. When the open page policy is more effective, the learning algorithm can adapt to use the open page policy. The performance of the learning algorithms is usually

better than the slower policy (open or closed) and usually matches that of the best policy in this work.

None of the policies explored can reduce the relatively long response time of a single memory request. However, if these times are overlapped, the controller can achieve a high throughput. This is reflected in the bus utilization of the system: bandwidth can be used to improve throughput. However, the full throughput was not achieved (100% bus utilization) by using the various techniques discussed in previous chapters. This implies that there are still some improvements possible to the design.

One improvement which could drastically improve performance is the level of bank parallelism. This is important enough to outweigh page hit rates in most situations observed in the experimentation. The low level of parallelism available to the controller leaves much room for improvement. The ideal is to have each request in the queue use a different bank for a maximum of 12 banks being used in parallel. A maximum of 6 are used at once for the results in Chapter 6. The custom addressing mode, which increases the bank parallelism, results in higher performance in all conditions tested. This suggests that the bank parallelism is a key performance factor in a DDR SDRAM memory system.

Chapter 8

Conclusions

The results of the experimentation using a realistic description of a hardware design have revealed some insights into the performance of a DDR SDRAM memory system and the controller used in such a system.

First, some general observations about DRAM-based memory can be made. These observations, coupled with the novel contributions of this work, can indicate where future work in the field should be done to further improve the performance of the memory system.

8.1 General Observations

The key to performance at the memory controller is throughput: using concurrency and parallelism to hide latencies. Whether this is achieved by re-ordering requests or through bank parallelism, taking advantage of parallelism to overlap and hide latencies is crucial. This allows the long response time of single requests to be ameliorated by a much higher throughput. The same idea lies behind pipelining and non-blocking caches. Because improvements in memory performance often occur for bandwidth but not for latency, techniques which improve the throughput of a memory system are a natural match to emerging DRAM-based systems.

Most servers and even desktops now focus on throughput, which indicates overall system performance. Multiprocessors and multi-threading are now gaining prominence, and an approach which is oriented towards the throughput of systems will be better suited to these future designs. It may be possible to optimize the throughput of memory systems even though individual latencies for requests (the response time) remain relatively high.

The primary limiting factor of DDR SDRAM throughput is parallelism (requests should go to independent banks if possible). Without parallelism, very little latency can be overlapped since the memory requests will have to wait until previous requests finish before they can begin. A secondary limiting factor of DDR SDRAM throughput is the page hit rate. If the memory controller can reorder the processing of requests to match the non-uniform access times of the memory technology, it can significantly increase the throughput of the system by almost 50%.

Both of these limiting factors are affected by queueing. If the controller is able to deal with a set of requests instead of one at a time, parallelism in the request stream can be translated into multiple concurrent commands being issued to the memory system. A set of requests can also be reordered to take advantage of open page states in the memory system so that requests which have page hits can finish more quickly. In this design, reordering requires a host interface that allows the host to continue to issue requests while waiting on the results of previous requests.

8.2 Novel Contributions

Memory controller architecture has been previously studied. However, none of the previous works has implemented a dynamic page policy to improve the memory controller's performance for differing conditions.

This work shows that the use of a reinforcement learning algorithm can allow the controller to adapt well enough to mimic the best choice (open or

closed page) in most situations. An adaptive controller reduces the trade-offs a designer would have to make when selecting a page policy: instead of choosing one policy in order to fit most of the applications (which is vulnerable to changing or unforeseen applications), an adaptive policy will achieve nearly the same performance as either open or closed page policies and is likely to outperform at least one of them in most situations.

Despite the small difference between open and closed page policies for the reordering scheduler, the dynamic page policy would often shift its behaviour towards the better one. In applications where the difference between open and closed page is more pronounced, the learning algorithm will be even more useful, especially when the best choice cannot be predicted beforehand. This seems to be especially relevant to systems that only support one memory operation at a time and for which a queue would not be of any benefit. In such situations, the learning algorithm would be better than a queue: while look-ahead would not be possible, the algorithms perform prediction which would still be able to help the performance.

8.3 Future Work

8.3.1 Learning Algorithms

The learning algorithms used to implement the controller policies are not the most sophisticated reinforcement learning techniques. The basic algorithms can be augmented in several ways. The greedy action selection policy used to select whether to keep a page open or closed does not allow the most exploration. Other action selection such as ϵ -Greedy allow the algorithm to explore the choices more often and could perhaps improve learning performance. Other methods, such as the use of eligibility traces, could also improve learning performance [21].

Different features could affect the performance of the function approximation version of the learning algorithm. The range of Q-values can also

affect the results because they determine the maximum amount of reinforcement before "saturation" occurs at which point further positive (or negative) rewards do not significantly change the goodness (or badness) of a particular state-action pair. The learning algorithm can also be applied in different ways. For example, a learning algorithm could be used to try to predict which idle banks should be opened and what pages should be activated in advance of their use. In the case of access reordering, the algorithm could be modified to only learn when it is actually performing control instead of using simulations.

The learning rate (α) and discount rate (γ) affected the learning algorithms' performance: the conditions under which certain values should be chosen could be studied. It may even be possible to vary these values according to the feedback the algorithm is receiving from the environment. The effect of this on the complexity and hardware costs would also have to be investigated.

8.3.2 Bank Parallelism

One of the most important improvements would be in improving bank parallelism. This seems to have the largest influence on controller performance. If the queue length is 12, then at most 12 banks can be used in parallel by 12 memory operations. However, only $\frac{1}{2}$ of this is achieved in the best case. Allowing bank parallelism without reducing page hit rates would improve performance beyond what was observed in the results presented here. However, due to the flexibility of the memory organization (DIMMs of different sizes and different page sizes), this could be very challenging. Software approaches may be useful in working together with hardware to improve the parallelism in the memory system.

8.3.3 Implementation

An important aspect of the controller that has not been explored is the hardware implementation. Completing synthesis of the designs will provide more information on the costs of implementing the various algorithms, implementing machine learning, and implementing a queue for memory requests. The hardware costs and performance of the synthesized circuits will also determine the suitability of these features in a memory controller. These factors will provide more information on the trade-offs of the various approaches for a realizable design.

When the bus utilization has reached its maximum, further enhancements to a DDR SDRAM memory system will not be possible. Even with accelerated traces, none of the simulations were able to achieve more than 80% utilization. Because bandwidth measures the throughput of the memory system, future work must focus on ways to allow the memory system's bandwidth to become fully utilized. This will lower the average latency, by using latency hiding, and bring it closer to the limits of the technology. Achieving this goal is important to high performance systems where the memory access latency is the limiting factor to system performance.

Appendix A

Format of Raw Simulation Results

Results from the VHDL simulation are used to create graphs and tables in Chapter 6. The general form of these measurements is presented below using the example of a short simulation.

The output for a simulation is as follows:

```
ncsim: 05.40-s008: (c) Copyright 1995-2005 Cadence Design
Systems, Inc.
ncsim> run
Page policy : Open Page
Scheduling : Reordering
Simulation Parameters
Trace file = trace_input.small
Memory speed = ddr400
Clock period = 5 ns
Addressing mode = linear
Request inter-arrival time = 1
```

This section describes simulation parameters such as the page and schedul-

ing policy.

Statistics Inferred During Simulation:

Time in initialization = 1255 ns

Number of transactions = 5000

Number of address-only = 24

Number of reads = 2475

Total serial time for reads = 331520 ns

Average serial time for reads = 133.947474 ns

Number of writes = 2501

Total serial time for writes = 324500 ns

Average serial time for writes = 129.7481 ns

Total serial latency = 656020 ns

Average serial latency = 131.836816 ns

Total time to finish all requests = 76110 ns

Total observed idle time = 0 ns

Total observed latency = 76110 ns

Average observed latency = 15.295418 ns

Total latency hidden = 579910 ns

Average latency hidden = 116.541398 ns

This section describes overall performance measurements. Serial latency is calculated by adding the total times for requests where the time per request is calculated from when the request enters the register file until it completes. The observed latency is the total time to finish all requests, minus the idle time, divided by the total number of requests. The latency hidden is calculated from these figures. Note: the serial latency is larger than it would

be if requests were issued serially because they must compete for bank and SDRAM address bus resources.

Statistics from Components:

Data collected from execution unit:

Number of operations = 4976

Number of reads = 2475

Number of writes = 2501

Number of reopens caused by refresh cycles = 0

Number of read page hits = 1718

Number of read page misses = 747

Number of read page opens = 10

Number of write page hits = 1765

Number of write page misses = 723

Number of write page opens = 14

Hit/Miss/Open bank distribution (15 downto 0):

Read hit distribution = { 287, 357, 258, 296, 88, 132, 33, 137, 0, 0, 0, 0, 73, 8, 27, 22 }

Read miss distribution = { 133, 83, 42, 44, 123, 90, 9, 34, 0, 0, 0, 0, 130, 28, 22, 9 }

Read open distribution = { 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 2, 1, 2 }

Write hit distribution = { 332, 432, 379, 414, 6, 5, 0, 3, 0, 0, 0, 0, 5, 168, 7, 14 }

Write miss distribution = { 178, 200, 64, 110, 62, 29, 6, 11, 0, 0, 0, 0, 20, 17, 14, 12 }

Write open distribution = { 1, 2, 1, 2, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0, 1, 0 }

Average queue length = 7

Average bank parallelism = 2

This section describes the data collected by the transaction execution unit. The statistics are interpreted as follows: *page hits* refers to requests that can directly issue a read/write command, *page misses* refers to requests that must first precharge the bank because the active row is not the one they need, and *page opens* refers to the case where a request must first open a bank. The distribution of these data among the banks are also collected and displayed.

Data collected from refresh unit:

Number of refresh cycles = 1

Cycles spent refreshing = 5

Cycles spent in initialization = 251

This section describes data collected by the refresh unit: it indicates how many refresh cycles has occurred and the time spent executing the refresh cycles. this unit is also responsible for the externally controlled initialization sequence and stores the number of cycles spent in initialization.

Statistics for banks 15 downto 0:

Number of opens = { 313, 284, 108, 156, 187, 121, 17, 47, 0, 0, 0, 0, 152, 47, 38, 23 }

Number of reads = { 421, 440, 301, 340, 211, 222, 42, 172, 0, 0, 0, 0, 205, 38, 50, 33 }

Number of auto precharge reads = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }

Number of writes = { 511, 633, 444, 525, 70, 36, 8, 15, 0, 0, 0, 0, 0, 25, 185, 22, 26 }

Number of auto precharge writes = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }

Number of precharges = { 314, 286, 109, 157, 188, 122, 18, 48, 3, 3, 3, 3, 153, 48, 39, 24 }

Number of auto refreshes = { 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 }

3, 3, 3, 3 }

The data collected by the bank models is shown next. It shows the distribution of the commands issued to the banks.

Statistics for bus:

Data bus cycles = 15473

Used data bus cycles = 9950

Useable data bus cycles = 15222

The bus model shows how many bus cycles were available for use (excluding those during initialization) and how many were actually used.

```
ASSERT/FAILURE (time 77367500 PS) from process :loader  
(architecture worklib.ddd_control_unit_tb:behaviour)
```

```
Simulation ended.
```

```
Assertion at 77367500 PS + 0
```

```
./ddd_control_unit_tb.vhd:786
```

```
assert false
```

```
ncsim> exit
```

This shows the simulation raising an assertion when the trace file has been completely read and all requests have finished their data transfers.

Appendix B

Source Code

B.1 Modified Binary to ASCII Trace Conversion Tool

The original source and header files (b2a.c and byutr.h) were modified for the experiments carried out. The original versions can be found at the BYU website (see [1] for an explanation of the trace format).

The C source file was modified to allow reading trace data from the standard input and to change the format of the output to one more easily parsed by the VHDL simulation. The C source file was also modified to run on a big-endian machine (the platform used for the VHDL simulation was 64-bit SPARC, which is big endian).

The header file was modified to use data types which had explicit lengths instead of relying on the more portable but less rigorously defined standard C types. A useful macro was also added for big-endian platforms.

B.1.1 Example of Converted Trace Data

This is an example of the trace data converted into ASCII for the trace `eon_cook_0` for the first 10 requests:

```

1 00000000 DEFERREPLY 0 00 0 00000000
2 071ddde0 FETCH      8 00 0 00000007
3 0025e440 MEMREADINV 8 00 0 00000060
4 0700ff11 MEMREADINV 8 00 0 00000008
5 001dde40 FETCH      8 00 0 00000010
6 3ded5ea1 MEMWRITE 8 00 0 00000010
7 0025e460 MEMREADINV 8 00 0 000001b7
8 2702cca0 MEMWRITE 8 00 0 00000007
9 001160a0 FETCH      8 00 0 00000018
10 3dee60a0 MEMWRITE 8 00 0 00000018

```

B.1.2 Modified b2a.c

```

1 #include <stdio.h>
2 #include <strings.h>
3
4 #define BIG_ENDIAN
5
6 #include "mybyutr.h"
7
8 p2AddrTr tr;
9
10 int main(int argc, char **argv)
11 {
12     FILE *ifp;
13     unsigned long i;
14
15     if(argc != 2)
16     {
17         fprintf(stderr, "usage: %s input_byutr_file|-\n", argv[0]);
18         exit(1);

```

```

19  }
20
21  if((strcmp(argv[1], "-") == 0) && (strlen(argv[1]) == 1))
22  {
23      /* Use standard input */
24      ifp = stdin;
25  }
26  else
27  {
28      if((ifp = fopen(argv[1], "rb")) == NULL)
29      {
30          fprintf(stderr, "cannot open %s for reading\n", argv[1]);
31          exit(1);
32      }
33  }
34
35  fread(&tr, sizeof(p2AddrTr), 1, ifp);
36  i = 1;
37  while(!feof(ifp))
38  {
39      printf("%08x ", SWAP_ENDIAN(tr.addr));
40      switch(tr.reqtype)
41      {
42          case 0x00:
43              printf("FETCH\t\t");
44              break;
45          case 0x01:
46              printf("MEMREAD\t");
47              break;
48          case 0x02:

```

```
49     printf("MEMREADINV\t");
50     break;
51 case 0x03:
52     printf("MEMWRITE\t");
53     break;
54 case 0x10:
55     printf("IOREAD\t\t");
56     break;
57 case 0x11:
58     printf("IOWRITE\t");
59     break;
60 case 0x20:
61     printf("DEFERREPLY\t");
62     break;
63 case 0x21:
64     printf("INTA\t\t");
65     break;
66 case 0x22:
67     printf("CNTRLAGNTRES\t");
68     break;
69 case 0x23:
70     printf("BRTRACEREC\t");
71     break;
72 case 0x31:
73     printf("SHUTDOWN\t");
74     break;
75 case 0x32:
76     printf("FLUSH\t\t");
77     break;
78 case 0x33:
```

```

79     printf("HALT\t\t");
80     break;
81     case 0x34:
82         printf("SYNC\t\t");
83         break;
84     case 0x35:
85         printf("FLUSHACK\t");
86         break;
87     case 0x36:
88         printf("STOPCLKAK\t");
89         break;
90     case 0x37:
91         printf("SMIACK\t\t");
92         break;
93 }
94 printf("%u\t%.2x\t%u\t%08x\n", tr.size, (0x03 & tr.attr),
95        tr.proc, SWAP_ENDIAN(tr.time));
96
97 fread(&tr, sizeof(p2AddrTr), 1, ifp);
98 i++;
99 if((i % 100000) == 0)
100     fprintf(stderr, "%lu samples processed\r", i);
101     fflush(stdout);
102     fflush(stderr);
103 }
104
105 fclose(ifp);
106 fflush(stdout);
107 fflush(stderr);
108

```

```

109  return(0);
110 }
111

```

B.1.3 Modified byutr.h

```

1  #include <sys/types.h>
2
3  typedef struct BYUADDRESSTRACE
4  {
5      uint32_t addr;
6      uint8_t reqtype;
7      uint8_t size;
8      uint8_t attr;
9      uint8_t proc;
10     uint32_t time;
11 } p2AddrTr;
12
13 /* reqtype values */
14 #define FETCH      0x00 // instruction fetch
15 #define MEMREAD    0x01 // memory read
16 #define MEMREADINV 0x02 // memory read and invalidate
17 #define MEMWRITE   0x03 // memory write
18
19 #define IOREAD      0x10 // IO read
20 #define IOWRITE     0x11 // IO Write
21
22 #define DEFERREPLY  0x20 // deferred reply
23 #define INTA        0x21 // interrupt acknowledge
24 #define CNTRLAGNTRES 0x22 // central agent response
25 #define BRTRACEREC  0x23 // branch trace record

```

```

26
27 #define SHUTDOWN      0x31  // shutdown
28 #define FLUSH          0x32  // flush
29 #define HALT           0x33  // halt
30 #define SYNC           0x34  // sync
31 #define FLUSHACK       0x35  // acknowledge flush
32 #define STOPCLKACK     0x36  // acknowledge stop clock
33 #define SMIACK         0x37  // acknowledge SMI mode
34
35
36 #ifdef BIG_ENDIAN
37 /* if you are using this program on a big-endian machine
38 (something other than an Intel PC or equivalent) you will need to
39 use this function on tr.addr and tr.time.  Just replace references to
40 tr.addr with swap_endian(tr.addr) */
41
42 uint32_t swap_endian (uint32_t num)
43 {
44     return(((num << 24) & 0xff000000) | ((num << 8) & 0x00ff0000) |
45     ((num >> 8) & 0x0000ff00) | ((num >> 24) & 0x000000ff) );
46 }
47
48 /* this might be useful */
49
50 int is_big_endian()
51 {
52     uint32_t *a;
53     uint8_t p[4];
54
55     a = (uint32_t *)p;

```

```
56  *a = 0x12345678;
57  if(*p == 0x12)
58  {
59      fprintf(stderr,"Big Endian System\n");
60      return(1);
61  }
62  else
63  {
64      fprintf(stderr,"Little Endian System\n");
65      return(0);
66  }
67 }
68
69 #define SWAP_ENDIAN(x) swap_endian(x)
70
71 #else
72
73 #define SWAP_ENDIAN(x) (x)
74
75 #endif
```


Bibliography

- [1] Trace Distribution Center. <http://tds.cs.byu.edu/tds/>.
- [2] The Standard Performance Evaluation Corporation. <http://www.spec.org/>.
- [3] Transaction Processing Performance Council. Tpc benchmark c. <http://www.tpc.org/tpcc/detail.asp>.
- [4] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor dram-system performance? In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 62–71. ACM SIGARCH, IEEE Computer Society, ACM Press, 2001.
- [5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233. IEEE Computer Society, IEEE, 1999.
- [6] J. Carter et al. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79. IEEE, IEEE, January 1999.
- [7] S. Rixner et al. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138. ACM, ACM Press, 2000.

- [8] W f. Lin, S. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. *7th International Symposium on High-Performance Computer Architecture*, January 2001.
- [9] L. Fausett. *Fundamentals of Neural Networks: Architecture, Algorithms and Applications*, pages 1–37. Prentice-Hall Inc., 1994.
- [10] VHDL 200X Working Group. Vhdl-200x fast track. <http://www.eda.org/vhdl-200x/vhdl-200x-ft/packages/files.html>.
- [11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, pages 189–209, 224–240, 435–443, 454–460, 608–615. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., third edition, 2003.
- [12] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 343–354. IEEE Computer Society, IEEE, 2004.
- [13] Micron Technology Inc. <http://www.micron.com/products/dram/ddrsdram/>.
- [14] Intel Corporation. *P6 Family of Processors*, September 1998. Hardware Developer’s Manual.
- [15] Intel Corporation. *Pentium®III Processor for the PGA370 Socket at 500 MHz to 1.13 GHz*, revision 8 edition, June 2001. Datasheet.
- [16] Intel Corporation. *Intel®875P Chipset Datasheet*, February 2004. Intel®82875P Memory Controller Hub (MCH).
- [17] B. Jacob. A case for studying dram issues at the system level. *IEEE Micro*, 23(4):44–56, July–August 2003.
- [18] JEDEC Solid State Technology Association, <http://www.jedec.org/>. *Double Data Rate (DDR) SDRAM Specification*, January 2004. JESD79D.

- [19] S. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23, March 1995.
- [20] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd workshop on Memory Performance Issues*, pages 80–87. ACM, ACM Press, 2004. In conjunction with the 31st International Symposium on Computer Architecture.
- [21] S.P. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.
- [22] R.S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044, 1996.
- [23] R.S. Sutton and A.G. Barto. *Reinforcement Learning*, chapter 3,6,8. MIT Press, Cambridge, Mass., 1998.
- [24] T. Ungerer, B. Robič, and J. Silč. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.
- [25] J. van Lunteren. Towards memory centric computing: A flexible address mapping scheme. In *IEEE Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 385–390. IEEE, IEEE, 1999.
- [26] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (Micro-33)*, pages 32–41. IEEE, IEEE, 2000.
- [27] Z. Zhang, Z. Zhu, and X. Zhang. Breaking address mapping symmetry at multi-levels of memory hierarchy to reduce dram row-buffer conflicts. *The Journal of Instruction-Level Parallelism*, 3, 2002.