ABSTRACT

MODERN DRAM ARCHITECTURES

by

Brian Thomas Davis

Co-Chair: Assistant Professor Bruce Jacob

Co-Chair: Professor Trevor Mudge

Dynamic Random Access Memories (DRAM) are the dominant solid-state memory devices used for primary memories in the ubiquitous microprocessor systems of today. In recent years, processor frequencies have grown at a rate of 80% per year, while DRAM latencies have improved at a rate of 7% per year. This growing gap has been referred to as the "Memory Wall." DRAM architectures have been going through rapid changes in order to reduce the performance impact attributable to this increasing relative latency of primary memory accesses. This thesis examines a variety of modern DRAM architectures in the context of current desktop workstations. The DRAM examined include those which are available today, as well as a number of architectures which are expected to come to market in the near future.

Two simulation methodologies are used in comparing system architectures. DRAM models common to both methodologies have been developed for these experiments, and are parameterizable to allow for variation in controller policy and timing. Detailed statistics about the DRAM activity are maintained for all simulations. Experiments examining the underlying performance enhancing characteristics of each architecture are described, with attention to parameters and results.

The choice of DRAM architecture and controller policy are shown to significantly affect the execution of representative benchmarks. A 75% reduction in access latency (128 Byte L2 line) from a PC100 architecture, and a 34% reduction in execution time from a PC100 architecture result from using a cache enhanced DDR2 architecture. More significant results examine which aspects of the DRAM contribute to the increase in performance. Bus utilization, effective cache hit rate, frequency of adjacent accesses mapping into a common bank, controller policy performance, as well as access latency are examined with regard to their impact upon execution time. Not only are the highest performance DRAM determined, the factors contributing to their low latencies and execution times are also identified.

Modern DRAM Architectures

by

Brian Thomas Davis

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2001

Doctoral Committee:

Assistant Professor Bruce Jacob, Co-Chair
Professor Trevor Mudge, Co-Chair
Professor Richard Brown
Professor Emeritus Edward Davidson
Professor Emeritus Ronald Lomax

© Brian Thomas Davis 2001
All Rights Reserved

For the person who has always supported and encouraged me when I have doubted myself, Lillian Davis. For my role model in personality, behavior and demeanor, my father. For my wonderful and loving mother. And for the purest joy of my life, Autumn Marie.

Acknowledgements

I wish to express my sincerest gratitude to the people who made my research possible. This includes advisors, industry associates, compatriots, funding agencies, and friends.

Thanks to the members of my thesis committee, who have provided the prerequisite knowledge, and given me quality advice and guidance over my tenancy at the University of Michigan, in the classroom, in research group meetings, at informal hallway meetings, and time whenever asked. Thanks especially to my thesis chairs, whose guidance in my research endeavors has been essential.

A number of associates from industry have taken the time to examine and review my work, or provide traces, advice or assistance. Thanks to Mike Peters of Enhanced Memory Systems, for reviewing preliminary results, as well as serving as a continuos source of information. Thanks to Bill Gervasi of Transmeta corporation for the gathering of DRAM access traces and review of my DRAM models. Thanks to Paul Coteus of IBM for reviewing my work early in the development, and providing feedback on the realism of some simulation configurations. Thanks to the members of the JEDEC 42.3 Future DRAM Task Group who have allowed me to be involved in multiple conference calls, and the DRAM definition process.

This research was partially funded by DARPA grant DABT63-97-C-0047, so finally thanks to the American taxpayer.

Preface

In the timeframe between when I initiated this research, and the completion of this thesis much has changed in the topic area being discussed. Research has been done to investigate the characteristics of microprocessors which are now obsolete. Models have been developed for architectures which are no longer available. Memory system architectures and controllers have been designed for fabrication processes which are no longer in existence. Simulation configurations which were considered optimistic six months ago are now pessimistic. The sequence of completed tasks which were subsequently determined to be obsolete or unnecessary has encompassed many years.

In the arena of computer architecture, we are researching a moving target, and the target, as well as the rate at which the target moves are continuously changing. Perhaps one of the most valuable items which was learned during this course of research is the perspective from which to examine viable research topics and timelines. While the dominant architecture and philosophy of the day may change, the ability to plan and accommodate these changes is a consistently useful skill. “There is nothing permanent except change.“ ~ Heraclitus ~

Revision Log

- November 16, 2000

First version finallized and submitted to the University of Michigan and Rackham libraries.

- June 15, 2001

Due to errors identified in the simulations of DRDRAM, this document was revised as follows. Figures 6.14, 6.15 and 6.16 are updated with corrected data. Table 6.4 is updated with the corrected normalized execution time(s) for DRDRAM. Finally, text pertaining to this data in sub-chapter 6.11 and chapter 7 were also updated.

Table of Contents

Dedication ii

Acknowledgements iii

Preface iv

Revision Log v

List of Tables viii

List of Figures ix

Chapter 1 Introduction 11

1.1 Primary Memory 12
1.2 Impact of Primary Memory upon Performance 13
1.3 Terminology 15
1.4 Organization 16

Chapter 2 Background 18

2.1 Memory Architectures & Controller Schemes 20
2.2 System-Level Performance Techniques 21
2.3 Prior Work 25
2.4 Current Research 28

Chapter 3 DRAM Architectures & Controller Policies 30

3.1 Memory Controller / DRAM Relationship 31
3.2 Commonalities of DRAM 32
3.3 Dynamic Random Access Memories (DRAM) 35
3.4 DRAM with On-Chip Cache Capabilities 58
3.5 Controller Policies 66
3.6 Primary Memory in the Future 72

Chapter 4 Methodology 74

4.1 Introduction 74
4.2 Trace Driven 76
4.3 Execution Driven 81

Chapter 5 Simulation Models 83

5.1 Compartmentalization 83
5.2 Model Description 84
5.3 Statistic Generation 88

5.4 Validation 88
5.5 Other Simulation Code 91

Chapter 6 Results 93

6.1 Introduction 93
6.2 Processor Frequency 98
6.3 L2 Cache Interaction 99
6.4 MSHR Effects 104
6.5 Controller Policies 107
6.6 Address Remapping 110
6.7 Bus Utilization 112
6.8 Cache Hit Rates 114
6.9 Access Concurrency 115
6.10 Average Latency 117
6.11 Execution Time 119

Chapter 7 Conclusion 124

7.1 DRAM Contributions 124
7.2 Research Contributions 128
7.3 Future Work 130
7.4 Epilogue 133

Appendix 135

Bibliography 225

List of Tables

Table 3.1: Controller Policy Advantages & Disadvantages 68
Table 5.1: DRAM Model Validation using Synthetic Traces 89
Table 6.1: Overview of DRAM Characteristics 94
Table 6.2: Benchmarks & Traces for Simulation Input 95
Table 6.3: Average Latency 119
Table 6.4: Average Normalized Execution Time 122

List of Figures

Figure 1.1:	Memory Hierarchy	12
Figure 1.2:	Processor and DRAM Speed Improvements	14
Figure 2.1:	Latency vs. Bandwidth Terminology	19
Figure 2.2:	Non-Interleaved Memory Banks	22
Figure 2.3:	Interleaved Memory Banks	23
Figure 2.4:	Interleaved Bus Accesses	24
Figure 3.1:	DRAM Array - Cell Placement	33
Figure 3.2:	Timing Phases of a DRAM Access	34
Figure 3.3:	Conventional DRAM architecture	39
Figure 3.4:	EDO Page-Mode Read Interface Timing	41
Figure 3.5:	Extended Data Out (EDO) DRAM block diagram	42
Figure 3.6:	Burst EDO DRAM block diagram	43
Figure 3.7:	Synchronous DRAM block diagram	44
Figure 3.8:	SDR SDRAM Timing Diagram	45
Figure 3.9:	DDR SDRAM Read Timing Diagram	46
Figure 3.10:	256 Mbit DDR2 Architecture	48
Figure 3.11:	DDR2 Posted CAS Addition	50
Figure 3.12:	DDR2 Non-Zero WL Addition	51
Figure 3.13:	4Mbit RDRAM Block Diagram	52
Figure 3.15:	DRDRAM Interface - Simple Read Operation	53
Figure 3.14:	128 Mbit DRDRAM Architecture	54
Figure 3.16:	FCRAM Timing	56
Figure 3.17:	Mosys Multibanked DRAM Architecture Block Diagram	58
Figure 3.18:	M5M4V4169 Cache DRAM Block Diagram	61
Figure 3.19:	Asynchronous Enhanced DRAM Architecture	63
Figure 3.20:	Synchronous Enhanced DRAM Architecture	64
Figure 3.21:	Virtual Channel Architecture	65
Figure 4.1:	Memory System Architecture	75
Figure 4.2:	DRAM Bus level Trace Driven Simulation	79
Figure 4.3:	Execution Driven Simulation	82
Figure 5.1:	Conceptual Diagram of Simulation Models	85

Figure 6.1:	Processor Frequency	99
Figure 6.2:	Post L2 Cache Accesses	100
Figure 6.3:	L2 Cache Size Impact (execution driven)	101
Figure 6.4:	L2 Cache Size (OLTP traces)	103
Figure 6.5:	Impact of MSHRs Upon Access Concurrency	105
Figure 6.6:	MSHR Impact on Execution Cycles	106
Figure 6.7:	Controller Policy in Execution Driven Simulations	108
Figure 6.8:	Controller Policy in Trace Driven Simulations	109
Figure 6.9:	Address Remapping	111
Figure 6.10:	Bus Utilization	113
Figure 6.11:	Effective Cache Hit Rates	114
Figure 6.12:	Access Adjacency	116
Figure 6.13:	Average Latency	118
Figure 6.14:	Execution time for DRAM Architectures	120
Figure 6.15:	Normalized Benchmark Execution Times at 5Ghz	121
Figure 6.16:	Normalized Benchmark Execution Times at 1Ghz and 10Ghz	122

Chapter 1
Introduction

Primary memory latencies significantly impact microprocessor system performance. As processor performance continues to increase, by exploiting the parallelism in applications or multiple concurrent applications, the demands placed upon the memory system, including primary memory, magnify the impact of memory system latencies upon the microprocessor system. This thesis describes the application of primary memory technologies and mechanisms used to extract performance from commercially available systems. The examination of primary memory, and specifically DRAM has attracted significant attention since 1994. At approximately this time it became obvious that processor speeds had been increasing so rapidly that they were soon going to outpace the techniques which had up until that point been used to mitigate the affects of the increasing relative latency. Subsequent to that, many approaches were examined to allow the microprocessor to tolerate the latency of primary memory system accesses. This research follows a different approach and attempts to reduce the latency of DRAM accesses by modification of the DRAM architecture and controller scheme.

Research contribution

The DRAM technology landscape changes rapidly, and the DRAM industry is driving most of this change based upon profit, as is inherently the case in a capitalistic market. One role of academia in this market is to provide a foundation of research such that the varying industry proposals have a framework for comparison. In this thesis, I present a framework for making comparisons between DRAM implementations, based on a number of perspectives and criteria. These include the bus utilization, average latency, and execution time of a DRAM architecture. This framework is then used to make comparisons between a number of leading DRAM architectures giving special

consideration to those common features of multiple architectures which impact the DRAM selection criteria.

1.1 Primary Memory

The intent of this thesis is to examine the impact of primary memory architecture and performance upon overall system performance. In application, the primary memory is almost always composed of Dynamic Random Access Memory (DRAM). For this reason, the majority of this thesis is spent examining DRAM architectures and performance. DRAM is not the only technology applicable for use as the primary memory. Cray used Static Random Access Memory (SRAM) as the primary memory for his supercomputer systems because of the longer, and non-deterministic latencies of DRAM [Kontoth94]. A precursor to DRAM technologies, magnetic core memory has also been used for primary memory. DRAM is a technology which currently fits the requirements of primary memory better than any other available technology. In the future, other architectures may surpass DRAM as the dominant memory technology used for primary memory. This possibility for other technologies to overtake DRAM usage in primary memory systems is discussed in Section 3.6.

The primary memory is situated in the memory hierarchy, shown in Figure 1.1 at a

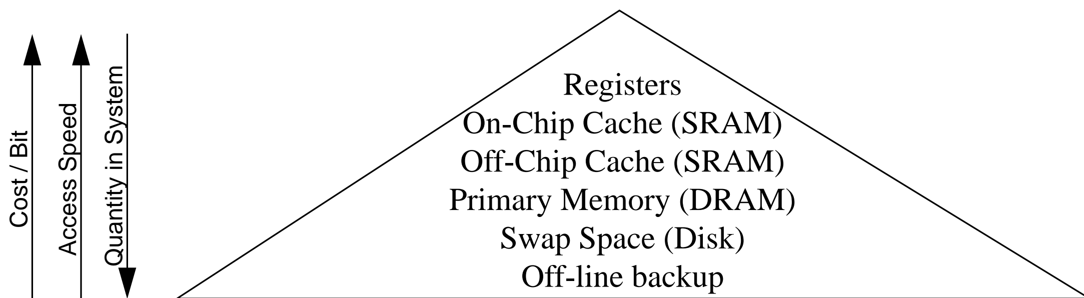


Figure 1.1: Memory Hierarchy

This pyramid shows the levels of memory present in a typical desktop workstation. Arrows on the left indicate the relationship between memory types for the shown characteristics.

level between high-speed memories, and disk. The constraints at this level in the hierarchy are what motivates the use of DRAM in this application. These constraints include the price per bit of the memory, the access latency of the memory, the physical sizes of the memory structures, the bandwidth of memory interface, among other characteristics. This

hierarchy includes all memory used by the computer system. DRAM is the lowest level of memory which is classified as volatile, meaning that the contained information is lost when power to the device is removed. Each level has its own set of constraints. At the highest level, the microprocessor register file, access speed is paramount, with cost per bit almost irrelevant. While at the lowest level, disk or backup, the cost for storage (Gigabytes or Terabytes) is the determining factor where access speeds are so slow relative to the processor speed that access speed is secondary.

The memory hierarchy has the intent of providing the image of a single flat memory space to the processor. Maintaining the facade of this single memory space is a task which requires the cooperation of the processor, caches, chipset, and operating system. Going down the memory hierarchy pyramid, the access size increases. Access size at the registers is a word, at the cache(s) a line, at the primary memory a DRAM row or page, and at the swap space an operating system page. Similarly, the cost of each bit decreases as you go down the pyramid, and the time for access increases. These characteristics are what dictate the pyramid structure, as any one technology is logically sufficient to generate a computer, but with poor performance, or excessive cost depending upon the single technology chosen.

1.2 Impact of Primary Memory upon Performance

Prior to processors achieving 12Mhz clock cycle operation, there was little interest in improving the performance of primary memory. At this clock frequency, the cycle time of processors and DRAM were approximately the same, and a memory reference could be serviced by the DRAM every processor clock cycle. This is no longer the case as we enter the era of Gigahertz frequency processors. Since 1980, processor speeds have improved at a rate of 80% annually, while DRAM speeds have improved at a rate of 7% annually [Wulf95]. To be more specific, DRAM row access times have declined at roughly 7% annually, while DRAM column access times have declined at a rate of approximately 25% annually [Przbylski96]. In order to reduce the performance impact of this rapidly increasing gap, multiple levels of caches have been added, processors have been designed to prefetch and tolerate latency, and the DRAM interface and architecture has undergone

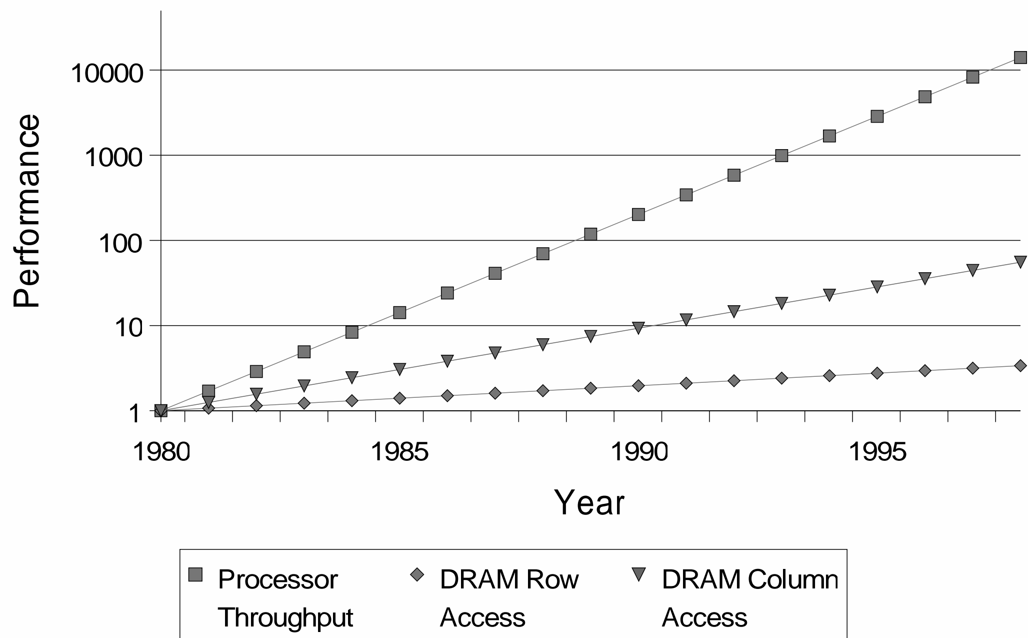


Figure 1.2: Processor and DRAM Speed Improvements

This charts places next to one another the rate of performance improvement for processors, and the rate of change for two components of DRAM latency, row access latency and column access latency, from 1980 to 1998. The gap between the processor performance improvements and DRAM access time improvements is commonly characterized as the memory gap or "memory wall". [Patterson98] [Przbylski96]

many revisions. These revisions often followed multiple concurrent paths, some evolutionary, some revolutionary. In most cases, the innovative portion is the interface or access mechanism, while the DRAM core remains essentially unchanged. The asynchronous evolution was conventional DRAM, to fast-page-mode (FPM), to extended-data-out (EDO) to burst-EDO (BEDO), each of which provided faster cycle times and thus more bandwidth than the predecessor. All modern DRAM architectures are synchronous, though Synchronous DRAM (SDRAM) typically refers to a subset of architectures. The range of synchronous architectures encompasses single-data-rate (SDR) SDRAM, double-data-rate (DDR) SDRAM, DDR2, Rambus DRAM (RDRAM) and Direct Rambus DRAM (DRDRAM), four of which are currently in use, and DDR2 which is in development, and likely to capture significant market share. Additional core enhancements, which may be applied to many, if not all, of these interface specifications include Virtual Channel (VC) caching, Enhanced Memory System (EMS) caching and Fast-Cycle (FC) core pipelining.

Common improvements in these DRAM devices include some form of caching of recent sense amplifier data, pipelining of the unique bank requests, increased frequency, multiple smaller banks and/or interface changes like DDR signalling where unique data is driven and received on both the rising and the falling edge of the clock. Even with these significant redesigns, the cycle time - as measured by end-to-end access latency- has continued to improve at a rate significantly lower than microprocessor performance. These redesigns have been successful at improving bandwidth, but latency continues to be constrained by the area impact and cost pressures on DRAM core architectures.

This is because the latency of these devices has been determined by the analog characteristics of the DRAM array. While it is possible to impact these latencies, by increasing the relative capacitance of the capacitors to the bit-lines, or similar means, doing so requires increased area which drives the DRAM out of the niche it occupies in the memory hierarchy pyramid. The approaches of many of these novel DRAM architectures attempt to increase the performance of the device, with minimal impact upon area or cost, which would place them in competition with other technologies, such as SRAM.

1.3 Terminology

DRAM architecture can be broken down into a number of component parts. The interface specification, though a conceptually distinct communication protocol, is often bound with the architecture. The specification aspect which is most strictly the architecture, and which any specification must contain, is the layout of the DRAM devices, both internally, and how they may be used in conjunction to cover the communication bus. This is typically represented as a block diagram of functional units comprising the DRAM memory system. Lastly, the controller policies and protocols have such an impact upon DRAM performance that they are frequently characterized as part of the architectural specification. These three components: 1) interface, 2) architecture, and 3) controller policies referred to in some set the DRAM architecture of a device, and all have significant impact upon the performance as well as dependence upon one another.

An example of a communication protocol as part of a DRAM architecture is the Direct Rambus Channel specification within the Direct Rambus Architecture specification. There is no reason that this interface protocol could not be used with a device other than a Direct Rambus device.

The interface communication protocol need not be tied to the architecture. An example of this are Synchronous DRAM (SDRAM) devices designed such that the same IC can be used with either a single data rate (SDR) or double data rate (DDR) interface depending upon the bond-out decisions. Other architectures such as the fast cycle (FCDRAM) also are available in multiple interface implementations. It may well be the case that future DRAM device architectures will make less modification to the interface specification or policy decisions in an effort to increase re-use of the core layout.

1.4 Organization

The organization of this dissertation is as follows. Chapter 2 reviews previous work and background information essential for understanding later chapters. Chapter 3 describes the variety of DRAM architectures proposed and available, some of which will be examined and some of which are described only for reference. Chapter 4 covers the methodologies used for the experiments described in this dissertation. Chapter 5 covers the simulation models for the DRAM architectures in more depth. Chapter 6 describes the results of experiments for a variety of specific DRAM characteristics. The performance differences between the simulated DRAM architectures can result in a 75% reduction in access latency (128 Byte line) from a PC100 architecture, and a 34% reduction in execution time from a PC100 architecture. More significant for the designers of future DRAM, the characteristics allowing these performance differences are examined in depth. Chapter 7 concludes with a summary of the results inferred from the experiments, as well as a discussion of the unique contributions of this thesis. These contributions fall into three primary categories, the comparison of DRAM architectures, the advancements in methodology for DRAM architecture studies, and the examination of the characteristics of DRAM design which determine performance. The intent is that all chapters be read in sequential order, but Chapters 6 and 7 truly contain the bulk of the information not

available elsewhere. If examining this thesis purely for results, a reader can read the Chapters in 1, 2, 6, 7 order and call upon Chapters 3-5 where required for reference.

Chapter 2
Background

A wide variety of novel format DRAM devices are becoming available on today's market. These DRAM, some of which are backwards compatible with existing DRAM organizations, encompass a large number of features intended to reduce latency or increase bandwidth of the DRAM device. It is the aim of this thesis to identify which enhancements to DRAM are valuable improvements to be integrated into memory controller chip sets and which can justifiably be allowed to disappear as many other specifications have, without a great loss to the computer system design community.

What are enhanced DRAM? This is a question which has a different answer with each new product introduction by the DRAM manufacturers. For example consider the evolution from conventional DRAM to Fast Page Mode (FPM) DRAM to Extended Data Output (EDO) DRAM to Burst EDO (BEDO) DRAM which was the last high volume asynchronous DRAM specification. Following this evolution of asynchronous DRAM, we are currently in the midst of an evolution of SDRAM. Synchronous DRAM first became available at 50Mhz, then 66Mhz, currently the most common DRAM devices are PC100 (100Mhz) SDRAM devices. Competing with this are DRDRAM devices (at 300 and 400 Mhz). The frequencies of SDRAM will continue to increase with almost certainly devices produce at all of the following architectural specifications: PC133, PC2100(DDR266), PC2400(DDR300), DDR2, and cache enhanced SDRAM devices. What was yesterday's enhanced DRAM becomes tomorrow's standard. Complete descriptions of each of these DRAM technologies will be given in Section 3.3. Many of the new features available in enhanced DRAMs are not being utilized in current applications because the interface chip sets are treating the enhanced DRAMs as conventional DRAMs. The cost advantages of mass production mean that many of the new enhanced DRAM features may never be

widely utilized because the interface chip sets (i.e., memory controllers) produced will not make use of the features of each specific DRAM architecture, in order to be compatible with as many DRAM manufacturers and organizations as possible.

The two main parameters that differentiate DRAM devices are the latency and the bandwidth observed when these devices are placed in equivalent systems running the same benchmarks. The two primary parameters of concern therefor are: 1) the number of clock cycles that elapse between a processor's request for data and the arrival of the first line of the requested data at the processor input pins (latency); and 2) the rate at which the subsequent data lines are returned after the first line arrives (bandwidth). Figure 2.1 shows

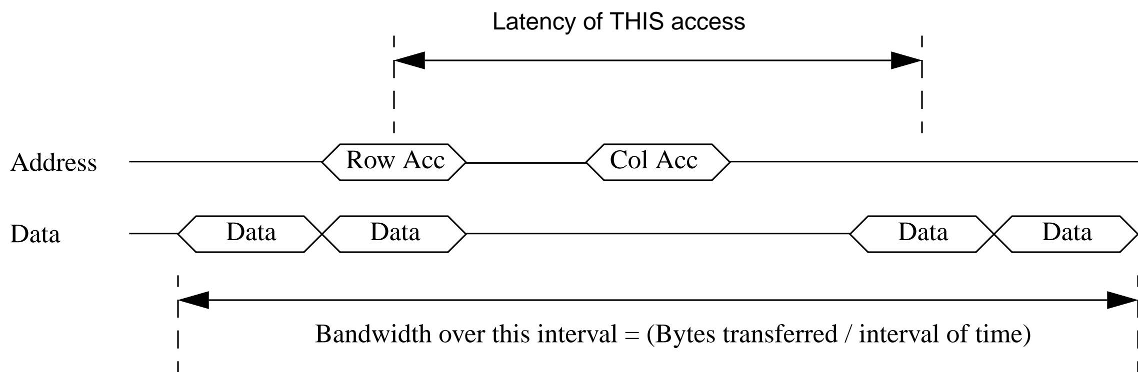


Figure 2.1: Latency vs. Bandwidth Terminology

Latency is measured in time, typically nanoSeconds,
Bandwidth is measured in data per time, typically MBytes/Sec. or GBytes/Sec.
The first two Data packets shown in the above figure are assumed to be from prior request

a rather generic example to illustrate the terminology. For the transaction shown in Figure 2.1, the bank to be accessed is already precharged (as would be the case if the controller was using a close-page-autoprecharge policy), and the latency of the access is the time from the access to the response of the data. Latency can be specified as latency of a specific access, or as the average latency of all accesses in a given workload. If we assume that the above is a 2-2-2 PC100 SDRAM, with each interval being 10nS, then the latency of the device in Figure 2.1 access is 40nS. Bandwidth is typically described in one of two ways, as potential (upper bound) bandwidth or as utilized (effective) bandwidth. In exploring utilized bandwidth, we will calculate the bandwidth of the interval below the bus diagram. The utilized bandwidth of the above interface — assuming the interval

shown on the bottom is 7 time units and the center idle section of the data bus is 3 time units — is going to be 4/7 of the potential bandwidth. If we again assume each interval to be a cycle in a PC100 DRAM, this would generate a utilized bandwidth, over this interval, of (4 * 8 bytes) / (7 * 10 nS) or 457 MBytes/Sec. Other characteristic parameters affecting performance for specific applications are: the degree that memory accesses can be pipelined; the frequency with which the dynamic cells need to be refreshed; and can one (or more) bank be refreshed while another bank is being accessed. The extent to which these factors affect the system performance or the performance of the memory system depends upon the loading of the memory system.

One of the problems encountered when studying enhanced DRAM configurations is that different vendors give similar configurations different names. A prominent example of this is that Cache DRAM offered by Mitsubishi is often referred to as an “enhanced DRAM” but is completely different from the Enhanced DRAM being offered by Ramtron. To avoid ambiguity, Chapter 3 describes each of the types of DRAM which we will study, the name by which we will refer to them, and their configuration or architectural specifics.

2.1 Memory Architectures & Controller Schemes

The cores of DRAM devices, typically referred to as the array, are very similar, regardless of the device architecture. The differences in array implementation between the devices that we examine are most commonly in the array size, the number of arrays per device, the peripheral circuits extracting data from the array; or the interface between the device and the memory controller. In conventional desktop workstation system architectures, the controller is located on the north bridge of the motherboard chipset. Only one architecture significantly changes the layout of the DRAM array, and this is the Fast Cycle (FCDRAM) which pipelines the array itself by placing latches into bit-lines and the wordlines[Fujitsu99]. This pipelining of the array into multiple segments has the advantageous consequence of reducing the capacitance of the bit lines, enabling faster sense-amp detection of the bit values on a read. Other core enhancements are focussed upon changing the number of arrays or the size of the arrays, but the array itself remains relatively unchanged.

The smaller each array in the device is, the smaller the capacitance associated with the word line and the bit lines. The smaller the capacitance associated with each of these signal paths, the shorter the access time of the array. This is the approach taken in many embedded applications, such as the Mosys 1T-SRAM approach [Mosys00], or the IBM embedded DRAM architecture [IBM00b].

The unique DRAM architectures are discussed in Chapter 3, however there are some system-level techniques which are used uniformly across all architectures. A few of these techniques which require a redesign at the memory system level are presented in Section 2.2. These techniques preceded the development of modern DRAM architectures and were used to build high performance memory systems out of commodity lower performance asynchronous DRAM devices, when that was the only commercially available choice.

2.2 System-Level Performance Techniques

Devising new memory architectures is not the only method which can be used for improving memory response. Other techniques exist, both old and new, which attempt to increase memory performance. It might be worth noting at this point that memory performance can be characterized by a number of parameters, and therefor one memory architecture can not always be said to be absolutely superior to another except with respect to a single parameter or application. As we have seen, two of the most critical parameters by which a memory can be characterized are latency and bandwidth. As has been mentioned, when modifications are made to a memory systems in order to improve performance with respect to an application, care must be taken to insure that the modification does not degrade performance on other applications. The techniques outlined below have the intent of improving performance, but it is possible that they could hinder performance in certain unusual cases.

One of the motivations for this thesis, is an investigation into how the standard interface, currently the interface of the SIMM, should be modified for higher memory performance. Each of these techniques attempts to change the system level DRAM bus in

such a way that increased performance is feasible. Each technique can be used individually, or they may be used in conjunction.

2.2.1 Interleaved Address Space

Interleaving memory is a relatively old technique in computer architecture for optimizing the use of the bus to memory, and decreasing access latencies. The essential concept behind this technique is that a large number of banks of memory, operating independently, can share the same transaction buses.

A multibanked memory structure without interleaving is shown in Figure 2.2.

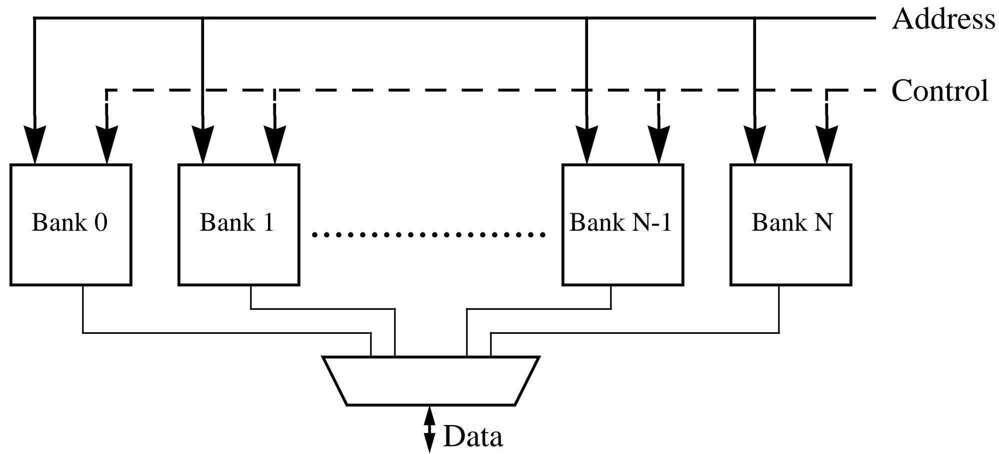


Figure 2.2: Non-Interleaved Memory Banks

Non-interleaved memory banks share all interface signals, address, data, and control. If devices in this configuration have a dedicated bus, then the shared system has a bus which is dedicated.

Interleaving separates control between the many banks which compose the memory [Ng92]. This allows faster average access times because the bank being accessed can perform the requested operation, while other banks remain in the precharged state, awaiting requests. In some cases, where the bus allows split-response transactions (such as modern SDRAM, and Rambus architectures), an interleaved memory system can allow multiple outstanding and concurrent transactions at the same time, to unique banks, even using asynchronous memory devices. This combination acts in much the same way that multiple functional units do in a superscalar microprocessor such that each bank, or functional unit, can perform an independent task during each cycle. Figure 2.3 shows how the interleaving changes the block diagram of the memory banks. This figure assumes a

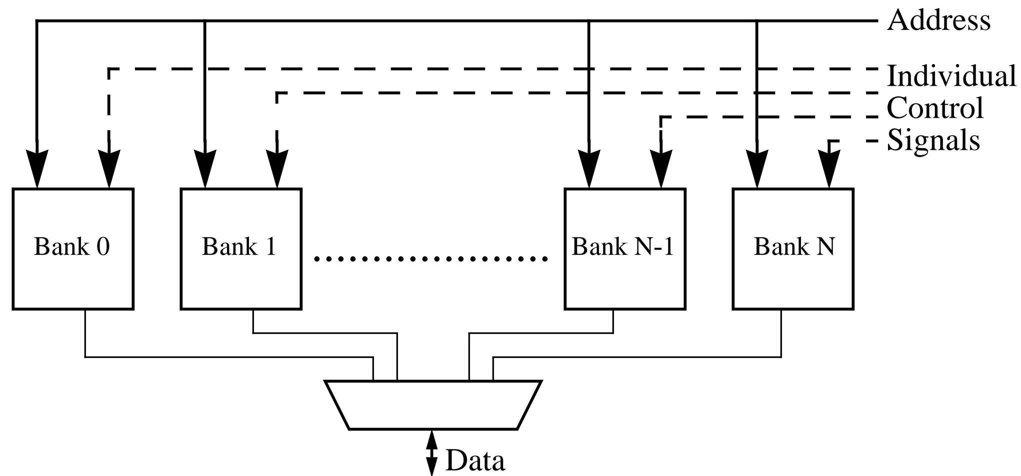


Figure 2.3: Interleaved Memory Banks

Interleaved memory banks make use of individual control signals to share the address and data bus. In this way a device which is designed with a dedicated interface can share interface signals

single address and data buses for all banks. More complex interleaved implementations could utilize multiple address and data buses for higher theoretical bandwidth. Interleaving with a split-response bus has many attractive effects, primarily the address and data buses can achieves a higher efficiency, which is one of the primary claims of the manufacturers of Section 3.3.11 — Multibanked DRAM (MDRAM), which uses a highly interleaved scheme [Hennesy90][Rau79].

Care must be taken to distinguish between memories which contain multiple banks and memories which are interleaved. If a memory is interleaved it implies that the memory contains multiple banks, but the converse is not true. It is possible for a memory with multiple banks to share control among all banks, and in this case the multiple bank memory is non-interleaved.

Interleaving is an idea which should certainly be employed in whatever DRAM technology proves to be the successor for today's conventional DRAM. Interleaving allows full usage of the multiple banks of memory. However, because interleaving requires significantly more control signals on the system level bus, this approach is becoming less common in all but the most expensive of system implementations.

The interleaved or partitioned address space, where all banks are identical and function independently, has a number of interesting consequences. At runtime or boot-

time, if a bank is determined to be defective, the memory system can disable this bank in the memory system. This requires a controller with significant intelligence, but is enabled by a controller which contains the address of each individual bank within a volatile register which is part of the bank address translation circuitry. This allows for a system to continue functioning even if a hard error occurs in one of the DRAM devices. Other possible applications of an interleaved address space involve partitioning the address space between a multiple processor system, or between applications. Either of these techniques would require significant co-design with the operating system. An interleaved or partitioned address space is a more flexible, higher performance, higher cost design. The modern synchronous architectures are able to capture much of the performance with significantly less cost.

2.2.2 Interleaved Bus Accesses

In addition to interleaving the address space, accesses can be interleaved in time. In Figure 2.3 the data buses prior to the MUX are non-interleaved, however the data bus comprising the output of the MUXes is interleaved. Figure 2.4 shows a series of accesses

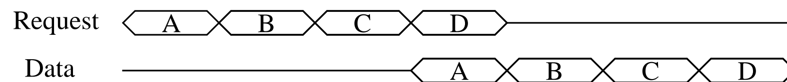


Figure 2.4: Interleaved Bus Accesses

Shown is a communication bus which allows a split request/response protocol. Since three accesses are in-flight this bus has an interleaving factor of 3.

which are interleaved in time. This style of bus is also referred to as a split request/respond bus as opposed to a dedicated bus. This approach does not inherently change the latency of the requests, however it does require that the request and responses be latched at the devices to which the requests are directed. Interleaved bus accesses are becoming more common within modern DRAM, increasing the bandwidth across the DRAM bus without placing significant additional costs upon the system level implementation. Beyond more complex interface logic at the devices (as is the case for SDRAM and RDRAM) the system level costs are no more than those for a dedicated bus.

Interleaving in older non-synchronous DRAM systems allows for a limited amount of concurrency, supported by a memory controller that has independent address buses to each bank of DRAM devices, and that controls a MUX to share the data bus between all banks. This configuration is costly. It is the intent of synchronous memory systems to enable a similar level of concurrency through the shared synchronous interface. In order to achieve the highest amount of concurrency in a memory system, regardless of technique, accesses adjacent in time should be directed to unique banks. In interleaved memory systems, this involves locating different portions of the memory space in different banks.

It is possible to have interleaved bus accesses without interleaved address space. This is essentially what the transition from asynchronous to synchronous DRAM interface(s) has enabled. All synchronous DRAM have some level of support for interleaved bus accesses, or access concurrency. This level of support varies by architecture and will be described more in Chapter 3.

2.3 Prior Work

Recent interest in the primary memory system was motivated by observations that the memory system was becoming a major hindrance to system performance [Wulf95][Sites96]. This processor-centric view meant that research has not concentrated upon how to redesign the DRAM architecture for improved performance, but rather how to make a processor that tolerates increased primary memory access latencies.

There has been significant research on designing the CPU to tolerate long memory latencies. Many techniques have been proposed, and have to varying degrees been successful. Both instruction and data prefetching, driven by either hardware components or explicit software prefetch instructions have increased the lead time of the memory access, reducing the associated penalty when the load misses in upper level cache(s) [Callahan91] [Fu91] [Klaiber91] [Mowry92] [Dundas98] [Farkas97] [Roth99]. Out of order execution allows execution of instruction B to bypass or precede execution of instruction A, though the ordering of these two instructions is A->B. This allows functional units to remain busy and work to be completed even when a preceding instruction is waiting on an outstanding load [Tomasulo67] [Hwu87] [Sohi87]. Making

use of the parallelism available in the memory access stream is a common trait of many of the techniques for tolerating memory latencies, as well as the new DRAM architectures. In order to enable parallelism in the memory stream, a non-blocking memory system including lock-up free caches is necessary [Kroft81] [Sohi91]. CPUs have attempted to avoid the latency associated with data loads by hoisting loads out of a conditional code segment to earlier in the execution stream, and executing them speculatively based upon the condition of the code segment. This requires the ability to squash incorrect speculation, and can provide performance advantages, especially in applications which have low levels of parallelism and long dependence chains [Rogers92] [Reinman98]. Prefetching of memory values reduces the impact of memory access latency upon the superscalar microprocessor. One method of prefetching requires predicting data addresses before the effective address can be resolved, due to data dependencies. This technique has been shown to be effective in reducing the impact of long memory access latencies [Cheng98] [Roth98] [Beckerman99]. If the prediction of an address stream is possible, and the microprocessor is capable of recovering from a misprediction, short cutting the memory access of the prediction results in a technique called data value prediction. This technique, like all prediction techniques, requires recovery methods for misprediction, but has again been shown to significantly reduce the effect of long memory access latencies upon the superscalar microprocessor [Lipasti96] [Sazeides97] [Wang97] [Calder99]. A technique closer to the DRAM level for reducing the average observed latency is memory-request reordering at the memory-controller level. This technique has proven to be successful in reducing latency, even within a fixed DRAM architecture [McKee95] [McKee96] [Swanson98] [Carter99] [Tang99]. Finally, we are beginning to see significant changes to DRAM architecture in an attempt to address the same issue from the other side of the "Wall". Many of the novel DRAM architectures proposed by industry are examined in depth in Chapter 3. Examples include the novel interface and bank architecture of the Direct Rambus architecture — Section 3.3.9 [Rambus99], the cache enhanced architectures: Enhanced Memory Systems (EMS) — Section 3.4.2 [EMS00] and Virtual Channel (VC) — Section 3.4.3 [NEC99], and the partitioned DRAM array technology Fast-Cycle DRAM (FCDRAM) — Section 3.3.10 [Fujitsu99] proposed by Fujitsu. These are all evidence that now, in addition to the microprocessor being designed to tolerate

latency, DRAM manufacturers are seeking design changes to increase bandwidth, decrease latency, or in some way reduce the impact of memory accesses upon microprocessor performance.

Note that many of these mechanisms exploit concurrency to improve performance - in the long run, parallelism in the memory system is likely to achieve the same sort of performance gains as parallelism in instruction execution (the benefits of which have been quite clearly demonstrated over the years). Older, non-synchronous DRAM systems only service a single memory request at a time. Because older processors stall on every cache miss, this arrangement poses no significant problems. However, modern processor designs have added support for concurrency in the form of split transaction buses and lockup-free caches [Kroft81] [Sohi91]. As a result, modern systems are capable of performing multiple operations in parallel as well as performing useful work during the servicing of a cache miss. To fully exploit this feature, modern DRAM memory controllers must support some degree of concurrency: that is, they must be able to handle multiple, overlapping requests, otherwise the processor's overlapping memory requests will stall, rendering the non-blocking cache useless. Today's most advanced DRAMs support some degree of concurrency directly—this is usually on the order of two to three concurrent requests to independent banks which can be pipelined. What is not known is how much performance this arrangement buys, and what degree of concurrency the DRAM system must support to obtain reasonable results.

Most work in the area of DRAM has been done by industrial groups who are involved in the design, manufacture or application of DRAM devices [EMS00] [Farkas97] [Fujitsu99] [Mosys94] [Rambus99] [Sites96] [Woo00]. Industry, unfortunately, is often hesitant to publish results, and thus we may never see much of the research which has been done in this area. For example, both VIA and IBM use mechanisms for address remapping or hashing (see Section 3.5.4) in their memory controllers, but do not disclose the specifics of these mechanisms, nor the comparative performance of the techniques if such results exists. The research of industry, as valuable as it may be, is often buried as proprietary knowledge despite its value to the community as a whole.

Exceptions to this silence from industrial sources are those organizations which are trying to effect a change upon the industry, and thus must justify this change. For this

reason, Rambus, Enhanced Memory Systems, Mosys, and Fujitsu have all published technical, or white papers about their own technologies [Rambus93][EMS00][Mosys94] [Fujitsu00]. Some of these include comparative results, but it is not a a surprise that when this is the case, these results typically favor the architecture of the publishing organization. Additionally, standards organizations, such as JEDEC or IEEE have published specifications for their respective standards, some of which encompass the DRAM domain, such as [JEDEC00] [IEEE1596.4]. Unfortunately, these standards, while voluminous and highly technical, rarely contain the data used to guide the specification process.

2.4 Current Research

Some work has been done on DRAM in academe, but this has only been recently, predominantly since attention was called to the growing gap between processor and DRAM speeds or "Memory Wall" [Wulf95]. Prior to the public appeal to examine the problem of memory latency, a significant amount of research was being done upon microprocessor architectures. Due to the inertia involved in academic research projects, the first examinations of methods for coping with memory latency were microarchitectural techniques for tolerating latency. Much of this work is discussed in Section 2.3. Following this, some techniques for decreasing the DRAM latency and addressing this performance impact in the memory system have been performed.

Some of the work presented in this thesis has been presented in other forums over the past years [Cuppu99] [Davis00a] [Davis00b]. This research on DRAM architectures has been initiated, and is continuing to be worked upon by Vinodh Cuppu, Brian Davis, Bruce Jacob and Trevor Mudge. This joint work between researchers at multiple campuses will continue. There is no foreseeable end to the problems being generated by a unified memory system attached to one or more high-bandwidth processors. This will provide a breadth of research problems, as described in Section 7.3, to occupy this research group for years to come.

The Impulse project at the University of Utah [Carter99] is one of the leading examples of work being done in the area of primary memory system. The research being

done in this group is focussed upon the memory controller architecture, enhancements and policies rather than DRAM architectures. This allows the results to be portable across the rapidly changing DRAM landscape.

Beyond this, industry continues to innovate and propose new DRAM technologies at a rapid pace [Cataldo00][Dipert00]. Balloting continues on potential changes to the next generation of JEDEC DDR specifications [Peters00]. There is an entire community of industry and academic personnel committed to generating the best solid-state memories to give their future devices the highest possible memory system performance. There is no reason to believe that the newfound exploration of memory technology architectures will decrease any time in the near future.

Chapter 3
DRAM Architectures & Controller Policies

The acronym RAM refers to a Random Access Memory. RAM that uses a single transistor-capacitor pair for each binary value (bit) stored is referred to as a Dynamic Random Access Memory or DRAM. This circuit is dynamic because the capacitor must be periodically refreshed for the information to be maintained. The use of the RAM acronym in describing DRAM may be misleading, as we will see later, but it was coined to distinguish it from early drum and tape memories. The inherent refresh requirement of DRAM is in contrast to Static RAM (SRAM) which will retain a value so long as the power rails of the circuit remain active. It is also in contrast to the class of non-volatile memories, drum, tape, eeprom, flash, etc., which maintain values in the absence of a power supply. An SRAM however requires more transistors for each bit, meaning that fewer SRAM data bits can be stored per unit area than DRAM bits. For this reason, SRAM is typically used only where speed is a premium, and DRAM is used for the larger portion of data storage.

DRAM, as the most cost effective solid-state storage device, is used in a wide variety of applications, from embedded appliances to the largest of supercomputers. Today "computers" span the spectrum of servers, workstations, desktops, notebooks, internet appliance, and game boxes. Each has unique set of design requirements for their respective memory systems. The variance in design requirements becomes even larger when graphics cards, routers and switches, mobile electronics, games and all other electronic devices using DRAM are included. In 1999 only 67% of DRAM produced was used by computer systems [Woo00]. DRAM must be targeted at a wide variance of application, not exclusively at computer systems. Because DRAM is a commodity part, and the costs are largely based upon volumes, it is attractive to have a single DRAM

solution which applies equally well to all applications. Clearly, whenever a single tool or device is crafted for multiple uses, that tool may not perform as well as a domain specific solution. We intend to examine the application space of DRAM in conjunction with the current and near-term DRAM architectures, focusing on those elements which favor each architecture with respect to specific applications.

3.1 Memory Controller / DRAM Relationship

The term memory controller is somewhat ambiguous, as there are both hardware and software functional blocks which serve as controller for some level of the memory hierarchy. The Memory Management reference [Xanalys00] states that memory management can be divided into three areas: (1) Memory Management Hardware, (2) Operating System memory management and (3) Application memory management. When we refer to the memory management implemented by the memory controller, a hardware component, we are referring to the first area. The second and third areas are elements of software systems for managing memory. In most modern desktop machines, the memory controller is a finite state machine located in a companion device to the microprocessor. This is largely because the number of pins required for the interfaces prohibits the controller functional block from being placed on the microprocessor, though this prohibition is being relaxed by modern technologies such as the Direct Rambus Channel and packages with larger pin-counts.

The memory controller — a functional block, commonly located in the north-bridge of a motherboard chipset - has many roles. The controller must guarantee that each row within each DRAM bank is refreshed within each refresh period. For example, the IBM0364164C PC100 SDRAM must receive 4096 refresh cycles/64ms period. This corresponds to the number of rows in a bank (the same row in all banks in the device are refreshed simultaneously) divided by a time period determined by the capacitance and leakage of each of the capacitors comprising the DRAM array [IBM98]. Fortunately these devices are designed such that a single refresh command can perform the required refresh for all banks concurrently. If this were not the case, the number of refresh cycles would increase linearly with the amount of memory in the system and would begin to have a

serious impact upon system performance. A second role of the controller is to be a bridge between components performing memory requests and the DRAM devices. If there are many sources of DRAM requests (multiple processors in an SMP, AGP graphics, DMA I/O devices, etc.) the memory controller functions as an arbiter between these devices, allocating a limited resource, the DRAM bus, to the device(s) based upon priority or policy.

In playing each of these roles, the controller designer commonly has a variety of methods which can be chosen — typically referred to as policies. It is difficult to discuss DRAM architectures in isolation from controller policies. Two dominant controller policies are the open-page and close-page-autoprecharge controller policies. These two, as well as many additional policies, are discussed in Section 6.5. Different DRAM architectures are best suited to different applications [Davis00b]. The choice of policies is also dependent upon architecture and application. Both DRAM architectures and memory controller policies are examined in this chapter. These two aspects of the memory system are highly dependent upon one another, in that the choice of controller policy is dependent upon the choice of DRAM architecture. For example, the use of an open-page policy with any of the cached architectures provides no additional performance over the caching structures, and eliminates the ability to hide the precharge of a page-miss access.

3.2 Commonalities of DRAM

Dynamic Random Access Memories (DRAM) are based upon circuits which have a single capacitor-transistor pair for each bit of information stored in the device. With devices currently available with 256 Mbits, and fabricated in laboratories with 1 Gbits, the capacity and scale of these modern devices is enormous. As noted earlier, these circuits are characterized as dynamic because if they are not periodically refreshed, the data in the capacitors will be lost due to transistor leakage.

All DRAM have an array of these single bit cells, which are addressed via the row (or word) lines and from which the data is accessed via the bit lines. This array is common among all DRAM devices, but the number of arrays which are located in each device is a design/performance trade-off which varies from architecture to architecture. Smaller

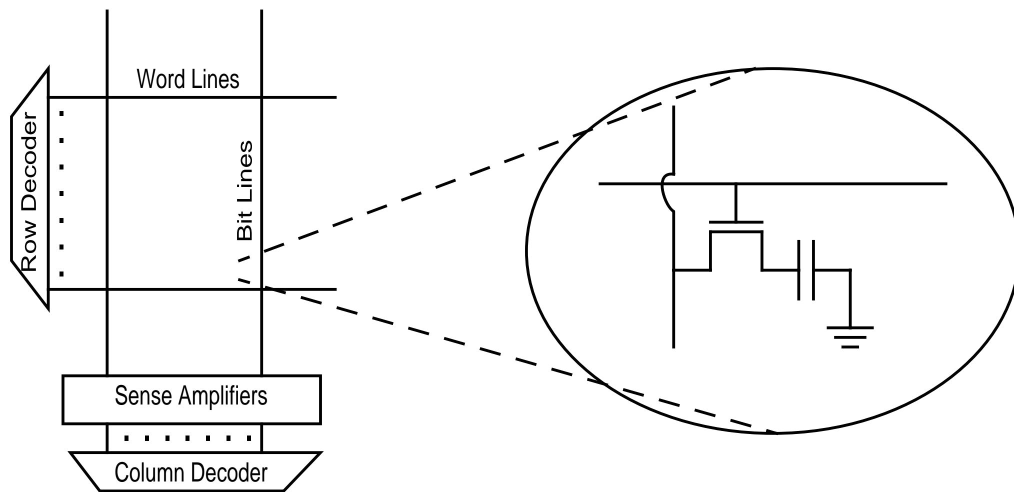


Figure 3.1: DRAM Array - Cell Placement

At the junction of each bit and word line, a cell like the one on the right with a single transistor capacitor pair is located

arrays are generally lower latency because of the lower capacitance associated with the word and bit lines. A larger number of arrays for a fixed address space will generally yield a higher level of parallelism because of the limitation on pipelining consecutive accesses to the same array or bank. Trying to improve performance either through increasing the number of arrays in the device, or by reducing the size of the arrays in the device will increase the amount of die area per bit required for the layout of the device. Increasing the area per bit of the device increases the cost per bit of the device, and thus we have a situation where cost and performance correlate positively, while it is advantageous to increase performance and decrease cost.

All DRAM accesses have at least three common phases through which the array must proceed. These phases are precharge, row access and column access. Figure 3.2 shows how these phases appear in the timing diagram for a hypothetical DRAM. In precharge, all word lines are deactivated, and the bit lines are driven to a median voltage. In row access (also known as activate), a single word line is activated, the capacitors of that row are allowed to discharge pulling the bit lines either lower or higher, a change which is sensed and amplified by the sense-amps. Finally, in the column access phase the subset of the data required from the entire row (or page) being held in the sense-amps is selected by the column decoder. Each of these phases has a characteristic timing

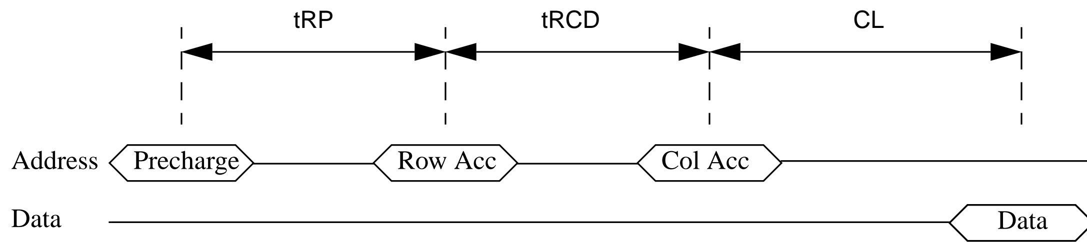


Figure 3.2: Timing Phases of a DRAM Access

This is a highly simplified diagram for a hypothetical DRAM access used to illustrate the three phases which must comprise all DRAM accesses.

parameter. These phases have different symbolic names depending upon the technology. Typically, however they are described as tRP — precharge time, tRCD — $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ delay, and CL — $\overline{\text{CAS}}$ latency or column access time. These are the symbolic names we will use for the remainder of this thesis. All three of these phases must take place for each and every DRAM access which occurs, but it is possible to amortize the precharge and row access times across multiple accesses, as well as hide the precharge time in certain architectures. Reducing the average observed latency in a DRAM device focusses upon hiding these access phases when possible.

DRAM have high yield requirements because of the small price margins. For this reason DRAM devices typically include some redundant bits in the architecture such that if a single cell or line fails, the device can still be configured to have full functionality, and thus be sold rather than scrapped. One of the reasons that smaller arrays incur area penalties is that as the arrays are reduced in size, the absolute number of redundant bits must be increased, and therefor the percentage of area which is devoted to these redundant bits is increased. Additionally small arrays generally result in a larger number of overhead circuits such as sense-amplifiers and buffers per storage cell than do larger arrays with equivalent capacity.

In discussing DRAM architectures, frequent references will be made to die area and the impact upon die area of a specific technique. Device cost is inversely related to die area exponentiated to a constant power, thus any technique which increase area increases cost at a dramatic rate [Patterson98]. The tight price margins on DRAM imply that any change which increases die area may price the device out of the marketplace. Currently,

only speciality DRAM such as graphics specific DRAM are successful at justifying increased costs for increased device performance.

3.3 Dynamic Random Access Memories (DRAM)

Memory constructed from ferrite cores was first demonstrated in a laboratory environment in 1948 by An Wang at the Harvard Computer Lab. Following this demonstration, so called "core memory" was first used in Project Whirlwind circa 1950-53. The single transistor Dynamic Random Access Memory (DRAM) was invented, by Dr. Robert H. Dennard, a Fellow at the IBM Thomas J. Watson Research Center, in 1966, and subsequently patented by Dr. Dennard and IBM in 1968. In 1970 Intel began production of the 1103, the first commercially available DRAM, a semiconductor device using PMOS logic circuits to store 1 Kilobit. By the mid to late 1970s the use of core memory was limited to unique applications such as the space shuttle where non-volatility was important. DRAM devices are being manufactured today in much the same way as they were in 1970, though with much smaller design rules and much greater densities. In the modern era of computing, DRAM has served as the primary memory technology of choice.

Initially, DRAM devices were designed with a minimum of I/O pins because the cost of chip manufacture was largely dominated by the number of I/O pins required by the package [Hennesy96]. During this same phase of development DRAM chips quite often had but a single data I/O pin, requiring eight chips for a Byte-wide bus or sixteen chips for a dual-Byte bus. This packaging model was sufficient when DRAM device sizes were 8 Kilobits, however as device sizes have grown, to 64 Mbit with 256 Mbit available in 2Q2000, the number of data designated I/O pins has increased to eight or sixteen per device.

Given the variety of device capacities and I/O, system designers have sought a means to use a variety of devices transparently within a system. This led to the development of the expandable primary memory system with sockets, and more recently slots holding removable carriers containing the DRAM devices. This model was propagated by the use of standardized Single In-line Memory Modules (SIMMs) which

integrated multiple DRAM chips into a standardized package, and subsequently DIMMs (Dual In-line Memory Modules) and RIMMs (Rambus In-line Memory Modules). The initial constraints limiting the number of I/O in DRAM design have some lingering affects upon the DRAM being produced even today, despite the fact that these constraints may no longer hold true. This consequence can be seen most immediately in the fact that the address pins for most DRAM are still multiplexed, and the width of the data bus on a conventional DRAM is only now beginning to exceed a single Byte. Recently, as the integration levels have increased, it is no longer the case that I/O dominates manufacturing costs. As a consequence, pin-out has increased, primarily in the data I/O where up to 16 data pins are used for some modern DRAM chips. This makes the generation of a wide data bus more feasible at lower memory size configurations, because the number of chips required for the bus size does not drive up the smallest possible memory configuration.

Bandwidth

Bandwidth alone cannot solve the problem characterized as the memory wall. Some performance loss results from the processor core sitting idle during the DRAM latency for critical data before it can continue execution [Cuppu99]. Bandwidth can be increased for any interface or architecture by increasing the bus width or replicating the channel. Thus we see a proposal for the next generation Alpha microprocessor that contains four DRDRAM channels for a potential primary memory bandwidth of 6.4 GB/s [Gwennap98]. However, for a fixed size working set and reasonable latency, an interface with higher bandwidth is going to be utilized a lower percentage of the time, meaning an access issued during that time is less likely to be queued up waiting for prior requests to be serviced. In this way bandwidth can have some effect upon observed processor latency. There are applications — primarily streaming applications — that are bandwidth limited due to access pattern. For these applications, and due to the reduced queuing latencies observed with increased bandwidth, increasing the bandwidth to the primary memory system will increase performance. The downside to increasing the bandwidth by expanding bus width is an increase in system cost and the fact that memory must be added in larger quantities. Increasing the bus speed will also increase the bandwidth, without the width costs, but results in higher complexity and higher power consumption.

Access Concurrency

One of the questions we seek to answer is how much performance may improve with support for concurrency in the DRAM system, and what degree of concurrency the DRAM system must support to obtain reasonable results. There have been no published limit studies that determine how much DRAM-level concurrency a modern CPU can exploit, and there has been no published comparison between DRAM systems with and systems without support for multiple simultaneous transactions. What has been published suggests that there are gains to be had exploiting concurrency at the DRAM level, but they require sophisticated hardware such as MSHRs or some other mechanism for supporting multiple outstanding memory requests, and resolving the associated dependencies [Cuppu99].

In the following sections, we will discuss 1) architectural level configurations, 2) interface specifications, 3) array enhancement techniques, 4) cache enhancement techniques, and combinations of the above. While each of these approaches to improving DRAM performance have been referred to as a DRAM architecture enhancement, it is important to differentiate between them, in that some combinations of multiple approaches are possible. Section 3.3.1 — Conventional Asynchronous DRAM through Section 3.3.9 — Direct Rambus (DRDRAM) could be classified as both architecture and interface specifications. Section 3.3.10 — FCDRAM and Section 3.3.11 — Multibanked DRAM (MDRAM) are both core improvement techniques which could be used with most DRAM implementations. These enhancements require interface modifications for optimal performance. Section 3.4.1 — Cache DRAM (CDRAM) through Section 3.4.3 — Virtual Channel DRAM (VC DRAM) are proposals for integration of SRAM cache onto the DRAM device with the intent of increasing the performance of that device. Not all of these architectures will be simulated, but they will all be discussed with regard to which characteristics provide improved performance, and at what cost.

3.3.1 Conventional Asynchronous DRAM

Conventional DRAM follows the description given in Section 3.3, without any of the enhancements that differentiate the DRAM which will be described in the subsequent sections. It is convenient that the DRAM architecture which is chronologically first is also

the most basic in function and simplest to understand. Conventional DRAM is no longer manufactured, but most modern asynchronous DRAM are backwards compatible to this specification. This is because it takes very little die area to add the additional features found on updated asynchronous DRAM.

Conventional DRAM use a split addressing mechanism, but with a dedicated bus. The split addressing mechanism means that row and column address components are sent separately at two different times on the bus. This technique, still used in the majority of DRAM in production, is a carryover from the days when chip I/O was the price determining factor. Nevertheless, this mechanism has worked well with the DRAM architecture. In standard DRAM addressing, the address bus is multiplexed between these two components, row and column. In order to tell the DRAM chip core which (if either) of these signals is being driven on the address bus, the row and column address strobe signals, $\overline{\mathrm{RAS}}$ and $\overline{\mathrm{CAS}}$ respectively, are asserted, allowing the appropriate values to be latched by the DRAM core. This addressing scheme matches the DRAM architecture because in typical DRAM core designs, half of the address (the row value) is required a significant period of time before the other half of the address (the column value). This is a consequence of the fact that a DRAM is typically configured as an array as shown in Figure 3.3, “Conventional DRAM architecture”. In this arrangement, the first step of an access is to apply the row bits of the address, causing the retrieval of a complete row from the memory array. It is not until these row values from the array have propagated down the bit lines to the sense amps, and the sense amps have stable output that the column values are selected. The column component of the address is required for the column decoder MUXes, which follow the sense amps, to select the appropriate data and drive it to the output pins. Therefor, the multiplexed address bus is not necessarily a critical limitation to the cycle time of DRAM [Ng92].

The primary consequence of the dynamic nature of DRAM is that for a certain quantum of every unit time, the memory array is not available because the internal circuits must be periodically recharged. With early DRAM circuits, the refresh was executed by a set sequence or combination of inputs to the DRAM. This meant that the memory controller was responsible for refreshing the circuit, or the memory contents could be lost. This externally generated refresh has the positive attribute that the controller can execute

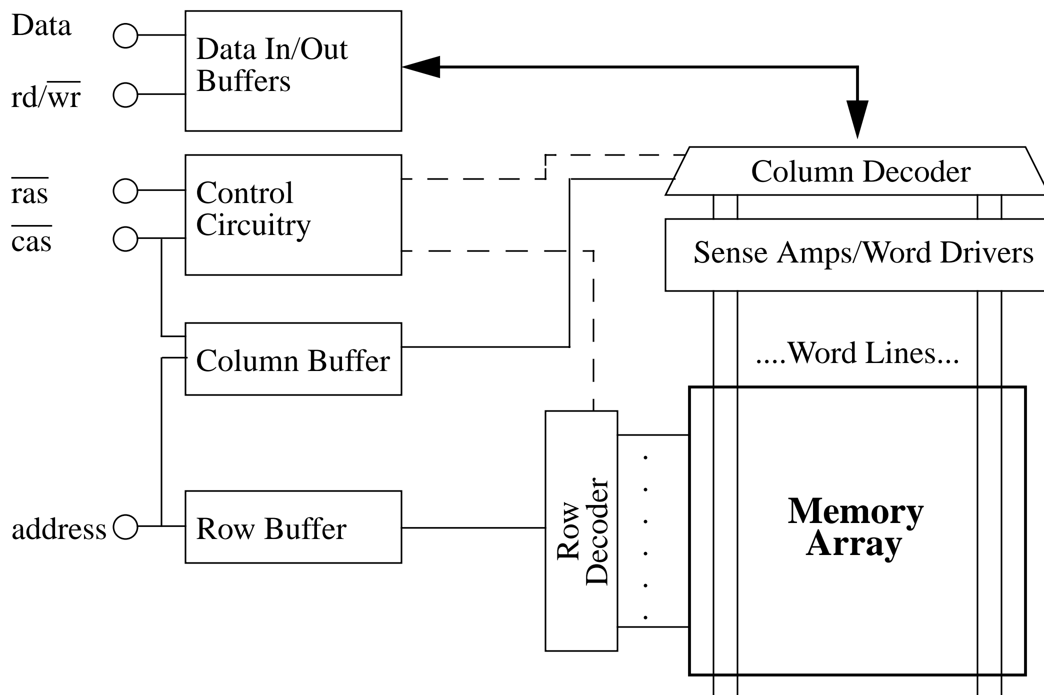


Figure 3.3: Conventional DRAM architecture

The block diagram for a conventional DRAM shows buffers or drivers in a number of locations, but the key item to note is there are no clocked latches or circuits - control is maintained over all internal signals by the memory controller

refresh cycles when it knows that there are no memory requests pending. More recently, DRAMs have started to integrate internal refresh capability, along with the compatibilities for externally generated refresh onto the DRAM die. This choice allows for external refresh, with knowledge about pending requests where the controller is intelligent enough to take advantage of this capability, with the convenience of internally managed refresh for simplicity in the memory controller.

Figure 3.3, “Conventional DRAM architecture,” on page 39 is a highly simplified version of an actual implementation. Signals and circuitry not explicitly shown will become more apparent as this circuit is compared to various enhanced DRAM.

3.3.2 Fast-Page Mode (FPM) DRAM

Fast-Page mode DRAM was the first enhancement to conventional asynchronous DRAM. The primary difference between a fast-page mode DRAM and a conventional asynchronous DRAM is that in a conventional DRAM, even if two sequential accesses

share the same row bits in the address, the $\overline{\text{RAS}}$ signal must be de-asserted and re-asserted providing the same values. A large majority of observed accesses are to the same row due to sequential data accesses or large cache line fills. In such cases, this extra set of transitions in the $\overline{\text{RAS}}$ signals adds time to the overall access profile. In the case of a fast-page mode DRAM, so long as the row component of the address remains constant, the $\overline{\text{RAS}}$ signal may remain asserted, allowing for a faster access cycle in the case of subsequent accesses to addresses with identical row component addresses [Ng92].

The change to the DRAM architecture which enables the fast-page mode operation is a simple modification. The sense amps must hold a value through a change on the address inputs and column address latch, while the row address latch values remain stable. This requires modification to clock signals and the row and column decoders. The modification facilitates accepting and propagating a change in column address after a transition in $\overline{\text{CAS}}$, without a preceding transition in $\overline{\text{RAS}}$. There would be no change in the block diagram from the one shown in Figure 3.3, “Conventional DRAM architecture,” on page 39. The impact of this modification on the die area of a conventional DRAM is insignificant. Because the area impact is insignificant, DRAM manufacturers can produce FPM DRAM and market them as either FPM DRAM or conventional DRAM and take advantage of the cost savings inherent in mass production. This statement is also true of a number of the other modified architectures for DRAM which do not involve placing significant cache on the DRAM die.

3.3.3 Extended Data Output (EDO) DRAM

EDO DRAM or Extended Data Output DRAM is sometimes referred to as hyper-page mode DRAM. The difference between a conventional DRAM and an EDO DRAM are again very small, but in this case there are circuits to be added to the die to facilitate the modification. To create an EDO DRAM, add to the conventional DRAM configuration an additional latch between the sense-amps and the outputs of the DRAM package. This latch holds output pin state and permits the $\overline{\text{CAS}}$ to return high much more rapidly, allowing the memory array to begin precharging faster, in preparation for the subsequent access [Bursky95]. In addition to allowing the precharge to begin faster, the latch in the output path also allows the data on the outputs of the DRAM circuit to remain valid for

longer into the next clock phase. This makes it easier to meet hold constraints on the latches. Figure 3.4 shows how the addition of latches following the column decoders in the

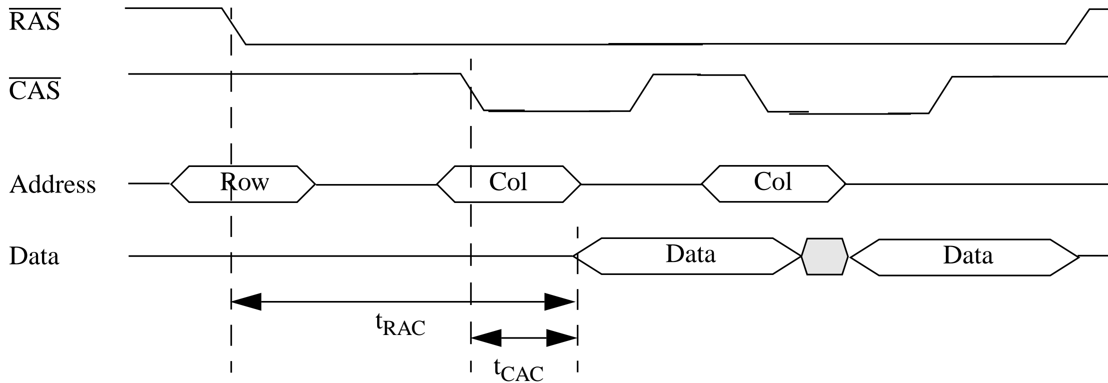


Figure 3.4: EDO Page-Mode Read Interface Timing

This diagram shows the increased overlap between valid data and the column for the subsequent access

devices allows EDO DRAM to continue to maintain valid data even after the $\overline{CAS}$ line has been deasserted, and the retrieval of subsequent data from the open page has begun.

So far as the block diagram of an EDO DRAM is concerned, the only modifications made from the block diagram for a conventional DRAM are those shown in Figure 3.5. These modifications reflect the addition of a latching circuit in the path from the column decoder to the data outputs, and an additional signal from the control circuitry required to strobe that latch.

3.3.4 Burst EDO DRAM

Burst EDO DRAM are designed similarly to the EDO DRAM shown in Figure 3.5 but include an internal counter holding the column address. This allows a toggle of the $\overline{CAS}$ line to step through the sequence programmed into the burst counter. This in turn provides for the possibility of higher bandwidth data transfers [Bursky95] without ever requiring access to the memory array. The outputs of the sense amps are used as a localized cache of the same width as one row in the memory array.

The modifications which would be required to an EDO DRAM in order to enable a bursting capability are the addition of a burst counter into the column address path, changes to the clocking and refresh circuitry to deal with the addition of the burst counter,

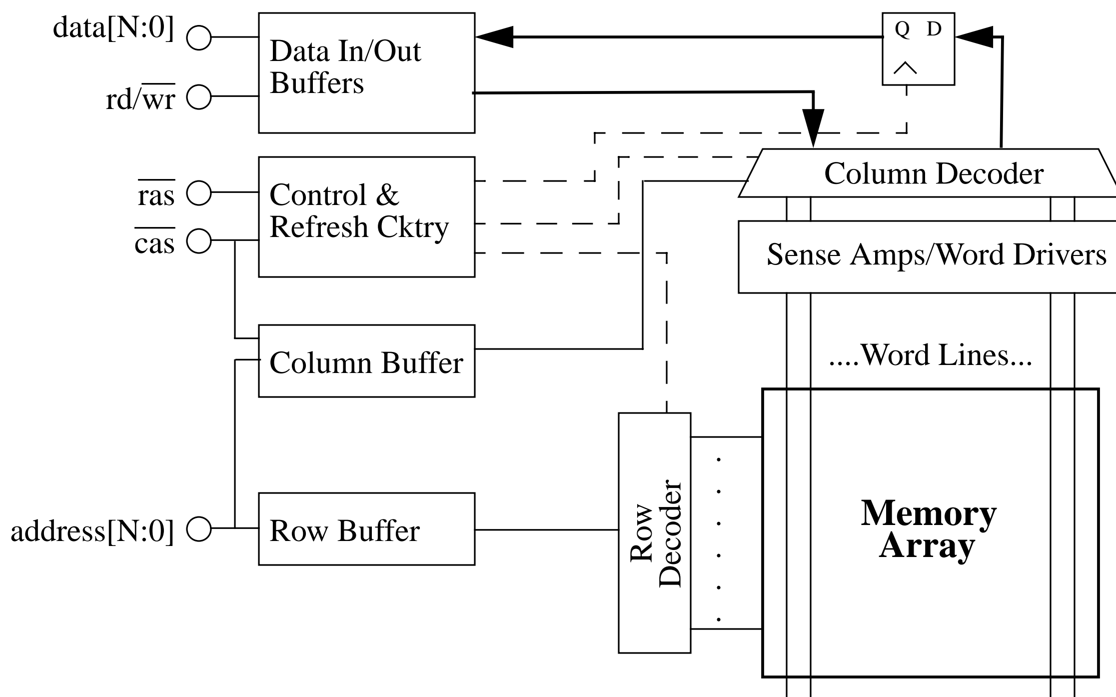


Figure 3.5: Extended Data Out (EDO) DRAM block diagram

The primary change from a Fast-Page-Mode Asynchronous DRAM to a EDO Asynchronous DRAM is the addition of latches after the column decoder which allows the data to continue to be driven on the output bus, even while the array is being precharged, or new data is being selected by the column decoder

and an adder to facilitate toggling through sequential addresses. A block diagram of the Burst EDO Dram architecture is shown in Figure 3.6. This figure shows the addition of the burst counter, as well as the fact that the address going to the column decoder can come from either the burst counter, or the column buffer. These may be one register combining both functions. The burst EDO DRAM requires a few additional interface pins as well, to differentiate between a standard transaction and a burst transaction. In this configuration, and in all burst EDO DRAM currently available, a burst cannot cross a row address boundary. A change between row addresses requires a transition on the $\overline{\text{RAS}}$ signal, which effectively forces the circuit to exit the burst mode.

3.3.5 SDR Synchronous DRAM (SDRAM)

Synchronous DRAM, as the name implies, requires a clock for usage, unlike the DRAM previously discussed which are all asynchronous. They may contain latched circuitry, however the control signals are derived from interface signals, and only refresh

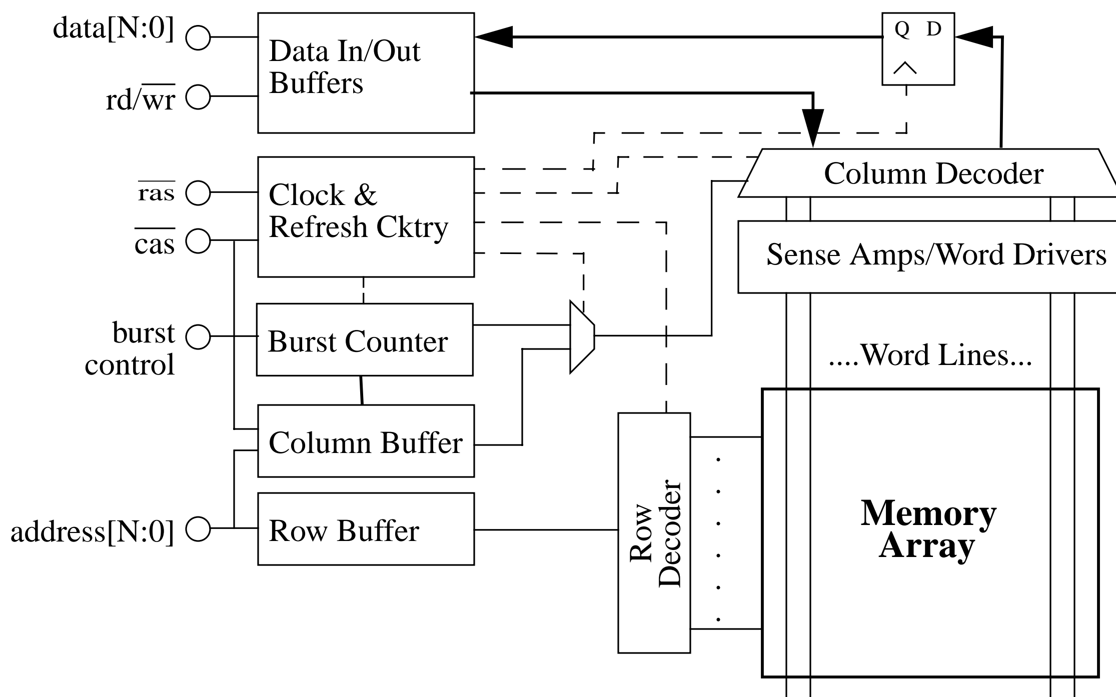


Figure 3.6: Burst EDO DRAM block diagram

The primary change between an EDO and Burst EDO asynchronous DRAM is the addition of a burst counter to allow a single $\overline{\text{CAS}}$ command to initiate multiple responses on the data bus.

serves as an upper bound on the access timings. One of the disadvantages of moving to a synchronous DRAM is that access requests must be presented to the DRAM at a specific point in the clock cycle. This means that the data is not necessarily available at a specific latency following the request, as is the case with asynchronous designs. The latency of a request is variable and depends when in the DRAM cycle the request arrives at the DRAM input pins [Jones92].

The synchronous DRAM does not have any inherent characteristics which make the core timings (tRP, tRCD, CL) any lower than those of an asynchronous DRAM equivalent in size and process. SDRAM makes advances in performance by the concurrency and bandwidth of the interface. In addition, because the control signals for SDRAM functional blocks are internally generated, and based off of an interface clock, the latencies are less than those for asynchronous DRAM which require driving the control signals across the entire length (and load) of a shared motherboard bus. The block

diagram for Synchronous DRAM, Figure 3.7. has a number of changes from the previous

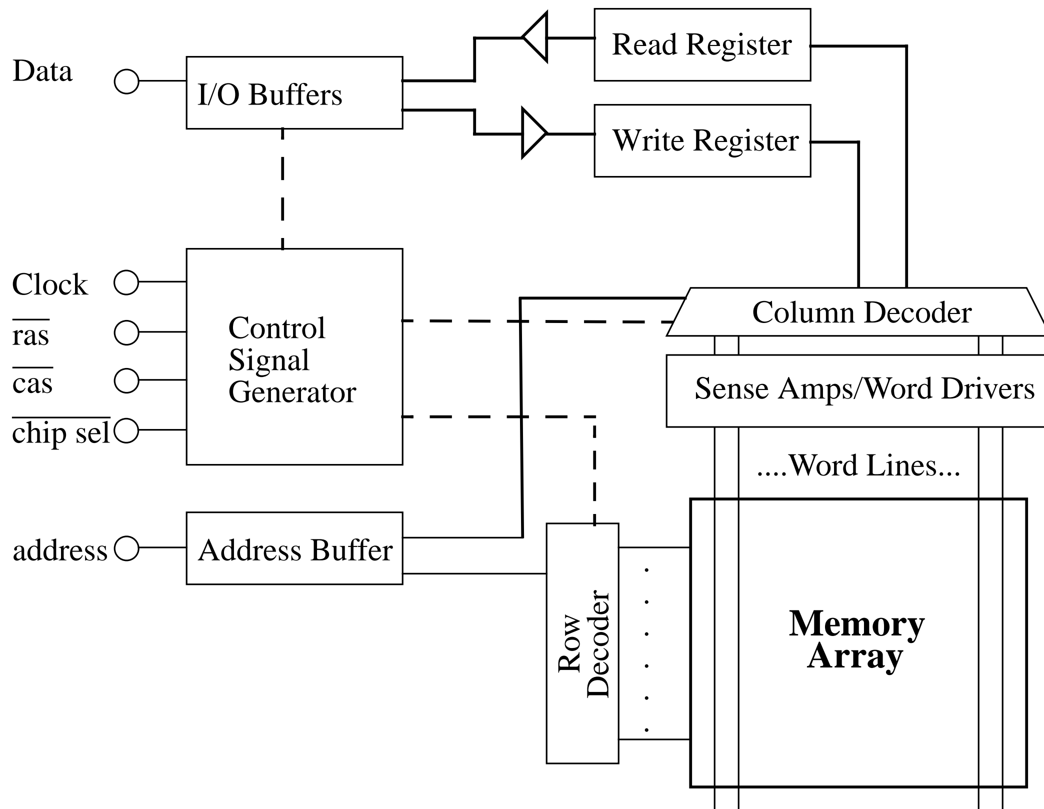


Figure 3.7: Synchronous DRAM block diagram

Synchronous DRAM require the addition of control logic because the off-chip interface can now be shared between multiple accesses. Buffers are also required to enable pipelined data to be driven onto the data portion of the interface.

block diagrams. The most apparent of these is the relocation of the sense-amps and word drivers to the opposite end of the memory array. Because of the manner in which these circuits function, essentially just driving or sensing minor charge changes in the word lines and amplifying them, it is irrelevant at which end of the circuit they are located. The block diagrams given in this document are general overviews, each vendor may choose to structure their products in a different layout. The differences which are of interest in Figure 3.7 are the facts that the control signal generator now has a clock coming into it, and that all of the control signals to the decoders, I/O buffers, and circuit components not shown originate at the control signal generator. The external clock simplifies the generation of many of these signals, and the regular timing of the control signals make the design of the memory array less complex [Mitsubishi95a].

The SDRAM interface is the first significant divergence from the asynchronous interface. Figure 3.8 shows the interface to a PC100 SDR SDRAM device. The

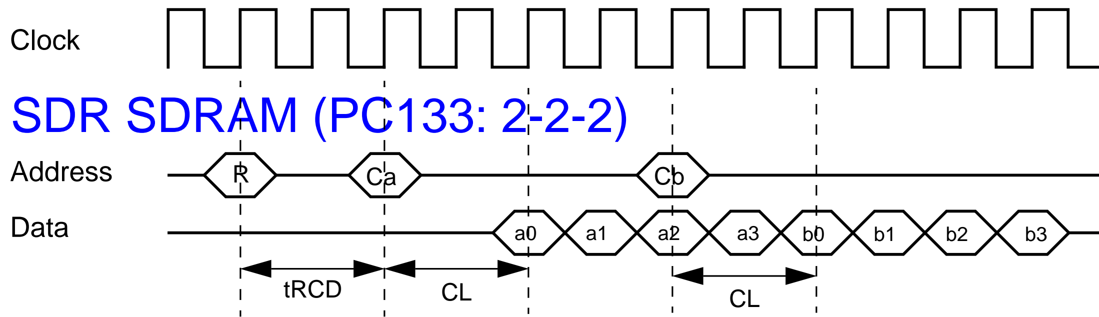


Figure 3.8: SDR SDRAM Timing Diagram

This diagram shows accesses to two unique columns of a single page. Precharge is assumed to happen prior to this timing diagram. tRCD, the $\overline{RAS}$ to $\overline{CAS}$ delay, is 2 cycles, as well as the $\overline{CAS}$-latency.

immediately apparent difference between an asynchronous DRAM and a synchronous DRAM is the presence of the clock in the interface. The most significant result of this clock is that the interface must no longer be dedicated to a single transaction from initiation of a request to completion of a request. As is shown in Figure 3.8 a request may be initiated on the address lines, and prior to the response to this request on the data lines, a subsequent request can be initiated on the address lines. This has the effect of interleaving the bus accesses to achieve access concurrency as is described in Section 2.2.2. SDRAM maintains the ability to burst out multiple bus cycles of data, as was introduced in the BEDO asynchronous DRAM. For example, a PC100 SDRAM device must be able to perform accesses of 1, 2, 4, 8, or a full-page data in order to conform to the standard [IBM98].

Synchronous DRAM are currently available at clock speeds from 66 MHZ to 150 MHz, implying a clock period on the DRAM bus of between 15nS and 6.67nS respectively. This initially appears to be a substantial improvement over asynchronous DRAM, which are available in the range of 50-80 nS. However, synchronous DRAM require multiple cycles (precise number depending upon the state of the device) to complete a transaction, and incur additional synchronization penalties if the access does

not arrive at the interface precisely at the start of a cycle. Thus the advantages of Synchronous DRAM are overemphasized by these figures.

3.3.6 DDR SDRAM

Double Data Rate (DDR) SDRAM is different from SDR in that unique data is driven and sampled at both the rising and falling edges of the clock signal. This effectively doubles the data bandwidth of the bus versus an SDR SDRAM running at the same clock frequency. DDR266 devices are very similar to SDR SDRAM in all other characteristics. Figure 3.9 shows how DDR allows unique data to be transmitted on both the rising and the

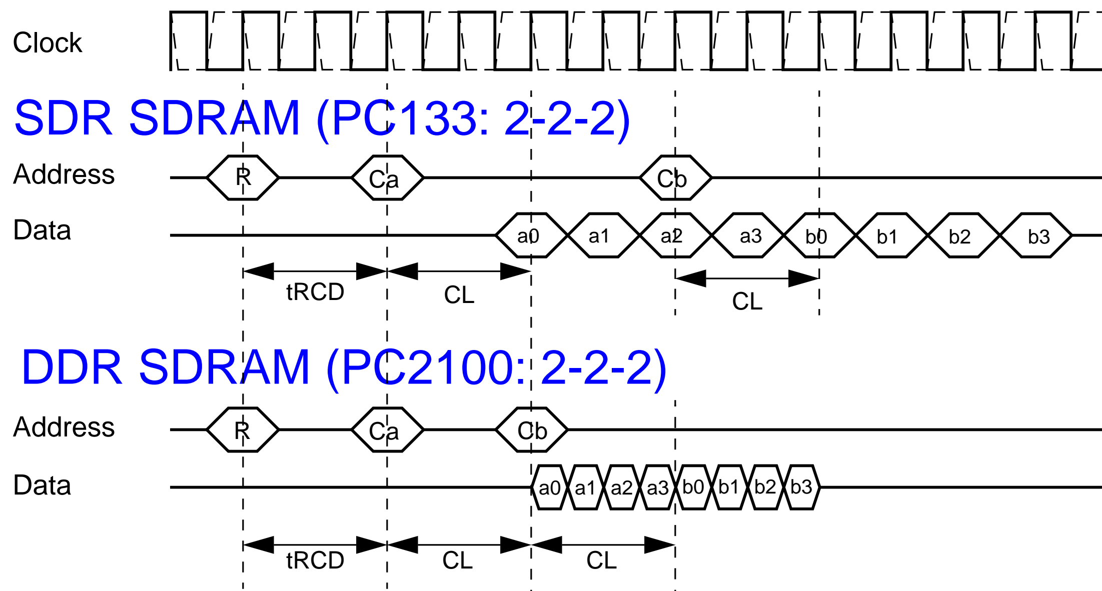


Figure 3.9: DDR SDRAM Read Timing Diagram

This figure compares the DDR and SDR DRAM interfaces. The potential bandwidth of the DDR interface is twice that of the SDR interface, but the latencies are very similar.

falling edges of the interface clock. The interface clock for DDR SDRAM is a differential signal as opposed to the single signal clock of the SDR SDRAM interface. Beyond this, SDR SDRAM and DDR SDRAM use the same signalling technology, the same interface specification, and similar pinouts on the DIMM carriers. The JEDEC specification for DDR266 devices provides for a number of "$\overline{\text{CAS}}$-latency" speed grades. The $\overline{\text{CAS}}$ latency (CL) determines the number of clock cycles between the column portion of the address being driven onto the address signals of the bus and the data requested being driven onto

the data signal. Chipsets are currently under development for DDR266 SDRAM and are expected to reach the market in 4Q2000. DRAM manufacturers are confident in their ability to yield on DDR SDRAM parts which will run at 133 MHz, 150 MHz, and 166 MHz. These devices will be referred to as DDR266, DDR300 and DDR333 respectively, while the DIMMs carrying these devices will be marketed under the bandwidth designations of PC2100, PC2400 and PC2600 respectively [Peters00].

3.3.7 DDR2

The DDR2 specification under development by the JEDEC 42.3 Future DRAM Task Group is intended to be the follow-on device specification to DDR SDRAM. While DDR2 will have a new pin-interface, and signalling method (SSTL[1]), it will leverage much of the existing engineering behind current SDRAM. The initial speed for DDR2 parts will be 200 MHz in a bused environment, and 300Mhz in a point-to-point application, with data transitioning on both edges of the clock. Experiments discussed in Chapter 6 will focus upon the bused environment, as that is the structure of the memory system most commonly used in a general purpose computer. Other significant changes from past SDRAM architectures are: a fixed burst length of 4 data cycles, a programmable additive latency for enabling posted-$\overline{\text{CAS}}$ transactions, a write latency not equal to one, differential clock signaling and micro-BGA packaging. These changes will be described in more detail further into this section. The DDR2 specification is still subject to revision as of this writing, but the information contained here is based upon the most recent drafts for DDR2 devices and conversations with JEDEC members.

The DDR2 specification is an attempt to provide a common design target for many DRAM vendors and system architects. It is hoped that early standardization will minimize design fragmentation and thus benefit the consumer through lower prices. While it contains significant changes, DDR2 utilizes much of the same architecture of the earlier SDRAM. However, there are some notable differences that are not simply derived from

1. SSTL - Stub Series Terminated Logic: Described in JEDEC standard No. 8 Series, Low Voltage Interface Standards; JESD8-8 (3.3v) and JESD8-9 (2.5v) available at www.jedec.org

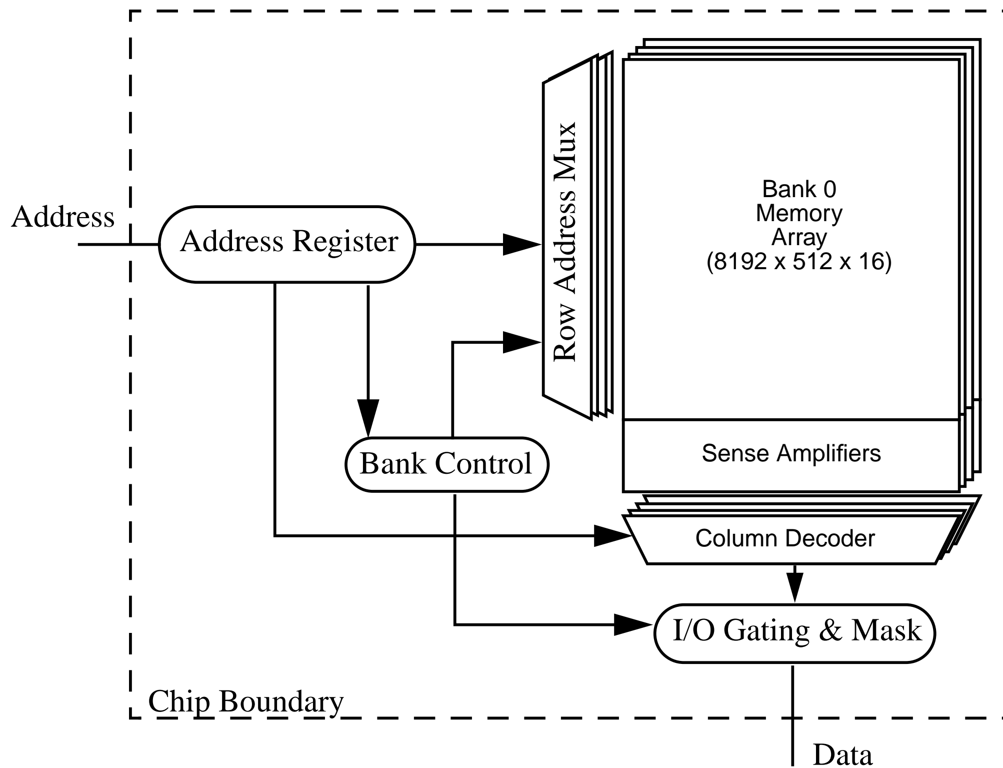


Figure 3.10: 256 Mbit DDR2 Architecture
This figure shows the architecture of the DDR2 core, which includes 4 64Mbit banks, and the associated interface logic required for the DDR I/O.

evolution of existing SDRAM, but are intended to provide better resource utilization or lower latencies in the DDR2 devices.

DDR2 with a 64 bit wide desktop bus, switching data signals at 400 Mhz, has a potential bandwidth of 3.2 GB/s; this surpasses any current DRAM architecture. Further, if server bus widths of 256 bits remain constant when DDR2 is introduced to the server architecture, potential bandwidths of 12.8 GB/s will force the re-design of the processor front-side bus to support this throughput.

DDR2 has support for concurrency in the DRAM system, but no more so than other SDRAM architectures. Additionally, since DDR2 is targeted at large devices (greater than 256 Mbit) with only four banks, it may be that the amount of attainable concurrency is less than that of architectures containing more numerous smaller banks with the same address space. Low latency variants are intended to support for more

concurrency, as they can service requests out of the cache while longer-latency accesses are serviced by the DRAM core.

Access Granularity of Four

The DDR2 specification, in the most recent draft, sets the access granularity of all reads, and the maximal size write of DDR2 devices to 4 data cycles, or 2 clock cycles. This is in contrast to PC100 parts that allow bursts of 2, 4, 8, or full-page [IBM98] and Direct Rambus parts that allow bursts of any power-of-2 octcycles (128 bit quantities) up to the page size [IBM99]. What impact does this access granularity limitation impose upon the DDR2 parts? If we examine a 256 Byte transaction which would require 4 $\overline{\text{CAS}}$ requests using a burst size of 8 in a PC100 environment, or a single COL packet in a DRDRAM environment, the same transaction will require 8 $\overline{\text{CAS}}$ requests using the fixed access size of 4 in the DDR2 environment. Data bus usage (in bytes) is constant in this example; however the fraction of time that the address bus is utilized increases for DDR2. It remains to be seen if this additional loading of the address bus will impact performance. It may potentially reduce the ability to perform tasks that do not require the data bus (i.e. refresh) in the background while performing reads from independent banks. One motivation for taking this approach is that the DDR2 interface does not support interrupting transactions once they have been initiated. In PC100 or DRDRAM systems bursting a full-page of data, it may be required to terminate the transaction early for another, higher priority, transaction. Since the DDR2 transactions are smaller, the support for termination of in-flight accesses need not be present.

Additive Latency (Posted-$\overline{\text{CAS}}$) and Write Latency

The posted-$\overline{\text{CAS}}$ enhancement of the DDR2 specification hinges upon the addition of a parameter called Additive Latency (AL). The AL parameter enables the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ packets in a DRAM access to be driven onto the address bus in adjacent clock cycles. Figure 3.11 shows the impact of a non-zero AL upon the timing diagram of a DDR2 read transaction. In the second portion of the diagram, the $\overline{\text{CAS}}$ packet is driven onto the address bus in the cycle immediately following the transmission of the $\overline{\text{RAS}}$ packet. The DDR2 device interface then holds the $\overline{\text{CAS}}$ command for the AL latency prior to issuing it to the core. This allows for a more logical ordering of the access packets occurring on the

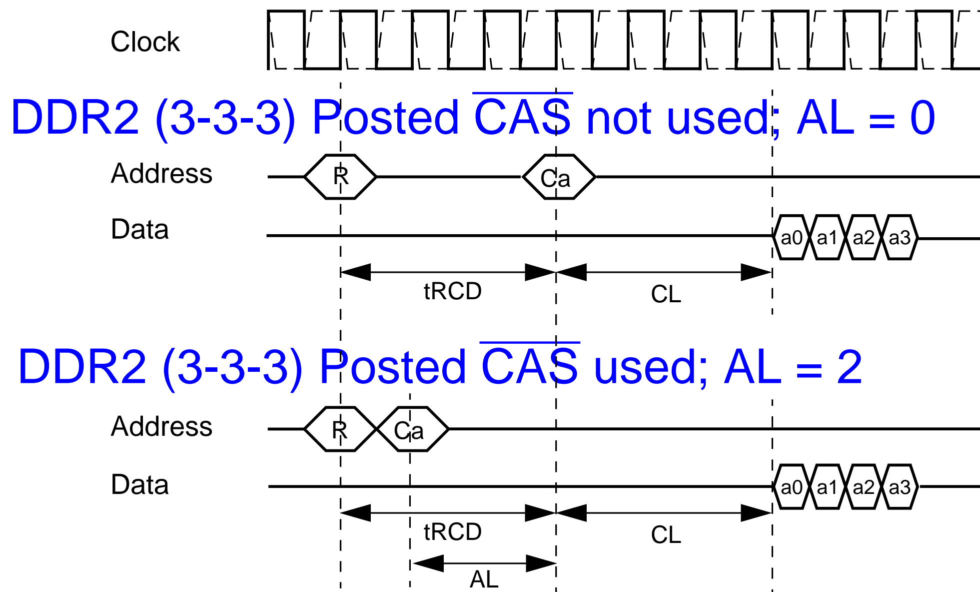


Figure 3.11: DDR2 Posted $\overline{\text{CAS}}$ Addition

The posted $\overline{\text{CAS}}$ addition to the DDR2 specification allows the $\overline{\text{CAS}}$ packet to be transmitted in the next clock cycle following the $\overline{\text{RAS}}$ packet. This allows two packets which are logically part of a single transaction to be placed onto the bus together, and may allow for higher address bus utilization.

address bus and may allow for a higher address bus utilization than without the posted-$\overline{\text{CAS}}$ enhancement

Write Latency (WL) for a DDR2 device is not a single cycle, as it is for current SDRAM, it is instead set to Read Latency (RL) minus 1, i.e. WL = (RL-1). RL is programmable in bus cycles, providing flexibility for devices with differing core parameters. Figure 3.12 shows the difference between a DDR266 device and a DDR2 device with regard to a write. The relationship between RL and WL added in the DDR2 specification has the property of eliminating the idle data bus cycles associated with transitioning from a write to a read command in the current SDRAM bus protocols. Similar to the posted-$\overline{\text{CAS}}$ enhancement, a WL greater than 1 also has a simplifying effect upon the access stream timing diagrams, and allows for higher utilization of the data bus.

3.3.8 Conventional RAMBUS (RDRAM)

Rambus is the name of a company as well as a family of DRAM memory system products. Rambus is a “fab-less” corporation which designed a new interface for DRAM

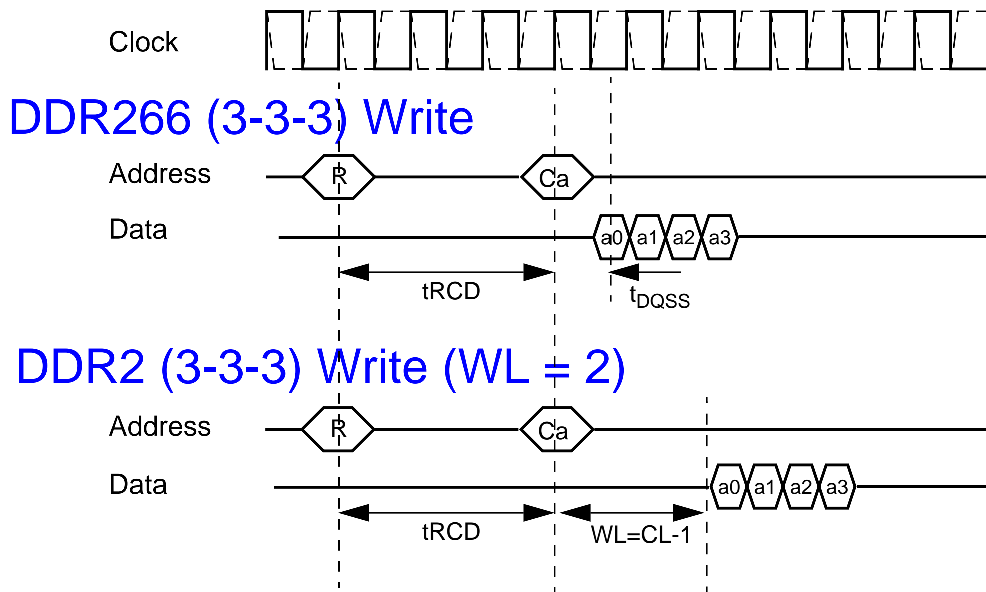


Figure 3.12: DDR2 Non-Zero WL Addition

Each of these architectures would utilize a different frequency clock, but assuming a 133 Mhz clock for the DDR266, and a 200 Mhz clock for the DDR2 architecture, this figure shows the difference in the profile of the write transaction. The intent making WL = RL - 1 in the DDR2 architecture is to eliminate the idle bus cycles associated with the transition from a write to a read transition in the DDR266, and other SDRAM, architectures.

as well as new circuit configurations. A large number of fabrication houses, some of which are Intel, Micron, Infineon, Hitachi, LG Semiconductor, NEC, Oki, Samsung, and Toshiba, have licensed the Rambus architecture from Rambus Inc. and are currently manufacturing Rambus devices.

The Rambus architecture utilizes a packet based interface, on the physical layer called the Rambus channel. The Rambus channel is at least as innovative as the Rambus architecture itself. There are thirty communication lines which encompass a Rambus channel. Of these thirty physical connections only 9 are actually used for communication, the other 21 are used for power distribution, reference voltage distribution, clock distribution, and bus arbitration. These 9 communication signals are very high speed signals, operating at 500MHz. This speed and bus size gives a single Rambus channel a theoretical peak bandwidth of 500 MBytes per second. This bandwidth can never be achieved however because some fraction of the data traversing the channel is control and access information. The bandwidth of an overall system can be increased by increasing the number of Rambus channels in the memory architecture. For instance, since each Rambus

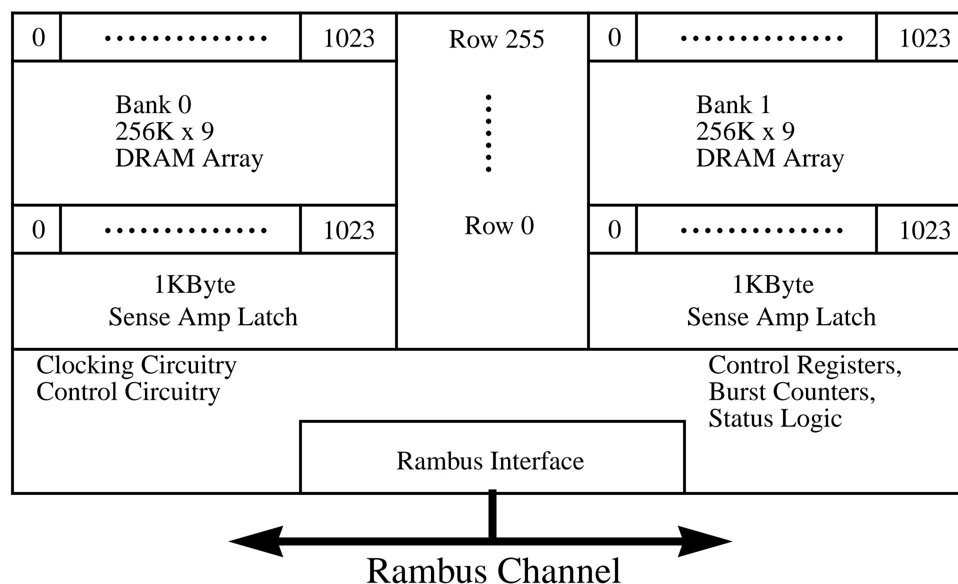


Figure 3.13: 4Mbit RDRAM Block Diagram

The 4Mbit Conventional RDRAM device had 2 256x1KByte banks, and a 9-bit shared address/data channel. [Rambus93]

channel essentially provides a single Byte each 2 nS, four Rambus channels could be configured to provide 4 Bytes, or a 32 bit word each 2 nS. [Rambus92] The Silicon Graphics, Inc. Indigo^2 IMPACT system uses six Rambus channels for a theoretical peak 3 GBytes/sec. bandwidth. The problem with increasing the number of channels in this manner is that both the cost of the system increases, and the minimal amount of memory for which the system can be designed increases.

The block diagram for the Rambus architecture, as given in the Rambus Architecture Overview [Rambus93] is shown in Figure 3.13, “4Mbit RDRAM Block Diagram”. Note that this diagram is for the 4 Mbit RDRAM die, but the 16 and 64 Mbit versions vary significantly from this diagram. For instance, the 64 Mbit version has four independently operating banks, rather than the two banks accessed in parallel.

Conventional RDRAM is most notable in its use in the Nintendo64 game machine. This design was rapidly foregone in favor of higher bandwidth solutions such as Rambus’s follow-up design the Direct Rambus specification.

3.3.9 Direct Rambus (DRDRAM)

The Direct Rambus architecture was the third specification to be released by Rambus Inc. following on Conventional and Concurrent RDRAM. Direct Rambus DRAMs use a 3-Byte-wide channel (2 for data, 3 bits for row, 5 bits for column) to limit the number of interconnections required. The total signals required for the interface also include 4 clock signals (two differential clocks) and 4 signals for serial communication used in initialization. This means only 32 (34 with parity) signals are required to cross the chip boundary on the DRAM device or the controller, whether that controller is located on a bridge-chip, processor or graphics engine. The DRDRAM interface was originally targeted to operate at 400MHz, however devices have been released to operate at 300MHz, 350MHz, 400MHz, and 533MHz. There are many aspects of the DRDRAM architecture which make it different from SDRAM architectures. Primary among these may be the fact that each DRDRAM device is intended to service the entire bus width when an access maps into the address space covered by that device. This is in contrast to SDRAM where a single access will retrieve data from 16, 8 or 4 devices concurrently, depending upon whether the carrier (DIMM) uses x4, x8 or x16 devices. Figure 3.14 shows the architectural diagram for a 128Mbit DRDRAM device. This figure includes the “core” of the DRAM device with the data storage elements, but does not show how these devices are used together in a primary memory system. Figure 3.15 shows the DRDRAM interface for

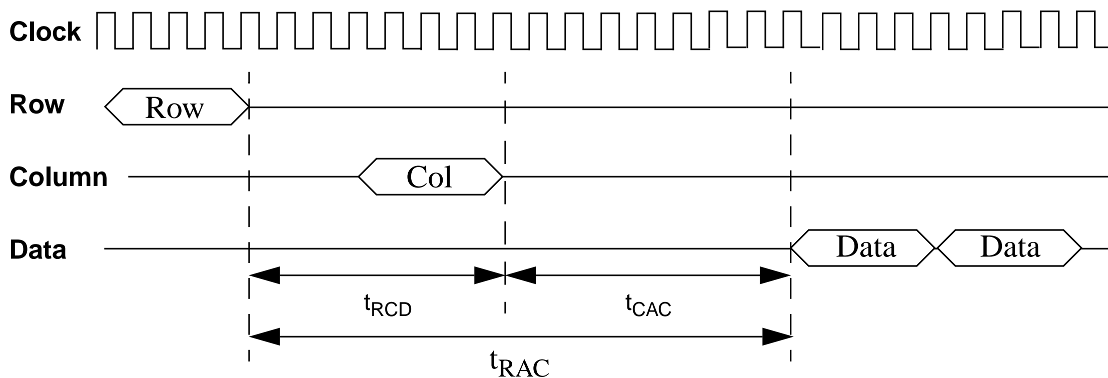


Figure 3.15: DRDRAM Interface - Simple Read Operation

This figure shows the interface between the DRDRAM and the controller.

a series of read commands followed by a series of write commands. The DRDRAM

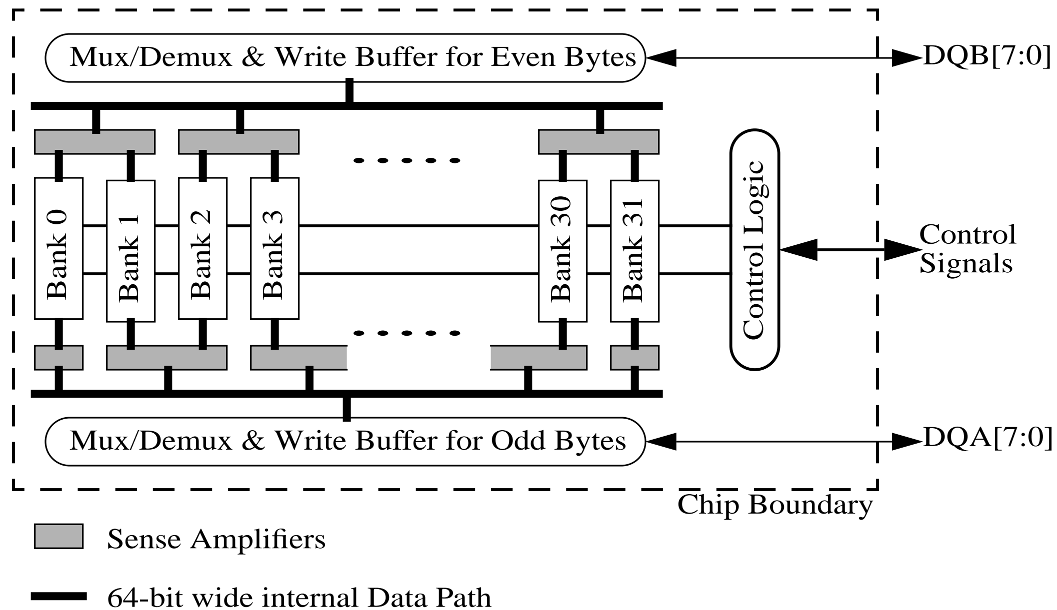


Figure 3.14: 128 Mbit DRDRAM Architecture

The DRDRAM architecture allows a single device to cover the entire bus, has many more banks for a fixed address space, and shares sense-amps between neighboring banks.

devices use DDR signalling, for both address and data signals, implying a maximum bandwidth of 1.6 Gbytes/s. These devices have many banks in relation to SDRAM devices of the same size. Each sense-amp, and thus row buffer, is shared between adjacent banks. This implies that adjacent banks cannot simultaneously maintain an open-page, or maintain an open-page while a neighboring bank performs an access. This organization has the result of increasing the row-buffer miss rate as compared to having one open row per bank, but it reduces the cost by reducing the die area occupied by the row buffer [IBM99].

The DRDRAM specification, even after years of development may still be “under development” as recent news suggests that the architecture may change such that the number of banks per device may be reduced in order to decrease the cost premiums associated with producing the current 32 bank per 128Mbit devices [Cataldo00]. This move, referred to as the “4i” initiative, is likely to reduce the concurrency, or parallelism available to the application, however with address remapping, or similar techniques, the performance impact should be minimal. The DRDRAM specification already has the inherent advantage that even with a reduced number of banks per device, there are likely

to be more banks mapping a given address space because the bus is not shared among multiple devices as it is in SDRAM (both SDR and DDR) devices. While the existing DRDRAM specification has 32 dependent banks (sense-amps are shared) the new architecture will not share sense-amps. This independence between banks may be the source of the “i” in the “4i” initiative designation.

3.3.10 FCDRAM

Fast Cycle DRAM (FCRAM) developed by Fujitsu is an enhancement to SDRAM which allows for faster repetitive access to a single bank. This is accomplished by dividing the array not only into multiple banks but also small blocks within a bank. This decreases each block’s access time due to reduced capacitance, and enables pipelining of requests to the same bank. Multistage pipelining of the core or array hides precharge allowing it to occur simultaneously with input-signal latching and data transfer to the output latch. Figure 3.16 shows a direct comparison between a DDR SDRAM and a DDR FCRAM where all reads are directed at the same bank. The FCRAM advantage is in that the pipelining of the DRAM array allows accesses to the same bank to be pipelined in much the same way that the synchronous interface of SDRAM allows accesses to unique banks to be pipelined. FCDRAM is currently sampling in 64MBit quantities, utilizing the JEDEC standard DDR SDRAM interface, but is hampered by a significant price premium based upon the die area overhead of this technique [Fujitsu00]. Fujitsu is currently sampling FCDRAM devices which utilize both SDR and DDR SDRAM interfaces, additionally low-power devices targeted at the notebook design space are available.

The FCDRAM core enhancements could be applied to most DRAM architectures. The controller would have to be aware that it was communicating with FCDRAM devices in order to utilize the shorter timing parameters and unique access profile. This technique would most certainly increase the area of the DRAM array, however Fujitsu has not revealed the extent of this increase in area.

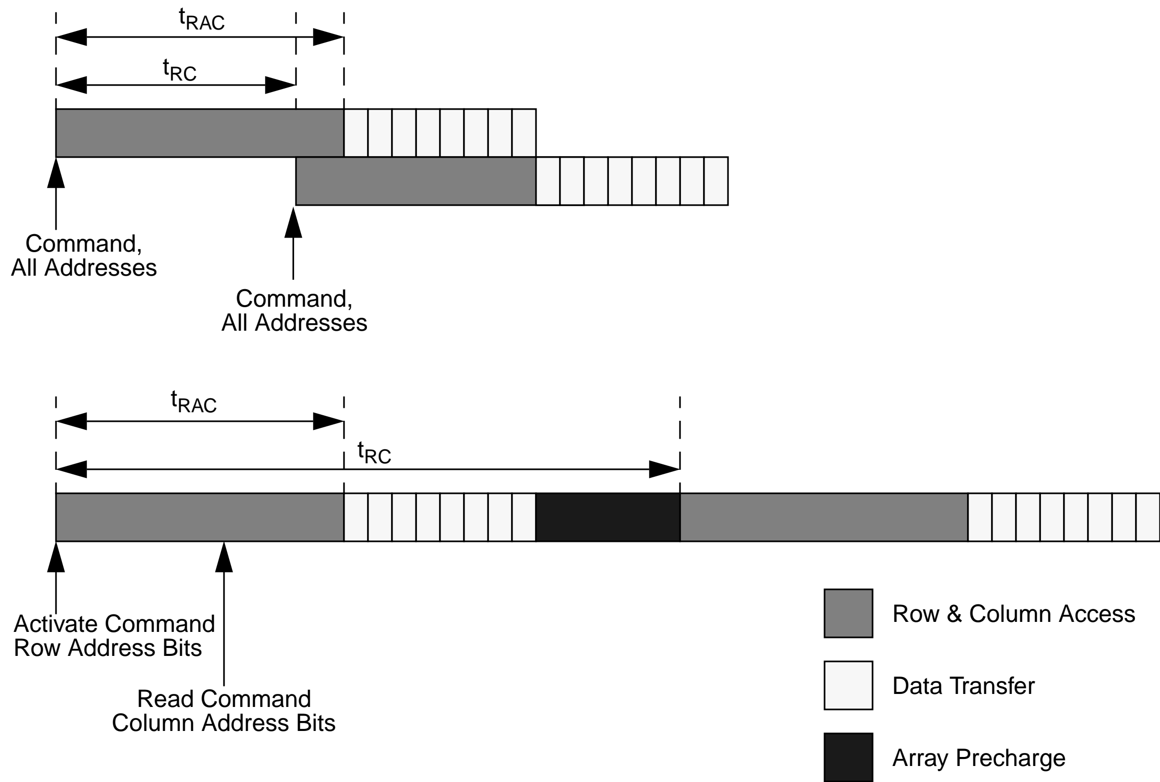


Figure 3.16: FCRAM Timing
This diagram shows how the FCRAM core enhancement decreases cycle times for sequential accesses to unique rows in the same bank [Fujitsu99]. This is an abstract timing diagram showing no clock signal because this technique has been applied to devices complying to more than one interface specification.

3.3.11 Multibanked DRAM (MDRAM)

The Multibanked architecture proposed by Mosys in [Mosys94] has never drawn significant acceptance, but it is an interesting study. The fundamentals of this architecture have placed Mosys in the highly specialized graphics specific DRAM design space. Their further architecture proposals, such as the “1T-SRAM” focus on using DRAM structures in an embedded application with emphasis on minimal latency at the cost of increased DRAM area[Mosys00].

While the Multibanked architecture has no explicit SRAM caching structures, it has a significant quantity of effective on-chip caching, when using an open-page policy, due to the large number of sense-amps per DRAM bit. The terminology Multibanked

DRAM is somewhat misleading in that many existing and new design DRAMs contain multiple banks, but the Mosys products are actually named Multibanked. Multibanked DRAM is an architecture built upon the performance advantages of many small banks, which can increase concurrency as well as device cost. One of the largest claims by Mosys of the Multibanked DRAM architecture is that this architecture has better utilization of the bus between the processing element and the memory array than their competitors.

Most DRAM devices, especially as more bits are placed onto a single DRAM chip, contain multiple banks. This allows the bit lines to be small, and have a low capacitance such that the charge stored in a single bit cell is sufficient to generate an observable change at the bit line sense amp. Some SDRAM have dual bank accessing, allowing independent access to the two individual bank. Rambus DRAM (RDRAM) have either dual bank accessing, 16Mbit, or quad bank accessing, 64Mbit. [Wilson95] However, MDRAM takes this even further, to quote their thesis, *to gain the real benefit of a multibank architecture there must be substantially more than two banks in a memory system*. [Mosys94] The Multibanked DRAM architecture is set up so that each bank is 32Kbyte in size, and the number of banks in the memory system is determined by the size of the memory system. This generates a highly interleaved address space, with the intent of increasing the utilizable parallelism. These banks of DRAM memory are connected together internally, in places where on-chip connections are possible, by a high speed bank-to-pin interface. The external memory interface is a synchronous interface, with a 16 bit Data bus, and transitioning on dual edges of the clock. [Mosys94]

The architecture of the Multibanked DRAM provides for two buses, of identical configuration, but different speeds, as shown in Figure 3.17. This architecture is relatively flexible, allowing for a variable number of 32 Kilobyte banks per MDRAM die. All of the MDRAM banks on a single die share a common internal bus, but each of the banks function as independent memory entities. The internal bus is identical to the external bus in all but speed, where the external bus is slower. The internal bus between the numerous banks is self-timed and designed such that it can exceed bandwidths of 800 MBytes/sec. The number of MDRAM die which can be attached to the external bus is also variable allowing for easy upgrading of a system. The MDRAM interface is specified with an I/O

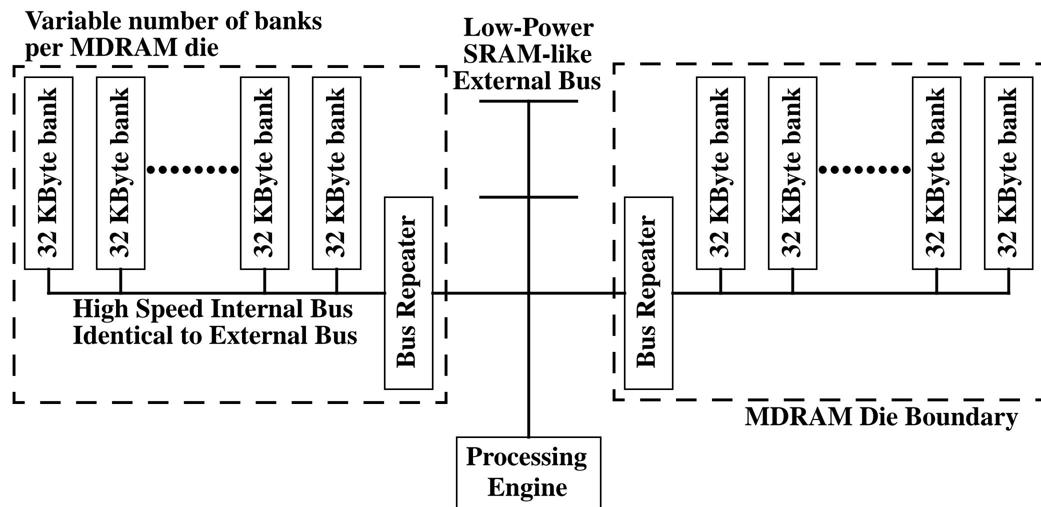


Figure 3.17: Mosys Multibanked DRAM Architecture Block Diagram

The multibanked architecture is targeted at exploiting the highest level of concurrency in the memory system. The large number of banks minimize bank conflict.

frequency of 166 MHz with a reported net bandwidth of 490 MBytes/s at 74% bus efficiency. [Mosys94]

Multibanked DRAM, using an open-page policy, has a large effective on-chip cache because within each bank of the memory system, the most recently accessed 128 Byte row or page is kept in the sense-amps for quick access. This being the case, the sense-amps of each individual bank effectively act as a cache. While this is true of any DRAM operating with an open-page controller policy, MDRAM have a larger number of sense-amp bits per DRAM bits than most other architectures because of the large number of small DRAM banks. With there being 128 bytes of sense amps for each 32 KBytes of memory, this amount of cache is reported by Mosys to achieve an approximate 91% hit-rate.[Mosys94]

3.4 DRAM with On-Chip Cache Capabilities

It seems intuitive given the configuration of DRAM, with a wide memory array accessed in parallel, that to place a cache on the DRAM chip would help boost performance given the high locality of reference in data streams. Many vendors have realized this, and a number of configurations of DRAM with on-chip cache are becoming available, under a variety of names, each with a different architecture.

The factors contributing to the convenience of DRAM with on-chip cache are that in the DRAM memory array, a large number of bits, all in the same row, are all accessed at once. This access is performed using the row portion of the address (signified by the $\overline{\text{RAS}}$ signal), but very few of the signals accessed during this cycle are required during this cycle. Typically a DRAM will have 1, 4, 8 or 16 data I/O pins allowing for this many signals to be driven off the DRAM die per access cycle. The same access cycle will by design extract 1024, 2048 or 4096 bits from the memory array into the sense amps. In a conventional DRAM using a close-page-autoprecharge policy, the remaining data elements which have been extracted, but not utilized are lost. Even in the case of DRAM using an open-page policy, those bits can only be maintained until the next refresh cycle. In each of the new DRAM with on-chip cache configurations, the method used to reduce the latency for the access is to keep these data elements which have been extracted from the array in a localized cache such that subsequent accesses which address the same row in the memory array(s) need not wait the full access time, only the time required to index into the line cache.

In addition to increasing performance of DRAM memory systems these DRAMs with on-chip caches often simplify the design of the system level of a microprocessor system. Some DRAMs with on-chip cache are intended to be used in memory system configurations where previous to the existence of these circuit configurations, a level of cache would be required in addition to the DRAM. Utilizing the DRAM with on-chip cache is likely to reduce the need for intermediate cache chips, as well as cache controllers, and the design tasks associated with a multi-leveled memory system. In this way, the use of DRAM with on-chip cache not only increases system performance, but quite often simplifies system design.

The architecture used, the size and configuration of the memory array line caches, and how various vendors integrate their solution into a new DRAM architecture are the primary differences between the following DRAM with in-cache configurations.

Most of the cache enhancements discussed herein can be applied to devices mated with a range of interfaces. In some cases, the interface is fully specified (CDRAM) and in others the interface requires some expand ability (VC) — however adding SRAM cache to

the device is a core enhancement, and to some degree the core and the interface are two decoupled implementations.

3.4.1 Cache DRAM (CDRAM)

Mitsubishi chose an unadorned name for their new DRAM with in-cache capabilities. The chip is a DRAM which has a small SRAM cache on the die in addition to the DRAM core. The SRAM cache, as has been discussed in the general overview for all DRAMs with on-chip capabilities, is able to make use of a very wide bus between the DRAM core(s) and the SRAM cache(s).[Ng92] In this specific architecture the SRAM and DRAM are integrated with each other using four structures, two read buffers, and two write buffers.

The M5M4V4169TP from Mitsubishi is a 4 Mbit Cache DRAM offering. The DRAM array for this chip is organized as 256K x 16 bits, with the addition of 16Kbits of on chip SRAM cache. The interface of the CDRAM products contains a synchronous clock, much like synchronous DRAM. The interface of the CDRAM is more complex than the traditional DRAM interface, including pins for addressing the on-chip cache independently from the DRAM. Despite these two distinct address buses, all data must be passed through the SRAM or the read buffer number one structure lines to reach the single set of data output pins. The block diagram for the Cache DRAM architecture is given in Figure 3.18. [Mitsubishi95b] As can be seen from this diagram, the output (DQ[0-15]) can only be provided from the Read Buffer (RB1) and the SRAM. This means that if the data which is being requested resides in the DRAM block when it is requested that it must first be transferred through the RB2 to the RB1 from which it can be sent to the data output pins. This increases the access time of the DRAM as it is viewed from the outside of the Cache DRAM package, but automatically promotes the most recent accesses to the SRAM cache. The access times for this product, again the highest performance four MByte CDRAM, is a 15 nS cycle on the SRAM, and a 120 nS cycle on the DRAM. This DRAM cycle is significantly longer than the cycle time of even most conventional DRAM being produced today. The SRAM cycle of 15 nS is hardly the highest performance SRAM announced, however it is much closer than the DRAM core. The interface logic in this Cache DRAM architecture appears, from these numbers, to hinder the latency of the

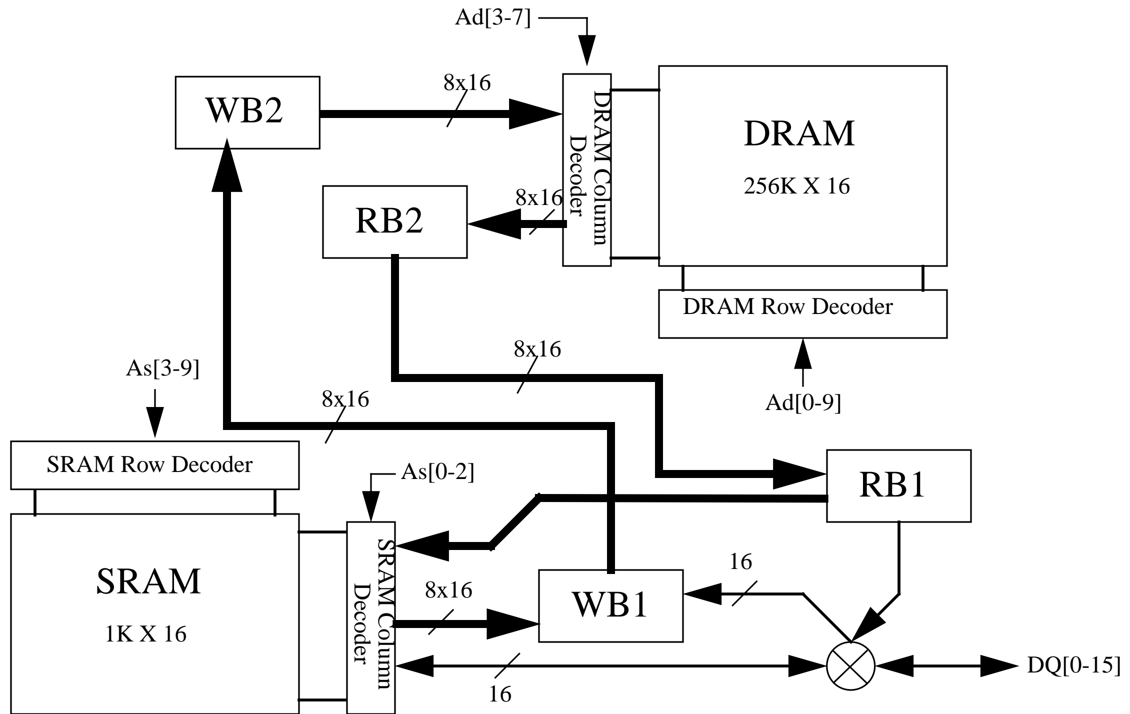


Figure 3.18: M5M4V4169 Cache DRAM Block Diagram

The CDRAM architecture is significantly more complex than other architectures, as it attempts to integrate what are typically system-level functions onto the DRAM device.

memory cores. How the architecture performs in actual applications has yet to be determined.

The separate cache on the DRAM chip does significantly increase die area over the size of an equivalent size DRAM alone. This means that there is no way to avoid pricing a CDRAM higher than an equivalent size DRAM. Regardless, the CDRAM architecture is still worth study as a comparison between DRAM and more non-conventional DRAM with on-chip cache.[Bursky95]

As of yet, no second sources have been announced for Cache DRAM. This could negatively impact the demand for CDRAM due to the hesitancy of many manufacturers to avoid single source products.

3.4.2 Enhanced DRAM (EDRAM & ESDRAM)

Ramtron has been a proponent of cache-enhanced DRAM for a number of years. A wholly owned subsidiary of Ramtron, Enhanced Memory Systems (EMS) is currently

driving the initiative for one of the more promising cache enhanced DRAM architectures. Unfortunately, the name Enhanced DRAM is somewhat confusing, because it is not the only enhancement possible for DRAM devices, however this terminology has become accepted. They have specified and manufacturing cache enhanced DRAM which comply to both asynchronous and synchronous interfaces. All Enhanced DRAM which follow the Ramtron architecture have a single cache line for each bank in the device, located between the sense-amps and the column decoder. This technique minimizes the area impact of the cache enhancement while allowing: 1) accesses to a previously loaded cache line to incur only the column access (CL) latency; 2) precharge to take place without disturbing the contents of the cache; 3) refresh to occur without disturbing the contents of the cache; and 4) a write to take place through the sense-amps without disturbing the contents of the cache. Cache enhanced devices using this architecture have been proposed and in some cases produced for a number of interface specifications.

The Enhanced asynchronous DRAM architecture has a single row cache line for each of the memory arrays. The 4Mbit DM2223/2233 EDRAM has four memory arrays, each of which has their own, in the terminology of Ramtron, row register. This architecture is given in the block diagram shown in Figure 3.19. This means that if two consecutive accesses map into the same memory array, as determined by bits 8 and 9 of the column address, then the first access will no longer be in the cache following the second access. The on-chip cache could then be said to be a direct mapped cache, and is referred to by the manufacturer as a “row cache” [Ramtron94].

Figure 3.20 shows an architectural diagram for the synchronous version of these cache enhanced devices. Currently available in 64Mbit devices, ESDRAM devices are a high-performance DRAM which, at this point, have very little market share, most probably because of their increased price over standard SDRAM. A number of chipsets, from Intel and VIA technologies, have been produced which will interface to both 100Mhz and 133Mhz SDR ESDRAM. The no-write-transfer ability of the ESDRAM specification allows a write to bypass the cache, storing into the DRAM array (via the sense-amps) without affecting the state of the cache. This ability is useful if the controller is able to identify a write-only stream or page, in these cases, pollution of the cache can be

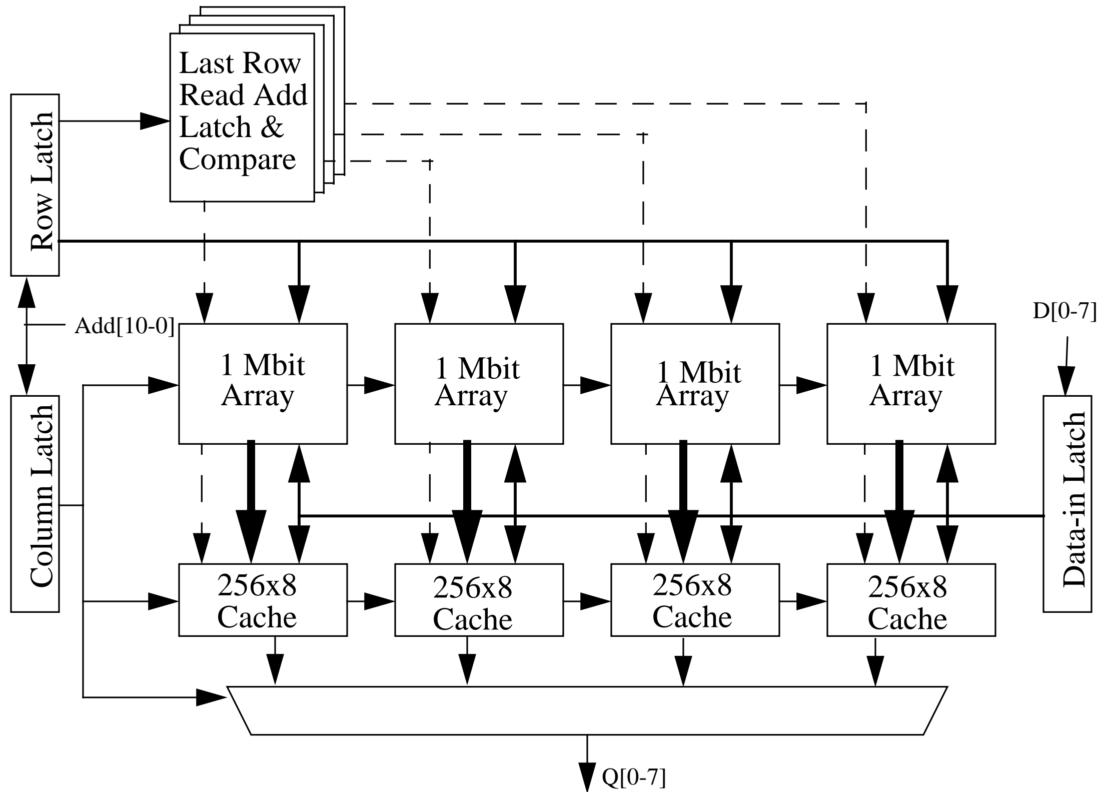


Figure 3.19: Asynchronous Enhanced DRAM Architecture

The Diagram above is for a 4Mbit device, ESDRAM devices are currently available in densities up to 64Mbit
The caches and arrays scale as with a conventional SDRAM

avoided. The ESDRAM devices have circuitry to guarantee that if this functionality is used, the cache never contains stale data.

It has been proposed that this architecture be extended to the DDR2 interface. JEDEC, and the low-latency DRAM working group, are currently considering this option, but it seems likely that there will be either an optional or required addition of the enhanced memory caching architecture to this next generation DDR SDRAM [Peters00]. The die area overhead for the Enhanced DRAM cache enhancement has been estimated at 1.4% for a 256 Mbit 4 bank DDR2 device [Peters00].

3.4.3 Virtual Channel DRAM (VC DRAM)

NEC has proposed a cache enhanced SDRAM variant based upon their Virtual Channel (VC) architecture. The intent of the VC architecture is to reduce the average latency of a DRAM access via two mechanisms: (1) to use on-chip SRAM cache to reduce

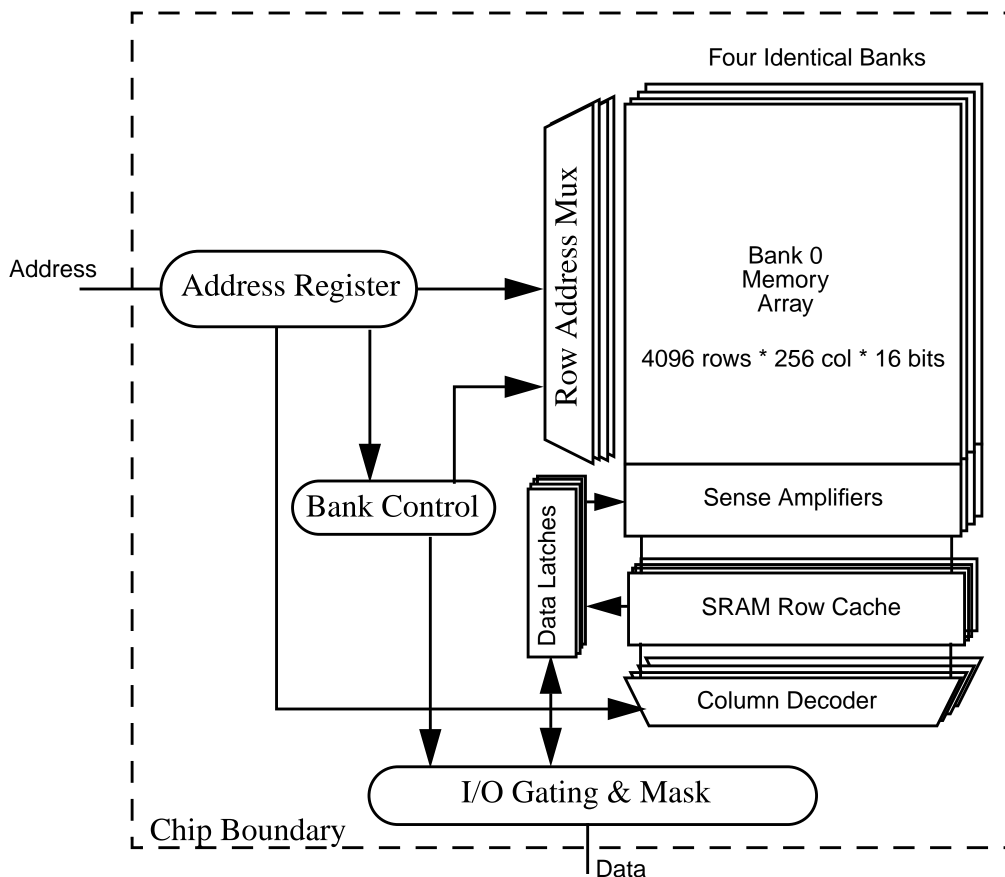


Figure 3.20: Synchronous Enhanced DRAM Architecture

The 64Mbit ESDRAM Architecture adds a direct mapped Cache on the DRAM device, with an SRAM row attached to each set of sense-amps. The no-write transfer ability requires that the data latches be able to write into the sense-amplifiers without disturbing the cache.

the number of accesses to the DRAM array; and (2) to organize this SRAM cache in such a manner that it allows for multiple open channels to the same bank, reducing latency in cases where you have two or more access streams alternating between different rows in the same bank [NEC99]. The VC implementation may be unique for each device, many of which have yet to be precisely determined, with the number of banks being 2 or 4, the number of cache lines 8, 16 or 32, the size of the cache lines almost certainly 1/4 the DRAM row size, and the associativity of the cache lines being anywhere from fully-associative through 4-way set associative. The VC architecture refers to each of these cache lines as a “Virtual Channel” and relies upon the memory controller to manage the allocation, placement, and write-back. This allows the controller to maintain or write-back

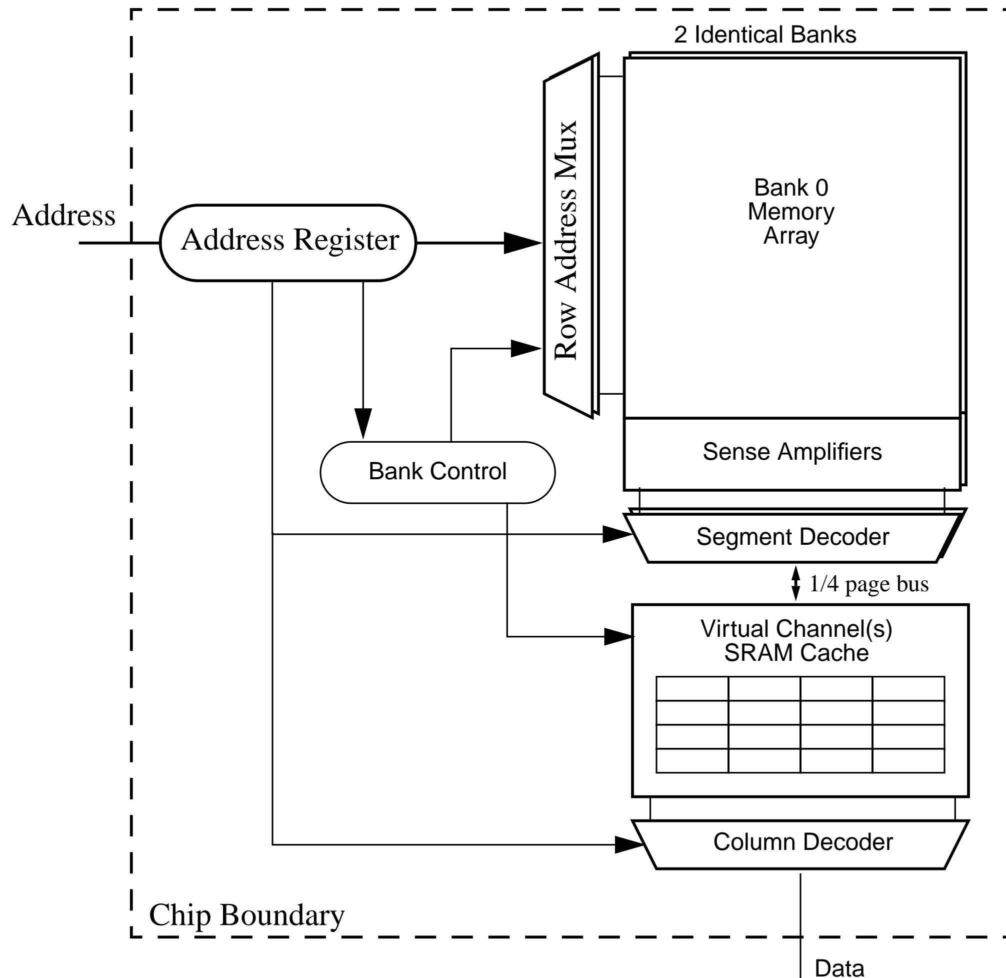


Figure 3.21: Virtual Channel Architecture

This diagram shows a two-bank implementation of a 128 Mbit VCDRAM device. Accesses to a Virtual Channel device must go through two logically separate stages, prefetch from array to channel, then read from channel to interface.

dirty channels, or allocate the channels based on any policy from an LRU algorithm to a bus-master oriented algorithm. One aspect of the VC architecture which can negatively impact performance is that if all channels are dirty, and a new channel must be written back prior to the device being able service an access, the access must wait until this writeback occurs. This can as much as double the observed access latency, however with appropriate channel writeback controller policies, this penalty due to all channels being dirty can be almost always avoided.

The Virtual Channel architecture can be applied to devices targeted at any interface. Currently there are Virtual Channel enhanced SDR SDRAM devices available, and proposals have been made to produce VC enhanced DDR2 interface devices. One issue involved in this is that the Virtual Channel architecture requires additional commands in the interface protocol for channel allocation, restore, and other architecture specific functions. The area impact of adding the Virtual Channel enhancements to a DRAM device are hard to determine, but has been appraised by NEC at 4.3 to 5.5 percent depending upon the implementation [Yabu99].

3.5 Controller Policies

In early asynchronous DRAM, the controller policy was simply refresh policy. As DRAM have added capacity and features, the state contained in the memory devices, as well as the memory controllers, has become more significant. DRAM have not, since the advent of page-mode DRAM, been truly “random access”. As devices become more complex with more internal state they continue to exhibit higher variation in access latency. With the variety of possible states for a modern DRAM device, or bank, the controllers goal is to manage the devices to minimize the latency observed by accesses. The method used is to pick a management strategy, or policy, which has shown to minimize latency for a given system configuration. Unless the controller has specialized knowledge about the access stream, this approach is the most likely to provide the best performance. Controller policies have a significant ability to affect system performance for modern DRAM devices.The remainder of this section will discuss the variety of controller policies available.

The memory, or DRAM, controller is located on the North-Bridge chipset device in modern desktop system architectures. This location provides a centralized point through which all devices that access DRAM, processor, graphics engine and I/O subsystem, may access the primary memory space. However, this location also has negative impacts upon performance because each access must arbitrate for two buses (front-side and DRAM), both of which require a delay for synchronization to a lower clock, as well as traversing the controller and performing the access in the DRAM devices.

It has been proposed that some of this latency could be removed by placing the memory controller on the die with a bus separate from the processor front-side bus. This approach is being taken with the Intel Timna [Lammers00] as well as the Alpha 21364 [Gwennap98]. This is enabled by the lower signal count of Rambus architectures.

The memory controller is usually responsible for device refresh, as well as providing an interface between the DRAM an the remainder of the system. In performing this function, there is a variety of policies which a memory controller may utilize dependent upon system parameters and application. These include, but are not limited to, maintaining the sense-amps in an open or closed state, address remapping, prefetching, access coalescing, access re-ordering, cache allocation (VC) and cache-bypass (EMS - no write-transfer). The evaluation of the entirety of policy choices is beyond the scope of this thesis, but these policy decisions can have significant impact upon system performance [Carter99].

The first DRAM to support any level of controller policy flexibility was Fast-Page Mode (FPM) DRAM. Use of the fast access to the page was enabled by an Open-Page policy for the DRAM devices. As DRAM devices became more advanced they enabled more complex controller policies to manage functionality.

Table 3.1. shows some of the controller policies discussed in the following sections. The choice of controller policy is one which is application specific. The different policies, as shown in Table 3.1 each have advantages for specific circumstances. Choosing a single policy which is going to provide the best performance uniformly across all applications is typically impossible.

3.5.1 Refresh Policy

Because of the Dynamic nature of DRAM, refresh is essential. Devices typically specify a refresh period, such as 64mS during which all rows must be refreshed. Thankfully, most interfaces include a function to refresh a row of ALL banks concurrently. Even when using this potential, there are decisions to be made. All N rows in the device can be refreshed sequentially, making the DRAM system unavailable for a time period of (N*tRP) once in every 64mS period. Alternatively, one row can be refreshed every 64mS/N. This time-interspersed refresh policy allows for less average latency than the all-at-

Table 3.1: Controller Policy Advantages & Disadvantages

Policy	Advantage	Disadvantage
Open-Page	if the next bank request is in the same page Latency = CL	if the next bank request is in a unique page Latency = tRP + tRCD + CL
Close-Page-Autoprecharge	if the next bank request is in a unique page Latency = tRCD+CL	if the next bank request is in the same page Latency = tRCD + CL
Address Remapping	Reduces the fraction of adjacent accesses which map to the unique rows in the same bank	Complexity and possibly additional latency in the memory controller
No Write Transfer (EMS cache-enhanced specific)	Reduces conflict misses in Direct mapped row caches	Difficult to determine whether a write should be cached

once refresh. In some studies, this simple choice of refresh period shows a 50% mean difference in overall benchmark execution time [Cuppu99].

Other refresh policies exist. If the controller contains a higher complexity state machine, it may choose to schedule refresh, and possibly prefetch, when the DRAM bus is idle, in anticipation of a stream of requests subsequent in the refresh period - thus reducing refresh/access collisions. Other refresh policies include oddities such as graphics applications where refresh is never explicitly scheduled because it is known that each page will be read each frame update, and the frame update period is less than the refresh period.

3.5.2 Close Page Autoprecharge

The Close-Page-Autoprecharge controller policy is the simplest for the controller to implement, and requires the least amount of state in the controller device. After each access, the controller closes the page of the bank being accessed, and initiates a precharge on that bank. Whenever an access is initiated, it must perform both a row access or $\overline{RAS}$ and a column access or $\overline{CAS}$ because all pages in the memory system are closed. This

means that for most accesses the latency will be (tRCD + CL). Occasional access will not see this latency because of the effects of refresh, or because the access in question closely followed another access directed at the same bank, in which case that bank may still be precharging.

In applications where there is a significant amount of cache memory higher in the memory hierarchy, either due to a large cache associated with a single processor, or multiple caches associated with an SMP, the close-page-autoprecharge policy has been shown to provide better performance than open-page [Coteus00]. Close-page-autoprecharge could be viewed as the conservative controller policy, as its latency is fairly deterministic relative to the other options.

3.5.3 Open Page

The open-page controller policy implies that after an access to a DRAM page, that page is maintained "open" in the sense-amps of the bank. The intent when choosing this policy for the controller is that subsequent requests to the bank will be directed at the same page, eliminating the necessity of a row access or $\overline{\text{RAS}}$. In this case, the access will only incur the $\overline{\text{CAS}}$ latency or CL. However, if the subsequent request is directed at another, non-open, page within the bank, then prior to performing a row-access to the desired page, the bank must be precharged. In this case, the access latency is (tRP + tRCD + CL) or the worst case latency. The open-page policy is less deterministic in latency, and the performance of this controller policy is highly dependent upon the page-hit rate of the application. This is determined by the amount of cache in the higher levels of the memory system and the access behavior of the application. In a case where the access behavior is strictly linear streaming data or has good locality, the open-page policy does very well. However, in cases where the accesses are essentially random or show a low degree of locality, such as in a transaction-processing application, the average latency for an open-page controller policy can be higher than that of the close-page-autoprecharge policy [Davis00a].

The open-page policy requires additional state in the memory controller beyond that required by a close-page-autoprecharge policy. To enable the controller to know what pages are currently open, thus enabling the controller to do a "page-mode" access, the

controller must have tags associated with each row of sense-amps in the memory system. This means one set of tags per bank. For example, if you have an desktop environment where each device may have 4 banks, each DIMM may have two sets of devices, and a motherboard has 4 DIMM slots, the controller must hold tags for 32 banks. If any of these limits are exceeded, such as a new DIMM configuration with more than 4 banks, the memory controller can not make use of page-mode accesses for those banks beyond its tag limit. Examine Section 3.5.6 to see how changing the architecture (in this case to a VC implementation) may increase the amount of state required in a controller.

3.5.4 Address Remapping (Hashing)

In a synchronous environment it is useful to re-map the processor address space into a DRAM address space with a modified layout. Performing the best re-mapping is dependent upon a number of parameters: the size of the DRAM pages, the number of banks in the system, the number of devices in the system, and the lowest level cache size and mapping policy [Lin99]. The intent of this remapping is to locate banks that are likely to be accessed in close temporal proximity to independent banks in the DRAM address space to enable the highest amount of overlap between the accesses. Remapping of the address space can be accomplished by the DRAM controller [Rambus99]. When performing comparisons of DRAM architectures, it is important to consider the impact of re-mapping the address space on maximizing concurrency.

3.5.5 Enhanced Memory Systems Specific Controller Issues

The Enhanced Memory System modifications to the conventional DRAM, as described in Section 3.4.2, impact the controller design significantly less than other architectural changes. Controllers which are capable of managing devices using an open-page policy already have all of the tags and state required for determining row-cache hit status. The difference with an Enhanced device is that these tags need not be invalidated upon a refresh to the device.

One potentially significant change to the controller state machine is the possible support for use of the no-write-transfer mode. Use of this capability requires that the

controller somehow be able to differentiate between a write which should be done through the caches, and a write which should bypass the caches. This bypass should occur based on whether the write in question is to a stream or location which is write-only, or a stream or location which services both reads and writes. To make this determination, would require either a significant state machine in the controller, or use of hint bits by the compiler. Use of the no-write-transfer is optional in all Enhanced DRAM architectures, but may increase performance.

3.5.6 Virtual Channel Specific Controller Issues

The Virtual Channel architecture enhancements, regardless of the interface to which they are applied, place additional demands upon the controller. The level of state required by the controller is larger than that of most competitive architectures. This is because, in the interest of associativity, the Virtual Channel architecture has a large number of channels associative over a smaller set of banks, where competitive architectures have a single row-buffer or row-cache directly mapped to each bank. The controller must then maintain a larger number of tags (because of the larger number of cache lines) and the tags themselves must be larger (because of the higher associativity). The associativity provided by virtual channels certainly allows increases in performance, but as with many changes to DRAM architecture, they also increase the cost, in this case of the controller device.

In addition to state concerns, the Virtual Channel specification relies upon the controller to do the channel allocation and dirty channel writeback, so the state machine (and possibly LRU tags) to handle allocation and writeback must be on the controller device. This adds complexity to the DRAM controller. This additional complexity in the controller will equate to additional area required for the a finite state machine(s) to implement these new functions. Despite this additional complexity required by a chipset to interface to Virtual Channel devices, a number of manufactures, ALi, SiS, and VIA technologies have all produced, or announced chipsets to interface to this specification.

The feature of the Virtual Channel architecture which allows the DRAM controller to manage allocation of channels may impact the area and complexity of the controller, but it allows wide breadth of controller policies, not all of which need be identified at this

time. One possible allocation policy proposed by Virtual Channel advocates divides device the channels between possible bus masters, such as distributing sixteen channels as follows: 12 for the processor, 2 for the graphics device, and 2 for I/O systems. This policy may minimize the conflict misses observed by I/O and graphics devices. Because these allocation decisions are made by the controller, they do not impact the device architecture, and may be changed in future controller generations. In addition to the controller being required to manage allocation, since the cache on Virtual Channel devices is associative, the controller must also manage cache line replacement. Lastly, because dirty channels can increase the latency when an access does not have an available channel, the controller must manage the writeback of dirty channels into the DRAM array. The Virtual Channel architecture allows for lightweight writeback of dirty channels such that these writeback commands should not affect bus utilization, and the case where all channels are dirty causing an access to incur additional latency should be a highly rare occurrence. These requirements placed upon the controller by the Virtual Channel architecture allow for a significantly larger variety of controller policy choices than competing architectures.

3.6 Primary Memory in the Future

DRAM has been the primary memory of choice since its introduction in 1971. However, there is no fundamental reason another technology could not surpass DRAM in the price/performance niche of primary memory. Magnetic RAM (MRAM) is currently being examined by a number of manufacturers, including Motorola. This technology has the advantage of being non-volatile, like core memory, and is being enabled by the research into micro-mechanical structures [SemiBN00]. At the moment however, MRAM remains a research topic in itself. If Magnetic RAM becomes commercially viable, how to integrate this non-volatile, differently organized technology into the memory system will introduce a variety of challenges. Another memory technology being examined is optical storage in a two or three dimensional array. This technology, enabled by optical networking research, has potential advantages of allowing people to transport their entire system memory image, containing massive amounts of information in a small three dimensional carrier [C-3D00]. Primary memory will remain an element of computers so

long as the conventional VonNeumann model persists, however the technology implementing this primary memory may change in the future, as it has in the past. This being said, within the near-term (10 years), updated DRAM technologies will remain the technologies of choice for primary memory.

Chapter 4
Methodology

In examining memory systems and their performance, a number of evaluation techniques are used. The first approach is a comparison of existing and available hardware with regard to execution time, or whatever parameters are the basis for comparison. This is not always an option, most notably when the devices to be investigated are pre-production or not available. In this case there remains two approaches. Trace driven simulation, relies upon a trace of execution to exercise a simulation of the systems being evaluated. Execution driven simulation emulates execution of an application binary to simulate the entire system being evaluated. These two simulation based techniques are discussed further in Section 4.2 — Trace Driven and Section 4.3 — Execution Driven respectively.

There are a variety of possible methods for performing memory system analysis, some of which overcome the excessive time involved in simulating the upper layers of the machine when doing studies on low level memory systems. These methods are presented as possible alternatives to instruction level tracing. Each of the possible alternatives has associated advantages and disadvantages. The characteristics of three methods are presented for the reader to draw his or her own conclusions.

4.1 Introduction

Depending upon the context of an investigation and the results being examined, different simulation methodologies may be best suited to the goals. In the simulations discussed here both trace driven and execution driven were used. Since each of these approaches has advantages, it is best to make use of both in examining DRAM structures. Traces can be used to debug simulation models - particularly synthetic traces whose effect upon the DRAM system can be calculated analytically. Traces are also useful for

calculating the hit rates on any SRAM structures the DRAM device may hold and for determining the upper limit of the DRAMs bandwidth. Subsequently, execution driven simulations can be done for more focussed studies that require a higher degree of confidence in the results.

The objective of simulation, regardless of technique is to model the memory architecture of the computer system being simulated. Figure 4.1 shows a common

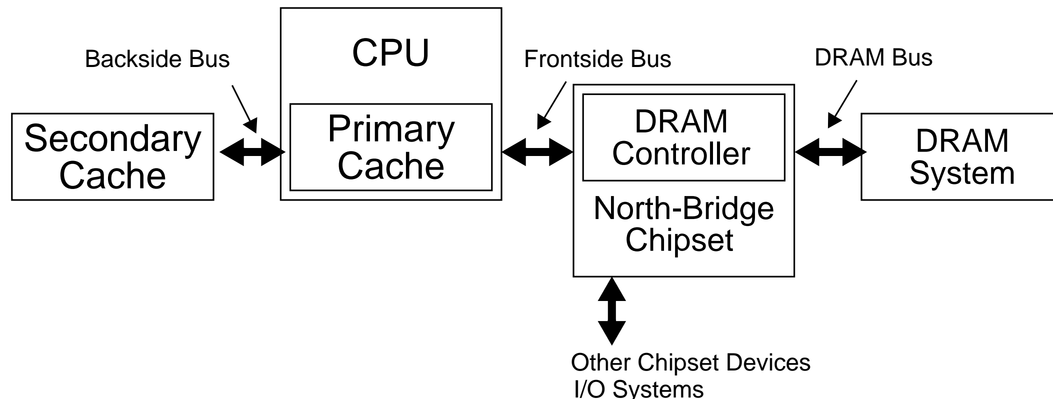


Figure 4.1: Memory System Architecture

This is a conventional memory system model for many desktop computers produced. As such, this is a target for the simulations to be performed on DRAM performance.

configuration for the memory system a desktop computer. This is the system which we would like to model in both the execution and the trace driven simulations. Not all computer systems precisely follow this model, some contain all cache on the CPU die, some have multiple microprocessors (or CPUs), a limited few have multiple DRAM controllers, and as was mentioned in referencing CRAY, some have no DRAM at all. However, this architecture is fairly common, and is the target for both of our simulation methodologies.

The two techniques can share some of the source code if the design of the simulation environment is sufficiently modular. In the simulations described in Chapter 6 the DRAM simulation models are common to both trace driven and execution driven simulations. The source of the accesses being provided to those DRAM models is a simple file reader and re-formatting engine in the case of trace driven simulations, while it is a complex microprocessor model in the case of the execution driven simulations.

4.2 Trace Driven

Trace driven simulation is performed by gathering a trace of system activity, at the instruction level, or in the case of DRAM investigations at the memory controller-DRAM interface level; then at some later point that trace is used to exercise a simulation engine modeling the target system. One primary advantage of trace driven simulation is speed. If the trace contains strictly DRAM activity, the simulation of the memory system alone can be done very rapidly. The disadvantage of trace driven simulation is that the parameters of the simulation: processor speed, bus speed, bus bandwidth, number of DRAM devices, and so forth, can not be varied without impacting the accuracy of the trace. Another potential disadvantage is the lack of accurate timing information. The worst case would be to have no timing information in the trace, the best situation being timestamps based on a DRAM system similar to that of the target system. Absence of timestamps during simulation has the consequence that the DRAM access stream is compacted in time to the minimal amount of time required to perform all accesses. However, the presence of timestamps does not necessarily guarantee great accuracy, even when using many of the same system parameters. If the DRAM latency has significantly changed from gathered to target systems, the interspatial access timings may be significantly altered, again affecting the accuracy of the DRAM trace.

4.2.1 Instruction Level Tracing

The concept behind instruction level trace driven simulation is that a software engine can precisely model a complete microprocessor system. The parameters to the software engine can be more rapidly changed than equivalent changes in hardware simulation, or actual hardware. The observed changes in performance at the software simulation level can then be used to refine subsequent software designs in a loop oriented design cycle, until the system design achieves the required goals of the specification.

Instruction level trace driven simulation, including analysis down to the DRAM level can accurately model microprocessor and memory system activity to the cycle, provided that the simulator is written with that level of precision. In ILTDS the complete state of the microprocessor, including the contents of all caches and memory structures, is

maintained, and the instructions which are executed by the microprocessor are modeled to effect changes in this stored state. Instruction level tracing is a slow process, as all operations which are normally performed in fast hardware, such as dependence checking or determining whether an item is present in a cache, must now be performed with slow software routines in the machine performing the simulation.

The inefficiencies of instruction level trace driven simulation for other types of architectural analysis has been previously discussed [Noonburg94]. Instruction level trace driven simulation varies in accuracy, primarily depending upon the precision of the simulation engine. However for the application proposed herein, instruction level trace driven simulation has a variety of problems. The most obvious problem with ILTDS for investigation of low level memory systems (i.e. DRAM level) is that in order to get a significant number of DRAM references, an instruction level trace driven simulator with a complete memory hierarchy must execute for prohibitively long periods of time. This is due to the fact that the vast majority of the CPU time is spent simulating the effects of each instruction upon the microprocessor and upper levels of the memory hierarchy and rarely getting to the level under investigation. When this is compounded with the fact that, regardless of the model used for simulation, the DRAM level hierarchy must be simulated a large number of times, once for each set of benchmarks and once for each memory hierarchy configuration, the excessive time required increases dramatically limiting the number of configurations which can be examined.

One positive side of using instruction level tracing is that the complete machine state is modeled. Therefor there is no loss of information provided that the simulation engine is accurate. If the simulator is written properly, the performance of a microprocessor to be designed should be identical to the performance characteristics generated by the simulator. This exact accuracy is not completely possible in those architectures which have inherent non-deterministic transfer times, such as those with memory being retrieved across a shared medium, i.e. a shared memory bus, multiprocessor environment, PCI bus, or Ethernet.

The negative side of using instruction level tracing, if it has not already been overemphasized, is that simulating a microprocessor of any complexity with complete cache system state, and running an application of any complexity, takes excessive amounts

of time. In cases of simulating microprocessor systems with large caching systems, instruction level tracing can also require huge amounts of memory on the machine performing the simulation. If the factors of the operating system (O/S) for multi-tasking and memory paging come into play, then the requirements for both time and memory become notably larger. It is unfortunately the case that in investigating DRAM systems, it is exactly these situations requiring larger amounts of both time and memory which are of most interest. When studying DRAM level memory systems it is commonplace, if not required, that the simulation go to the level of O/S and interactions of multiple application running under a multi-tasking environment because the affects of paging and context switching have a large impact upon the DRAM access patterns. If it were feasible to simulate these applications using ILTDS, the amount of compute time required to investigate each of the multiple possible memory hierarchies via simulation may very well exceed the amount of time required (or scheduled) for the memory system design. This characteristic makes ILTDS very unattractive for the enhanced DRAM study proposed in this thesis.

4.2.2 DRAM Bus Level Tracing

One option for eliminating the excessive amount of time required in doing DRAM level simulations would be to utilize traces from the architected bus directly above the memory system of interest. This process involves taking the instruction level traces for the benchmarks to be studied, running these instruction level traces through an engine which simulates the microprocessor, the upper level memory systems and caches in order to produce traces which contain memory references at the lower, in this case DRAM level. The memory references in these generated traces would be only those references which escape the lowest level of the memory system encompassed in the simulation engine, be that L1, L2 or possibly even L3 cache. This memory reference stream could then be used as the input to a simulator which performs memory system analysis solely below the bus upon which the stream is based. There are advantages and disadvantages to this approach for DRAM level memory system analysis.

The DRAM bus level trace approach allows the generation of a single DRAM reference trace, which then can be used repeatedly in the simulation of a variety of DRAM

level configurations. Figure 4.2 shows a conceptual diagram of how the trace driven

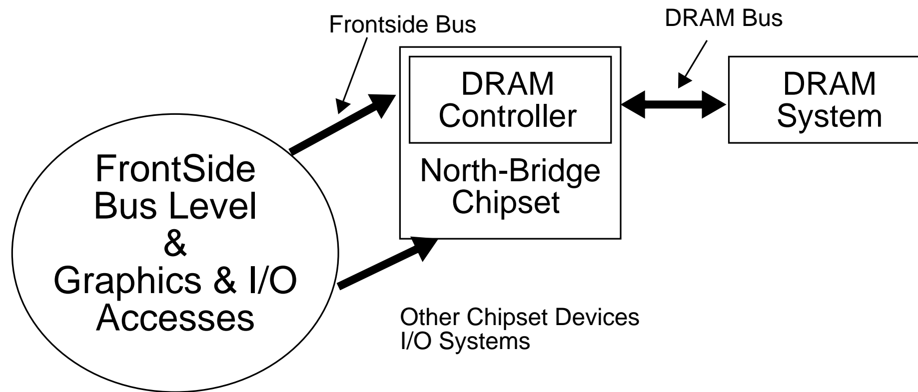


Figure 4.2: DRAM Bus level Trace Driven Simulation

The Trace Driven simulation environment models only latency through the controller and DRAM system for the entire access stream, the dependence between accesses and speculative activity are lost.

simulation operates, and what elements of the system are encoded into the trace. The higher levels of the memory system, including the microprocessor are either a real system from which the trace is being gathered, or only need to be simulated once at the time of trace generation. The CPU time required to perform these upper level simulations, in the case of a simulation based trace generation, can then be amortized across all of the DRAM level trace analyses which are performed. The DRAM bus trace allows the DRAM analysis to execute very rapidly because no CPU time need be expended modeling the microprocessor and cache, this has all been done ahead of time. The machine performing these final DRAM analyses can therefor concentrate solely upon the DRAM modeling.

In a microprocessor based system there exists a feedback path from the DRAM through the cache(s) to the microprocessor, whereby both the temporal locality and the occurrences of DRAM references change when the DRAM level response times change. This feedback is most obvious when considering a non-lockup implementation of a superscalar microprocessor with speculative execution. In this configuration, a long latency load could cause subsequent speculative loads to be initiated by the microprocessor, or the cache to become polluted with speculative data. Under the same assumptions, it is conceivable that if the initial load had been handled by a fast access cache, these speculative accesses need not have occurred, implying a different state for the machine being simulated. This means that the machine state is dependent upon to the

latency of all previously performed DRAM accesses. Unfortunately, this cannot be emulated in cases where the latency of DRAM references are unknown, such as when a DRAM level bus trace for use in multiple simulations is used as the model of the upper level system activity. This DRAM level bus trace necessarily and implicitly encodes a particular DRAM latency as well as cache and microprocessor configuration. Essentially, if the DRAM level approach is used the hope is that the impact of the DRAM configuration upon the upper level simulation and DRAM trace generation is minimal to the point of being “in the noise”.

The most apparent situation where the DRAM model affects the actual content of a DRAM trace (not the temporal locality, but content) is where speculative execution, including speculative prefetch cache structures as stream buffers, is concerned. The latency at which data is returned from DRAM could affect the DRAM level trace by allowing, or preventing speculative fetches, as described above. Even without speculation however, the latency of the DRAM can affect the distribution of memory references, implying that the feedback path from DRAM to the microprocessor is required to be maintained for accurate simulation.

The cache hierarchy present when a DRAM bus trace is collected or generated can also play a large role in the same manner as the DRAM themselves, since the cache(s) are in the feedback path described above. When generating a DRAM bus trace it is common to set the cache hierarchy to a fixed configuration and then study the various DRAM level proposals under the presumption of this fixed caching scheme. While it is possible to study a variety of unique DRAM structures with identical cache hierarchies using a fixed caching structure, the advent of the enhanced category of DRAM circuits which include on-chip caching make this strategy less attractive.

The summary with regard to static DRAM tracing then is that if it is acceptable to lose the exact behavior of the system due to the removal of the feedback path whereby DRAM latencies impact higher level structures, then this approach is acceptable. In the case of the investigation proposed herein it is presumed that it would be preferable to avoid this loss of accuracy. For this reason it is felt that another method for performing DRAM memory studies must be developed.

4.3 Execution Driven

Execution driven simulation is performed by taking a program, typically a compiled binary, and running that binary through an execution driven simulator such as SimpleScalar [Burger97]. Over the course of simulation: (1) this tool will generate all of the DRAM accesses required for program execution; (2) these DRAM accesses can be processed by a DRAM model; and (3) statistics can be produced by the model. The execution driven model is preferred for a number of reasons: The parameters to the simulation can be easily changed, and the DRAM access stream remains accurate. Hypothetical systems which have not been produced, and from which it would therefore be impossible to gather a trace, can be modeled. And, finally while it is not always possible to gather a trace for a application of interest, it is typically possible to acquire a binary for that application. A significant disadvantage of execution driven simulation is that this level of simulation requires more system elements be modeled and thus is very time consuming. One consequence of being dependent upon the execution driven simulator, is that most of these toolsets model the execution of a single binary on a single processor, ignoring OS interaction, multiprocess effects, and SMP configurations.

The execution driven model, as shown in Figure 4.3, "Execution Driven Simulation" takes a compiled application and simulates each instruction. Each instruction accesses memory, but a large fraction are serviced by the level one instruction cache. The memory hierarchy, as shown in Figure 1.1 serves as a filter reducing the number of requests. Only those accesses which miss all levels of the cache escape the SimpleScalar simulation environment to exercise the DRAM controller and device models.

Both of the simulation methodologies used for the studies presented in Chapter 6 are intended to model the memory system architecture given in Figure 4.1. The trace driven methodology is much faster to execute because of the reduced computational effort. However, the trace driven methodology is comparatively less accurate, abstracting away access dependencies and idle time of the DRAM bus. The execution driven methodology is slower to execute because it models more elements of the system, including instructions which generate no memory system activity. The execution driven simulation does not model the activity of bus masters other than the CPU executing the benchmark. Neither of

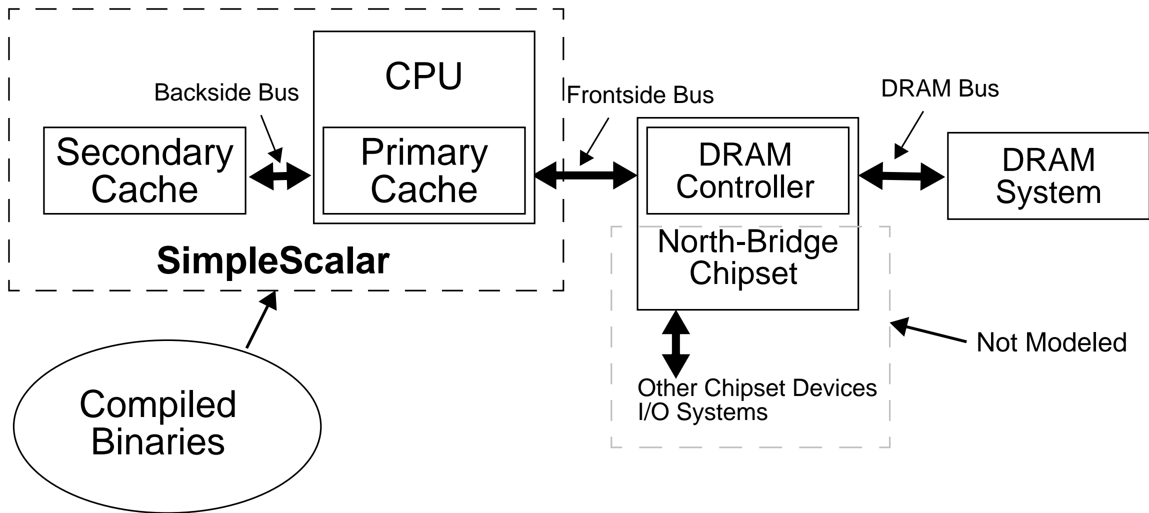


Figure 4.3: Execution Driven Simulation

The Execution Driven simulation models most elements of the computer system, including the dependence between accesses, only operating system, graphics, and I/O devices are unmodeled.

these simulation methodologies include the activity of an operating system. These two simulation methodologies, together with the available benchmarks and traces, provide a set of methods for examining the variety of possible primary memory system architectures.

Chapter 5
Simulation Models

The DRAM simulation models which have been written are specified in the Java programming language. This language was chosen because of the time required for program development and debug, not because of speed of simulation. It is known that emulated Java bytecode runs slower (how much depends upon many factors, primarily the virtual machine implementation) than equivalent compiled C code [Flanagan97]. However, the time spent in development of the software is less for Java, at least for the developers of the DRAM models described, motivating the use of Java. Java code can be made to run in comparable time to compiled C code, but this requires compilation to native code as opposed to execution of emulated bytecode. Because of the hybrid solution of the execution driven methodology, where the application contained both C (SimpleScalar) and Java (DRAM models) source code, a compilation to native code was not undertaken. Additionally, the Java platform provides for a larger degree of portability between platforms, but this advantage has yet to be exploited as all simulations have been done on a common platform.

5.1 Compartmentalization

An attractive feature to using the Java programing language is the inherent compartmentalization which comes from writing (properly organized) object-oriented code. Within each of the three models discussed in Figure 5.2, there are a number of classes, each of which correspond to concrete objects in a simulation environment. These include: controller, bus, device, bank, schedule, access, and other concrete items within the DRAM architecture, as shown in Figure 5.1. This modular approach makes the logical understanding of the devices and debug of the models more understandable for a person

familiar with the hardware implementation. These attributes are certainly possible within another object-oriented language, however the framework of Java is more conducive to quality software design.

The three DRAM models are currently unique packages, however the work is underway to unify them under a single abstract interface. This would allow an easier integration of all DRAM models into a simulation environment. This abstraction is complete for the most recently developed model (DDR2) however it has not been completed for the remaining models. This initiative is aimed at making easier the integration of all DRAM Java models into a simulation environment.

The DDR2 model also is composed of a number of classes, but as it was the last model developed, and was intended to be extensible to other DRAM models, it has a layer of interfaces above the model definition to enable easy transitions between the various DRAM models. The interfaces specified for this model are dram_ctrl which contains all elements of the interface for initialization of the model, and dram_trans which contains interface methods for interrogating a specific transaction for schedule, latency, hit/miss status and other transaction information. The ddr2_ctrl class extends the dram_ctrl interface, and the ddr2_trans class extends the dram_trans interface. Future development will allow the DRDRAM and SDRAM models to implement the same abstract interface.

5.2 Model Description

Three unique DRAM models have been written, two of which are somewhat similar (SDRAM and DDR2), but each of which has unique elements and source. These three models are referred to as DRDRAM, SDRAM and DDR2 though their individual coverage extends beyond a single technology. Three individual packages were written because of the complexities involved in scheduling transactions for each of the interfaces. As will be described, the intent is to encompass these three models behind a single abstract interface, though this work is not yet complete.

The DRAM models are designed to process a sequence of accesses and provide accurate timing information for each phase of the access. The DRAM models themselves

are not designed to do any access re-ordering or coalescing, however the controller class is modular enough that these enhancements could be added into the controller.

Conceptually, the DRAM models have a single entry point, as is also the case with the DRAM memory system. A simple block, or object, diagram of the DRAM models is given in Figure 5.1. This figure shows each of the objects which encompass the model. A

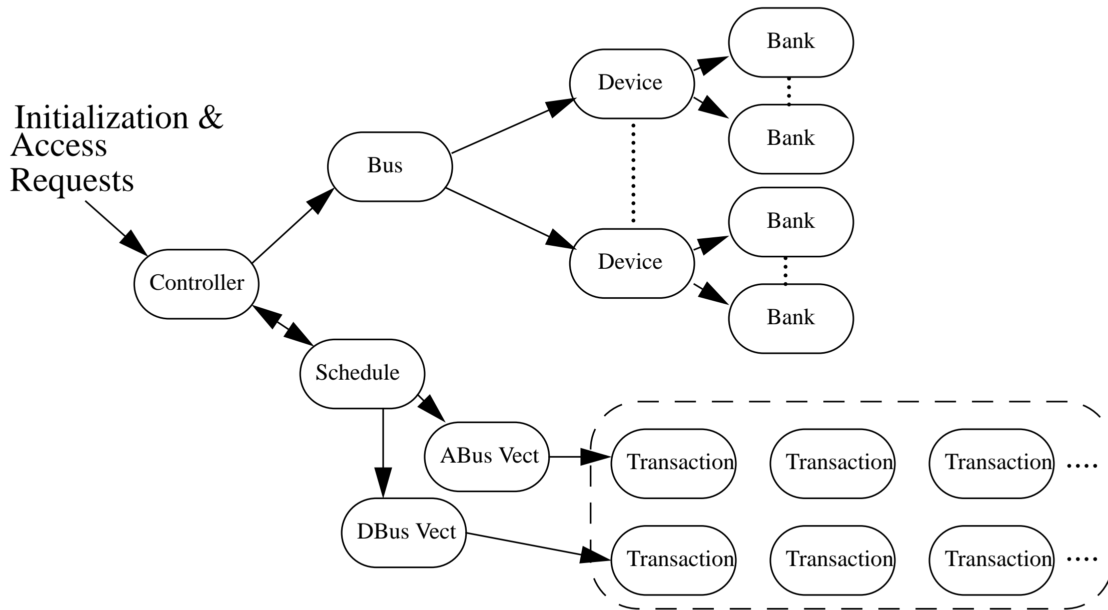


Figure 5.1: Conceptual Diagram of Simulation Models

This diagram is generic in that it is an attempt to show the structure of all three DRAM models in a single figure. Each of the above balloons represents an object in the model. The models have a single access point through the controller, and the accesses require two steps, first traversing the bus to determine access type, then traversing the schedule to determine timing information. The DRDRAM model has 3 transaction vectors contained within the schedule, corresponding to its three DRAM bus components.

controller object issues access requests to the bus (or channel) object to which it is associated. Logically, there are essentially two phases in each access. The first phase is the interrogation of the internal data structures of the DRAM devices to determine the type of read which is required — is a precharge required, is an activate/$\overline{\text{RAS}}$ required, is the bank busy. After this “type” for the access has been determined, then the models can examine the current bus/channel schedule and arrange the access with other bus activity for the lowest possible latency. The latency of the access is based upon the scheduled time(s) for prior accesses, and the availability of the bus.

The three models will be discussed in the chronological order of their development. Other models preceded these, such as a conventional (RDRAM) Rambus model, but results for those are not included in this thesis, and thus we will limit our discussion to the models with results presented.

5.2.1 DRDRAM

The DRDRAM model is specific to the Direct Rambus architecture. This model supports 600 MHz, 800 MHz and the recently announced 1066 MHz device speeds, but the results shown in Chapter 6 reflect operating speeds of 800MHz. Parameters within this model are tRCD, tRP, tRAS, RIMM size {64MB_8by64Mb, 128MB_16by64Mb, 128MB_4by128Mb, 256MB_8by128Mb} and controller policy {open-page, close-page-autoprecharge}. The DRDRAM model was the first developed, and contains the most unique code of the models presented here. Because the DRDRAM is fairly unique in hardware and implementation respects, this requires that the modeling software also be significantly different than the SDRAM models. For instance, the fact that the DRDRAM interface has 3 bus components as opposed to the two for the SDRAM devices required significant differences in the scheduling algorithms. Additionally, the shared sense-amps of the Direct Rambus architecture make maintaining the state about open pages unique to this simulation model.

The classes within the DRDRAM model are drd_ctrl, drd_channel, drd_device, drd_bank, drd_schedule, drd_trans, and drd_cycle. The Java source code for these classes is included in Appendix A.3, page 189. Each of these classes implements the corresponding hardware or conceptual object in a DRDRAM memory system implementation.

5.2.2 SDRAM

The SDRAM model supports simulation of PC100, PC133, DDR266 and DDR333. It actually will allow a higher level of clock speeds based upon these SDR and DDR SDRAM interface, but the above mentioned interface specifications are those most likely to be produced which are also supported by this model. Parameters within this

model are SDR or DDR data signalling, bus frequency, CL, tRCD, tRP, DIMM size of {64MByte; 128 MByte}, controller policy of open-page or close-page-autoprecharge.

The SDRAM model like the others is composed of a number of classes. The classes within the SDRAM model are sdram_ctrl, sdram_device, sdram_bank, sdram_bus, schedule and trans. The Java source code for these classes is included in Appendix A.1, page 135. The SDRAM classes together comprise the simplest of these three models. They model the simplest set of DRAM architectures, with the fewest choices of parameters. However, this model accurately simulates the devices which currently comprise the majority of the primary memories for computer systems sold thus far in 2000.

5.2.3 DDR2

The DDR2 models supports simulation of DDR2, DDR2VC and DDR2EMS at 200MHz. This simulation engine has support for address remapping at the controller. The parameters allow for the user to set CL, tRCD, tRP, address remapping either on or off, set DIMM sizes to 256 MByte or 512MByte, set the controller policy to open-page or close-page-autoprecharge. Additionally, in the case of a DDR2EMS simulation the controller can be set to do all writes in either no-write-transfer or write-transfer mode(s). And in the case of a DDR2VC simulation, the controller can be set to allocate virtual channels in a Random or true least-recently-used (LRU) policy. The DDR2 model is the most parameterizable with regard to controller policy, and also includes the ability to enable controller address remapping.

The DDR2 model also is composed of a number of classes, but as it was the last model developed, and was intended to be extensible to other DRAM models, it has a layer of interfaces above the model definition to enable easy transitions between the various DRAM models. The interfaces specified for this model are dram_ctrl which contains all elements of the interface for initialization of the model, and dram_trans which contains interface methods for interrogating a specific transaction for schedule, latency, hit/mis status and other transaction information. The ddr2_ctrl class extends the dram_ctrl interface, and the ddr2_trans class extends the dram_trans interface. Future development will allow the DRDRAM and SDRAM models to be integrated behind a single abstract

interface, making the integration of all of the simulation models into a single simulation an easier undertaking. The Java source code for these classes, as well as the abstract interface definition is included in Appendix A.2, page 156.

5.3 Statistic Generation

All of the models collect common statistics, typically in the same portion of the same classes when possible. As discussed in Section 5.2 — Model Description there are two phases to the access handling. Statistics which are not time dependent — number of read and writes, number of cache hits, bytes per access, number of temporally adjacent accesses to each bank, etc. — are gathered in the first phase where the access "type" is determined. Statistics which are time dependent — access latency, bus utilization, time elapsed, etc. — are gathered after completion of the second phase when the access being processed has been scheduled, prior to the latency being returned to the invoking controller object.

There are some special case statistics in the three models, such as the Rambus bus utilization, which is composed of 3 rather than 2 components, and the cache hit rates in the cache-enhanced DDR2 architectures. In these cases, the statistic gathering code fragments are placed where they best fit between the two phases of the access.

The Java language has a common method in all classes used to provide information about an object. The tostring() method is used for printing object statistics in all cases. At the completion of each simulation, these methods are used to print the statistics from the DRAM simulations to a text file. Subsequently these text files are processed by Perl post-processing programs to generate tables of the interesting results.

5.4 Validation

Validation of a simulation engine is a critical portion of the design process. When performing comparisons based upon existing hardware, verification is simply comparing the output of applications to a known good result. In the case of a software simulation, there are many things which can be coded incorrectly, not all of which would generate an erroneous result. In software, the behavior of the hardware being modeled could be

incorrectly characterized in a manner that is impossible to fabricate. The validation process guarantees that the software model accurately reflects the behavior of the hardware which is being simulated.

Validation of the models was performed using synthetic traces which have a very regular, known behavior and for which timings can thus be calculated. Table 5.1 shows

Table 5.1: DRAM Model Validation using Synthetic Traces

Trace	DRAM Architecture	Calculated Hit Rate	Calculated Cycles / access (sequential)	Calculated Data Cycles / access	Simulated Hit Rate	Simulated Cycles / access
Synthetic Read only 128 Byte accesses 256 Byte stride	PC100	0.875	17	16	0.86580	16.67078
	DDR266	0.9375	8.375	8	0.93060	8.27738
	DDR2	0.9375	8.5625	8	0.93750	8.47491
	DRDRAM	0.75	12	8	0.73710	10.09906
Synthetic Read-only 128 Byte accesses 512 Byte stride	PC100	0.75	18	16	0.74518	17.29153
	DDR266	0.875	8.75	8	0.86814	8.52734
	DDR2	0.875	9.125	8	0.87500	8.91421
	DRDRAM	0.5	16	8	0.50000	11.81772
Synthetic Read only 128 Byte accesses 1024 Byte stride	PC100	0.5	20	16	0.49086	18.54453
	DDR266	0.75	9.5	8	0.74307	9.02750
	DDR2	0.75	10.25	8	0.75000	9.79285
	DRDRAM	0	24	8	0.00000	20.61815
Synthetic Read only 128 Byte accesses 2048 Byte stride	PC100	0	24	16	0.00000	20.99763
	DDR266	0.5	11	8	0.49098	10.03562
	DDR2	0.5	12.5	8	0.50000	11.55022
	DRDRAM	0	24	8	0.00000	20.60199

Table 5.1: DRAM Model Validation using Synthetic Traces

Trace	DRAM Architecture	Calculated Hit Rate	Calculated Cycles / access (sequential)	Calculated Data Cycles / access	Simulated Hit Rate	Simulated Cycles / access
Synthetic Read only 128 Byte accesses 4096 Byte stride	PC100	0	24	16	0.00000	20.99518
	DDR266	0	14	8	0.00000	11.99906
	DDR2	0	17	8	0.00000	15.05708
	DRDRAM	0	24	8	0.00000	20.57135

the results of a set of simulations and the calculated values for a subset of the synthetic traces that were executed. The results in this table have been modified accounting for the fact that the DDR architecture limits the access granularity to 4 bus-widths. This means that each 128 Byte access results in 4 bus-level accesses. The results have been compensated to allow comparison between the multiple architectures. The three middle columns are calculated values for both the row-cache hit rate and cycles per access. The two right hand columns are the simulated values for both of the both the row-cache hit rate and cycles per access.

The calculated hit rate does not take into account the effects of refresh, and is thus an upper bound on the simulated hit rate. In all cases, the simulated hit rate is the same or slightly below the calculated hit rate. This validates the model's ability to determine the hit or miss status of the accesses in the DRAM device which it simulates.

The calculated cycles / access (sequential) is calculated by taking the end-to-end time of each access, from the first precharge, row or column packet to the last data Byte transferred, and averaging this value across all accesses. This value can be used as an upper bound of the simulated cycles / access because it is average access time assuming no overlap between sequential accesses. The calculated data cycles / access is the average number of data cycles required by each access. This value can serve as a lower bound of the simulated cycles / access because regardless of the overlap achievable - each access must occupy the data bus for the number of cycles required to transfer data. The simulated cycles / access is verified by falling between the two of these values. For all of the synthetic traces, the simulated cycles / access falls between the upper and lower bounds

determined by these two calculated values. This validates the model's ability to correctly schedule the DRAM accesses and determine the access latencies.

5.5 Other Simulation Code

The DRAM models are used in both the trace driven methodology as well as the execution driven methodology. In order to enable operation of the same models in both these configurations auxiliary code was required.

In the case of the trace driven methodology, the auxiliary code encompassed a simple front-end to read in the trace elements and provide them to the DRAM models in as a simulated access stream. This code was written in Java, and is not included in the appendixes.

In the case of the execution driven methodology, the auxiliary code encompassed the incorporation of the Java Virtual Machine (JVM) as well as the DRAM models into the SimpleScalar environment. This code was written in C, but relied heavily upon the Java Native Interface (JNI) libraries to enable the invocation of Java methods within a C environment. This code is contained in Appendix A.4, page 215, and is compiled along with the SimpleScalar 2.0-MSHR source to produce the execution driven simulation engines. These engines rely upon the Java class files produced by compilation of the code in Sections A.1 through A.3, but these classes are not loaded until run-time.

Finally, each simulation run, whether trace or execution driven, produces a log file which contains the data pertaining to the simulated execution of a particular benchmark, on a particular DRAM, at a processor speed, with a given cache configuration and using a specific controller policy. There are other parameters, but the output of each of these permutations is a single log file. The program which is required for the generation of meaningful results graphs is a PERL script which can take these hundreds of log files and produce tab delineated tables, in a text format, based upon parameters to the PERL script, and intended for import into Excel, StarOffice or an equivalent graph production package.

These auxiliary programs, are as necessary as the DRAM models to the simulation flow, and are part of the framework which generated the results presented in Chapter 6. It is built upon prior research packages such as Dinero, SimpleScalar, and more common

applications. This framework will continue to be utilized for ongoing research, after this thesis is completed and defended.

Chapter 6
Results

The experiments presented in this chapter are comprised of the simulations of computer systems, each with a specified configuration, on a known benchmark or DRAM access trace. Results have been gathered, for a wide range of configurations, under the methodologies described in Section 4.2 — Trace Driven, and Section 4.3 — Execution Driven. Each simulation results in a output file describing the behavior and performance of the memory system, and in the case of the execution driven simulations, the processor system and caches. The number of simulations and variety of system architectures do not allow for all information to be shown in a single graph, or as a single result. The sections of this chapter are intended to each cover either a factor contributing to the performance of the DRAM access stream, or a resulting metric upon which DRAM systems may be evaluated. The sections in this chapter are intended to be read in order, as they build one upon each other, but may be examined individually if the reader is only interested in a specific element of the DRAM architecture or performance.

6.1 Introduction

This thesis is based upon a long schedule of research and simulation which began with comparison of cache behavior in Alpha, x86 and MIPS processors, and followed a course of evolution which has led to the data presented in this chapter comparing DRAM architecture performance. By no means will all of the simulations or experiments performed be presented. Only those experiments which form a cohesive investigation into the attributes of DRAM which impact system performance will be presented. Some of these results therefor have been previously published in [Cuppu99][Davis00a] or other venues, and not all are from the same simulation environment.

Simulations have been performed over a number of generations of the simulation engines and benchmarks. The execution based results presented here are from two distinct points in the evolution. The latest change to the simulation environment was made to accommodate the use of the SPEC2000 benchmark suite.

6.1.1 DRAM Characteristics

Not all of the DRAM discussed in Chapter 3 will be examined in the simulations presented in this chapter. Characteristics for those DRAM which are simulated are given in Table 6.1. For some of these architectures there are multiple possible choices for these

Table 6.1: Overview of DRAM Characteristics

	PC100	DDR266 (PC2100)	DRDRAM	DDR2	DDR2EMS	DDR2VC
Potential Bandwidth	0.8 GB/s	2.133 GB/s	1.6 GB/s	3.2 GB/s	3.2 GB/s	3.2 GB/s
Interface	Bus 64 Data bits 168 pads on DIMM 100 Mhz	Bus 64 Data bits 168 pads on DIMM 133 Mhz	Channel 16 Data Bits 184 pads on RIMM 400 Mhz	Bus 64 Data bits 184 pads on DIMM 200 Mhz	Bus 64 Data bits 184 pads on DIMM 200 Mhz	Bus 64 Data bits 184 pads on DIMM 200 Mhz
CL - tRCD - tRP (clock cycles)	3-3-2	2-2-2	8-7-8	3-3-3 AL = 0	2-4-3	2-4-3
Latency to first 64 bits (Min. : Max)	(3 : 9) cycles (22.5 : 66.7) nS	(2.5 : 6.5) cycles (18.8 : 48.8) nS	(14 : 32) cycles (35 : 80) nS	(3.5 : 9.5) cycles (17.5 : 47.5) nS	(2.5 : 9.5) cycles (12.5 : 47.5) nS	(2.5 : 18.5) cycles (12.5 : 92.5) nS
Notes:			More Banks Shared Sense-Amps	Limited Burst Size	1.4% Area Overhead	4.3-5.5% Area Overhead

parameters, such as the multiple possible $\overline{\text{CAS}}$ grades (2-3-2, 3-3-2 and 3-3-3) available for PC100. Similarly, the version of EMS cache enhanced DDR2 proposed for incorporation into the JEDEC standard specifies a 3-3-3 timing. The results using a 2-4-3 DDR2EMS timing model were generated prior to release of this ballot. Further work should include different timings for all architectures. The values used in the simulations correspond to those in Table 6.1. As can be seen from the bandwidth values, all of the DDR2 interface architectures have an advantage in raw or potential bandwidth. There are

a number of ways which this could be factored out, either by replicating the bus so that the bandwidth of all technologies was matched, or by matching the number of pins in the interface such that each interface used the same number of pins. Previous research [Cuppu99] has done this, but for the experiments presented in Chapter 6 all of the interfaces are as specified in Table 6.1.

6.1.2 Benchmarks

The benchmarks simulated are intended to cover all types of applications, and are shown in Table 6.2,. The execution driven benchmarks are drawn from a number of

Table 6.2: Benchmarks & Traces for Simulation Input

Name	Source	Input Set / Notes
cc1	SPEC95	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O spec95-input/gcc/*.i
compress	SPEC95	spec95-input/compress/test.in
go	SPEC95	spec95-input/go/null.in
ijpeg	SPEC95	-image_file ../spec95-input/jpeg/penguin.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp
li	SPEC95	spec95-input/li/boyer.lsp
linear_walk	hand-coded	
mpeg2dec	Mediabench	-b mei16v2.m2v -r -f -o0 rec%d
mpeg2enc	Mediabench	100M.options.par out.m2v
pegwit	Mediabench	-e my.pub kennedy.xls pegwit.enc < encryption_junk
perl	SPEC95	scrabbl.pl < scrabbl.in
random_walk	hand-coded	
stream (unrolled)	McCalpin	
stream_no_unroll	McCalpin	
oltp1w	IBM	(trace) 2 GByte address space
oltp8w	IBM	(trace) 2 GByte address space
xm_access	Transmeta	(trace)

Table 6.2: Benchmarks & Traces for Simulation Input

Name	Source	Input Set / Notes
xm_cpumark	Transmeta	(trace)
xm_gcc	Transmeta	(trace)
xm_quake	Transmeta	(trace)
apsi	SPEC2000	(test input)
applu	SPEC2000	(test input) applu.in
fpppp	SPEC2000	(test input) natoms.in
hydro2d	SPEC2000	(test input) hydro2d.in
mgrid	SPEC2000	(test input) mgrid.in
su2cor	SPEC2000	(test input) su2cor.in
swim	SPEC2000	(test input) swim.in

sources: the SPEC95 suite [SPEC95], the SPEC 2000 suite [SPEC00], the Mediabench suite [Lee97], two versions of the McCalpin Stream benchmark [McCalpin00], each compiled with different optimizations, and finally linear walk and random_walk, hand coded as bandwidth limited applications with predictable and unpredictable access patterns respectively. The traces used are drawn from two places. The IBM online-transaction-processing (OLTP) trace gathered on a one-way and 8-way SMP, and the Transmeta traces for Access, Cpumark, Gcc and Quake. These are certainly not the only benchmarks applicable to this type of a study, but it is our hope that these selections encompass enough breadth that all application behavior patterns are represented.

6.1.3 Comparative Framework

We present at least two types of investigation here. The first, and probably more telling are examinations of how system parameters (processor frequency, cache configuration, number of MSHRs, and other inputs) change the performance for a specific DRAM architecture. These provide insights into how the DRAM architecture is able to cope with a change of system. The second type of comparison is for a given system configuration, which DRAM architecture (PC100, DDR266, DDR2, DRDRAM, etc.) provides the best system performance. These latter simulations, while of great interest to

system designers in the current market context, provide less understanding of the architectures themselves and more information of a comparative nature. While being accurate within the current design parameters they are short-sighted with regard to development of next generation high performance DRAM architectures.

Examinations of how system parameters affect the performance of a fixed DRAM architecture are insightful because they provide understanding on how the input parameters to the system generate improvements or reduction based solely upon the DRAM performance. In addition, these input parameter studies give us direction for the future changes which are likely to take place with system designs.

Comparisons between architectures are best made when each DRAM architecture is performing at the best level achievable with a reasonable controller. Previous work has established a set of reasonable controller policies which yield the lowest average execution time for each of the DRAM architectures examined. For architectural comparisons between PC100, DDR266 and DRDRAM an open-page policy will be used, for DDR2 an open-page with address-remapping will be used, and for DDR2EMS and DD2VC a close-page-autoprecharge with address-remapping will be used [Davis00a]. Data has been collected using other controller policies for each of these architectures, but these policies have typically, though not always, provided the best performance on the benchmarks examined.

In performing simulations of DRAM performance, it is difficult to identify benchmarks that will exercise the entire memory system as is done with a hardware system. The execution driven simulations do not include the operating systems activity of a true system. This unfortunately significantly changes the DRAM access pattern, one aspect of which is a reduction in the memory space utilized in comparison to the same benchmark running on a conventional operating system. Additionally, the SPEC suites are known to have a small memory footprint, that frequently allows them to fit entirely into a L2 cache. Stream is a benchmark explicitly added to reflect the memory intensive class of applications which focus upon streaming media.

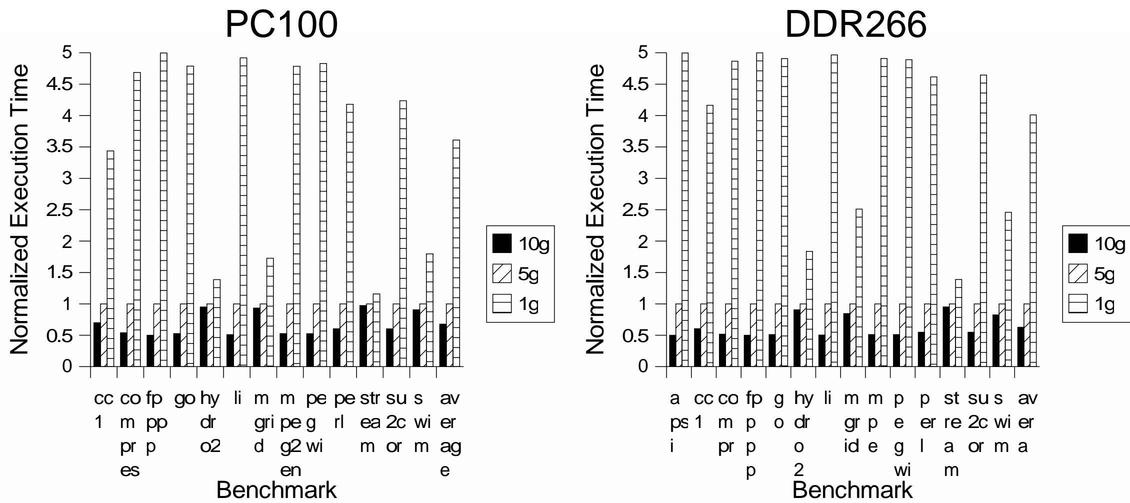
The traces also are simulated in a manner which allows them to emulate streaming media applications. The traces used in these simulations are processed in a manner which is very optimistic about the arrival time of accesses. This is because some traces do not

contain timestamps, and other traces contain timestamps which are only accurate within the context of a specific memory system architecture. Absence of timestamps during simulation has the consequence that the DRAM access stream is compacted in time to the minimal amount of time required to perform all accesses. Both the Transmeta and the IBM OLTP traces also contain the access stream resulting from the operating system activity, giving them a larger memory footprint than the same application in an execution driven configuration. This arrangement places significant stress on the memory system, as if the traced applications were highly memory intensive.

6.2 Processor Frequency

The frequencies of many components of the computer system affect the performance of the DRAM system. The bus frequency of the DRAM devices is the most significant contributor to the performance and this is reflected in the increasing frequency of DRAM in evolutionary advancements. Secondary to this bus frequency is the processor frequency which affects the DRAM performance most directly through the inter-access timing. Figure 6.1 shows the impact of processor frequency upon the overall benchmark execution time for three different DRAM architectures. For all of these benchmarks, the execution time has been normalized to the 5Ghz processor execution time such that all of the benchmark execution times can be displayed upon the same graph. As can be seen, the change in relative execution time is least significantly affected for those benchmarks which are highly memory intensive, namely the McCalpin Stream benchmark.

This shows that as benchmarks become more memory intensive, they are going to be more dependent upon the performance of the DRAM memory system, and less upon strictly the processor operating frequency, which has served as the major marketing tool for computer systems for a number of years. Memory intensive applications are any applications which have a significant amount of accesses escape from the cache system, because they are streaming unique data for each access, have a large footprint and little locality in the accesses, similar to database or transaction-processing, or for other reasons. As applications with very little data re-use (streaming media) become more common, the



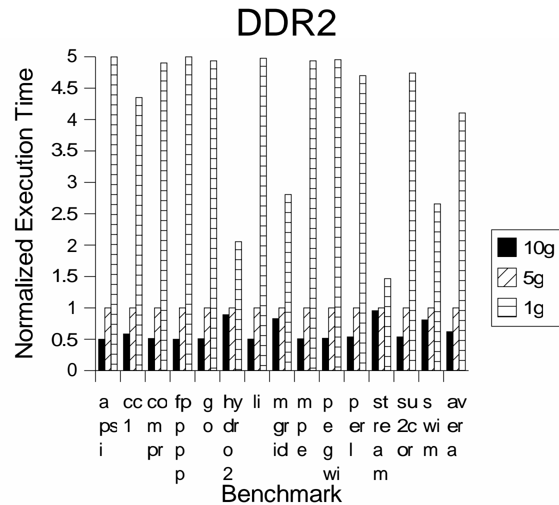


Figure 6.1: Processor Frequency

These graphs all assume an 8-wide superscalar processor, 256KB L2 cache, 16 MSHRs

Top Left: PC100; Top Right: DDR266; Bottom Right DDR2

memory access behavior of programs will emphasize the performance of the lower levels of the memory hierarchy.

6.3 L2 Cache Interaction

The caches at a higher level in the memory hierarchy not only affect the number of accesses which reach the DRAM, but also affect the spatial and temporal locality of the references. Larger caches typically have longer line sizes, and these longer lines generate larger granularity requests to the lower levels of the memory hierarchy. In addition, larger caches generally result in fewer accesses to the DRAM memory system, except in the case of streaming media applications where the caches are not effective because of the lack of data re-use.

In cases where the effective cache size is sufficiently large that access locality after the caches does not justify using an open page controller policy, a close page policy is used. This is typical for large SMP configurations [Coteus00]. Because of the relatively small data footprint of the majority benchmarks used in these studies, i.e. SPEC, we have not generally explored cache sizes large enough to eliminate the effectiveness of an open page controller policy.

6.3.1 Execution Driven Cache differences

Only three cache configurations were examined within the execution driven framework. These were: 1) 32KB-I/32KB-D L1 caches with 32 Byte linesizes and no L2; 2) 32KB-I/32KB-D L1 caches with 32 Byte linesize and a 256KB unified L2 cache with 128 Byte linesize; and 3) 64KB-I/64KB-D L1 caches with 32 Byte linesizes and a 1MB unified L2. cache with 256 Byte linesize. Configuration number two serves as the baseline for most of the other simulations done, where cache configuration is not the item of interest.

Figure 6.2 shows the accesses made to the DRAM controller for each of the three

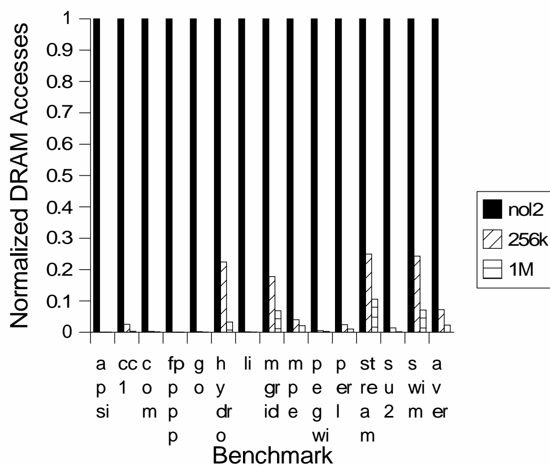


Figure 6.2: Post L2 Cache Accesses

All DRAM architectures have the same number of accesses to the controller following the same cache configuration.

Left are the normalized controller accesses which escape each of the cache configurations, for all simulated DRAM architectures

cache configurations described. Since the accesses generated by the application at the processor are the same, and the L2 caches are the same, the accesses which escape these caches are also the very similar. Some small differences come from incorrect speculation, or prefetches, but the impact of these non-application generated accesses are relatively

small. Thus, the normalized controller accesses in Figure 6.2 are the accesses which escape the L2 for all of the simulated DRAM architectures.

As can be seen, even a 256k unified L2 cache is able to filter out over 75% of the accesses for all benchmarks, and over 90% of the accesses on average. An L2 cache can thus significantly increase the performance of a microprocessor system. It is interesting to examine the relative improvement in execution time in comparison to the improvement in the number of accesses.

Figure 6.3 shows the effects upon the execution time for each of three different

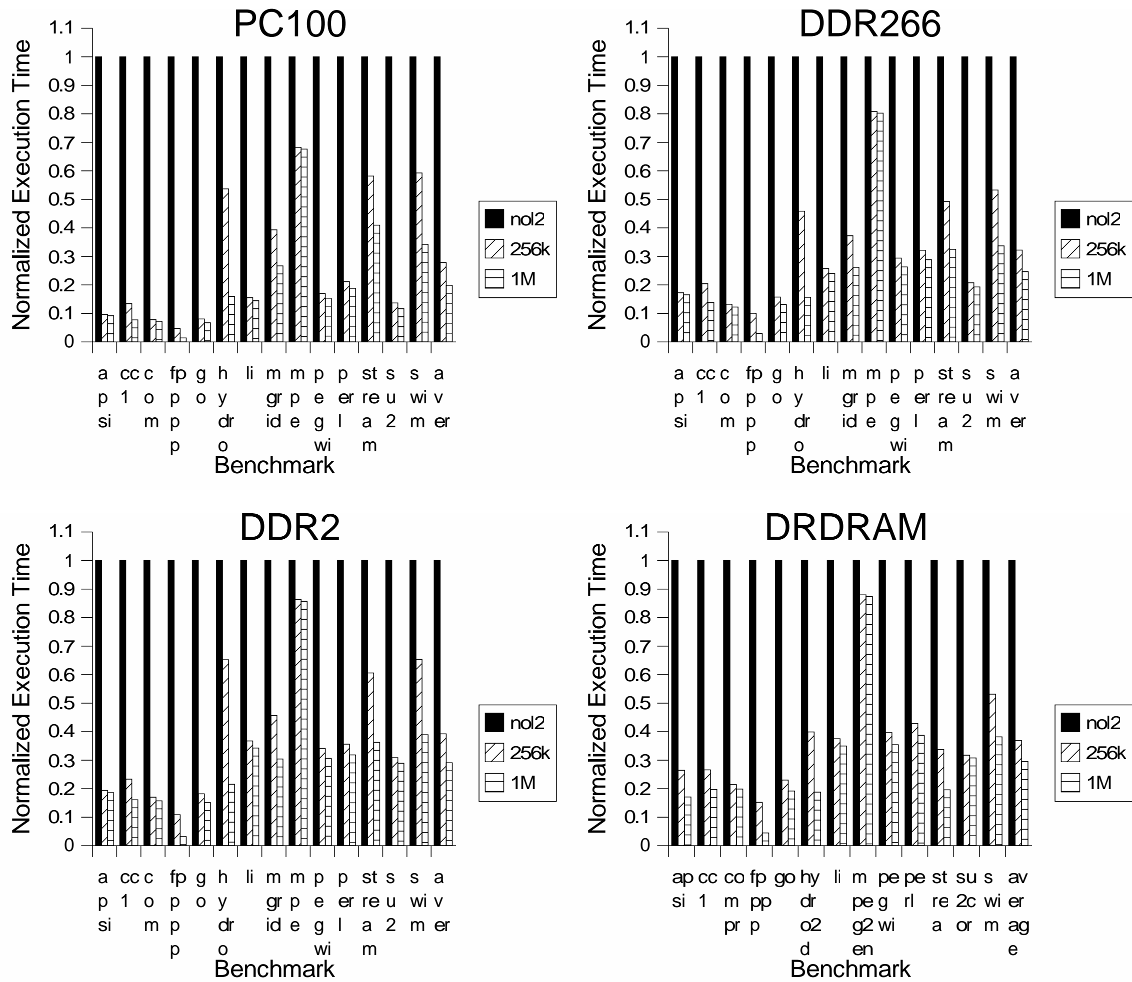


Figure 6.3: L2 Cache Size Impact (execution driven)

Impacts upon benchmark execution time and the number of DRAM accesses observed with three cache configurations

Top Left: PC100; Top Right DDR266; Bottom Left: DDR2; Bottom Right: DRDRAM

cache configurations in four simulated memory systems: PC100, DDR266, DDR2 and

DRDRAM. In all cases, the execution time is greatly reduced by the addition of an L2 cache. This reduction in execution time is on average 70%, with the one megabyte configuration increasing the average reduction to approximately 75%. The two benchmarks which have the least reduction in execution time, mpeg2enc and stream, have a correspondingly higher number of accesses which escape the L2 cache to become DRAM accesses. The number of DRAM accesses is reduced by an even larger percentage, over 90% for both the cache sizes on the PC100 and DDR266 memory systems, and over 80% for both cache sizes on the DDR2 memory system. The reduction in the number of accesses is not surprising, that is the intent of cache. What is surprising is the differences between these architectures, and the fraction of benchmarks which do not exercise the DRAM system when run in a system with 1 MByte of cache. The addition of an operating system, background tasks, or I/O would likely increase the number of L2 cache misses.

6.3.2 OLTP Large-Cache Simulations

The two On-Line Transaction Processing (OLTP) traces provided by IBM were gathered by IBM, using PowerPC machines. The first trace monitors the activity of a single processor system (oltp1w), the second trace monitors the activity of an 8-way SMP (oltp8w). In both cases the activity was monitored at the level of the post-L1 cache snooped bus. The primary caches are split instruction and data, 128Kbytes each, 4-way set-associative, with 128-Byte lines. Three configurations were simulated for these traces: 1) without any additional L2 cache; 2) with an additional 1Mbyte unified L2 cache; and 3) with an additional 4Mbyte unified L2 cache.

Figure 6.4 shows the number of unique accesses and the trace access fulfillment time for both OLTP traces, and for each of the three cache configurations described above. These graphs are strongly affected by the linesize of the lowest level of the caching structure. In those traces where there is no L2 cache, all accesses are of size 128 bytes. For both the 1Mbyte and 4Mbyte L2 caches, the linesize, and thus access size, is 256 bytes. This increase in linesize in a transaction processing environment causes the load on the DRAM system, and thereby the execution time, to increase. The number of unique accesses decreases for the PC100 and DDR266 configurations, as the upper level caches filter out accesses, as caches are intended to do. The number of bytes transferred across the

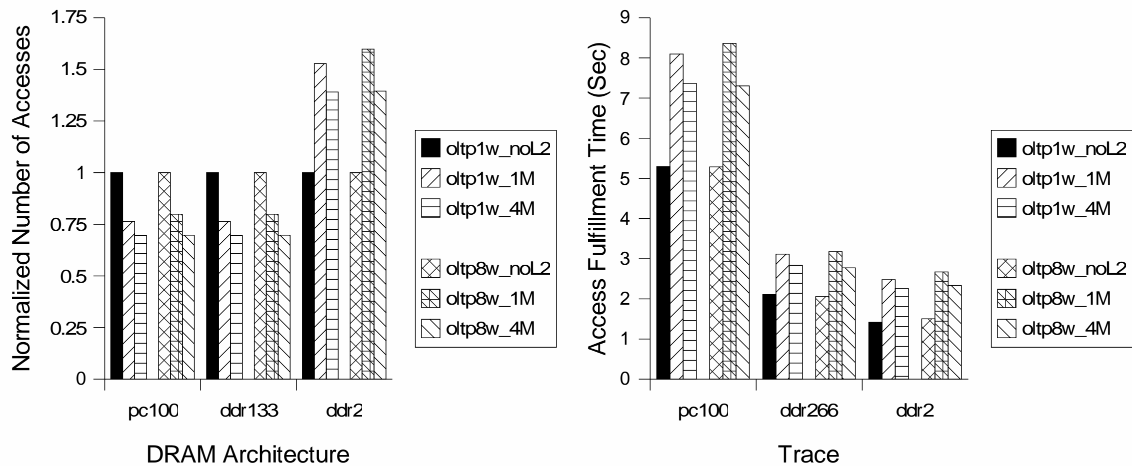


Figure 6.4: L2 Cache Size (OLTP traces)

These graphs make use of the IBM OLTP Traces. In the case where an L2 is present the Dinero cache simulator was used to preprocess the trace before processing by the DRAM models
Left: Normalized Number of Accesses; Right: Seconds for DRAM Access fulfillment

DRAM bus by each trace is the same for all architectures, and matches the graph of the number of accesses for the DDR2 architecture, as that architecture has a fixed number of bytes per access. In examining the absolute number of accesses, rather than normalized values, the PC100 and DDR266 values are identical, while the DDR2 executes 4 accesses for each PC100 access in the no L2 (128 Byte line) case, and 8 unique accesses for each PC100 access in both of the 256 Byte linesize L2 cases. This is because of the burst limit which was adopted in the DDR2 specification. The bandwidth of each of these technologies comes into play when we examine the access fulfillment time. As was mentioned above the number of bytes transferred for each trace is the same across all DRAM architectures. The bandwidth of the three technologies allows DDR2 to transfer that number of bytes in a shorter period of time. This shows us that for the trace driven simulations, because of the lack of access dependence, and the fact that we are measuring the time for access fulfillment rather than benchmark completion, the bandwidth of the interface is the primary characteristic determining fulfillment time.

It would be interesting to perform the same experiments, with L2 linesizes of 128 bytes, which would eliminate the increase in the number of bytes transferred as the L2

cache is added. This was study was not possible at this time due to the disk space and compute time available.

6.4 MSHR Effects

For any modern DRAM architecture, whether it be an asynchronous interleaved configuration or a synchronous configuration, one aspect to increasing primary memory system performance or equivalently reducing system performance degradation due to memory access latency, is parallelism in the memory system. In order to provide the memory system with a parallelizable workload or set of accesses, the processor(s) generating primary memory accesses must be capable of generating multiple outstanding requests. Miss information / Status Holding Registers (MSHRs) are one mechanism to support multiple outstanding memory requests from a single processor [Kroft81]. This is the mechanism which is supported in the version of SimpleScalar used for all execution driven simulations presented in this chapter. Doug Burger added support of MSHRs to SimpleScalar version 2.0, and on top of this was added the support for DRAM models implemented in Java. The MSHR support in this version of SimpleScalar allowed for changing the number of MSHRs allocated for the cache, prefetch and targets. We have explored 5 MSHR configurations within the execution driven simulations. They each correspond to settings for these three MSHR values of: 16 MSHR - 16 cache, 4 prefetch, 4 targets; 8 MSHR - 8 cache, 2 prefetch, 2 targets; 4 MSHR - 4 cache, 2 prefetch, 2 targets; 2 MSHR - 2 cache, 1 prefetch, 1 target; 1 MSHR - 1 cache, 1 prefetch, 1 target. It is not possible to specify ZERO MSHRs within this simulation environment. Therefor, even when the settings are in the 1MSHR configuration, the processor is still able to continue processing memory accesses which hit in the upper levels of the cache while there is an outstanding DRAM access.

Figure 6.5 shows the reduction in DRAM concurrency for each of the five MSHR configurations. DRAM concurrency in this graph is defined as the fraction of time that there are simultaneously two accesses utilizing the DRAM interface. This occurs when one access is occupying the address signals of the bus, and the other access is occupying the data signals of the bus. The first observation from this figure is that the DDR2

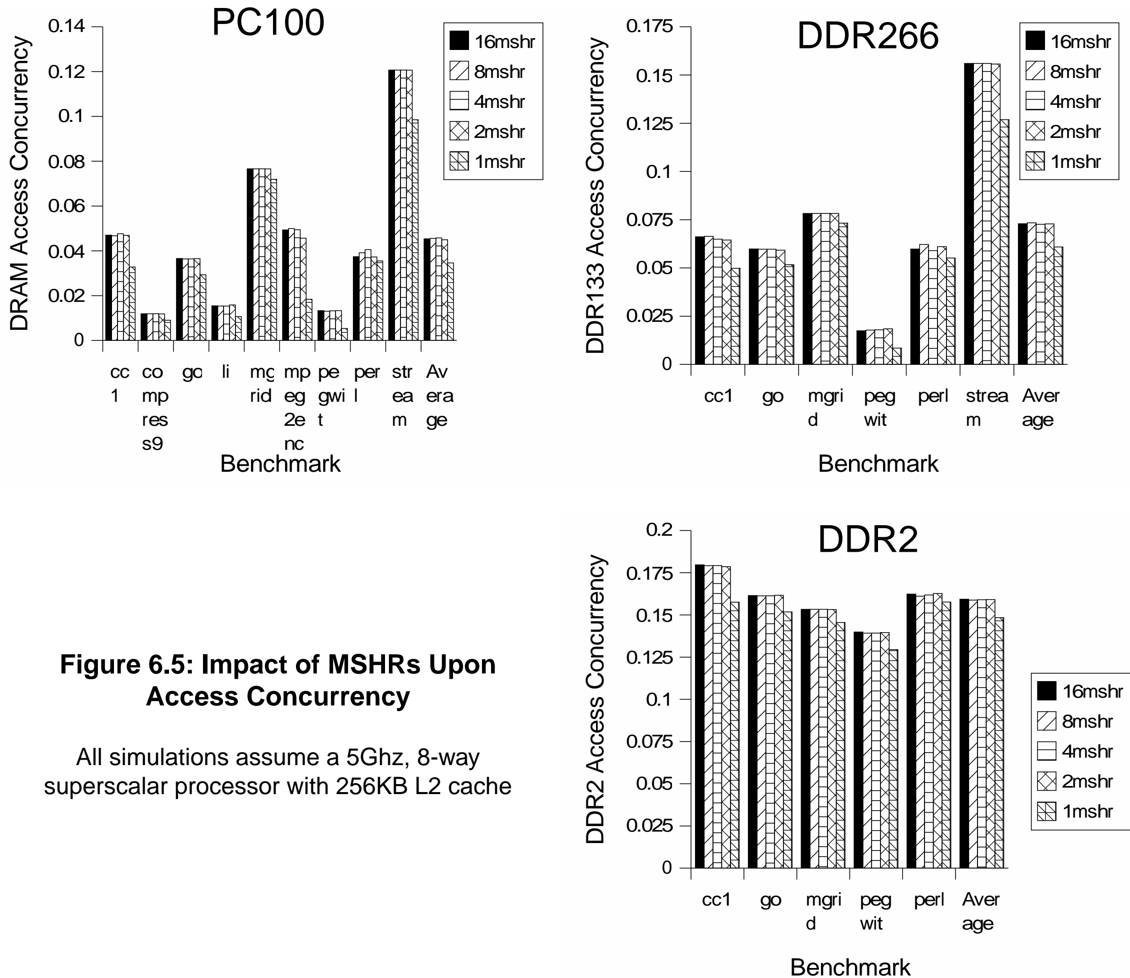


Figure 6.5: Impact of MSHRs Upon Access Concurrency

All simulations assume a 5Ghz, 8-way superscalar processor with 256KB L2 cache

architecture provides support for higher levels of concurrency than either the PC100 or DDR266 architectures. This is due to a number of characteristics. First, the DDR2 architecture, as simulated limits the burst size of accesses to 4 bus widths (see Section 3.3.7) and this results in a much larger number of accesses, and specifically accesses which are sequential and page hits. Second, the DDR2 architecture has changed the profile of the access timings such that accesses can be overlapped with fewer “dead” or transition cycles to reduce the amount of utilized, and concurrent cycles. On average the reduction in concurrency which results from a reduction in MSHRs is small, less than 2%, but this is partially because the DRAM memory systems investigated only support 3 accesses in flight. If the memory system supported a larger number of accesses, either through a partitioned memory space, or a higher concurrency architecture, the number of MSHRs may impact the performance more significantly.

Figure 6.6 shows the impact of the same reductions in the number of MSHRs upon

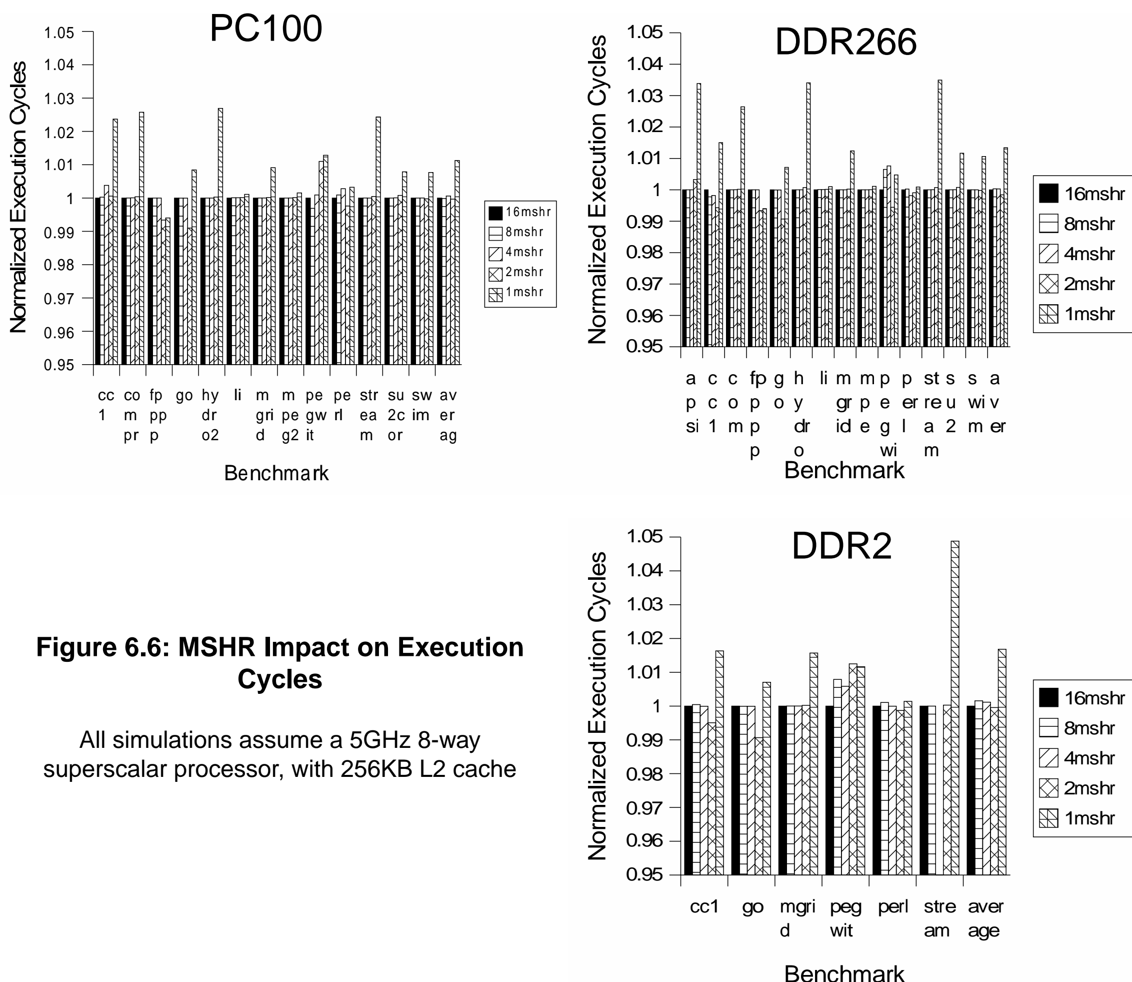


Figure 6.6: MSHR Impact on Execution Cycles

All simulations assume a 5GHz 8-way superscalar processor, with 256KB L2 cache

the benchmark execution times. The most significant observation from these graphs is that the number of MSHRs only measurably affects those benchmarks which are highly memory intensive such as the stream benchmark. This is because, to a first order, the DRAM itself does not significantly affect the execution times of the SPEC2000 benchmarks which dominate these graphs. The observation that the performance can actually improve in some cases when the number of MSHRs is reduced can be explained by the reduction in the number of prefetch MSHRs which increase the load upon the DRAM bus.

The average increase in execution time when decreasing the number of MSHRs to 1 is lower than might be expected, approximately 2% in these simulations. This indicates that the parallelism available in 16 MSHRs is not being fully exploited by these simulations. The applications which do have the largest degree of available memory

system parallelism, such as pegwit, hydro2d, and stream, show the largest degree of degradation when the number of MSHRs is decreased. Even for these benchmarks, four MSHRs is sufficient to exploit all of the parallelism which is provided by the DRAM architectures examined.

This result shows that the 16 MSHRs used in the remainder of the simulations is more than sufficient, within the architecture simulated by SimpleScalar, to support all the available memory system parallelism. In order to increase the amount of parallelism generated by the simulation engine, some of the techniques described in Section 2.3, such as speculative data prefetching, or data value prediction where values are used for address generation, could be implemented in this microprocessor simulation engine.

6.5 Controller Policies

As was discussed in Section 3.5, the policies exercised by the DRAM controller can significantly affect the performance of a DRAM memory system. In some cases, there is an obvious choice of controller policies based upon the architecture of the device. For example, using an open-page policy with a Virtual Channel device provides no advantage, and will always yield lower performance than a close-page-autoprecharge policy. However for a number of device architectures, policy choices are based upon the overall memory system architecture (i.e. how much cache precedes the DRAM) and the type of benchmarks to be run. The fundamental choice between maintaining the most recently accessed DRAM row as open, or closing that row and precharging the bank is the choice between the close-page-autoprecharge and open-page controller policies.

Figure 6.7 shows the impact of controller policy upon the execution driven simulations. The performance of the open-page policy in the execution driven simulations is very much dependent upon the page or line size of the DRAM arrays. The architecture which provides the best performance in the open-page configuration is the DRDRAM architecture which also has the smallest page size and the largest number of banks used to implement a fixed address space. The architecture which provides the worst performance in the open-page configuration is the DDR2 architecture which has the largest page size and the smallest number of banks used to implement a fixed address space. The PC100

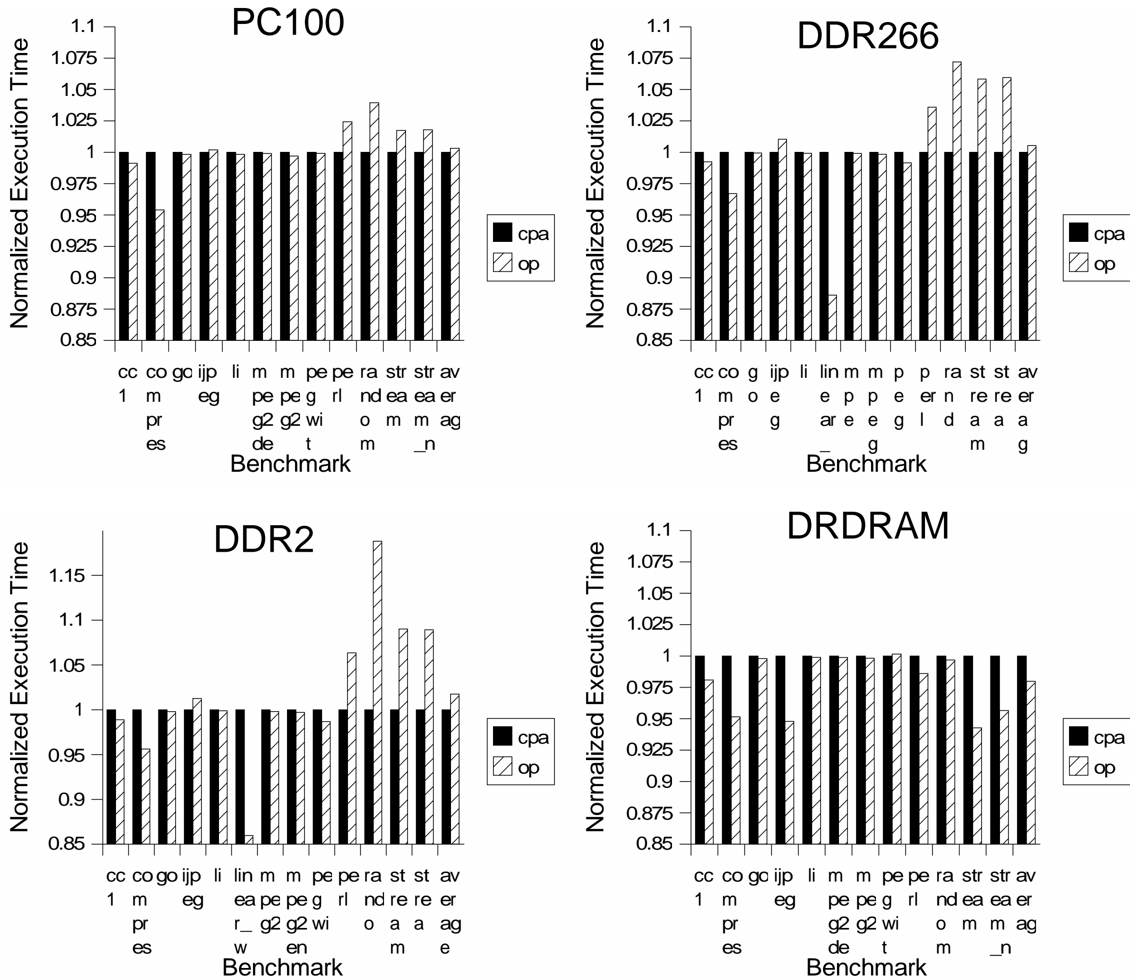


Figure 6.7: Controller Policy in Execution Driven Simulations

Impact of controller policy upon the normalized execution time of these benchmarks for the close-page-autoprecharge and open-page controller policies

and DDR266 architectures perform between DDR2 and DRDRAM for the open-page controller policy, with the PC100 performing slightly better due to the slightly smaller page and bank sizes. With the small bank/page sizes of current architectures, the open page controller policy can provide higher performance on applications which have a significant amount of linear data access or data reuse. As bank/page sizes increase, or the applications become more random access with a larger data set, the advantages of using the open-page policy decrease.

Figure 6.8 shows the impact of controller policy upon the trace driven simulations. The trend is similar to that of the execution driven simulations in that the smaller the bank

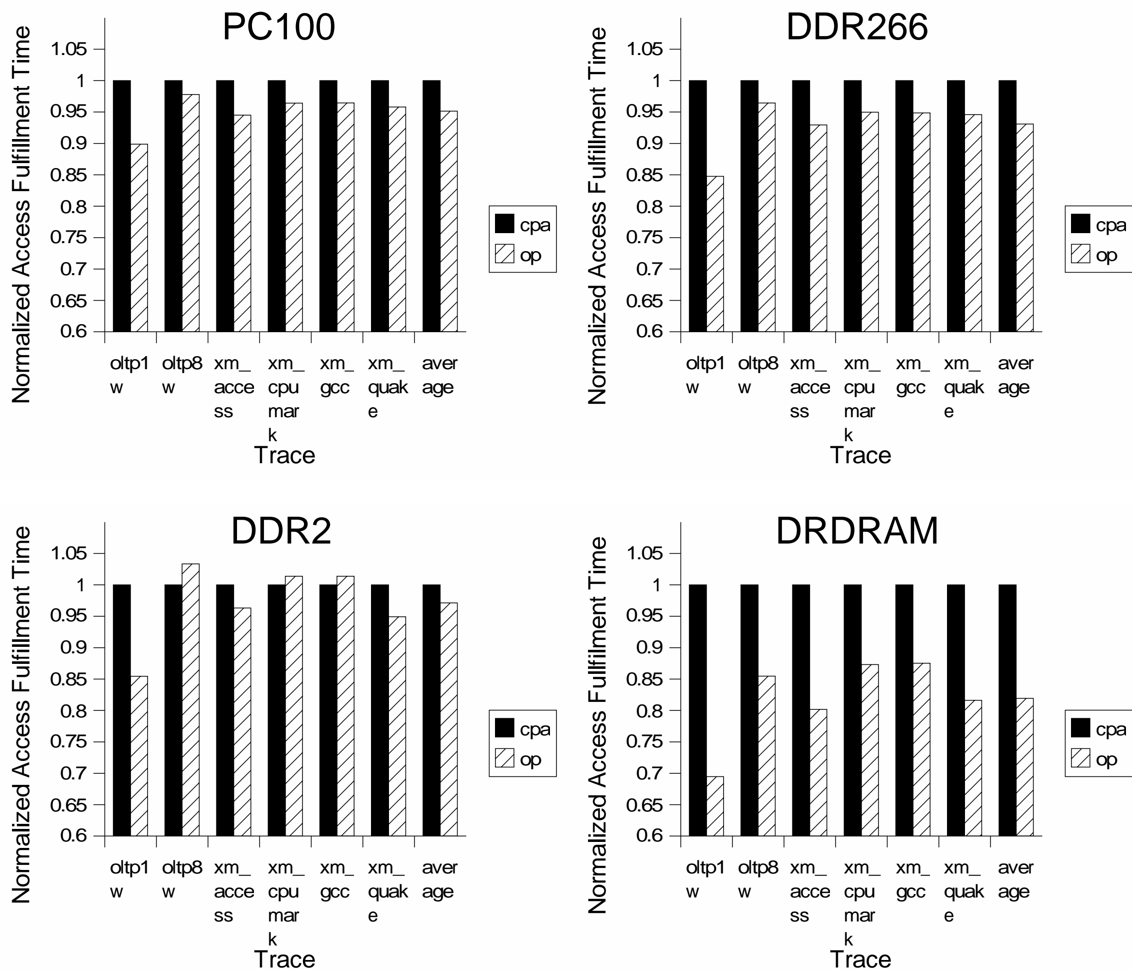


Figure 6.8: Controller Policy in Trace Driven Simulations

Impact of controller policy upon the normalized access fulfillment time of these DRAM access traces for the close-page-autoprecharge and open-page controller policies

and page size, the better the open-page controller policy performs. However, the absolute values are very different. The performance of the trace-driven simulations using the open-page controller policy is comparatively better than that observed for the execution-driven simulations. This is because the trace driven simulations are not sensitive to access latency due to the lack of feedback from an access to the address generation of subsequent accesses. This is a consequence of the fact that, as shown in Figure 4.2, the execution of the instructions occurred prior to the DRAM access trace generation, and it is impossible to determine the inter-access timings. The time on the Y axis is referred to as access fulfillment time, rather than execution time because it illustrates the amount of time

required to fulfill all accesses gathered at the time of execution, a lower bound on actual execution.

The open-page policy performs better on the trace driven simulations than upon the execution driven simulations. This results from a number of factors. First, because the traces do not contain any inter-access timings, the access streams is compacted in time, and the data contained in the sense-amps is less likely to be invalidated by a refresh between two accesses to the same page. Second, there are a number of system-level characteristics missing from the execution driven simulations which are likely to increase the open-page performance. The execution driven simulations do not include operating system activity such as page swapping, graphics system accesses like a frame buffer copy, or hard-drive activity. All of these access types are large-granularity accesses from uncached memory which are likely to improve the open-page performance. Finally, the traces are inherently bandwidth limited and thus place a higher load upon the memory system. The fact that open-page accesses can be performed with a shorter latency and require fewer inter-access refreshes allows more accesses to be serviced in a fixed period of time. The increase in performance for the trace driven simulation is due to the increase in effective bandwidth this allows For these reasons, as well as the fact that most modern, small-system DRAM controllers utilize a limited open-page policy, the open-page policy serves as the baseline configuration for all except the cache-enhanced DRAM configurations for the remainder of these experiments.

6.6 Address Remapping

Address remapping, as discussed in Section 3.5.4 allows the processor and the DRAM address spaces to differ. This provides an advantage in that addresses which are spatially proximate in the processor address space, and therefor likely to be accessed with with high temporal locality are placed into unique DRAM banks. The goal is that items likely to be accessed in close temporal proximity, but not in the same page, are more likely to be able be accessed in a pipelined fashion because they reside in unique banks. The intent of address remapping is to reduce the number of temporally adjacent accesses going to unique rows within the same bank. This situation limits performance because two

accesses which meet those criterion must be serialized in the DRAM array. The address remapping scheme used in these experiments is a relatively simple one, the address bits above those required to select a DRAM page (or row) are reversed in order. This technique is simple and would add no latency to the path through the DRAM controller. The address remapping scheme could certainly be improved from this technique, to be specific to a memory system implementation, or possibly implemented as a hash for significant redistribution, but the simulations shown use this basic bit inversion technique.

Figure 6.9 shows the effect of address remapping on a DDR2 architecture. The

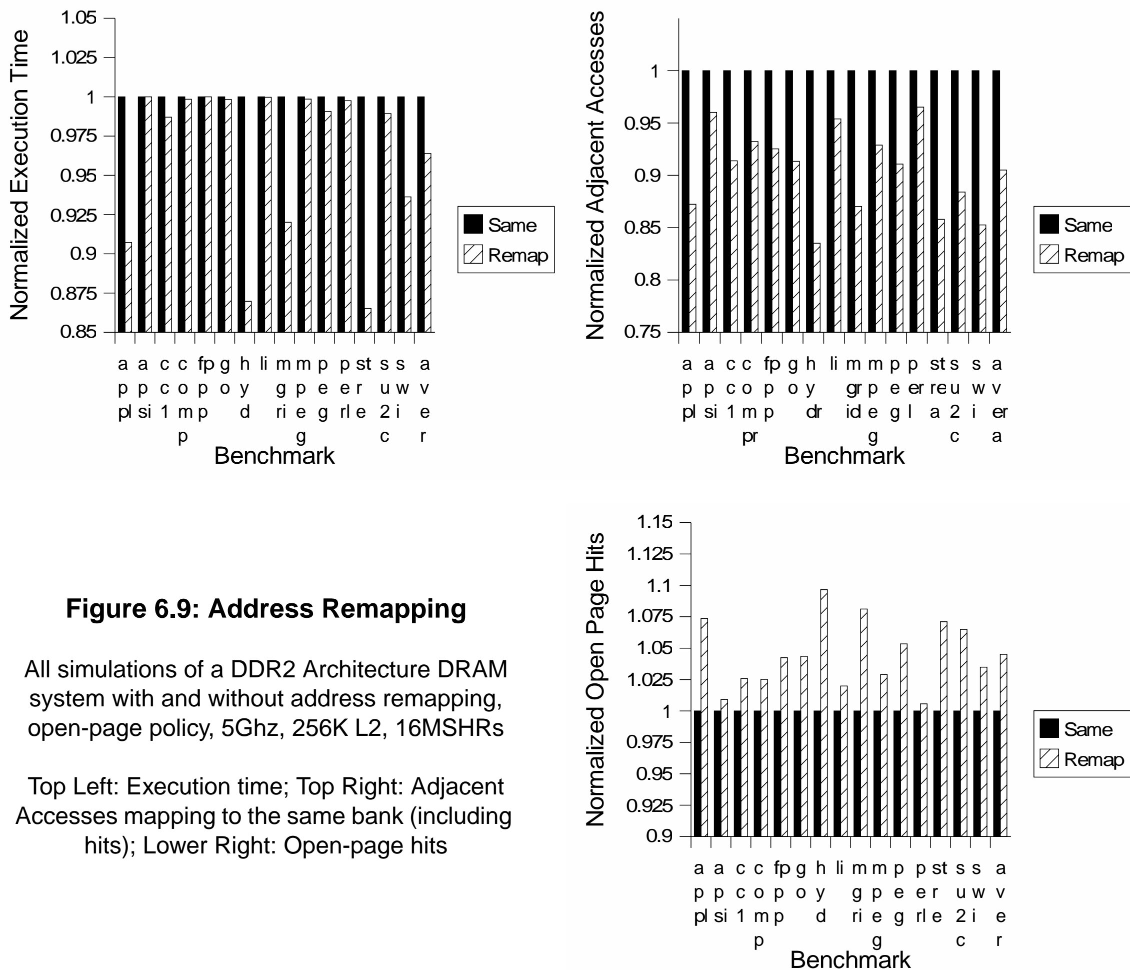


Figure 6.9: Address Remapping

All simulations of a DDR2 Architecture DRAM system with and without address remapping, open-page policy, 5Ghz, 256K L2, 16MSHRs

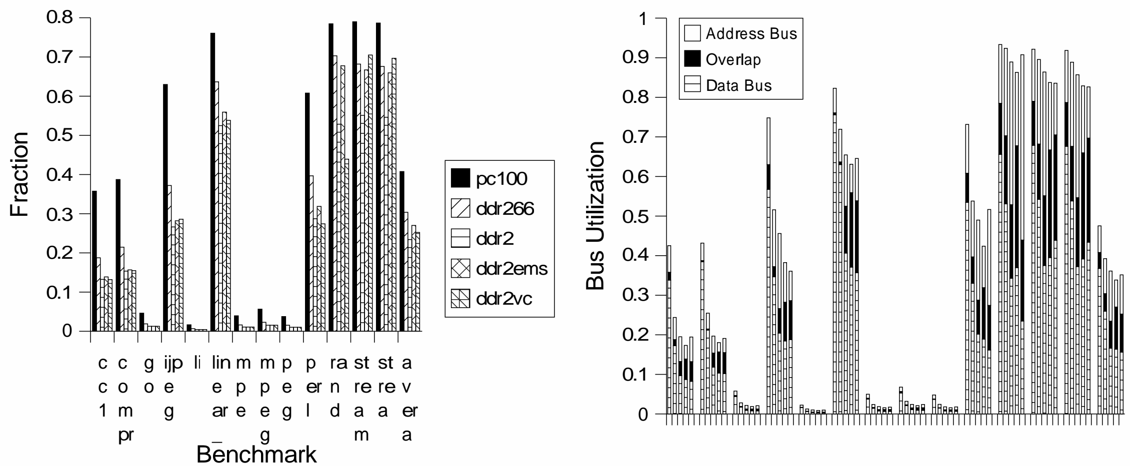
Top Left: Execution time; Top Right: Adjacent Accesses mapping to the same bank (including hits); Lower Right: Open-page hits

remapping reduces the average number of a adjacent accesses which map into the same bank by approximately 9.5%. An interesting thing to note is that included in the “adjacent accesses” are the adjacent accesses which hit in the open page. Ideally what we would like to examine is the reduction in the number of adjacent non-hit accesses only. This is certain

to be significantly larger than 9.5% because with the DDR2 architecture and the 128 cache linesizes used in these experiments, a minimum of 75% of accesses (the last three 32 Byte accesses of each group of four cumulatively accessing 128 bytes) are cache hits. Thus, the remapping technique has the potential to reduce the average number of adjacent non-sense-amp-hit accesses on the order of 36%. The end result of address remapping, and the corresponding increase in access concurrency, is that we see on-average, a reduction in execution time of 2.5%, with reductions as high as 13% for stream and hydro2d. These are both applications which alternate accesses between multiple streams of data, presumably which were within the same bank without address remapping, and which mapped into unique banks after the address remapping technique was applied. This technique is universally applicable to all DRAM technologies, DDR2 interface or otherwise. The only potential downsides are possible increased latency through the controller for schemes more complex than the one shown here, or an increase in the area of the DRAM controller state machine.

6.7 Bus Utilization

Bus utilization refers to the fraction of execution time that the bus is occupied by a transaction. It is inappropriate to examine bus utilization for traces because they do not contain accurate timing information, and thus the bus utilization is invalid. In most DRAM architectures there are two components to the DRAM bus, the address and the data signals. In the case of a Direct Rambus (DRDRAM) channel, there are three components, row, column and data. Figure 6.10 shows bus utilizations for the DRAM architectures. The top left figure shows strictly data bus utilization, the top right side figure shows utilization of both the address and the data signals on the bus, thus is inclusive of the data on the left. The final figure on the bottom right shows the bus utilization for the perl95 benchmark for the PC100, DDR266, DDR2, DDR2EMS and DDR2VC DRAM architectures in that order. The reason for discrepancy between this figure and the perl results shown in the above figures is that the perl95 benchmark is run with a larger data set (524278546 instructions) and than the perl (2000) benchmark (73527235 instructions) and the two are different binaries.



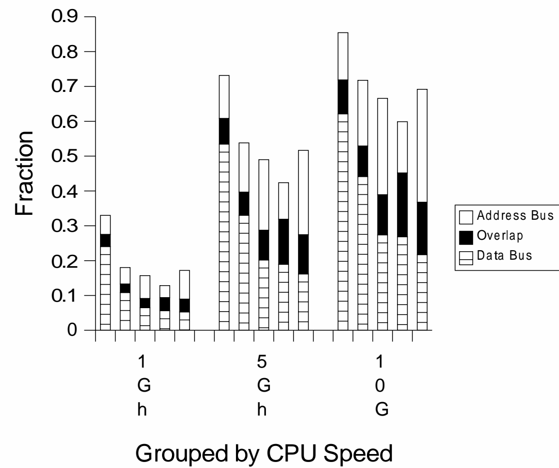


Figure 6.10: Bus Utilization

For execution driven simulations only.
Top Left: data bus utilization only; Top Right: the same data, but also includes the address bus utilization and overlap; Bottom Right: Perl95 long execution run

For both right hand figures the DRAM architectures are ordered the same as the top left: PC100, DDR266, DDR2, DDR2EMS, DDR2VC

As can be seen from Figure 6.10, there are two characteristic behaviors. Benchmarks either utilize all the bandwidth that the memory system makes available to it, as is the case with linear_walk, random_walk and stream, or the benchmark makes little use of the memory system. For the first class of applications, increasing the bandwidth to the memory system can improve performance. For the second class of applications, increasing the bandwidth will not significantly reduce execution time. Regardless of the behavior of the application, reducing the latency of the average memory access will improve performance of any benchmark.

The perl95 graph shows the typical effects of the processor frequency upon bus utilization. As the frequency of the processor is increased, it causes a corresponding increase in the utilization of the bus. This is due to a number of factors. The reduced inter-access times required to complete arithmetic, or control instructions between memory

accesses mean that the processor spends less time on non-memory instructions. With a faster processor, a larger number of instructions are able to be processed during the latency of a memory access, and the memory system is more able to find memory parallelism within this larger instruction window. Together this results in increased bus utilization, to transfer a fixed number of bytes, over a shorter period of time.

6.8 Cache Hit Rates

Cache hit means different things in the context of each of the unique architectures. For PC100, DDR266, DDR2 and DRDRAM it means hitting in an open sense-amp page while a controller is managing the device using an open-page policy. For DDR2EMS and DDR2VC it means hitting in the SRAM cache located on the DRAM die. Each of these various types of “cache hits” is included on a single set of graphs to allow comparison between the various DRAM architectures. In all cases, a “cache” hit allows the access to complete at the CL latency only.

Figure 6.11 shows the cache hit rates for each of the benchmarks, at a 5ghz processor speed. The cache hit rates are, on average, better for the DDR2 architectures for

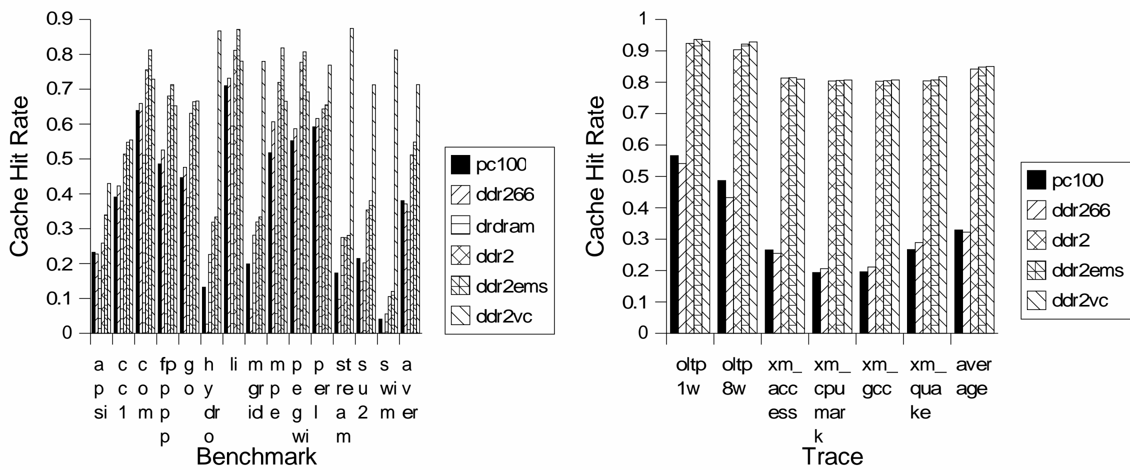


Figure 6.11: Effective Cache Hit Rates

Left: Execution driven simulations assuming a 5Ghz, 8-way superscalar, 256KB L2, 16MSHRs configuration; Right: Trace driven simulations

a number of reasons. Primary among these is the fact that, for the DDR2 interface devices,

this data reflects DRAM level accesses, and because of the limited burst size of 4, each 128-Byte controller access results in four 32-Byte DRAM level accesses, implying a lower bound sense-amp hit-rate of 75%. Secondly, the results for the DDR2 (and DDR2EMS and DDR2VC) simulations utilize the address remapping technique described in Section 6.6. This technique more fully utilizes all of the banks in the device, allowing better cache hit rates. The results in Figure 6.11 show that the cache-hit rate of an application is highly dependent upon the application behavior. Examine the difference between mpeg2enc, which has a highly linear access pattern, and stream, which has a highly associative access pattern, on the DDR2EMS and DDR2VC architectures. Highly associative in this context means that the benchmark, while accessing data, streams concurrently from multiple different arrays. This can lead to significant conflict misses if the streams are mapped to the same bank, and the DRAM architecture is direct mapped in its cache implementation. For stream, the DDR2VC architecture has a much higher cache hit-rate than any other architectures. For mpeg2enc, the DDR2EMS architecture, which contains only direct mapped cache lines, has a higher cache hit rate. Overall, the DDR2VC architecture provides the best cache hit rates, due to the associativity of its cache implementation.

6.9 Access Concurrency

The relationship between a pair of DRAM accesses adjacent in time is significant. Primarily because accesses directed at unique banks can be pipelined to execute concurrently, while accesses directed at the same bank must be performed sequentially. This condition where two sequential accesses map to the same bank requiring that they be performed sequentially is also referred to as bank conflict. Further, there is some latency involved in “turning the bus around” or transitioning from doing a read to a write. This typically means that the data signals of the bus must idle for one cycle when making such a transition.

One approach to reducing the number of adjacent access pairs mapping to unique rows in a common bank, i.e. bank conflict, is to remap the address bits from the processor address space to the DRAM address space. This remapping is sometimes implemented and referred to as hashing, and attempts to combat the locality of reference. Other

techniques for changing the access stream to improving the bus utilization involve re-ordering the accesses, clustering the reads and writes together, or coalescing neighboring reads. These techniques can significantly increase the complexity of the controller, but may provide improved performance [Carter 99].

Figure 6.12 shows the fraction of adjacent access pairs which map to a common

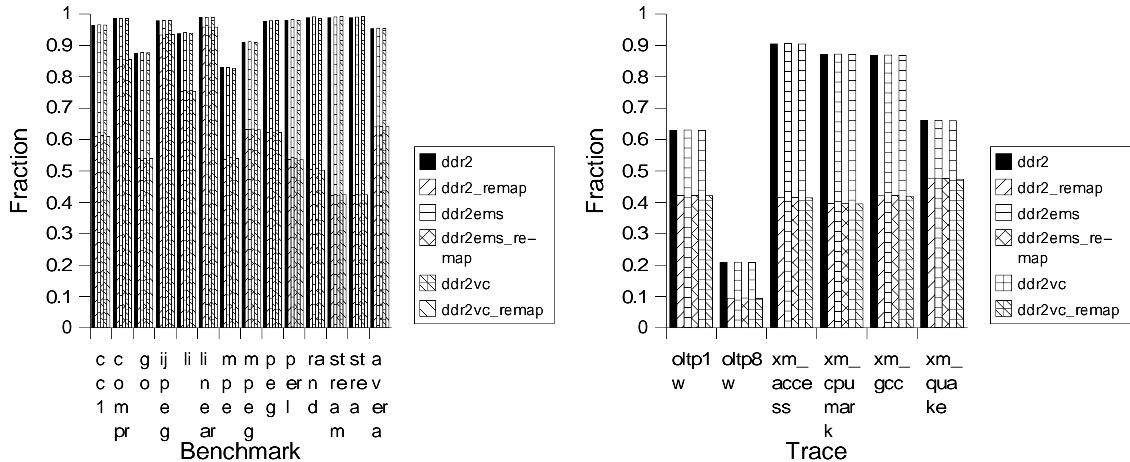


Figure 6.12: Access Adjacency

Both show the fraction of adjacent accesses which map into the same bank for multiple DDR2 architectures both with and without address space remapping
Left: execution driven simulation; Right: trace driven simulations

bank. For this figure, the three DDR2 interface architectures are shown both with and without address remapping to illustrate the effectiveness of this technique. The data presented in Figure 6.12 reflects controller level accesses. Were this not the case, a higher access adjacency rate, both with and without remapping, would be observed due to burst size limits. The number of sequential controller level accesses which map into the same DRAM bank is calculated by taking the number of adjacent DRAM accesses which map into the same bank, and subtracting from that three times the number of accesses which are split by the controller (i.e. the number of sequential cache line fill accesses which are guaranteed to be page-hits) and dividing the result by the number of accesses split by the controller. This calculation results in, the number of temporally adjacent controller level accesses, which map into the same DRAM page.

The reduction in adjacent access pairs due to address remapping can be easily observed in Figure 6.12. For the execution driven simulations, the average reduction in

adjacent accesses mapping to the same bank is consistently from 95% to 64%, or a reduction of 31%. For the trace driven simulations the average reduction in adjacent accesses mapping to the same bank is 69% to 37%, or a reduction of 32%. This values are consistent across all three DDR2 architectures, as controller address remapping is orthogonal to the caching technique used within the DRAM devices. Figure 6.12 includes as adjacent accesses, those accesses mapping to the same bank those adjacent accesses in which the second access is a sense-amp, row-cache, or virtual channel hit, but are not due to cache-line fills. The address remapping scheme does not intend to, and will not remap these addresses, as addresses which are in the same DRAM page in the processor address space should remain in the same page in the remapped DRAM address space. This reduction in the number of adjacent accesses leads to a corresponding reduction in execution time due to the increased ability to pipeline the adjacent accesses.

6.10 Average Latency

While bandwidth has been cited as the solution to the "Memory Wall" [Wulf95], the latency of accesses can be even more significant in determining the execution time of a task or program [Cuppu 99][Davis00a]. Bandwidth may be increased by widening the bus between the processor and the memory devices, a change which increases the system cost. This approach becomes easily implimented when memory and processing core are on the same die. Decreasing latency however requires an architectural - either system or DRAM - change, until recently this almost always involved increasing the cache size(s). The degree to which latency impacts execution time is dependent upon the level of memory access parallelism in the task. The execution time of an application which traverses a linked list or other sequentially accessed data structure is going to be highly dependent upon average latency, while an application such as stream which has effectively unlimited memory parallelism is going to be more dependent upon memory bandwidth.

Figure 6.13 shows the average DRAM access latency, in nanoSeconds, on both the execution and trace driven simulations. Latency in this figure is defined as the time from the first bus activity, whether it is explicit precharge, $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ until the final byte of data has been received at the controller. Thus the latency shown in Figure 6.13 includes

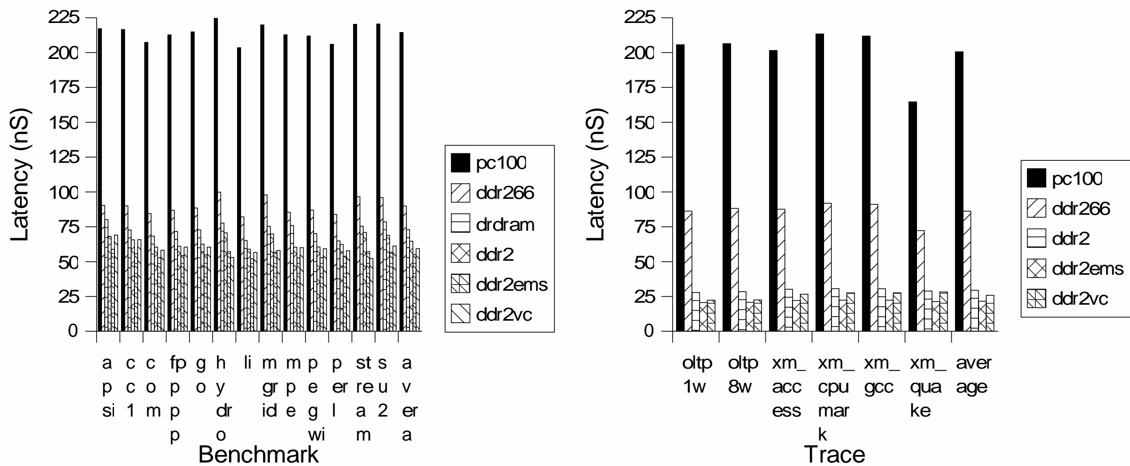


Figure 6.13: Average Latency

On the left is average latency for execution driven, on the right is average latency for trace driven. Note that access sizes are 128 bytes for PC100, DDR266 and DRDRAM, 32 bytes for DDR2, DDR2EMS and DDR2VC devices.

the transfer time of all bytes in the access. The execution driven simulations have been compensated such that all accesses are 128 Byte accesses. The trace driven simulations are split into two groups. For the PC100, DDR266 and DRDRAM accesses, each access is a 128 bytes access, and for the DDR2, DDR2EMS and DDR2VC accesses each access is a 32 bytes per access. This results from the burst size limit on the DDR2 interface devices, which requires that the memory controller translate a 128 Byte L2 cache line fill into four discrete DRAM accesses. These two groups are a source of the visible discrepancy between the DDR2 interface devices and the other DRAM architectures in the trace driven simulation graph. In some cases, the latencies shown in Figure 6.13 are larger than those in the bounds of Table 6.1. This is because the latency in Figure 6.13 includes the transfer time of all bytes in the request, whereas the latencies in Table 6.1 are described to the first 32 bit response. As can be seen, the DDR2 architectures consistently have the lowest average latency. This is without question partially due to the higher potential bandwidth of these devices, and the fact that these latencies include the time to transfer 128 bytes in all cases except DDR2 trace driven, where it includes the time to transfer 32 bytes. Of the DDR2 architectures, the DDR2EMS architecture has the lowest average latency. The average latencies of DRAM accesses for the variety of DRAM architectures are also given in Table 6.3. Again, for the trace driven simulations the PC100 and DDR266 latencies are

Table 6.3: Average Latency

DRAM Architecture	PC100	DDR266	DRDRAM	DDR2	DDR2EMS	DDR2VC
Execution Driven Average	214.47	89.92	72.99	64.65	55.08	59.39
Trace Driven Average	200.58	86.23	n/a	28.37	21.60	25.81

for 128 Byte accesses, while the DDR2, DDR2EMS and DDR2VC latencies are for 32 Byte accesses. Table 6.3 shows that the average latency, in the execution driven methodology, for the DRDRAM architecture is better than the DDR266 architecture, even though the DDR266 architecture has a higher bandwidth, and is thus able to transfer those 128 bytes required faster. Nonetheless, if latency is the determining characteristic for DRAM selection, the cache enhanced DDR2 devices, specifically DDR2EMS, provide the best latency of the simulated architectures.

6.11 Execution Time

The final metric for evaluating the performance of any computer system enhancement is the execution time of the system. This section is comprised of two different ways of looking at execution time. The first set of graphs, Figure 6.14, compares the runtime of four benchmarks across the three processor frequencies simulated, and six of the DRAM architectures modeled. Four representative benchmarks were chosen because they cover the variety of behavior observed in all benchmarks. Data was collected for all of the benchmarks in Table 6.2, but only four are shown here. The second set of graphs, Figure 6.15 and Figure 6.16 compare the execution time of all benchmarks for the three processor speeds simulated, 1Ghz, 5Ghz and 10Ghz, for all of the benchmarks simulated. The final table, Table 6.4 provides the averages from the data in both Figure 6.15 and Figure 6.16.

Figure 6.14 shows the relative execution time holding the benchmark constant, but varying the DRAM architecture and processor frequency. The four graphs show execution time for four benchmarks: cc1, stream, mgrid and pegwit. This figure shows a cross-section of the benchmarks simulated intended to show both those applications which are

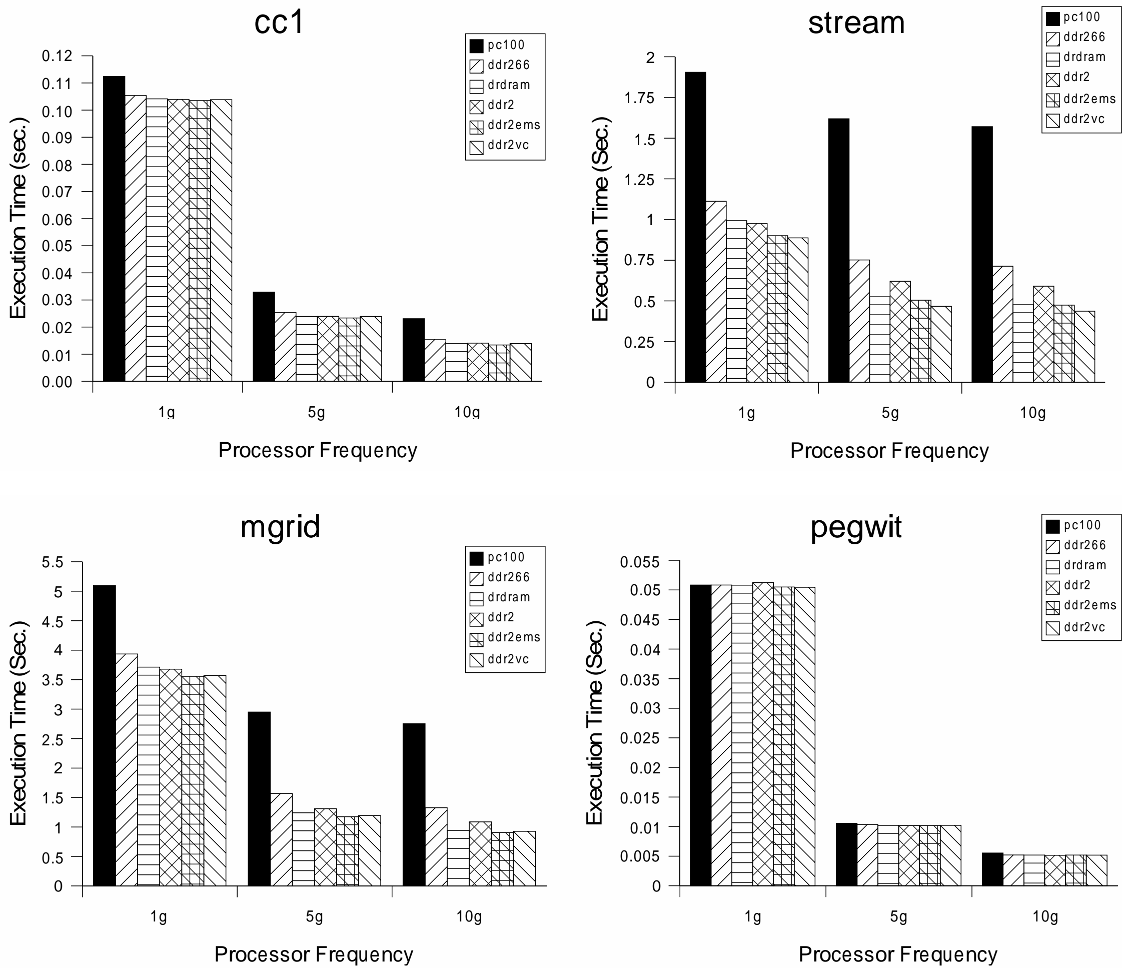


Figure 6.14: Execution time for DRAM Architectures

Each figure shows execution times for a single benchmark over varying processor speeds and DRAM architectures. The execution times are shown as in seconds.

bandwidth limited and latency limited. The stream application (top right) is the most explicitly bandwidth limited. This is shown by the fact that the performance is predictable based upon the bandwidth of the DRAM technology, and changes relatively little as processor frequency is increased. The pegwit application (bottom right) is the most explicitly latency limited. This is evidenced by the flat execution time for all DRAM architectures with a similar latency. The significant reduction in execution time as the processor frequency is increased also shows that this benchmark (pegwit) is more compute bound than I/O or memory bound. Another interesting observation from Figure 6.14 is the identification of multiple types of bandwidth limited applications. Mgrid is a bandwidth

limited application with a very linear access stream; stream is a bandwidth limited application with a highly associative (multiple streams) access pattern. Examining the performance difference between these two applications, explicitly examining the highly associative DRAM architectures (DRDRAM and DDR2VC) versus the direct mapped DRAM architectures (DDR2 and DDR2EMS) we see that for mgrid the direct mapped architectures supply higher performance, while for stream the associative DRAM architectures supply higher performance. Each of these dichotomies: bandwidth versus latency bound; associative versus linear accesses; and compute versus memory bound serve as an axis in the space which we can locate all applications. Understanding the behavior of an application is essential when using benchmarks to compare the performance of any component of a computer system. These three axis are very indicative in the case of primary memory analyses.

Figure 6.15 shows the normalized execution times, as well as average normalized

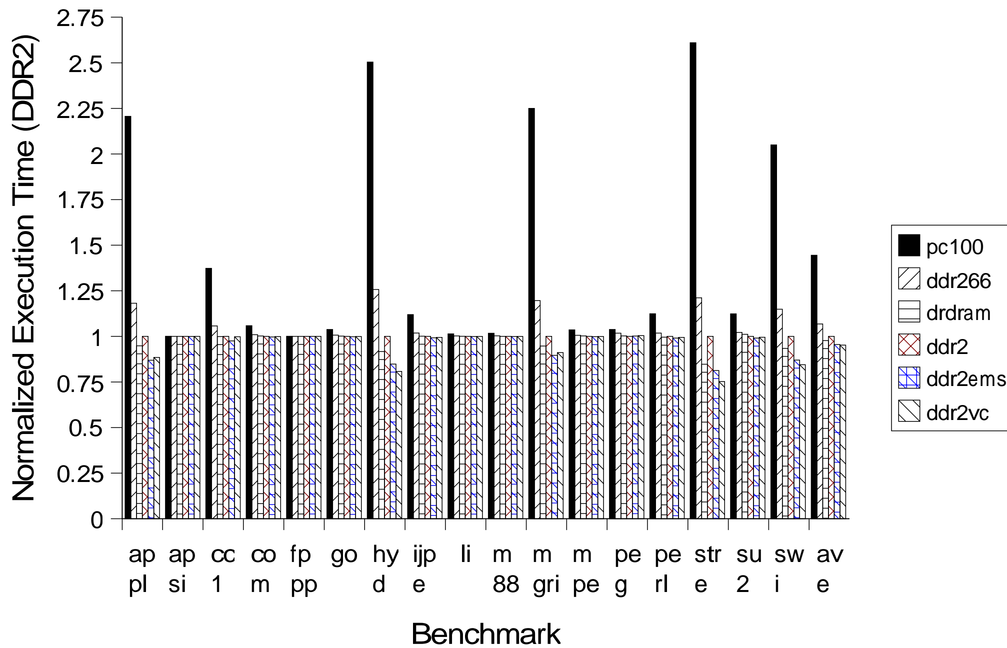


Figure 6.15: Normalized Benchmark Execution Times at 5Ghz

Simulations assume 5Ghz 8-way superscalar processor; 256 Kilobyte L2; 16 MSHRs for all DRAM architectures

execution time, for a variety of applications at 5Ghz. Figure 6.16 shows the normalized execution times for a variety of applications at 10Ghz and 1Ghz. In general, these three

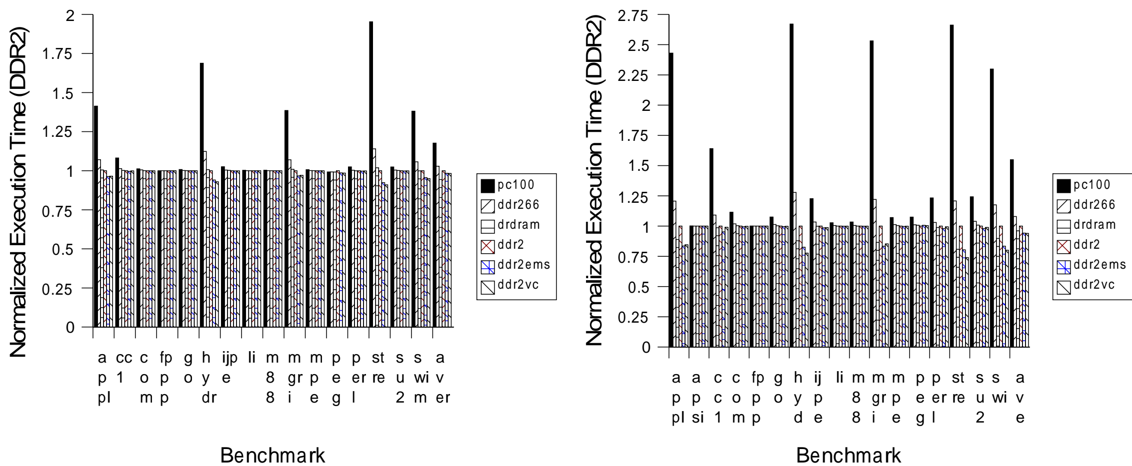


Figure 6.16: Normalized Benchmark Execution Times at 1Ghz and 10Ghz

Simulations assume 8-way superscalar processor; 256 KByte L2; 16 MSHRs for all DRAM architectures
Left: 1Ghz processor; Right: 10Ghz processor

graphs show that as the processor speed increases, the DRAM architecture choice has a slightly larger impact upon the execution time. This is especially true in the case of the latency limited applications, which are able to extract more parallelism from a larger number of instructions which are executed during the latency of a DRAM access. Table 6.4 shows the average normalized execution times from the benchmarks shown in

Table 6.4: Average Normalized Execution Time

DRAM Architecture	PC100	DDR266	DRDRAM	DDR2	DDR2EMS	DDR2VC
1g	1.23	1.04	1.00	1.00	0.98	0.98
5g	1.45	1.07	0.98	1.00	0.96	0.95
10g	1.55	1.08	0.96	1.00	0.94	0.94

Figure 6.15 and Figure 6.16. This table shows that the most modern of the DRAMs examined (DRDRAM, DDR2EMS and DDR2VC) continue to show higher performance — relative to DDR2 — as the frequency of the processor increases. DDR266 performs very well, relative to the other DRAM architectures, but this may be somewhat due to the best-case timing parameters (2-2-2) where DDR2 was simulated with the specified, first generation timing parameters of (3-3-3). DRDRAM performs equally to DDR2 at 1ghz,

but the performance improves as does the processor speed. @@@ This is somewhat a result of the application set used here. Most of the SPEC benchmarks have a relatively small memory footprint, and high cache hit rates. This, plus the lack of operating system activity limits the number of accesses which are directed at the primary memory system. Understanding these results depends upon an understanding of the benchmarks and traces used to generate the values in Table 6.4.

Chapter 7
Conclusion

Increasing the performance of DRAM architectures is a highly constrained problem. In a market where price drives consumption, increasing system performance concurrent with increasing costs is a questionable business strategy. There are known techniques for making DRAM which will operate in the performance domain of SRAM [Glaskowsky99][IBM00b], however the cost of these techniques price these devices out of the competitive DRAM market. Some of the techniques presented in this thesis for improving primary memory system performance increase the system cost, but the relative cost increase for performance increase is a critical question.

The dynamic nature of the DRAM market has been demonstrated by the rapid changes in the dominant technology. This thesis has explored the DRAM technologies which will dominate the market for the next 2 years. Beyond that, it is difficult to determine the market conditions which are necessary to provide a DRAM architecture with the support necessary to achieve acceptance. However, the characteristics of the devices examined in Chapter 6 which have been determined to be performance enhancements will continue to allow increased performance beyond this foreseeable timeframe.

7.1 DRAM Contributions

Most of the DRAM enhancements discussed represent additional cost in the memory system. The difficult aspect is to determine the price overhead for the value added. Where possible the price overhead has been characterized by an estimate of the additional die area required for the enhancement. Area is only one factor in the price of a DRAM device. Yield, partially based upon area, and volume are two other significant

contributors to price. As these factors are beyond our ability to determine from a specification, we have attempted to provide information which shows the value added of a DRAM enhancement, together with the additional area required, such that evaluations can be made based upon these two factors.

There are some approaches common to multiple platforms which provide additional performance. In order for a technique to be applicable to multiple architectures, it must be outside the boundaries of the DRAM device. Therefor, system or controller level approaches are those which can frequently be applied to multiple architectures. Two system level techniques were described in Section 2.2. Additionally, many techniques in the controller, such as access coalescing and reordering, can provide performance improvements regardless of the DRAM technology used.

Address remapping at the memory controller level is another technique which shows substantial improvements in memory system performance regardless of the DRAM architecture. The performance gains achievable are dependent upon the number and size of the banks in the memory system. This technique has cost only at the level of the memory controller which must contain additional hardware to perform the remapping. For the technique used to generate the results in Section 6.6, there was no additional controller overhead, as the technique simply swapped the order of the address wires. A more complex higher performance technique, such as hashing addresses, may however require additional logic on the memory controller. Even with this minimalist approach, Section 6.6 shows that significant gains can be made by remapping the processor address space to a unique DRAM address space. Additionally, as the size of memory devices increase, and correspondingly the size of memory banks increase, the impact of memory address remapping will increase. The results shown in Figure 6.9 suggest that some address remapping technique should be employed in memory controllers currently under design. Address remapping is currently used in some memory controllers, and has been examined in the past for interleaved asynchronous DRAM systems. The address remapping results in this thesis suggest that a productive area of research might be further exploring other remapping techniques beyond the rather simplistic technique employed here.

As fabrication limits increase, the capacity of the DRAM devices in production also increase. This drives the costs per bit of successive generations of DRAM devices

down, providing the consumer with more memory at a reasonable cost. However, the user is then constrained to purchase larger devices. This increases the small end of the feasible systems, as well as increasing the smallest possible incremental system upgrades which can be made. Further, to keep area and costs small, the number of sense-amp bits per DRAM bit has gradually decreased as device sizes have increased. This reduces the performance of the DRAM devices by reducing the amount as well as number of lines of effective cache in the primary memory system.

Increasing cache line sizes does not always improve the performance of the system. This is important when considering how increasing the DRAM page size, a default in subsequent generations of DRAM, will affect the performance of applications on these larger DRAM devices. The decrease in performance associated with an increase in linesize is demonstrated in the portion of Figure 6.11 covering the cache hit rates for trace driven simulations, which shows that the PC100 configuration has a higher average cache hit rate than the DDR266 configuration. In the case of applications which have a highly random access pattern, such as transaction processing workloads, the line in the cache is very rarely reused. This is especially true of direct mapped caches, which experience significant amounts of conflict misses. In the context of the two cache enhanced DRAM architectures which were examined, the EMS devices are direct mapped with longer line sizes, and thus provide lower performance on random access applications. The VC caches target specifically these applications, with shorter linesizes — 1/4 page — and higher associativity. A design superior in performance to either of these might place four cache lines, of full page size, between the sense amplifiers and the column decoders. Unfortunately, such a device would have a significantly higher area overhead and cost than either DDR2EMS or DDR2VC devices. As DRAM sizes continue to increase, generating a corresponding increase in page size, the issue of cache-hit performance based upon DRAM page size will become more important, potentially motivating a higher number of banks per device than would be minimal in area.

The difference between trace driven and execution driven simulations is significant, especially in regards to the L2 cache simulations. One result of these experiments is to demonstrate that it is not accurate to attempt to gather timing information about DRAM memory system changes from trace driven simulations. Trace

driven simulation is sufficient with regard to measuring the effects upon cache performance and access adjacency of controller policy (including remapping) changes, but not upon the timing of the execution. This is due to the limitations discussed in Section 4.2.2 which involve the lack of accurate inter-access timing once the memory system configuration has been changed. The fact that trace driven simulations cannot accurately predict the performance impact of primary memory system changes increases the amount of time required for evaluating variations upon DRAM configurations. The alternative method proposed here, execution driven simulation, is significantly more compute intensive and requires a longer period of time to complete equivalent simulations than using the trace driven methodology.

The limit upon the burst-size for the DDR2 interface increases the loading upon the address bus by increasing the number of $\overline{CAS}$ packets which must be transmitted for a fixed number of data bytes. While most current microprocessors (Pentium; PentiumII; Celeron; Katmai; Coppermine; K6) utilize a 32-Byte linesize on their lowest level of cache, some more recent microprocessors (K7) have a 64-Byte linesize on their lowest level of cache. The specification on the DDR2 interface which limits the burst-size to only granularities of four is a shortsighted decision which will increasingly impact the DRAM bus loading as average access granularities increase. In designing an architecture for the future generations of microprocessors, the specification should be flexible such that the increasing capabilities of microprocessors do not cause a degradation in associated aspects of the system. The limitation on DDR2 interface devices which requires all accesses be 32-Byte accesses will certainly require that in the future some devices perform multiple accesses for each and every cache line-fill. This necessity increases the loading upon the address bus. In addition to the loading on the address bus, the increased number of $\overline{CAS}$ packets crossing the bus expends additional, unnecessary power. The impact of this increased loading is not certain, but Figure 6.10 shows that the pure DDR2 devices have a comparatively lower data bus utilization than equivalent architectures.

An important discovery of this work is the distinction between bandwidth and latency dependent applications. This difference can be most easily observed by examining Figure 6.14, which shows the differences between applications, and the relative dependence upon peak bandwidth or access latency. The bandwidth dependent

applications have a flat performance for all of the DRAM technologies which have equivalent interface bandwidth, and drop off rapidly with decreasing bandwidth. The latency dependent applications have relatively flat performance for all DRAM technologies with comparable latencies (DDR266, DDR2, DDR2EMS and DDR2VC) with a higher execution time for those technologies (DRDRAM and PC100) with higher access latencies. This dichotomy does not cover all applications, as there is of course a full spectrum of applications ranging between the extremes, but this helps explain the different conclusions which can be drawn on DRAM performance based upon the benchmark set used for experimentation.

As device sizes and bank sizes increase, the open-page policy becomes less useful because the likelihood of an access being in an open-page reduces. This assumes a fixed workload or application, and the application determines where precisely the performance of the open-page policy falls behind that of the close-page-autoprecharge policy. The downside of the open-page policy, the precharge incurred in a page-miss, is removed by adding SRAM cache to the DRAM die. The importance, and performance advantage, of this SRAM cache will become more significant as the device and bank sizes continue to increase. Thankfully, the area impact of this SRAM cache upon the DRAM die will reduce as the device and bank sizes increase.

Each of these results, taken individually provides insight into how DRAM architecture, or benchmark behavior impacts the performance of the microprocessor, based upon the memory system configuration. Taken in conjunction these results can guide the designers of future DRAM architectures insuring that they are aware of the impact of changing the characteristics of the status quo into the next generations.

7.2 Research Contributions

The examination of the underlying characteristics of DRAM architectures which determine their performance upon a classification of applications is a primary result of this research. These underlying characteristics allow prediction of the performance of future DRAM architectures. Many of the paragraphs in Section 7.1 attempt to address a single

characteristic of DRAM and allow insight into the impact of varying that characteristic on a hypothetical device.

Many aspects of the research conducted are unique and useful beyond the results presented in this thesis. One such contribution is the identification of the differences between trace driven and execution driven simulation in the context of primary memory system analysis. Trace driven simulation can be utilized only to identify time independent aspects of the memory system performance, such as cache hit rates, bank conflict, access adjacency, and the impacts of address remapping. If some method were determined to provide a trace driven methodology with accurate timing feedback to subsequent DRAM accesses it could more accurately predict the performance of a redesigned primary memory system. This might be able to bring the trace driven accuracy into parity with that of the execution driven methodology. Without this, while the trace driven methodology can provide some accurate results, it does not simulate the timing behavior of the accesses and is for that reason limited in its application.

The identification of the differences between trace driven and execution driven simulation with respect to DRAM investigations had not been examined. The assertion that trace driven simulation is as accurate and effective as execution driven simulation within the DRAM design space has been invalidated by the simulations showing the difference in results between these two approaches. Sections 6.3 and 6.7 show that there is a significant difference between the behaviors of the two methodologies as they have been presented and used.

From the simulations presented in Chapter 6, it is possible to identify a number of elements, structures or techniques, either at the controller level, or at the DRAM architecture level which provide performance improvements and will continue to do so into future generations of DRAM. One goal of this work has been to identify these elements common to multiple DRAM technologies which provide performance consistently across architecture. Cache enhancement of the DRAM device has been shown to uniformly improve performance. There are no circumstances in which this adversely affects performance. Unfortunately, adding cache to the DRAM device increases the cost and area creating a cost versus performance trade-off. Address remapping also provides universal performance improvement across all the DRAM to which it was applied by

reducing contention in a single bank for the sense-amps. Identification of aspects such as these which provide improved performance across all DRAM is more important than performance analysis of specific architectures as these characteristics or techniques are likely to continue to apply even beyond the DRAM which are currently specified.

An important contribution of this work is identification and examination of a set of characteristics which determine the DRAM system performance. These characteristics may be attributes of the interface, of the architecture, of the controller, or of the access trace, but their individual impact upon the system performance can be observed. Researchers, prior to this approach, determined how the DRAM architecture impacted microprocessor performance, but rarely delved into the characteristics determining DRAM performance. A number of characteristics of this nature, access adjacency, bus utilization, on-DRAM cache hit rates, and controller policy impact have been discussed with regard to their impact upon the DRAM performance, and consequently upon the microprocessor performance. It is important when examining the performance of a large block of the system, such as the DRAM, to examine these underlying performance determinants. Identification of many characteristics which do impact overall system performance has been a major contribution of this research.

Lastly, the methodologies used in the simulations are significant in that they are distinct from prior research. In most simulations of the microprocessor, the lower levels of the memory system are abstracted away, as is the case in the standard version of SimpleScalar, which presumes a constant latency for all DRAM accesses. The research here shows that this lowest level of the memory hierarchy does generate performance differentials, and abstracting it away damages the integrity of simulation-based research. We have attempted to explain why the decisions were made to use two unique simulation methodologies, each of which have advantages, intending to establish a precedent for future research in this domain.

7.3 Future Work

For the majority of current designs, the DRAM controller is currently located on the north-bridge device of the motherboard chipset. This configuration is in a state of

change, and future designs ranging from the single-chip solutions like the Intel Timna to high performance processors like the Alpha 21364 are being designed to exploit the latency reduction associated with bringing the DRAM controller onto the processor die. With the DRAM on the processor die, the set of constraints in examining the primary memory system architecture change significantly. The bus latencies in both directions between the microprocessor and the controller are eliminated, as are the bus synchronization delays. However the number of pins used for the interface becomes more significant. This is one reason that the Direct Rambus architecture is planned to be used for both the Timna and the 21364. Bringing the DRAM controller onto the processor die opens avenues for increased performance, but requires a novel understanding of the application to be optimized.

The increase in the number of transistors which can be placed on a single device is motivating research on embedded DRAM architectures. In much the same way that microprocessors overcame mainframes when the number of transistors which could be placed onto a single device became sufficient for a full processor core, we now are entering an era where for many small systems, the entire system, processor core, memory system, and I/O can be placed onto a single device. Initially, this system-on-a-chip (SOC) approach will only be feasible for the smallest, i.e. embedded systems, but as integration levels continue to grow, the size of the feasible system-on-a-chip will also continue to grow. The simulations and experiments which have been presented here assume the basic system architecture shown in Figure 4.1. Once the entire memory system, or even a significant portion thereof, is placed onto a single device, the constraints governing the layout of the memory system will have radically changed. Communication between the memory system and the processor core is no longer constrained by a off-chip bus. The bandwidth can be made arbitrarily large, depending upon the width of busses and thus area the designers are willing to expend upon the processor-memory interconnection. Support for access parallelism can also be expanded by partitioning the on-chip memory system into multiple banks, each with an independent controller. One avenue of future research would examine architectural solutions and controller topologies for such a system.

With the continuing increase in the number of transistors available on a single die, it is a certainty that multi-core devices, or multiprocessors on a die will be produced. The

motivation for this is similar to the system-on-a-chip approach, in that it answers the question of what to do with a transistor budget larger than that required for a single processor core, but it targets a different design space. Where the system-on-a-chip uses the available transistors to target a system which is fixed in size and of moderate performance, the Chip Multi Processor (CMP) uses the available transistors to place more than one processor core on a single die. This motivates some interesting research. The memory controller(s) of a CMP/SMP system must be able to provide significantly more bandwidth than that of a uniprocessor system. The conventional server approach to this problem is to increase the width of the DRAM bus. A more expensive, but potentially higher performance solution is to partition the primary memory into multiple address spaces each controlled by a unique memory controller. This system effectively doubles the number of accesses which can be supported in parallel, but increases the cost of the system by replication. Research on how to partition the application memory space has been done in the context of multiprocessor parallel computers. This work is dominated by methods for maintaining the data required on a local processor node to eliminate the need to access data across a low-latency inter-node interconnection. However, hardware partitioning of a unified memory space between partitions with similar if not equal latencies is a novel technique which can support significantly more memory parallelism than available from even current single wide server bus memory systems.

As was shown in Section 6.6, address remapping can significantly improve performance with very little impact upon system architecture. The address remapping scheme used for those experiments was very simplistic, inverting the order of the bits above the page index. Additional performance improvements are certainly possible, and have been shown for the Direct Rambus architecture [Lin99]. This motivates investigations of other address remapping implementations which generalize to a variety of architectures. Most modern DRAM controllers are designed to interface to multiple DRAM architectures, and multiple DIMM sizes. For this type of application a general purpose address remapping scheme which adapts, dynamically allowing multiple size(s) of DIMMs in a single address space is required. Further considerations in developing address remapping schemes are the added latency through the logic (if any), and the area required on the DRAM controller die for the remapping logic.

The Virtual Channel architecture places a significant burden upon the memory controller, as discussed in Section 3.4.3 and Section 3.5.6. The many possibilities for memory controller policies, in both channel allocation and dirty channel writeback present a open opportunity for research. The same substantial set of allocation and writeback policies which have been examined for L1 and L2 caches could be explored for the VC on DRAM cache. The only publicly discussed work that has been undertaken in this field up until this point has varied the channel allocation policies between random and LRU [Davis00a]. This is only a small set of the controller choices available with a Virtual Channel Device. Similarly, the EMS caching architecture, while it does not have as many functions controlled by the memory controller as the Virtual Channel architecture, it does allows two types of write, by enabling or disabling the write to the cache line upon a write to any DRAM page. The controller policies for determining which write is applicable has yet to be explored. Determination of a strictly write stream or page by instruction or address and is an open research problem.

From this section, the observation can be made that there is a significant amount of research yet to be explored within this design space. The models and methodologies developed as part of this work will continue to be utilized in this future research.

7.4 Epilogue

Novel DRAM technologies are being proposed on a rapid basis. Many of them will never be more than a conceptual design exercise. The key is to identify those which have the elements likely to deliver performance superior to their competitors, at minimal price premium. Some new DRAM technologies seek an evolutionary approach, by changing only a few of the characteristics, and relying upon the DRAM consumers leveraging their existing knowledge base and engineering to maintain their marketshare. Other new technologies choose a revolutionary approach in the hopes that these techniques will provide performance that evolutionary devices can not match, and rely upon the consumers to be drawn to the performance. The fundamental characteristics of the DRAM array typically remain unchanged, regardless of approach, and the consumption of DRAM is determined as much by prices, thus volume, as it is by performance. The DRAM

research which is most likely to yield improvements in performance to future consumers is the identification of performance enhancing characteristics without association with a specific interface or architecture.

Appendix A
Program Listings

This appendix provides source code for the programs used in the experiments.

A.1 SDRAM Model

```
// package sdram_sim

/*
  File : sdram_ctrl

  Author : Brian Davis

  */

public class sdram_ctrl {

    //
    // Constants
    //
    static final boolean DEBUG = false;
    static final boolean debug_split = false;
    static final String PC100_SDRAM = "PC100 SDRAM";
    static final String DDR133_SDRAM = "DDR133 SDRAM";
    static final String DDR133_CAS3_SDRAM = "DDR133 cas3 SDRAM";
    static final String CLOSEAUTOPRE = "Close-Page-Autoprecharge";
    static final String OPENPAGE = "Open-Page";
    static final int DIMM_64MB = (1<<26);
    static final int DIMM_256MB = (1<<28);

    private static final int PC100 = 0x1;
    private static final int DDR133 = 0x2;
    private static final int DDR133C3 = 0x4;
    private static final int CPA = 0x10;
    private static final int OP = 0x20;

    static final int PC100_CPA = PC100 | CPA;
    static final int PC100_OP = PC100 | OP;
    static final int DDR133_CPA = DDR133 | CPA;
    static final int DDR133_OP = DDR133 | OP;
    static final int DDR133_CAS3_CPA = DDR133C3 | CPA;
    static final int DDR133_CAS3_OP = DDR133C3 | OP;

    static final int TRANS_READ = (1<<0);
    static final int TRANS_WRITE = (1<<1);
    static final int TRANS_REFRESH = (1<<2);

    //
    // Controller Parameters
    //

    boolean policy_closeautoprecharge;
    boolean policy_openpage;

    //
    // Controller Constants
    //
    String type_string;
    String policy_string;
```

```
long refresh_rate;
long refresh_cycles;
long cas_lat;
long Tras;
long Trcd;
long Trp;
long BusWidth;
double clock_period;
long ctrler_accesses = 0;
long multi_trans_accesses = 0;
double total_bytes = 0;

// Class Variable
// Instance Variable(s)
int display_mode = 0;
sdram_bus the_bus;
schedule the_schedule;
long row_shift;

//
// Constructor
//
sdram_ctrl() {
      this(PC100_SDRAM, CLOSEAUTOPRE);
}

sdram_ctrl(int type_int) {
      this((((type_int & DDR133) != 0) ?
            DDR133_SDRAM :
            ((type_int & DDR133C3) != 0) ?
            DDR133_CAS3_SDRAM :
            PC100_SDRAM),
           (((type_int & OP) != 0) ?
            OPENPAGE :
            CLOSEAUTOPRE));
}

sdram_ctrl(String type) {
      this(type, CLOSEAUTOPRE);
}

sdram_ctrl(String type, String policy) {

      if (type == PC100_SDRAM) {
          type_string = PC100_SDRAM;
          /* dram_cycles per refresh */
          /* 4096 refresh cycles / 64mS */
          refresh_rate = 1562;
          refresh_cycles = 3;
          Tras = 5;
          Trcd = 3;
          Trp = 2;
          cas_lat = 3;
          BusWidth = 8;
          clock_period = 10E-9;
          row_shift = 11;
      }
      else if ((type == DDR133_SDRAM)  ||
       (type == "ddr133_cas2_sdram")) {
          type_string = DDR133_SDRAM;
          /* dram_cycles per refresh */
          /* 8096 refresh cycles / 64mS */
          refresh_rate =1054;
          refresh_cycles = 3;
          cas_lat = 2;
          Tras = 5;
          Trcd = 2;
          Trp = 2;
          BusWidth = 16;
          clock_period = 7.5E-9;
          row_shift = 12;
      }
      else if (type == DDR133_CAS3_SDRAM) {
          type_string = DDR133_CAS3_SDRAM;
          /* dram_cycles per refresh */
          /* 8096 refresh cycles / 64mS */
          refresh_rate = 1054;
          refresh_cycles = 3;
          cas_lat = 3;
          Tras = 5;
          Trcd = 2;
```

```
            Trp = 2;
            BusWidth = 16;
            clock_period = 7.5E-9;
            row_shift = 12;
        }
        else {
            System.out.println("ERROR : Illegal type parameters to sdram_ctrl() constructor\n");
            System.exit(1);
        }

        if (policy == CLOSEAUTOPRE) {
            policy_string = CLOSEAUTOPRE;
            policy_closeautoprecharge = true;
            policy_openpage = false;
        } else if (policy == OPENPAGE) {
            policy_string = OPENPAGE;
            policy_closeautoprecharge = false;
            policy_openpage = true;
        } else {
            System.out.println("ERROR : Illegal policy parameter to sdram_ctrl() constructor\n");
            System.exit(1);
        }

        the_bus = new sdram_bus(this);
        the_schedule = new schedule(this);

        // End constructor
  }

  public boolean addDevice(int dev_size) {
        String str = null;
        if (dev_size == DIMM_64MB) {
            str = sdram_device.DIMM_64MB;
        } else if (dev_size == DIMM_256MB) {
            str = sdram_device.DIMM_256MB;
        } else {
            System.out.println("ERROR : SDRAM device of size "+ dev_size +
               "could not be created in " + type_string +
               " Environment\n");
            return false;
        }
        return addDevice(str);
  }

  public boolean addDevice(String dev_type) {
        if ((dev_type == sdram_device.DIMM_64MB) &&
            (type_string == PC100_SDRAM)) {
            the_bus.addDevice(dev_type);
        } else if ((dev_type == sdram_device.DIMM_256MB) &&
           ((type_string == DDR133_SDRAM) ||
            (type_string == DDR133_CAS3_SDRAM))) {
            the_bus.addDevice(dev_type);
        } else {
            System.out.println("ERROR : SDRAM device "+ dev_type +
               "could not be created in " + type_string +
               "Environment\n");
            return false;
        }
        return true;
  }

  public trans access(long time, int trans_type, long addr, int num_bytes) {
        //
        // Verify valid input
        //
        if ((num_bytes <= 0) ||
            ((trans_type != TRANS_READ) && (trans_type != TRANS_WRITE))) {
            System.out.println("ERROR : Illegal parameters to access()"+
               " in controller");
            System.exit(1);
        }
        if (!the_bus.addrMapped(addr)) {
            System.out.println("ERROR : Address "+Long.toHexString(addr)+
               " not contained within Memory System");
            return null;
        }

        // Update time
        if (time > the_schedule.currentTime()) {
            the_schedule.advanceCurrentTime(time);
        }
```

```
        ctrler_accesses++;
        total_bytes += num_bytes;

        //
        // Verify that this transaction only spans a single device/bank
        //

        trans this_trans = null;
        long last_addr = addr + (num_bytes - 1);
        if ((addr >> row_shift) == (last_addr >> row_shift)) {

            // Create Transaction
            this_trans = new trans(trans_type, addr, num_bytes);

            // Initiate Access on channel
            the_bus.access(this_trans);

            // Schedule Transaction
            the_schedule.schedTrans(this_trans);

            // Update bus Timings
            // the_bus.updateBusTimings(this_trans);
        } else {
            multi_trans_accesses++;

            this_trans = split_access(trans_type, addr, num_bytes);

            // return null;
        }

        return this_trans;
}

private trans split_access(int type, long start_addr,
         int num_bytes) {
        trans this_trans = null;
        long last_addr = start_addr + (num_bytes - 1);

        if (debug_split) {
            String str = "DEBUG(drd_ctrl) : Source of Split Access\n";
            str += "addr = "+Long.toHexString(start_addr)+"\n";
            str += "loc_bytes = "+num_bytes+"\n";
            System.out.println(str);
        }

        long local_addr = start_addr;
        while (local_addr < last_addr) {
            sdram_device local_dev = the_bus.devForAddr(local_addr);
            sdram_bank local_bank = local_dev.whichBank(local_addr);
            int local_row = local_bank.rowIndex(local_addr);
            long row_end_addr = local_bank.rowEndAddr(local_row);
            long to_end_of_row = (row_end_addr - local_addr) + 1;
            long to_end_of_trans = (last_addr - local_addr) + 1;
            int loc_bytes = (int) java.lang.Math.min(to_end_of_row,
             to_end_of_trans);

            if (debug_split) {
        String str = "DEBUG(drd_ctrl) : Split Access\n";
        str += "row = "+local_row+"\n";
        str += "addr = "+Long.toHexString(local_addr)+"\n";
        str += "loc_bytes = "+loc_bytes+"\n";
        System.out.println(str);
            }

            // Create Transaction
            this_trans = new trans(type, local_addr,
           loc_bytes);

            // Initiate Access on channel
            the_bus.access(this_trans);

            // Schedule Transaction
            the_schedule.schedTrans(this_trans);

            local_addr += loc_bytes;
        }
        return this_trans;
}

public long maxAddr() {
```

```
        return the_bus.maxAddr();
    }

    public boolean endSimulation() {
        return the_schedule.endSimulation();
    }

    //
    // toString
    //
    public void printYourself() {
        System.err.println(this.toString(0xFFFFFFFF));
    }

    public String toString() {
        return this.toString(display_mode);
    }

    public String toString(int dm) {
        String str = new String();

        str += "sdram_ctrl ["+super.toString()+"]\n";
        str += "\tDRAM Type\t\t: "+type_string+"\n";
        str += "\tCtrler Policy\t\t: ";
        if (policy_closeautoprecharge) {
            str += "Close-Page-Autoprecharge\n";
        } else if (policy_openpage) {
            str += "Open-Page\n";
        } else {
            str += "UNKNOWN\n";
        }

        str += "\tCAS Latency\t\t: "+cas_lat+"\n";
        str += "\ttRAS\t\t: "+Tras+"\n";
        str += "\ttRCD\t\t: "+Trcd+"\n";
        str += "\ttRP\t\t: "+Trp+"\n";
        // str += "\ttWR\t\t: "+Twr+"\n";

        str += "\tCtrler Nominal row size : "+(1 << row_shift)+"\n";
        str += "\tTime Elapsed\t\t: "+
            ((double)the_schedule.currentTime()) * clock_period+" (sec)\n";

        str += "\tController Accesses\t: "+ctrler_accesses+"\n";
        str += "\tTotal Bytes Transfered\t: "+total_bytes+"\n";
        str += "\tAvg Request size\t: "+
            (total_bytes / ctrler_accesses) + "\n";
        str += "\tMultiTrans Accesses\t: "+multi_trans_accesses+"\n";

        str += the_schedule.toString(schedule.STATS);
        str += the_bus.toString(sdram_bus.STATS);
        return str;
    }

} // sdram_ctrl

====================================================================================================

// package drdram_sim

/*
  File : SDRAM_BUS

  Author : Brian Davis

  */

import java.util.Vector;

public class sdram_bus {

    //
    // Constants
    //
    static final boolean debug = false;
    static final int DEBUG = (1<<0);
    static final int STATS = (1<<1);
    static final String SPACES8 = "        ";
    static java.text.NumberFormat nf;

    //
```

```
// Variables
//
Vector device_list;
sdram_ctrl the_ctrler;
// Refresh
private long access_calls = 0;
private double access_bytes = 0;
private long read_hits = 0;
private long read_misses = 0;
private long write_hits = 0;
private long write_misses = 0;
private long refresh_calls = 0;
// Display
int display_type = 0;

//
// Constructors
//
sdram_bus(sdram_ctrl ctrl_host) {
      the_ctrler = ctrl_host;
      device_list = new Vector();
}

public sdram_device addDevice(String dev_type) {
      long new_start;
      if (device_list.isEmpty()) {
          new_start = 0;
      } else {
          new_start = ((sdram_device)(device_list.lastElement())).addr_end + 1;
      }
      sdram_device newDevice = new sdram_device(the_ctrler, dev_type, new_start);
      if (newDevice instanceof sdram_device) {
          // Add to Vector
          device_list.addElement(newDevice);
      }
      return newDevice;
}

public boolean addrMapped(long addr) {
      if (device_list.isEmpty()) {
          if (debug) {
      System.out.println("sdram_bus device_list is Empty");
          }
          return false;
      }

      long first_start =
          ((sdram_device)(device_list.firstElement())).addr_start;

      long last_end = ((sdram_device)(device_list.lastElement())).addr_end;

      boolean ret_val = ((first_start <= addr) &&
         (addr <= last_end));

      if (debug && (!ret_val)) {
          System.out.println("first_start = "+first_start+
             "\nlast_end = "+last_end);
      }

      return ret_val;
}

long maxAddr() {
      long last_end = ((sdram_device)(device_list.lastElement())).addr_end;
      return last_end;
}

sdram_device devForAddr(long addr) {
      for (int j = 0 ; j < device_list.size() ; j++) {
          sdram_device dev = ((sdram_device)device_list.elementAt(j));
          if ((dev.addr_start <= addr) &&
      (dev.addr_end >= addr)) {
      return dev;
          }
      } // for
      return null;
}

public boolean access(trans this_trans) {
      //
      // Advance time
```

```
        //
        access_calls++;
        access_bytes += this_trans.num_bytes;

        //
        // Perform Transaction
        //
        sdram_device dev_for_trans;

        dev_for_trans = devForAddr(this_trans.address);

        //
        // Perform Access
        //
        if (dev_for_trans instanceof sdram_device) {
            dev_for_trans.access(this_trans);
        } else {
            System.out.println("ERROR : no device found for Transaction :\n"+
               this_trans);
            return false;
        }

        if (this_trans.read) {
            if (this_trans.SAHit)
        read_hits++;
            else
        read_misses++;
        } else if (this_trans.write) {
            if (this_trans.SAHit)
        write_hits++;
            else
        write_misses++;
        }

        return true;
    }

    int refreshAllDevices() {
        int ret_row = 0;
        refresh_calls ++;
        for (int j = 0 ; j < device_list.size() ; j++) {
            sdram_device dimm = (sdram_device) device_list.elementAt(j);
            ret_row = dimm.refreshAllBanks();
        }
        return ret_row;
    }

    //
    // toString
    //
    public String toString(int dt) {
        display_type = dt;
        return this.toString();
    }

    public String toString() {
        String str = new String();
        str += "sdram_bus ["+super.toString()+"]\n";
        str += "\tSpans Addresses\t\t: (";
        str += "0x"+java.lang.Long.toHexString(((sdram_device)(device_list.firstEle-
ment())).addr_start);
        str += " : ";
        str += "0x"+java.lang.Long.toHexString(((sdram_device)(device_list.lastElement())).addr_end);
        str += ")\n";
        str += "\tNumber of Devices\t: "+device_list.size()+"\n";
        str += "\tAccesses crossing bus\t: "+access_calls+"\n";
        str += "\tBus Access Hits\t\t: "+(read_hits + write_hits)+"\t\t";
        str += percent8Str((read_hits + write_hits)/((double)access_calls))+
            "\n";
        if (access_calls != 0) {

            long reads = (read_hits + read_misses);
            str += "\tBus Reads\t\t: "+reads+"\t\t";
            str += percent8Str(reads/((double)access_calls))+"\n";
            if (reads != 0) {
        str += "\tBus Read Hits\t\t: "+read_hits+"\t\t";
        str += percent8Str(read_hits/((double)access_calls))+"\t\t";
        str += percent8Str(read_hits/((double)reads))+"\n";
        str += "\tBus Read Misses\t: "+read_misses+"\t\t";
        str += percent8Str(read_misses/((double)access_calls))+"\t\t";
        str += percent8Str(read_misses/((double)reads))+"\n";
```

```
        } // reads

            long writes = (write_hits + write_misses);
            str += "\tBus Writes\t\t\t: "+writes+"\t\t";
            str += percent8Str(writes/((double)access_calls))+"\n";
            if (writes != 0) {
        str += "\tBus Write Hits\t\t: "+write_hits+"\t\t";
        str += percent8Str(write_hits/((double)access_calls))+"\t\t";
        str += percent8Str(write_hits/((double)writes))+"\n";
        str += "\tBus Write Misses\t\t: "+write_misses+"\t\t";
        str += percent8Str(write_misses/((double)access_calls))+"\t\t";
        str += percent8Str(write_misses/((double)writes))+"\n";
            } // Writes

        } // If Accesses

        str += "\tBytes crossing bus\t: "+access_bytes+"\n";
        str += "\tAvg Bytes / access\t: "+
            (access_bytes / access_calls)+"\n";
        if ((display_type & STATS) != 0) {
            // str += "\tRefresh Transactions\t: "+refresh_calls+"\n";


            for (int j = 0 ; j < device_list.size() ; j++) {
        sdram_device dimm = (sdram_device) device_list.elementAt(j);
        str += dimm.toString(sdram_device.STATS);
            }
        }

        if ((display_type & DEBUG) != 0) {
        }

        return str;
    }

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

} // class sdram_bus

=====================================================================================================
// package drdram_sim

/*
  File : SDRAM_BUS

  Author : Brian Davis

  */

import java.util.Vector;

public class schedule {

    //
    // Constants
    //
    static final boolean debug = false;
    static final boolean debug_shutdown = false;
    static final int MAX_LIST_SIZE = 10;
    static final int ERROR_LIST_SIZE = 250;

    static final int DEBUG = (1<<0);
    static final int STATS = (1<<1);
    static final String SPACES8 = "        ";

    //
    // Variables
    //
    sdram_ctrl the_ctrler;
    Vector addrTrans;
    Vector dataTrans;
    long reads_sched = 0;
    long writes_sched = 0;
```

```
long refr_sched = 0;
long total_sched = 0;
double total_latency = 0.0;
long adj_bank_accesses = 0;
// time variables
long current_cycle = -1;
long last_retired = 0;
long used_cycles = 0;
long addr_used_cycles = 0;
long data_used_cycles = 0;
long olap_used_cycles = 0;
// Refresh
private long last_refresh_time = 0;
private long last_refresh_iter = 0;
// display variables
int display_type = 0;
static java.text.NumberFormat nf;
trans lastTrans;

//
// Constructor
//
schedule(sdram_ctrl host_ctrler) {
      the_ctrler = host_ctrler;
      addrTrans = new Vector();
      dataTrans = new Vector();
}

long schedTrans(trans new_trans) {

      long l_pre_start = -1;
      long l_row_start = -1;
      long l_col_start = -1;
      long l_addr_start = -1;
      long l_addr_end = -1;
      long l_data_start = -1;
      long l_data_end = -1;

      if ((lastTrans instanceof trans) &&
          (current_cycle < lastTrans.start_cycle)) {
          advanceCurrentTime(lastTrans.start_cycle);
      }

      long earliest_possible = (current_cycle > 0 ) ? current_cycle : 0;

      boolean occupies_data = false;

      /*
      ** Prior transaction(s) on bus
      */
      trans prev_a_trans = null;
      try {
          prev_a_trans = (trans)addrTrans.lastElement();
      } catch (java.util.NoSuchElementException e) {
          // Do nothing it remains
          // prev_a_trans = null;
      }

      trans prev_d_trans = null;
      try {
          prev_d_trans = (trans)dataTrans.lastElement();
      } catch (java.util.NoSuchElementException e) {
          // Do nothing it remains
          // prev_d_trans = null;
      }

      /*
      ** check to see if there is already a transaction CURRNTLY using
      ** the addr portion of the DRAM bus
      */
      if ((prev_a_trans instanceof trans) &&
          (prev_a_trans.addrEnd > earliest_possible)) {

          earliest_possible = prev_a_trans.addrEnd;
      }

      /*
      ** Determine # of data cycles which will be required
      */
      long data_cycles = (long)
          (java.lang.Math.ceil(((double)new_trans.num_bytes)/the_ctrler.BusWidth));
```

```
/*
** Assume we can schedule starting at earliest_possible
*/
if (new_trans.read) {
    occupies_data = true;
    reads_sched++;
    /*
    ** Read transaction timings
    */
    if ((!new_trans.SAHit) &&
(!new_trans.BankPrecharged)) {
/*
** Must do precharge
*/
l_pre_start = earliest_possible;
l_row_start = l_pre_start + the_ctrler.Trp;
l_col_start = l_row_start + the_ctrler.Trcd;
l_addr_start = l_pre_start;
l_addr_end = l_col_start;
l_data_start = l_col_start + the_ctrler.cas_lat;
l_data_end = l_data_start + data_cycles;
    } else if ((!new_trans.SAHit) &&
       (new_trans.BankPrecharged)) {
/*
** already precharged, but must access row
*/
l_pre_start = -1;
l_row_start = earliest_possible;
l_col_start = l_row_start + the_ctrler.Trcd;
l_addr_start = l_row_start;
l_addr_end = l_col_start;
l_data_start = l_col_start + the_ctrler.cas_lat;
l_data_end = l_data_start + data_cycles;
    } else if (new_trans.SAHit) {
/*
** requested row already in open page! YEAH!
*/
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
l_addr_start = l_col_start;
l_addr_end = l_col_start;
l_data_start = l_col_start + the_ctrler.cas_lat;
l_data_end = l_data_start + data_cycles;
    } else {
System.out.println("ERROR : Logical impossibility");
System.exit(1);
    }
} else if (new_trans.write) {
    /*
    ** Write transaction timings
    */
    occupies_data = true;
    writes_sched++;
    if ((!new_trans.SAHit) &&
(!new_trans.BankPrecharged)) {
/*
** Must do precharge
*/
l_pre_start = earliest_possible;
l_row_start = l_pre_start + the_ctrler.Trp;
l_col_start = l_row_start + the_ctrler.Trcd;
l_addr_start = l_pre_start;
l_addr_end = l_col_start;
l_data_start = l_col_start;
l_data_end = l_data_start + data_cycles;
    } else if ((!new_trans.SAHit) &&
       (new_trans.BankPrecharged)) {
/*
** already precharged, but must access row
*/
l_pre_start = -1;
l_row_start = earliest_possible;
l_col_start = l_row_start + the_ctrler.Trcd;
l_addr_start = l_row_start;
l_addr_end = l_col_start;
l_data_start = l_col_start;
l_data_end = l_data_start + data_cycles;
    } else if (new_trans.SAHit) {
/*
```

```
** requested row already in open page! YEAH!
*/
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
l_addr_start = l_col_start;
l_addr_end = l_col_start;
l_data_start = l_col_start;
l_data_end = l_data_start + data_cycles;
    } else {
System.out.println("ERROR : Logical impossibility");
System.exit(1);
    }
} else {
    System.out.println("ERROR : Access neither read nor write"+
       " in schedTrans()");
    System.exit(1);
}

if (false && debug) {
    System.out.println("local variables scheduled");
}

/*
** Check for conflicts with prior accesses
** (prev_a_trans & prev_d_trans) and advance ALL l_* vars
** if conflict exists
*/

long addr_spacing = 0;
long data_spacing = 0;
long conflict_delta = 0;

/*
** Determine addr spacing from adjacency, bank & access type
*/
if ((prev_a_trans instanceof trans) &&
    (prev_a_trans.access_bank == new_trans.access_bank)) {
    // verify adjacent access spacing
    if (prev_a_trans.dataBusReqd() && new_trans.dataBusReqd()) {
adj_bank_accesses++;
    }
    addr_spacing = the_ctrler.Trp;
}

if ((prev_d_trans instanceof trans) &&
    (prev_d_trans.access_bank == new_trans.access_bank)) {
    // verify adjacent access spacing
    data_spacing = 0;
}

// Must check for time conflicts between adjacent accesses
if ((prev_a_trans instanceof trans) &&
    (l_addr_start < (prev_a_trans.addrEnd + addr_spacing))) {
    conflict_delta = (prev_a_trans.addrEnd + addr_spacing) -
l_addr_start;
}

if ((prev_d_trans instanceof trans) &&
    (l_data_start < (prev_d_trans.dataEnd + data_spacing))) {
    long data_delta = (prev_d_trans.dataEnd + data_spacing) -
l_data_start;

    if (data_delta > conflict_delta) {
conflict_delta = data_delta;
    }
}

// Case where two adjacent requests go to different rows of the
// same bank : Pg 23 IBM 256Mb DDR SDRAM datasheet
if ((prev_d_trans instanceof trans) &&
    (prev_d_trans.access_bank == new_trans.access_bank) &&
    (new_trans.SAHit == false)) {
    long adj_samebank_delta = 0;
    long pre_happens = (l_pre_start >= 0) ? l_pre_start :
(l_row_start - the_ctrler.Trp);
    if ((pre_happens >= 0) &&
((pre_happens + the_ctrler.cas_lat) < prev_d_trans.dataEnd)) {
adj_samebank_delta = prev_d_trans.dataEnd -
    (pre_happens + the_ctrler.cas_lat);
    }
```

```
            if (adj_samebank_delta > conflict_delta) {
        conflict_delta = adj_samebank_delta;
            }
        }

        if (debug) {
            System.out.println("conflict delta ("+conflict_delta+
                ") determined");
        }

        if (conflict_delta > 0) {
            if (l_pre_start >= 0)
        l_pre_start += conflict_delta;
            if (l_row_start >= 0)
        l_row_start += conflict_delta;

            l_col_start += conflict_delta;
            l_addr_start += conflict_delta;
            l_addr_end += conflict_delta;

            if (l_data_start >= 0)
        l_data_start += conflict_delta;
            if (l_data_end >= 0)
        l_data_end += conflict_delta;
        }

        new_trans.preStart = l_pre_start;
        new_trans.rowStart = l_row_start;
        new_trans.colStart = l_col_start;
        new_trans.addrStart = l_addr_start;
        new_trans.addrEnd = l_addr_end;
        new_trans.dataStart = l_data_start;
        new_trans.dataEnd = l_data_end;
        new_trans.start_cycle = l_addr_start;
        new_trans.end_cycle = (l_data_end >= 0) ? l_data_end : l_addr_end;

        //
        // by here must have set (in trans):
        // long start_cycle;
        // long end_cycle;
        // long rowStart;
        // long colStart;
        // long dataStart, dataEnd;
        //

        //
        // Update Latency metric values
        //
        total_sched++;
        long this_latency = (new_trans.end_cycle - new_trans.start_cycle);
        total_latency += this_latency;
        if (this_latency > 999) {
            System.out.println("ERROR (arbitrary) : this_latency > 999 in "+
                "schedule.schedTrans()");
        }

        if (debug) {
            System.out.println("SCHEDULING:\n"+
               new_trans.toString(0xFFF));
        }

        while(addrTrans.size() >= MAX_LIST_SIZE) {
            if (removeFromAddrTrans(addrTrans.elementAt(0)) == false) {
        break;
            }
        }

        addrTrans.addElement(new_trans);
        if (occupies_data) {
            while(dataTrans.size() >= MAX_LIST_SIZE) {
        dataTrans.removeElementAt(0);
            }
            dataTrans.addElement(new_trans);
        }
        lastTrans = new_trans;
        return l_addr_start;
}

long scheduleRefresh(long cycle_sched, int row_index) {

        long l_pre_start = -1;
```

```
        long l_row_start = -1;
        long l_addr_start = -1;
        long l_addr_end = -1;

        /*
        if ((lastTrans instanceof trans) &&
            (current_cycle < lastTrans.start_cycle)) {
            advanceCurrentTime(lastTrans.start_cycle);
        }
        */

        /*
        ** Determine start time for refresh
        */
        trans prev_a_trans = null;
        try {
            prev_a_trans = (trans) addrTrans.lastElement();
        } catch (java.util.NoSuchElementException e) {
            // Do nothing it remains
            // prev_a_trans = null;
        }

        long refresh_cycle = cycle_sched;

        if ((prev_a_trans instanceof trans) &&
            (refresh_cycle < (prev_a_trans.addrEnd + the_ctrler.Trp))) {
            //
            // Wait refresh until Addr bus avail
            //
            refresh_cycle = prev_a_trans.addrEnd + the_ctrler.Trp;
        }

        //
        // Refresh ALWAYS requires precharge, and row access only!
        //

        l_pre_start = refresh_cycle;
        l_row_start = l_pre_start + the_ctrler.Trp;
        l_addr_end = l_row_start + the_ctrler.Trcd;
        l_addr_start = l_pre_start;

        //
        // Create Refresh Transaction
        //

        trans refresh_trans = new trans(the_ctrler.TRANS_REFRESH, 0L, 0,
        l_addr_start,
        l_addr_end);

        refresh_trans.addrStart = l_addr_start;
        refresh_trans.addrEnd = l_addr_end;
        refresh_trans.preStart = l_pre_start;
        refresh_trans.rowStart = l_row_start;
        refresh_trans.start_cycle = l_addr_start;
        refresh_trans.end_cycle = l_addr_end;

        if (debug) {
            System.out.println("Refresh row "+row_index+" scheduled from "+
               l_addr_start+" to "+l_addr_end);
        }

        //
        // Add Transaction to Schedule
        //

        refr_sched++;
        while(addrTrans.size() >= MAX_LIST_SIZE) {
            if (removeFromAddrTrans(addrTrans.elementAt(0)) == false) {
        break;
            }
        }
        addrTrans.addElement(refresh_trans);

        lastTrans = refresh_trans;

        return l_addr_start;
}

long currentTime() {
        return current_cycle;
}
```

```
 boolean advanceCurrentTime(long new_time) {
       if (new_time < current_cycle) return true;
      //
      // Check for refresh
      //
      for (long r_iter = (last_refresh_iter+1) ;
           r_iter < (new_time/the_ctrler.refresh_rate) ;
           r_iter++) {

          int refr_row = the_ctrler.the_bus.refreshAllDevices();
          last_refresh_iter = r_iter;
          /*
          long refr_time = ((current_cycle <
             (r_iter*the_ctrler.refresh_rate)) ?
            (r_iter*the_ctrler.refresh_rate) :
            current_cycle);
          */
          long refr_time = (r_iter*the_ctrler.refresh_rate);
          last_refresh_time =
      scheduleRefresh(refr_time, refr_row);
      } // for r_iter

    current_cycle = new_time;

      if (debug) {
          System.out.println("schedule : cycleClock advanced to "+
             current_cycle);
      }
      return(true);
} // advanceBusTime

public boolean endSimulation() {
      if (debug) {
          System.out.println("schedule.endSimulation() called");
      }

      return retireAll();
}

public boolean retireAll() {
      //
      // Advance Bus Time to end
      //
      trans le = (trans) addrTrans.lastElement();
      if (debug_shutdown) {
          System.out.println("DEBUG(schedule) le.end_cycle = "+
             le.end_cycle);
          System.out.println("DEBUG(schedule) current_cycle = "+
             current_cycle);
          System.out.println("DEBUG(schedule) retired_to = "+
             last_retired);
      }
      advanceCurrentTime(le.end_cycle);
      retireTo(le.end_cycle);
      if (debug_shutdown) {
          System.out.println("DEBUG(schedule) le.end_cycle = "+
             le.end_cycle);
          System.out.println("DEBUG(schedule) current_cycle = "+
             current_cycle);
          System.out.println("DEBUG(schedule) retired_to = "+
             last_retired);
      }
      return true;
}

private boolean removeFromAddrTrans(Object to_be_removed) {
      if (!(to_be_removed instanceof trans)) {
          System.out.println("ERROR : item "+to_be_removed+
             " to be removed from schedule is not "+
             "a transaction object");
          return false;
      }
      trans tbr = (trans) to_be_removed;

      if ((addrTrans.size() > ERROR_LIST_SIZE) &&
          (tbr.end_cycle > current_cycle))
      {
          System.out.println("ERROR : attempt to remove transaction "+
             "prior to time advance beyond end\n"+
             "\tMight want to increase MAX_LIST_SIZE");
```

```
            System.out.println("\ttrans end_cycle\t= "+tbr.end_cycle);
            System.out.println("\tCurrent bus cycles\t= "+
               current_cycle);
            return false;
        }

        //
        // Determine usage for cycles to completion of this transaction
        //
        retireTo(tbr.end_cycle);

        addrTrans.remove(to_be_removed);

        return true;
}

boolean cycleUsedAddr(long cyc) {
        boolean ret_val = false;
        for (int j = 0 ; j < addrTrans.size() ; j++) {
            trans t = (trans) addrTrans.elementAt(j);
            // Early Term
            if (cyc < t.addrStart) {
        break;
            }
            // Cycle Used
            if ((t.addrStart <= cyc) &&
        (cyc <= t.addrEnd))
        { return true; }
        }
        return (ret_val);
}

boolean cycleUsedData(long cyc) {
        boolean ret_val = false;
        for (int j = 0 ; j < dataTrans.size() ; j++) {
            trans t = (trans) dataTrans.elementAt(j);
            // Early Term
            if (cyc < t.dataStart) {
        break;
            }
            // Cycle used
            if ((t.dataStart <= cyc) &&
        (cyc <= t.dataEnd))
        { return true; }
        }
        return (ret_val);
}

boolean retireTo(long ret_to) {
        if (ret_to < last_retired) { return false; }
        for (long cyc = (last_retired+1) ; cyc <= ret_to ; cyc++) {
            boolean cua = cycleUsedAddr(cyc);
            boolean cud = cycleUsedData(cyc);
            if (cua || cud) {
        used_cycles++;
            }
            if (cua && cud) {
        olap_used_cycles++;
            }
            if (cua) {
        addr_used_cycles++;
            }
            if (cud) {
        data_used_cycles++;
            }
        }
        last_retired = ret_to;
        return true;
}



//
// ToString
//
public String toString(int dt) {
        display_type = dt;
        return this.toString();
}

public String toString() {
```

```
        String str = new String();

        str += "Schedule ["+super.toString()+"]\n";

        double tot_sched = (double) (reads_sched + writes_sched + refr_sched);

        if ((display_type & STATS) != 0) {
            str += "\tReads Scheduled\t\t: " + reads_sched +"\t"+
        percent8Str(reads_sched / tot_sched) + "\n";
            str += "\tWrites Scheduled\t: "+writes_sched +"\t"+
        percent8Str(writes_sched / tot_sched) + "\n";
            str += "\tRefresh Scheduled\t: "+refr_sched +"\t"+
        percent8Str(refr_sched / tot_sched) + "\n";
            str += "\tAverage latency\t\t: "+
        ((total_latency / ((double)total_sched)) *
         the_ctrler.clock_period) + "\n";
            str += "\tAdjacent Bank Accesses\t: "+adj_bank_accesses+"\t"+
        percent8Str(adj_bank_accesses / ((double)total_sched)) + "\n";
            str += "\tCurrent cycle\t\t: "+current_cycle+"\n";
            str += "\tLast cycle Retired\t: "+last_retired+"\n";
            str += "\tUsed Cycles\t\t: "+used_cycles+"\n";
            double rc = (double) last_retired;
            str += "\tAddr Used Cycles\t: " + addr_used_cycles +"\t"+
        percent8Str(addr_used_cycles / rc ) + "\n";
            str += "\tData Used Cycles\t: "+data_used_cycles + "\t"+
        percent8Str(data_used_cycles / rc ) + "\n";
            str += "\tOverlap Cycles\t\t: "+olap_used_cycles+"\t"+
        percent8Str(olap_used_cycles / rc ) + "\n";
        }

        if ((display_type & DEBUG) != 0) {
            str += "\nADDRESS TRANSACTIONS:";
            for (int j = (addrTrans.size()-1) ; j >= 0 ; j--) {
        trans t = (trans)addrTrans.elementAt(j);
        str += "\n"+t.toString(0xFFF);
            }

            str += "\nDATA TRANSACTIONS:";
            for (int j = (dataTrans.size()-1) ; j >= 0 ; j--) {
        trans t = (trans)dataTrans.elementAt(j);
        str += "\n"+t.toString(0xFFF);
            }
        } // DEBUG

        return str;
    }

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }


}



=============================================================================================================

// package drdram_sim

/*
  File : SDRAM_device

  Author : Brian Davis
  **
  ** all devices are defined to be BUS-WIDE devices
  */

import java.util.Vector;

public class sdram_device {

    //
    // Constants
    //
```

```
static final boolean debug = false;
static final String DIMM_64MB = "64MByte DIMM";
static final String DIMM_256MB = "256MByte DIMM";
static final long SIZE_64MB = ((0x1L)<<26); // 2^26 bytes
static final long SIZE_256MB = ((0x1L)<<28); // 2^28 bytes
static final String SPACES8 = "        ";

//
// display items
//
static final int DEBUG = (1<<0);
static final int STATS = (1<<1);
int display_mode = 0;
static java.text.NumberFormat nf;

//
// Device specific values
//
sdram_ctrl the_ctrler;
String type_string;
long device_size;
long addr_start;
long addr_end;
int num_banks;
sdram_bank [] banks;
long bank_mask;
int bank_shift;

//
// Statistics
//
long accesses = 0;
long sa_hits = 0;
long pc_hits = 0;

//
// Constructor
//
sdram_device(sdram_ctrl ctrler_host, String type, long start) {
      the_ctrler = ctrler_host;
      type_string = type;
      int rows_per_bank = 0;
      if ((the_ctrler.type_string == sdram_ctrl.PC100_SDRAM) &&
          (type == DIMM_64MB)) {
          device_size = SIZE_64MB;
          num_banks = 8;
          bank_mask = (0x7 << 22);
          bank_shift = 22;
          rows_per_bank = 4096;
          // row_mask = (((long)0xFFF) << 11);
          // row_shift = 11;
      } else if (((the_ctrler.type_string == sdram_ctrl.DDR133_SDRAM) ||
          (the_ctrler.type_string == sdram_ctrl.DDR133_CAS3_SDRAM)) &&
         (type == DIMM_256MB)) {
          device_size = SIZE_256MB;
          num_banks = 8;
          bank_mask = (0x7 << 24);
          bank_shift = 24;
          rows_per_bank = 8192;
          // row_mask = (((long)0x1FFF) << 12);
          // row_shift = 12;
      } else {
          System.out.println("ERROR : undefined device type!\n");
          System.exit(1);
      }
      // start should of course be aligned
      addr_start = start;
      addr_end = start + (device_size-1);
      long bank_size = device_size / num_banks;
      // Define banks
      banks = new sdram_bank[num_banks];
      long bank_start_addr = addr_start;
      for (int j = 0 ; j < banks.length ; j++) {
          banks[j] = new sdram_bank(bank_size,
            rows_per_bank,
            bank_start_addr);
          bank_start_addr += bank_size;
      }

      if (debug) {
          System.out.println("SDRAM device "+type+" created\n"+
```

```
            "\taddr_start = "+addr_start+"("+
            Long.toHexString(addr_start)+
            ")\n\taddr_end = "+addr_end+"("+
            Long.toHexString(addr_end)+")");

      }

} // sdram_device

boolean access(trans this_trans) {
      accesses++;
      //
      // Primary purpose is to determine hit/miss status
      //

      sdram_bank this_bank = whichBank(this_trans.address);
      if (!(this_bank instanceof sdram_bank)) {
          System.out.println("ERROR : undefined bank in "+
             "sdram_device.access()");
          System.exit(1);
      }
      this_bank.accesses++;

      long row_addr = this_bank.rowIndex(this_trans.address);

      /*
      ** Check bank conditions
      */
      if (this_bank.rowInSA(row_addr)) {
          // Condition 1;
          this_trans.SAHit = true;
          this_trans.BankPrecharged = false;
          sa_hits++;
          this_bank.sa_hits++;
      } else if (this_bank.precharged()) {
          // Condition 2;
          this_trans.SAHit = false;
          this_trans.BankPrecharged = true;
          pc_hits++;
          this_bank.pc_hits++;
      } else {
          // Condition 3;
          this_trans.SAHit = false;
          this_trans.BankPrecharged = false;
      }
      this_trans.access_bank = this_bank;

      /*
      ** Update bank conditions
      */
      if (the_ctrler.policy_closeautoprecharge) {
          this_bank.prechargeBank();
      } else if (the_ctrler.policy_openpage) {
          this_bank.setOpen(row_addr);
      } else {
          System.out.println("ERROR : unknown controller policy");
          System.exit(1);
      }

      return true;
}

//
// Address mapping dependant
//

int bankIndex(long addr) {
      int index = ((int)((addr & bank_mask) >> bank_shift));
      return index;
}

sdram_bank whichBank(long addr) {
      sdram_bank ret_bank = null;
      int index = bankIndex(addr);
      if ((index >= 0) &&
          (index < banks.length)) {
          ret_bank = banks[index];
      }
      return ret_bank;
}
```

```
    int refreshAllBanks() {
          int row = -1;
          for (int j = 0 ; j < banks.length ; j++) {
              row = banks[j].refreshNextRow();
          }
          return row;
    }

    //
    // toString
    //
    public String toString() {
          return this.toString(display_mode);
    }

    public String toString(int dm) {
          String str = new String();
          str += "sdram_device ["+super.toString()+"]\n";
          str += "\tDevice type \t\t: "+type_string+"\n";
          str += "\tSpans Addresses\t\t: (";
          str += "0x"+java.lang.Long.toHexString(addr_start);
          str += " : ";
          str += "0x"+java.lang.Long.toHexString(addr_end);
          str += ")\n";
          if ((dm & STATS) != 0) {
              str += "\tDevice Accesses\t\t: "+accesses+"\n";
              double ac = (double) accesses;
              str += "\tDev SAmp Hits\t\t: "+sa_hits+"\t"+
          percent8Str(sa_hits / ac ) + "\n";
              str += "\tDev PreChg Hits\t\t: "+pc_hits+"\t"+
          percent8Str(pc_hits / ac ) + "\n";
          }
          for (int j = 0 ; j < banks.length ; j++) {
              str += banks[j].toString(dm);
          }
          return str;
    }

    private String percent8Str(double in) {
          if (!(nf instanceof java.text.NumberFormat)) {
              nf = java.text.NumberFormat.getPercentInstance();
              nf.setMinimumFractionDigits(2);
          }
          String ret_str;
          ret_str = SPACES8 + nf.format(in);
          return ret_str.substring(ret_str.length() - 8);
    }

} // CLASS sdram_device

==========================================================================================================

// package sdram_sim

/*
  File : trans

  Author : Brian Davis

  */

public class trans {

  //
  // Constants
  //
  static final int TRANS_READ = sdram_ctrl.TRANS_READ;
  static final int TRANS_WRITE = sdram_ctrl.TRANS_WRITE;
  static final int TRANS_REFRESH = sdram_ctrl.TRANS_REFRESH;


  // Class Variables
  static long trans_created = 0;
  static int display_mode = 0;

  // Instance Variable(s)

    //
    // Transaction Attributes
    int type;
    int num_bytes;
```

```
  long address;
  boolean read = false;
  boolean write = false;
  boolean SAHit = false;
  boolean BankPrecharged = false;
  Object access_bank = null;


  //
  // Overall Transaction bounds
  long start_cycle;
  long end_cycle;

  //
  // Component Times
  long preStart;
  long rowStart;
  long colStart;
  long addrStart;
  long addrEnd;
  long dataStart, dataEnd;

  long bus_cycles_reqd;

  // Constructor(s)
  trans(long addr, int bytes) {
        this(TRANS_READ, addr, bytes, -1L, -1L);
  }

  trans(int rw, long addr, int bytes) {
        this(rw, addr, bytes, -1L, -1L);
  }

trans(int rw, long addr, int bytes, long start, long end) {

  trans_created++;

  type = rw;
  if (rw == TRANS_READ)
    read = true;
  else if (rw == TRANS_WRITE)
    write = true;

  address = addr;
  num_bytes = bytes;

  start_cycle = start;
  end_cycle = end;

  //
  // Check to verify withing above [start:end] bounds
  rowStart = colStart = dataStart = dataEnd = -1L;
}

// Class Methods

//
// Instance Methods
//

  public boolean dataBusReqd() {
        if (((type & TRANS_READ) != 0) || (( type & TRANS_WRITE) != 0)) {
            return true;
        }
        return false;
  }

public boolean rowStart(long cycle) {
  //
  // Check for potential ERROR conditions
  if (rowStart != -1L) {
    System.out.println("ERROR : multiple definition of Row Start for transaction");
    return true;
  }

  if ((start_cycle != -1L) || (start_cycle > cycle)) {
    System.out.println("ERROR : Illegal definition of row given Start for transaction");
    return true;
  }

  rowStart = cycle;
```

```
    start_cycle = cycle;

    return false;
  }

  public boolean colStart(long cycle) {
    //
    // Check for potential ERROR conditions
    if (colStart != -1L) {
      System.out.println("ERROR : multiple definition of Column Start for transaction");
      return true;
    }

    /*
    if (rowHit == true)
      start_cycle = cycle;
    */

    return false;
  }

  private boolean startAt(long new_start) {
    // Check for valid start
    if ((end_cycle != 0) && (new_start > end_cycle)) {
      // ERROR ILLEGAL START
      return true;
    } else {
      start_cycle = new_start;
      return false;
    }
  }

  //
  // To String
  //
  public String toString(int new_mode) {
    display_mode = new_mode;
    return toString();
  }

  public String toString() {
    String str = new String();
    str += "Transaction ["+super.toString()+"]";

    if ((display_mode & (1<<0)) != 0) {
      str += "\n\tTrans type\t: "+((read) ? "Read" : ((write) ? "Write" : "UNKNOWN"));

      str += "\n\tAddress\t\t: " + java.lang.Long.toHexString(address);

      str += "\n\tNumber Bytes\t: "+num_bytes;

      str += "\n\tHit Status\t: ";
      if (SAHit) {
            str += "SenseAmp Hit";
      } else if (BankPrecharged) {
            str += "Precharge Hit";
      } else {
            str += "SenseAmp/Precharge Miss";
      }

      str += "\n\tTrans Span\t: ( "+start_cycle+" : "+end_cycle+" )";

      str += "\n\tPre Start\t: "+preStart;

      str += "\n\tRow Start\t: "+rowStart;
      str += "\n\tCol Start\t: "+colStart;
      str += "\n\tAddr Start\t: "+addrStart;
      str += "\n\tAddr End\t: "+addrEnd;
      str += "\n\tData Start\t: "+dataStart;
      str += "\n\tData End\t: "+dataEnd;

    }

    return str;
  } // toString

}
```

A.2 DDR2 Model

```
/*
  File : SDRAM_BUS

  Author : Brian Davis

  */

abstract class dram_ctrl {
    static final String PC100_SDRAM = "PC100 SDRAM";
    static final String DDR133_SDRAM = "DDR133 SDRAM";
    static final String DDR133_CAS3_SDRAM = "DDR133 cas3 SDRAM";
    static final String DDR2_200 = "DDR2 200Mhz DRAM";
    static final String DDR2_EMS = "DDR2 200Mhz ESDRAM-lite";
    static final String DDR2_VC = "DDR2 200MHz VC Enabled";

    static final String CLOSEAUTOPRE = "Close-Page-Autoprecharge";
    static final String OPENPAGE = "Open-Page";
    // static final String ES_OP_ALLWRITEXFER = "Open-Page All-Write-Xfer";
    static final String ES_CPA_ALLWRITEXFER =
          "Close-Page-Prechg All-Write-Xfer";
    static final String ES_CPA_NOWRITEXFER =
          "Close-Page-Prechg 100% No-Write-Xfer";

    static final int PC100 = 0x1;
    static final int DDR133 = 0x2;
    static final int DDR133C3 = 0x4;
    static final int DDR2 = 0x8;
    static final int ES_DDR2 = 0x10;
    static final int VC_DDR2 = 0x20;

    static final int POLICY_SHIFT = 8;
    static final int CPA = (0x1 << POLICY_SHIFT);
    static final int OP = (0x2 << POLICY_SHIFT);
    static final int OP_AWX = (0x4 << POLICY_SHIFT);
    static final int CPA_AWX = (0x8 << POLICY_SHIFT);
    static final int CPA_WX = (0x10 << POLICY_SHIFT);

    static final int PC100_CPA = PC100 | CPA;
    static final int PC100_OP = PC100 | OP;
    static final int DDR133_CPA = DDR133 | CPA;
    static final int DDR133_OP = DDR133 | OP;
    static final int DDR133_CAS3_CPA = DDR133C3 | CPA;
    static final int DDR2_CPA = DDR2 | CPA;
    static final int DDR2_OP = DDR2 | OP;
    static final int DDR2VC_CPA = VC_DDR2 | CPA;
    static final int DDR2VC_OP = VC_DDR2 | OP;
    static final int DDR2EMS_CPA_AWX = ES_DDR2 | CPA_AWX;
    static final int DDR2EMS_CPA_NWX = ES_DDR2 | CPA_WX;

    static final String POLICY_LRU = "Least-Recently-Used";
    static final String POLICY_RANDOM = "Random";
    static final String POLICY_ASSOC = "Limited Associativity LRU";
    static final String POLICY_P12IO4 = "Processor 12 / IO 4";

    static final int TRANS_REFRESH = (1<<0);
    // static final int TRANS_PRECHARGE = (1<<1);
    static final int TRANS_READ_NOPRE = (1<<2);
    static final int TRANS_READ_PRE = (1<<3);
    static final int TRANS_WR_NOPRE = (1<<4);
    static final int TRANS_WR_PRE = (1<<5);
    static final int TRANS_WR_NOXFER = (1<<6);
    static final int TRANS_READ =
          TRANS_READ_NOPRE | TRANS_READ_PRE;
    static final int TRANS_WRITE =
          TRANS_WR_NOPRE | TRANS_WR_PRE | TRANS_WR_NOXFER;

    static final String SPACES8 = "        ";
    static java.text.NumberFormat nf;

    abstract boolean endSimulation();

    abstract dram_trans access(long time, int trans_type,
                long addr, int num_bytes);

    abstract boolean addDevice(int dev_int);
```

```
    abstract boolean addDevice(String dev_type);

    abstract void enableRemap();

    abstract long maxAddr();

    abstract long currentTime();

}

==========================================================================================================

/*
  File : dram_trans

  Author : Brian Davis

  */

abstract class dram_trans {
    //
    // Constants
    //

    static final int TRANS_REFRESH = dram_ctrl.TRANS_REFRESH;
    static final int TRANS_READ_NOPRE = dram_ctrl.TRANS_READ_NOPRE;
    static final int TRANS_READ_PRE = dram_ctrl.TRANS_READ_PRE;
    static final int TRANS_WR_NOPRE = dram_ctrl.TRANS_WR_NOPRE;
    static final int TRANS_WR_PRE = dram_ctrl.TRANS_WR_PRE;
    static final int TRANS_WR_NOXFER = dram_ctrl.TRANS_WR_NOXFER;

    static final int TRANS_READ = dram_ctrl.TRANS_READ;
    static final int TRANS_WRITE = dram_ctrl.TRANS_WRITE;

} // class dram_trans

==========================================================================================================

// package sdram_sim

/*
  File : sdram_ctrl

  Author : Brian Davis

  */
import java.math.BigInteger;

public class ddr2_ctrl extends dram_ctrl {

    //
    // Constants
    //
    static final boolean DEBUG = false;
    static final boolean debug_split = false;
    static final boolean debug_remap = false;
    static final int DIMM_256MB = (1<<28);
    static final int DIMM_512MB = (1<<29);

    //
    // Controller Parameters
    //
    boolean policy_closeautoprecharge = false;
    boolean policy_openpage = false;
    boolean policy_noxferallwrite = false;
    boolean enable_remap = false;

    //
    // Controller Constants
    //
    String type_string = null;
    String policy_string = null;
    String vc_policy = null;
    boolean VCEnabled = false;
    boolean EMSEnabled = false;
    long refresh_rate;
    long refresh_cycles;
    long cas_lat;
    // ^^ RCHit_todata ^^
    // ^^ VCHit_todata
```

```
long Tras;
long Trcd;
long Trp;
long Twr;
// long RCHit_todata = -1;
long TvcEvict = -1;
long BusWidth;
double clock_period;
long ctrler_accesses = 0;
long multi_trans_accesses = 0;
double total_bytes = 0;

// ReMap Variables
long max_addr;
int used_bits;
long bank_mask;
long row_mask;
long index_mask;
int dev_msb, dev_lsb;
int bank_msb, bank_lsb;
int row_msb, row_lsb;
int debug_prints = 0;

// Class Variable
// Instance Variable(s)
int display_mode = 0;
ddr2_bus the_bus;
ddr2_sched the_schedule;
long row_shift;
int max_cycles_per_trans;
boolean mem_changed = true;
//
// Constructor
//
ddr2_ctrl() {
      this(DDR2_200, OPENPAGE);
}

ddr2_ctrl(int type_int) {
      this(
           // type
           (((type_int & DDR2) != 0) ?
            DDR2_200 :
            ((type_int & ES_DDR2) != 0) ?
            DDR2_EMS :
            ((type_int & VC_DDR2) != 0) ?
            DDR2_VC :
            null),
           // policy
           (((type_int & OP) != 0) ?
            OPENPAGE :
            ((type_int & CPA) != 0) ?
            CLOSEAUTOPRE :
            //      ((type_int & OP_AWX) != 0) ?
            //ES_OP_ALLWRITEXFER:
            ((type_int & CPA_AWX) != 0) ?
            ES_CPA_ALLWRITEXFER :
            ((type_int & CPA_WX) != 0) ?
            ES_CPA_NOWRITEXFER :
            null));
}

ddr2_ctrl(String type, String policy) {

      if ((!(type instanceof String)) ||
          (!(policy instanceof String))) {
          System.out.println("ERROR : NULL parameters to "+
             "sdram_ctrl() constructor");
          System.out.println("\ttype = "+type);
          System.out.println("\tpolicy = "+policy);
          System.exit(1);
      }

      if (type == DDR2_200) {
          type_string = DDR2_200;
          /* dram_cycles per refresh */
          /* 8096 refresh cycles / 64mS */
          refresh_rate = 1581; // upd 1/14/99
          refresh_cycles = 3; //
          // posted_cas = 0; // unsimulated as of yet
          Tras = 5;//
```

```
    Trcd = 3;//
    Trp = 3;//
    Twr = 3;
    cas_lat = 3;//
    BusWidth = 16; // upd 1/14/99
    clock_period = 1/(200E6); // upd 1/14/99
    row_shift = 11; //  To split up multi-requests
    max_cycles_per_trans = 2;
}
else if (type == DDR2_VC) {
    type_string = DDR2_VC;
    /* dram_cycles per refresh */
    /* 8096 refresh cycles / 64mS */
    refresh_rate = 1581; // upd 1/14/99
    refresh_cycles = 3;
    Tras = 5;//
    Trcd = 4;//
    Trp = 3;//
    Twr = 3;
    cas_lat = 2;//
    TvcEvict = 9;
    BusWidth = 16; // upd 1/14/99
    clock_period = 1/(200E6); // upd 1/14/99
    row_shift = 11; //  To split up multi-requests
    max_cycles_per_trans = 2;
    VCEnabled = true;
    // vc_policy = ;
    vc_policy = POLICY_LRU;
}
else if (type == DDR2_EMS) {
    type_string = DDR2_EMS;
    /* dram_cycles per refresh */
    /* 8096 refresh cycles / 64mS */
    refresh_rate = 1581; // upd 1/14/99
    refresh_cycles = 3;
    EMSEnabled = true;
    Tras = 5;//
    Trcd = 3;//
    Trp = 3;//
    Twr = 3;
    cas_lat = 3;//
    // RCHit_todata = 3;
    BusWidth = 16; // upd 1/14/99
    clock_period = 1/(200E6); // upd 1/14/99
    row_shift = 11; //  To split up multi-requests
    max_cycles_per_trans = 2;
}
else {
    System.out.println("ERROR : Illegal type parameters to "+
       "sdram_ctrl() constructor\n");
    System.exit(1);
}

if ((policy == CLOSEAUTOPRE) ||
    (policy == ES_CPA_ALLWRITEXFER)) {
    policy_noxferallwrite = false;
    policy_closeautoprecharge = true;
    policy_openpage = false;
} else if (policy == OPENPAGE) {
    policy_noxferallwrite = false;
    policy_closeautoprecharge = false;
    policy_openpage = true;
} else if (policy == ES_CPA_NOWRITEXFER) {
    policy_noxferallwrite = true;
    policy_closeautoprecharge = false;
    policy_openpage = false;
    if (type != DDR2_EMS) {
System.out.println("ERROR : Illegal DRAM type / "+
   "policy pairing to "+
   "sdram_ctrl() constructor\n");
System.exit(1);
    }
}
else {
    System.out.println("ERROR : Illegal policy parameter to "+
       "sdram_ctrl() constructor\n");
    System.exit(1);
}
policy_string = policy;
```

```
        the_bus = new ddr2_bus(this);
        the_schedule = new ddr2_sched(this);

        // End constructor
    }

    public boolean changeVCAllocPolicy(String new_policy) {
        if ((new_policy == POLICY_LRU) ||
            (new_policy == POLICY_RANDOM) ||
            (new_policy == POLICY_ASSOC) ||
            (new_policy == POLICY_P12IO4)) {
            vc_policy = new_policy;
            return false;
        }
        return true;
    }

    public boolean addDevice(int dev_size) {
        String str = null;
        if (dev_size == DIMM_256MB) {
            str = ddr2_device.DIMM_256MB;
        } else {
            System.out.println("ERROR : SDRAM device of size "+ dev_size +
               "could not be created in " + type_string +
               " Environment\n");
            return false;
        }
        return addDevice(str);
    }

    public boolean addDevice(String dev_type) {
        if (dev_type == ddr2_device.DIMM_256MB) {
            mem_changed = true;
            the_bus.addDevice(dev_type);
        } else {
            System.out.println("ERROR : SDRAM device "+ dev_type +
               "could not be created in " + type_string +
               "Environment\n");
            return false;
        }
        return true;
    }

    public void enableRemap() {
        enable_remap = true;
    }

    public dram_trans access(long time, int trans_type,
              long addr, int num_bytes) {
        //
        // Verify valid input
        //
        if ((num_bytes <= 0) ||
            ((trans_type != TRANS_READ) && (trans_type != TRANS_WRITE))) {
            System.out.println("ERROR : Illegal parameters to access()"+
               " in controller");
            System.out.println("num_bytes = "+num_bytes+"/n");
            System.out.println("type = "+trans_type+"/n");
            System.exit(1);
        }
        if (!the_bus.addrMapped(addr)) {
            System.out.println("ERROR : Address "+Long.toHexString(addr)+
               " not contained within Memory System");
            return null;
        }

        // Update time
        if (time > the_schedule.currentTime()) {
            the_schedule.advanceCurrentTime(time);
        }

        ctrler_accesses++;
        total_bytes += num_bytes;

        //
        // Determine a more specific trans_type
        //
        if (trans_type == TRANS_READ) {
            if (policy_openpage)
        trans_type = TRANS_READ_NOPRE;
            else if ((policy_closeautoprecharge) ||
```

```
            (policy_noxferallwrite))
        trans_type = TRANS_READ_PRE;
            else
        System.out.println("ERROR : Unknown READ Policy in  "+
           "ddr2_ctrl.access()");
        } else if (trans_type == TRANS_WRITE) {
            if (policy_openpage)
        trans_type = TRANS_WR_NOPRE;
            else if (policy_closeautoprecharge)
        trans_type = TRANS_WR_PRE;
            else if (policy_noxferallwrite)
        trans_type = TRANS_WR_NOXFER;
            else
        System.out.println("ERROR : Unknown WRITE Policy in  "+
           "ddr2_ctrl.access()");
        } else {
            System.out.println("ERROR : Unknown Trans type in "+
               "ddr2_ctrl.access()");
        }

        //
        // Verify that this transaction only spans a single device/bank
        //

        ddr2_trans this_trans = null;
        long last_addr = addr + (num_bytes - 1);
        if ((num_bytes <= (max_cycles_per_trans * BusWidth)) &&
            (addr >> row_shift) == (last_addr >> row_shift)) {

            //
            // Munge/re-order address bits to accomplish pseudo-interleaving
            // of DRAM pages between different banks.
            //
            long dram_addr = map_procaddr_to_dramaddr(addr);

            // Create Transaction
            this_trans = new ddr2_trans(trans_type, dram_addr, num_bytes);

            // Initiate Access on channel
            the_bus.access(this_trans);

            // Schedule Transaction
            the_schedule.schedTrans(this_trans);

        } else {
            multi_trans_accesses++;

            this_trans = split_access(trans_type, addr, num_bytes);

            // return null;
        }

        return this_trans;
}

private ddr2_trans split_access(int type_param, long start_addr,
         int num_bytes) {
        ddr2_trans this_trans = null;
        long last_addr = start_addr + ((long)(num_bytes - 1));
        int type = -1;
        int open_type = -1;

        if (debug_split) {
            String str = "DEBUG(drd_ctrl) : Source of Split Access\n";
            str += "addr = "+Long.toHexString(start_addr)+"\n";
            str += "loc_bytes = "+num_bytes+"\n";
            System.out.println(str);
        }

        if ((open_type & TRANS_READ) != 0) {
            open_type = TRANS_READ_NOPRE;
        } else if ((open_type & TRANS_WRITE) != 0) {
            open_type = TRANS_WR_NOPRE;
        } else {
            System.out.println("ERROR : Logical impossiblity 02.03.00");
        }

        long local_addr = start_addr;
        while (local_addr < last_addr) {
            ddr2_device local_dev = the_bus.devForAddr(local_addr);
            ddr2_bank local_bank = local_dev.whichBank(local_addr);
```

```
	int local_row = local_bank.rowIndex(local_addr);
	long row_end_addr = local_bank.rowEndAddr(local_row);
	long to_end_of_row = (row_end_addr - local_addr) + 1L;
	long to_end_of_trans = (last_addr - local_addr) + 1L;
	int loc_bytes = (int) java.lang.Math.min(to_end_of_row,
	 to_end_of_trans);
	if (loc_bytes > (BusWidth * max_cycles_per_trans))
    loc_bytes = (int)(BusWidth * max_cycles_per_trans);
	if (debug_split) {
    String str = "DEBUG(drd_ctrl) : Split Access\n";
    str += "row = "+local_row+"\n";
    str += "addr = "+Long.toHexString(local_addr)+"\n";
    str += "loc_bytes = "+loc_bytes+"\n";
    System.out.println(str);
	}

	if (loc_bytes <= 0) {
    System.out.println("Fatal Error (negative loc_bytes) "+
       "in split_access");

    String str = "row_end_addr\t= "+row_end_addr+"\n";
    str += "last_addr\t= "+last_addr+"\n";
    str += "local_addr\t= "+local_addr+"\n";
    str += local_bank.toString(0xFFF);
    str += local_dev.toString(0xFFF);
    System.out.println(str);
    System.exit(1);
	}

	// Don't close when splitting accesses
	if ((loc_bytes == to_end_of_row) ||
    (loc_bytes == to_end_of_trans)) {
    type = type_param;
	} else {
    type = open_type;
	}

	//
	// Munge/re-order address bits to accomplish pseudo-interleaving
	// of DRAM pages between different banks.
	//
	long dram_addr = local_addr;
	if (enable_remap)
    dram_addr = map_procaddr_to_dramaddr(local_addr);

	// Create Transaction
	this_trans = new ddr2_trans(type, dram_addr,
    loc_bytes);

	// Initiate Access on channel
	the_bus.access(this_trans);

	// Schedule Transaction
	the_schedule.schedTrans(this_trans);

	local_addr += ((long)loc_bytes);
    }
    return this_trans;
}

long map_procaddr_to_dramaddr(long procaddr) {

    if (mem_changed) {

	//
	// Determine Number of bits used in memory system.
	//
	max_addr = maxAddr();
	String s1 = (new Long(max_addr)).toString();
	// System.out.println("@@@ Max_addr = "+s1+"\n");
	used_bits = (new BigInteger(s1)).bitLength();

	//
	// Determine memory system/device boundaries
	//
	// Assume that all devices/banks are of the same size, and
	// therefor we can get critical values from first device/bank
	//
	ddr2_device dev_0 = (ddr2_device) the_bus.device_list.elementAt(0);
	if (! (dev_0 instanceof ddr2_device)) {
    System.out.println("ERROR : unable to determine device bits "+
```

```
   "in ddr2_ctrl.map_procaddr_to_dramaddr()");
return procaddr;
    }

    bank_mask = dev_0.bank_mask;
    s1 = (new Long(bank_mask)).toString();
    bank_msb = (new BigInteger(s1)).bitLength();
    bank_lsb = (new BigInteger(s1)).getLowestSetBit();

    //
    // retrieve First Bank & thus Row info
    //
    ddr2_bank bank_0 = dev_0.banks[0];
    if (! (bank_0 instanceof ddr2_bank)) {
System.out.println("ERROR : unable to determine bank bits "+
   "in ddr2_ctrl.map_procaddr_to_dramaddr()");
return procaddr;
    }
    row_mask = bank_0.row_mask;
    s1 = (new Long(row_mask)).toString();
    row_msb = (new BigInteger(s1)).bitLength();
    row_lsb = (new BigInteger(s1)).getLowestSetBit();

    //
    // From above derive device info
    //
    dev_msb = used_bits;
    dev_lsb = bank_msb + 1;

    index_mask = ((1<<row_lsb) - 1);

    mem_changed = false;
} // mem_changed

//
// Perform Re-mapping
//

// Inversion Remap

long dramaddr = procaddr & index_mask;

long loop_bits = 0;
//System.out.println("prior to j loop ; row_lsb = "+row_lsb+
//   "\tused_bits = "+used_bits+"\n");
for (int j = row_lsb ; j < used_bits ; j++) {
    // System.out.println("In j loop; j = "+j+"/n");
    long bit_mask = (0x1L<<j);
    if ((bit_mask & procaddr) != 0) {
loop_bits = (loop_bits << 1) | 0x1;
// System.out.println("inserting 0x1\n");
    } else {
loop_bits = (loop_bits << 1) | 0x0;
    }
}
loop_bits = loop_bits << row_lsb;
dramaddr = dramaddr | loop_bits;

//
// Debug print
//

if (debug_remap) {
    debug_prints++;
    String str = "Loop Bits = 0x"+Long.toHexString(loop_bits)+"\n";
    str += "Processor Address = 0x"+Long.toHexString(procaddr)+
"\n\t";
    str += "bank = 0x"+Long.toHexString(procaddr & bank_mask);
    str += "\n\trow  = 0x"+Long.toHexString(procaddr & row_mask);
    str += "\nDRAM Address = 0x"+Long.toHexString(dramaddr)+
"\n\t";
    str += "bank = 0x"+Long.toHexString(dramaddr & bank_mask);
    str += "\n\trow  = 0x"+Long.toHexString(dramaddr & row_mask) +
"\n";
    System.out.print(str);
    if (debug_prints > 200)
System.exit(1);
}

return dramaddr;
```

```
    }

    public long maxAddr() {
          return the_bus.maxAddr();
    }

    public long currentTime() {
          return the_schedule.currentTime();
    }

    public boolean endSimulation() {
          return the_schedule.endSimulation();
    }

    //
    // toString
    //
    public void printYourself() {
          System.err.println(this.toString(0xFFFFFFFF));
    }

    public String toString() {
          return this.toString(display_mode);
    }

    public String toString(int dm) {
          String str = new String();

          str += "ddr2_ctrl ["+super.toString()+"]\n";
          str += "\tDRAM Type\t\t: "+type_string+"\n";
          str += "\tCtrler Policy\t\t: ";
          if (policy_closeautoprecharge) {
              str += "Close-Page-Autoprecharge\n";
          } else if (policy_openpage) {
              str += "Open-Page\n";
          } else {
              str += "UNKNOWN\n";
          }
          str += "\tCAS Latency\t\t: "+cas_lat+"\n";
          str += "\ttRAS\t\t: "+Tras+"\n";
          str += "\ttRCD\t\t: "+Trcd+"\n";
          str += "\ttRP\t\t: "+Trp+"\n";
          str += "\ttWR\t\t: "+Twr+"\n";

          str += "\tCtrler Nominal row size : "+(1 << row_shift)+"\n";
          str += "\tTime Elapsed\t\t: "+
              ((double)the_schedule.currentTime()) * clock_period+" (sec)\n";

          str += "\tController Accesses\t: "+ctrler_accesses+"\n";
          str += "\tTotal Bytes Transfered\t: "+total_bytes+"\n";
          str += "\tAvg Request size\t: "+
              (total_bytes / ctrler_accesses) + "\n";
          str += "\tMultiTrans Accesses\t: "+multi_trans_accesses+"\n";
          if (VCEnabled) {
              str += "\tVirtual Channel Enabled\n";
              str += "\tVirtual Channel Allocation Policy\t: "+vc_policy+"\n";
          }

          str += the_schedule.toString(ddr2_sched.STATS);
          str += the_bus.toString(ddr2_bus.STATS);
          return str;
    }

} // sdram_ctrl

=====================================================================================================

// package drdram_sim

/*
  File : SDRAM_BUS

  Author : Brian Davis

  */

import java.util.Vector;

public class ddr2_bus {
```

```
//
// Constants
//
static final boolean debug = false;
static final int DEBUG = (1<<0);
static final int STATS = (1<<1);
static final String SPACES8 = "        ";
static java.text.NumberFormat nf;

//
// Variables
//
Vector device_list;
ddr2_ctrl the_ctrler;
// Refresh
private long access_calls = 0;
private double access_bytes = 0;
private long read_hits = 0;
private long read_misses = 0;
private long write_hits = 0;
private long write_misses = 0;
private long refresh_calls = 0;
// Display
int display_type = 0;

//
// Constructors
//
ddr2_bus(ddr2_ctrl ctrl_host) {
      the_ctrler = ctrl_host;
      device_list = new Vector();
}

public ddr2_device addDevice(String dev_type) {
      long new_start;
      if (device_list.isEmpty()) {
          new_start = 0;
      } else {
          new_start = ((ddr2_device)(device_list.lastElement())).addr_end + 1;
      }
      ddr2_device newDevice = new ddr2_device(the_ctrler, dev_type, new_start);
      if (newDevice instanceof ddr2_device) {
          // Add to Vector
          device_list.addElement(newDevice);
      }
      return newDevice;
}

public boolean addrMapped(long addr) {
      if (device_list.isEmpty()) {
          if (debug) {
      System.out.println("ddr2_bus device_list is Empty");
          }
          return false;
      }

      long first_start =
          ((ddr2_device)(device_list.firstElement())).addr_start;

      long last_end = ((ddr2_device)(device_list.lastElement())).addr_end;

      boolean ret_val =
          ((first_start <= addr) && (addr <= last_end));

      if (debug && (!ret_val)) {
          System.out.println("first_start = "+first_start+
             "\nlast_end = "+last_end);
      }

      return ret_val;
}

long maxAddr() {
      long last_end = 0;
      if (device_list.size() > 0)
          last_end = ((ddr2_device)(device_list.lastElement())).addr_end;
      return last_end;
}

ddr2_device devForAddr(long addr) {
      for (int j = 0 ; j < device_list.size() ; j++) {
```

```
            ddr2_device dev = ((ddr2_device)device_list.elementAt(j));
            if ((dev.addr_start <= addr) &&
        (dev.addr_end >= addr)) {
        return dev;
            }
        } // for
        return null;
}

public boolean access(ddr2_trans this_trans) {
        //
        // Advance time
        //
        access_calls++;
        access_bytes += this_trans.num_bytes;

        //
        // Perform Transaction - Transaction allocated
        //
        ddr2_device dev_for_trans;

        //
        // Determine device via Address mapping
        //
        dev_for_trans = devForAddr(this_trans.address);

        //
        // Determine VC based on

        //
        // Perform Access
        //
        if (dev_for_trans instanceof ddr2_device) {
            dev_for_trans.access(this_trans);
        } else {
            System.out.println("ERROR : no device found for Transaction :\n"+
               this_trans);
            return false;
        }

        if ((this_trans.type & dram_trans.TRANS_READ) != 0) {
            if (this_trans.SAHit)
        read_hits++;
            else
        read_misses++;
        } else if ((this_trans.type & dram_trans.TRANS_WRITE) != 0) {
            if (this_trans.SAHit)
        write_hits++;
            else
        write_misses++;
        }

        return true;
}

int refreshAllDevices() {
        int ret_row = 0;
        refresh_calls ++;
        for (int j = 0 ; j < device_list.size() ; j++) {
            ddr2_device dimm = (ddr2_device) device_list.elementAt(j);
            ret_row = dimm.refreshAllBanks();
        }
        return ret_row;
}

boolean prechargeAllDevices() {
        boolean ret_val = false;
        for (int j = 0 ; j < device_list.size() ; j++) {
            ddr2_device dimm = (ddr2_device) device_list.elementAt(j);
            dimm.prechargeAllBanks();
        }
        return ret_val;
}

//
// toString
//
public String toString(int dt) {
        display_type = dt;
        return this.toString();
}
```

```
    public String toString() {
          String str = new String();
          str += "ddr2_bus ["+super.toString()+"]\n";
          str += "\tSpans Addresses\t\t: (";
          str += "0x"+java.lang.Long.toHexString(((ddr2_device)(device_list.firstEle-
ment())).addr_start);
          str += " : ";
          str += "0x"+java.lang.Long.toHexString(((ddr2_device)(device_list.lastElement())).addr_end);
          str += ")\n";
          str += "\tNumber of Devices\t: "+device_list.size()+"\n";
          str += "\tAccesses crossing bus\t: "+access_calls+"\n";
          str += "\tBus Access Hits\t\t: "+(read_hits + write_hits)+"\t\t";
          str += percent8Str((read_hits + write_hits)/((double)access_calls))+
              "\n";
          if (access_calls != 0) {

              long reads = (read_hits + read_misses);
              str += "\tBus Reads\t\t: "+reads+"\t\t";
              str += percent8Str(reads/((double)access_calls))+"\n";
              if (reads != 0) {
          str += "\tBus Read Hits\t\t: "+read_hits+"\t\t";
          str += percent8Str(read_hits/((double)access_calls))+"\t\t";
          str += percent8Str(read_hits/((double)reads))+"\n";
          str += "\tBus Read Misses\t\t: "+read_misses+"\t\t";
          str += percent8Str(read_misses/((double)access_calls))+"\t\t";
          str += percent8Str(read_misses/((double)reads))+"\n";
          } // reads

              long writes = (write_hits + write_misses);
              str += "\tBus Writes\t\t: "+writes+"\t\t";
              str += percent8Str(writes/((double)access_calls))+"\n";
              if (writes != 0) {
          str += "\tBus Write Hits\t\t: "+write_hits+"\t\t";
          str += percent8Str(write_hits/((double)access_calls))+"\t\t";
          str += percent8Str(write_hits/((double)writes))+"\n";
          str += "\tBus Write Misses\t: "+write_misses+"\t\t";
          str += percent8Str(write_misses/((double)access_calls))+"\t\t";
          str += percent8Str(write_misses/((double)writes))+"\n";
              } // Writes

          } // If Accesses

          str += "\tBytes crossing bus\t: "+access_bytes+"\n";
          str += "\tAvg Bytes / access\t: "+
              (access_bytes / access_calls)+"\n";
          if ((display_type & STATS) != 0) {
              // str += "\tRefresh Transactions\t: "+refresh_calls+"\n";


              for (int j = 0 ; j < device_list.size() ; j++) {
          ddr2_device dimm = (ddr2_device) device_list.elementAt(j);
          str += dimm.toString(ddr2_device.STATS);
              }
          }

          if ((display_type & DEBUG) != 0) {
          }

          return str;
    }

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

} // class sdram_bus

==========================================================================================

// package drdram_sim

/*
  File : SDRAM_device
```

```
  Author : Brian Davis
  **
  ** all devices are defined to be BUS-WIDE devices
  */

// import java.util.Vector;
import java.math.BigInteger;

public class ddr2_device {

    //
    // Constants
    //
    static final boolean debug = false;
    static final String DIMM_64MB = "64MByte DIMM";
    static final String DIMM_256MB = "256MByte DIMM";
    static final long SIZE_64MB = ((0x1L)<<26); // 2^26 bytes
    static final long SIZE_256MB = ((0x1L)<<28); // 2^28 bytes
    static final int VCHANNELS_PER_DEVICE = 16;
    static final String SPACES8 = "        ";

//     static final String POLICY_LRU = dram_ctrl.POLICY_LRU;
//     static final String POLICY_RANDOM = dram_ctrl.POLICY_RANDOM;
//     static final String POLICY_ASSOC = dram_ctrl.POLICY_ASSOC;
//     static final String POLICY_P12IO4 = dram_ctrl.POLICY_P12IO4;

    //
    // display items
    //
    static final int DEBUG = (1<<0);
    static final int STATS = (1<<1);
    int display_mode = 0;
    static java.text.NumberFormat nf;
    static java.util.Random rand = null;

    //
    // Device specific values
    //
    ddr2_ctrl the_ctrler;
    String type_string;
    long device_size;
    long addr_start;
    long addr_end;
    int num_banks;
    ddr2_bank [] banks;
    vc_buffer [] vchannels = null;
    //     String vc_policy = null;
    int debug_ctr = 0;
    long bank_mask;
    int bank_shift;

    //
    // Statistics
    //
    long accesses = 0;
    long vc_hits = 0;
    long rc_hits = 0;
    long sa_hits = 0;
    long pc_hits = 0;

    //
    // Constructor
    //
    ddr2_device(ddr2_ctrl ctrler_host, String type, long start) {
          the_ctrler = ctrler_host;
          type_string = type;
          int rows_per_bank = 0;
          if (type == DIMM_256MB) {
              device_size = SIZE_256MB;
              num_banks = 8;
              rows_per_bank = 8192;
          } else {
              System.out.println("ERROR : undefined device type!\n");
              System.exit(1);
          }
          // start should of course be aligned
          addr_start = start;
          addr_end = start + (device_size-1);
          long bank_size = device_size / num_banks;
          //
          bank_mask = (device_size-1) & (~(bank_size-1));
```

```
      String s1 = (new Long(bank_mask)).toString();
      bank_shift = (new BigInteger(s1)).getLowestSetBit();

      // Define banks
      banks = new ddr2_bank[num_banks];
      long bank_start_addr = addr_start;
      for (int j = 0 ; j < banks.length ; j++) {
          banks[j] = new ddr2_bank(this,
           bank_size,
            rows_per_bank,
            bank_start_addr);
          bank_start_addr += bank_size;
      }

      if (the_ctrler.VCEnabled) {
          vchannels = new vc_buffer[VCHANNELS_PER_DEVICE];
          // Added 5/8/2000
          // vc_policy= the_ctrler.POLICY_LRU;
          for (int j = 0 ; j < vchannels.length ; j++) {
      vchannels[j] = new vc_buffer();
          }
      }

      if (debug) {
          System.out.println("DDR2 device "+type+" created\n"+
             "\taddr_start = "+addr_start+"("+
             Long.toHexString(addr_start)+
             ")\n\taddr_end = "+addr_end+"("+
             Long.toHexString(addr_end)+")");

      }

} // sdram_device

boolean access(ddr2_trans this_trans) {
      accesses++;
      //
      // Primary purpose is to determine hit/miss status
      //

      ddr2_bank this_bank = whichBank(this_trans.address);
      if (!(this_bank instanceof ddr2_bank)) {
          System.out.println("ERROR : undefined bank in "+
             "sdram_device.access()");
          System.exit(1);
      }
      this_bank.accesses++;

      long row_addr = this_bank.rowIndex(this_trans.address);

      /*
      ** Check bank conditions
      */
      if (the_ctrler.VCEnabled) {
          vc_buffer chan = this_bank.getActiveVCBuffer(this_trans.address);
          if (chan instanceof vc_buffer) {
      // VC Possible only
      this_trans.VCHit = true;
      this_trans.vc_used = chan;
      vc_hits++;
          } else {
      // VC Must Allocate Channel
      this_trans.VCHit = false;
      // Get LRU clean (or whatever policy buffer)
      chan = determineVCForEvict(this_bank, this_trans.address);
      this_trans.vc_used = chan;
      // Where does the allocate buffer scheduling go?
      // in ddr2_sched.schedTrans()
          }
          if (debug) {
      debug_ctr++;
      if ((debug_ctr % 100) == 0) {
          System.out.println(this.toString(0xFFF));
      }
      if ((debug_ctr % 1000000) == 0) {
          System.out.println("Terminating due to debug_ctr "+
             "in ddr2_device");
          System.exit(0);
      }
          }
      } else if ((the_ctrler.EMSEnabled) &&
```

```
            (this_bank.inRowCache(row_addr))) {
            // EMS Possible only
            this_trans.RCHit = true;
            rc_hits++;
        } else if (this_bank.rowInSA(row_addr)) {
            // Condition 1;
            this_trans.SAHit = true;
            this_trans.PrechargeHit = false;
            sa_hits++;
            this_bank.sa_hits++;
        } else if (this_bank.precharged()) {
            // Condition 2;
            this_trans.SAHit = false;
            this_trans.PrechargeHit = true;
            pc_hits++;
            this_bank.pc_hits++;
        } else {
            // Condition 3;
            this_trans.SAHit = false;
            this_trans.PrechargeHit = false;
        }
        this_trans.access_bank = this_bank;
        this_trans.access_row = row_addr;

        //
        // Don't want to update bank conditions until after scheduling
        // because Refresh might be forced to happen first!
        //
//      /*
//      ** Update bank conditions
//      */
//      if ((this_trans.type == dram_trans.TRANS_READ_PRE) ||
//          (this_trans.type == dram_trans.TRANS_WR_PRE)) {
//          this_bank.prechargeBank();
//      } else if ((this_trans.type == dram_trans.TRANS_READ_NOPRE) ||
//                 (this_trans.type == dram_trans.TRANS_WR_NOPRE)) {
//          this_bank.setOpen(row_addr);
//      } else {
//          System.out.println("ERROR : unknown bank status from Trans type");
//          System.out.println("type = this_trans.type");
//          System.exit(1);
//      }

        return true;
    }

    //
    // Associativity of Channels dependant
    //
    boolean inActiveChannel(ddr2_bank in_bank, long addr) {
        if (!the_ctrler.VCEnabled) {
            System.out.println("ERROR : device.inActiveChannel should not "+
               "be called in a non-VCEnabled environment");
            return false;
        }
        if (vchannels == null) {
            System.out.println("ERROR : device.vchannels not initialized");
            return false;
        }
        for (int j = 0 ; j < vchannels.length ; j++) {
            if (vchannels[j].contains(addr)) {
        return true;
            }
        }
        return false;
    }

    vc_buffer getActiveVCBuffer(ddr2_bank in_bank, long addr) {
        if (!the_ctrler.VCEnabled) {
            System.out.println("ERROR : device.getActiveVCBuffer should not "+
               "be called in a non-VCEnabled environment");
            return null;
        }
        if (vchannels == null) {
            System.out.println("ERROR : device.vchannels not initialized");
            return null;
        }
        for (int j = 0 ; j < vchannels.length ; j++) {
            if (vchannels[j].contains(addr)) {
        return vchannels[j];
            }
```

```
      }
      return null;
}

vc_buffer determineVCForEvict(ddr2_bank in_bank, long addr) {
      vc_buffer ret_buf = null;
          for (int j = 0 ; j < vchannels.length ; j++) {
      if (vchannels[j].contains(addr)) {
          System.out.println("ERROR : determineVCForEvict should "+
             "not be called if buffer is already "+
             " allocated");
          return vchannels[j];
      }
          }
      if (the_ctrler.vc_policy == dram_ctrl.POLICY_LRU) {
          // Initially Assume fully associative mapping
          long oldest_use = Long.MAX_VALUE;
          for (int j = 0 ; j < vchannels.length ; j++) {
      if (vchannels[j].last_access_cycle < oldest_use) {
          oldest_use = vchannels[j].last_access_cycle;
          ret_buf = vchannels[j];
      }
          }
      } else if (the_ctrler.vc_policy == dram_ctrl.POLICY_RANDOM) {
          if (rand == null) {
      rand = new java.util.Random();
          }
          int index = rand.nextInt(vchannels.length);
          ret_buf = vchannels[index];
      } else if (the_ctrler.vc_policy == dram_ctrl.POLICY_P12IO4) {
          System.out.println("ERROR : policy busmaster variant undef "+
             "determineVCForEvict()");
      } else {
          //
          // What state to change to denote/indicate allocation process
          // to sched methods??
          //
          System.out.println("ERROR : Unknown policy in "+
             "determineVCForEvict()");
      }

      return ret_buf;
}

//
// Address mapping dependant
//

int bankIndex(long addr) {
      int index = ((int)((addr & bank_mask) >> bank_shift));
      return index;
}

ddr2_bank whichBank(long addr) {
      ddr2_bank ret_bank = null;
      int index = bankIndex(addr);
      if ((index >= 0) &&
          (index < banks.length)) {
          ret_bank = banks[index];
      }
      return ret_bank;
}

void prechargeAllBanks() {
      for (int j = 0 ; j < banks.length ; j++) {
         banks[j].prechargeBank();
      }
}

int refreshAllBanks() {
      int row = -1;
      for (int j = 0 ; j < banks.length ; j++) {
          row = banks[j].refreshNextRow();
      }
      return row;
}

//
// toString
//
public String toString() {
```

```
            return this.toString(display_mode);
    }

    public String toString(int dm) {
            String str = new String();
            str += "ddr2_device ["+super.toString()+"]\n";
            str += "\tDevice type \t\t: "+type_string+"\n";
            str += "\tSpans Addresses\t\t: (";
            str += "0x"+java.lang.Long.toHexString(addr_start);
            str += " : ";
            str += "0x"+java.lang.Long.toHexString(addr_end);
            str += ")\n";
            if ((dm & STATS) != 0) {
                str += "\tDevice Accesses\t\t: "+accesses+"\n";
                double ac = (double) accesses;
                str += "\tDev VChan Hits\t\t: "+vc_hits+"\t"+
            percent8Str(vc_hits / ac ) + "\n";
                str += "\tDev RowCache Hits\t: "+rc_hits+"\t"+
            percent8Str(rc_hits / ac ) + "\n";
                str += "\tDev SAmp Hits\t\t: "+sa_hits+"\t"+
            percent8Str(sa_hits / ac ) + "\n";
                str += "\tDev PreChg Hits\t\t: "+pc_hits+"\t"+
            percent8Str(pc_hits / ac ) + "\n";
                str += "\tbank_mask\t\t: 0x"+java.lang.Long.toHexString(bank_mask);
                str += "\n\tbank_shift\t\t: "+bank_shift+"\n";
            }

            if (vchannels != null) {
                str += "\tVirtual Channels\t\t: "+vchannels.length+"\n";
                //     str += "\tVC allocation policy\t: "+vc_policy+"\n";
                for (int j = 0 ; j < vchannels.length ; j++) {
            str += vchannels[j].toString(dm);
                }
            }
            for (int j = 0 ; j < banks.length ; j++) {
                str += banks[j].toString(dm);
            }
            return str;
    }

    private String percent8Str(double in) {
            if (!(nf instanceof java.text.NumberFormat)) {
                nf = java.text.NumberFormat.getPercentInstance();
                nf.setMinimumFractionDigits(2);
            }
            String ret_str;
            ret_str = SPACES8 + nf.format(in);
            return ret_str.substring(ret_str.length() - 8);
    }

} // CLASS ddr2_device

=========================================================================================================

// package sdram_sim

/*
  File : DDR2_BANK

  Author : Brian Davis

  */
// import java.util.Vector;
import java.math.BigInteger;

public class ddr2_bank {

    // Class Variables
    static int banks_created = 0;
    static boolean debug = false;

    //
    // display items
    //
    static final int DEBUG = (1<<0);
    static final int STATS = (1<<1);
    static final int VC_FRACTION_OF_ROW = 4;
    int display_mode = 0;
    static java.text.NumberFormat nf;
    static final String SPACES8 = "        ";
```

```
// Instance Variables
ddr2_device the_device;
long addr_start;
long addr_end;
int NumRows;
long row_size;
long row_mask;
int row_shift;
boolean precharged;
boolean validSA;
long sa_contains;
boolean validRC = false;
long rc_contains;
long bank_ready;

// Stats
long accesses = 0;
long pc_hits = 0;
long sa_hits = 0;

//
int last_refreshed_row = -1;

// debug variables
long lgst_row_seen = -1;

ddr2_bank(ddr2_device device_host, long bank_size, int rows, long s_addr) {
      banks_created++;
      the_device = device_host;
      precharged = false;
      validSA = false;
      sa_contains = -1;
      NumRows = rows;
      addr_start = s_addr;
      addr_end = addr_start + (bank_size - 1);
      row_size = bank_size / NumRows;
      row_mask = (bank_size-1) & (~(row_size-1));
      String s1 = (new Long(row_mask)).toString();
      row_shift = (new BigInteger(s1)).getLowestSetBit();
      bank_ready = 0;
}

int rowIndex(long addr) {
      int rowIndex = ((int)((addr & row_mask) >> row_shift));
      return rowIndex;
}

long rowEndAddr(int row) {
      if ((row < 0) || (row >= NumRows)) {
          System.out.println("ERROR(sdram_bank) : attempt to find EndAddr "+
             "of out-of-bounds row");
          return(-1L);
      }
      long re_addr = addr_start + ((row + 1) * row_size) - 1;
      return re_addr;
}


boolean rowInSA (long row) {
      if (validSA && (sa_contains == row)) {
          return true;
      }
      return false;
}

boolean inRowCache(long row) {
      if ((the_device.the_ctrler.EMSEnabled) &&
          (validRC) &&
          (rc_contains == row)) {
          return true;
      }
      return false;
}

boolean inActiveChannel(long addr) {
      return the_device.inActiveChannel(this, addr);
}

vc_buffer getActiveVCBuffer(long addr) {
      return the_device.getActiveVCBuffer(this, addr);
}
```

```
boolean precharged() {
      return precharged;
}

void prechargeBank() {
      precharged = true;
      validSA = false;
}

boolean setOpen(long row) {
      if (!(row < NumRows)) {
          return false;
      }
      if (row > lgst_row_seen) {
          lgst_row_seen = row;
      }
      precharged = false;
      validSA = true;
      sa_contains = row;
      return true;
}

boolean setRowCache(long row) {
      if ((row < 0) ||
          (row >= NumRows)) {
          return false;
      }
      validRC = true;
      rc_contains = row;
      return true;
}

boolean setChannelOpen(vc_buffer channel, long addr, long cycle) {
      if (channel instanceof vc_buffer) {
          //
          // address currently contained in buffer
          // no need to open new buffer
          //
          channel.setAccessState(addr, cycle);
      } else {
          //
          // If this is the case, prior to the transaction
          // this a buffer should have been made empty
          //
          System.out.println("ERROR : ddr2_bank.setChannelOpen() : "+
             "Buffer not allocated prior to access");
      }
      return false;
}

boolean prefetchRowIntoChan(vc_buffer new_chan, long addr) {
      //
      // Determine address range to be contained by Row
      //

      // generating WRONG row & row_start
      //int row = rowIndex(addr);
      // long row_start = row << row_shift;

      long row_start = addr & (~(row_size-1));
      long row_end = (row_start + row_size - 1);
      long chan_size = row_size / VC_FRACTION_OF_ROW;

      if (debug) {
          String str = "In prefetchRowIntoChan\n";
          str += "\taddr = 0x"+java.lang.Long.toHexString(addr)+"\n";
          str += "\trow_start = 0x"+java.lang.Long.toHexString(row_start)+
      "\n";
          str += "\trow_end = 0x"+java.lang.Long.toHexString(row_end)+"\n";
          str += "\tchan_size = 0x"+java.lang.Long.toHexString(chan_size);
          System.out.println(str);
      }
      int vc_found = 0;
      long vc_start, vc_end;
      for (vc_start = row_start ;
           vc_start < row_end ;
           vc_start += chan_size) {
          vc_end = vc_start + (chan_size - 1);
          //
          // Stop this loop if addr in range
```

```
            //
            if ((vc_start <= addr) &&
        (addr <= vc_end)) {
        vc_found++;
        new_chan.setToContain(this, vc_start, vc_end);
        break;
            }
        }

        // Error Found 4/27
        // vc_found != 1 frequently!!!
        if (vc_found != 1) {
            System.out.println("ERROR : ddr2_bank.prefetchRowIntoChan() "+
           "vc_found != 1\n");
        }

        return false;
    }

    int refreshNextRow() {
        // Refresh command INCLUDES prefetch transaction in timing
        //
        // if (!precharged) {
        // System.out.println("ERROR : refresh of non-precharged bank");
        // }
        int row = (last_refreshed_row + 1) % NumRows;
        last_refreshed_row = row;
        if (!the_device.the_ctrler.EMSEnabled) {
            precharged = true;
            validSA = false;
            sa_contains = row;
        }
        return row;
    }

    String toString(int dm) {
        String str = new String();
        str += "ddr2_bank ["+super.toString()+"]\n";
        str += "\tSpans Addresses\t\t: (";
        str += "0x"+java.lang.Long.toHexString(addr_start);
        str += " : ";
        str += "0x"+java.lang.Long.toHexString(addr_end);
        str += ")\n";
        str += "\tRows in bank\t\t: "+NumRows+"\n";
        str += "\tRow size\t\t: "+row_size+"\n";
        if ((dm & STATS) != 0) {
            str += "\tBank Accesses\t\t: "+accesses+"\n";
            double ac = (double) accesses;
            str += "\tBank SAmp Hits\t\t: "+sa_hits+"\t"+
        percent8Str(sa_hits / ac ) + "\n";
            str += "\tBank PreChg Hits\t: "+pc_hits+"\t"+
        percent8Str(pc_hits / ac ) + "\n";
        }
        if ((dm & DEBUG) != 0) {
            str += "\tLgst Row Set Open\t: "+lgst_row_seen+"\n";
            str += "\trow_mask\t: 0x"+java.lang.Long.toHexString(row_mask);
            str += "\n\trow_shift\t: "+row_shift+"\n";
        }
        return str;
    }

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

} // class sdram_bank

==========================================================================================

// package drdram_sim

/*
  File : SDRAM_BUS

  Author : Brian Davis
```

```
  */

import java.util.Vector;

public class ddr2_sched {

    //
    // Constants
    //
    static final boolean debug = false;
    static final boolean debug_shutdown = false;
    static final int MAX_LIST_SIZE = 10;
    static final int ERROR_LIST_SIZE = 1000;

    static final int DEBUG = (1<<0);
    static final int STATS = (1<<1);
    static final String SPACES8 = "        ";

    //
    // Variables
    //
    ddr2_ctrl the_ctrler;
    Vector addrTrans;
    Vector dataTrans;
    long reads_sched = 0;
    long writes_sched = 0;
    long refr_sched = 0;
    long total_sched = 0;
    double total_latency = 0.0;
    long adj_bank_accesses = 0;
    // time variables
    long current_cycle = -1;
    long last_retired = 0;
    long used_cycles = 0;
    long addr_used_cycles = 0;
    long data_used_cycles = 0;
    long olap_used_cycles = 0;
    // Refresh
    private long last_refresh_time = 0;
    private long last_refresh_iter = 0;
    // display variables
    int display_type = 0;
    static java.text.NumberFormat nf;
    ddr2_trans lastTrans;

    //
    // Constructor
    //
    ddr2_sched(ddr2_ctrl host_ctrler) {
          the_ctrler = host_ctrler;
          addrTrans = new Vector();
          dataTrans = new Vector();
    }

    long schedTrans(ddr2_trans new_trans) {

          long l_evict_start = -1;
          long l_pre_start = -1;
          long l_row_start = -1;
          long l_col_start = -1;
          long l_post_start = -1;
          long l_addr_start = -1;
          long l_addr_end = -1;
          long l_data_start = -1;
          long l_data_end = -1;

          if ((lastTrans instanceof ddr2_trans) &&
              (current_cycle < lastTrans.start_cycle)) {
              advanceCurrentTime(lastTrans.start_cycle);
          }

          long earliest_possible = (current_cycle > 0 ) ? current_cycle : 0;

          boolean occupies_data = false;

          /*
          ** Prior transaction(s) on bus
          */
          ddr2_trans prev_a_trans = null;
          try {
```

```
        prev_a_trans = (ddr2_trans)addrTrans.lastElement();
    } catch (java.util.NoSuchElementException e) {
        // Do nothing it remains
        // prev_a_trans = null;
    }

    ddr2_trans prev_d_trans = null;
    try {
        prev_d_trans = (ddr2_trans)dataTrans.lastElement();
    } catch (java.util.NoSuchElementException e) {
        // Do nothing it remains
        // prev_d_trans = null;
    }

    /*
    ** check to see if there is already a transaction CURRENTLY using
    ** the addr portion of the DRAM bus
    */
    if ((prev_a_trans instanceof ddr2_trans) &&
        (prev_a_trans.addrEnd > earliest_possible)) {
        earliest_possible = prev_a_trans.addrEnd;
    }

    /*
    ** Determine # of data cycles which will be required
    */
    long data_cycles = (long)
        (java.lang.Math.ceil(((double)new_trans.num_bytes)/the_ctrler.BusWidth));

    //
    // Check for requirement of VC allocation phase to transaction
    //
    if ((the_ctrler.VCEnabled) &&
        (new_trans.VCHit == false)) {
        //
        // Check for writeback of dirty Channel req'd
        //
        if (!(new_trans.vc_used instanceof vc_buffer)) {
    System.out.println("ERROR : vc_used not initialized prior "+
       "to schedTrans()");
    System.exit(1);
        }
        if (new_trans.vc_used.dirty) {
    //
    // Writeback req'd
    //
    l_evict_start = earliest_possible;
    earliest_possible = l_evict_start + the_ctrler.TvcEvict;

    if (debug) {
        System.out.println("DEBUG : Channel Ejection Scheduled\n"+
           "Re-used vc : \n"+
           new_trans.vc_used.toString(0xFFF)+
           "Full Device Info : \n"+
           new_trans.access_bank.the_device.
       toString(0xFFF));
    }

    new_trans.vc_used.evict(l_evict_start);

    // This time (TvcEvict) does not include
    // precharge time!!
    if (new_trans.SAHit || new_trans.PrechargeHit ||
        new_trans.VCHit || new_trans.RCHit) {
        //
        // Since VC Should always be used in conjunction with
        // a CPA policy, this should not be an issue.
        //
        System.out.println("ERROR : ddr2_sched Eviction will not "+
           "account for precharge");
    }
        } // dirty
        //
        // Update channel state for this transaction
        //
        new_trans.access_bank.prefetchRowIntoChan(new_trans.vc_used,
          new_trans.address);

    } // VC Enabled & VC Miss
    /*
    ** Assume we can schedule starting at earliest_possible
```

```
*/
if ((new_trans.type & dram_trans.TRANS_READ) != 0) {
    occupies_data = true;
    reads_sched++;
    /*
    ** Read transaction timings
    */
    if ((!new_trans.SAHit) &&
(!new_trans.PrechargeHit) &&
(!new_trans.VCHit) &&
        (!new_trans.RCHit)) {
/*
** Must do precharge
*/
l_pre_start = earliest_possible;
l_row_start = l_pre_start + the_ctrler.Trp;
l_col_start = l_row_start + the_ctrler.Trcd;
// l_addr_start = l_pre_start;
    } else if (new_trans.PrechargeHit) {
/*
** already precharged, but must access row
*/
l_pre_start = -1;
l_row_start = earliest_possible;
l_col_start = l_row_start + the_ctrler.Trcd;
// l_addr_start = l_row_start;
    } else if (new_trans.SAHit) {
/*
** requested row already in open page! YEAH!
*/
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else if (new_trans.VCHit) {
//
// Virtual Channel Hit
//
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else if (new_trans.RCHit) {
//
// Row Cache Hit
//
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else {
System.out.println("ERROR : Logical impossibility");
System.exit(1);
    }
    //
    // Common elements of all reads
    //
    l_addr_end = l_col_start;
    l_data_start = l_col_start + the_ctrler.cas_lat;
    l_data_end = l_data_start + data_cycles;
} else if ((new_trans.type & dram_trans.TRANS_WRITE) != 0) {
    /*
    ** Write transaction timings
    */
    occupies_data = true;
    writes_sched++;
    if ((!new_trans.VCHit) &&
(!new_trans.RCHit) &&
(!new_trans.SAHit) &&
(!new_trans.PrechargeHit)) {
/*
** Must do precharge
*/
l_pre_start = earliest_possible;
l_row_start = l_pre_start + the_ctrler.Trp;
l_col_start = l_row_start + the_ctrler.Trcd;
// l_addr_start = l_pre_start;
    } else if (new_trans.PrechargeHit) {
/*
** already precharged, but must access row
*/
```

```
l_pre_start = -1;
l_row_start = earliest_possible;
l_col_start = l_row_start + the_ctrler.Trcd;
// l_addr_start = l_row_start;
    } else if (new_trans.SAHit) {
/*
** requested row already in open page! YEAH!
*/
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else if (new_trans.VCHit) {
//
// Virtual Channel Hit
//
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else if (new_trans.RCHit) {
//
// Row Cache Hit
//
l_pre_start = -1;
l_row_start = -1;
l_col_start = earliest_possible;
// l_addr_start = l_col_start;
    } else {
System.out.println("ERROR : Logical impossibility");
System.exit(1);
    }
    //
    // Common elements of all writes
    //
    l_addr_end = l_col_start;
    l_data_start = l_col_start;
    l_data_end = l_data_start + data_cycles;
} else {
    System.out.println("ERROR : Access neither read nor write"+
       " in schedTrans()");
    System.exit(1);
}

l_addr_start = l_col_start;
if (l_row_start != -1)
    l_addr_start = l_row_start;
if (l_pre_start != -1)
    l_addr_start = l_pre_start;
if (l_evict_start != -1)
    l_addr_start = l_evict_start;

if (false && debug) {
    System.out.println("local variables scheduled");
}

/*
** Check for conflicts with prior accesses
** (prev_a_trans & prev_d_trans) and advance ALL l_* vars
** if conflict exists
*/

long addr_spacing = 0;
long data_spacing = 0;
long conflict_delta = 0;

/*
** Determine addr spacing from adjacency, bank & access type
**
// Only need to concern with adjacent bank !IF! this is not an
// open-page hit, VCHit or RCHit, true?
*/
if ((prev_a_trans instanceof ddr2_trans) &&
    (prev_a_trans.access_bank == new_trans.access_bank)) {
    // This is a trace/access stream stat
    if (prev_a_trans.dataBusReqd() && new_trans.dataBusReqd()) {
adj_bank_accesses++;
    }
    // Added 02/11/00
    if ((!new_trans.VCHit) &&
(!new_trans.RCHit) &&
```

```
(!new_trans.SAHit)) {
//
// adjacent access spacing
// Is bank_read used to determine timing in the case of
// implicit precharge?
// Is adjacent access completely handled with above timing
// requirements
//
addr_spacing = the_ctrler.Trp;
    }
}

if ((prev_d_trans instanceof ddr2_trans) &&
    (prev_d_trans.access_bank == new_trans.access_bank)) {
    //
    //  Adjacent access spacing
    // Is bank_read used to determine timing in the case of
    // implicit precharge?
    // Is adjacent access completely handled with above timing
    // requirements
    //
    data_spacing = 0;
}

// Must check for time conflicts between adjacent accesses
if ((prev_a_trans instanceof ddr2_trans) &&
    (l_addr_start < (prev_a_trans.addrEnd + addr_spacing))) {
    conflict_delta = (prev_a_trans.addrEnd + addr_spacing) -
l_addr_start;
}

if ((prev_d_trans instanceof ddr2_trans) &&
    (l_data_start < (prev_d_trans.dataEnd + data_spacing))) {
    long data_delta = (prev_d_trans.dataEnd + data_spacing) -
l_data_start;

    if (data_delta > conflict_delta) {
conflict_delta = data_delta;
    }
}

// Case where two adjacent requests go to different rows of the
// same bank : Pg 23 IBM 256Mb DDR SDRAM datasheet
if ((prev_d_trans instanceof ddr2_trans) &&
    (prev_d_trans.access_bank == new_trans.access_bank) &&
    (new_trans.SAHit == false)) {
    long adj_samebank_delta = 0;
    long pre_happens = (l_pre_start >= 0) ? l_pre_start :
(l_row_start - the_ctrler.Trp);
    if ((pre_happens >= 0) &&
((pre_happens + the_ctrler.cas_lat) < prev_d_trans.dataEnd)) {
adj_samebank_delta = prev_d_trans.dataEnd -
    (pre_happens + the_ctrler.cas_lat);
    }
    if (adj_samebank_delta > conflict_delta) {
conflict_delta = adj_samebank_delta;
    }
}

if (debug) {
    System.out.println("conflict delta ("+conflict_delta+
       ") determined");
}

if (conflict_delta > 0) {
    if (l_evict_start >= 0)
l_evict_start += conflict_delta;
    if (l_pre_start >= 0)
l_pre_start += conflict_delta;
    if (l_row_start >= 0)
l_row_start += conflict_delta;

    l_col_start += conflict_delta;
    l_addr_start += conflict_delta;
    l_addr_end += conflict_delta;

    if (l_data_start >= 0)
l_data_start += conflict_delta;
    if (l_data_end >= 0)
l_data_end += conflict_delta;
}
```

```
new_trans.preStart = l_pre_start;
new_trans.rowStart = l_row_start;
new_trans.colStart = l_col_start;
new_trans.addrStart = l_addr_start;
new_trans.addrEnd = l_addr_end;
new_trans.dataStart = l_data_start;
new_trans.dataEnd = l_data_end;
new_trans.start_cycle = l_addr_start;
new_trans.end_cycle = (l_data_end >= 0) ? l_data_end : l_addr_end;

//
// by here must have set (in trans):
// long start_cycle;
// long end_cycle;
// long rowStart;
// long colStart;
// long dataStart, dataEnd;
//

/*
** Precharge & Bank Ready STATUS & Timing Issues
*/
if (((new_trans.type & dram_trans.TRANS_READ_PRE) != 0) ||
    ((new_trans.type & dram_trans.TRANS_WR_PRE) != 0)) {

    /*
    ** Schedule implicit Precharge
    ** does not requre Address bus usage
    */
    if ((new_trans.type & dram_trans.TRANS_READ) != 0) {
new_trans.access_bank.bank_ready = l_data_end;
    } else if ((new_trans.type & dram_trans.TRANS_WRITE) != 0) {
new_trans.access_bank.bank_ready = l_data_end + the_ctrler.Twr;
    }
    //
    // Change Device/Bank State
    //
    new_trans.access_bank.prechargeBank();
} else if (((new_trans.type & dram_trans.TRANS_READ_NOPRE) != 0) ||
   ((new_trans.type & dram_trans.TRANS_WR_NOPRE) != 0) ||
   ((new_trans.type & dram_trans.TRANS_WR_NOXFER) != 0)) {
    /*
    ** No scheduling necessary
    ** is this block req'd? - Verify Not precharged &
    ** open-page status?
    */
    new_trans.access_bank.bank_ready = new_trans.dataEnd;
    //
    // Change Device/Bank State
    //
    // what does it mean for bank to be ready in a non-
    // precharge policy?
    //
    if ((new_trans.type & dram_trans.TRANS_WR_NOXFER) == 0) {
new_trans.access_bank.setOpen(new_trans.access_row);
    }
} else {
    System.out.println("ERROR : Unknown TRANS type to determine "+
       "precharge state");
}

//
// Update state for Cache-enhanced or unique DDR2 architectures
//
if (the_ctrler.VCEnabled) {
    new_trans.access_bank.setChannelOpen(new_trans.vc_used,
 new_trans.address,
 l_data_end);
    //  4/27 - why is dirty set here?
    // new_trans.vc_used.dirty = true;

    if ((new_trans.type & dram_trans.TRANS_WRITE) != 0) {
new_trans.vc_used.dirty = true;
    }
}
if (the_ctrler.EMSEnabled) {
    if ((new_trans.type != dram_trans.TRANS_WR_NOXFER) ||
(new_trans.RCHit)) {
new_trans.access_bank.setRowCache(new_trans.access_row);
    }
```

```
      } // EMS Enabled

      //
      // Update Latency metric values
      //
      total_sched++;
      long this_latency = (new_trans.end_cycle - new_trans.start_cycle);
      total_latency += this_latency;
      if (this_latency > 999) {
          System.out.println("ERROR (arbitrary) : this_latency > 999 in "+
             "ddr2_sched.schedTrans()");
      }

      /*
      ** Schedule : Add to addrTrans & dataTrans vectors
      */
      if (debug) {
          System.out.println("SCHEDULING(placing in transaction queues):\n"+
             new_trans.toString(0xFFF));
      }

      while(addrTrans.size() >= MAX_LIST_SIZE) {
          if (removeFromAddrTrans(addrTrans.elementAt(0)) == false) {
      break;
          }
      }
      addrTrans.addElement(new_trans);

      if (occupies_data) {
          while(dataTrans.size() >= MAX_LIST_SIZE) {
      dataTrans.removeElementAt(0);
          }
          dataTrans.addElement(new_trans);
      }
      lastTrans = new_trans;
      return l_addr_start;
} // schedTrans()

long scheduleRefresh(long cycle_sched, int row_index) {

      long l_pre_start = -1;
      long l_row_start = -1;
      long l_addr_start = -1;
      long l_addr_end = -1;

      // Invoked prior to scheduleRefresh call-site
      // prechargeAllBanks();

      /*
      ** Determine start time for refresh
      */
      ddr2_trans prev_a_trans = getPrevATrans();

      long refresh_cycle = cycle_sched;

      if ((prev_a_trans instanceof ddr2_trans) &&
          (refresh_cycle < (prev_a_trans.addrEnd + the_ctrler.Trp))) {
          //
          // Wait refresh until Addr bus avail
          //
          refresh_cycle = prev_a_trans.addrEnd + the_ctrler.Trp;
      }

      //
      // Refresh ALWAYS requires precharge, and row access only!
      //

      l_pre_start = refresh_cycle;
      l_row_start = l_pre_start + the_ctrler.Trp;
      l_addr_end = l_row_start + the_ctrler.Trcd;
      l_addr_start = l_pre_start;

      //
      // Create Refresh Transaction
      //

      ddr2_trans refresh_trans = new ddr2_trans(the_ctrler.TRANS_REFRESH,
        0L, 0,
        l_addr_start,
        l_addr_end);
```

```
      refresh_trans.addrStart = l_addr_start;
      refresh_trans.addrEnd = l_addr_end;
      refresh_trans.preStart = l_pre_start;
      refresh_trans.rowStart = l_row_start;
      refresh_trans.start_cycle = l_addr_start;
      refresh_trans.end_cycle = l_addr_end;

      if (debug) {
          System.out.println("Refresh row "+row_index+" scheduled from "+
             l_addr_start+" to "+l_addr_end);
      }

      //
      // Add Transaction to Schedule
      //

      refr_sched++;
      while(addrTrans.size() >= MAX_LIST_SIZE) {
          if (removeFromAddrTrans(addrTrans.elementAt(0)) == false) {
      break;
          }
      }
      addrTrans.addElement(refresh_trans);

      lastTrans = refresh_trans;

      return l_addr_start;
}

/*
** This method should typically be called prior to any multi-bank
** refresh command, as all refresh commands require that the bank being
** refreshed (or read) be first precharged
*/
boolean prechargeAllBanks(long earliest_cycle) {
      boolean ret_val = false;

      //
      // All banks can be precharged simultaneously
      // As described in page 18 of IBM DDR2 datasheet
      // labeled ADVANCE (11/99)
      //
      ret_val |= SchedulePrechargeAll(earliest_cycle);
      ret_val |= the_ctrler.the_bus.prechargeAllDevices();

      return ret_val;
}

boolean SchedulePrechargeAll(long earliest_cycle) {
      boolean ret_val = false;
      ddr2_trans prechg_trans = null;
      ddr2_trans prev_a_trans = getPrevATrans();

      //
      // Determine when precharge happens
      //
      long precharge_cycle = earliest_cycle;
      if ((prev_a_trans instanceof ddr2_trans) &&
          (precharge_cycle < (prev_a_trans.addrEnd + the_ctrler.Trp))) {
          //
          // Wait refresh until Addr bus avail
          //
          precharge_cycle = prev_a_trans.addrEnd + the_ctrler.Trp;
      }

      System.out.println("ERROR : ddr2_sched.SchedulePrechargeAll()"+
         "never completed");

      return ret_val;
}

ddr2_trans getPrevATrans() {
      ddr2_trans prev_a_trans = null;
      try {
          prev_a_trans = (ddr2_trans) addrTrans.lastElement();
      } catch (java.util.NoSuchElementException e) {
          // Do nothing it remains
          // prev_a_trans = null;
      }
      return prev_a_trans;
}
```

```
long currentTime() {
      return current_cycle;
}

 boolean advanceCurrentTime(long new_time) {
       if (new_time < current_cycle) return true;
      //
      // Check for refresh
      //
      for (long r_iter = (last_refresh_iter+1) ;
           r_iter < (new_time/the_ctrler.refresh_rate) ;
           r_iter++) {
          //
          // Determine refresh time & update state
          //
          last_refresh_iter = r_iter;
          long refr_time = (r_iter*the_ctrler.refresh_rate);
          //
          // Prior to refreshing devices, devices should be precharged
          //
          // prechargeAllBanks(refr_time);
          //
          // Refresh command scheduled below INCLUDES refresh
          //
          // state in devices is changed prior to scheduling because
          // this follows the access model (allows us to know hit/miss
          // status at schedule time)... but is not necessary for
          // refresh transactions
          //
          int refr_row = the_ctrler.the_bus.refreshAllDevices();
          last_refresh_time =
      scheduleRefresh(refr_time, refr_row);
      } // for r_iter

    current_cycle = new_time;

      if (debug) {
          System.out.println("schedule : cycleClock advanced to "+
             current_cycle);
      }
      return(true);
} // advanceBusTime

public boolean endSimulation() {
      if (debug) {
          System.out.println("schedule.endSimulation() called");
      }

      return retireAll();
}

public boolean retireAll() {
      //
      // Advance Bus Time to end
      //
      ddr2_trans le = (ddr2_trans) addrTrans.lastElement();
      if (debug_shutdown) {
          System.out.println("DEBUG(schedule) le.end_cycle = "+
             le.end_cycle);
          System.out.println("DEBUG(schedule) current_cycle = "+
             current_cycle);
          System.out.println("DEBUG(schedule) retired_to = "+
             last_retired);
      }
      advanceCurrentTime(le.end_cycle);
      retireTo(le.end_cycle);
      if (debug_shutdown) {
          System.out.println("DEBUG(schedule) le.end_cycle = "+
             le.end_cycle);
          System.out.println("DEBUG(schedule) current_cycle = "+
             current_cycle);
          System.out.println("DEBUG(schedule) retired_to = "+
             last_retired);
      }
      return true;
}

private boolean removeFromAddrTrans(Object to_be_removed) {
      if (!(to_be_removed instanceof ddr2_trans)) {
```

```
            System.out.println("ERROR : item "+to_be_removed+
               " to be removed from schedule is not "+
               "a transaction object");
            return false;
        }
        ddr2_trans tbr = (ddr2_trans) to_be_removed;

        if ((addrTrans.size() > ERROR_LIST_SIZE) &&
            (tbr.end_cycle > current_cycle))
        {
            System.out.println("ERROR : attempt to remove transaction "+
               "prior to time advance beyond end\n"+
               "\tMight want to increase MAX_LIST_SIZE");
            System.out.println("\ttrans end_cycle\t= "+tbr.end_cycle);
            System.out.println("\tCurrent bus cycles\t= "+
               current_cycle);
            return false;
        }

        //
        // Determine usage for cycles to completion of this transaction
        //
        retireTo(tbr.end_cycle);

        addrTrans.remove(to_be_removed);

        return true;
    }

    boolean cycleUsedAddr(long cyc) {
        boolean ret_val = false;
        for (int j = 0 ; j < addrTrans.size() ; j++) {
            ddr2_trans t = (ddr2_trans) addrTrans.elementAt(j);
            // Early Term
            if (cyc < t.addrStart) {
        break;
            }
            // Cycle Used
            if ((t.addrStart <= cyc) &&
        (cyc <= t.addrEnd))
        { return true; }
        }
        return (ret_val);
    }

    boolean cycleUsedData(long cyc) {
        boolean ret_val = false;
        for (int j = 0 ; j < dataTrans.size() ; j++) {
            ddr2_trans t = (ddr2_trans) dataTrans.elementAt(j);
            // Early Term
            if (cyc < t.dataStart) {
        break;
            }
            // Cycle used
            if ((t.dataStart <= cyc) &&
        (cyc <= t.dataEnd))
        { return true; }
        }
        return (ret_val);
    }

    boolean retireTo(long ret_to) {
        if (ret_to < last_retired) { return false; }
        for (long cyc = (last_retired+1) ; cyc <= ret_to ; cyc++) {
            boolean cua = cycleUsedAddr(cyc);
            boolean cud = cycleUsedData(cyc);
            if (cua || cud) {
        used_cycles++;
            }
            if (cua && cud) {
        olap_used_cycles++;
            }
            if (cua) {
        addr_used_cycles++;
            }
            if (cud) {
        data_used_cycles++;
            }
        }
        last_retired = ret_to;
        return true;
```

```
    }

    //
    // ToString
    //
    public String toString(int dt) {
          display_type = dt;
          return this.toString();
    }

    public String toString() {
          String str = new String();

          str += "Schedule ["+super.toString()+"]\n";

          double tot_sched = (double) (reads_sched + writes_sched + refr_sched);

          if ((display_type & STATS) != 0) {
              str += "\tReads Scheduled\t\t: " + reads_sched +"\t"+
          percent8Str(reads_sched / tot_sched) + "\n";
              str += "\tWrites Scheduled\t: "+writes_sched +"\t"+
          percent8Str(writes_sched / tot_sched) + "\n";
              str += "\tRefresh Scheduled\t: "+refr_sched +"\t"+
          percent8Str(refr_sched / tot_sched) + "\n";
              str += "\tAverage Trans Cycles\t: "+
          (total_latency / total_sched) + "\n";
              str += "\tAverage latency\t\t: "+
          ((total_latency / ((double)total_sched)) *
           the_ctrler.clock_period) + "\n";
              str += "\tAdjacent Bank Accesses\t: "+adj_bank_accesses+"\t"+
          percent8Str(adj_bank_accesses / ((double)total_sched)) + "\n";
              str += "\tCurrent cycle\t\t: "+current_cycle+"\n";
              str += "\tLast cycle Retired\t: "+last_retired+"\n";
              str += "\tUsed Cycles\t\t: "+used_cycles+"\n";
              double rc = (double) last_retired;
              str += "\tAddr Used Cycles\t: " + addr_used_cycles +"\t"+
          percent8Str(addr_used_cycles / rc ) + "\n";
              str += "\tData Used Cycles\t: "+data_used_cycles + "\t"+
          percent8Str(data_used_cycles / rc ) + "\n";
              str += "\tOverlap Cycles\t\t: "+olap_used_cycles+"\t"+
          percent8Str(olap_used_cycles / rc ) + "\n";
          }

          if ((display_type & DEBUG) != 0) {
              str += "\nADDRESS TRANSACTIONS:";
              for (int j = (addrTrans.size()-1) ; j >= 0 ; j--) {
          ddr2_trans t = (ddr2_trans)addrTrans.elementAt(j);
          str += "\n"+t.toString(0xFFF);
              }

              str += "\nDATA TRANSACTIONS:";
              for (int j = (dataTrans.size()-1) ; j >= 0 ; j--) {
          ddr2_trans t = (ddr2_trans)dataTrans.elementAt(j);
          str += "\n"+t.toString(0xFFF);
              }
          } // DEBUG

          return str;
    }

    private String percent8Str(double in) {
          if (!(nf instanceof java.text.NumberFormat)) {
              nf = java.text.NumberFormat.getPercentInstance();
              nf.setMinimumFractionDigits(2);
          }
          String ret_str;
          ret_str = SPACES8 + nf.format(in);
          return ret_str.substring(ret_str.length() - 8);
    }

}
```

==

```
// package sdram_sim

/*
  File : trans

  Author : Brian Davis
```

```
  */

public class ddr2_trans extends dram_trans {

    // Class Variables
    static long trans_created = 0;
    static int display_mode = 0;

    // Instance Variable(s)

    //
    // Transaction Attributes
    int type;
    int num_bytes;
    long address;
    private boolean read = false;
    private boolean write = false;
    boolean VCHit = false;
    boolean RCHit = false;
    boolean SAHit = false;
    boolean PrechargeHit = false;
    ddr2_bank access_bank = null;
    vc_buffer vc_used = null;
    long access_row = -1L;

    //
    // Overall Transaction bounds
    long start_cycle;
    long end_cycle;

    //
    // Component Times
    long preStart;
    long rowStart;
    long colStart;
    long addrStart;
    long addrEnd;
    long dataStart, dataEnd;

    long bus_cycles_reqd;

    // Constructor(s)
    ddr2_trans(int rw, long addr, int bytes) {
          this(rw, addr, bytes, -1L, -1L);
    }

    ddr2_trans(int rw, long addr, int bytes, long start, long end) {

          trans_created++;

          type = rw;

          if ((rw & TRANS_READ) != 0)
              read = true;
          else if ((rw & TRANS_WRITE) != 0)
              write = true;

          address = addr;
          num_bytes = bytes;

          start_cycle = start;
          end_cycle = end;

          //
          // Check to verify withing above [start:end] bounds
          rowStart = colStart = dataStart = dataEnd = -1L;
    }

  // Class Methods

  //
  // Instance Methods
  //

    public boolean dataBusReqd() {
          if (((type & TRANS_READ) != 0) || (( type & TRANS_WRITE) != 0)) {
              return true;
          }
          return false;
    }
```

```
public boolean rowStart(long cycle) {
  //
  // Check for potential ERROR conditions
  if (rowStart != -1L) {
    System.out.println("ERROR : multiple definition of Row Start for transaction");
    return true;
  }

  if ((start_cycle != -1L) || (start_cycle > cycle)) {
    System.out.println("ERROR : Illegal definition of row given Start for transaction");
    return true;
  }

  rowStart = cycle;

  start_cycle = cycle;

  return false;
}

public boolean colStart(long cycle) {
  //
  // Check for potential ERROR conditions
  if (colStart != -1L) {
    System.out.println("ERROR : multiple definition of Column Start for transaction");
    return true;
  }

  /*
  if (rowHit == true)
    start_cycle = cycle;
  */

  return false;
}

private boolean startAt(long new_start) {
  // Check for valid start
  if ((end_cycle != 0) && (new_start > end_cycle)) {
    // ERROR ILLEGAL START
    return true;
  } else {
    start_cycle = new_start;
    return false;
  }
}

//
// To String
//
public String toString(int new_mode) {
  display_mode = new_mode;
  return toString();
}

public String toString() {
  String str = new String();
  str += "Transaction ["+super.toString()+"]";

  if ((display_mode & (1<<0)) != 0) {
    str += "\n\tTrans type\t: "+((read) ? "Read" : ((write) ? "Write" : "UNKNOWN"));

    str += "\n\tAddress\t\t: " + java.lang.Long.toHexString(address);

    str += "\n\tNumber Bytes\t: "+num_bytes;

    str += "\n\tHit Status\t: ";
    if (VCHit) {
          str += "Virtual Channel Hit";
    } else if (RCHit) {
          str += "Row Cache Hit";
    } else if (SAHit) {
          str += "SenseAmp Hit";
    } else if (PrechargeHit) {
          str += "Precharge Hit";
    } else {
          str += "SenseAmp/Precharge Miss";
    }

    str += "\n\tTrans Span\t: ( "+start_cycle+" : "+end_cycle+" )";
```

```
      str += "\n\tPre Start\t: "+preStart;

      str += "\n\tRow Start\t: "+rowStart;
      str += "\n\tCol Start\t: "+colStart;
      str += "\n\tAddr Start\t: "+addrStart;
      str += "\n\tAddr End\t: "+addrEnd;
      str += "\n\tData Start\t: "+dataStart;
      str += "\n\tData End\t: "+dataEnd;

    }

    return str;
  } // toString

}
```

A.3 DRDRAM Model

```
// package drdram_sim

/*
  File : DRD_ctrl

  Author : Brian Davis

  */

public class drd_ctrl {

    //
    // Constants
    //
    static final boolean DEBUG = false;
    static final long Tcycle = 1;
    static final long OCTCYCLE = (4*Tcycle);
    static final long Trcd = (7*Tcycle);

    static final int TRANS_READ = (1<<0);
    static final int TRANS_WRITE = (1<<1);
    static final int TRANS_REFRESH = (1<<2);

    static final int CFG_CLOSEPAGEAUTO = (1<<0);
    static final int CFG_OPENPAGE = (1<<1);

    //
    // Controller Parameters
    //
    boolean policy_closeautoprecharge;
    boolean policy_openpage;

    //
    // Timing & Channel Parameters
    //
    long BytePerCycle = 16;
    // 32 mS / 16384 refresh = 1953.125 nS between refresh
    // 1953 / 2.5nS ~= 781
    long refresh_rate = 781;

    // Class Variable
    // Instance Variable(s)
    int display_mode = 0;
    long ctrler_accesses = 0;
    long multi_trans_accesses = 0;
    drd_channel the_channel;
    drd_schedule the_schedule;
    drd_trans last_trans = null;
    long row_shift;

    // Constructor(s)
    drd_ctrl() {
          this(CFG_OPENPAGE);
    }

    drd_ctrl(int cfg) {
          the_channel = new drd_channel();
          row_shift = 10;
```

```
        the_schedule = new drd_schedule(this);
        switch (cfg) {
        case CFG_OPENPAGE:
            policy_closeautoprecharge = false;
            policy_openpage = true;
            break;
        case CFG_CLOSEPAGEAUTO:
            policy_closeautoprecharge = true;
            policy_openpage = false;
            break;
        default:
            System.out.println("Illegal Configuration to drd_ctrl() "+
               "Constructor");
            System.exit(1);
        }
  }

// Class Methods

// Instance Methods
public boolean addDevice(drd_device new_dev) {
  if (new_dev instanceof drd_device) {
    return the_channel.addDevice(new_dev);
  }
  return true;
}

public drd_trans access(long new_time,
          int trans_type, long addr, int num_bytes) {

    advanceClockTo(new_time);
    return access(trans_type, addr, num_bytes);
}

public drd_trans access(int trans_type, long addr, int num_bytes) {

    ctrler_accesses++;

  //
  // Verify valid input
  //
  if ((num_bytes <= 0) ||
        ((trans_type != TRANS_READ) && (trans_type != TRANS_WRITE))) {
    System.out.println("ERROR : Illegal parameters to access() in controller");
    System.exit(1);
  }

  if (!the_channel.isWithin(addr)) {
    System.out.println("ERROR : Address "+Long.toHexString(addr)+
         " not contained within Channel");
    return null;
  }

  //
  // Advance time if req'd
  //
  if ((last_trans instanceof drd_trans) &&
        (last_trans.start_cycle > the_schedule.current_cycle)) {
        advanceClockTo(last_trans.start_cycle);
  }

  //
  // Verify that this transaction does not cross device/bank boundaries
  // or split into multiple transactions
  //

  drd_trans this_trans = null;
  long last_addr = addr + (num_bytes - 1);
  if ((addr >> row_shift) == (last_addr >> row_shift)) {

        // Create Transaction
        this_trans = new drd_trans(trans_type, addr, num_bytes);

        // Initiate Access on channel
        the_channel.access(this_trans);

        // Schedule Transaction
        the_schedule.schedTrans(this_trans);
  } else {
```

```
        multi_trans_accesses++;

        this_trans = split_access(trans_type, addr, num_bytes);

  }

  last_trans = this_trans;

  return this_trans;
} // access

  private drd_trans split_access(int type, long start_addr,
           int num_bytes) {
        drd_trans this_trans = null;
        long last_addr = start_addr + (num_bytes - 1);

        if (DEBUG) {
            String str = "DEBUG(drd_ctrl) : Source of Split Access\n";
            str += "addr = "+Long.toHexString(start_addr)+"\n";
            str += "loc_bytes = "+num_bytes+"\n";
            System.out.println(str);
        }

        long local_addr = start_addr;
        while (local_addr < last_addr) {
            drd_device local_dev = the_channel.which_device(local_addr);
            drd_bank local_bank = local_dev.whichBank(local_addr);
            int local_row = local_bank.rowIndex(local_addr);
            long row_end_addr = local_bank.rowEndAddr(local_row);
            long to_end_of_row = (row_end_addr - local_addr) + 1;
            long to_end_of_trans = (last_addr - local_addr) + 1;
            int loc_bytes = (int) java.lang.Math.min(to_end_of_row,
             to_end_of_trans);

            if (DEBUG) {
        String str = "DEBUG(drd_ctrl) : Split Access\n";
        str += "row = "+local_row+"\n";
        str += "addr = "+Long.toHexString(local_addr)+"\n";
        str += "loc_bytes = "+loc_bytes+"\n";
        System.out.println(str);
            }

            // Create Transaction
            this_trans = new drd_trans(type, local_addr,
               loc_bytes);

            // Initiate Access on channel
            the_channel.access(this_trans);

            // Schedule Transaction
            the_schedule.schedTrans(this_trans);

            local_addr += loc_bytes;
        }
        return this_trans;
  }




//
// Clock Handling Methods
//
public boolean advanceClockTo(double new_clock) {

    long drd_tics = (long)new_clock;

    if ((double)drd_tics < new_clock)
          drd_tics++;

    return(the_schedule.advanceClockTo(drd_tics));
}

public boolean advanceClock(int cycles) {
  return the_schedule.advanceClock(cycles);
}

public boolean advanceClockTo(long new_clock) {
  return the_schedule.advanceClockTo(new_clock);
}
```

```
    public long maxAddr() {
          return the_channel.maxAddr();
    }

    public boolean endSimulation() {
          return the_schedule.endSimulation();
    }

    public long currentTime() {
          return the_schedule.currentTime();
    }

  //
  // To String
  //
  public void printYourself() {
    System.err.println();
    System.err.println("DRDRAM Simulation Parameters");
    System.err.println("\t64Mbit Devices Specified\t: " +
                the_channel.num_devices());
    System.err.println(this.toString(0xFFFFFFFF));
  }

  public String toString(int new_mode) {
    display_mode = new_mode;
    return toString();
  }

  public String toString() {
    String str = new String();
    str += "DRDRAM Controller["+super.toString()+"]\n";
    str += "\tController Policy\t: ";
    if (policy_closeautoprecharge) {
          str += "ClosePageAutoPrecharge\n";
    } else if (policy_openpage) {
          str += "OpenPage\n";
    } else {
          str += "Unknown\n";
    }

    str += "\tNominal Row Size\t: 0x"+Long.toHexString(1<<row_shift)+"\n";
    str += "\tController Accesses\t: "+ctrler_accesses+"\n";
    str += "\tMultiTrans Accesses\t: "+multi_trans_accesses+"\n";

    str += the_schedule.toString(display_mode);
    str += the_channel.toString(display_mode);

    return str;
  } // toString

} // end drd_ctrl class

==========================================================================================

// package drdram_sim

/*
  File : DRD_CHANNEL

  Author : Brian Davis

  */

import java.util.Vector;

public class drd_channel {

    // Constants
    boolean debug = true;
    static final String SPACES8 = "        ";
    static java.text.NumberFormat nf;
    int display_mode = 0;

    // Class Variables

    // Instance Variables
    Vector devices;

    long accesses = 0;
    long read_hits = 0;
    long read_misses = 0;
```

```
  long write_hits = 0;
  long write_misses = 0;

  double total_bytes_transacted = 0.0;

  // Constructors
  drd_channel() {
        devices = new Vector();
  }

// Class Methods

//
// Instance Methods
//
public boolean addDevice(drd_device new_device) {
  if (new_device instanceof drd_device) {
    //
    devices.addElement(new_device);
    return true;
  }
  else
    return false;
}

drd_device which_device(long addr) {
  for (int j = 0 ; j < devices.size() ; j++) {
    drd_device this_dev = (drd_device) devices.elementAt(j);
    if (this_dev.isWithin(addr)) {
        return this_dev;
    }
  }
  return null;
}

public boolean access(drd_trans the_trans) {

  if (!(the_trans instanceof drd_trans)) {
    System.out.println("Invalid parameters to drd_channel access()\n");
    System.exit(1);
  }

  drd_device the_dev = which_device(the_trans.address);

  //
  // Check for repetitive Channel/Transaction Access
  //
  if (the_trans.channel_loc != null) {
    System.out.println("Repetitive Channel Access with single transaction\n");
    System.exit(1);
  } else {
    the_trans.channel_loc = this;
  }

  if (the_dev instanceof drd_device) {
    accesses++;

    total_bytes_transacted += the_trans.num_bytes;

    the_dev.access(the_trans);

    if (the_trans.read) {

        if (the_trans.SAHit)
          read_hits++;
        else
          read_misses++;

    } else if (the_trans.write) {

        if (the_trans.SAHit)
          write_hits++;
        else
          write_misses++;

    }

    return(true);
  }

  return(false);
```

```
}

public boolean isWithin(long addr) {
    for (int j = 0 ; j < devices.size() ; j++) {
          drd_device this_dev = (drd_device) devices.elementAt(j);
          if (this_dev.isWithin(addr))
              return(true);
    }
    return(false);
}

public boolean rowHit(long addr) {
  drd_device the_dev = which_device(addr);
  if (the_dev instanceof drd_device) {
    return the_dev.rowHit(addr);
  }
  return false;
}

public int num_devices() {
  return devices.size();
}

  public long maxAddr() {
        drd_device ld = (drd_device) devices.lastElement();
        return ld.lastAddr();
  }

  public int refreshNextRow() {
        int this_row = -1;
        int ret_row = -1;
        for (int j = 0 ; j < devices.size() ; j++) {
            drd_device dev = (drd_device) devices.elementAt(j);
            this_row = dev.refreshAllBanks();
            if ((debug) &&
        (j != 0) &&
        (this_row != ret_row)) {
        System.out.println("DEBUG : Successive calls to "+
           "dev.refreshAllBanks() return different "+
           "row values");
            }
            if ((ret_row == -1) ||
        (this_row != ret_row)) {
        ret_row = this_row;
            }
        }

        return ret_row;
  }


//
// To String
//
public String toString(int mode) {
  display_mode = mode;
  return this.toString();
}

public String toString() {
  String str = new String();
  str += "DRDRAM Channel ["+super.toString()+"]\n";

  // Number of Devices
  if ((display_mode & (1<<0)) != 0) {
    str += "\tChannel contains "+devices.size()+" device(s)\n";
  }

  // Accesses
  if ((display_mode & (1<<1)) != 0) {
    str += "\tChan Accesses\t\t: "+accesses+"\n";
    str += "\tChan Access Hits\t\t: "+(read_hits + write_hits)+"\t\t";
    str += percent8Str((read_hits + write_hits)/((double)accesses))+"\n";
    str += "\tChan Avg access bytes\t: "+
        (total_bytes_transacted/((double)accesses))+"\n";

    if (accesses != 0) {

        long reads = (read_hits + read_misses);
        str += "\tChan Reads\t\t\t: "+reads+"\t\t";
        str += percent8Str(reads/((double)accesses))+"\n";
```

```
          if (reads != 0) {
            str += "\tChan Read Hits\t\t: "+read_hits+"\t\t";
            str += percent8Str(read_hits/((double)accesses))+"\t\t";
           str += percent8Str(read_hits/((double)reads))+"\n";
            str += "\tChan Read Misses\t: "+read_misses+"\t\t";
            str += percent8Str(read_misses/((double)accesses))+"\t\t";
            str += percent8Str(read_misses/((double)reads))+"\n";
          } // reads

          long writes = (write_hits + write_misses);
          str += "\tChan Writes\t\t\t: "+writes+"\t\t";
          str += percent8Str(writes/((double)accesses))+"\n";
          if (writes != 0) {
            str += "\tChan Write Hits\t\t: "+write_hits+"\t\t";
            str += percent8Str(write_hits/((double)accesses))+"\t\t";
           str += percent8Str(write_hits/((double)writes))+"\n";
            str += "\tChan Write Misses\t\t: "+write_misses+"\t\t";
            str += percent8Str(write_misses/((double)accesses))+"\t\t";
            str += percent8Str(write_misses/((double)writes))+"\n";
          } // Writes

      } // If Accesses

    }

    // Device Info
    if ((display_mode & (1<<4)) != 0) {
      for (int j = 0 ; j < devices.size() ; j++) {
          drd_device this_dev = (drd_device) devices.elementAt(j);
          str += this_dev.toString(display_mode);
      }
    }
    return str;
  }

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

}
```

==

```
// package drdram_sim

/*
  File : DRD_DEVICE
  */

import java.math.BigInteger;

public class drd_device {

    //
    // Constants
    //
    // Sizes described in bits, numerically in bytes
    static final boolean DEBUG = false;
    static final long RIMM_64MB_8by64Mb = (1<<26) | (1<<0);
    static final long RIMM_128MB_16by64Mb = (1<<27) | (1<<0);
    static final long RIMM_128MB_4by128Mb = (1<<27) | (1<<1);
    static final long RIMM_256MB_8by128Mb = (1<<28) | (1<<1);
    static final int access_gran = 16;
    static final String SPACES8 = "        ";
    static java.text.NumberFormat nf;
    int display_mode = 0;

    // Class Variables
    static int devices_created = 0;

    // Instance Variables
    drd_ctrl the_ctrler = null;
    private long first_addr, last_addr;
    long rimm_size;
    long device_size;
```

```
  int num_devices;
  long device_mask;
  int device_shift;
  int num_banks;
  long bank_size;
  long bank_mask;
  int bank_shift;
  int num_rows;
  long row_size;
  drd_bank [] banks;

  // Statistic maint variables
  long accesses = 0;
  long page_crossing_accesses = 0;
  long read_hits = 0;
  long read_misses = 0;
  long write_hits = 0;
  long write_misses = 0;

  // Constructors
  drd_device(long rimm_type, long start_addr, drd_ctrl ctrl) {
        devices_created++;
        the_ctrler = ctrl;

        if (rimm_type == RIMM_64MB_8by64Mb) {
            rimm_size = ((0x1L)<<26);
            num_devices = 8;
            device_size = rimm_size / num_devices;
            num_banks = 8*16;
            bank_size = rimm_size / num_banks;
            num_rows = 8*16* 512;
            row_size = rimm_size / num_rows;
        } else {
              System.out.println("Illegal rimm_type to DRD_DEVICE "+
         "constructor\n");
              System.exit(1);
        }

        // Dependant variables
        first_addr = start_addr;
        //System.out.println("@@@ device_created : start address parameter ="+start_addr+"\n");
        last_addr = first_addr + (rimm_size - 1);

        // device
        device_mask = (rimm_size -1) & (~(device_size-1));
        String s1 = (new Long(device_mask)).toString();
        device_shift = (new BigInteger(s1)).getLowestSetBit();
        // bank
        bank_mask = (rimm_size - 1) & (~(bank_size-1));
        String s2 = (new Long(bank_mask)).toString();
        bank_shift = (new BigInteger(s2)).getLowestSetBit();

        if (DEBUG) {
            System.out.println("device_created : \n"+this.toString(1));
        }

        // Create & Initialize Banks
        banks = new drd_bank[num_banks];
        long bank_start = first_addr;
        for (int j = 0 ; j < banks.length ; j++) {
            banks[j] = new drd_bank(bank_size, bank_start, the_ctrler);
            bank_start += bank_size;
        }

        // Set up Adjoining banks
        for (int dev = 0 ; dev < num_devices ; dev++) {
            int banks_per_dev = (num_banks/num_devices);
            for (int bank = 0 ; bank < banks_per_dev ; bank++) {
        drd_bank prev = ((bank == 0) ? null :
         banks[(dev * banks_per_dev) + (bank-1)]);
        drd_bank cur = banks[(dev * banks_per_dev) + bank];
        drd_bank next = ((bank == (banks_per_dev-1)) ? null :
         banks[(dev * banks_per_dev) + (bank+1)]);

        cur.setAdjoining(prev, next);
            } // bank
        } // dev

  } // constructor

//
```

```
// Class Methods
//

  public long lastAddr() {
        return last_addr;
  }

  // Instance Methods
  public boolean isWithin(long addr) {
        return ((first_addr <= addr) &&
        (addr <= last_addr));
  }

  int bankIndex(long addr) {
        int bank_index = (int)((addr & bank_mask) >> bank_shift);
        return bank_index;
  }

public drd_bank whichBank(long addr) {
    int bank_index = bankIndex(addr);
    if ((bank_index >= 0) && (bank_index < banks.length))
          return banks[bank_index];
    return null;
}

public boolean rowHit(long addr) {
  drd_bank the_bank = whichBank(addr);
  if (the_bank instanceof drd_bank)
    return the_bank.rowHit(addr);
  return false;
}

/*
** access()
**
** process a request which entails:
**   (1) Enter into the schedule the busy slots cooresponding to this
**       access
**   (2) Add to the statistics for this device based on the hit/miss
**       status of this access
*/
public boolean access(drd_trans the_trans) {

  long addr = the_trans.address;
  int num_bytes = the_trans.num_bytes;

  //
  // Check for repetitive Device/Transaction Access
  //
  if (the_trans.device_loc != null) {
    System.out.println("Repetitive Device Access with single transaction\n");
    System.exit(1);
  } else {
    the_trans.device_loc = this;
  }

  //
  // Check for row-crossing accesses
  //
  if (whichBank(addr) != whichBank(addr+(num_bytes-1))) {
        page_crossing_accesses++;
        //
        // Check that it only crosses a single page boundary
        //
        int index_dif = bankIndex(addr+(num_bytes-1)) - bankIndex(addr);
        if (index_dif > 1) {
            System.out.println("ERROR : Access illegal due access of at least 3 banks");
            System.out.println("\tAddress\t= "+Long.toHexString(addr));
            System.out.println("\tnum bytes\t= "+num_bytes);
            return false;
        }

        //
        // Generate info for two transactions
        // the_trans must be last for latency reasons
        //
        // long first_bank_end = whichBank(addr) ;
            //first_trans_bytes =
            // if (DEBUG) {
            // }
            // the_trans.num_b
```

```
      //
      if (false) {
          System.out.println("ERROR : Access illegal due to data from multiple-banks");
          System.out.println("\tAddress\t= "+Long.toHexString(addr));

          drd_bank bank = whichBank(addr);
          System.out.println("\tFirst Byte Bank\t= " +
             ((bank instanceof drd_bank) ?
      bank.toString(1<<0) : "NULL"));

          bank = whichBank(addr+num_bytes);
          System.out.println("\tLast Byte Bank\t= " +
             ((bank instanceof drd_bank) ?
      bank.toString(1<<0) : "NULL"));

          System.out.println("FROM : "+this.toString(1<<0));

      }

  return false;
} // multiple banks

//
// First check for correct access pattern
//
if (!isWithin(addr)) {
  System.out.println("ERROR : Access of device not holding address");
  return false;
}

accesses++;

int bank_index = (int)((addr & bank_mask) >> bank_shift);
if (bank_index >= banks.length) {
  System.out.println("ERROR : Invalid bank_index ("+
       bank_index+" MAX is "+(banks.length-1)+
       ") in device.access()");
  System.exit(1);
}
drd_bank which_bank = banks[bank_index];
if (DEBUG) {
      System.out.println("DEBUG(drd_device) bank_index = "+bank_index);
}
if (!which_bank.isWithin(addr)) {
  System.out.println("\nERROR(drd_dev.access() : "+
       "Access of bank not holding address");
  System.out.println("Access Address : 0x"+
       java.lang.Long.toHexString(addr));
  System.out.println("Bank Index = "+bank_index);
  System.out.println("Bank Mask = 0x"+
       java.lang.Long.toHexString(bank_mask));
  System.out.println("Bank shift = "+bank_shift);
  System.out.println("Bank Spans (0x"+
       java.lang.Long.toHexString(which_bank.first_addr)+
       " : 0x"+
       java.lang.Long.toHexString(which_bank.last_addr)+")"
       );
  System.out.println("RIMM Spans (0x"+
       java.lang.Long.toHexString(first_addr)+
       " : 0x"+
       java.lang.Long.toHexString(last_addr)+")");
  return false;
}

which_bank.access(the_trans);

if (the_trans.read) {

  if (the_trans.SAHit)
      read_hits++;
  else
      read_misses++;

} else if (the_trans.write) {

  if (the_trans.SAHit)
      write_hits++;
  else
      write_misses++;
```

```
  }

  return true;
} // access

  //
  // RefreshAllBanks()
  //
  int refreshAllBanks() {
        int this_row = -1;
        int ret_row = -1;
        for (int j = 0 ; j < banks.length ; j++) {
            this_row = banks[j].refreshNextRow();
            if ((DEBUG) &&
        (ret_row != -1) &&
        (this_row != ret_row)) {
        System.out.println("DEBUG : Successive calls to "+
           "bank.refreshNextRow() return different "+
           "row values");
            }
            if ((ret_row == -1) ||
                (this_row != ret_row)) {
        ret_row = this_row;
            }
        }
        return ret_row;
  } // refresh All Banks

//
// To String
//
public String toString(int mode) {
  display_mode = mode;
  return this.toString();
}

public String toString() {
  String str = new String();
  str += "DRDRAM Device ["+super.toString()+"]\n";

  // Address Span
  if ((display_mode & (1<<0)) != 0) {
    str += "\tDevice Spans\t\t: (0x" +
          java.lang.Long.toHexString(first_addr) + " : 0x" +
          java.lang.Long.toHexString(last_addr)+")\n";
    str += "\tRimm Size\t\t: 0x"+java.lang.Long.toHexString(rimm_size);
    str += "\n\tDevice Size\t\t: 0x"+java.lang.Long.toHexString(device_size);
    str += "\n\tNum Devices\t\t: "+num_devices;
    str += "\n\tDevice Mask\t\t: 0x"+java.lang.Long.toHexString(device_mask);
    str += "\n\tDevice Shift\t\t: "+device_shift;
    str += "\n\tBank Size\t\t: 0x"+java.lang.Long.toHexString(bank_size);
    str += "\n\tNum Banks\t\t: "+num_banks;
    str += "\n\tBank Mask\t\t: 0x"+java.lang.Long.toHexString(bank_mask);
    str += "\n\tBank Shift\t\t: "+bank_shift+"\n";


  }

  // Access numbers
  if ((display_mode & (1<<1)) != 0) {
    str += "\tDevice Accesses\t: "+accesses+"\n";
    if (accesses != 0) {

        long reads = (read_hits + read_misses);
        str += "\tDev Reads\t\t: "+reads+"\t\t";
        str += percent8Str(reads/((double)accesses))+"\n";
        if (reads != 0) {
          str += "\tDev Read Hits\t\t: "+read_hits+"\t\t";
          str += percent8Str(read_hits/((double)accesses))+"\t\t";
         str += percent8Str(read_hits/((double)reads))+"\n";
          str += "\tDev Read Misses\t\t: "+read_misses+"\t\t";
          str += percent8Str(read_misses/((double)accesses))+"\t\t";
          str += percent8Str(read_misses/((double)reads))+"\n";
        } // reads

        long writes = (write_hits + write_misses);
        str += "\tDev Writes\t\t: "+writes+"\t\t";
        str += percent8Str(writes/((double)accesses))+"\n";
        if (writes != 0) {
          str += "\tDev Write Hits\t\t: "+write_hits+"\t\t";
          str += percent8Str(write_hits/((double)accesses))+"\t\t";
```

```
          str += percent8Str(write_hits/((double)writes))+"\n";
            str += "\tDev Write Misses\t\t: "+write_misses+"\t\t";
            str += percent8Str(write_misses/((double)accesses))+"\t\t";
            str += percent8Str(write_misses/((double)writes))+"\n";
          } // Writes

      } // If Accesses

    } // Display Mode

    // Individual Bank info
    if ((display_mode & (1<<4)) != 0) {
      for (int j = 0 ; j < banks.length ; j++) {
          drd_bank this_bank = banks[j];
          str += this_bank.toString(display_mode);
      }
    }
    return str;
  } // toString

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

}

=====================================================================================================

// package drdram_sim

/*
  File : DRD_BANK

  Author : Brian Davis

  */

import java.math.BigInteger;

public class drd_bank {

    //
    // Constants
    //
    // Sizes described in bits, numerically in bytes
    static final boolean DEBUG = false;
    final long BANK_SIZE_512KB = (1<<19); // 2^19 Bytes
    static final String SPACES8 = "        ";
    static java.text.NumberFormat nf;

    // Class Variables
    static int banks_created = 0;

    // Instance Variables
    int display_mode = 0;
    long first_addr, last_addr;
    long row_mask, row_shift;
    long row_size;
    int num_rows;
    int last_refreshed_row = -1;
    drd_ctrl the_ctrler = null;
    drd_bank prev_bank = null;
    drd_bank next_bank = null;
    // bank-state
    boolean precharged = false;
    boolean validSA = false;
    int sa_contains = 0;

    // Statistic Variables
    long accesses = 0;
    long read_hits = 0;
    long read_misses = 0;
    long write_hits = 0;
    long write_misses = 0;
```

```
// Constructor(s)
drd_bank(long bank_size, long start_addr, drd_ctrl ctrl) {
      banks_created++;

      the_ctrler = ctrl;

      // set mask & shift as well as checking for legal parameters
      if (bank_size == BANK_SIZE_512KB) {
          num_rows = 512;
          row_size = bank_size / num_rows;
      } else {
          System.out.println("Illegal Parameters to DRD_BANK constructor\n");
          System.exit(1);
      }

      first_addr = start_addr;
      last_addr = start_addr + bank_size - 1;

      row_mask = (bank_size-1) & (~(row_size-1));
      String s1 = (new Long(row_mask)).toString();
      row_shift = (new BigInteger(s1)).getLowestSetBit();

      if (DEBUG) {
          System.out.println("DEBUG(drd_bank)\n\t"+
             "\tbank_size\t: "+bank_size+"\n"+
             "\tnum rows\t: "+num_rows+"\n"+
             "\trow_size\t: "+row_size+"\n"+
             "\trow_mask\t: 0x"+
             java.lang.Long.toHexString(row_mask)+"\n"+
             "\trow_shift\t: "+row_shift);
      }

      if (false && DEBUG) {
          System.out.println("DEBUG(drd_bank): Bank created spanning (0x"+
             java.lang.Long.toHexString(first_addr)+
             " : 0x"+
             java.lang.Long.toHexString(last_addr)+")");
      }

} // constructor

// Class Methods

// Instance Methods
public boolean isWithin(long addr) {
      return ((first_addr <= addr) &&
      (addr <= last_addr));
}

public boolean access(drd_trans the_trans) {
      long addr = the_trans.address;

      if (!isWithin(addr)) {
          System.out.println("Access of bank not holding address\n");
          return false;
      }

      if (the_trans.access_bank != null) {
          System.out.println("Repetitive Bank Access with"+
             " single transaction\n");
          System.exit(1);
      } else {
          the_trans.access_bank = this;
      }

      this.accesses++;

      if (DEBUG) {
          System.out.println("DEBUG(drd_bank)\taccess(0x"+
             java.lang.Long.toHexString(addr)+
             ") goes to Row "+
             rowIndex(addr));
      }

      if (rowHit(the_trans.address)) {
          the_trans.SAHit = true;
      } else {
          the_trans.SAHit = false;
      }

      if (the_trans.read) {
```

```
            if (the_trans.SAHit)
        read_hits++;
            else
        read_misses++;
        } else if (the_trans.write) {
            if (the_trans.SAHit)
        write_hits++;
            else
        write_misses++;
        }

        //
        // Set up Row Cache Validity
        //
        if (the_ctrler.policy_openpage) {
            sa_contains = rowIndex(addr);
            validSA = true;
            precharged = false;
        } else if (the_ctrler.policy_closeautoprecharge) {
            precharged = true;
            validSA = false;
        } else {
            System.out.println("ERROR : Unknown Policy in drd_bank.access()");
            System.exit(1);
        }

        // Set Adjoining rowCacheValid False
        if (prev_bank instanceof drd_bank)
            prev_bank.validSA = false;
        if (next_bank instanceof drd_bank)
            next_bank.validSA = false;

        return true;
  }

  public void setAdjoining(drd_bank prev, drd_bank next) {
        prev_bank = prev;
        next_bank = next;
  }

  public boolean rowHit(long addr) {
        int cur_row = rowIndex(addr);
        return (validSA && (sa_contains == cur_row));
  }

  public int rowIndex(long addr) {
        return ((int)((addr & row_mask) >> row_shift));
  }

  long rowEndAddr(int row) {
        if ((row < 0) || (row >= num_rows)) {
            System.out.println("ERROR(sdram_bank) : attempt to find EndAddr "+
               "of out-of-bounds row");
            return(-1L);
        }
        long re_addr = first_addr + ((row + 1) * row_size) - 1;
        return re_addr;
  }

  int refreshNextRow() {
        int ret_row = ((last_refreshed_row + 1) % num_rows);
        //
        // Process refresh
        //
        last_refreshed_row = ret_row;
        precharged = true;
        validSA = false;
        sa_contains = -1;
        //
        // Return row precharged
        //
        return ret_row;
  }


//
// To String
//
public String toString(int mode) {
  display_mode = mode;
  return this.toString();
```

```
  }

  public String toString() {

    String str = new String();
    str += "DRDRAM Bank ["+super.toString()+"]\n";

    // Address Span
    if ((display_mode & (1<<0)) != 0) {
      str += "\tBank Spans\t: (0x" + java.lang.Long.toHexString(first_addr) +
            " : 0x" +
            java.lang.Long.toHexString(last_addr)+")\n";
      str += "\tNumber of rows\t: "+num_rows+"\n";
      str += "\trow size\t: "+row_size+"\n";
      str += "\tRow Mask\t: 0x"+java.lang.Long.toHexString(row_mask)+"\n";
      str += "\tRow Shift\t: "+row_shift+"\n";
    }

    // Access numbers
    if ((display_mode & (1<<1)) != 0) {
      str += "\tBank Accesses\t: "+accesses+"\n";

      if (accesses != 0) {

          long reads = (read_hits + read_misses);
          str += "\tBank Reads\t; "+reads+"\t\t";
          str += percent8Str(reads/((double)accesses))+"\n";
          if (reads != 0) {
            str += "\tBank Read Hits\t: "+read_hits+"\t\t";
            str += percent8Str(read_hits/((double)accesses))+"\t\t";
           str += percent8Str(read_hits/((double)reads))+"\n";
            str += "\tBank Read Misses\t: "+read_misses+"\t\t";
            str += percent8Str(read_misses/((double)accesses))+"\t\t";
            str += percent8Str(read_misses/((double)reads))+"\n";
          } // reads

          long writes = (write_hits + write_misses);
          str += "\tBank Writes\t\t: "+writes+"\t\t";
          str += percent8Str(writes/((double)accesses))+"\n";
          if (writes != 0) {
            str += "\tBank Write Hits\t: "+write_hits+"\t\t";
            str += percent8Str(write_hits/((double)accesses))+"\t\t";
           str += percent8Str(write_hits/((double)writes))+"\n";
            str += "\tBank Write Misses\t: "+write_misses+"\t\t";
            str += percent8Str(write_misses/((double)accesses))+"\t\t";
            str += percent8Str(write_misses/((double)writes))+"\n";
          } // Writes

      } // If Accesses

    }

    return str;
  } // toString

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

}
==========================================================================================================

// package drdram_sim

/*
  File : DRD_schedule

  Author : Brian Davis

  */

import java.util.Vector;

public class drd_schedule {
```

```
  // Debug Flags
  static final boolean DEBUG = false;
  static final boolean ARRAYDEBUG = false;
  static final int ERROR_LIST_SIZE = 500;
  static final int MAX_LIST_SIZE = 10;
  static final String SPACES8 = "        ";
  static java.text.NumberFormat nf;

  // Constants
  static final long Tcycle = drd_ctrl.Tcycle;
  static final double cycleNs = 2.5E-9;
  static final long OCTCYCLE = drd_ctrl.OCTCYCLE;
  static final long Tcc = 4 * Tcycle;
  static final long Tcac = 8 * Tcycle;
  static final long Tcwd = 6 * Tcycle;
  static final long Trcd = 7 * Tcycle;
  static final long Trc = 28 * Tcycle;
  static final long Trp = 8 * Tcycle;
  static final long Tras = 20 * Tcycle;
  static final long Trdly = 0; // Dependant upon the # of devices in Channel

  // Parameters
  boolean data_sched = false;

  // Class Variables

  // Instance Variable(s)
  drd_ctrl the_ctrler;
  Vector rowTrans,
        colTrans,
        dataTrans;
  long current_cycle = -1;
  long retired_cycle = 0;
  long overlap_cycles = 0;
  long utilized_cycles = 0;
  double trans_cycles = 0.0;
  long reads_sched = 0;
  long writes_sched = 0;
  long refr_sched = 0;
  long total_sched = 0;
  double total_latency = 0.0;
  long adj_bank_accesses = 0;
  // Refresh
  private long last_refresh_time = 0;
  private long last_refresh_iter = 0;
  // Display
  int display_mode = 0;


  // Constructor(s)
  drd_schedule(drd_ctrl ctloc) {
        rowTrans = new Vector();
        colTrans = new Vector();
        dataTrans = new Vector();
        the_ctrler = ctloc;
  }

// Class Methods

  // Instance Methods

  //
  // Schedule Transaction
  //
  public long schedTrans(drd_trans new_trans) {

        long l_pre_start = -1;
        long l_row_start = -1;
        long l_row_end = -1;
        long l_col_start = -1;
        long l_col_end = -1;
        long l_data_start = -1;
        long l_data_end = -1;
        long l_start = -1;
        long l_end = -1;

        long earliest_possible = (current_cycle > 0 ) ? current_cycle : 0;

        //
        // Prior transactions
```

```
//
drd_trans prev_r_trans = lastRowTrans();
drd_trans prev_c_trans = lastColTrans();
drd_trans prev_d_trans = lastDataTrans();

//
// Determine # of OCTCYCLES req'd for data Xfer
long data_cycles = (long)
    (java.lang.Math.ceil(((double)new_trans.num_bytes)/
 the_ctrler.BytePerCycle));

//
// Determine slot for row / act packets
//
if (new_trans.read) {
    reads_sched++;
    if ((!new_trans.SAHit) &&
(!new_trans.BankPrecharged)) {
//
// Must do precharge
//
if ((prev_r_trans instanceof drd_trans) &&
    (prev_r_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_r_trans.rowEnd;
}
l_pre_start = earliest_possible;
l_row_start = l_pre_start + Trp;
l_row_end = l_row_start + OCTCYCLE;
l_col_start = l_row_start + Trcd;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcac;
l_data_end = l_data_start + data_cycles;
l_start = l_pre_start;
l_end = l_data_end;
    } else if ((!new_trans.SAHit) &&
       (new_trans.BankPrecharged)) {
//
// already precharged, but must access row
//
if ((prev_r_trans instanceof drd_trans) &&
    (prev_r_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_r_trans.rowEnd;
}
l_pre_start = -1;
l_row_start = earliest_possible;
l_row_end = l_row_start + OCTCYCLE;
l_col_start = l_row_start + Trcd;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcac;
l_data_end = l_data_start + data_cycles;
l_start = l_row_start;
l_end = l_data_end;
    } else if (new_trans.SAHit) {
//
// Sense Amp Hit
//
// Row bus component not used!
prev_r_trans = null;
if ((prev_c_trans instanceof drd_trans) &&
    (prev_c_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_c_trans.rowEnd;
}
l_pre_start = -1;
l_row_start = -1;
l_row_end = -1;
l_col_start = earliest_possible;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcac;
l_data_end = l_data_start + data_cycles;
l_start = l_col_start;
l_end = l_data_end;
    } else {
System.out.println("ERROR : Logical impossibility");
System.exit(1);
    }
} else if (new_trans.write) {
    //
    // new_trans.read == false
    // presume that new_trans is a write transaction
    //
    writes_sched++;
```

```
    if ((!new_trans.SAHit) &&
(!new_trans.BankPrecharged)) {
//
// Must do precharge
//
if ((prev_r_trans instanceof drd_trans) &&
    (prev_r_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_r_trans.rowEnd;
}
l_pre_start = earliest_possible;
l_row_start = l_pre_start + Trp;
l_row_end = l_row_start + OCTCYCLE;
l_col_start = l_row_start + Trcd;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcwd;
l_data_end = l_data_start + data_cycles;
l_start = l_pre_start;
l_end = l_data_end;
    } else if ((!new_trans.SAHit) &&
       (new_trans.BankPrecharged)) {
//
// already precharged, but must access row
//
if ((prev_r_trans instanceof drd_trans) &&
    (prev_r_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_r_trans.rowEnd;
}
l_pre_start = -1;
l_row_start = earliest_possible;
l_row_end = l_row_start + OCTCYCLE;
l_col_start = l_row_start + Trcd;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcwd;
l_data_end = l_data_start + data_cycles;
l_start = l_row_start;
l_end = l_data_end;
    } else if (new_trans.SAHit) {
//
// Sense Amp Hit
//
// Row bus component not used!
prev_r_trans = null;
if ((prev_c_trans instanceof drd_trans) &&
    (prev_c_trans.rowEnd > earliest_possible)) {
    earliest_possible = prev_c_trans.rowEnd;
}
l_pre_start = -1;
l_row_start = -1;
l_row_end = -1;
l_col_start = earliest_possible;
l_col_end = l_col_start + OCTCYCLE;
l_data_start = l_col_end + Tcwd;
l_data_end = l_data_start + data_cycles;
l_start = l_col_start;
l_end = l_data_end;
    } else {
System.out.println("ERROR : Logical impossibility : "+
   "Impossible condition in schedule Write");
System.exit(1);
    }
} /* write */
else {
    System.out.println("ERROR : Logical impossibility : "+
       "neither read nor write");
    System.exit(1);
}

/*
** Check for conflicts with prior accesses
** (prev_a_trans & prev_d_trans) and advance ALL l_* vars
** if conflict exists
*/

long row_spacing = 0;
long col_spacing = 0;
long data_spacing = 0;
long conflict_delta = 0;

/*
** Determine addr spacing from adjacency, bank & access type
*/
```

```
if ((prev_r_trans instanceof drd_trans) &&
    (prev_r_trans.access_bank == new_trans.access_bank)) {

    row_spacing = Tras - OCTCYCLE;
}
if ((prev_c_trans instanceof drd_trans) &&
    (prev_c_trans.access_bank == new_trans.access_bank)) {

    col_spacing = 0;
    // Added 07/07/00 BTD
    if (prev_c_trans.dataBusReqd() && new_trans.dataBusReqd()) {
adj_bank_accesses++;
    }
}
if ((prev_d_trans instanceof drd_trans) &&
    (prev_d_trans.access_bank == new_trans.access_bank)) {

    data_spacing = 0;
}

//
// Must check for time conflicts between adjacent accesses
//
if ((prev_r_trans instanceof drd_trans) &&
    (l_row_start < (prev_r_trans.rowEnd + row_spacing))) {
    conflict_delta = (prev_r_trans.rowEnd + row_spacing) -
l_row_start;
}
if ((prev_c_trans instanceof drd_trans) &&
    (l_col_start < (prev_d_trans.colEnd + col_spacing))) {
    long col_delta = (prev_d_trans.colEnd + col_spacing) -
l_col_start;
    if (col_delta > conflict_delta) {
conflict_delta = col_delta;
    }
}
if ((prev_d_trans instanceof drd_trans) &&
    (l_data_start < (prev_d_trans.dataEnd + data_spacing))) {
    long data_delta = (prev_d_trans.dataEnd + data_spacing) -
l_data_start;
    if (data_delta > conflict_delta) {
conflict_delta = data_delta;
    }
}

if (conflict_delta > 0) {
    if (l_pre_start >= 0)
l_pre_start += conflict_delta;
    if (l_row_start >= 0)
l_row_start += conflict_delta;
    if (l_row_end >= 0)
l_row_end += conflict_delta;
    if (l_col_start >= 0)
l_col_start += conflict_delta;
    if (l_col_end >= 0)
l_col_end += conflict_delta;
    if (l_data_start >= 0)
l_data_start += conflict_delta;
    if (l_data_end >= 0)
l_data_end += conflict_delta;
    l_start += conflict_delta;
    l_end += conflict_delta;
}

new_trans.preStart = l_pre_start;
new_trans.rowStart = l_row_start;
new_trans.rowEnd = l_row_end;
new_trans.colStart = l_col_start;
new_trans.colEnd = l_col_end;
new_trans.dataStart = l_data_start;
new_trans.dataEnd = l_data_end;
new_trans.start_cycle = l_start;
new_trans.end_cycle = l_end;

if (l_row_start > 0)
    addToRowTrans(new_trans);
if (l_col_start > 0)
    addToColTrans(new_trans);
if (l_data_start >0)
    addToDataTrans(new_trans);
```

```
      //
      // Update Latency metric values
      //
      total_sched++;
      long this_latency = (new_trans.end_cycle - new_trans.start_cycle);
      total_latency += this_latency;
      if (this_latency > 999) {
          System.out.println("ERROR (arbitrary) : this_latency > 999 in "+
             "drd_schedule.schedTrans()");
      }

      if (DEBUG) {
          System.out.println("DEBUG(drd_schedule) : Transaction Scheduled\n"+
             new_trans.toString(1));
      }

      return new_trans.start_cycle;
} // scheduleTrans

public long scheduleRefresh(long cycle, long row) {
      long l_pre_start = -1;
      long l_row_start = -1;
      long l_start = -1;
      long l_end = -1;

      drd_trans prev_r_trans = lastRowTrans();

      if (DEBUG) {
          System.out.println("In scheduleRefresh, Prior Row Trans is :\n"+
             ((prev_r_trans instanceof drd_trans) ?
      prev_r_trans.toString(0xFFF) :
      "NULL"));
      }

      long refr_cycle = cycle;
      if ((prev_r_trans instanceof drd_trans) &&
          (refr_cycle < (prev_r_trans.rowEnd + Tras))) {
          refr_cycle = (prev_r_trans.rowEnd + Tras);
      }

      if (DEBUG) {
          System.out.println("In scheduleRefresh, refr_cycle = "+refr_cycle);
      }

      l_pre_start = refr_cycle;
      l_row_start = l_pre_start + Trp;
      l_start = l_pre_start;
      l_end = l_row_start+OCTCYCLE;

      // Create Refresh Transaction
      drd_trans refr_trans = new drd_trans(drd_trans.TRANS_REFRESH);
    refr_trans.preStart = l_pre_start;
      refr_trans.rowStart = l_row_start;
      refr_trans.rowEnd = l_end;
      refr_trans.start_cycle = l_start;
      refr_trans.end_cycle = l_end;

      if (DEBUG) {
          System.out.println("DEBUG(drd_schedule) : Refresh Scheduled\n"+
             refr_trans.toString(1));
      }
      refr_sched++;
      addToRowTrans(refr_trans);

      return refr_trans.start_cycle;
}

public drd_trans lastRowTrans() {
      return lastTrans(rowTrans);
}

public drd_trans lastColTrans() {
      return lastTrans(colTrans);
}

public drd_trans lastDataTrans() {
      return lastTrans(dataTrans);
}

public drd_trans lastTrans(Vector v) {
      drd_trans lt = null;
```

```
        try {
            lt = (drd_trans)v.lastElement();
        } catch (java.util.NoSuchElementException e) {
            // Do nothing it remains
            // prev_a_trans = null;
        }
        return lt;
  }

//
// Add Transaction to Vector
//
public void addTransToList(drd_trans newTrans, Vector vect) {
    while (vect.size() >= MAX_LIST_SIZE) {
          drd_trans tbr = (drd_trans) vect.elementAt(0);
          if (tbr.end_cycle > current_cycle) {
              if (vect.size() > ERROR_LIST_SIZE) {
          System.out.println("ERROR : Trans Vector size ("+
             vect.size()+
             ") exceeds MAX : while first element"+
             " end cycle greater than "+
             "current_cycle");
          // System.out.println("\tVector size = "+vect.size());
              }
              break;
          } else {
              vect.removeElement(tbr);
          }
    }
    vect.addElement(newTrans);
}

  public void addToRowTrans(drd_trans newTrans) {
        addTransToList(newTrans, rowTrans);
  }

//
// Add Transaction to Column Array
//
public void addToColTrans(drd_trans newTrans) {
  // addElement() adds to the end of the vector
    //colTrans.addElement(newTrans);
    addTransToList(newTrans, colTrans);
}

//
// Add Transaction to Data Array
//
public void addToDataTrans(drd_trans newTrans) {
  // addElement() adds to the end of the vector
    // dataTrans.addElement(newTrans);
    addTransToList(newTrans, dataTrans);

}

//
// Display the Array of Row Transactions
//
public void displayRowTrans(int mark) {
    String str = new String();
    str += "=== ROW ARRAY ===\n";
    for (int j = 0 ; j < rowTrans.size() ; j++) {
        drd_trans jth_trans = (drd_trans) rowTrans.elementAt(j);
        str += "Row["+j+"] = ( "+jth_trans.rowStart+" : "+(jth_trans.rowStart+OCTCYCLE)+" )";
        str += ((mark == j) ? " <-- NEW\n" : "\n");
    }
    System.out.print(str);
}

//
// Display the array of Data Transactions
//
public void displayDataTrans(int mark_index) {
  String str = new String();
  str += "=== DATA ARRAY ===\n";
  for (int j = 0 ; j < dataTrans.size() ; j++) {
    drd_trans jth_trans = (drd_trans) dataTrans.elementAt(j);
    str += "Data["+j+"] = ( "+jth_trans.dataStart+" : "+jth_trans.dataEnd+" )";
    str += ((mark_index == j) ? " <-- NEW\n" : "\n");
  }
  System.out.print(str);
```

```
}

//
// Advance Clock
//
public boolean advanceClock(int cycles) {
  if (cycles >= 0) {
    advanceClockTo(current_cycle + cycles);
    return false;
  }
  return true;
}

  long currentTime() {
        return current_cycle;
  }

  public boolean advanceClockTo(long new_clock) {
        if (new_clock < current_cycle) return true;
        if (DEBUG) {
            System.out.println("DEBUG(drd_schedule) : "+
               "advancing clock to "+new_clock);
        }
        //
        // Check for refresh in the interval between current_cycle
        // and retire_to
        //
        for (long r_iter = (last_refresh_iter+1) ;
             r_iter < (new_clock/the_ctrler.refresh_rate) ;
             r_iter += 1) {
            int refr_row = the_ctrler.the_channel.refreshNextRow();
            last_refresh_iter = r_iter;
            long refr_time = ((current_cycle <
               (r_iter*the_ctrler.refresh_rate)) ?
              (r_iter*the_ctrler.refresh_rate) :
              current_cycle);
            last_refresh_time =
        scheduleRefresh(refr_time, refr_row);
        } // for r_iter

    /*
    ** determine overlap cycles
    */
    for (long j = (retired_cycle + 1) ; j <= current_cycle ; j++) {
          int used = 0;
          for (int n = 0 ; n < dataTrans.size() ; n++) {
              drd_trans nth_trans = (drd_trans) dataTrans.elementAt(n);

              if ((j >= nth_trans.start_cycle) &&
          (j < nth_trans.end_cycle)) {
          used++;
              }
          } /* for n transactions */

          if (used >= 1) {
              utilized_cycles++;

              if (used > 1) {
          overlap_cycles++;
              }
          }
          trans_cycles += used;
    } /* for j cycles */
    retired_cycle = current_cycle;

    /*
    ** Actually Advance Clock
    */
    current_cycle = new_clock;
    return false;
} // advanceClockTo

boolean cycle_used_row(long test_cycle) {
  for (int j = 0 ; j < rowTrans.size() ; j++) {
    drd_trans jth_trans = (drd_trans) rowTrans.elementAt(j);

    if ((test_cycle >= jth_trans.rowStart) &&
          (test_cycle < (jth_trans.rowStart + OCTCYCLE))) {
          return true;
    }
  }
```

```
  return false;
}

boolean cycle_used_col(long test_cycle) {
  for (int j = 0 ; j < colTrans.size() ; j++) {
    drd_trans jth_trans = (drd_trans) colTrans.elementAt(j);

    if ((test_cycle >= jth_trans.colStart) &&
          (test_cycle < (jth_trans.colStart + OCTCYCLE))) {
          return true;
    }
  }
  return false;
}

boolean cycle_used_data(long test_cycle) {
  for (int j = 0 ; j < dataTrans.size() ; j++) {
    drd_trans jth_trans = (drd_trans) dataTrans.elementAt(j);

    if ((test_cycle >= jth_trans.dataStart) &&
          (test_cycle < (jth_trans.dataEnd))) {
          return true;
    }
  }
  return false;
}

  boolean endSimulation() {
        drd_trans last = lastTrans();
        // Call twice - the second retires to the first
        advanceClockTo(last.end_cycle);
        advanceClockTo(last.end_cycle);
        return true;
  }

  drd_trans lastTrans() {
        drd_trans last = (drd_trans) dataTrans.lastElement();
        drd_trans temp = (drd_trans) colTrans.lastElement();
        if (temp.end_cycle > last.end_cycle) {
            last = temp;
        }
        temp = (drd_trans) rowTrans.lastElement();
        if (temp.end_cycle > last.end_cycle) {
            last = temp;
        }
        return last;
  }

//
// ToString
//
public String toString(int new_mode) {
  display_mode = new_mode;
  return toString();
}

public String toString() {
  String str = new String();
  str += "DRDRAM Schedule ["+super.toString()+"]\n";
  double all_sched = reads_sched + writes_sched + refr_sched;
  str += "\tReads scheduled\t\t\t: "+reads_sched+"\t";
  str += percent8Str(reads_sched / all_sched) + "\n";
  str += "\tWrites scheduled\t\t: "+writes_sched+"\t";
  str += percent8Str(writes_sched / all_sched) + "\n";
  str += "\tRefresh scheduled\t\t: "+refr_sched+"\t";
  str += percent8Str(refr_sched / all_sched) + "\n";
  // Added 07/07/00 BTD
  str += "\tAverage Trans Cycles\t: "+
        (total_latency / total_sched) + "\n";
  //    str += "\tAverage latency\t\t: "+
  //    ((total_latency / ((double)total_sched)) *
  //     the_ctrler.clock_period) + "\n";
  str += "\tAverage latency\t\t: "+
        ((total_latency / ((double)total_sched)) *
         cycleNs) + "\n";
  str += "\tAdjacent Bank Accesses\t: "+adj_bank_accesses+"\t"+
        percent8Str(adj_bank_accesses / ((double)total_sched)) + "\n";

  str += "\tCycles elapsed\t\t\t: "+current_cycle+"\n";
  str += "\tCycles retired\t\t\t: "+retired_cycle+"\n";
  str += "\tUtilized cycles\t\t\t: "+utilized_cycles+"\t";
```

```
    str += percent8Str(utilized_cycles / ((double)retired_cycle)) +"\n";
    str += "\tOverlap cycles\t\t\t: "+overlap_cycles+"\t";
    str += percent8Str(overlap_cycles / ((double)retired_cycle)) + "\t";
    str += percent8Str(overlap_cycles / ((double)utilized_cycles)) + "\n";
    str += "\tTransaction cycles\t\t: "+trans_cycles+"\n";
    str += "\tDegree of (retired) concurrency\t: ";
    str += (trans_cycles/retired_cycle) + "\n";
    str += "\tDegree of (util) concurrency\t: ";
    str += (trans_cycles/utilized_cycles) + "\n";
    str += "\tTime elapsed\t\t\t: "+(cycleNs * current_cycle)+" Seconds\n";
    return str;
  } // toString

    private String percent8Str(double in) {
        if (!(nf instanceof java.text.NumberFormat)) {
            nf = java.text.NumberFormat.getPercentInstance();
            nf.setMinimumFractionDigits(2);
        }
        String ret_str;
        ret_str = SPACES8 + nf.format(in);
        return ret_str.substring(ret_str.length() - 8);
    }

}

===================================================================================================

// package drdram_sim

/*
  File : DRD_trans

  Author : Brian Davis

  */

public class drd_trans {

    //
    // Constants
    //
    static final long OCTCYCLE = drd_ctrl.OCTCYCLE;
    static final int TRANS_READ = drd_ctrl.TRANS_READ;
    static final int TRANS_WRITE = drd_ctrl.TRANS_WRITE;
    static final int TRANS_REFRESH = drd_ctrl.TRANS_REFRESH;


    // Class Variables
    static long trans_created = 0;
    static int display_mode = 0;

    // Instance Variable(s)

    //
    // Transaction Attributes
    int num_bytes;
    long address;
    int trans_type;
    boolean read = false;
    boolean write = false;
    // boolean rowHit = false;
    boolean SAHit = false;
    boolean BankPrecharged = false;

    //
    // Overall Transaction bounds
    long start_cycle;
    long end_cycle;

    //
    // Component Times
    long preStart;
    long rowStart;
    long rowEnd;
    long colStart;
    long colEnd;
    long dataStart, dataEnd;

    //
    // Pointers
    drd_bank access_bank = null;
```

```
  drd_device device_loc = null;
  drd_channel channel_loc = null;

  // Constructor(s)
  drd_trans(int type) {
        this(type, 0L, 0, -1L, -1L);
  }

  drd_trans(long addr, int bytes) {
        this(TRANS_READ, addr, bytes, -1L, -1L);
  }

  drd_trans(int rw, long addr, int bytes) {
        this(rw, addr, bytes, -1L, -1L);
  }

  drd_trans(int rw, long addr, int bytes, long start, long end) {

        trans_created++;
        trans_type = rw;
        if (rw == TRANS_READ)
            read = true;
        else if (rw == TRANS_WRITE)
            write = true;

        address = addr;
        num_bytes = bytes;

        start_cycle = start;
        end_cycle = end;

        //
        // Check to verify withing above [start:end] bounds
        rowStart = colStart = dataStart = dataEnd = -1L;
  }

// Class Methods

//
// Instance Methods
//

  public boolean dataBusReqd() {
        if (((trans_type & TRANS_READ) != 0) ||
            (( trans_type & TRANS_WRITE) != 0)) {
            return true;
        }
        return false;
  }

public boolean rowStart(long cycle) {
  //
  // Check for potential ERROR conditions
  if (rowStart != -1L) {
    System.out.println("ERROR : multiple definition of Row Start for transaction");
    return true;
  }

  if ((start_cycle != -1L) || (start_cycle > cycle)) {
    System.out.println("ERROR : Illegal definition of row given Start for transaction");
    return true;
  }

  rowStart = cycle;

  start_cycle = cycle;

  return false;
}

public boolean colStart(long cycle) {
  //
  // Check for potential ERROR conditions
  if (colStart != -1L) {
    System.out.println("ERROR : multiple definition of Column Start for transaction");
    return true;
  }

  if (SAHit == true)
    start_cycle = cycle;
```

```
    return false;
  }

  private boolean startAt(long new_start) {
    // Check for valid start
    if ((end_cycle != 0) && (new_start > end_cycle)) {
      // ERROR ILLEGAL START
      return true;
    } else {
      start_cycle = new_start;
      return false;
    }
  }

  //
  // To String
  //
  public String toString(int new_mode) {
    display_mode = new_mode;
    return toString();
  }

  public String toString() {
    String str = new String();
    str += "DRDRAM Transaction ["+super.toString()+"]\n";

    if ((display_mode & (1<<0)) != 0) {
      str += "\tTrans type\t: ";

      if (trans_type == TRANS_READ) {
            str += "Read\n";
      } else if (trans_type == TRANS_WRITE) {
            str += "Write\n";
      } else if (trans_type == TRANS_REFRESH) {
            str += "Refresh\n";
      } else {
            str += "UNKNOWN\n";
      }

      str += "\tAddress\t\t: " + java.lang.Long.toHexString(address) + "\n";

      str += "\tNumber Bytes\t: "+num_bytes + "\n";

      str += "\tSAmp cache\t: " + ((SAHit) ? "Hit\n" : "Miss\n");

      str += "\tTrans Span\t: ( "+start_cycle+" : "+end_cycle+" )\n";

      if (SAHit == false)
          str += "\tRow Span\t: ( "+rowStart+" : "+rowEnd+" )\n";

      str += "\tCol Span\t: ( "+colStart+" : "+colEnd+" )\n";

      str += "\tData Span\t: ( "+dataStart+" : "+dataEnd+" )\n";

    }

    return str;
  } // toString

}

===================================================================================================

// package drdram_sim

/*
  File : DRD_CYCLE

  Author : Brian Davis

  */

public class drd_cycle {

  // Class Variables
  long cycles_created = 0;

  // Instance Variable(s)
  int display_mode = 0;
  long cycleno;
  drd_trans rsigs, // Row signals
```

```
            csigs_c, // Column signals
            csigs_mx, // Column signals
            dsigs; // Data signals

  // Constructor(s)
  drd_cycle() {
    cycles_created++;
    rsigs = csigs_c = csigs_mx = dsigs = null;
  }

  // Class Methods

  // Instance Methods
  public String toString() {
    String str = new String();
    str += "DRDRAM cycle\n";
    switch (display_mode) {
    case 999:
      if (rsigs != null)
          str += "Row signals hold "+rsigs;
      else
          str += "Row signals idle";

      if (csigs_c != null)
          str += "Column signals hold "+csigs_c;
     else
          str += "Column signals idle";

      if (csigs_mx != null)
          str += "Column signals hold "+csigs_mx;
     else
          str += "Column signals idle";

      if (dsigs != null)
          str += "Data signals hold "+dsigs;
      else
          str += "Data signals idle";

      break;

    default:
      str += super.toString();
    }
    return str;
  } // toString

}
```

A.4 SimpleScalar JNI Interface

```
/*
** Author : btdavis
**
** Definition of variables used by the JVM
**
*/



#ifndef DRAM_JV_H
#define DRAM_JV_H

JavaVM *jvm;
JNIEnv *env;
JDK1_1InitArgs vm_args;

/*
** drdram variables
*/

// Can be trans or drd_trans :
jclass trans_jcl;

jobject ctrler_obj;
```

```
jmethodID ctrlaccess_jmid;
jmethodID endsim_jmid;
jmethodID ctrlprintyou_jmid;

jfieldID sc_jfid;
jfieldID ec_jfid;
jfieldID ds_jfid;

/*
** sdram varables
*/

#endif

extern int sdram;
extern int drdram;
extern int ddr2_200;
extern int ddr2_200_ems;
extern int ddr2_200_vc;

extern int use_ctrler_remap;
extern char *vc_alloc_policy;

extern jint dram_trans_read;
extern jint dram_trans_write;

==========================================================================================

/*
 * dram_jv.c - Support for Java DRAM routines
 *
 * This file is a part of the SimpleScalar tool suite written by
 * Todd M. Austin as a part of the Multiscalar Research Project.
 *
 * The tool suite is currently maintained by Doug Burger and Todd M. Austin.
 *
 * Copyright (C) 1994, 1995, 1996, 1997 by Todd M. Austin
 *
 * This source file is distributed “as is” in the hope that it will be
 * useful.  The tool set comes with no warranty, and no author or
 * distributor accepts any responsibility for the consequences of its
 * use.
 *
 * Everyone is granted permission to copy, modify and redistribute
 * this tool set under the following conditions:
 *
 *    This source code is distributed for non-commercial use only.
 *    Please contact the maintainer for restrictions applying to
 *    commercial use.
 *
 *    Permission is granted to anyone to make or distribute copies
 *    of this source code, either as received or modified, in any
 *    medium, provided that all copyright notices, permission and
 *    nonwarranty notices are preserved, and that the distributor
 *    grants the recipient permission for further redistribution as
 *    permitted by this document.
 *
 *    Permission is granted to distribute this file in compiled
 *    or executable form under the same conditions that apply for
 *    source code, provided that either:
 *
 *    A. it is accompanied by the corresponding machine-readable
 *       source code,
 *    B. it is accompanied by a written offer, with no time limit,
 *       to give anyone a machine-readable copy of the corresponding
 *       source code in return for reimbursement of the cost of
 *       distribution.  This written offer must permit verbatim
 *       duplication by anyone, or
 *    C. it is distributed by someone who received only the
 *       executable form, and is accompanied by a copy of the
 *       written offer of source code that they received concurrently.
 *
 * In other words, you are welcome to use, share and improve this
 * source file.  You are forbidden to forbid anyone else to use, share
 * and improve what you give them.
 *
 *
 * CREATED : Brian Davis 12/14/99
 *
 */
```

```
#include <string.h>
#include <jni.h>

#include "misc.h"

#include "dram_jv.h"

int use_ctrler_remap = FALSE;
char *vc_alloc_policy = "lru";

/* initialize the Java Virtual Machine */
void jvm_init(char *dram_string, char *dram_policy)
{
  /* variables */
  int res, j, num_devices = 16;
  long long base_addr = 0, device_size;
  jmethodID mid;
  jclass ctrl_jcl = NULL;
  jclass dev_jcl;
  jint cfg_param_jint = 0;

  /* JVM setup */

  JavaVMInitArgs vm_args;
  JavaVMOption options[4];

  /* user classes */
  vm_args.nOptions = 1;
  options[0].optionString =
    "-Djava.class.path=/nfs/shaitan.eecs/l/users/btdavis/java.classes";

  /* disable JIT */
  // options[0].optionString = "-Djava.compiler=NONE";

  /* set native library path */
  // options[2].optionString = "-Djava.library.path=c:\mylibs";

#ifdef DEBUG
  /* print JNI-related messages */
  options[vm_args.nOptions].optionString = "-verbose:jni";
  vm_args.nOptions += 1;
#endif

  vm_args.version = JNI_VERSION_1_2;
  vm_args.options = options;
  vm_args.ignoreUnrecognized = TRUE;

  /* Note that in the Java 2 SDK, there is no longer any need to call
   * JNI_GetDefaultJavaVMInitArgs.
   */
  res = JNI_CreateJavaVM(&jvm, (void **)&env, &vm_args);
  if ((res < 0) || (jvm == NULL) || (env == NULL)) {
    fatal("BTD/JAVA : problems with JNI_CreateJavaVM");
  }

#ifdef DEBUG
    fprintf(stderr, "DEBUG: Return from JNI_CreateJavaVM Invokation\n");
#endif

    if ((!mystricmp(dram_string, "ddr2")) ||
        (!mystricmp(dram_string, "ddr2vc")) ||
          (!mystricmp(dram_string, "ddr2ems"))) {
      /*
      ** Variables
      */
      char *field_name = NULL;

#ifdef DEBUG
      fprintf(stderr,
                "DEBUG: Creating structures nessesary for DDR2 memory system\n");
#endif

      /*
      ** Intialize for DDR2
      */
      ctrl_jcl = (*env)->FindClass(env, "ddr2_ctrl");
      if (ctrl_jcl == NULL) {
          fatal("BTD/JAVA : ddr2_ctrl class could not be found");
      }

      if (!mystricmp(dram_string, "ddr2")) {
```

```
        if (!mystricmp(dram_policy, "cpa")) {
          field_name = "DDR2_CPA";
        } else if (!mystricmp(dram_policy, "op")) {
          field_name = "DDR2_OP";
        } else {
          fatal("DRAM_JV : Unknown dram_policy for ddr2 system");
        }
    } else if (!mystricmp(dram_string, "ddr2ems")) {
        if (!mystricmp(dram_policy, "cpa")) {
          field_name = "DDR2EMS_CPA_AWX";
        } else if (!mystricmp(dram_policy, "nwx")) {
          field_name = "DDR2EMS_CPA_NWX";
        } else {
          printf("policy = %s\n", dram_policy);
          fatal("DRAM_JV : Unknown dram_policy for ddr2ems system");
        }
    } else if (!mystricmp(dram_string, "ddr2vc")) {
        if (!mystricmp(dram_policy, "cpa")) {
          field_name = "DDR2VC_CPA";
        } else if (!mystricmp(dram_policy, "op")) {
          field_name = "DDR2VC_OP";
        } else {
          fatal("DRAM_JV : Unknown dram_policy for ddr2vc system");
        }
    } else {
        fatal("DRAM_JV : Unknown dram_string for ddr2 system");
    }

    // local variable region
    {
        jfieldID fid = NULL;
        fid = (*env)->GetStaticFieldID(env, ctrl_jcl, field_name, "I");
        if (fid == NULL) {
          printf("field_name = %s\n", field_name);
          fatal("DRAM_JV : ddr2_ctrl(field_name) could not be found");
        }
        cfg_param_jint = (*env)->GetStaticIntField(env, ctrl_jcl, fid);
        if (cfg_param_jint == 0) {
          fatal("DRAM_JV : ddr2_ctrl(field_name) yields ZERO cfg value");
        }
    }

    /*
    ** Create & Initialize DDR2 memory system
    */

    mid = (*env)->GetMethodID(env, ctrl_jcl, "<init>", "(I)V");

    if (mid != NULL) {
        ctrler_obj = (*env)->NewObject(env, ctrl_jcl, mid, cfg_param_jint);
    } else {
        fatal("DRAM_JV : ddr2_ctrl Constructor not found");
    }

    if (ctrler_obj == NULL)
        fatal("DRAM_JV : ddr2_ctrl object is NULL");

#ifdef DEBUG
    fprintf(stderr,
             "DEBUG: ddr2_ctrl object created & valid\n");
#endif

    /*
    ** If dram type is VC, it may be req'd to change the
    ** Channel Allocation policy
    */
    if (!mystricmp(dram_string, "ddr2vc")) {
        // May 10 2000
        // add code for changeVCAllocPolicy(vc_alloc_policy);
        jfieldID fid = NULL;
        jobject alloc_str = NULL;

        mid = (*env)->GetMethodID(env, ctrl_jcl, "changeVCAllocPolicy",
          "(Ljava/lang/String;)Z");
        if (mid == NULL) {
        fatal("DRAM_JV : ctrler.changeVCAllocPolicy() not found");
        }

        if (!mystricmp(vc_alloc_policy, "lru")) {
          field_name = "POLICY_LRU";
        } else if (!mystricmp(vc_alloc_policy, "rand")) {
```

```
            field_name = "POLICY_RANDOM";
          } else if (!mystricmp(vc_alloc_policy, "assoc")) {
            field_name = "POLICY_ASSOC";
          } else if (!mystricmp(vc_alloc_policy, "busmaster")) {
            field_name = "POLICY_P12IO4";
          } else {
            fatal("DRAM_JV : vc_alloc_policy unknonwn");
          }

          fid = (*env)->GetStaticFieldID(env, ctrl_jcl, field_name,
           "Ljava/lang/String;");
          if (fid == NULL) {
            fprintf(stderr, "Field Name = %s\n", field_name);
            fatal("DRAM_JV : vc_alloc_policy FIELD could not be found");
          }

          alloc_str = (*env)->GetStaticObjectField(env, ctrl_jcl, fid);
          if (alloc_str == NULL) {
            fatal("DRAM_JV : alloc_str PTR could not be found");
          }

          if ((*env)->CallBooleanMethod(env, ctrler_obj, mid, alloc_str) != 0) {
            fatal("DRAM_JV : invocation of changeVCAllocPolicy() returns error");
          }

      } // if ddr2vc

      /*
      ** Controller has been created & is valid, create devices
      */

      mid = (*env)->GetMethodID(env, ctrl_jcl, "addDevice",
          "(I)Z");

      if (mid == NULL) {
          fatal("DRAM_JV : ddr2_ctrl.addDevice() not found");
      }

      {
          jfieldID fid = NULL;
          jint dimm_size;
          fid = (*env)->GetStaticFieldID(env, ctrl_jcl, "DIMM_256MB",
                 "I");
          if (fid == NULL) {
            fatal("JNI_CALLS : ddr2_ctrl.addDevice() init FIELD not found");
          }
          dimm_size = (*env)->GetStaticIntField(env, ctrl_jcl,
                fid);

          if ((*env)->CallBooleanMethod(env, ctrler_obj, mid, dimm_size) == 0) {
            fatal("JNI_CALLS : ddr2_ctrl.addDevice() returned error");
          }
      }
#ifdef DEBUG
      fprintf(stderr, "DEBUG : ddr2 ctrler created, Devices ADDed\n");
#endif

      /*
      ** Setup access methods
      */

      trans_jcl = (*env)->FindClass(env, "ddr2_trans");

      ctrlaccess_jmid = (*env)->GetMethodID(env, ctrl_jcl, "access",
              "(JIJI)Ldram_trans;");

      endsim_jmid = (*env)->GetMethodID(env, ctrl_jcl,
          "endSimulation",
          "()Z");

      ctrlprintyou_jmid = (*env)->GetMethodID(env, ctrl_jcl,
                "printYourself",
                "()V");

      // local variable region
      {
          jfieldID fid;
          fid = (*env)->GetStaticFieldID(env, ctrl_jcl, "TRANS_READ", "I");
          if (fid == NULL) {
            fatal("DRAM_JV : ddr2_ctrl(field_name) could not be found");
          }
```

```
            dram_trans_read = (*env)->GetStaticIntField(env, ctrl_jcl, fid);
        }
        // local variable region
        {
            jfieldID fid;
            fid = (*env)->GetStaticFieldID(env, ctrl_jcl, "TRANS_WRITE", "I");
            if (fid == NULL) {
              fatal("DRAM_JV : ddr2_ctrl(field_name) could not be found");
            }
            dram_trans_write = (*env)->GetStaticIntField(env, ctrl_jcl, fid);
        }

        if ((trans_jcl == NULL) ||
              (ctrlaccess_jmid == NULL) ||
              (endsim_jmid == NULL) ||
              (ctrlprintyou_jmid == NULL)) {
            fprintf(stderr, "trans_jcl = 0x%08X\n", ((unsigned int)trans_jcl));
            fprintf(stderr, "ctrlaccess_jmid = 0x%08X\n",
            ((unsigned int)ctrlaccess_jmid));
            fprintf(stderr, "endsim_jmid = 0x%08X\n", ((unsigned int)endsim_jmid));
            fprintf(stderr, "ctrlprintyou_jmid = 0x%08X\n",
            ((unsigned int)ctrlprintyou_jmid));
            fatal("DRAM_JV(DDR2) : required method pointers not initialized");
        }

        /*
        ** End DDR2 init
        */

#ifdef DEBUG
    fprintf(stderr, "DEBUG: Exitting DDR2 initialization routines\n");
#endif

    } else if (!mystricmp(dram_string, "drdram")) {

    /*
    ** Variables
    */
    char *field_name = NULL;
    jmethodID add_mid;

    /*
    ** Initialize for DRDRAM
    */

    ctrl_jcl = (*env)->FindClass(env, "drd_ctrl");
    if (ctrl_jcl == NULL)
      fatal("BTD/JAVA : drdctrl class could not be found");

    // Select CFG int for controler policy

    if (!mystricmp(dram_policy, "cpa"))
      field_name = "CFG_CLOSEPAGEAUTO";
    else
      field_name = "CFG_OPENPAGE";

    {
      jfieldID fid;
      fid = (*env)->GetStaticFieldID(env, ctrl_jcl, field_name, "I");
      cfg_param_jint = (*env)->GetStaticIntField(env, ctrl_jcl, fid);
      if ((fid == NULL) ||
            (cfg_param_jint == 0)) {
          fatal("BTD/JAVA : drdctrl(field_name) could not be found");
      }
    }

    /*
    ** Create & Initialize DRDRAM memory system
    */

    mid = (*env)->GetMethodID(env, ctrl_jcl, "<init>", "(I)V");

    if (mid != NULL) {
      ctrler_obj = (*env)->NewObject(env, ctrl_jcl, mid, cfg_param_jint);
    } else
      fatal("BTD/JAVA : drd_ctrl Constructor not found");

    if (ctrler_obj == NULL)
      fatal("BTD/JAVA : drd_ctrl object is NULL");

    /*
```

```
** Controller has been created & is valid, create DRDRAM devices
*/

add_mid = (*env)->GetMethodID(env, ctrl_jcl,
        "addDevice",
        "(Ldrd_device;)Z");

dev_jcl = (*env)->FindClass(env, "drd_device");

mid = (*env)->GetMethodID(env, dev_jcl, "<init>", "(JJLdrd_ctrl;)V");

if ((add_mid == NULL) ||
      (dev_jcl == NULL) ||
      (mid == NULL)) {
  fprintf(stderr, "add_mid = 0x%08X\n", ((unsigned int)add_mid));
  fprintf(stderr, "dev Class = 0x%08X\n", ((unsigned int)dev_jcl));
  fprintf(stderr, "dev Constructor = 0x%08X\n", ((unsigned int)mid));
  fatal("BTD/JAVA : Add (DRD) Device methods not found");
}

// Local variable region
{
  jfieldID fid;

  fid = (*env)->GetStaticFieldID(env, dev_jcl,
           "RIMM_64MB_8by64Mb", "J");

  if (fid == NULL)
      fatal("BTD/JAVA : Field ID for device size not found");

  device_size = (*env)->GetStaticLongField(env, dev_jcl, fid);

  /* fprintf(stdout, "DEBUG : Device size = %08lx\n", device_size); */
}

for (j = 0 ; j < num_devices ; j++) {
  jobject drddev_obj;

  /*
  ** Create DRDRAM device & Add to ctrler/channel
  */

  /*    fprintf(stdout,
          "DEBUG : before call : drd_device(%08llx, %i, %08llx)\n",
          device_size, 16, base_addr); */

  drddev_obj = (*env)->NewObject(env, dev_jcl, mid,
           device_size, base_addr,
           ctrler_obj);

  (*env)->CallBooleanMethod(env, ctrler_obj, add_mid, drddev_obj);

  base_addr += device_size;

  /*
      ctrler.addDevice(new drd_device(drd_device.DEVICE_SIZE_64Mb,
      16,
      base_addr));
      base_addr += drd_device.DEVICE_SIZE_64Mb;
  */
} // For

/*
** Prepare class pointers for later use
*/

trans_jcl = (*env)->FindClass(env, "drd_trans");

if (trans_jcl == NULL) {
  fatal("BTD/JAVA : required class pointers not initialized");
}

/*
** Prepare method pointers for later use
*/

ctrlaccess_jmid = (*env)->GetMethodID(env, ctrl_jcl, "access",
        "(JIJI)Ldrd_trans;");

endsim_jmid = (*env)->GetMethodID(env, ctrl_jcl,
            "endSimulation",
```

```
                "()Z");

    ctrlprintyou_jmid = (*env)->GetMethodID(env, ctrl_jcl,
              "printYourself",
              "()V");

    if ((ctrlaccess_jmid == NULL) ||
         (endsim_jmid == NULL) ||
         (ctrlprintyou_jmid == NULL)) {
      fatal("BTD/JAVA : required method pointers not initialized");
    }

    /*
    ** Prepare field pointers for later use
    */
    /* Commented out out 4/10/2000

    sc_jfid = (*env)->GetFieldID(env, trans_jcl, "start_cycle", "J");

    ec_jfid = (*env)->GetFieldID(env, trans_jcl, "end_cycle", "J");

    ds_jfid = (*env)->GetFieldID(env, trans_jcl, "dataStart", "J");

    if ((sc_jfid == NULL) ||
         (ec_jfid == NULL) ||
         (ds_jfid == NULL)) {
      fatal("BTD/JAVA : required field pointers not initialized");
    }

         */

  } else if ((!mystricmp(dram_string, "pc100_sdram")) ||
              (!mystricmp(dram_string, "ddr133")) ||
              (!mystricmp(dram_string, "ddr133_cas2"))) {
    /*
    ** Variables
    */
    jint init_param_jint = 0;
    char *field_name = NULL;

    /*
    ** Statements
    */
#ifdef DEBUG
    fprintf(stderr, "DEBUG : Into SDRAM Init\n");
#endif

    /*
    ** find sdram_ctrl class
    */
    ctrl_jcl = (*env)->FindClass(env, "sdram_ctrl");
    if (ctrl_jcl == NULL)
      fatal("BTD/JAVA : sdram_ctrl class could not be found");

    /*
    ** Get Config String (parameters) for sdram_ctrl constructor
    */
    if (!mystricmp(dram_string, "ddr133")) {
      if (!mystricmp(dram_policy, "cpa"))
         field_name = "DDR133_CAS3_CPA";
      else
         field_name = "DDR133_CAS3_OP";
    } else if (!mystricmp(dram_string, "ddr133_cas2")) {
      if (!mystricmp(dram_policy, "cpa"))
         field_name = "DDR133_CPA";
      else
         field_name = "DDR133_OP";
    } else {
      if (!mystricmp(dram_policy, "cpa"))
         field_name = "PC100_CPA";
      else
         field_name = "PC100_OP";
    }

    // Local variables section
    {
      jfieldID fid;
      fid = (*env)->GetStaticFieldID(env, ctrl_jcl, field_name, "I");
      init_param_jint = (*env)->GetStaticIntField(env, ctrl_jcl, fid);
      if ((fid == NULL) ||
           (init_param_jint == 0)) {
```

```
            fatal("BTD/JAVA : sdramctrl(field_name) could not be found");
        }
    }

    /*
    ** Get sdram_ctrl() constructor
    */
#ifdef DEBUG
    fprintf(stderr, "DEBUG : retriving sdram_ctrl constructor\n");
#endif
    mid = (*env)->GetMethodID(env, ctrl_jcl, "<init>", "(I)V");
    ctrler_obj = (*env)->NewObject(env, ctrl_jcl, mid, init_param_jint);
    if ((mid == NULL) ||
        (ctrler_obj == NULL)) {
      fatal("BTD/JAVA : sdram_ctrl Constructor not found or returned NULL");
    }

    /*
    ** Invoke addDevice()
    **
    ** retrieve Method ID first
    */
    mid = (*env)->GetMethodID(env, ctrl_jcl, "addDevice",
                "(I)Z");
    if (mid == NULL) {
      fatal("JNI_CALLS : sdram_ctrl.addDevice() method not found");
    }

    {
      jfieldID fid = NULL;
      jint dimm_size;
      if (!mystricmp(dram_string, "pc100_sdram")) {
          fid = (*env)->GetStaticFieldID(env, ctrl_jcl, "DIMM_64MB",
                  "I");
      } else if ((!mystricmp(dram_string, "ddr133")) ||
           (!mystricmp(dram_string, "ddr133_cas2"))) {
          fid = (*env)->GetStaticFieldID(env, ctrl_jcl, "DIMM_256MB",
                  "I");
      } else {
          fatal("Logical Impossibility #218");
      }
      if (fid == NULL) {
          fatal("JNI_CALLS : sdram_ctrl.addDevice() init String FIELD not found");
      }

      dimm_size = (*env)->GetStaticIntField(env, ctrl_jcl,
               fid);

      if ((*env)->CallBooleanMethod(env, ctrler_obj, mid, dimm_size) == 0) {
          fatal("JNI_CALLS : sdram_ctrl.addDevice() returned error");
      }
    }
#ifdef DEBUG
    fprintf(stderr, "DEBUG : ctrler created, Devices ADDed\n");
#endif

    /*
    ** Setup access methods
    */

    trans_jcl = (*env)->FindClass(env, "trans");

    ctrlaccess_jmid = (*env)->GetMethodID(env, ctrl_jcl, "access",
           "(JIJI)Ltrans;");

    endsim_jmid = (*env)->GetMethodID(env, ctrl_jcl,
            "endSimulation",
           "()Z");

    ctrlprintyou_jmid = (*env)->GetMethodID(env, ctrl_jcl,
              "printYourself",
              "()V");

    if ((trans_jcl == NULL) ||
          (ctrlaccess_jmid == NULL) ||
          (endsim_jmid == NULL) ||
          (ctrlprintyou_jmid == NULL)) {
      fatal("BTD/JAVA(SDRAM) : required method pointers not initialized");

    }
```

```
  } else {
    fatal("Unknown dram_type in jvm_init()\n");
  }

    /*
    ** Prepare field pointers for later use
    */

    sc_jfid = (*env)->GetFieldID(env, trans_jcl, "start_cycle", "J");

    ec_jfid = (*env)->GetFieldID(env, trans_jcl, "end_cycle", "J");

    ds_jfid = (*env)->GetFieldID(env, trans_jcl, "dataStart", "J");

    if ((sc_jfid == NULL) ||
          (ec_jfid == NULL) ||
          (ds_jfid == NULL)) {
      fatal("BTD/JAVA(ALL) : required field pointers not initialized");
    }

    /*
    ** Turn on Controller re-mapping if indicated
          if (use_remap) {
              new_ctrler.enableRemap();
          }
    */

    if (use_ctrler_remap) {
      mid = (*env)->GetMethodID(env, ctrl_jcl, "enableRemap",
          "()V");

      if (mid == NULL) {
          fatal("DRAM_JV : ctrler.enableRemap() not found");
      }

      (*env)->CallVoidMethod(env, ctrler_obj, mid);

    }



} /* jvm_init() */
```

Bibliography

[Anderson67] Anderson, Sparacio, and Tomasulo. "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling." IBM Systems Journal, January 1967.

[Beckerman99] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz and Uri Weiser. “Correlated Load-Address Predictors.” *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 54 - 63, 1999.

[Bondurant93] Bondurant, D.W., “High-Performance DRAMs”, Electronic Products, June 1993, p.47-51.

[Burger96] D. Burger, J. R. Goodman, and A. Kagi. 1996. “Memory bandwidth limitations of future microprocessors.” In *Proc. 23rd Annual International Symposium on Computer Architecture* (ISCA’96), pages 78–89, Philadelphia PA.

[Burger97] D. Burger and T. M. Austin. 1997. “The SimpleScalar tool set, version 2.0.” Technical Report CS-1342, University of Wisconsin-Madison.

[Bursky93a] Bursky, D., “Synchronous DRAMs Clock at 100 MHz”, Electronic Design, February 18, 1993, p.45-49.

[Bursky93b] Bursky, D., “Fast DRAMs Can Be Swapped For SRAM Caches”, Electronic Design, July 22, 1993, p.55-67.

[Bursky95] Bursky, Dave, “Advanced DRAM’s Deliver Peak Systems Performance”, Electronic Design, August 7, 1995, p.42-51.

[BYTE94] “Mosys Offers Better Memory for Video”, BYTE, October 1994.

[C-3D00] C-3D. 2000. “C3D Data Storage Technology.” Constellation 3D, http://www.c-3d.net/tech.htm

[Calder99] Brad Calder, Glenn Reinman and Dean M. Tullsen. “Selective Value Prediction.” *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 64 - 74, 1999.

[Callahan91] David Callahan, Ken Kennedy and Allan Porterfield. 1991. “Software Prefetching.” In *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), Pages 40 - 52.

[Cataldo00] A. Cataldo, “Intel, Rambus mull overhaul for RDRAM.” EE Times, Aug 25, 2000. http://www.eet.com/story/OEG20000825S0065.

[Carter99] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and et al. 1999. “Impulse: Building a smarter memory controller.” In *Proc. Fifth International Symposium on High Performance Computer Architecture* (HPCA’99), pages 70–79, Orlando FL.

[Cheng98] Ben-Chung Cheng, Daniel A. Connors and Wen-mei W. Hwu. "Compiler-Directed Early Load-Address Generation." *Proceedings of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, pp. 138 - 147, 1998.

[Comerford92] Comerford, Richard and Watson, George, "Memory Catches Up", IEEE Spectrum, October 1992, p.34-35.

[Coteus00] Paul Coteus. International Business Machines. Personal Communication, May 2000.

[Cuppu99] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures." In *Proc. 26th Annual International Symposium on Computer Architecture* (ISCA'99), pages 222–233, Atlanta GA.

[Davis00a] B. Davis, T. Mudge, B. Jacob, V. Cuppu, "DDR2 and and low-latency variants." In *Proc. Solving the Memory Wall Workshop*, held in conjunction with the 27th International Symposium on Computer Architecture (ISCA-2000). Vancouver BC, Canada, June 2000.

[Davis00b] B.Davis, T.Mudge, B.Jacob, "The New DRAM Interfaces: SDRAM, RDRAM and Variants." In *Proc. Third International Symposium on High Performance Computing* . Yoyogi, Tokyo, JAPAN, October 2000.

[DeHon95] Andre DeHon. "Notes on Integrating Reconfigurable Logic with DRAM Arrays.", MIT Transit Project, Transit Note #120, April 8, 1995.

[Dilpert00] B. Dilpert. 2000. "The Slammin' Jammin' DRAM Scramble." Electronic Design News, Jan 20, 2000, pages 68-80. http://www.ednmag.com/ednmag/reg/2000/01202000/pdfs/02cs.pdf

[Dundas98] James David Dundas. "Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream." Ph..D. Thesis, The University of Michigan, 1998.

[Dundas97] James Dundas and Trevor Mudge. "Improving data cache performance by pre-executing instructions under a cache miss." *Proc. 1997 ACM Int. Conf. on Supercomputing*, July 1997, pp. 68-75.

[EMS00] EMS. 2000. "64Mbit - Enhanced SDRAM". Enhanced Memory Systems, http://www.edram.com/Library/datasheets/SM2603,2604pb_r1.8.pdf.

[Farkas97] Keith I. Farkas, Paul Chow, Norman P. Jouppi and Zvonko Vranesic. "Memory-System Design Considerations for Dynamically-Scheduled Processors." *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 133 - 143, 1997.

[Flanagan97] David Flanagan. "Java in a Nutshell: A Desktop Quick Reference." Second Edition, O'Reilly & Associates Inc. 1997.

[Foss92] Richard Foss and Betty Prince. “Fast interfaces for DRAMs.” IEEE Spectrum, pp.54-57, October 1992.

[Fu91] John W. C. Fu and Janak H. Patel. “Data Prefetching in Multiprocessor Vector Cache Memories” *Proceedings of the 18th annual International Symposium on Computer Architecture*, pp. 54 - 63, 1991.

[Fujitsu99] “High-End Memory for High-End Graphics”, 64M (x32) DDR-SDRAM with Fast-Cycle RAM Core Technology - MB81P643287, Fujitsu Semiconductor, http://www.fujitsumicro.com/memory/fcram.htm.

[Fujitsu00] Fujitsu. 2000. “8 x 256K x 32 BIT Double Data Rate (DDR) FCRAM.” Fujitsu Semiconductor, http://www.fujitsumicro.com/memory/fcram.htm.

[Gill00] Jim Gill. “In the trenches with high-performance memory: a designer’s perspective.” EDN Access, June 22, 2000. Web Exclusive. http://www.ednmag.com/ednmag/reg/2000/06222000/webex4.pdf

[Glaskowsky99] Peter N. Glaskowsky. “Mosys Explains 1T-SRAM Technology.” Microprocessor Report, Volume 13, Issue 12, September 13, 1999.

[Gwennap96] Linley Gwennap. “Digital 21264 Sets New Standard.” Microprocessor Report, Volume 10, Issue 14, Oct. 28, 1996.

[Gwennap98] Linley Gwennap. “Alpha 21364 to Ease Memory Bottleneck.” Microprocessor Report, pp. 12-15, Oct. 26, 1998.

[Hennesy96] J.L. Hennesy and D.A. Patterson. “Computer Architecture: A Quantitative Approach.” Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[Horiguchi95] Masashi Horiguchi, et al., “An Experimental 220MHz 1Gb DRAM”, 1995 IEEE International Solid-State Circuits Conference, p.252-253.

[Hwu87] W. M. Hwu and Y. N. Patt. “Checkpoint Repair for Out-of-Order Execution Machines.” *The 14th Annual International Symposium on Computer Architecture*, pp. 18 - 26, 1987.

[IBM98] “64 Mb Synchronous DRAM”. International Business Machines, 1998. http://www.chips.ibm.com/products/memory/03K2149/03K2149.pdf

[IBM99] “128Mb Direct RDRAM”. International Business Machines, 1999. http://www.chips.ibm.com/products/memory/19L3262/19L3262.pdf

[IBM00a] “256Mb Double Data Rate Synchronous DRAM.” International Business Machines, 2000. http://www.chips.ibm.com/products/memory/29L0011/29L0011.pdf

[IBM00b] “ASIC SA-27E Databook.“ International Business Machines, Sept. 20, 2000. http://www.chips.ibm.com/techlib/products/asics/databook.html

[IEEE1596.4] IEEE Draft Standard for High-Bandwidth Memory Interface Based on SCI Signaling Technology (Ramlink), Draft 1.25 IEEE P1596.4-199X, The Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1995.

[IEEE1596.x] Draft Standard for A High-Speed Memory Interface (SyncLink), Draft 0.5 IEEE P1596.x-199X, The Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1995.

[JEDEC00] Joint Electronic Device Engineering Council. "JESD21C - Configurations for Solid State Memories." 2000. http://www.jedec.org/download/pub21/default.cfm.

[Jones92] Fred Jones, et al., "A new era of fast dynamic RAMs." IEEE Spectrum, October 1992, p.43-49.

[Kang94] H.K. Kang, et al., "Highly Manufacturable Process Technology for Reliable 256 Mbit and 1 Gbit DRAMs." IEDM, pp.635-638, 1994.

[Klaiber91] Alexander C. Klaiber and Henry M. Levy. "An Architecture for Software-Controlled Data Prefetching." *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 43 - 53, 1991.

[Kontoth94] L. I. Kontothanassis, et. al. 1994. "Cache performance in vector supercomputers." *Proceedings of the Conference on Supercomputing '94*, pages 255-264, November 14 - 18, 1994, Washington United States.

[Kroft81] David Kroft. "Lockup-Free Instruction Fetch / Prefetch Cache Organization." *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 81-88, 1981.

[Lammers00] D. Lammers, 2000. "Intel sticks to its guns on Rambus." EE Times, Feb. 18, 2000. http://www.eetimes.com/conferences/idf/story/OEG20000217S0025

[Lee97] Chunho Lee, M. Potkonjak, W.H. Mangione-Smith. 1997. "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems." In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (Micro '97).

[Levy95] Markus Levy, "The dynamics of DRAM technology multiply, complicate design options", EDN, January 5, 1995, p.46-56.

[Levy96] Markus Levy, "Souped-up Memories", Electronic Design, January 4, 1996, p.38-52.

[Lin99] W. Lin. 1999. "Prefetching at the Interface between Level-two Cache and Direct Rambus." Research Report, May 4, 1999. University of Michigan.

[Lipasti96] Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen. "Value Locality and Load Value Prediction." *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138 - 147, 1996.

[Mahapatra99] Nihar R. Mahapatra and Balakrishna Venkatrao, "The Processor-Memory bottleneck: Problems and Solutions.", ACM Crossroads, http://www.acm.org/crossroads/xrds5-3/pmgap.html.

[McCalpin00] J. McCalpin. 2000. "STREAM: Sustainable Memory Bandwidth in High Performance Computers.", http://www.cs.virginia.edu/stream/

[McComas00] B. McComas. 2000. Radio Wall Street Interview at Microsoft WinHEC 2000 conference. April 27, 2000. http://media.vcall.com/archive/ram/mswinhec2000/mccomas.rpm.

[McComas99] B. McComas. 1999. "DDR vs. Rambus: A Hands-on Performance Comparison", November 1999, InQuest Market Research, http://www.inqst.com/ddrvrmbs.htm.

[McKee95] S. A. McKee and W. A. Wulf. 1995. "Access ordering and memory-conscious cache utilization." In Proc. International Symposium on High Performance Computer Architecture (HPCA'95), pages 253–262, Raleigh NC.

[McKee96] S. McKee, A. Aluwihare, B. Clark, R. Klenke, T. Landon, C. Oliver, M. Salinas, A. Szymkowiak, K. Wright, W. Wulf, and J. Aylor. 1996. "Design and evaluation of dynamic access ordering hardware." In Proc. International Conference on Supercomputing, Philadelphia PA.

[Mitsubishi95a] DRAM Memories, Databook, Misubishi Semiconductors, 1995.

[Misubishi95b] 4MCDRAM: 4M (256K-word by 16-bit) Cached DRAM with 16K (1024-word by 16-bit) SRAM, M5M4V4169TP Target Specification (Rev. 4.0), March 1995, Mitsubishi LSIs, Mitsubishi Electric.

[Mosys00] "Embedded 1T-SRAM Technology : The Embedded Memory Technology For System-on-Chip." Mosys Incorporated, 2000. http://www.mosysinc.com/tech1ts/

[Mosys94] Multibanked DRAM Technology White Paper, Mosys Incorporated, July 1994.

[Mowry92] Todd C. Mowry, Monica S. Lam and Anoop Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching." *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[NEC99] "128M-BIT Virtual Channel SDRAM". NEC Electronics Inc, 1999. http://www.necel.com/home.nsf/ViewAttachments/M14412EJ3V0DS00/$file/M14412EJ3V0DS00.pdf.

[Ng92] Ng, Ray, “Fast computer memories”, IEEE Spectrum, October 1992, p.36-39.

[Noonburg94] Noonburg, D.B. and Shen, J.P., “Theoretical Modeling of Superscalar Processor Performance”, MICRO 27, November 1994, p.52-62.

[Patterson98] David A. Patterson and John L. Hennessy. “Computer Organization & Design : The Hardware Software Interface”, Second Edition, Morgan Kaufmann Publishers, Inc., 1998.

[Peters00] Mike Peters. Enhanced Memory Systems. Personal Communication, September 2000.

[Przbylski96] S. Przybylski. 1996. New DRAM Technologies: A Comprehensive Analysis of the New Architectures. MicroDesign Resources, Sebastopol CA.

[Rambus92] Rambus Technology Guide, Revision 0.90-Preliminary, May 1992, Rambus Inc., 2465 Latham St., Mountain View CA.

[Rambus93] Rambus Architectural Overview, Copyright 1992, 1993, Rambus Inc., 2465 Latham St., Mountain View CA.

[Rambus99] Rambus. 1999. “Direct RMC.d1 Data Sheet” Rambus, http://www.rambus.com/developer/downloads/RMC.d1.0036.00.8.pdf.

[Ramtron 94] DM2223/2233 Sync Bursting EDRAM, 512Kb X 8 Enhanced Dynamic RAM, Preliminary Datasheet, 1994, Ramtron.

[Rau79] B.R. Rau. “Program Behavior and the Performance of Interleaved Memories”, IEEE Transactions on Computers, Vol. C-28, No. 3, pp.191-199, March 1979.

[Reinman98] Glenn Reinman and Brad Calder. “Predictive Techniques for Aggressive Load Speculation.” *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 127 - 137, 1998.

[Robertson00] J. Robertson. 2000. “Advanced DRAM alliance works on specs for release later this year”, Semiconductor Business News, Mar. 07, 2000. http://www.semibiznews.com/story/OEG20000307S0006.

[Rogers92] Anne Rogers and Kai Li. “Software Support for Speculative Loads; *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 38 - 50, 1992.

[Roth98] Amir Roth, Andreas Moshovos and Gurindar S. Sohi. “Dependence Based Prefetching for Linked Data Structures.” *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 115 - 126, 1998.

[Roth99] Amir Roth and Gurindar S. Sohi. “Effective Jump-Pointer Prefetching for Linked Data Structures.” *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 111 - 121, 1999.

[Rudolph95] L. Rudolph and D. Criton. “Creating a Wider Bus using Caching Techniques”, Proc. 1st Int’l Symp. High Performance Computer Architecture. IEEE Comp Society Press, pp. 90-99. 1995.

[Sazeides97] Yiannakis Sazeides and James E. Smith. “The Predictability of Data Values.” *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248 - 258, 1997.

[SemiBN00] “Motorola believes MRAM could replace nearly all memory technologies.” Semiconductor Business News, May 10, 2000. http://www.semibiznews.com/story/OEG20000510S0031

[Shibahara94] K. Shibahara, et al., “1 GDRAM Cell with Diagonal Bit-Line (DBL) Configuration and Edge Operation MOS (EOS) FET”, IEDM, 1994, p.639-642.

[Sites96] Richard Sites. "It’s the Memory, Stupid!" Microprocessor Report, p. 18, Volume 10, Issue 10, Sept 5, 1996.

[Sohi87] G. S. Sohi and S. Vajapeyam. “Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors.” *The 14th Annual International Symposium on Computer Architecture*, pp. 27 - 34, 1987.

[Sohi91] Gurindar S. Sohi and Manoj Franklin. “High-Bandwidth Data Memory Systems for Superscalar Processors.” *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53 - 62, 1991.

[SPEC00] SPEC. “ SPEC CPU2000 Benchmarks.” Standard Perfromance Evaluation Corporation, 2000.http://www.spec.org/osg/cpu2000/

[SPEC95] SPEC. “SPEC CPU95 Benchmarks.” Standard Perfromance Evaluation Corporation, 1995. http://www.spec.org/osg/cpu95/.

[SPEC92] SPECInt92, Release V1.1, Dec. 1992.

[Swanson98] Mark Swanson, Leigh Stoller and John Carter. “Increasing TLB Reach Using Superpages Backed by Shadow Memory.“ *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 204 - 213, 1998.

[Sugibayashi95] T. Sugibayashi, et al., “A 1Gb DRAM for File Applications”, 1995 IEEE International Solid-State Circuits Conference, p.254-255.

[Takai94] Y. Takai, et al., “250 MByte/s Synchronous DRAM using a 3-stage-Pipelined Architecture”, IEEE Journal of Solid-State Circuits, Vol. 29, No. 4, April 1994, p.426-430.

[Tang99] Hong Tang, Kai Shen and Tao Yang. “Compile/Run-Time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines.” *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 107 - 118, 1999.

[Tarui94] Y. Tarui, “Future DRAM Development and Prospects for Ferroelectric Memories”, IEDM, 1994, p.7-16.

[Tomasulo67] R.M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, Vol 11, pp. 25-33, Jan. 1967.

[Wang97] Kai Wang and Manoj Franklin. “Highly Accurate Data Value Prediction Using Hybrid Predictors.” *Proceedings of the Thirtieth Annual IEEE/ ACM International Symposium on Microarchitecture*, pp. 281 - 290, 1997.

[Weiss95] R. Weiss, “64 Gbit DRAMs, 1 GHz microprocessors expected by 2010”, Computer Design, May 1995, p.50-51.

[Wilkes95] M.V. Wilkes, “The Memory Wall and the CMOS End-Point”, ACM Computer Architecture News, Vol. 23, No. 4, Sept. 1995, p. 4-6.

[Wilson95] R.Wilson, “Rambus partners to build 64-Mbit RDRAMs.” Electrical Engineering Times, Sept. 18, 1995, p.16.

[Woo00] S.Woo, “Architecture and Performance of the Direct RDRAM.“ Rambus Inc. A presentation for *Stanford University Computer Systems Laboratory EE380 Colloquium*, Feb 2, 2000. http://www.stanford.edu/class/ee380/

[Wulf95] W.A. Wulf and S.A. McKee, “Hitting the Memory Wall: Implications of the Obvious.” ACM Computer Architecture News, Vol. 23, No. 1, March 1995, p.20-24.

[Xanalys00] “The Memory Management Reference.” Xanalys Software Tools, http://www.xanalys.com/software_tools/mm/

[Yabu99] Akira Yabu. “Low Latency DRAM Working Group Activity Report.” JEDEC 42.3 FDTG. December 3, 1999.