

ABSTRACT

Title of Document: HIGH-PERFORMANCE DRAM SYSTEM DESIGN CONSTRAINTS AND CONSIDERATIONS
Joseph G. Gross, Master of Science, 2010

Thesis Directed By: Dr. Bruce L. Jacob, Assistant Professor, Department of Electrical and Computer Engineering

The effects of a realistic memory system have not received much attention in recent decades. Often, the memory controller and DRAMs are modeled as a fixed-latency or random-latency system, which leads to simulations that are less accurate. As more cores are added to each die and CPU clock rates continue to outpace memory access times, the gap will only grow wider and simulation results will be less accurate.

This thesis proposes to look at the way a memory controller and DRAM system work and attempt to model them accurately in a simulator. It will use a simulated Alpha 21264 processor in conjunction with a full system simulator and memory system simulator. Various SPEC06 benchmarks are used to look at runtimes. The process of mapping a memory location to a physical location, the algorithm for choosing the ordering of commands to be sent to the DRAMs and the method of managing the row buffers are examined in detail. We find that the choice in these algorithms and policies can affect application runtime by up to 200% or more. It is also shown that energy use can vary by up to 300% by changing changing the address mapping policy. These results show that it is important to look at all the available policies to optimize the memory system for the type of workload that a machine will be running. No single policy is best for every application, so it is important to understand the interaction of the application and the memory system to improve performance and reduce the energy consumed.

HIGH-PERFORMANCE DRAM SYSTEM DESIGN CONSTRAINTS AND
CONSIDERATIONS

by

Joseph Gross

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2010

Advisory Committee:
Professor Bruce L. Jacob, Chair
Professor Donald Yeung
Professor Gang Qu

© Copyright
Joseph Gross
2010

Table of Contents

Chapter 1 Introduction.....	1
1.1 The Problem in Detail.....	2
1.2 Contributions and Work.....	2
Chapter 2 DRAM Devices.....	5
2.1 Introduction.....	5
2.2 Device Organization.....	5
2.2.1 DRAM Devices in Greater Detail.....	9
2.2.2 A Row Access in a DRAM Device.....	11
2.3 Mode Registers.....	14
2.3.1 Burst Length.....	15
2.3.2 Burst Type.....	16
2.3.3 Write Recovery, CAS Latency and Additive Latency.....	16
2.3.4 Auto Self Refresh and Self Refresh Temperature	18
Chapter 3 Memory System Organization.....	19
3.1 Typical Memory System Organization.....	19
3.2 Naming Conventions.....	20
3.2.1 Channel.....	21
3.2.2 Rank.....	22
3.2.3 Bank.....	24
3.2.4 Row and Column.....	25
3.3 Memory Modules.....	26
3.3.1 SIMM.....	27
3.3.2 DIMM and SODIMM.....	28
3.3.3 ECC DIMM.....	29
3.3.4 Registered DIMM.....	29
3.3.5 FB-DIMM.....	30
3.3.6 SPD chip.....	31
Chapter 4 DRAM Protocol and Timing.....	33
4.1 DRAM Commands: An Overview.....	33
4.1.1 DRAM Command Illustrations Explained.....	34
4.1.2 Row Access Command.....	36
4.1.3 Column Read Command.....	37
4.1.4 Column Write Command.....	38
4.1.5 Precharge Command.....	39
4.1.6 Row Refresh Command.....	40
4.2 Read Cycle.....	42
4.2.1 Read Cycle with Read-and-Precharge.....	43
4.2.2 Posted CAS.....	45
4.3 Command Interactions.....	46
4.3.1 Consecutive Reads To Different Rows In A Bank.....	47
4.3.2 Consecutive Reads To Different Rows In A Bank, Worst Case.....	48

4.3.3 Reads to Different Banks (Bank Conflict).....	49
4.3.4 Consecutive Reads to an Open Row.....	51
4.3.5 Consecutive Reads to Open Rows within a Rank.....	52
4.3.6 Reads to Different Ranks.....	53
4.3.7 Write to Write, Different Ranks with Open Banks.....	54
4.3.8 Write to Precharge.....	55
4.3.9 Write to Write with Bank Conflict.....	56
4.3.10 Read to Write, No Conflict, Different Ranks.....	57
4.3.11 Read to Write with Bank Conflict.....	58
4.3.12 Write to Read in the Same Rank.....	59
4.3.13 Write to Read in Different Ranks.....	59
4.3.14 Write to Read with Bank Conflict, Same Bank.....	60
4.3.15 Write to Read with Bank Conflict, Same Rank.....	61
4.3.16 Column Read-and-Precharge Timing.....	62
4.3.17 Column Write-and-Precharge Timing.....	63
4.4 Power and Performance Constraints.....	64
4.4.1 Four Bank Activation Window.....	65
4.4.2 Row-to-Row Activation Delay.....	66
4.4.3 2T Command Timing.....	68
Chapter 5 Power Modeling	69
5.1 Overview.....	69
5.2 Background Power.....	70
5.2.1 All Banks Precharged.....	71
5.2.2 One or More Bank Activated.....	72
5.3 Event Power.....	72
5.3.1 Activate Power.....	73
5.3.2 Read Power.....	75
5.3.3 Write Power	76
5.3.4 Termination Power.....	77
5.3.5 Refresh Power.....	78
5.4 Derating Power For Specific Systems.....	79
Chapter 6 Experimental Setup.....	80
6.1 Simulator Setup.....	80
6.1.1 Multithreaded Workloads and Thread Synchronization.....	83
6.1.2 Simulator Data Movement.....	83
6.2 Transaction Queue.....	85
6.2.1 Refresh Queue.....	88
6.3 Per-Bank Command Queues.....	89
6.4 Row Buffer Management Policies.....	91
6.4.1 Close Page.....	92
6.4.2 Open Page.....	93
6.4.3 Close Page Aggressive.....	94
6.4.4 Open Page Aggressive.....	94
6.4.5 Row Buffer Management Policy and Its Effects on Power.....	95

6.5 Address Mapping Policies.....	97
6.6 Command Ordering Algorithms.....	100
6.6.1 Timing Requirements.....	101
6.6.2 Timing Requirements – Channel.....	102
6.6.3 Timing Requirements – Rank.....	102
6.6.4 Timing Constraints – Bank.....	103
6.6.5 Command Ordering Algorithm: Strict.....	104
6.6.6 Command Ordering Algorithm: Bank/Rank Round Robin.....	106
6.6.7 Command Ordering Algorithm: First Available.....	108
6.6.8 Command Ordering Algorithm: Command Pair Rank Hop.....	109
6.7 Random Address Simulation Mode.....	112
6.8 Simulation Setup: DRAMsimII and M5.....	113
6.8.1 Benchmarks.....	115
6.8.2 Methodology.....	116
Chapter 7 Results.....	118
7.1 Power Results.....	136
7.2 Detailed Power Comparison – LBM.....	141
7.3 Command Ordering Algorithm Performance.....	143
7.3.1 Saturation Mode.....	144
Chapter 8 Contributions and Related Work.....	153
8.1 Summary and Contributions.....	153
8.2 Related Works.....	156
Chapter 9 Conclusions and future work.....	159
9.1 A Word on Multithreading.....	160
9.2 Future Work.....	161
Chapter 10 Bibliography.....	163

CHAPTER 1 INTRODUCTION

Over the past forty years, the performance of computer systems has steadily increased. The transistor count of integrated circuits has approximately doubled every 18 to 24 months, as predicted by the so-called “Moore's Law”. Increased transistor counts have contributed to the doubling of system performance as well as increased operating frequencies of those components. Until recently, it was possible to improve the speed of a single processor either through process technology or more efficient design. Designers are finding it very difficult to improve single-threaded performance now, because processor speeds are growing faster than the memory elements upon which they depend.

Recently there has been a trend toward having more cores on the same die in an attempt to increase performance by having more processors available to perform calculations [Jacob 07]. To keep up with these faster, more numerous processors, much larger and faster caches have been added to keep more of the data closer to the processors. Even though L3 caches may now be larger than 8MB, the problem still remains that at some point the processors will have to wait tens to hundreds of cycles for main memory. As working sets become larger, the growing performance disparity between processing elements and memory elements becomes more pronounced. Main memory is not a commonly or accurately modeled aspect of system simulation, so it is important to simulate this correctly as memory latency comes to dominate execution times.

In this thesis we explore the interactions of processors and memory, the arrangement of the memory system, with the aim of optimizing performance and reducing power usage. This is done by accurately modeling a real DDRx (Double Data Rate, generation 1, 2, 3, ...) DRAM (dynamic random access memory) system and looking to see what features and optimizations affect the execution time of various benchmarks as well as power dissipation for the same simulation runs.

1.1 The Problem in Detail

As computer systems become faster and more complex, the need to simulate the interaction of all the components in a real system becomes important. There are more processors in a system than ever before, running faster and interacting with the memory subsystem in a more complex way. While processors are often well-studied in full-system simulators, the simulation detail tends to become less detailed past the L2 or L3 cache. Often, simulators will model DRAM as fixed-latency or random latency. As systems grow increasingly dependent on the speed at which they can move data around, losing simulation fidelity for main memory can greatly affect simulation results and lead to unrealistic simulated behaviors. As the number of cores in a system increases, the problem will only get worse.

Much of the problem lies in the fact that access times for DRAMs are non-deterministic and non-uniform. If a request is made at some time, it is possible to anticipate when that request will be returned based on how many and of what type of requests are before it in the memory controller queues [Jacob 03]. These, in addition to the timing parameters and configuration of the DRAM system would yield a predictable but variable value. However, because the memory system implements priority queues, a subsequent request may preempt an older request and increase the latency of the original request.

Added to this is the fact that many memory controllers are not well documented. Many manufacturers will never describe how the memory system is laid out nor what policies are being used internally, so it becomes difficult to model what a memory controller should look like, much less how the DRAMs should behave when controlled by one.

1.2 Contributions and Work

This work attempts to explore the design space of a DDR1/2/3 memory controller. It also seeks to establish a good framework from which to develop, test and simulate future optimizations to memory controllers. Specifically, the contributions of this work are as follows:

- We create an architectural model by which a memory controller can be accurately simulated, cycle by cycle. The simulator is extensible and configurable so features may be changed or added without needing to affect the rest of the system.
- A system is developed by which minimum distances required between commands are calculated. At the time a command is issued to a DRAM, the minimum time to wait for various other types of commands is known. Many of the scheduling algorithms must know which commands will be available to execute first, so this calculation methodology allows a simple lookup to take the place of the calculations usually required to determine minimum command spacing.
- Power models are added which calculate the main sources of power usage in DRAMs accurately. The timing of commands, their spacing as well as their location is considered and a detailed breakdown of how the power was dissipated is reported on demand. The data is reported periodically so that power usage vs. time may be analyzed and different algorithms can be evaluated as to their performance and power usage simultaneously.
- Several models are developed for mapping physical addresses to memory system locations. Several other models are adapted and compared as well to look at their effects on performance and power usage.
- Several models are created to choose from existing commands and order them. These command ordering algorithms are adapted and compared for power and performance to see what their effects are.
- The standard close page and open page row buffer management policies are modified and optimized to take better advantage of open rows and reduce memory controller resource utilization. They are then compared against their original versions and compared for power and performance effects.

- Processing scripts to quickly and accurately generate web-ready reports of simulation runs were developed. These enable the user to simply generate web pages and graphics necessary to analyze, in great detail, the effects modifications have had on the system.
- Detailed statistics are gathered to help show the effectiveness of address mapping policies on the distribution of requests to channels, ranks and banks, showing graphically how well distributed the requests are.
- Simulation and analysis of the results of a number of benchmarks from the SPEC2006 benchmark suite. Simulation attempts every combination of address mapping policy, row buffer management policy and command ordering algorithm.

CHAPTER 2 DRAM DEVICES

2.1 Introduction

In order to better appreciate the workings of a DRAM system, one must be familiar with the circuits that make them work and why they are organized in this way. Most other types of memory, like SRAM (static random access memory), will retain their values as long as the circuit is powered. DRAM, however, must be periodically restored in order to maintain the values in the cells due to the charge in the capacitors leaking through the substrate. A strong understanding of how the underlying circuits of a DRAM device will make it clearer how the devices work and what design constraints are present for memory systems.

This chapter will look at the basic cells that make up DRAMs, the sense amplifiers and their interaction with the cells, the control logic, mode registers and some examples of optimizations made to solve certain efficiencies.

2.2 Device Organization

The following figure shows the organization and internal structures of a typical DRAM device. The DRAM array may have several configurations depending on the capacity of the device. Usually the DRAM array has an aspect ratio close to 1:1, although it is not uncommon to find a device with a 4:1 ratio. Frequently, devices will be described by their row count (e.g. 32,768), column count (e.g. 512), the column width (e.g. 4, 8 or 16) and the number of banks (e.g. 8). Alternately, the device may be described as having a certain capacity (e.g. 256MB) and width (e.g. x4, x8). This capacity is simply the rows * columns * column width * banks. The RAS (row address strobe) pin is used to tell the device that there is a valid row address on the ADDR pins (e.g. 15 pins for a family of devices with up to 32,768 rows). The ADDR pins also include the BA (bank address) pins to select which bank the row will be chosen from. In Figure 2.1, the banks are

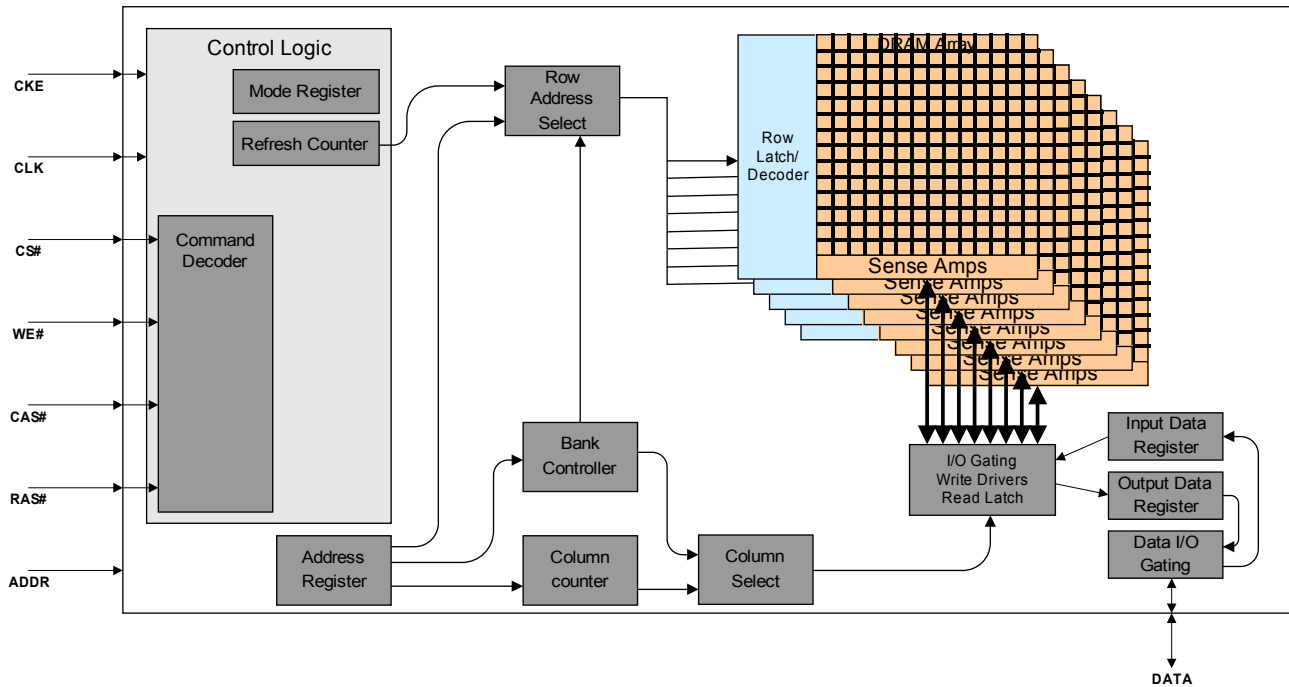


Figure 2.1: The organization of a typical DRAM device

shown as the stack of eight squares. Each of these banks has its own set of sense amps and therefore can be operated independently from the others.

Once the RAS pin is asserted, then the values in that row are read into the sense amps for that bank and it begins to restore the read values to the capacitors in that row. Once the values from the row are in the sense amps, the column address is placed on the ADDR pins and the CAS (column address strobe) pin is asserted. The number of columns is usually going to be far fewer than the number of rows, usually in the 128-512 range. Once the data is moved into the output registers, the device begins to stream out the data. Older devices will send the values in the requested location, while DDRx devices will send that value and subsequent values, thus sending more data per transaction. An optimization was created, called Fast Page Mode (FPM) DRAM, which allows multiple reads and writes to be performed after a single row activation. This is achieved by continuing to hold RAS low while issuing reads or writes. Not only is there no overhead for reactivating the row, there is no need to retransmit the row address repeatedly.

The CLK pin provides the clock to the device as a reference to synchronize the timing. Older DRAM devices were not synchronous, so the device would give a result as soon as it could. Adding a synchronous interface to DRAM, thus making SDRAM, now ensures that events in the chip and timing requirements are all defined in terms of cycles. Additionally, the timings of the SDRAM devices can be synchronized to match that of other system components.

The mode registers, which will be explained in more detail later, are to setup certain timings in the chip and to set behaviors like how many bits would be sent during a read burst, how often to refresh all the rows and what sort of delay to give read commands that are issued back-to-back with activate commands. These registers work with the control logic to define what timings should be used when controlling the sense amps as well.

The refresh counter maintains a counter that goes through all the rows in the banks. When a refresh command is issued or when the device is in self-refresh mode, this counter chooses which row will be refreshed at that time. Because each DRAM device keeps track of which row should be refreshed in turn, the memory controller does not need to keep track of this. Additionally, refresh commands do not need to send the row address with each refresh command. This saves having to store and transmit data to the device with each refresh command (refresh commands are quite common, approximately every 64ms). This counter is also used by the device when it is in self-refresh mode. Self-refresh mode is a way for the memory controller to effectively put the DRAMs into a sleep state but not lose the data. Because the cells will lose their values unless read from fairly often, the memory controller would need to continue to issue refresh commands as long as the system is running. Once in self-refresh mode, the DRAMs will do this automatically. The downside is that the device must be brought out of self-refresh mode before any activate, read or write commands can be issued.

Because the DRAM devices are meant to be used in conjunction with many others simultaneously, there may be hundreds of traces on a printed circuit board dedicated to the memory system. Designers have come up with many ideas to reduce the pin count of the devices as much as possible over the years. The row

and column addresses are multiplexed on the ADDR pins and row and column accesses are done separately. Although a read now is likely to require two commands, the pin count for the address bus is reduced significantly. The data bus is also bidirectional. Because reads and writes will not happen at the same time, the data bus is bidirectional. The data bus is the single largest use of traces in a memory system, so using the data bus bidirectionally greatly reduces the pin count and required real estate for the memory system.

Another evolution to the structure added the Read Latch between the sense amps and the output pins. Because the value is now buffered, the CAS pin can be de-asserted and a new transaction may begin while

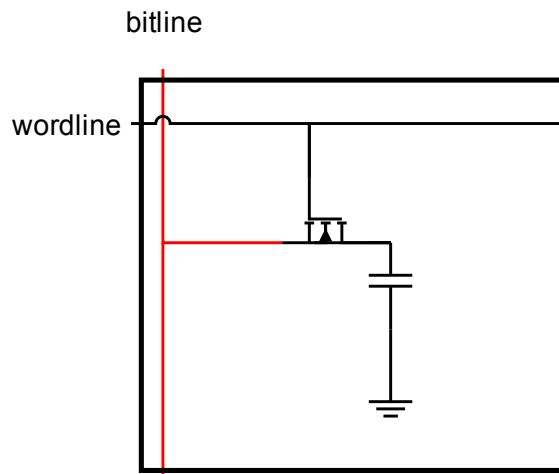


Figure 2.2: The 1T1C Storage Cell

the previous data is being read. This optimization was known as Extended Data Out (EDO) DRAM and can improve performance by 15% [Cuppu 2001].

A further refinement was the addition of a burst counter in Burst EDO DRAM (BEDO DRAM). This was an innovation which was the basis a common feature in DDRx DRAM, burst reads/writes. The burst counter was initially set to zero on a read command and incremented by one after each subsequent assertion of the CAS pin. The counter value was added to the column address so that the CPU could read many blocks in a row merely by toggling the CAS value. This cut down on the time required to get to the next value and removed the need to send column addresses between reads, reducing cycle latencies by as much as 30% [Prince 2000].

There are other improvements that have been made that will be explained later in this paper. The remainder of this chapter will describe DRAM devices in greater detail and show typical system organizations.

2.2.1 DRAM Devices in Greater Detail

Shown below is the most basic element of the DRAM array contained in each DRAM device. It consists of one capacitor and one transistor and is therefore commonly known as the 1T1C cell. The first DRAMs were created by Dr. Robert Dennard and built by IBM in 1966. This design, using capacitors for storage, was awarded patent 3,387,286 in 1968. In the following years, designs became available, including the Intel 1102 and 1103, which were very successful. Later, the Mostek MK4096 and 4116 followed, with the latter being a 16K chip and was very popular. The stored value of the capacitor is allowed to connect to the sense amps when its wordline is asserted. When a row is selected, it simultaneously chooses many wordlines to allow access to the many cells that make up a row. Depending whether the capacitor is mostly charged or mostly empty, it will make an effect on the bitline's value and allow a high or low charge to be amplified and sensed.

Because of the basic physical effects of having capacitors in integrated circuits, the charge stored in the capacitor will leak out into the substrate rather quickly, represented by the ground symbol in the diagram. Charge may leak through the access transistor or through the material surrounding the capacitor. If a cell is charged to V_{DD} , meaning that it represents a 1, there is a limited time before the value is reduced to the point where it can no longer be sensed definitively as either a 1 or a 0. To counteract this problem, DRAM systems have implemented refresh schemes for quite a long time. As mentioned previously, refreshing the rows must happen fairly often by sensing the values in the cells and either restoring the charge (or lack thereof). Due to process variation, some cells will retain charge much longer than others. However, the duration a value remains valid in a cell varies with age and temperature, so it would be very difficult to tune a system to refresh at a rate specific to that device. Therefore, manufacturers specify a maximum refresh interval (t_{REFI}) for the rows that, if not exceeded, should guarantee that the values remain valid in all situations. Often, the

value of t_{REFI} is $64\text{ms}/\#\text{rows}$ for normal situations and $32\text{ms}/\#\text{rows}$ for high temperature situations, as the leakage rate of the cells is directly proportional to temperature.

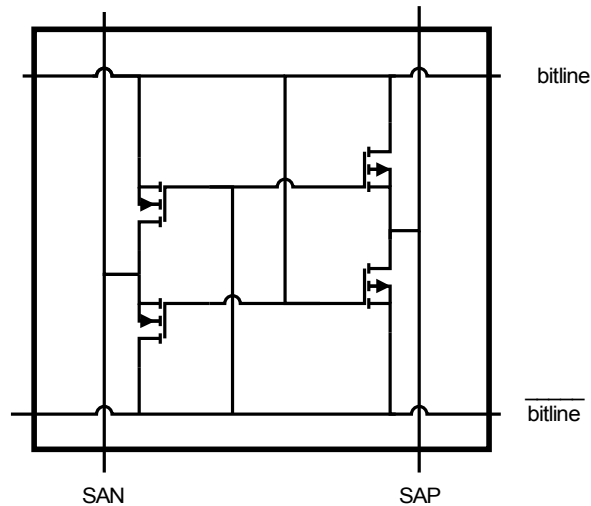


Figure 2.3: The sense circuit

The sensing circuit is a key part of the sense amplifiers. It consists of two NMOS and two PMOS transistors. The idea is that when the bitlines are charged to $V_{DD}/2$ and a capacitor is then connected to one or the other of these, the capacitor's V_{DD} or 0 value will slightly alter the charge on the line. This slight alteration will cause either the NMOS or PMOS circuit to begin to conduct for V_{DD} or 0, respectively. The SAN node is set to 0 and the SAP node is set to V_{DD} , so the other bitline will see increased or decreased voltage. The two rails are eventually driven to opposite voltages and thus the small amount of charge stored in the capacitor is amplified to V_{DD} and 0 on the bitlines.

There is also the voltage equalization circuit, shown in Figure 2.4, used to precharge the sense amplifiers' bitlines.

When the EQ line is taken to VDD, the three NMOS transistors are opened and this allows the bitlines to both be set to VDD. The other transistor ensures that they are precharged more quickly since they will have opposite charges and need to be set to the halfway point between them. This significance of this portion of the sense amplifier circuit is explained in the next section.

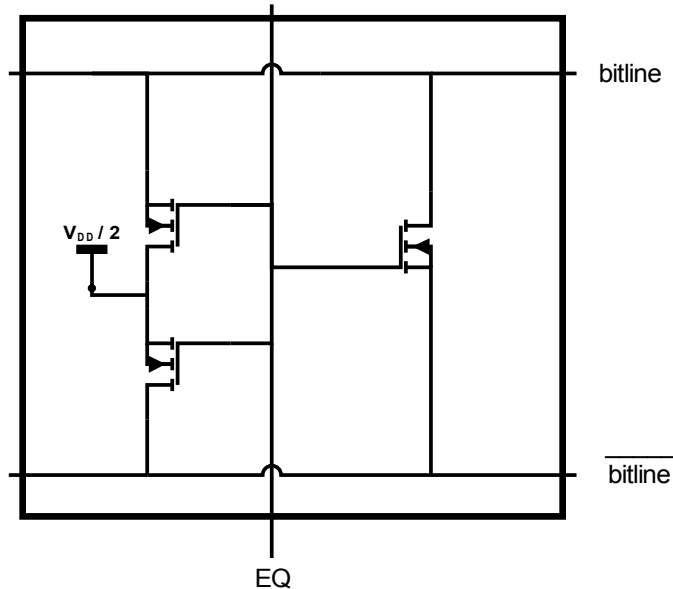


Figure 2.4: Voltage equalization circuit

2.2.2 A Row Access in a DRAM Device

In order to access a row in a DRAM, the row must first be opened. Then reads or writes may be performed and finally the sense amplifiers should be precharged. The precharge may happen at the end of the sequence or the beginning, but it must be done when leaving one open row to go activate another. Additionally, there need not be any reads or writes before a row is closed. There may also be a virtually unlimited number of reads and writes to the same row before it is closed (precharged). For the following diagrams, $V_{DD}/2+$ and $V_{DD}/2-$ represent values just above and below $V_{DD}/2$, respectively.

Figure 2.5 shows the first stage of a row access. The equalization signal is not asserted, so the positive and negative bitlines are effectively floating. The wordline is then activated to allow the capacitor's charge to influence the upper bitline. In this example, the capacitor was set to V_{DD} , so the bitline voltage increases.

Because the upper bitline voltage increased, the gate voltage of the NMOS transistor in the sense circuit was further opened, thus beginning to short the lower bitline to ground, taking its voltage to just below V_{DD} . Had there been no stored charge in the capacitor, the upper bitline would have been a slightly lower voltage and the lower bitline therefore would be a slightly higher voltage.

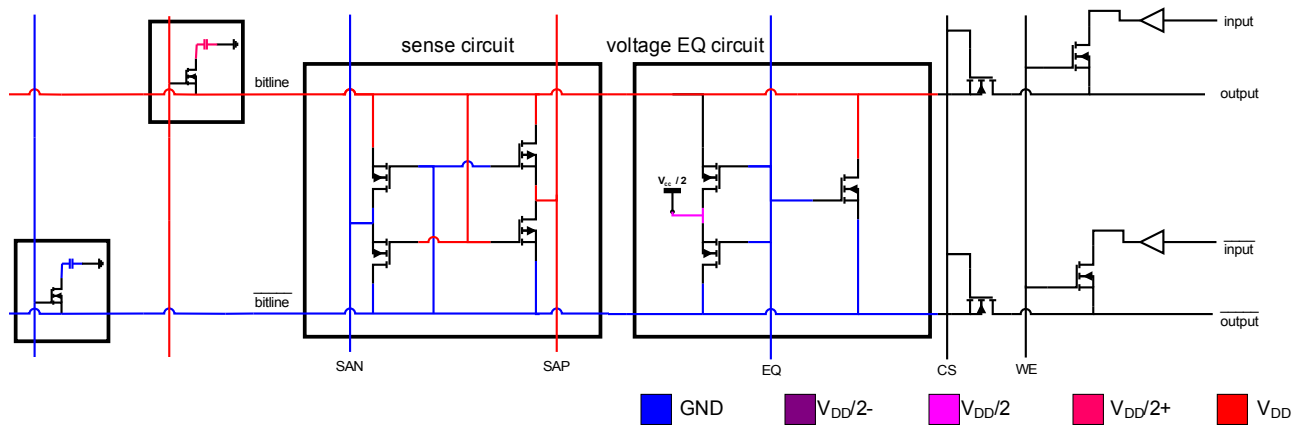


Figure 2.6: At the end of the sense portion of a row activation

After a short time, the upper bitline is driven to V_{DD} while the lower bitline is driven to 0V. The lower NMOS transistor is completely on at this point, while the lower PMOS transistor is completely off. Because the lower bitline is at 0V, the upper NMOS transistor is completely off and the upper PMOS transistor is completely on. Thus, the capacitor influenced the charge in the upper bitline. This in turn changed the lower bitline. Then this fed back to the upper bitline and eventually drove the bitlines to opposite values correctly representing the value stored in the capacitor.

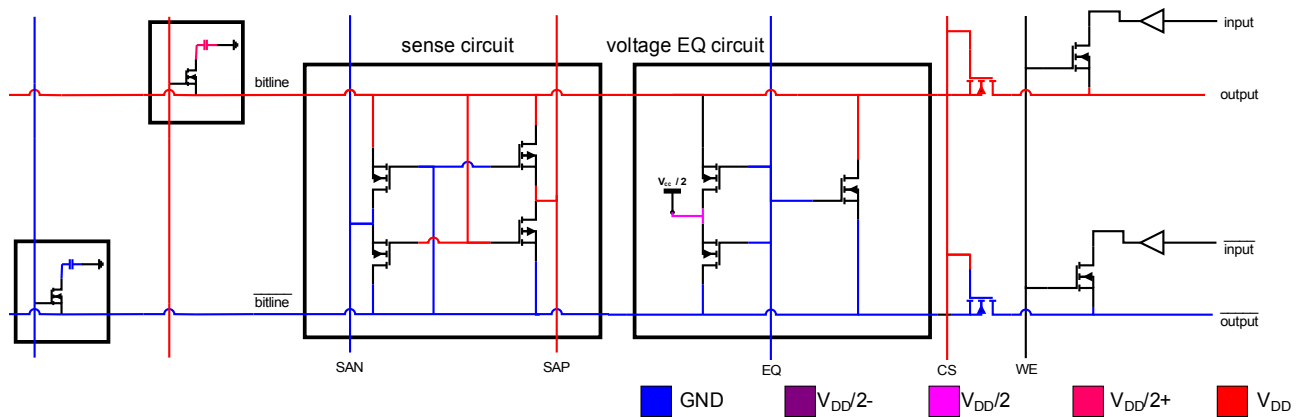


Figure 2.7: The restore and read portion a row activation

Once the value is sensed and the bitlines have settled, a read on any of the columns may be performed. This is accomplished by activating the chip select (CS) pin, which will connect the amplified value to the output of the DRAM device. There is the additional constraint of restoring the value from the capacitor to its original state. Because it was connected to a much later capacitance, the bitlines, its value

dropped to near $V_{DD}/2$. Likewise, if the capacitor had been at 0, its value would also be affected. By keeping the wordline open and maintaining this voltage on the bitlines for a minimum period of time, the original value can be restored. If this timing is not observed, the value may not be fully restored in the capacitor, leading to an undetectable value next time it is activated. The timing parameter t_{RAS} specifies the minimum time that must pass before the activated row is sensed and the values are returned to the cells. Once t_{RAS} has elapsed, then it is safe to assume that the values from the cells have been sensed and restored, so the row can be closed with a precharge operation. Most of the time a row will be opened for the purpose of reading from or writing to it. Because only the values on the bitlines must be stable for this to occur, the timing parameter t_{RCD} is defined to tell the system at what point after an activate a read or write may be performed.

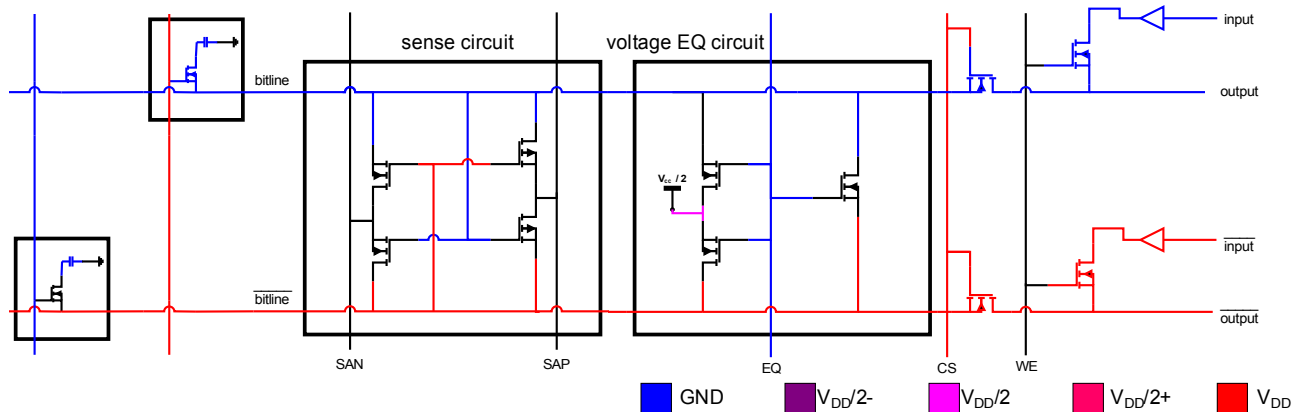


Figure 2.8: A write to an active row

While a row is open, it may also be written to. The wordline is kept open, as is the CS. Additionally, the write enable (WE) pin is asserted, which activates the transistor that controls access of the write buffer to the bitline. In Figure 2.8, the upper bitline was at V_{DD} , but the write buffer driver forced it to 0V. The other write buffer forced the lower bitline to V_{DD} so that the open row now contains the opposite value as before. At the same time, since the wordline remained open, the voltage in the capacitor was set to 0V, thus updating its stored value. Writing the value and charging the bitlines and the capacitor takes an extra amount of time after the restore is finished, typically given as t_{WR} in data sheets. After t_{WR} has passed, then the bitlines and capacitor are certain to be updated and the row may be precharged without leaving the storage cells in an ambiguous state.

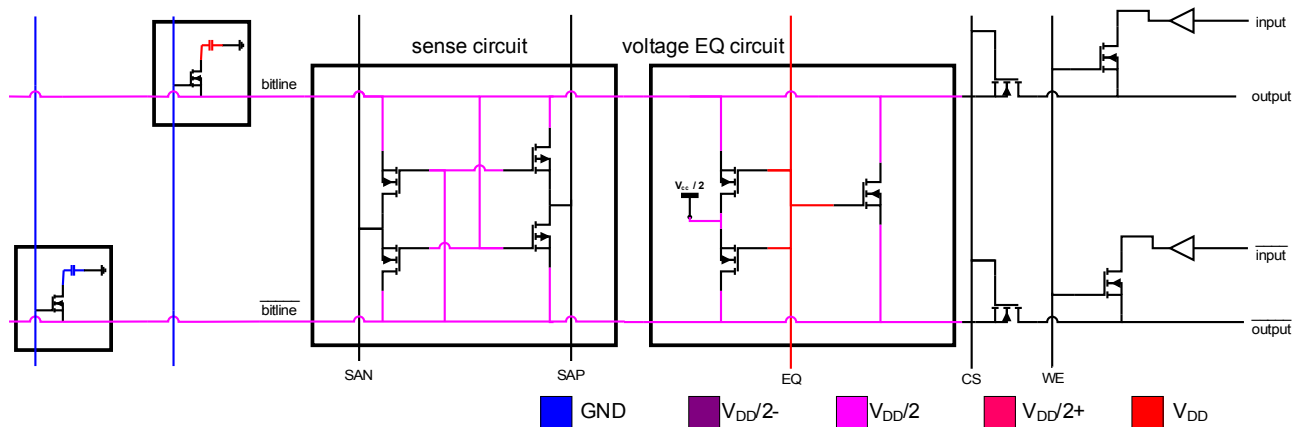


Figure 2.9: Precharging a row in preparation for a row activation

A final or first step in the activation process is the precharge. It is first or last because it must happen between row activations, so it is a matter of perspective as to whether it is first or last. As shown in Figure 2.9, the CS, WE and wordline controls are all de-asserted. The equalization (EQ) control is active. Just as mentioned for Figure 2.4, this sets the bitlines to $V_{DD}/2$, putting them in a metastable condition so that when a wordline is activated, the small perturbation caused by the charge or lack thereof from the capacitor will be sensed and amplified to a usable value. To ensure that the bitlines are fully precharged, the row precharge time (t_{RP}) must be followed. Once t_{RP} has passed, the bitlines will be equal, set to $V_{DD}/2$ and ready for another row activation.

2.3 Mode Registers

The mode registers control the behavior of the DRAM devices. Values in the mode registers are updated by sending a mode register set (MRS) command during initialization. The values remain until they are reset, the chip is reset via the RESET pin or the device is power cycled. When issuing MRS commands, one timing parameter that must be obeyed is t_{MRS} , which defines the minimum interval between MRS commands. Additionally, the memory controller must wait t_{MOD} for changes made to the registers are set and the behavior of the device is altered.

2.3.1 Burst Length

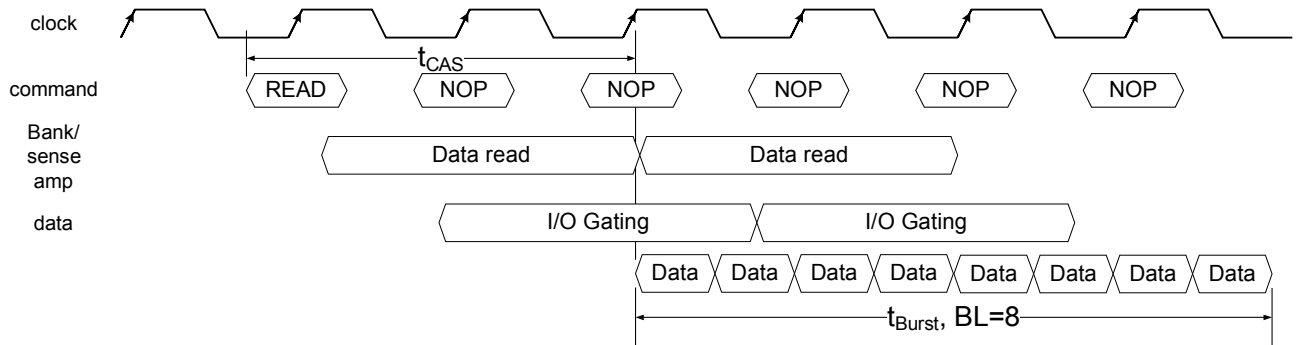


Figure 2.10: t_{CAS} and t_{Burst} , two mode register parameters, affect timing and behavior

One value that may be set in DDRx DRAMs is the burst length. The burst length defines how many columns are sent or received when a read or write is executed, respectively. For example, in a DDR2/3 SDRAM, bits n:2 are used to determine the block and the remaining bits are ignored when the burst length is 4. Similarly, bits n:3 are used to choose the block when the burst length is set to 8. Figure 2.10 shows the result of setting burst length to 8. The bits not used to select the block are used to select the starting byte within the block. When the starting byte is not set to 0, then the device will send the bytes, starting with the requested byte and wrap around when it reaches the end of the block (e.g. bits 2:0 \rightarrow 3, byte order \rightarrow 3, 4, 5, 6, 7, 0, 1, 2). The same applies for writes as well, the bytes may be written sequentially within a block, starting at any specified location. In DDR3, a burst length of 8 is typical and a burst length of 4 is known as 'burst chop.' This is because it is effectively chopping off the end of the burst. DDR3 also adds the additional feature of having a mode that allows the memory controller to dynamically select the burst length on each read or write command, so it can invoke a burst chop only when needed. This can be useful for times when longer or shorter bursts are needed. For example, when the memory controller is filling a request for a L2 cache, it may need to fill a 64-byte block. Because the data bus width is 64 bits wide, a burst of 8 will satisfy this request with one read command. However, if another device like a network makes a read request for 32 bytes, the memory controller can request a burst of 4 and not have to throw away the additional 4 bytes that were fetched but not needed. Additionally, this leaves 4 cycles open on the data bus that would otherwise

have been used. These cycles can be used to now read/write data from/to other DRAMs, rather than waiting for the burst of 8 to complete.

2.3.2 Burst Type

The burst type defines in what order the bytes within the selected block are read or written. The first byte will always be the one that is by address pins 2:0 or 3:0, but the following bytes may vary in order. If set to sequential mode, the bytes will be sent in the order as described previously, wrapping around if the initial byte is not at the beginning of the block. Interleaved mode, however, sends the bytes in a much different order. This is often done to reorder transmission errors. Error correcting schemes often expect that the errors in a transmitted group occur in a mostly uniformly distributed fashion, so if the errors occur in a large burst, the error correcting scheme may not be able to recover.

The other option for transmitting data is interleaved. Interleaving sends an entire block of 4 before sending the other block of 4, rather than treating the two blocks of 4 as one large block (as is done for sequential reads). Interleaving is essentially sending the entire upper or lower block before sending the other block, thus changing the byte transmission order. For DDRx memory, if the burst type is set to interleaved and the burst length is 4, the bytes will be sent in the order that wraps within the upper or lower 4 bytes (e.g. 3 → 3, 0, 1, 2 or 5 → 5, 6, 7, 4). If, however, the burst length is changed to 8, then the bytes will be sent with the upper or lower 4 bytes first, depending where the requested first byte is located. Then the other set of 4 will be sent, beginning with the same byte of the block as before (e.g. 3 → 3, 0, 1, 2, 7, 4, 5, 6 or 4 → 4, 5, 6, 7, 0, 1, 2, 3).

2.3.3 Write Recovery, CAS Latency and Additive Latency

The CAS latency and write recovery times may be programmed into the mode registers of a DRAM device. These values are helpful to help the device automatically carry out automated command sequences without losing data in the process. Although these values could be hard-coded into the DRAM devices themselves, rather than programmable registers, this would be problematic for DRAM device manufacturers.

Manufacturers can fabricate DRAM devices that are all intended to work using the most aggressive timing parameters. When imperfections in the fabrication process require some of the devices to be run at slower speeds, the only change that is needed is to program the mode registers differently. Additionally, some systems will want to use the devices at speeds slower than the rated maximum for increased reliability. This is easily done by simply programming the timing parameters differently at initialization.

The write recovery time specifies how long after a write has finished on the data bus that the input drivers, shown in Figure 2.8, must wait until the new values are fully written into the cells. Disconnecting the input drivers sooner than this could lead to capacitor values that are not recognizable as valid values when sensed the next time. This timing parameter is described in greater detail in a later section.

Although the memory controller can wait for this value, the DRAM device has complex commands that will group two or more commands into one command for the convenience of the memory controller. In this case, the “write and precharge” command. Once the write has finished, a precharge is automatically issued by the DRAM device itself. Therefore it must know what its own write recovery time is in order to delay the precharge to a point where the written bytes will be successfully stored before that row is close and the sense amplifiers are precharged in preparation for another row activation.

It is similar reasoning for the programmability of CAS latency, which is the delay between when a column read command is issued and the availability of the first bit of data on the data bus. If the data is not fully finished being sensed and the I/O drivers are not ready to transmit the correct data, then the system can read incorrect values from the DRAMs. The DRAM devices cannot know what speed grade they function correctly in, so the mode registers allow the memory controller to choose how long to wait for the data to be correctly read out of the cells.

Additive latency is a delay given to each read or write command once it is sent to the DRAM device. The point of this is to allow the memory controller to simplify its operation and reduce the amount of bookkeeping it must do. Once a row activation is issued, a column read or write may be sent immediately after.

2.3.4 Auto Self Refresh and Self Refresh Temperature

Self refresh is a mode of operation whereby the DRAM devices' CKE is set low and the clock to them is disabled. Then the device will continue to issue refresh commands automatically until it is returned to an active state again. This is a low power mode that will maintain the data in the devices for long periods of time without external intervention.

The Self Refresh Temperature (SRT) mode register is used to tell the DRAM devices what temperature they are expected to be operating in and what auto refresh rate they should use. When this is set, the internal self refresh rate is doubled. This is to counteract the effects of high temperature on the devices. Because the leakage current in the capacitors tends to grow greatly as the device exceeds 85°C, the rows must be refreshed at a greater rate to ensure that their values do not drop to unrecognizable levels.

Auto Self Refresh (ASR) is a feature that is new for DDR3 devices. If enabled, this can automatically switch the self refresh rate from 1x to 2x if the temperature exceeds 85°C, ensuring that there is minimal chance for data loss when the temperature fluctuates in self refresh mode.

CHAPTER 3 MEMORY SYSTEM ORGANIZATION

This chapter will take a look at the topology of common DDRx memory systems and explain how memory locations are addressed. Although single DRAM devices have been looked at before, only a few types of systems use single or few devices. Most systems group the devices into large groups and communicate with many at a time. The naming conventions presented in this chapter are important for later analysis of the performance characteristics of various configurations.

3.1 Typical Memory System Organization

The size of individual DRAM devices has been much too small to act as main memory for a very long time. Certain types of systems, like graphics accelerators and embedded systems, have different requirements and can function with just one or a handful of DRAMs connected to the memory controller. Most servers and workstations, however, need to have far more memory than this to function effectively. Overall system memory size has grown at roughly the same rate as DRAM device size, so there is a need to interconnect the devices to attain memory systems that are large enough to run modern workloads.

This chapter will look at some different memory system organizations and attempt to explain why such setups exist and what the limiting factors for those configurations are. In Figure 3.1, the memory system is organized as a group of DRAMs connected directly to a memory controller that can address any of the devices individually. The data bus may be shared amongst the devices or there may be multiple buses to allow concurrent reading and writing. There may be multiple memory controllers working simultaneously on different groups of DRAMs to fill requests for the CPUs, network cards, or I/O controllers more quickly.

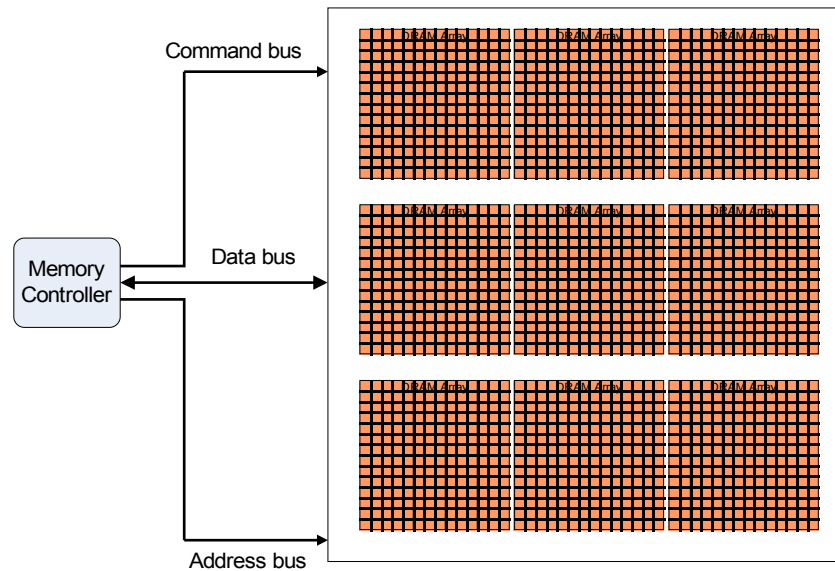


Figure 3.1: A simple memory system organization

The organization of DRAM devices into a memory system can have a great impact on performance and power. The latencies under heavy load are greatly affected, as well as the maximum and sustainable bandwidth. As more devices are added to a system, the connection to those devices becomes more important as well as the rate at which they operate. DDR3 devices may operate at 1600MHz and beyond. Packages rated at 1600MHz have to send and receive data at 1600MT/s (mega transfers per second). At these rates it becomes difficult to create printed circuit boards (PCBs) that can transmit this data consistently. Therefore, it is important to design the memory system so that the design requirements are not so great as to make system implementations prohibitively expensive or difficult.

3.2 Naming Conventions

It is therefore important to understand memory system design in greater detail. It is also important to know the nomenclature for this field so as to better understand the organizations and optimizations explained in the Experimental Setup and Results sections. Because the terms for the various parts of a memory system are unlike any other part of a computer system, a thorough look at the terms and organizations is key to seeing what is important in memory system design.

3.2.1 Channel

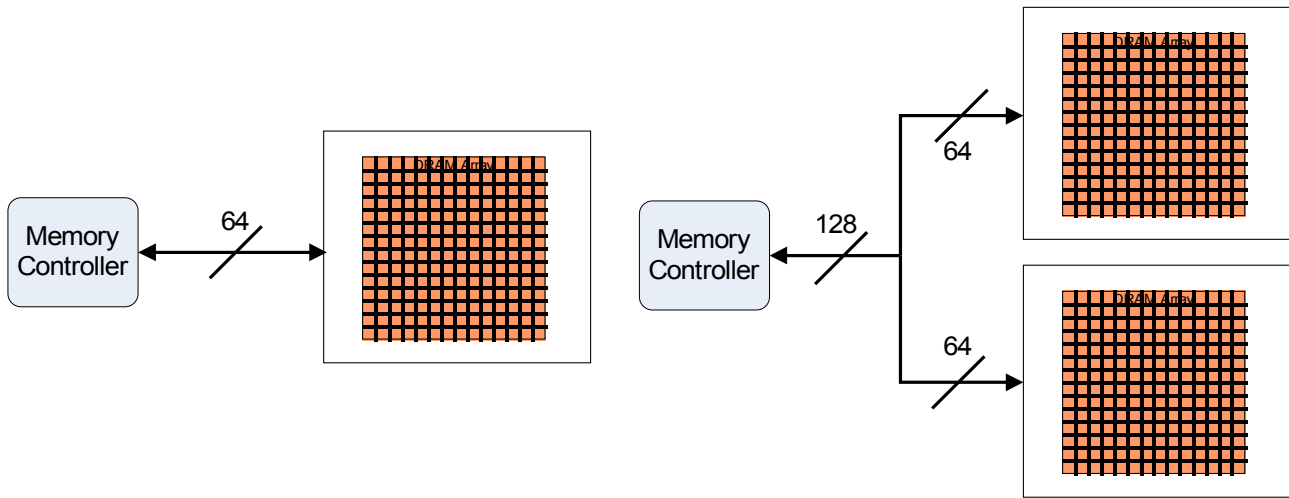


Figure 3.2: Two possible ways to configure a channel

Two different channel configurations are shown in Figure 3.2, one that connects directly to a DRAM device and one that connects to two different devices and treats them as one effective device. Each channel is controlled by a single memory controller, but there may be multiple controllers running channels in a system. In a typical workstation in 2009, it is quite common to have two channels. To achieve the 128-bit bus in the second example, two 64-bit memory modules are ganged together. These have separate data buses to allow simultaneous reads or writes, but share address and command buses. This is commonly known as a “dual channel” configuration. Some memory controllers, such as those on AMD CPUs (e.g. Athlon 64, Opteron, Phenom, Sempron) and many Intel system controllers (e.g. 940, 945, 975, 875) support dual channel configuration. Some of these are configurable so that the memory modules may be treated as a single 128-bit channel or two 64-bit channels. The advantage of having one 128-bit channel is that there are fewer commands to issue to store and retrieve data and the amount of data retrieved per read is doubled.

Using dual channel mode may not improve performance if the majority of the bytes from each transfer are not used. For example, if only one byte is used per read, then 15 bytes were moved that did not need to be. If the read was to satisfy a L2 miss from the CPU, 128 bits were moved into the cache. If this evicted one or more useful cachelines, which must then be written and reread later, performance will go

down. Power usage will also go up due to increased data movement as each read or write consumes a certain amount of power.

3.2.2 Rank

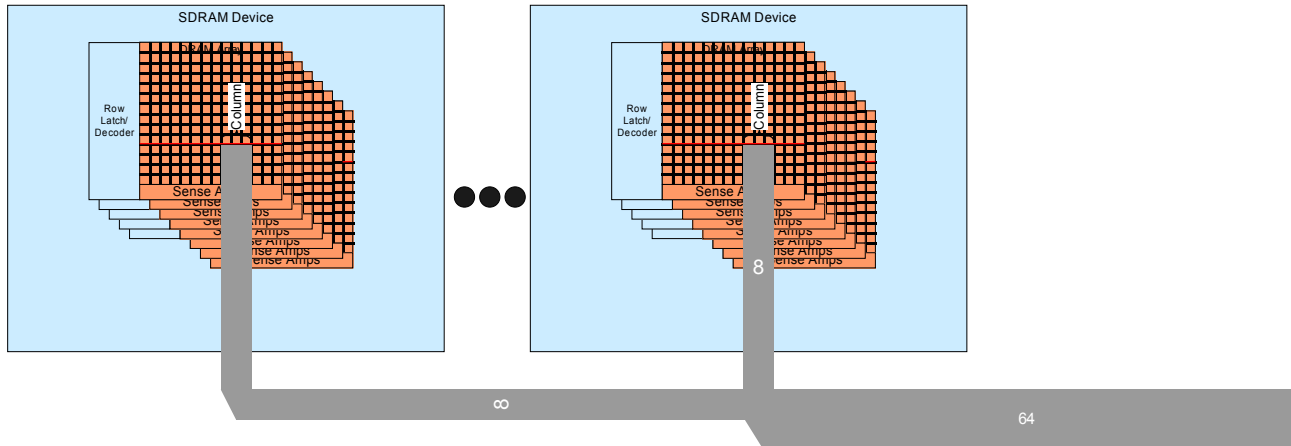


Figure 3.3: Controlling several DRAMs at a time to create a 64-bit channel

Looking at a DDR3 spec sheet [reference here], one of the first things listed is the fact that the part is available in x4, x8 and x16 variants. This refers to how many data pins or bits are sent at each interval during t_{Burst} . So the x8 part would have 8 data pins and addressing 8 devices simultaneously would give an effective channel width of 64 bits. Figure 3.3 shows how multiple DRAM devices can be combined to create a single, larger channel. All of the DRAMs are given the same address and control signals, so they act in concert when responding to commands. These groups are known as "ranks." It would be more logical, however, to refer to these groups as "banks," except that that bank is a term already reserved for the internal arrays within the DRAM devices. Since "bank" was taken, "rank" was chosen instead to attempt to avoid any confusion.

All ranks have common control, address and data buses. Because every device in every rank has every valid address, without some kind of arbitration mechanism, all the devices would fight for access to the data bus and the system would not work. To solve this problem, the memory controller has an additional control signal, the chip select (CS), to activate only the rank that it intends to address. The memory controller can then send commands at appropriate times to avoid having any contention for the data bus.

Figure 3.4 Shows one channel of a typical memory system. The channel has four ranks and four DRAM devices per rank. In a real system, this would appear to be two DIMMs (dual in-line memory module). Each DIMM has two ranks because the front and back are electrically separated, so each side is a rank with its own CS signal.

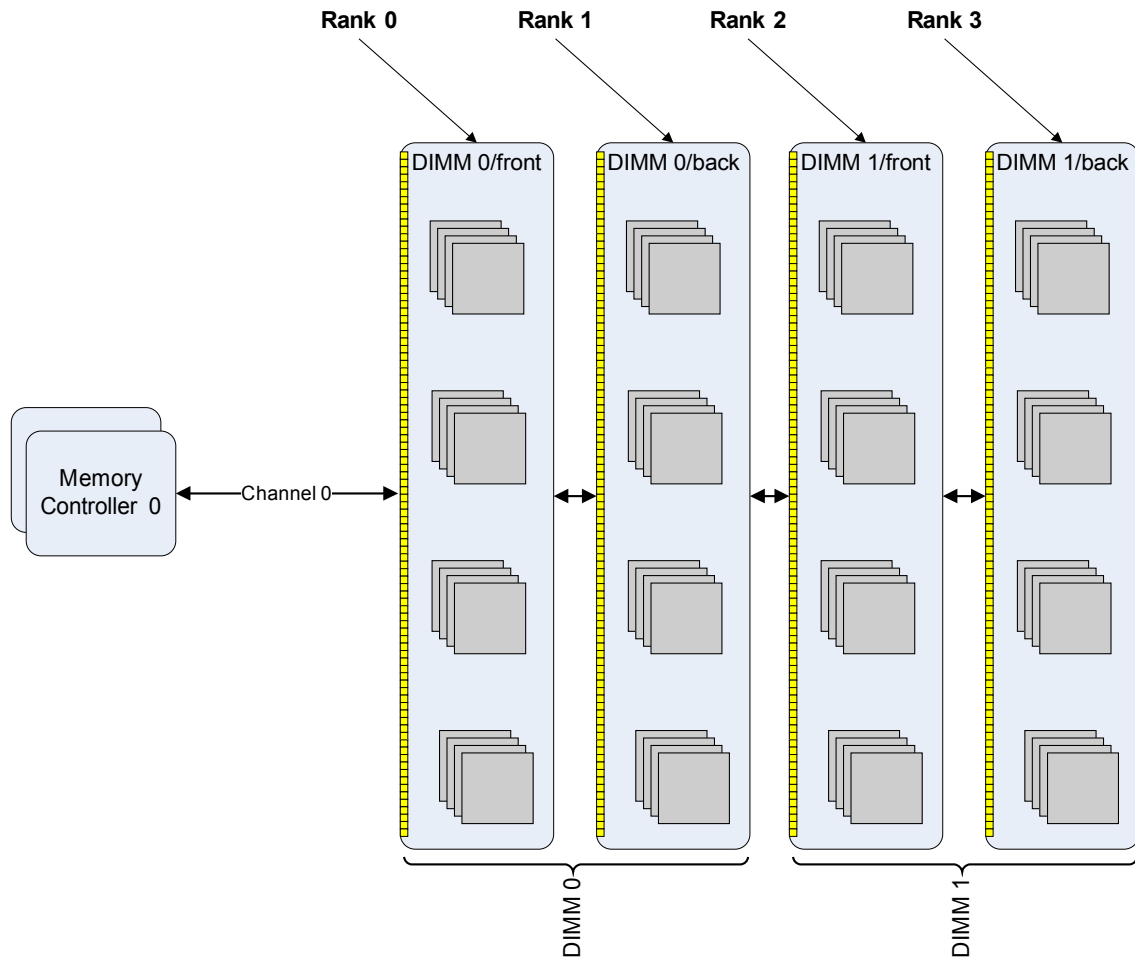


Figure 3.4: An illustration of how 4 ranks are grouped into a channel

Many modern systems are limited to 4 or possibly 8 ranks (2 or 4 DIMMS) per channel due to the fact that each device that is sitting on the shared buses tends to degrade the signal with reflections and capacitive effects. To mitigate this problem, DDR2 and DDR3 have added on-die termination (ODT) to attempt to perform impedance matching and reduce the reflections that degrade the signal. ODT is a resistor

network that is located on each chip and is activated every time there is a read or write to another DIMM on the channel. So one rank from each not-in-use DIMM must turn on ODT to improve reads and writes.

3.2.3 Bank

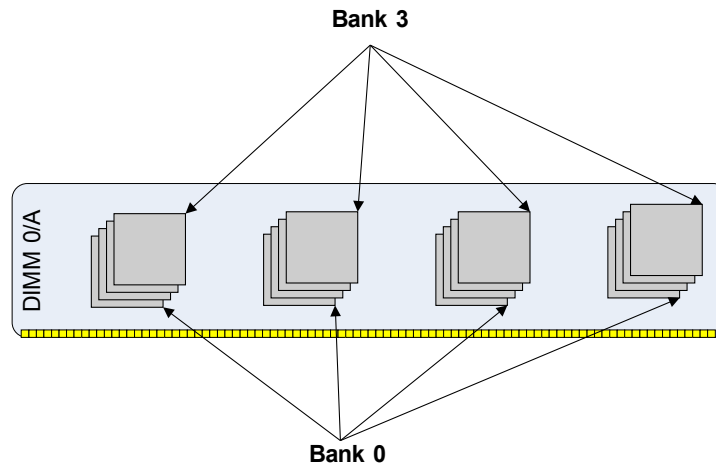


Figure 3.5: A single rank the the location of banks in relation to the rank

Although the term “bank” has been an overloaded word over the years, especially when referring to the memory hierarchy, in this case, a bank refers to the 2D array of capacitors and their associated sense amplifiers located within a DRAM device. Often, devices have 2, 4, 8 or 16 banks. The illustration in Figure 3.5 shows a single rank with 4 DRAM devices, each of which has 4 banks. Banks are independent from one another due to the fact that they can be simultaneously reading from or writing to a row or sense amplifier. One bank may be opening a row and sensing the data from it while another row is accepting data and writing it to the sense amplifiers and capacitors. The sense amplifiers, or “row buffers,” store an active row for that bank, as described earlier in this paper. They, however, share their control, address and data buses, so the memory controller may send commands to only one bank at a time, limiting their independence. Arbitrating between banks and scheduling commands is a critical issue for both power and performance, as will be discussed in the Experimental Setup and Results sections.

3.2.4 Row and Column

A row is a group of storage cells that are activated together when a row activation command is sent to the DRAM device. The active rows are buffered in sense amplifiers, where they can be read from or written to. In most typical DDRx systems where devices are arranged into ranks, a row refers to the open row spanning across all the devices that comprise that rank. Figure 3.3 illustrates this well. There is a row open in each of the 8 devices, but those are all combined to act as a row. This is because the devices in a rank are all addressed via the same signals and their data buses are combined.

Rows are also commonly referred to as “pages.” So when a row is activated, a page from main memory is opened. Typically these are several kilobytes in size. Within each row are many columns, or groups within a row as wide as the data bus. Depending what the burst length of the devices is set to, a DDRx device will send multiple columns per read request. In Figure 3.3, the columns that are being selected for a read or write are shown as connected to the data bus, meaning that any of the columns are eligible to be accessed at any point.

This arrangement was made to attempt to give an advantage to accesses with spatial and temporal locality. If two accesses occur at nearly the same time and are mapped to the same row, the access will be quicker. Specifically, it will only take t_{CAS} (the time it takes to move those values through the I/O gating and onto the bus) to read an additional value from a row already in the buffers. If the row must be close and another must be opened, then in addition to t_{CAS} , it will take t_{RP} (to precharge the sense amplifiers) and t_{RCD} (the time to open the row and sense the values). Mapping requests that are likely to occur within a short time of one another to similar locations will give great performance benefits. Also, because there are fewer commands sent and there are fewer precharge and sense operations, less power is dissipated. So finding ways to schedule requests to reuse open rows as many times as possible before switching is a key factor to consider when designing a memory controller.

3.3 Memory Modules

For several reasons, memory is often sold in packages containing many DRAM devices. Some architectures, like embedded devices and graphics cards, come with the DRAM devices soldered to the same circuit board as the processor. In the case of the embedded system, like a cellular phone or a portable music and video player, the device likely has one specific purpose that it will always be used for. It is unlikely to need memory upgrades for additional capacity or performance, so the added cost of sockets and add-on modules is unnecessary. It is also very likely that an embedded device is built to a specific price point, so reducing the number of required components and system complexity is of great importance. There is also the fact that these devices are meant to be portable, so they must be small, so having additional, expandable modules and sockets for them would likely make the device too large.

Likewise, a graphics card cannot incur the additional overheads of having memory modules for all the same reasons as an embedded device. However, graphics cards also have the additional constraint that they need higher performance DRAMs and are willing to sacrifice the flexibility of modularity to attain it. Adding an extra circuit board and the socket adds length to the address, control and data traces. It also reduces the signal quality because the socket is a worse connection than just a wire and it adds capacitance. When a DRAM device is sitting very close to a GPU (graphics processing unit), then these limitations are removed and the command and data buses can be run at significantly higher speeds with greater reliability. The connection is shorter and avoids many of the problems associated with longer trace lengths like capacitive coupling and crosstalk.

For servers, workstations and laptops, however, the requirements are different. A single model of server or workstation may have a dozen different memory configurations available. Some may ship with no memory installed at all, so the user will have the option of vendor, capacity, quantity and speed grade. So users of servers, workstations and laptops need to be able to purchase their own memory, replace memory if it fails, upgrade when needed and be able to ensure that what they buy will work in the system they already have.

3.3.1 SIMM

Beginning in the late 1980s through the mid to late 1990s, the single in-line memory module (SIMM) was the prevalent type of DRAM memory module. The distinction of being “single” and “in-line” is important because although there are contacts on both the front and back of the board, all of the contacts are connected to the contact directly on the other side of the board. Looking closely at Figure 3.6, one can see that each gold contact on the bottom of the board has a hole in it. That hole is plated with gold as well and makes an electrical connection to the pin on the other side.

Prior to SIMMs, system boards could be upgraded by plugging dual in-line pin (DIP) packaged DRAMs directly into the board. However, because there could be numerous DRAMs on a board, it could be difficult to locate and replace a faulty DRAM. Also, insertion and removal of these chips was difficult because the pins were fairly small and fragile and likely to bend. The SIMM addressed these problems by being easy to install and allowed faster diagnosis of bad DRAMs.

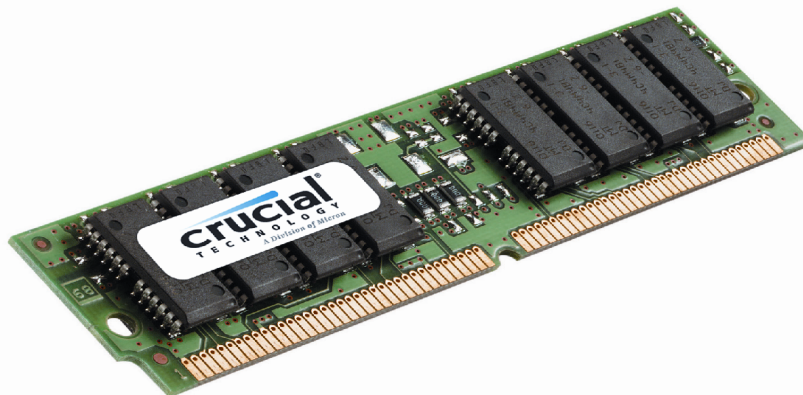


Figure 3.6: A 72-pin single in-line memory module

© Micron Technology, Inc.

There were two variants of the SIMM, the 30-pin version and later the 72-pin version. The 30-pin version had an 8-bit or 9-bit data bus, depending whether it had a parity line or not. Systems often required 4 identical 30-pin SIMMs to be installed at a time to make an effective data bus width of 32-bits. The 72-pin version had a 32-bit or 36-bit data bus, depending whether the SIMM had parity enabled or not. The parity version had one parity bit per 8 data bits. SIMMs usually had EDO or Fast Page Mode DRAMs on them as they predate synchronous DRAMs.

3.3.2 DIMM and SODIMM

Dual in-line memory modules (DIMMs) and small-outline dual in-line memory modules (SODIMMs) are the most common memory modules available today. DIMMs are most commonly found in servers and workstations, while SODIMMs are used almost exclusively in laptops and other portable devices. DIMMs improve upon the concept of the SIMM by implementing separate electrical contacts on each side of the circuit board. This gives designers the opportunity to have one rank on each side of the board and increase overall system memory capacity. This also allows the signal routing to be placed more closely than before and have close proximity for power and ground signals. Both of these help to reduce noise and improve signal integrity.



Figure 3.7: DDR333 SDRAM DIMM, 256MB with ECC

source: micron.com

Figure 3.7 shows a 184-pin DDR DIMM. This side has half of the DRAMs and the other side has the other half. Because there are two, electrically separate sides to this DIMM, they may be addressed individually as ranks. There is also the option to make a single-sided DIMM, with DRAMs on only one side and none on the other. In single data rate (SDR), DDR2 and DDR3 DIMMs, there are various numbers of pins. SDR had 168 pins and DDR2 and DDR3 have 240 pins. Although the pin counts for the various types of DIMMs differ, the width of the boards is the same. To prevent users from attempting to insert the wrong type of memory into a system board and possibly damaging the system, there is a notch in the bottom of the DIMM. For each type of memory, this notch is in a different position, so if someone is attempting to insert one type into the wrong system board, it will be physically impossible to insert it.

3.3.3 ECC DIMM

The device pictured in Figure 3.7 is a 333MHz DDR1 DIMM with error checking and correcting (ECC). Because the data bus must be 64-bits for this type of module, each of the DRAM devices must be a x8 part. However, there are actually 9 devices on this DIMM and this is because the last one serves to store the correction bits that will help to fix single-bit errors and detect double-bit errors. The memory controller uses these extra bits to store the bits of a Hamming code throughout the bits. In this way, if the bits are scattered across all of the DRAMs, any device failure will affect at most one chip. If the bits are positioned like this, the contents of memory can be reconstructed even without an entire chip. This scheme is often known as chipkill. In some more advanced server systems, once a chip has failed, a spare, previously unused chip can be activated to replace the damaged chip and operation can continue.

3.3.4 Registered DIMM

In large servers that handle many simultaneous requests, it is important to have plenty of main memory. Often in server environments, the available capacity of memory is the most important factor, so the registered DIMM is often used there. As figure Figure 3.8 shows, the data bus is connected to the memory controller, which is like a typical memory system, but the address and control buses are buffered through a registered latch.

The purpose of this is to reduce the loading effects on the bus by requiring the memory controller to drive only one device at a time, the register. Since the memory controller now sees only one device connected to the address and control buses, the capacitive effects from the longer traces are reduced. The portion of the address/control bus that is on the system board can be optimized to take advantage of the reduced loading and have more devices attached to the channel. On a typical system board, there are often 2 DIMMs per channel, but on a board that supports registered DIMMs, 4 or 8 DIMMs is common.

This does not come without a price, however. Because the signals must be registered and then passed along, accesses will take an extra cycle before the data can be sent or received. This is often considered a

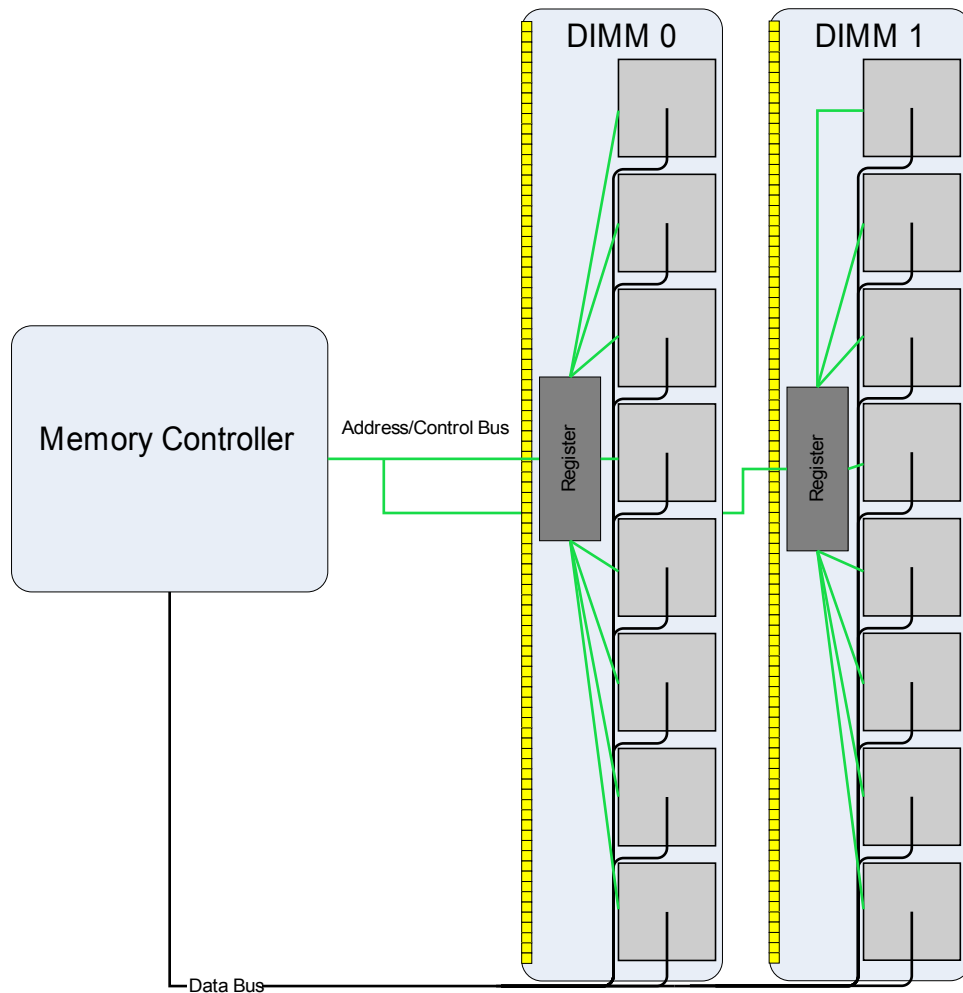


Figure 3.8: Memory controller connected to two registered DIMMs

small penalty in exchange for the ability to put more DIMMs onto a channel and have increased signal integrity to connected DIMMs.

3.3.5 FB-DIMM

The fully-buffered DIMM, or FB-DIMM is part of an alternate memory system that is meant to increase the performance, reliability and capacity of a memory system [Ganesh 07]. DIMMs in this type of memory configuration have the standard DDRx DRAM devices on the DIMMs, but also have an advanced memory buffer (AMB). The AMB's purpose, similar to that of the registered DIMM, is to buffer the signals and improve signal integrity. Instead of only buffering command and address signals, the AMB buffers all signals, including data.

The data transmitted between the memory controller and the AMBs is sent in a serialized format. The AMB deserializes the commands and data intended for the DRAMs and sends the data in parallel, just as a traditional memory controller would. When data is returned from the DRAMs, the data is serialized by the AMB and transmitted back to the memory controller.

The interface to the FB-DIMMs is relatively narrow, with two unidirectional buses. One, called “southbound,” is from the memory controller to the AMBs and is 14 lanes wide. The other, called “northbound,” goes from the AMBs to the memory controller and is 10 lanes wide. The reason that they are referred to as “lanes” is that they are not fixed in what they can do. These lanes must be 12x as fast as the basic memory clock, so if the underlying DDR3 DRAMs are running at 400MHz, the FB-DIMM channels will need to run at 4.8GHz. If one or more of the lanes are detected as malfunctioning, it will be disabled and the other lanes used instead, giving greater reliability. The system can suffer a loss of several bitlanes without losing the ability to transmit data by reducing the number of bits dedicated to cyclic redundancy check (CRC) in each transmission[Ganesh 07-2]. So as lanes become unavailable, the error detection ability is reduced, but performance is not degraded.

When a write occurs, the data is transferred to the FB-DIMMs more slowly than it is written to the DRAM devices. So the AMBs must buffer this data and delay until they can send the data burst continuously. The AMB also contains features more commonly associated with more complex controllers like temperature sensors, so it resides somewhere between being a register and a full-fledged controller. It is important to thermally control the AMBs, as the high-speed I/O drivers tend to consume a lot of power and may heat up the AMB to an unacceptable level during high load levels.

3.3.6 SPD chip

When looking at the upper right part of the board shown in Figure 3.7, one may notice a small, 8-pin IC that seems unrelated to the rest of the circuit. This is actually the serial presence detect (SPD) chip and it stores configuration information about the DRAM devices that are on this DIMM as well as the timings that

it supports. Because DRAMs do not have any ability to store their own capabilities and it would take a while to experimentally determine what timings and capacities are available in the DIMMs, the SPD

Once a system is turned on, it begins by performing a power-on self-test (POST). During this procedure, the DIMM slots are polled, via the System Management Bus (SMBus), to determine the amount of memory present as well as the timings available. Some systems will choose to set timings at the speeds of the slowest module, others will account for the timings of the separate DIMMs individually. Many systems will allow the user to specify timings explicitly to override the values in the SPD. Each SPD will contain three timing sets, one for the fastest available clock rate and two for progressively slower clock rates. Table 3.1 shows a few of the parameters read from an actual SPD.

Parameter	Value
Maximum module speed	800 MHz
Fundamental Memory Type	DDR2
Size	2048 MB
Banks, Rows, Columns, Bits	8, 16384, 1024, 72
Ranks	2
Module Type	Unbuffered DIMM
Interface Voltage	1.8V
Supported CAS Latencies	3T, 4T, 5T
t_{RP}	12.50 ns
t_{RCD}	12.50 ns
t_{RAS}	45 ns
t_{WR}	15 ns
t_{RC}	57.5 ns
t_{RTP}	7.5 ns
Manufacturer	Kingston
Manufacturing Date	2008, week 33
Serial Number	0xA6CCD088

Table 3.1: Several of the values contained in an actual SPD

CHAPTER 4 DRAM PROTOCOL AND TIMING

To fully understand the complexities of a memory system, one must see the timing and interactions of memory commands. Previous chapters briefly look at memory timings for a few simple commands. These will be revisited in greater detail and then examined in greater detail when several commands are grouped together.

The timing and structure of the memory access protocol plays a key role in the latency and bandwidth of a memory system. The memory access protocol is defined by the available commands at a given time and the timings that must be followed to ensure that data is read, written and preserved properly. Within these rules are opportunities to more efficiently move data or use power, but the protocol must be followed.

Chapter 4 will examine the memory access protocol in greater detail. It will focus on DDRx memory systems as this is the type of memory that is simulated in the results section and is quite commonly found in servers and workstations. Although there are additional memory variants which offer additional features like write buffers and limited row activations, this chapter will focus on DDRx commands so as to cover as many common memory systems as possible. This will allow analysis to be generic enough to resemble most common memory systems available today, but accurate enough to accurately represent what a real world memory system is doing in some detail.

4.1 DRAM Commands: An Overview

This chapter will examine in detail how commands are issued, what the restrictions are and what is necessary to achieve good performance when issuing multiple commands in a short span of time. The memory controller often has many pending requests that could be sent potentially all in sequence to a specific DRAM location. The controller cannot do this, however, because there are many timing restraints,

some of which have been discussed briefly in Chapter 2 . In many cases, if the minimum timings are not met, the transmitted or received data will possibly be corrupt or the data stored in the DRAMs will be corrupted.

Many of the timings are mostly based on resource usage. If the output drivers are in use for a period of time, then any command which would cause these drivers to be needed must wait. However, because the DRAMs use resources after some delay, the delay caused by the DRAMs themselves must be accounted for. For example, the data bus, a shared resource, is not used until t_{CAS} from the time the command is sent, but is then in use for t_{Burst} . So the memory controller must keep in mind that from t_{CAS} to $t_{CAS} + t_{Burst}$, the data bus is in use and any command that would requires the data bus in this space of time must delay until the bus is free[*Jacob 03*].

Some commands are limited not by resource contention but rather by power constraints. These few commands are limited by how often a particular bank or banks can be activated. This helps to ensure that the current through the device is kept below some threshold and the device will dissipate less power as a result.

4.1.1 DRAM Command Illustrations Explained

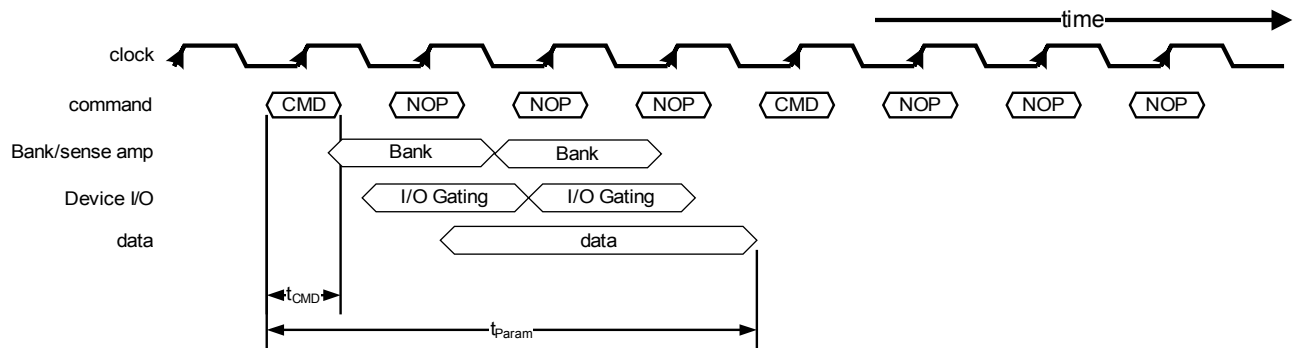


Figure 4.1: : The basic format used to describe DRAM commands

To describe commands in a generic and uniform way, the command or commands will be presented on a timing diagram like that of Figure 4.1. Each command takes t_{CMD} to be sent to the DRAMs. This will not change from command to command, so it is only shown once. Because the command and address bus, labeled as “command” for simplicity, is in use for t_{CMD} , no other commands may be sent until this first command is finished being transmitted[*Cuppu 99*].

At the top of the diagram is the is the clock. This clock is used to keep the DRAMs in time with the memory controller. The clock in a DDRx system is actually a differential clock, meaning that the clock and its inverse are transmitted to improve signaling characteristics. However, for the sake of simplicity, only the positive clock is shown in these timing diagrams.

Typically timing parameters are measured from the end of the command to the end of whatever sequence is being performed. This is because the command is registered on the clock edge and the action can start at this point. It also allows a uniform way to measure timing parameters. So if some minimum spacing is t_{Param} , then the memory controller must measure at least t_{Param} from the beginning of that command to the beginning of the next command or from the beginning of the first command to the beginning of the second command. Because t_{CMD} is fixed, it does not matter where the measurement is from as long as it is consistent. Figure 4.1 shows that t_{Param} is the limiting factor that delays the second command. The measurement is taken from the start of the first command to the start of the second command.

This chapter will examine commands and their timing requirements, when certain parts of the device are in use, when to expect data back from a read, and when to send data for a write. Then on to more complex commands, commands that are actually multiple commands at once and depend on mode registers for proper functionality. Finally, timing between multiple commands is studied, as good performance depends greatly on the ability to pipeline commands efficiently.

4.1.2 Row Access Command

A typical row access command is illustrated in Figure 4.2. As previously described, this command tells the wordlines for a particular row to charge and allow the sense amplifiers to read the values from an entire row of cells. Then the values are restored into the cells and the row activation is complete. Note that Figure 4.2 shows only one DRAM device. In a real system, multiple DRAMs in a rank would be addressed at the same time with the same timing constraints.

There are two important timing parameters associated with a row activate command (RAS, named because the Row Activate Strobe signal is asserted on the command bus), t_{RCD} and t_{RAS} . From the time the RAS command is received, t_{RCD} must elapse before the sense amplifiers have recorded the values correctly.

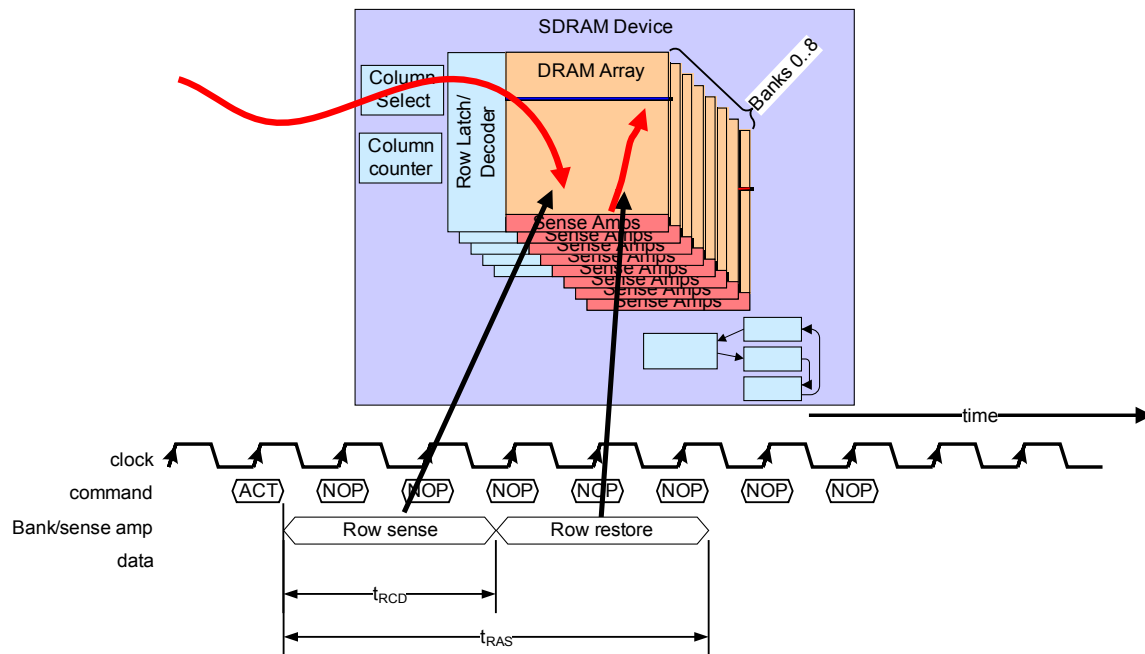


Figure 4.2: Timing and data movement of a row activation command

The memory controller must wait t_{RCD} before performing any operations that depend on having an open row or else the data will not be valid. Although the data is successfully sensed after t_{RCD} , the operation does not complete until the data is restored to the cells. The time it takes for sensing and restoration is called t_{RAS} . t_{RAS} is a significant parameter because it must have elapsed before the sense amplifiers can be precharged and another row opened.

4.1.3 Column Read Command

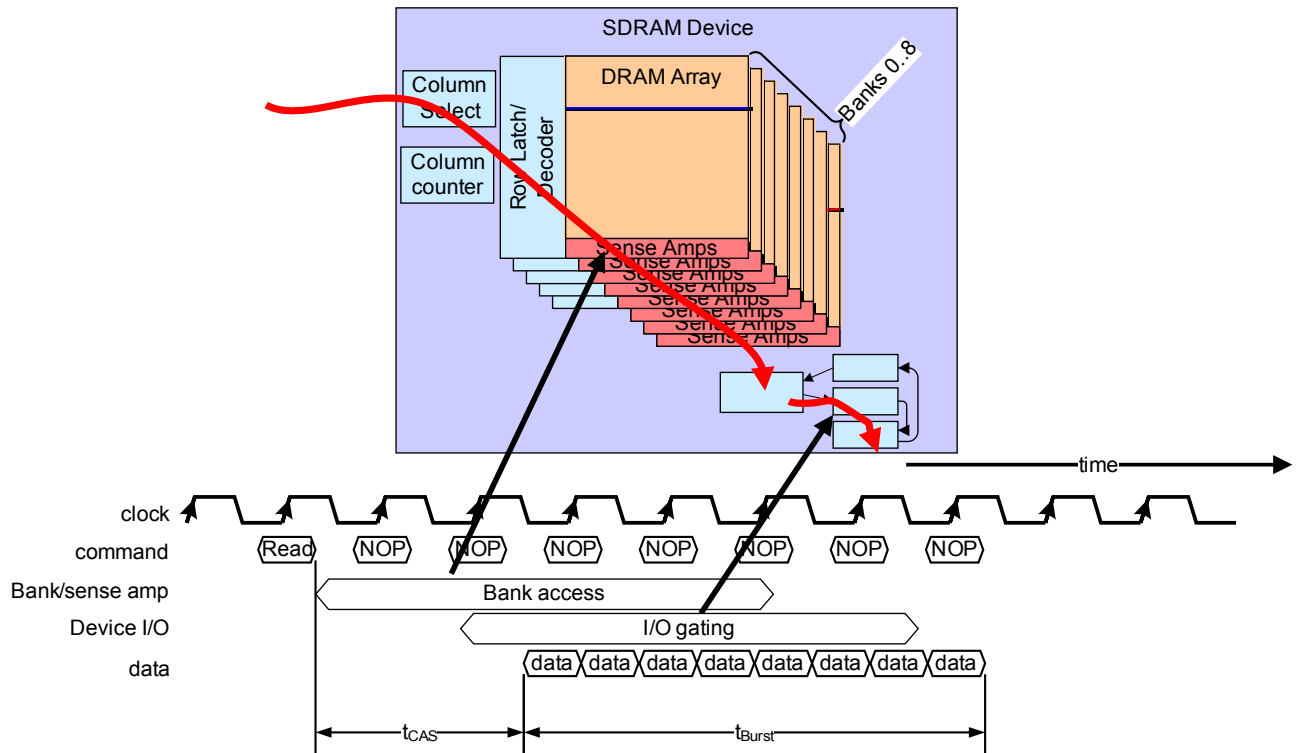


Figure 4.3: A column read command with a burst length of 8

Another very common command, the column read or CAS (Column Access Strobe, after the signal that signals that a read command should commence) moves a section of the data in the sense amplifiers and moves it to the output buffers. The requested column must be decoded and selected. After this, the data is moved to a series of registers that will be serialized for output. These steps take t_{CAS} and is programmable via the mode registers, usually 5-10 cycles. Although this can be programmed, setting the value too small will result in data that is not fully transferred to the I/O gating and thus corrupt data may be read.

From there, it is sent across the data bus in portions. The bus width per DRAM device is usually 4, 8 or 16. This is denoted as x4, x8 or x16 as described earlier. The burst length that has been programmed into the device's mode registers controls how many pieces of data are sent. Figure 4.3 shows a read command with a burst of 8, the longest allowable burst in DDR1/2/3. Instead of describing the data burst in terms of cycles and clock rates, it is simply defined as t_{Burst} .

4.1.4 Column Write Command

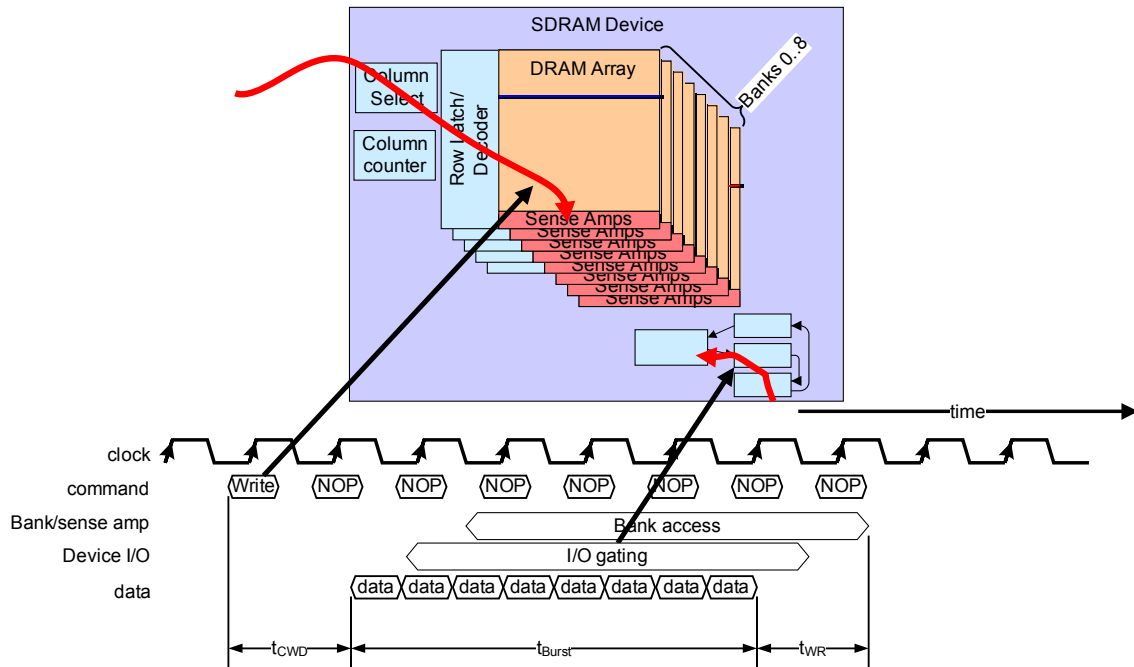


Figure 4.4: A typical write command in a DDRx DRAM

As shown in Figure 4.4, a column write is quite similar to a column read except that the stages are reversed, the data arriving first and the bank access happening last. One important parameter is t_{CWD} (sometimes known as t_{CWL}). t_{CWD} describes the delay that the memory controller must insert between when the write command and when the data. The value of t_{CWD} is protocol dependent and has varied with each generation of SDRAM. The values of t_{CWD} are listed in the table below.

Memory Type	t_{CWD}
SDRAM	0 cycles
DDR SDRAM	1 cycle
DDR2 SDRAM	$t_{CAS} - 1$ cycle
DDR3 SDRAM	programmable

When using SDRAM, the data and command may be sent at the same time without problem. DDR requires 1 cycle and DDR2 requires 1 cycle less than t_{CAS} , because t_{CAS} is programmable. DDR3 is a bit different in that t_{CWD} is somewhat independently programmable. This means that t_{CWD} may have specific allowed values per t_{CAS} setting. For example, in a Micron 1Gb x4 DDR3 SDRAM, t_{CWD} may be set to 5 ns

when t_{CAS} is set to 6 ns, but t_{CWD} must be 6ns when t_{CAS} is 7 or 8 ns. The memory controller must be careful to take note of which t_{CWD} settings are available for a given t_{CAS} setting to avoid violating timing constraints when using DDR3 SDRAM.

The final new parameter is t_{WR} , which is the time it takes for the data to be propagated to the cells after it has been received by the DRAM device. This restricts what activity may happen to the sense amplifiers as subsequent operations that must use them must wait to avoid corrupting the data that is being stored in the cells.

4.1.5 Precharge Command

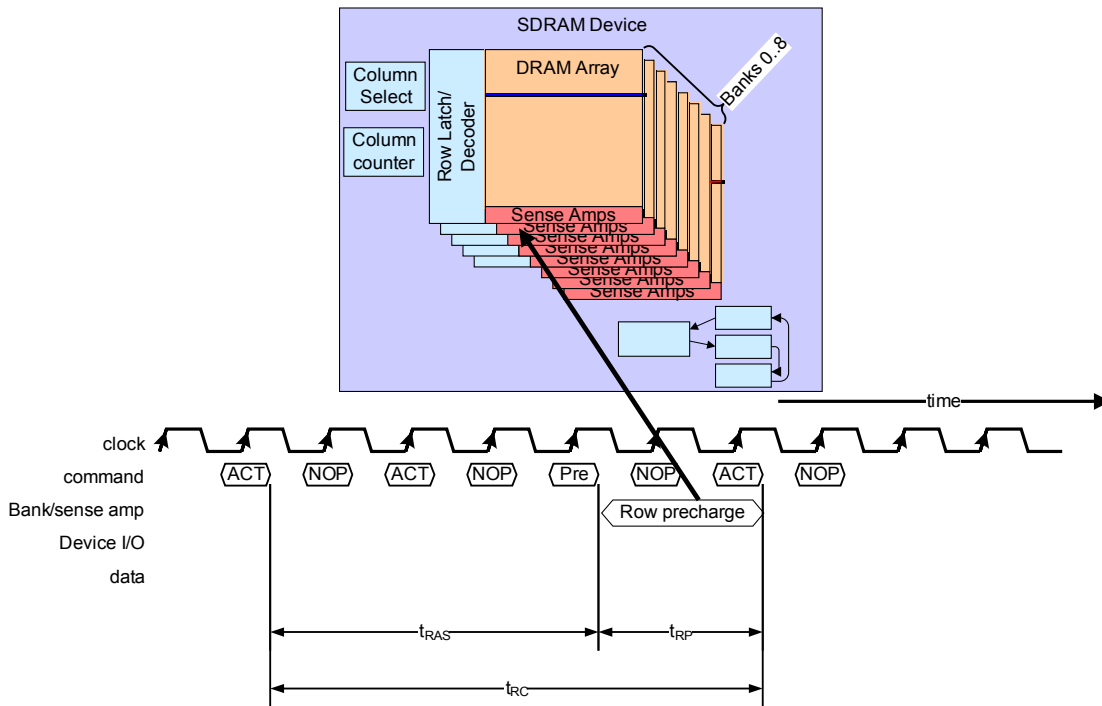


Figure 4.5: A precharge command preceded and followed by row activations

The precharge command, which charges the sense amps as described previously, is the counterpart the activate command. Whenever a row is opened and the memory controller needs to read or write to another row, a precharge command must first be issued to prepare the sense amps for another sense operation.

Figure 4.5 shows a row being activated, then precharged, then activated again to open another row. This diagram also shows the minimum available timing that these commands can be issued in. For the first row to be sensed and restored before the precharge, the precharge command must hold off for t_{RAS} . After t_{RAS} , the data is restored and a precharge command won't destroy any data. From the time the precharge is issued, the memory controller then must wait t_{RP} before activating another row. This gives the voltage equalization circuit time to charge the bitlines in preparation for sensing another row. The sum of these two timing parameters is t_{RC} , (row cycle) which is an important parameter for a memory system.

Because t_{RC} defines the minimum time that different rows can be opened, it is a fundamental limit on how quickly the data from different rows can be accessed. Thus it becomes important to either reuse rows as many times as possible to spread this cost over several reads and writes or to close rows as quickly as possible to be ready to open other rows. In modern systems with many running processes across several processors, the likelihood of finding multiple requests to reuse the same row is declining. Because different processes have ever-increasing data sets, the ability of a row to hold the good amount of that data set is reduced. There are several approaches to try to hide or reduce the impact of the row cycle time, some of which are discussed in the experimental setup and results sections.

4.1.6 Row Refresh Command

Because the values in DRAM cells are stored using a large array of capacitors, the system is subject to the same effects of other capacitors. This includes leakage over time, so the values in the cells leak into the substrate and become unrecognizable. In order to counteract this effect, the values must be read and restored periodically. This is the “dynamic” aspect in Dynamic Random Access Memory: the values will be lost if they are not restored from time to time. As the fabrication process varies from cell to cell, the leakage rates vary as well. This means that some cells can hold their values for a relatively long period of time while others must be refreshed more often. In order to ensure that the values are not lost in any of the cells, DRAM manufacturers specify a minimum interval that all the rows must be refreshed within. They specify a minimum value, t_{REFI} , that specifies the minimum average periodic refresh interval. This value is often

specified for two temperature ranges, up to and above 85°C. 64ms is often the interval below 85°C, while 32ms is specified for higher temperatures (within the functional range of the DRAM).

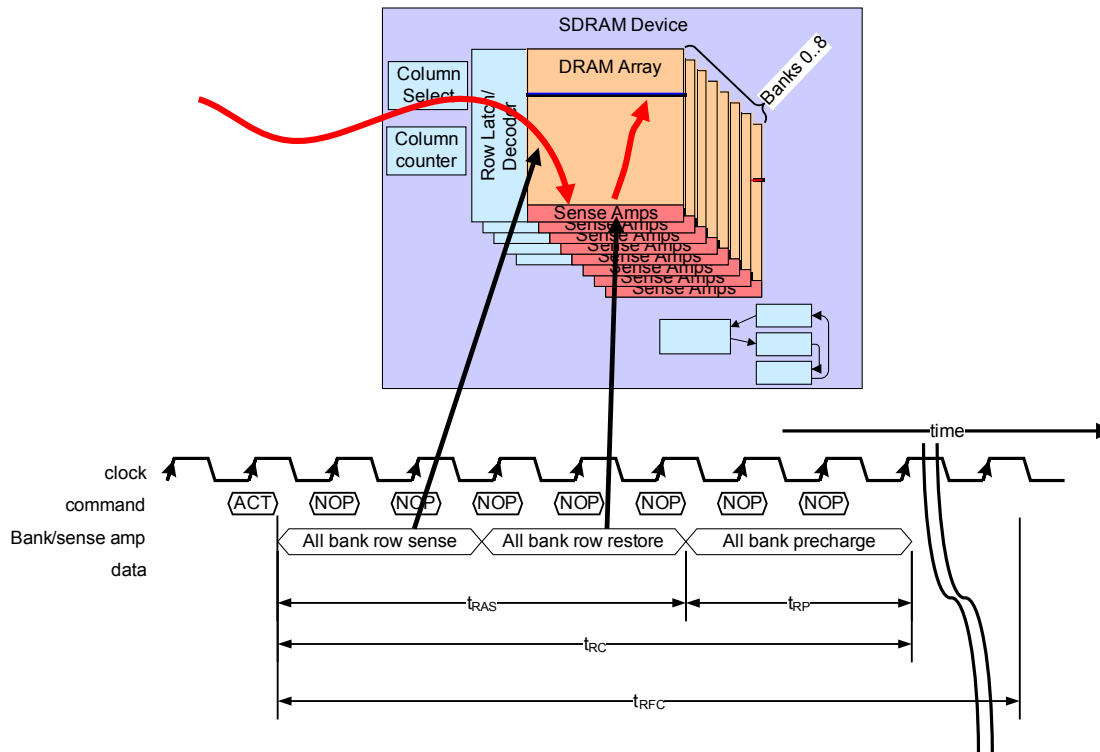


Figure 4.6: The stages of a refresh command

A refresh command is quite similar to an activate command followed by a precharge command, but with the exception that it performs this sequence for all the banks in a rank. DDRx DRAMs have an internal counter that chooses the row that will be refreshed, so the memory controller does not need to keep track of which row each rank it should refresh, it needs only to send the refresh command and wait.

Figure 4.6 shows the stages of the refresh command, which are just like an activate immediately followed by a precharge. Ordinarily, this would finish in t_{RC} , just as an activate and precharge would, but a refresh takes longer due to the greater number of banks involved. Because there may be 8 or 16 banks precharged, the DRAM has used a relatively large amount of current. Although it may seem that the resources of the DRAM are free after t_{RC} has elapsed, this is not the case. In order to ensure that this does not happen so often as to cause problems by drawing too much current, the parameter t_{RFC} is introduced to

prevent refresh commands from being issued too frequently. This parameter states that from when a refresh command is issued, another refresh or activation may not be issued to this rank until at least t_{RFC} has elapsed. t_{RFC} is always at least as long as t_{RC} , but is usually much longer. In a typical DDR3 DRAM, the minimum value of t_{RFC} is 110ns, while t_{RC} is only 52.5ns.

The memory controller must calculate when it should issue refresh commands in order to meet all timing requirements. They may not be issued faster than t_{RFC} , but not less often than it would take to issue N per t_{REFI} , where N is the number of rows. So if t_{REFI} is 64ms and there are 8192 rows, refresh commands should be issued approximately every 7.8 μ s. If a memory controller waits too long to issue a refresh command, it runs the risk of losing data. At the same time, a refresh command typically precludes the use of a rank for over a hundred cycles, causing all pending commands to that rank to stall. Additionally, all banks in a rank must be precharged so that they will be able to sense the data in the first part of the refresh command, so the memory controller must spend many cycles preparing the rank and waiting for the refresh to finish, all the while not hurting performance.

Because refresh commands can become a performance bottleneck to heavily-loaded systems, some memory controllers use alternative schemes to achieve the same effect. A memory controller can keep track of which row should be refreshed and send the activate and precharge commands individually so that the entire rank is not taken offline. This increases traffic on the command bus and adds complexity to the memory controller design, but it reduces the loading effect that normal refresh commands can have.

4.2 Read Cycle

A read cycle, at least in the context of a close-page system, consists of opening a row, reading or writing to it and then closing it. This is shown in Figure 4.7 as an activate command to open the row, a read command with a burst length of 4 to acquire data and a precharge command to close the row. This shows the case where only a single read is performed before closing the row. However, as it is possible and highly desirable to perform as many reads and/or writes to an open row as possible, this scenario is shown later.

Just as before, the activate command tells the precharged sense amplifiers to activate the wordlines and drive the bitlines to sense the data. Once the data is sensed, after t_{RCD} , the read may proceed. The values in the bank are read and sent to the I/O gating even while the values are still being restored to the cells.

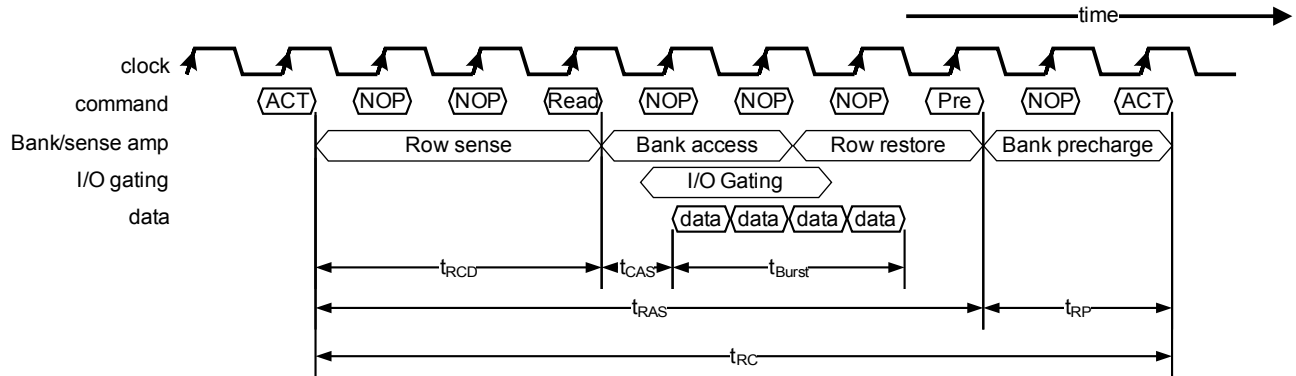


Figure 4.7: A read cycle in a close-page system

Once the read command is sent, the data goes to the output registers and is ready to stream back to the memory controller. After t_{CAS} , the data begins to be sent and is done after t_{Burst} . By this time, the restoration of data back to the DRAM cells is nearly done and once t_{RAS} has finished, the data is secure and the sense amplifiers may begin to precharge again. A final row activate, which begins another read cycle, follows t_{RP} afterward. The minimum time from an activate to a precharge is shown as t_{RAS} and the sum of t_{RAS} and t_{RP} is t_{RC} , the minimum row cycle time.

This scenario is common for applications that do not have a lot of locality in their address values. This means that the locations are not similar enough in location to be mapped to the same row, so each read or write is very likely to be translated to a different row, thus necessitating switching of rows and only a single use of an open row. It is much more efficient to reuse an open row for at least a few reads or writes, both for power and performance reasons.

4.2.1 Read Cycle with Read-and-Precharge

There are more complex commands available to a memory controller to automate some of the more common tasks. The read (or write) and precharge command combines the read and precharge commands into

one and reduces what was two commands into one. Because this command removes the need to have two separate commands sent to the DRAMs, the command bus is now free to send other commands at the time when there was previously a precharge command being sent.

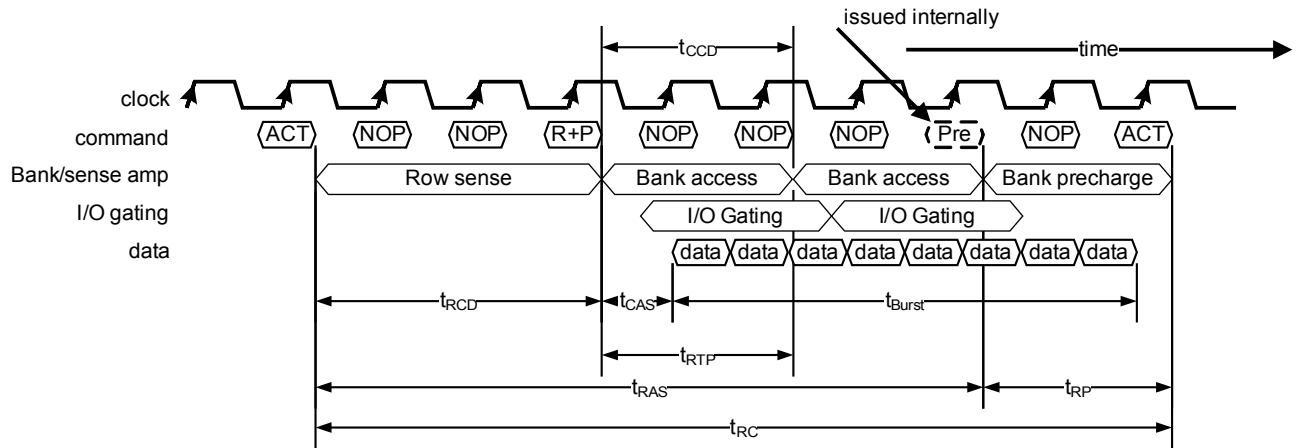


Figure 4.8: A read cycle using a read and precharge command

In Figure 4.8, a read cycle is described that uses a read and precharge command instead of two separate commands. The minimum spacing between the read and precharge command (issued internally) is $t_{RTP} + t_{Burst} - t_{CCD}$. Not only does this reduce the command bus bandwidth, it also reduces the complexity of the memory controller. If every read or write operation is followed by a precharge, as is used in a close-page memory system, then the memory controller does not need to keep track of when a precharge command should be issued. The DRAMs themselves know internally when a precharge should be issued and will not violate timing constraints imposed by the activate or read commands issued immediately before. In fact, the memory controller does not need to keep track of the precharge at all, so it takes fewer resources to keep track of what is going on in a given rank.

This sort of optimization is useful for embedded systems, where the power requirements for a memory controller are limited, so the complexity must be limited as well. If every read or write can be translated into exactly two commands, the activate and the read or write and precharge, then it becomes very simple to provide a basic functional interface to the DRAMs. High-performance memory systems can also take advantage of this. On systems with very little locality, such as those running many processes at a time

like a processing node, it is advantageous to assume each row will not be reused and simply close it. By combining these commands, the memory controller can now finish with a transaction quickly and have more available command bus bandwidth for other operations.

4.2.2 Posted CAS

Another optimization for DDR2/3 DRAMs was the inclusion of additive latency, AL or t_{AL} , to allow posted CAS commands. What this means is that a read or write command can be sent to a DRAM before it is able to be executed, as per timing specifications. The DRAM device will hold this command and delay it a specified amount of time, t_{AL} , and then execute it as though it had just received it. This can be used for regular read and write commands or for read and write-and-precharge as well.

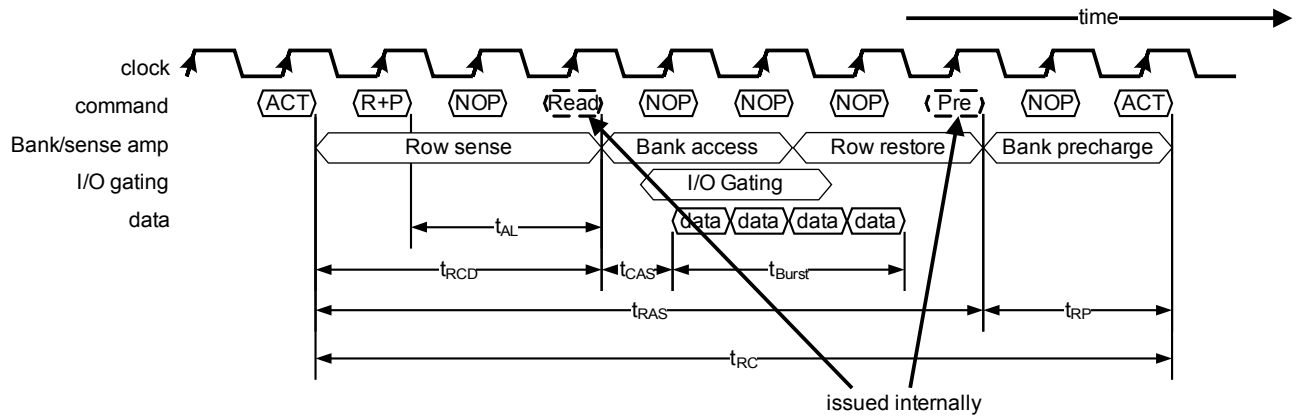


Figure 4.9: A posted CAS+Precharge command

The value of t_{AL} may be programmed using the mode registers. In a DDR3 device, t_{AL} may only be programmed relative to t_{CAS} (t_{CL}). This allows the memory controller to choose between $t_{CAS} - 1$ and $t_{CAS} - 2$, depending which is desired. This allows the memory controller to decide whether it wants to send the read or write immediately after the activate or to wait a cycle first. A DDR2 device, however, allows arbitrary values to be set, so the memory controller may choose any value, up to t_{CAS} .

It is important to have posted CAS available because it simplifies memory controller design. A very simple controller in an embedded system may simply translate each read or write into an activate and a read-and-precharge or write-and-precharge. The commands may be sent one after the other and the controller need

not keep track of when to send the read or precharge commands, as long as the mode registers are programmed correctly. In much the same way, a server has more flexibility in scheduling requests by being able to issue the reads and writes sooner and then move on to service another request. The posted CAS command is commonly used to simplify designs and better group row and column commands.

4.3 Command Interactions

Of key importance in achieving high performance memory systems is correctly scheduling the commands so as to reduce their latencies. In order to efficiently schedule commands, it is important to know what the implications are for a given command ordering. This section will look at a series of command combinations and the relevant timing parameters that they must meet to ensure correct operation. It will discuss how close page and open page row buffer management policies play a role in choosing commands and what timings must be observed for them.

An open page row buffer management policy takes advantage of the fact that once a row has been opened, the timing penalty for additional reads and writes is reduced until the row is closed. Thus, an open page policy attempts to perform reads and writes from an open row and leaves the row open with the expectation that it is likely that another access to the same row will be forthcoming and its latency will be relatively shorter. This is a fairly likely scenario in an embedded system or workstation where there are relatively few processes. Because it is likely that there are only a few processes running, their data is likely all grouped in the same segment of memory and the workload will exhibit a high degree of spatial and temporal locality. With many requests reading and writing to similar memory locations, it becomes relatively likely that there will be multiple memory requests to the same row and there will be a good amount of row reuse.

The downside to an open page policy is that the memory controller must keep track of open rows and know to close a row and open a new row when a request to a different row come along. This incurs an additional latency penalty to precharge the row, plus the requisite activation and read delays. In systems that

do not expect much row reuse may use a close page row buffer management policy. Unlike an open page policy, the basic close page policy assumes no row reuse so it automatically closes a row after the first read or write to reduce the timing penalty for switching rows.

This section, however, will examine mostly open page based command interactions as there are more possible interactions than for a close page system by far.

4.3.1 Consecutive Reads To Different Rows In A Bank

DRAM devices hold an active row in their sense amplifiers until a precharge command is issued, which is helpful for a memory controller that would like to perform multiple combinations of reads and writes on that particular row. Because a read or write transaction to the memory controller is finished with the read or write in the corresponding command sequence is completed, being able to complete the read or write without having to perform a precharge or row activation is quite advantageous. In fact, because the latency is reduced for an open row hit, the penalty for a row miss is increased. Instead of merely activating the row, a full row cycle time penalty is incurred.

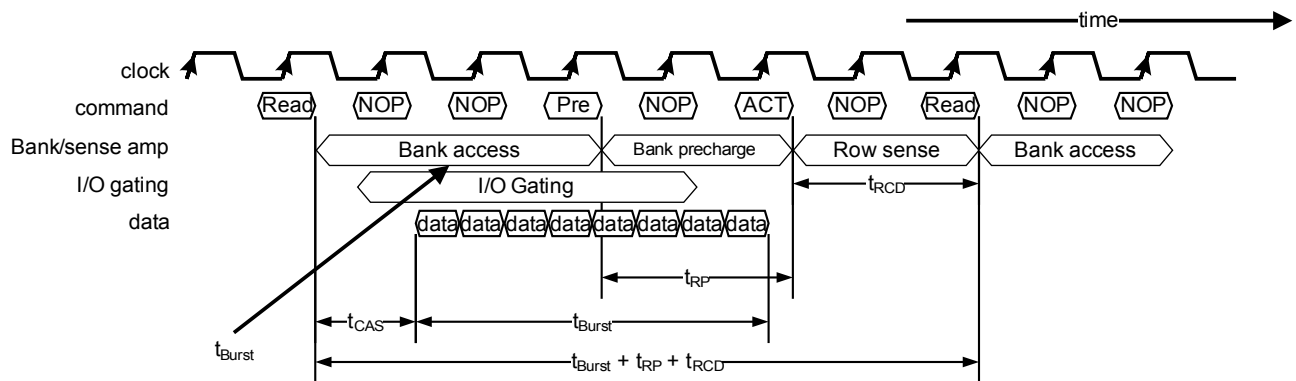


Figure 4.10: Timing of a read to a read command on a different row with rows already restored

Figure 4.10 shows the timing of two consecutive read commands to different rows within a bank. This is the better case because the row restore has already happened. The system must wait for t_{Burst} , which the bank access takes to move the necessary bits to the I/O drivers, then for the sense amplifiers to precharge and then for the row-to-column delay. This therefore means that if there is a bank conflict, when two

consecutive reads go to different banks, the minimum available spacing between the issuance of the read commands would be $t_{Burst} + t_{RP} + t_{RCD}$. Note that in DDR2 systems, the burst of 8 is actually 2 bursts of 4, so once the first burst is done then the precharge may proceed. This means that when using a DDR2 system, the minimum spacing caused by the bank access portion is reduced to $t_{Burst} / 2$.

On the other hand, if, following the row activation, the row sense has completed but the row restore has not, then the minimum spacing for two read commands will be greater.

4.3.2 Consecutive Reads To Different Rows In A Bank, Worst Case

If the row restore has not finished by the time the sense amps are no longer needed to stream data to the output buffers, the precharge command must wait.

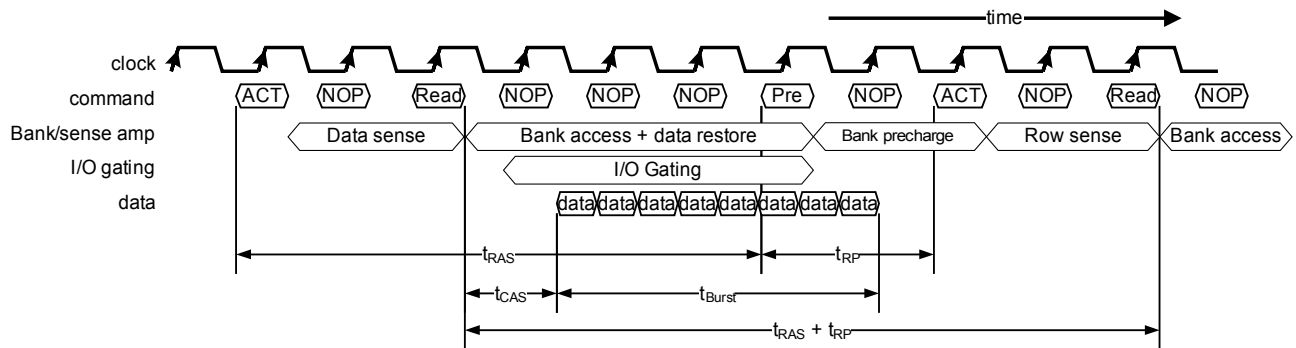


Figure 4.11: Consecutive reads that are delayed by waiting for data restoration to finish

Because the row activation occurs very shortly before the first read, t_{RAS} must elapse before the row can be precharged again. Figure 4.11 differs from Figure 4.10 because its activation was assumed to have happened long enough before the first read so that the row sense and restore had already passed, whereas Figure 4.11 has a closely-grouped activation and read sequence. Thus, in the worst case the reads are spaced by $t_{RAS} + t_{RP}$, rather than $t_{RP} + t_{RCD} + t_{Burst}$.

When the scheduler in a memory controller has only a few commands and both are going to the same bank, Figure 4.11 will often be the timing scenario. However, if the memory controller has more commands to choose from, it will be able to send activates to several rows and then return to send read commands to those same rows. If it is done right, some banks can be sending data while others are performing data sense

and restore. This ability to intelligently overlap operations can improve performance greatly and will be explored in some detail in the experimental setup section.

4.3.3 Reads to Different Banks (Bank Conflict)

When the memory controller has two consecutive reads that must go to different banks, the first bank may not have a conflict requiring a precharge before the activate and read commands, but the second might. When this happens, the memory controller can issue the read request to the first bank immediately, but must then precharge and activate the second bank before finally reading from it. When this happens, the memory controller has only a few options as to how it can optimize the scheduling of the commands.

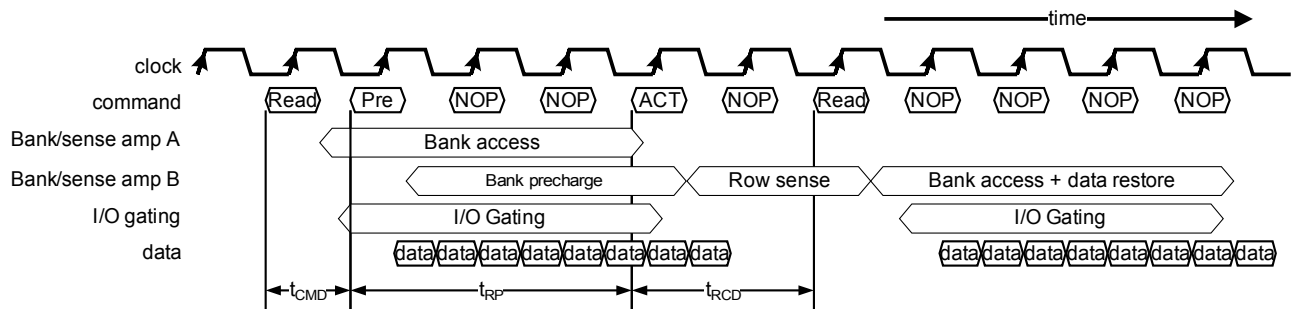


Figure 4.12: Consecutive reads to different banks without reordering and a conflict on the second bank

In Figure 4.12, the memory controller first sends the command to read from bank A and receives the data after t_{CAS} . Then the memory controller moves onto the read in bank B. Because this bank has its own set of sense amplifiers, it can begin to precharge them in preparation for a row activation. This step is necessary because there is a conflict between the last read or write to this bank. If the previous read or write had been to the same row, the read to bank B would have been delayed only to avoid contention on the data bus.

Because of the bank conflict on bank B, the second read request must be delayed $t_{CMD} + t_{RP} + t_{RCD}$ after the first. The timing parameter t_{CMD} is the minimum spacing allowed between two commands as the command bus can receive a command only every t_{CMD} . After this, the read command must wait for bank B to precharge (t_{RP}) and then sense the new row (t_{RCD}). Note that in Figure 4.12, the timing between the commands is measured from the start of a command to the start of a following command. This differs from the time that

the resources inside the DRAMs are actually in use but is equivalent. Measuring from the end of commands is the same as measuring from the beginning of commands, except that measuring from the end of commands will coincide with the beginning of when a particular resource is in use in the DRAM.

If the memory controller is a more complex design and is able to reorder the commands, the spacing between the read commands can be decreased. Because there is no contention for the sense amplifiers, each bank has its own, the command and data buses are the shared resources that must be considered.

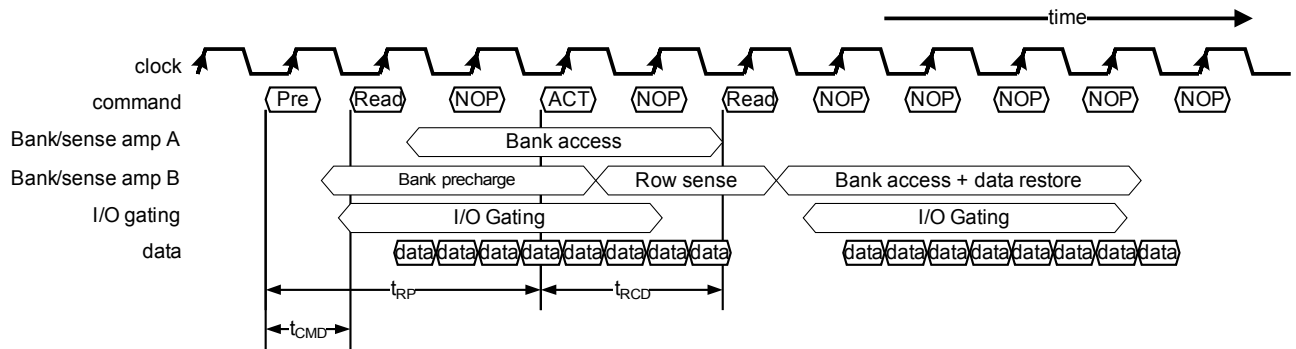


Figure 4.13: Consecutive reads to different banks, bank conflict and command reordering

By reordering the precharge to the second bank before the read to the first bank, the second bank can precharge while the first is moving data out of the sense amplifiers and into the output buffers. With these two commands switched around, the time between the two read commands is reduced to $t_{RP} - t_{CMD} + t_{RCD}$. This is $2 * t_{CMD}$ shorter than before and improves the efficiency of the system. This reordering causes the initial byte from the DRAM to be received t_{CMD} later than before. However, the initial byte from the second read is received t_{CMD} earlier, so the average latency is the same.

The major benefit to this rearrangement is that the two data bursts are now closer in time on the data bus. This means that the average utilization of the data bus is increased and efficiency is improved. Ideally, the data bus would be in use at all times and memory system would be at its theoretical maximum. However, the reality is that scheduling concerns cause gaps in use of the data bus that limit its efficiency. Being able to move data bursts closer together still helps. If these two read commands are closer together, so are their data

bursts and therefore they tend to occupy the bus for fewer cycles, leaving more cycles available for other read or write commands.

4.3.4 Consecutive Reads to an Open Row

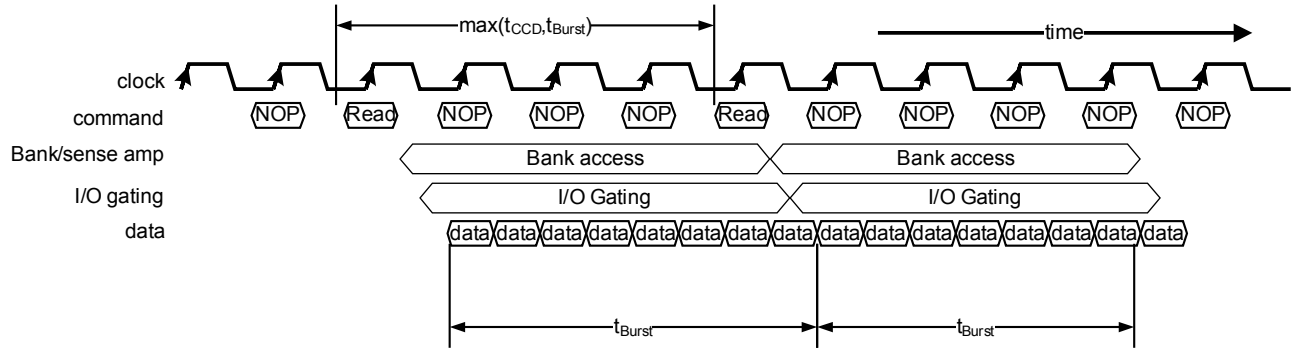


Figure 4.14: A read following a read to the same open row

When the memory controller is able to schedule to reads back-to-back, the data bus can be utilized fully. As shown in Figure 4.14, the delay between the two read commands is only t_{Burst} . No new row need be opened, nor does any sense amplifier need to be precharged, simply read the rows as often as possible. Even though this is unlikely, most access patterns will not have more than just a few consecutive reads to an open row, it cannot continue indefinitely. Because the rows must eventually be refreshed to maintain data integrity, there is a limit as to how long one row can be opened.

There is also the matter whether to wait t_{Burst} or t_{CCD} before issuing the second read command. In DDR2 memory, for example, the burst length can be 8 beats, while the internal burst length is 4. In order to send 8 bytes of data, there are two bursts, so simply waiting for the first internal burst to finish would not be enough to finish the burst of 8. So if the burst length is longer than the internal burst length (t_{CCD}), then the memory controller must wait for the entire burst to finish.

On the other hand, if the internal burst is 8 beats, as it is in DDR3 memory, then it is possible for the internal burst to be longer than the actual burst. This is known as burst chop. If the mode registers are set to send a burst of 4 or the read command asks for only 4 byte of memory, then the DRAMs send only 4 bytes.

However, the internal burst continues for the full 8 beats, so the memory controller would have to wait for this to finish before issuing any more read commands.

4.3.5 Consecutive Reads to Open Rows within a Rank

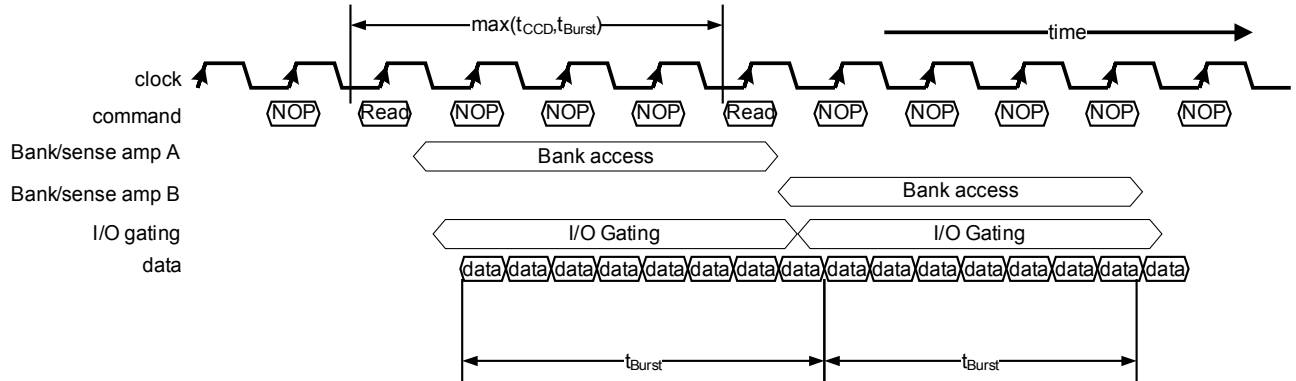


Figure 4.15: A read following a read to open banks within a rank

Very similar to two reads within a row is two reads in two open rows. As seen in Figure 4.15, the same timing constraints apply. Because the same I/O gating is used, the consecutive data bursts can be sent without any delay between them. Again, the data bus is the only shared resource that must not be used by more than one command at a time. Had these been writes, a similar timing constraint would be imposed. Because the memory controller was talking to one rank at a time, the minimum spacing would be $\max(t_{CCD}, t_{Burst})$. As shown later, when there are multiple ranks sending data consecutively, the spacing of the data burst cannot be as efficient.

4.3.6 Reads to Different Ranks

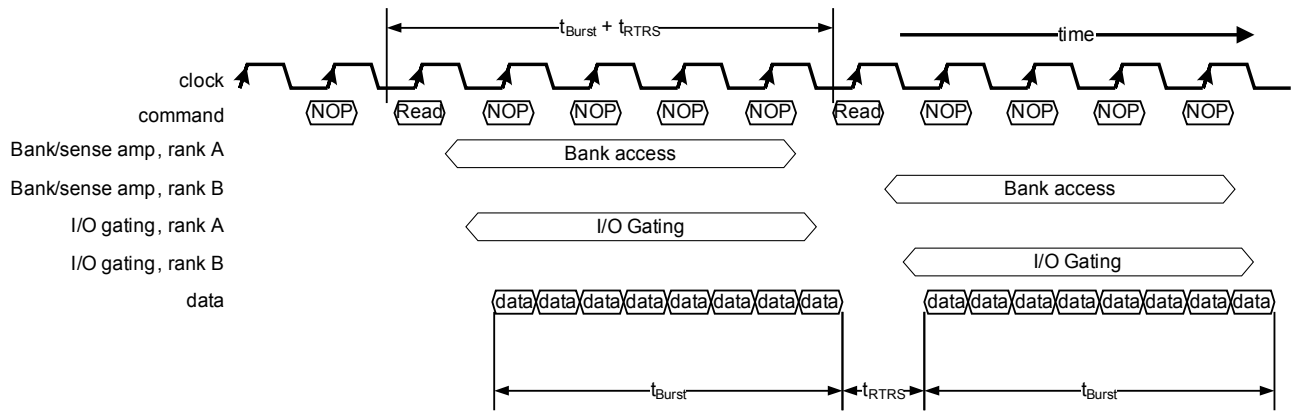


Figure 4.16: Consecutive reads to different ranks with open rows

When different banks are active and not constrained by timing requirements, as in Figure 4.15, the requests may be scheduled in such a way as to pipeline the requests and keep the data bus in use perpetually. This is, however, not true for reads that alternate between ranks. When the memory controller tells one rank to send data and then another to do the same, there must be some period of down time on the data bus to allow the new rank to synchronize before sending the data. This is illustrated in Figure 4.16 by the parameter t_{RTRS} , short for Rank To Rank Switching time. This often refers to the data strobe synchronization time and will vary based on the design of the memory system more than any internal aspect of the DRAMs. In this example, the memory controller must delay the second read by one cycle in order to allow rank A to release the data strobe and bus and allow rank B to take over. Although this is shown as one cycle, different system implementations will vary depending on implementation.

On older or slower memory systems, because the data bus runs at slower speeds, the value of t_{RTRS} may be zero. If the ranks can successfully switch from one rank to the next in a very short time, t_{RTRS} is not needed. However, in all DDRx systems, all of the ranks use the same data strobe to signal that a byte of data has been transferred. Because one rank must release this line, go into the high impedance state and another rank must leave high impedance and begin to drive the data strobe, the value t_{RTRS} cannot be zero.

4.3.7 Write to Write, Different Ranks with Open Banks

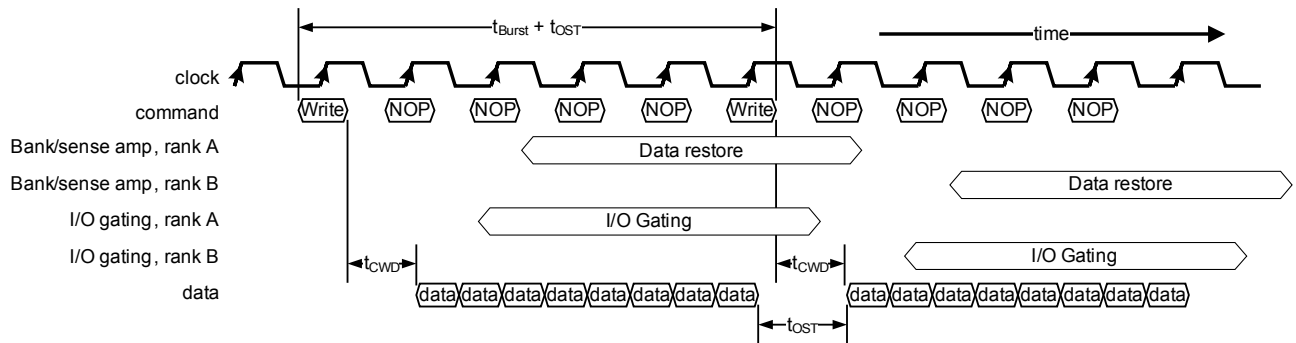


Figure 4.17: Consecutive writes to different ranks must wait for on-die termination to setup

The difficulty in scheduling read requests to different ranks arises because different devices must connect to the data bus without overlap. Thus, when one rank is done sending data, it must disconnect from the bus and then allow the next rank to connect. However, with writes this should not be a problem. Because the memory controller is the only device sending data, it can theoretically be the only device that connects to the bus, so there is no time allocated for connecting and disconnecting.

Indeed, this is the case for SDRAM and DDR SDRAM. The memory controller can schedule write requests back-to-back without any problems. In the case of DDR2/3 SDRAM, this no longer holds true. Because data rates continue to increase with each successive generation, DDR2/3 are significantly faster and data integrity over the data bus has become more difficult to achieve. The multi-drop bus architecture makes each device connected to the bus significant as far as signaling quality goes. In attempt to improve the signal quality, each DRAM device has an internal, programmable resistor network that is connected to the bus whenever other ranks are receiving writes or sending reads. This provides better characteristics than simply putting each set of I/O gating into a high impedance state. It is known as “on-die termination” (ODT) and reduces reflections and improves signal quality on the data bus. As a consequence, the memory controller must wait for the ODT to turn off for the next rank and turn on for the previous rank. In DDR2 DRAMs, turning on takes 2 cycles and turning off takes 2.5 cycles.

The net result is that consecutive writes in DDR2/3 systems must have a gap between them, as shown in Figure 4.17. The timing parameter, t_{OST} (on-die termination switching time), represents the time that

it takes all ranks that are not being written to to enable their ODT and the time for the destination rank to disable ODT. The same is true for reads, every rank not sending data should enable ODT to improve the signal quality of the data being sent from the DRAM. A system designer should keep this in mind when deciding how long to delay reads and writes in a DDR2/3 system. Reads are limited by t_{OST} and t_{RTRS} . Most of the time, however, t_{RTRS} will be longer than t_{OST} , so a system designer can limit reads to $t_{Burst} + t_{RTRS}$ and writes to $t_{Burst} + t_{OST}$.

4.3.8 Write to Precharge

Figure 4.18 shows a write command followed by a precharge command. The timing is quite similar to the read to precharge timing in that there is a delay after the command is issued, in this case it is t_{CWD} . Then

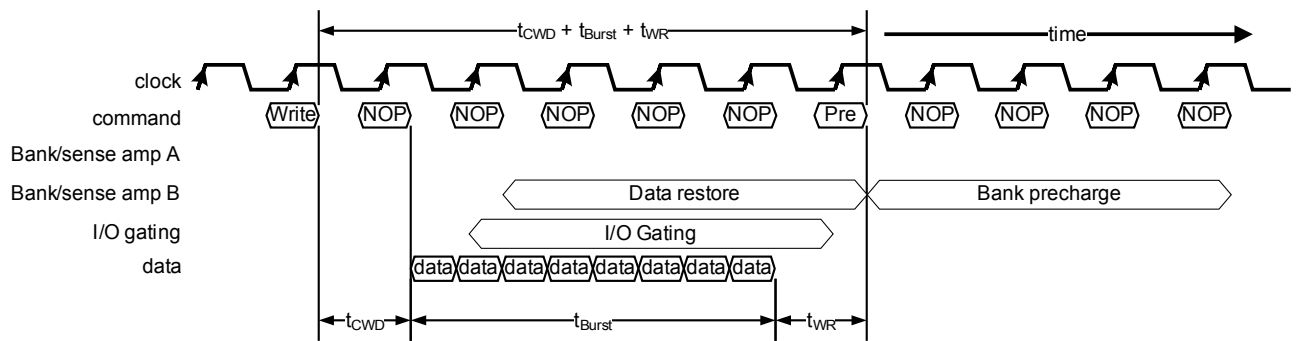


Figure 4.18: A write followed by a precharge

there is a data burst, t_{Burst} and finally there is t_{WR} . This parameter, the write recovery time, determines how long after a data burst is received that the row may be precharged. If it is precharged too soon then the values that were sent to the DRAM may be wiped out by the precharging before they are able to be written to the cells. The timing parameter t_{WR} ensures that the data is restored to the cells, but this is longer than when a read command follows a write command. When there is a subsequent read command, then it must only wait for the data to be correctly read and stored by the sense amplifiers. If the data is not yet restored to the cells then it does not matter, because the read command is only concerned with the values in the sense amplifiers and will not affect any data being restored to the storage cells.

4.3.9 Write to Write with Bank Conflict

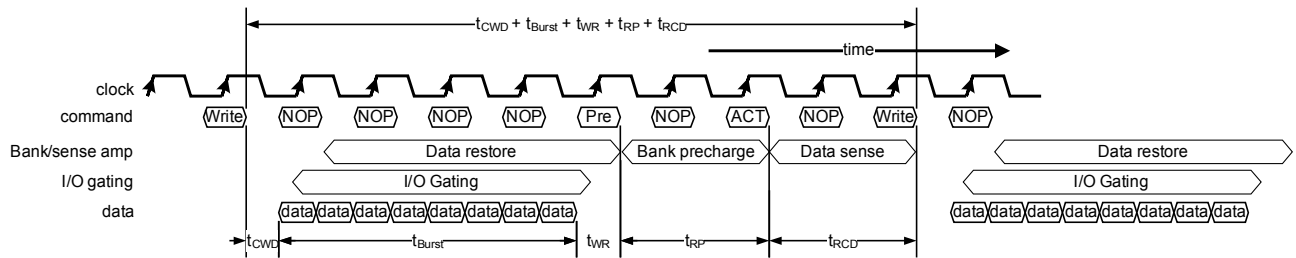


Figure 4.19: Consecutive writes to different banks with a bank conflict

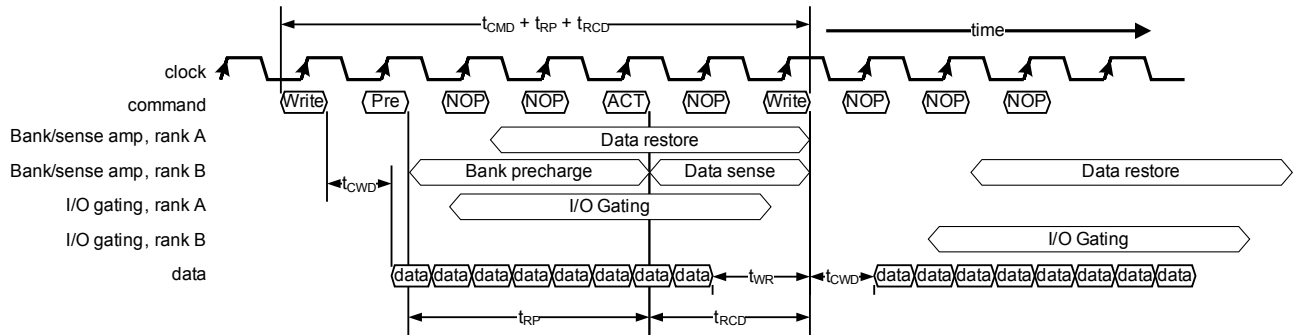


Figure 4.20: Consecutive writes to different ranks with a bank conflict

When two writes follow one another to the same bank, but are on different rows, the delay is similar to consecutive reads to a bank with a bank conflict. As shown in Figure 4.19, the precharge must respect t_{WR} , so the bank cannot be precharged until the data is stored back into the cells. Additionally, not shown in Figure 4.19 is the fact that the precharge must also respect t_{RAS} in addition to $t_{CWD} + t_{Burst} + t_{WR}$. It may happen that t_{RAS} is longer than the others and the precharge command will be delayed. In this example, it is assumed that t_{RAS} has already been satisfied and it need not be considered. Therefore, the minimum distance between writes must be the column write delay, the burst time, the write recovery time, the precharge to activate time and the row-to-column delay. This is shown as $t_{CWD} + t_{Burst} + t_{WR} + t_{RP} + t_{RCD}$.

If the writes are to separate ranks, assuming again that t_{RAS} has been satisfied, then the precharge to the second rank may proceed as soon as the command bus is available. From there, the precharge must finish and the new row is opened. This is quite similar to having just a bank conflict and having to switch rows to be able to write to a new row. Because the data burst is shorter than the precharge and data sense times, the data bus is never in demand by both at the same time. Therefore, the memory controller must wait $t_{CMD} + t_{RP}$

+ t_{RCD} before issuing the next write command. This is just t_{CMD} longer than a regular bank conflict while waiting for the command bus to become free.

4.3.10 Read to Write, No Conflict, Different Ranks

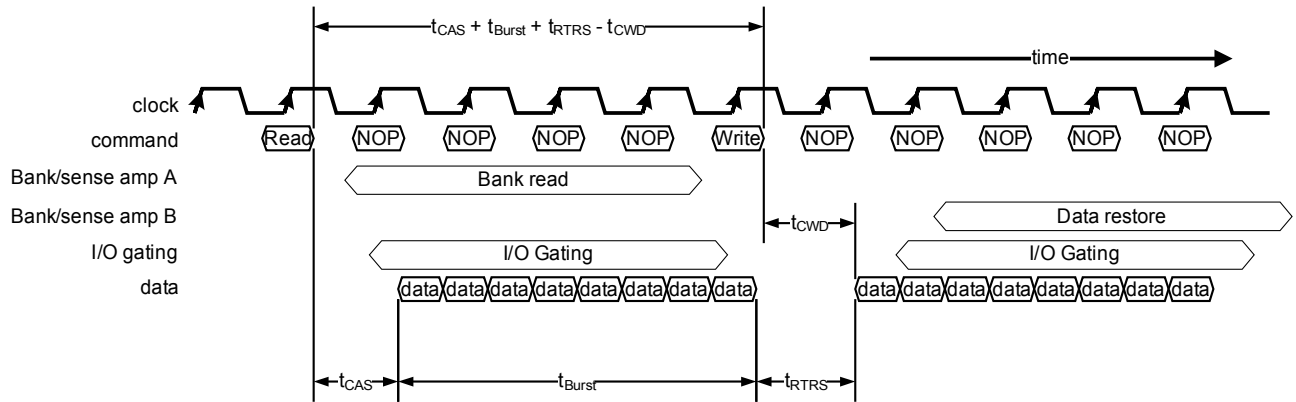


Figure 4.21: A write request following a read request to open banks

The situation of a write request following a read request is similar to consecutive writes, as shown in Figure 4.17. Because the commands are going to different ranks, there is no concern about the I/O gating, sense amplifiers or internal data movement. The only restrictions are on the command and data buses. Because the DRAMs are sending data on the data bus for the read and the memory controller is sending the data for the write, the data bursts must be separated by t_{RTRS} . This allows the devices to properly synchronize and be set for the upcoming transfer. The memory controller can, however, take advantage of the fact that there is a delay before sending data on a write, t_{CWD} . Although the memory controller must not send data until $t_{CAS} + t_{Burst} + t_{RTRS}$, the write command may be sent slightly before this. So the best case timing, assuming open banks, is $t_{CAS} + t_{Burst} + t_{RTRS} - t_{CWD}$. In a DDR system, t_{RTRS} and t_{CWD} are both 1 cycle, so the timing is effectively $t_{CAS} + t_{Burst}$. Likewise, in a SDRAM system, t_{RTRS} and t_{CWD} are both 0, so the effective timing is also $t_{CAS} + t_{Burst}$.

4.3.11 Read to Write with Bank Conflict

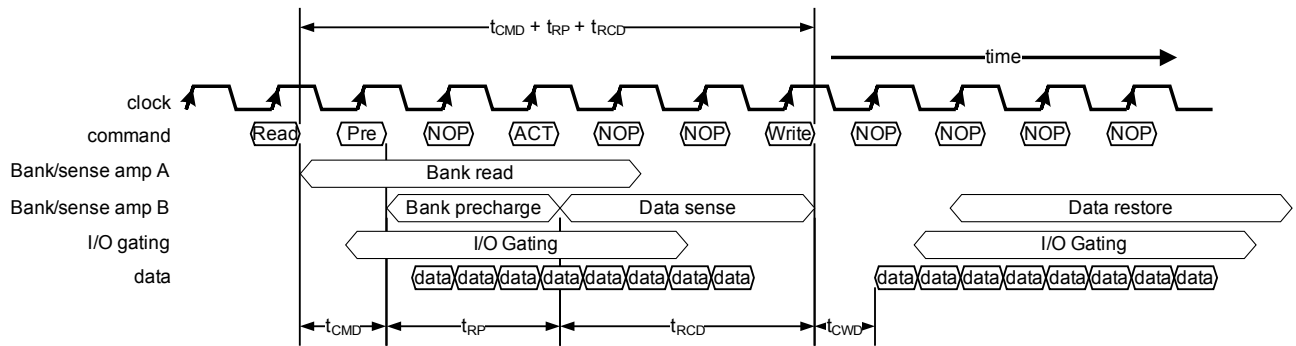


Figure 4.22: A write following a read with a bank conflict

In Figure 4.22, a write following a read is shown. Because the row that the write request must go to is not already open, there is a bank conflict and the correct row must be opened before the write may proceed. Because these two commands are to different banks, they have different sense amps and the precharge may begin immediately after the read command is issued. From this point, it is a matter of waiting for the precharge to complete and the data to be sensed before sending the write command, waiting t_{CWD} and sending the data.

Note that this is the best-case scenario. If t_{RAS} has not already elapsed in the bank requiring the precharge, the memory controller must first wait for the t_{RAS} period to pass and then begin the precharge. However, because in the example t_{RAS} has already elapsed, the precharge can begin immediately and the minimum scheduling distance between the read and the write is $t_{CMD} + t_{RP} + t_{RCD}$. This delay time can be further reduced if the memory controller supports command interleaving, the ability to overlap constituent commands that comprise a transaction. If this is possible, then it is reasonable to begin the precharge before the read command to begin to prepare the second row earlier. This will delay the read command by t_{CMD} , but will also move the write command up by t_{CMD} . This will move the read and write data bursts closer together in time which is an improvement in data bus utilization. If data bursts can be moved closer together, then the overall utilization of the data bus can be increased over time. This will lead to a net increase in effective data bus bandwidth.

4.3.12 Write to Read in the Same Rank

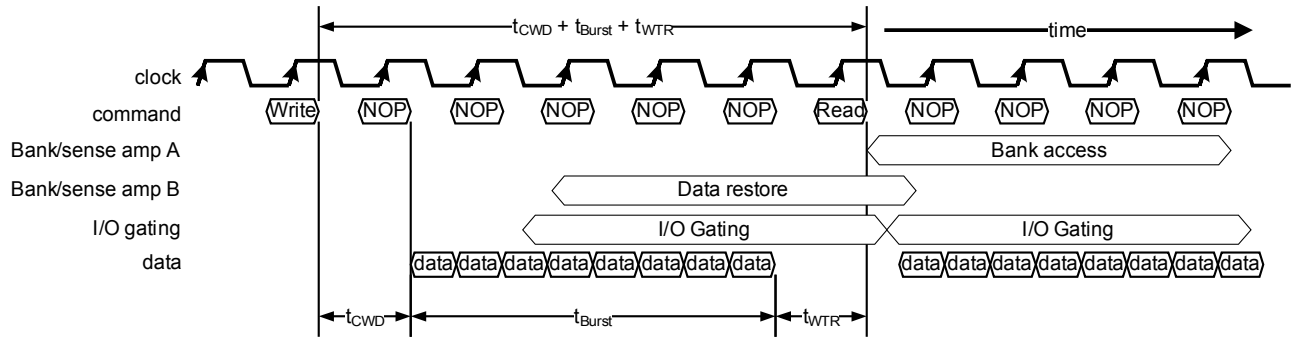


Figure 4.23: A read following a write to the same rank, different, open banks

In Figure 4.23, a read following a write is illustrated. Both commands are to the same rank, so the I/O drivers are shared, but the requests are to different, but open, banks. This is different from the case of consecutive reads or consecutive writes because the direction of the data movement differs. Because the write happens first, the data bus is first utilized, then the I/O drivers and lastly the data is restored to the DRAM cells. In the read command, the data is loaded into the I/O drivers and then sent over the data bus. The common element here is the I/O drivers. The fact that these drivers cannot be in use for two different operations simultaneously creates the delay that is shown in Figure 4.23.

Beginning with the write command, the data burst must wait for t_{CWD} and then t_{Burst} . After this, a new timing parameter, t_{wtr} , is introduced. This is the write to read timing, or the time from the end of a data burst until a read may be performed. It is similar to t_{WR} in the fact that it measures from the end of the data burst, but t_{WR} is the time until the data restore finishes and t_{wtr} is the time until the I/O drivers are free again. This means that the minimum write-to-read timing for open rows within a rank is $t_{CWD} + t_{Burst} + t_{wtr}$.

4.3.13 Write to Read in Different Ranks

In a slightly different case from that depicted in Figure 4.23, Figure 4.24 shows the case of a read following a write to different banks. In this case, the I/O gating is completely separate, as are the banks, so there is no concern about their utilization. The rows that must be accessed are already open, so there is no need to precharge and activate the row. The only shared resources are the data and command buses.

Therefore, the memory controller may issue the write, wait t_{CWD} , send the data burst, wait t_{RTRS} for the second rank to take control of the bus and expect to receive valid data at this point.

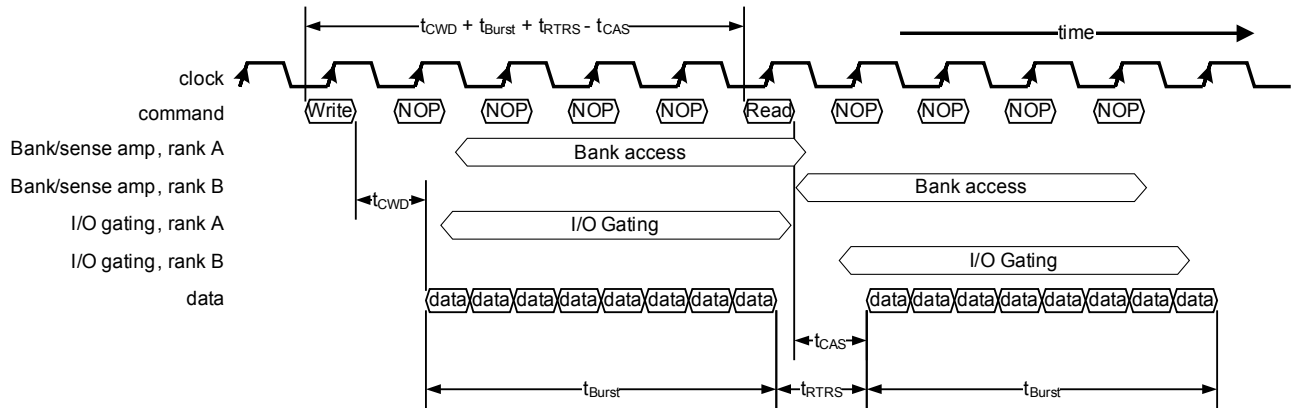


Figure 4.24: A read following a write in different ranks

The spacing between these commands is $t_{CWD} + t_{Burst} + t_{RTRS} - t_{CAS}$. The reason t_{CAS} is a negative value in this expression is that the rank-to-rank switching time can be overlapped with the delay waiting for the data from the second rank. So the read command may be issued before t_{RTRS} is elapsed, possibly even before t_{Burst} has finished. Because the data bus is the limited resource, the read command may lead the earliest available utilization by t_{CAS} .

For example, in a DDR2 SDRAM memory system, t_{CWD} is $t_{CAS} - 1$ cycle, and t_{RTRS} is one cycle. So $t_{CWD} + t_{RTRS} - t_{CAS}$ cancels to zero. Therefore, the minimum scheduling distance is t_{Burst} , which makes scheduling of these types of commands easier for a DDR2 memory controller.

4.3.14 Write to Read with Bank Conflict, Same Bank

As shown in Figure 4.25, the minimum scheduling distance between a read following a write to the same bank but different rows is quite lengthy. This example assumes that this is the best case scenario, where t_{RAS} has already passed and is not a consideration. If t_{RAS} has not yet passed by the time the precharge should be activated, then the memory controller must wait to issue this command.

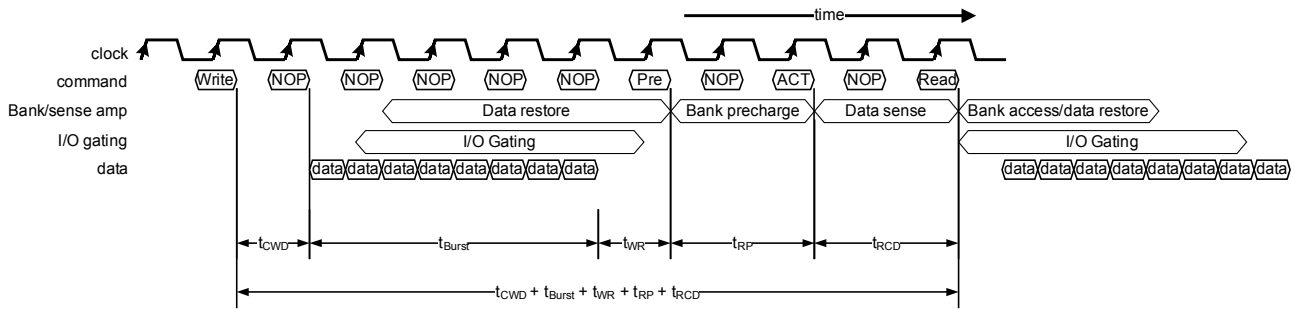


Figure 4.25: A read following a write to the same bank with a bank conflict

First the memory controller must wait to send the data, send the data and wait for the data to be successfully written back to the DRAMs before sending the precharge command. Then the precharge command must fully precharge the sense amplifiers before reading a new row. After this it must wait for the row-to-column delay to pass and then send the read command. After t_{CAS} has elapsed the data from the read will begin to return. The net delay from all of these processes is $t_{CWD} + t_{Burst} + t_{WR} + t_{RP} + t_{RCD}$.

4.3.15 Write to Read with Bank Conflict, Same Rank

In Figure 4.26, a read request follows a write request to different banks within the same rank. This example also assumes that t_{RAS} has already passed so there is no need to delay for it. The timing is quite

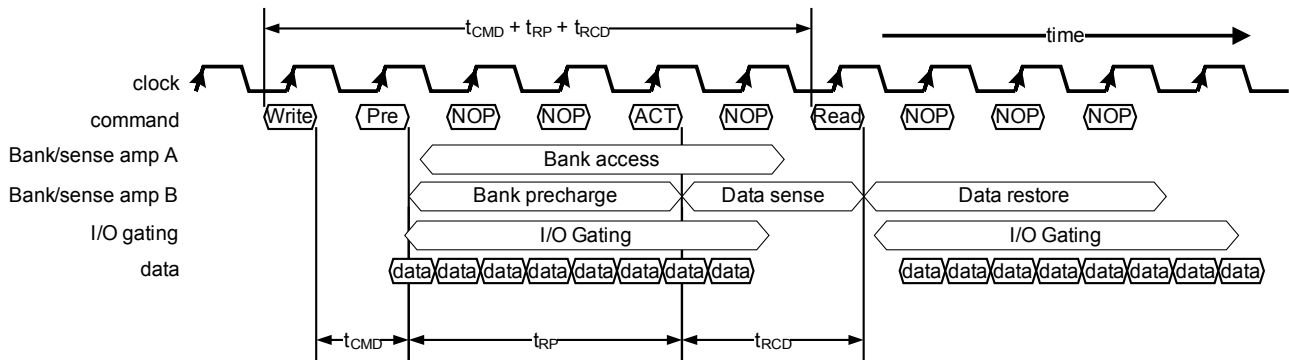


Figure 4.26: A write followed by a read to different bank with a bank conflict

similar to a simple bank conflict: the memory controller must wait for the the sense amplifiers to precharge and the row to be sensed before finally issuing the read command. An additional delay is added because the separate banks must share a common data bus, so the precharge command must wait t_{CMD} to be issued. The overall spacing of the commands is therefore $t_{CMD} + t_{RP} + t_{RCD}$. As with previous examples, by issuing the

precharge before the write command, the memory controller will delay the read command by t_{CMD} , but will also begin the process of switching the row for the read by t_{CMD} as well, and will reduce the spacing of the bursts on the data bus and achieve greater data bus utilization.

This illustration makes several assumptions that should be checked by a memory controller as well. First, it assumes that $t_{CWD} + t_{Burst} + t_{WR}$ is less than $t_{CMD} + t_{RP} + t_{RCD}$. If this is not true then the I/O driver will become in use by two commands at once and likely to cause an error. Therefore, the minimum spacing must be the larger of $t_{CMD} + t_{RP} + t_{RCD}$ and $t_{CWD} + t_{Burst} + t_{WR}$ in order to ensure that the I/O drivers are not a bottleneck.

4.3.16 Column Read-and-Precharge Timing

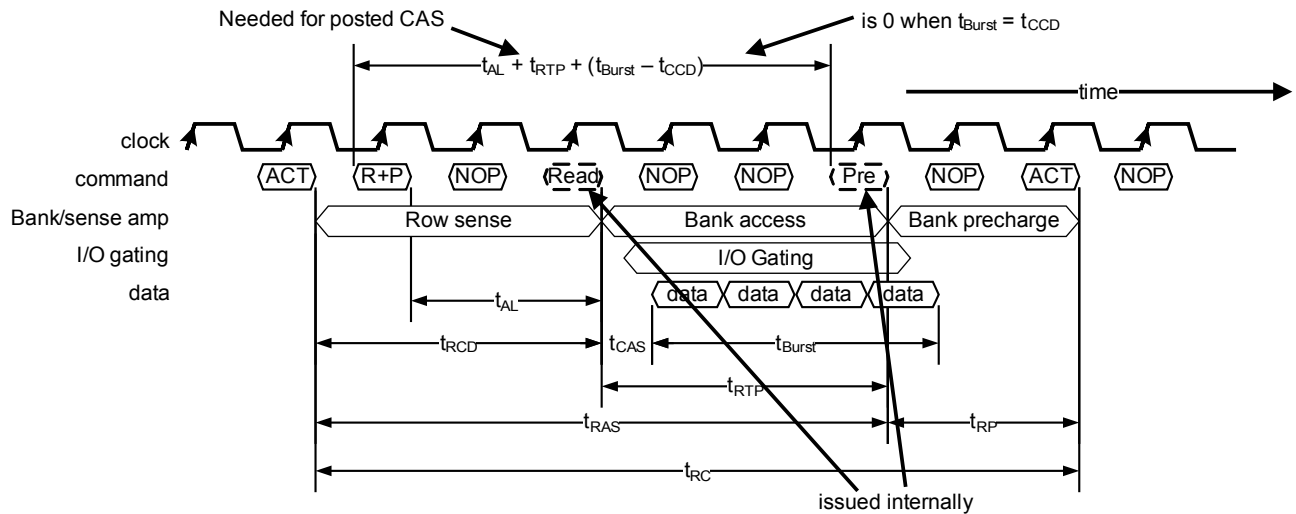


Figure 4.27: A read cycle using a read-and-precharge command with posted CAS enabled

A nice feature of modern DRAM devices is the ability to send complex commands that reduce the load on the command bus and reduce the number of timing parameters that a memory controller must keep track of. One of the most common such commands is the read-and-precharge command, which, along with additive latency, can reduce the memory controller interaction to a minimum. As shown in Figure 4.27, immediately after activating the row, the memory controller then issues a read-and-precharge command. Although t_{RCD} has not yet been satisfied, the DRAM has been programmed such that $t_{CMD} + t_{AL} = t_{RCD}$. The

DRAM internally queues up this read-and-precharge command to be issued later. Once the read-and-precharge command is internally issued, the precharge command is queued as well.

This precharge command will wait until t_{RAS} is satisfied, automatically, as this is a feature of DDR2/3 devices. This means that the memory controller must also keep track of t_{RAS} timing in order to know exactly when the DRAM device actually sends the precharge command internally.

4.3.17 Column Write-and-Precharge Timing

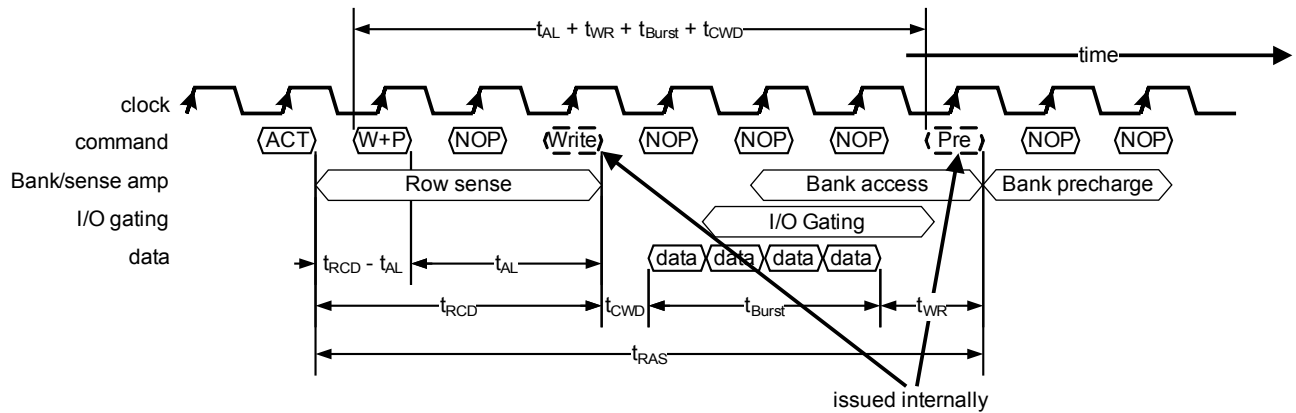


Figure 4.28: A write cycle using a write-and-precharge command with posted CAS enabled

Much like Figure 4.27, a write-and-precharge command immediately following an activation is described in Figure 4.28. This again utilizes additive latency to internally delay the write command. The DRAM will also make sure to respect t_{RAS} when issuing the precharge command internally. Usually t_{RAS} is defined to have enough time for the write to conclude, so the DRAM does not have to wait longer anyway. This means that activates followed immediately by writes will never run into timing problems due to t_{RAS} . This fact is often why DRAM devices are said to be write cycle limited. This means that the value of t_{RAS} is determined by how quickly the device can physically conclude an activate followed by a write.

Although it is easier on the memory controller to not have to keep the commands queued and to keep track of the activate to read and read to precharge timings, the memory controller must still know when the read and precharge are issued internally. This is because these commands still use the on-chip resources at the same time and the memory controller must not cause the DRAM to use resources that it does not have

available at that time. When later commands need to be issued, the memory controller must know when the precharge and reads happened in order to properly add delays.

4.4 Power and Performance Constraints

In addition to timing constraints that guarantee that certain elements of a DRAM device cannot be in use at the same time by different commands and finish correctly, there are timing constraints that attempt to limit the power dissipated by the devices and provide better signaling to each device. The following sections will provide a brief look at these constraints and attempt to explain why they were created. The power dissipation discussion will be somewhat brief as the following sections will look at power dissipation models in greater detail.

The main reason to impose timing constraints that limit how much a DRAM device is able to do in a certain time period is to limit the power that the device consumes. Power usage has become more important in a system's design than before. Because of the massive scale on which computers are deployed, trading a small amount of performance for power savings is often acceptable[Micron 07]. Although processors are often thought of as the greatest power draw in a system, servers with many gigabytes of memory may actually use more power to keep data in memory than to process it. Hard disk drives, network interface cards, routers and many other components of modern computer systems are all working to quantify and reduce their power usage, so it comes as no surprise that DRAM manufacturers have defined timing parameters that, when followed, will significantly reduce the total power dissipated by this subsystem.

The problem is becoming more pronounced as data rates continue to rise. As transmission frequencies increase, the I/O drivers must work at greater speeds, increasing power usage proportionally to the square of the operating frequency. DRAMs have more banks now than ever before, so there are more opportunities to overlap operations across several banks simultaneously. As seen in Figure 4.29, the current drawn by a single bank activating and precharging is quite significant. If several banks were to activate at once, the current profile would look like the summation of their respective current profiles. This can lead to a very large

instantaneous power draw and, if sustained, may begin heating the chip beyond acceptable limits. In order to help avoid this heating problem and help reduce the power consumption of DRAM devices, several new timing constraints have been introduced to try to limit how much power is being drawn at any one time.

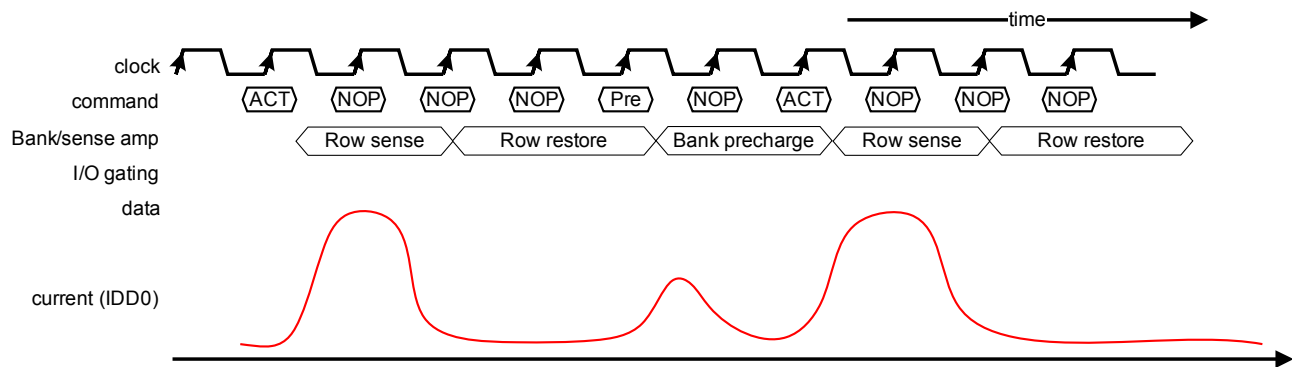


Figure 4.29: The current profile for a bank due to activates and precharges

4.4.1 Four Bank Activation Window

Defining the setup of a DRAM device is, from a manufacturer's perspective, a balancing act. If the number of banks is increased, the control logic becomes more complicated and the critical path becomes longer, thus decreasing the rate at which a DRAM can operate. Larger rows require larger sense amplifiers and thus will draw more power for each row activation. Not only this, but it takes longer to sense and restore the data, increasing latency for reads and increasing t_{RAS} and t_{RP} . However, these parameters make it possible to make larger DRAM devices, which consumers demand.

If a manufacturer decides to make an 8 bank device, it will take twice the current if all the banks are activated one after the next. The system will have to be designed to provide enough current to maintain steady voltage rails at this sort of current draw and the DRAMs will have to dissipate twice the heat of a 4-bank device. Because a row activation is among the largest uses of power and banks may overlap activations to multiply the value, manufacturers have targeted these commands for limitation[Micron 10]. They have defined a timing parameter, t_{FAW} , which establishes a window of time during which there can be no more than four activations to a device[Wang 05-2]. This effectively limits the maximum rate at which activates and, possibly, reads and writes can be issued to a rank. Although the memory controller may be able to reuse the

row for many reads and writes, the activations are what consume the most power, so t_{FAW} forces the memory controller to spread these over a greater amount of time.

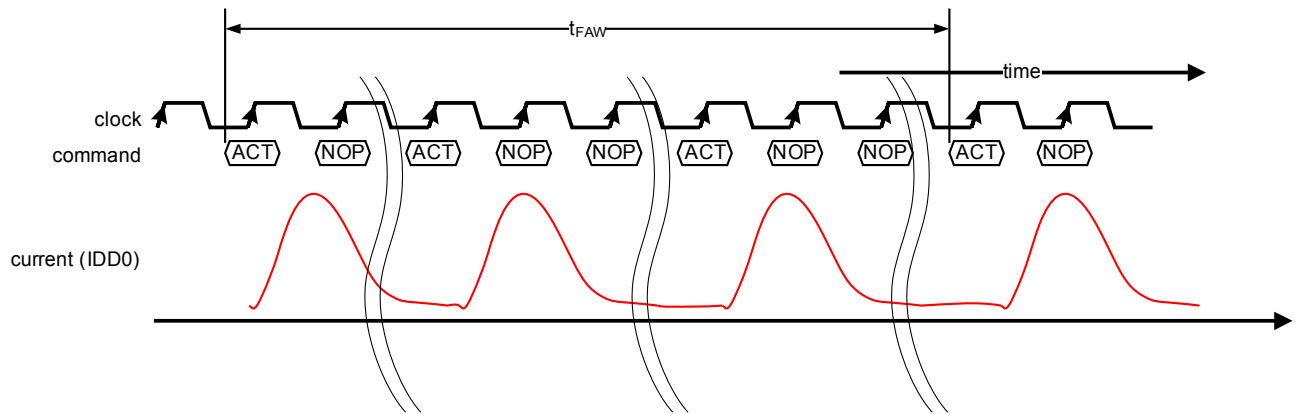


Figure 4.30: t_{FAW} and the corresponding current profile, activates may be to any bank in a rank

Figure 4.30 shows that the activates may be spread over time but may not exceed four in this moving window. Because the last activate occurs at the end of the time window, it is possible to issue another activate immediately after. However, a memory controller must keep in mind that once all activates to a rank have been used up, it may have to wait for t_{FAW} to expire before issuing any more activations to this rank.

t_{FAW} was introduced with DDR2 memory, but continues with DDR3 and will likely exist for some time. As each generation increases the row size in an effort to expand the capacity of DRAMs, the current profile (IDD0, activation current) will increase, likely causing t_{FAW} to become longer. In a Micron 1Gb DDR2 SDRAM, t_{FAW} is defined to be 37.5ns. For a similar Micron 1Gb DDR3 SDRAM, t_{FAW} is increased to 40ns (or 50ns for x16 parts). The value of t_{FAW} is likely increased in an effort to reduce power from one generation to the next. Making larger rows would surely have increased t_{FAW} by even more than this.

4.4.2 Row-to-Row Activation Delay

Much like t_{FAW} , the timing parameter t_{RRD} is intended to limit the instantaneous current draw of the DRAM devices and keep power dissipation under control. This parameter practically limits the total available bandwidth by limiting how often an activate to any rank may be given.

Assuming there are only a handful of reads and writes waiting for each row to be activated, in order to do useful work, the memory controller must activate many rows. The memory controller would also want to scatter those across as many banks as possible to utilize as much of the chip as possible in parallel. However, the t_{RRD} limit restricts how often activates can be sent, so the memory controller cannot simply open a new row every few cycles. The problem becomes more apparent as the number of banks increases. With more banks available, it is likely that a memory controller with numerous requests could spread those commands across many banks and achieve very high performance. However, many banks means that some banks will sit idle while waiting for an opportunity to activate another row once t_{RRD} has elapsed. As shown in Figure 4.31, the memory controller has many requests to send but must wait in order to not violate t_{RRD} . The current profile remains fairly low, however. If more banks were activated one after the next then the currents would add and the total power would grow proportionally to how many banks were doing a data sense and restore.

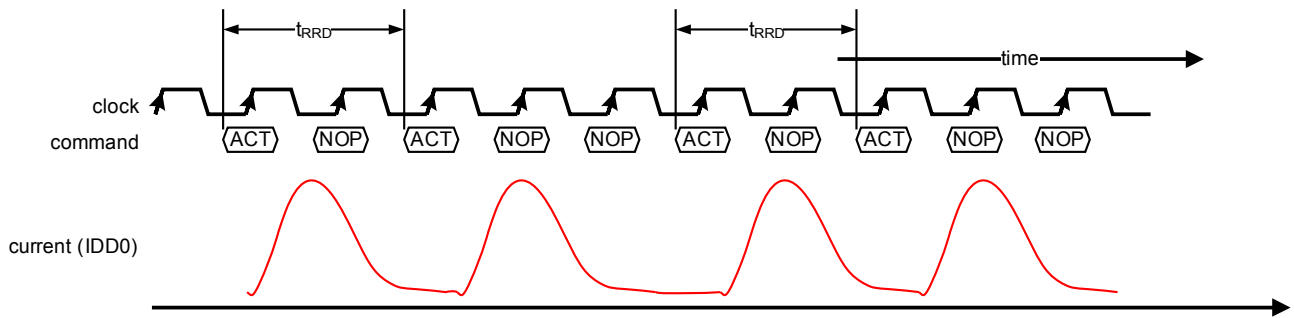


Figure 4.31: Two of the activate commands are limited by t_{RRD} and cannot be executed closer in time

With t_{RRD} limiting how often the banks can be activated, the current from the last activate has gone down from the previous activate already. In a Micron 1Gb DDR3 DRAM, the value of t_{RRD} is 6ns for the x4 and x8 parts, but increases to 7.5ns for the x16 parts. This is because the rows are larger, draw more current and take longer to finish the data sense. So t_{RRD} must be increased to make certain that there is no more overlap in the current spike in a x16 part than there is in a x4 part. Otherwise the x16 parts would draw more power than their x4 and x8 counterparts.

It is also interesting to note that the values of t_{RRD} (and t_{FAW}) are specified in nanoseconds rather than cycles. This is because the clock rates for each may change, but the data sense operations are not tied to a clock and will take the same amount of time regardless. Because decreasing the clock period will not reduce the sense time, it only makes sense to specify these values in ns.

4.4.3 2T Command Timing

In systems with a large number of ranks with unbuffered DIMMs, ensuring correct delivery of commands becomes more difficult. Each additional device connected to the buses adds capacitance that can degrade the quality of high-speed signals. The effect is that commands and addresses take longer to fully propagate to all the devices. If a device reads a command and the value is still changing, it may not recognize a read command properly or may not receive the correct address to read from. One undesirable way to alleviate this problem is to reduce the the address bus clock speed.

Another solution available in nearly every commercially available DDRx memory controller is 2T command timing. When 2T command timing is enabled, the memory controller will hold the values on the command and address buses for $2 * t_{ck}$ (cycles) rather than the usual 1. This gives the signals time to stabilize before they are sampled by the DRAM devices and reduces the number of signaling errors. Although this method improves signal integrity, it also means that the command and address bus bandwidth is halved. It is often desirable to attempt to run the system using 1T before resorting to 2T timing.

Fortunately, as the burst length has become longer, starting at 1 and now up to 8, the relative impact of 2T commands has become proportionally less. If a data burst can occur approximately every 2 cycles due to the 2T command timing, then there is a fundamental limit on how often a burst can happen. However, since the bursts are long compared to the addressing, $4 * t_{ck}$ (8 beats) by default in DDR3, this becomes less of an issue. Assuming there can be a data burst every 4 cycles, if the address bus is limited to 2 commands per 4 cycles, then the address bus can keep pace with the data bus and there will not be a performance decrease by using 2T timing.

CHAPTER 5 POWER MODELING

5.1 Overview

An increasingly important characteristic of any product is its resource usage. This is because the resources required to maintain and run a device or product have become relatively scarce and therefore more expensive than in the past. For example, with the price of fuel increasing by a factor of 3 or more in the past decade, cars are now evaluated on how much fuel they consume to transport the occupants a certain distance. Consumers are willing to pay more for a vehicle that is more efficient in hopes that it will save them money on fuel and maintenance over the life of the car. Also, major appliances are now rated by how much they will cost to run over the course of a year of regular use. They even give numbers based on gas versus electric heating in some cases. Cost of ownership is becoming a very important factor in the decision of whether to buy one product versus another.

Because resources are becoming more expensive and actually operating a device may comprise the majority of its cost, many companies who use lots of compute nodes are looking to build more energy-efficient solutions. Google Inc., a provider of many different Internet services, has studied the problem of power management in their numerous datacenters[Fan 07]. They looked at how power usage is divided up amongst the different components within a system and how the workloads and components affect how much is used by the system as a whole. It was noted that varied workloads will produce different power usage, even though components are usually specified in terms of one number, the peak power usage.

One interesting point that this paper made was that many workloads do not run while entirely utilizing any of the major subsystems in the server, they are tolerant of lower performance and do not drive the system at its full potential. Because of this, there are many times that the systems sit partially occupied or idle. When a system is designed to give great power performance at full utilization, but is less efficient in

terms of operations per watt at reduced speed, then the system is not ideal and could still be improved. They argue that designers must consider power usage for a wide range of applications and activity levels. If one system uses 70W at peak usage but drops to only 55W when idle versus a system at 95W peak and 25W idle, the system with the lower idle power may be preferred in situations where it is not expected to be constantly in use. Also, because many applications are written to run on multiple systems simultaneously, their power usage may be determined by how the systems are able to interact with one another. If an application does one part of the work on one machine and processes the data output from that machine on another node, it may be best to have a system optimized for high utilization doing the processing and a more efficient, slower machine doing the post-processing and data serving. These should be situations tested and run by system designers when simulating designs to ensure that new products are optimized for the situation that they will be used in, whether it is always idle, always in use or somewhere in between.

In a computer system, DRAM can be a major user of power, adding a good amount to the overall power used, depending on the quantity of DRAM installed, its specifications and the usage patterns. In order to help model the power usage, Micron Technology, Inc. has provided a guideline on how to estimate power usage of a memory system[Micron 07]. Understanding how leaving a bank open longer than necessary to improve performance may impact power usage is important to a system designer as they make decisions about how to manage the DRAM banks. One interesting piece to note is that if even a single bank is open in a rank, the entire rank is open. So the power draw from a rank with all the banks open is the same as a rank with a single rank open.

5.2 Background Power

The power dissipated by the DRAMs is divided up into two categories, background power and event power. Background power is dissipated at all times. There are several types of background power and which value is being dissipated depends on the state that the DRAMs are in. So, at all times there will be some sort

of background power being used, but always only one type. Because the voltage on the DRAM may be varied somewhat from system to system, only the current is specified for this type of power.

At any moment, a DRAM device may have a bank open within it or not. When one or more banks are open, the rank is considered to be active. When a precharge command is issued and all the banks are in the precharge state, the rank is considered to be in precharge state. In addition to affecting the state of the banks, the memory controller has access to the clock enable (CKE) pin on each of the DRAMs, which, when disabled, puts all the DRAMs into a lower power state with inputs. Using CKE effectively allows the memory controller to put the DRAMs into a low power state at virtually any time as long as the inputs are not needed. In order to leave this low power state, the CKE pin is deasserted and after a time the inputs will be available for commands and data once again. Because the time to leave the low power state is nonzero, the memory controller must be careful to take the right DRAMs out of the low power state soon after receiving a request in order to not have this exit time add to the transaction latency. The DRAMs may be configured in two different ways, known as “fast exit” and “slow exit.” The functional difference is how soon after CKE is deasserted the inputs are again available. If “fast exit” is chosen, the inputs will be available sooner and the memory controller will be able to make a request to the DRAMs in less time than a “slow exit.” The drawback to “fast exit,” and likely the reason that there are two different exit modes is that the value of I_{DD2P} is increased when the mode register is set for “fast exit.” I_{DD2P} is the current drawn by a DRAM device while all banks are precharged and the clock is disabled; this is the lowest power state. Because I_{DD2P} is higher, the minimum power that a DRAM may dissipate is increased and the power floor is raised on this system. In portable devices where memory is less of a bottleneck and performance is not the primary target for the system, the extra latency of a “slow exit” would be tolerated. However, when performance is paramount, “fast exit” and the additional power it brings must be chosen and power must be

5.2.1 All Banks Precharged

When all of the banks in a device are precharged but the clock enable is asserted, the device is in precharged, standby mode. While the device is in this state, it draws I_{DD2N} , the average current while in this

state. As long as all the banks remain precharged, the CKE determines how much power is being dissipated. When it goes from high to low, the current profile changes from I_{DD2N} to I_{DD2P} . So a good approximation for the power used while in these two states would be $V_{DD} * I_{DD2P}$ for precharged, power down or $V_{DD} * I_{DD2N}$ if it is precharged and in standby mode with CKE asserted.

5.2.2 One or More Bank Activated

If any of the banks are activated and hold a row in the sense amplifiers, the entire device is considered to be activated. At this point the memory controller may read or write to the open row. Being in this state consumes more power, so it is advantageous for the memory controller to keep the banks open as long as necessary and then close them to reduce power consumption. If the bank is active and the CKE is asserted, the device can receive commands and send or receive data. During this time, the device draws I_{DD3N} , so the power would be $V_{DD} * I_{DD3N}$. If CKE is deasserted, thus disabling commands, the current is I_{DD3P} . The power would then be $V_{DD} * I_{DD3P}$. This state is often used when the memory controller expects another command soon and it is faster to go into the power down state with banks open than to close the banks and possibly go into the power down state.

5.3 Event Power

Event power refers to the power drawn by the DRAM devices which is in addition to the background power and is associated with a command being issued. When an activate command is sent, sensing and amplifying the values stored in the DRAMs costs a certain amount of power, but is only incurred for the duration of that command. Likewise, when a read occurs, the power dissipated by the I/O buffers and the terminating resistors on the other ranks is incurred only while that read is being sent.

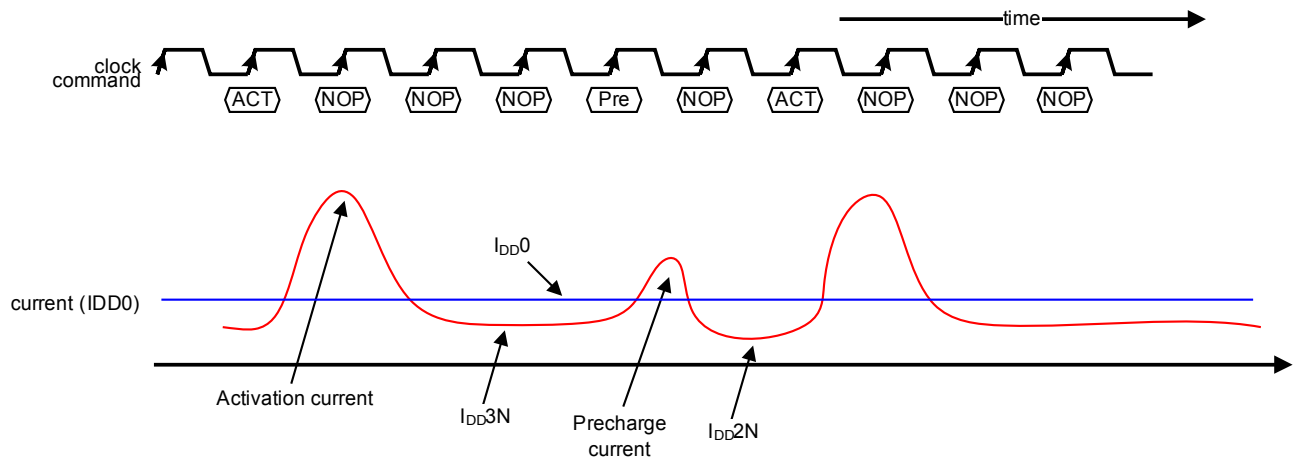


Figure 5.1: Current profile for a DRAM device with CKE asserted

5.3.1 Activate Power

In order to read or write data, the bits must be sensed and read into the sense amps via an activate command. The row decoder and sense amplifiers consume the majority of the power during an activate. Although the majority of the current is used at the beginning of the activate, the value I_{DD0} is specified for the entire activation, from the activation to the precharge (assuming t_{RC}), as an average of the current consumed.

It is important to note that this value is specified only if every activate to precharge time takes exactly t_{RC} . Because there are many banks available for the memory controller to use, it is very unlikely that it would happen that each activate would wait until all other rows are precharged before activating a new row. Because of this, power calculations must be adjusted to take into account increased utilization and the corresponding power increase that comes with it.

Figure 5.1 shows the various different currents described previously. I_{DD0} is shown as the blue line that is the average of the current over time. This value only holds true if the RAS to RAS timing is exactly equivalent to t_{RC} . If the activate commands come more frequently, the current spikes will happen more and more often, so the average current will be greater. The power due to activation commands at regular intervals is given by the formula:

$$\left(I_{DD0} - \frac{I_{DD3N} * t_{RAS} + I_{DD2N} * (t_{RC} - t_{RAS})}{t_{RC}} \right) * V_{DD}$$

Formula 5.1: Activate power

This formula attempts to account for the power contribution of the activate without including any of the background power components. Because I_{DD0} includes row decode, sense, restore and background power, it is necessary to subtract away the background power to have an accurate sense of what power the activation is adding beyond just background power. For the duration of t_{RAS} , the bank is in activated standby mode, so it consumes I_{DD3N} . For the remainder of the row cycle, $t_{RC} - t_{RAS}$, the bank is in precharged standby mode, so it consumes I_{DD2N} . Because the two states must take up the entire time of t_{RC} ($t_{RC} + (t_{RC} - t_{RAS}) \Rightarrow t_{RC}$), this sum is divided by t_{RC} to give the power due to only the activation. Then it is multiplied by voltage to give power consumed for this operation.

Most systems will not be able to space all their activates at exactly t_{RC} , so Formula 5.1 must be adjusted to incorporate the actual rate at which rows were activated. Because several banks are available, the memory controller may be able to schedule activations to many channels in a row, resulting in an average row-to-row activation rate smaller than t_{RC} . Instead of scaling every activate and precharge cycle, it is often more convenient to calculate the average activation rate, t_{RRDsch} . This value is then used in the calculation t_{RC} / t_{RRDsch} to give an appropriate scale factor for Formula 5.1. If t_{RRDsch} is larger than t_{RC} , then the average rate of activation was slower than expected, so power is being dissipated more slowly. Therefore, as t_{RRDsch} becomes larger, the power dissipated decreases. If t_{RRDsch} is exactly the same as t_{RC} , then the scale factor is 1 and Formula 5.1 accurately describes the power usage of this system. However, if t_{RRDsch} is shorter than t_{RC} , then this means that the memory controller was able to concurrently use several banks of the device and was able to average activations more frequently than t_{RC} . As expected, in this case the scale factor becomes greater than 1 and the power dissipated is expected to be larger. If two banks within a device are being activated at t_{RC} , then the activations are happening at a rate of $2 * t_{RC}$. In this case, the scale factor would be 2 and the activation power would be twice what Formula 5.1 predicts.

5.3.2 Read Power

Reads are represented by the current value I_{DD4W} . As seen in Figure 5.2, the current from the different consecutive reads can sum to give a greater current than is possible with a single read. The power for a read, excluding the background power while in active standby mode is:

$$(I_{DD4R} - I_{DD3N}) * V_{DD}$$

Formula 5.2: Read Power

The value I_{DD4W} represents the I/O drivers sending data continuously as well as background power, I_{DD3N} must be subtracted to give only power due to a read. Because the measurement I_{DD4W} assumes that data is being sent for all active cycles, the value given by Formula 5.2 must be scaled according to how many of the cycles were used for data transmission. So if there was one burst of 8 in 32 cycles, the data bus would be utilized for 25% of the time and Formula 5.2 would be scaled by .25.

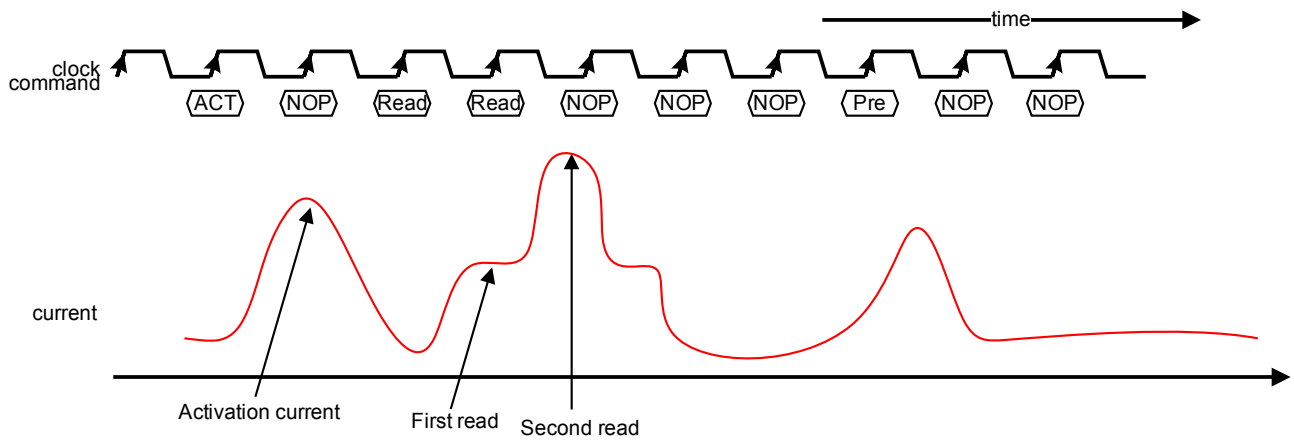


Figure 5.2: Read current for two consecutive reads to the same bank

5.3.3 Write Power

Writes proceed in almost exactly the same way as reads, with a current profile that looks quite similar to that of Figure 5.2, except with write commands instead of read commands. When multiple writes occur consecutively, the overlap also produces a larger overall current similar to what is noted as “second read.” In order to calculate the power due to only the write and not any background power, the following formula should be used:

$$(I_{DD4W} - I_{DD3N}) * V_{DD}$$

Formula 5.3: Write Power

This removes the active standby current that is part of I_{DD4W} and gives power due only to the write itself. In much the same way as a read, this power value assumes that the write data will be sent continuously over the time being calculated. In order to adjust, the power must be scaled by only the portion of time that was actually used for the data burst. For example, if the write was just a single burst of 8 and it was during a period of 32 cycles, the scale factor would be 0.25 and only $0.25 * \text{Formula 5.3}$ would be used.

5.3.4 Termination Power

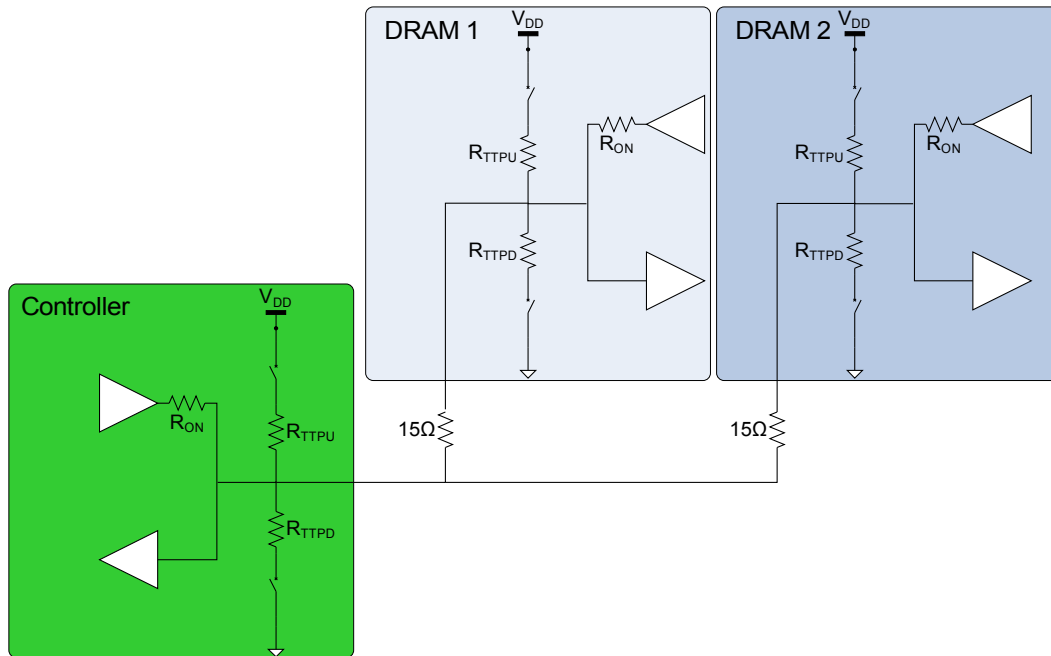


Figure 5.3: DDR3 DRAM Termination Scheme

As described previously, DRAM devices have on-chip resistor networks that they connect to the bus in various situations. When a DRAM or rank is not in use, they add an impedance to the bus to improve signal transmission characteristics. These additional impedances also dissipate power and must be calculated when considering DRAM power. So, in addition to dissipating power when driving the bus, either the memory controller or a DRAM also dissipates power in the resistor networks of adjacent DRAMs.

Unlike the other types of power, termination power increases as the number of devices on the bus increases, even though most of the devices are not in use. The values of the internal resistor networks may be chosen using the mode registers in the DRAMs, so it is possible to vary the power from setup to setup. Although this method will calculate the power using only DC analysis of the resistor networks, it provides an approximation that is usually good enough to get a good idea of what power will be dissipated.

In Figure 5.3, there are several impedances named R_{ON} . Each DRAM has a value of R_{ON} for their output drivers because this is the output impedance and is due simply to how the circuit is made. The 15Ω resistors represent the resistance inherent in the circuit, whether in the leads from the ICs or the traces on the

circuit board. Systems with especially thin or long traces will have values higher than 15Ω, but for the purpose of this example, the value 15Ω will be used.

When a DRAM is not involved in either reading or writing data on the bus, it will often set its resistor network to smaller values. For example, R_{TTPU} and R_{TTPD} will be set to 60Ω each, so the effective resistance of the DRAM is 30Ω[Micron 07].

5.3.5 Refresh Power

Periodically, a refresh command is sent to all the DRAMs in a rank in order to maintain the integrity of the data. The current associated with a refresh operation is reported in datasheets as I_{DD5} . I_{DD5} is a measurement of the average current when refresh commands are sent at exactly the minimum specified value of t_{RFC} . Because the DRAMs must be in active standby mode while doing a refresh, I_{DD5} also includes I_{DD3N} as a component. In order to account for only the power associated with the refresh operation, I_{DD3N} must be removed. Power due to a refresh, $P_{ds}(REF)$, is shown in Formula 5.4.

$$(I_{DD5} - I_{DD3N}) * V_{DD}$$

Formula 5.4: Refresh power

As seen in Formula 5.4, calculating the power attributed to the refresh operation is straightforward. Because it is very unlikely that a system designer would have reason to issue refresh commands at this rate, this power must be derated to reflect the actual rate at which they are being issued. DRAMs can typically hold values for milliseconds before any data corruption occurs, so it is best to issue refresh commands no more often than necessary to increase DRAM availability and reduce power consumption. DRAM availability is increased due to the fact that other commands will not have to wait for a refresh command to finish before issuing, nor will they have to wait for enough time to elapse before executing, as a refreshes make other commands wait. When refresh commands are issued over the recommended refresh interval, t_{REFI} , then the power dissipated is calculated as follows:

5.4 Derating Power For Specific Systems

To adjust the theoretical power used to a more realistic value for a real system, the power must be derated. All of the previous calculations consider that one is running the system at the specified voltage and specified frequency. If the system is run at any other value, then this change must be accounted for explicitly.

DRAMs give a maximum operating voltage and this is what the I_{DD} parameters are specified according to. Voltage typically scales as the square of the ratio of the nominal operating voltage versus the maximum voltage. All the different types of power are dependent on the voltage scaling, so it should be used to scale every type of power named previously. In order to adjust for whatever voltage is actually being used, the following formula should be used:

$$P_{sys}(type) = P_{sch}(type) * \left(\frac{V_{DD}(used)}{V_{DD}(maximum)} \right)^2$$

Likewise, frequency must be accounted for as well. Frequency only affects the background power and read and write power. Refresh power and activate power are not dependent on frequency. However, if the system is set to slow exit from precharge powerdown, then because the DLL is not enabled, it is not dependent on frequency. During fast exit, frequency scaling would still apply. Frequency is scaled as follows:

$$P_{sys}(type) = P_{sch}(type) * \left(\frac{f_{used}}{f_{spec}} \right)$$

CHAPTER 6 EXPERIMENTAL SETUP

In order to test the experimental setup and accurately determine which changes, if any, affect the performance of the system, a cycle-accurate DRAM simulator is needed. To accomplish this, DRAMsimII was designed to incorporate a framework for testing various algorithms and heuristics within a memory controller. The goal of the simulator was to give a level of accuracy that would show how not only how the behavior or reordering transactions and commands affect overall system performance, but also what effect the timing parameters have on power and performance.

6.1 Simulator Setup

DRAMsimII was built on the lessons learned from DRAMsim [Wang 05] and was rebuilt from the ground up to achieve greater flexibility improved simulation performance. The first improvement was to switch from per-cycle simulation to event-driven simulation. In a components like the cache or CPU pipeline, where some activity happens on nearly every cycle, it is convenient to simulate every cycle without checking to see if there is actually something to do first. However, when it comes to subsystems like the memory, it may actually be quite a lot of cycles between things happening, so simulating every cycle often ends up with the simulator wasting a lot of time doing nothing. To remedy this, event-driven models allow the various components to essentially accurately predict when the next operation will happen and schedule this. The simulator then keeps track of all the events for the different systems and goes and does work for them only when there is some work to be done. This improves simulator performance by only doing work when it is necessary, at the cost of keeping track of different events in a queue. The downside to event-driven simulation is that components that do work on the majority of available cycles tend to incur significant overhead to queue and schedule operations.

The memory system, however, does not do work on nearly every cycle, so it is compatible with the event-driven model. In fact, the underlying workings of a DRAM system are very similar to how an event-driven model works. For example, when the memory controller sends a CAS command to the DRAMs, it will expect a response to begin in $t_{CMD} + t_{AL} + t_{CAS}$. The data burst will finish at exactly $t_{CMD} + t_{AL} + t_{CAS} + t_{Burst}$. This means that the simulator can schedule events for the beginning and end of a data burst at these two times and not worry about the cycles before and in-between. The hardware would maintain counters to keep track of when the data was coming back, but the simulator can abstract this away. Likewise, if commands are sent using 2T command timing, this can be expressed as a longer interval that the command bus is not available.

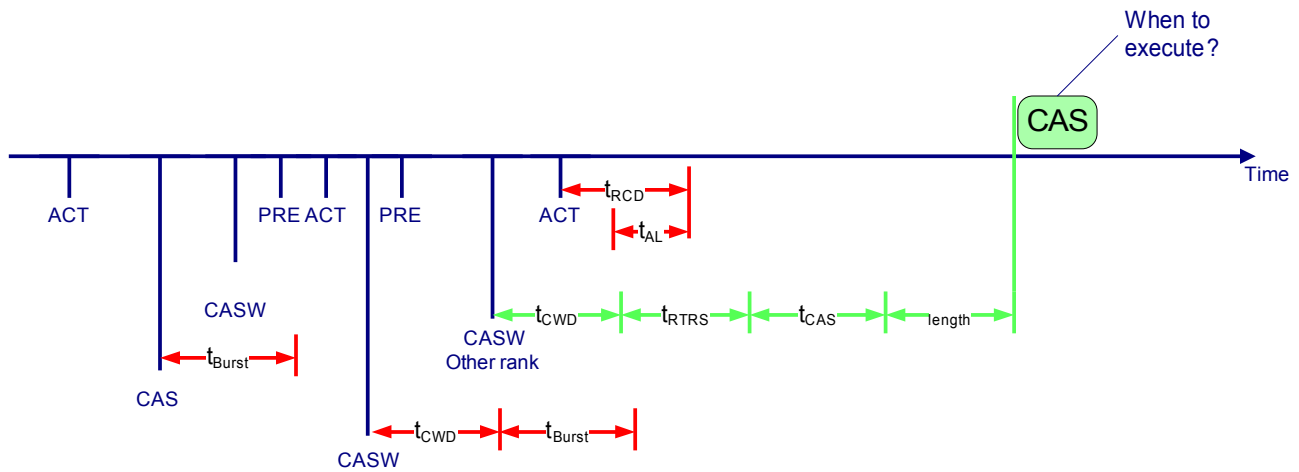


Figure 6.1: How timing parameters are enforced in an event-driven model

In order to enforce timing parameters, the next command is scheduled whenever all other relevant timing parameters have expired. In the case of Figure 6.1, there are several other ACT, CAS, CASW and Pre commands preceding the CAS command at the top of the diagram and the timing parameters must be respected. So DRAMsimII simply looks at all of the other timing parameters and determines which one prohibits the execution of the CAS the longest and uses this time to schedule the CAS command. As long as all the commands and data movement is restricted by minimum timings, the memory controller will not violate timing constraints and the simulated events would emulate what a real memory controller could do.

Values are tracked and calculated as events happen. So in addition to events which correspond to things that happen at certain times, the channels, ranks and banks store information about the minimal times at which events can happen. These values are used to prevent events from happening at times that would create timing problems for the DRAMs. Additionally, some of these parameters are there to simulate critical path timing or data bus utilization. For example, to emulate the time that the internal logic would take to decode and organize the appropriate commands for a particular transaction, there is a parameter that determines how much delay there must be from the time a transaction is received until it can be decoded. Although there may be room to decode the transaction immediately and store its constituent commands, it must still wait for this time to elapse. Likewise, each channel has a parameter that corresponds to how long it takes to transmit a command to the DRAMs, depending whether it is 1T or 2T timing. No other commands may be issued to other ranks, regardless of other timings, until this time has elapsed. Even though the fact that the bus is in use is expressed as a timing, it creates a uniform mechanism to enforce data transmission and timing restrictions.

As commands are executed or transactions decoded, internal values are updated to keep track of when the next commands may be issued. This time is a minimum value that each type of command must wait for to avoid violating any timing constraints. Because many commands are evaluated before one is chosen, it is much more efficient to keep track of when the next type of command may be executed than it is to keep track of when the last command of that type was executed and then apply all the timing rules. Also, only the relevant rules must be applied at the time of the command's issue. For example, a RAS command would trigger the rank to update the earliest time when another RAS command could be issued. It would be the maximum of t_{RRD} from now or t_{FAW} from the oldest time in the four activation window. Then, when the command ordering algorithm evaluates a RAS command to see if is eligible to be executed right away, it needs only to check the current time is greater than the earliest activate time in the respective channel, rank and bank.

6.1.1 Multithreaded Workloads and Thread Synchronization

When requests are sent from the CPU through the bus to DRAMsimII, they are converted to transactions and returned. However, there are a certain class of requests that are intended to facilitate ordering between threads and must be handled in an atomic manner. To allow thread synchronization, the

$$\frac{t_{RFC}(MIN)}{t_{REFI}} * P_{DS}(REF)$$

Formula 6.1: Refresh power when scheduled at t_{REFI}

ISA supports LLSC (load-link/store-conditional). This allows the threads to load a value from memory and have it protected until it is written back by that same thread. If another thread has modified this value then the store will fail. These requests should not be allowed to be reordered so that the program can run correctly, hence these requests are handled as they arrive. The locking portion is stored separately from the rest of memory and allows the requests to be responded to immediately. This means that these sorts of requests can be handled atomically and will ensure correct program operation.

6.1.2 Simulator Data Movement

As can be seen from Error: Reference source not found, DRAMsimII is essentially a large group of queues and algorithms where transactions and commands move through in the process of being completed and returned to the system component from which they came. Various heuristics for choosing the order in which transactions and commands are moved around can greatly affect the performance of the memory system. If the transaction queue cannot handle the rate at which new transactions are added from different sources, it will have to turn away requests and will create stalls in the CPU or network interface that is sending the transactions. Because every queue depends upon the queues before it to supply it with data and the queues after it to accept data, the choice in heuristics to manage the queues is especially important. Various policies must work together to improve the movement of data through the memory controller or performance will be reduced.

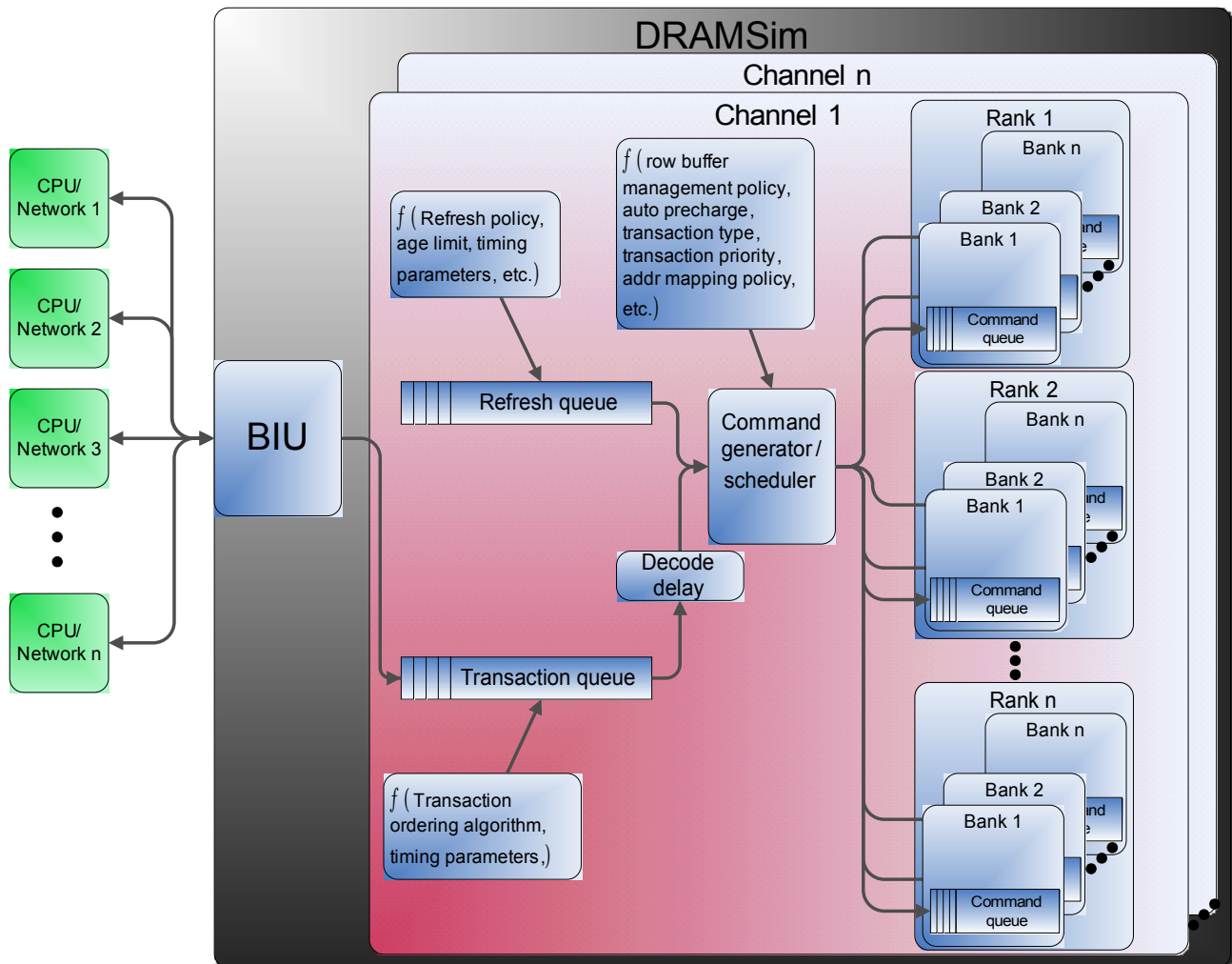


Figure 6.2: Data layout of DRAMsimII showing movement of transactions and commands

Likewise, to improve power usage, the policies must work in such a way as to work with the DRAMs to reduce power usage. If the DRAMs are left with rows active most of the time but are not servicing column requests, then the DRAMs are using more power than they could be. In an embedded system, where power is more important than performance, if the requests are not very numerous and they are well-distributed across several ranks, then these ranks are using power to stay active when they could easily be precharged with only one rank active. The following sections will look at some of the policies and heuristics that make up the memory controller and how they can impact performance.

6.2 *Transaction Queue*

The transaction queue is the first queue that a read or write request will enter when it is sent to the memory controller. The queue can be implemented as either a true queue, where requests are removed once they are converted to commands and placed in the command queues or as a series of holding registers that hold the transactions until they are finished, similar to MSHRs. DRAMsimII implements the transaction queue as the former, removing transactions once there is sufficient space to decode them into a series or one or more commands. The queue serves as the queue for all transactions in this memory system, regardless which channel they are bound for. Transactions cannot be routed to the channels until their addresses are decoded into locations, so when a transaction is instantly assigned a slot in the transaction queue, it is assumed that there was not time to determine which channel it will go to.

As the requests are placed into the transaction queue, there are two different ways that the transactions may be inserted: as a first-in-first-out or as RIFF. FIFO is as it sounds, simply placing the request at the end of the queue. This is the simplest approach, takes less logic to implement and is likely the fastest when implemented at a circuit level. RIFF stands for read and instruction fetch first, so reads and fetches are prioritized over writes and go to the front of the queue. It is almost like there are two queues in one, the reads at the front and the writes behind the reads. All types of reads have the same priority, so a fetch may arrive after a read, but will go to the end of the reads. RIFF is more complex to implement because it must check against RAW (read-after-write) and WAR (write-after-read) hazards. Write-after-write hazards are prevented by the fact that writes are inserted in-order with respect to other writes. Because each insertion must be checked against each existing item in the queue, using the RIFF policy will be slower and require more logic to implement. However, the benefit is that reads are prioritized and program execution can happen faster. Because the processors do not wait on writes, but do wait on reads and especially on instruction fetches, the memory controller should make an effort to prioritize these requests to help improve program performance.

As shown in Figure 6.3, when an instruction fetch is the next transaction to be inserted in the queue, then it will go immediately after the other reads and all writes. If it should conflict with a write already in the queue, then it would be placed immediately after the write that it conflicts with.

One additional improvement that may be possible is the prioritization of all instruction fetches over reads and writes. Because a processor frequently must stall a thread in the case of an instruction miss, execution of the application depends directly on the latency of the instruction fetches. So if the instruction fetches do not need to wait on the other reads then processor stalls can be better avoided.

The more information that can be passed with each request, the better the memory controller can use this information to prioritize the data. If a context ID is provided with each request, the memory controller

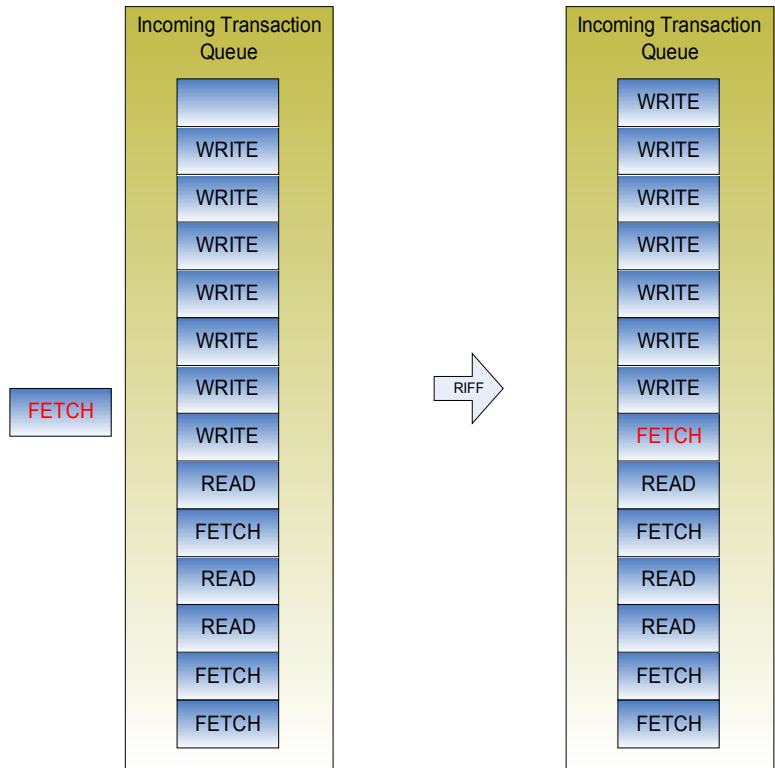


Figure 6.3: An instruction fetch is inserted into the transaction queue via RIFF

can be sure to prioritize the request appropriately to ensure a certain quality of service for that context ID. This will ensure a certain fairness between different threads competing for memory bandwidth and not allow certain threads to saturate the memory system and cause other threads to incur great delays in performing

memory reads or writes. Additionally, if the memory controller knows that some of the requests are prefetches, it can reduce the priority of these to be lower than that of regular requests that the processor is actively waiting on. If the memory controller is using read and write requests to predict addresses to prefetch on its own, then it can ignore prefetch requests from the CPU cache and more accurately predict what to prefetch. Essentially, as more metadata is passed to the memory controller with each request, the better it can decide how to handle the request and what data to collect from it.

When enough time has elapsed to simulate the translation from a transaction to several commands, the transactions can be removed from the transaction queue. In a manner similar to that of the issue window in a CPU, the requests in the transaction queue can be removed out-of-order if sufficient resources are available.

Figure 6.4 shows that several transactions can be removed if there is room for them in the command queues. The decode window can be set to a value smaller than the size of the queue to better simulate what actual hardware would be capable of. If the decode window is set to 1, then the queue will be able to remove only the first transaction each cycle.

The advantage to this is that the command queues can be filled more quickly and the requests can be finished sooner. If there is one transaction at the head of the queue that is waiting for a heavily utilized bank, then the rest of the transactions must sit and wait. If one bank is consistently heavily utilized, many other threads will be virtually starved while waiting for the requests to that particular bank to clear the transaction queue. With the decode window, this can be prevented and overall throughput to the memory system can be improved.

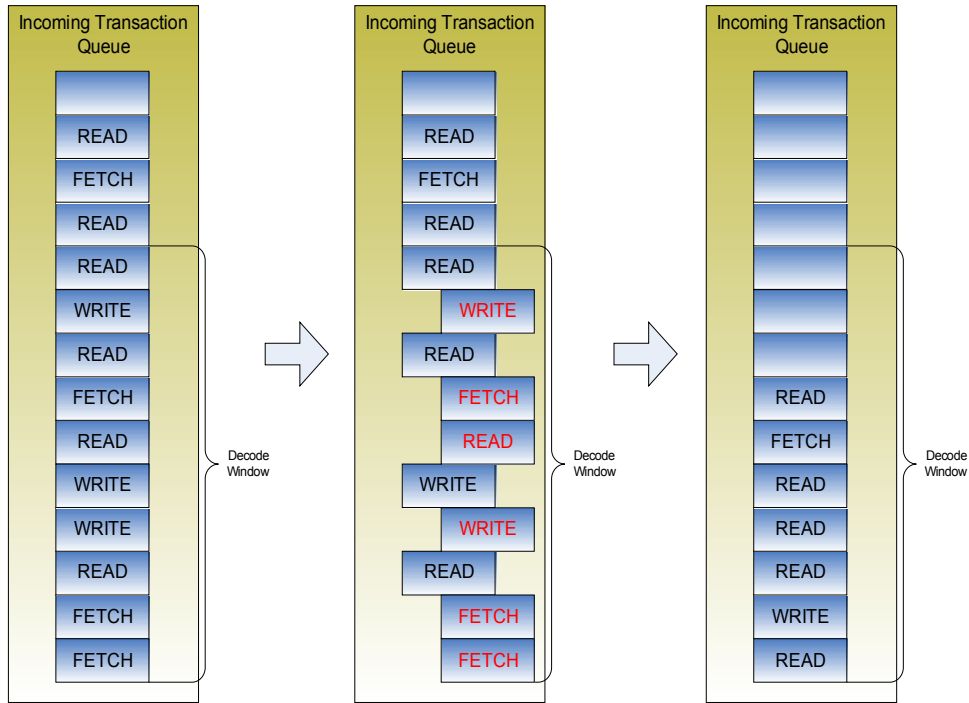


Figure 6.4: An illustration of how transactions can be removed from the transaction queue when they are ready and have available command queues

6.2.1 Refresh Queue

The refresh queue is quite similar to the transaction queue, but holds transactions periodically to attempt to send refresh commands to the ranks often enough to not violate t_{REFL} . When the memory controller is deciding which to choose from, the transaction queue or the refresh queue, it considers the age of the transaction in each. If they are similar in age, then the transaction from the transaction queue is chosen. However, if the transaction queue is empty then refresh queue is chosen. If the refresh queue holds a command that is relatively old then it is prioritized and chosen instead of the normal transactions. This is because the controller does not want to go too long between refresh commands and potentially corrupt data in the DRAMs.

When refresh transactions are decoded into commands, they are broken into as many commands as there are banks in a rank. This is to ensure that all the banks are closed when the refresh command happens as is required by the DRAM state machine. In the case of an open page row buffer management policy, the

refresh commands are all preceded by a precharge command to close out any row. When the front of every queue contains a refresh command, it is guaranteed that each bank is closed and the refresh command may be issued, so the refresh commands are dequeued from every per-bank queue at the same time and issued as one command.

There also exists the option to send refresh commands as a row activate followed immediately by a precharge. This allows for a more fine-grained control of the refreshes, but does not make use of the internal refresh counter in the DRAMs. The advantage of this scheme is that the banks will not sit idle while waiting for the other banks to finish any operations and then issue the refresh command. The banks will be refreshed individually. However, the per bank caches will have two additional commands each when a refresh is issued, which will increase their utilization and potentially impact performance. Also, the memory controller will need to keep track of which row in each bank should be refreshed next. With refresh commands, the DRAM keeps track of the next bank to be refreshed, but now the memory controller will need to have enough counters to handle as many banks as the memory system can be populated with.

6.3 Per-Bank Command Queues

In order to determine what bank commands should go to, they are decoded from transactions into commands. During this process, a particular bank is chosen, depending on the address. In order to alleviate some of the pressure on the incoming transaction queue, a queue for each bank is made. There is one queue per bank, so when it comes to choosing a command to execute next, the command selection algorithm can pick a queue and know that all commands in it correspond to one bank.

The depth of these queues can be adjusted to any depth the user wants. As the queues become longer, the better they are able to accommodate new commands. If the per-bank queues can hold more commands, then the transaction queue is better able to decode commands and clear itself out, so the memory controller will have a less full transaction queue and will have to turn away fewer commands. This means that the

CPUs can send away a memory request and wait for it to return, rather than having to stall to wait to send the request.

The downside to a deeper queue is that the overall complexity of the memory controller rises as it becomes longer. Because commands may be inserted in the middle or any other point of the queue, there must be hardware to compare every existing command already in the queue. Row buffer management policies like open page attempt to reuse open rows, so they group commands that are going to the same row. This means that each command must search the queue to see if a row is already open and then insert it behind the other commands going to that particular row. The queue can be implemented as a simple FIFO if a basic row buffer management policy like close page is used. In this case, commands are always inserted at the end of the queue, so the hardware to compare all of the commands becomes unnecessary.

6.4 Row Buffer Management Policies

The row buffer management policy describes how the memory controller manipulates the rows in a bank. It describes the strategy of how the memory controller is going to address the banks and in what state it will leave them. There are two basic policies: open page and close page. Since a row is often called a page, these are fairly descriptive names. The open page policy will close out an open row, activate a new row and perform a read or write, thus leaving the new row open. Likewise, the close page policy will open a row, perform a read or write and then close it when it is finished.

To choose between the two policies, one must consider the characteristics of the address stream and what sort of locality it has, spatially and temporally[Alakarhu 02]. Close page tends to assume that the addresses will be unlikely to be similar enough to map requests to the same row and therefore preemptively closes it out. This assumption is likely to be true in a heavily multithreaded environment, like a server running many virtual machines. Because their address spaces are completely separate, their requests will likely not map to many of the same rows, so closing out a row immediately after completing the read or write

will save time when the next request comes along. If the row stayed open, a precharge would first have to be issued, incurring a t_{RP} penalty before the activate could be performed.

Open page assumes the opposite. If the address stream has many requests to similar locations, such as a workstation or a compute node running simulations would, then it is likely that the requests would exhibit spatial and temporal locality. Because open page leaves rows open, if requests continue to map to open rows, then the memory controller needs only issue reads and writes and does not pay a time penalty for closing and opening the same rows again and again. Additionally, open page attempts to reorder the commands as they are placed in the per-bank queues to increase the amount of row reuse. Because addresses that map to different rows are guaranteed to be different physical addresses, there will not be data reordering errors. Data reordering errors, sometimes referred to as data hazards in the context of a CPU, occur when reads and writes to the same address are reordered and the resulting state is different from what it would have been if there was no reordering. For example, if a read is placed before a write to the same address, then the read will take the value in memory before that write has happened, even though it should have taken the value of the write. Likewise, if there are two write commands and they are switched in the order they are performed, then the memory will be left with the value of the earlier request because the latter request was switched and performed first.

To avoid this problem, when attempting to add a command to the middle of the queue, the queue is scanned from the back to the front. When a match to the same row is found, the new command is inserted immediately after the read or write command that matched. This strategy ensures that there the ordering of commands to the same row will be preserved and there will not be any data reordering errors.

There are several row buffer management policies available and, together with the address mapping policy and the command ordering algorithm, they determine much of how quickly a transaction will finish in the memory system.

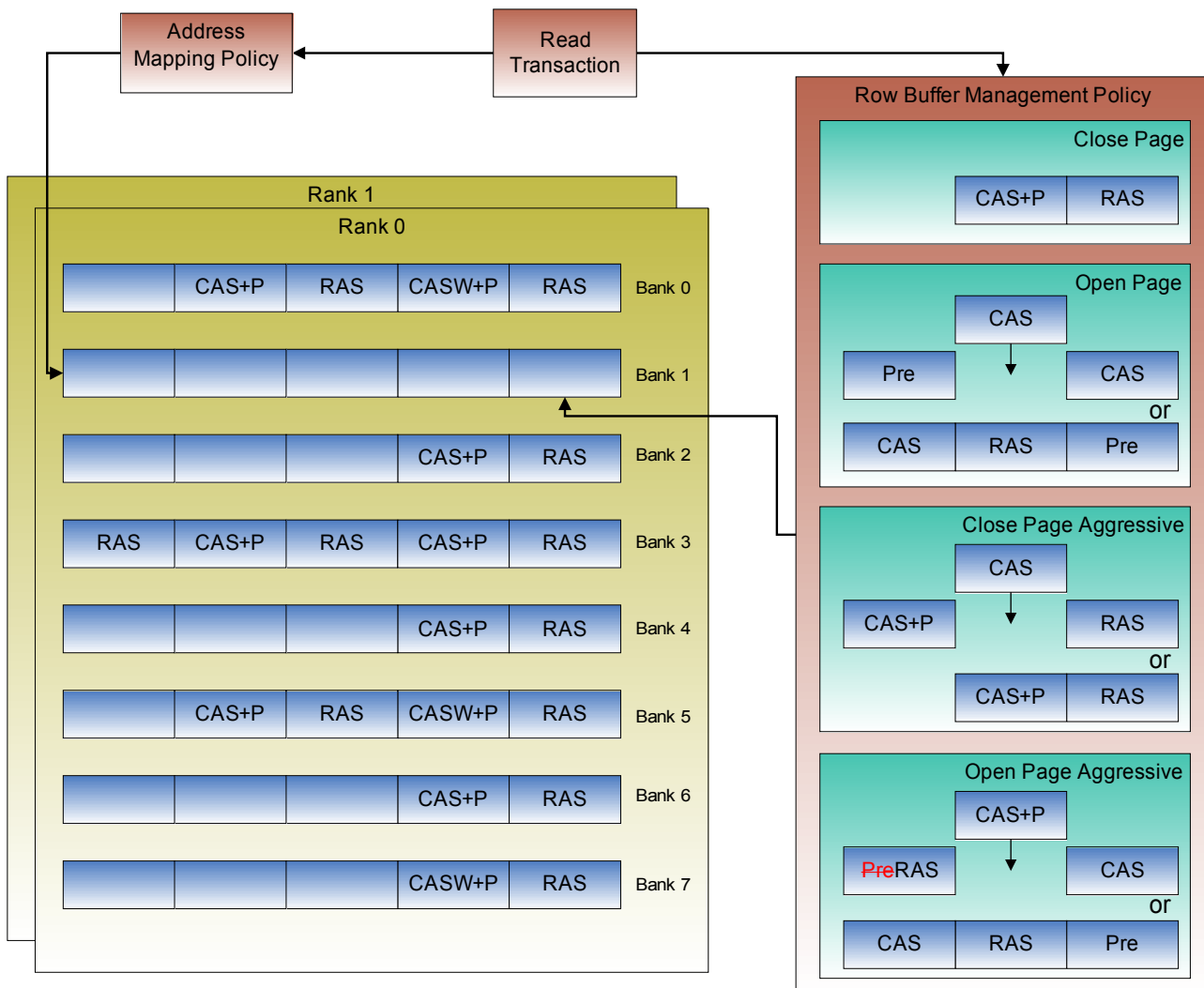


Figure 6.5: A comparison of how the various row buffer management policies work for a read transaction

6.4.1 Close Page

Close page is the simplest of the row buffer policies to implement and the simplest conceptually. The row is closed out once the commands are issued. In Figure 6.5, close page is shown at the top of the list as a RAS command followed by a CAS+Pre. If the DRAMs do not support auto-precharge, then separate CAS and precharge commands are issued. This policy does not support reordering, so there is no need to scan the entire queue before inserting the commands. It is therefore a very simple policy to implement, so there will be less hardware required to implement it and less power used by the memory controller when using it.

When a transaction is decoded into its constituent commands, it checks to see if there are enough slots in the per-bank queues to hold two commands. If the queue is too full, then the transaction is not

decoded until there is more room available. This would likely be a good policy for an embedded device, netbook or typical desktop system.

6.4.2 Open Page

Open page is essentially the reverse of close page, as it keeps the rows open until another command reuses the row or a new row is needed and it then closes the current row and opens another. Two variations of open page are available.

The simpler of the two looks at the end of the queue and determines if the read or write at the end of the queue will be using the same row as the new queue. If it is, then the memory controller simply inserts the command at the end of the queue and the row is used twice. If the queue is empty, the memory controller looks at which row is currently open. If the correct row is already open, then it simply inserts the read or write command into the empty queue and the open row will be reused. Should neither match the row of the new command, a precharge, RAS, CAS is inserted into the end of the queue. This scheme requires only a comparison of the open row and the last command in the row, making it reasonably simple to implement and will give better performance than close page if it is able to reuse the rows. Otherwise, performance will be worse as new commands will incur a penalty of the time it costs to precharge the row.

The other option for open page is to use all the same comparisons as the previous scheme, but also to scan through the queue to look for other instances where a command is going to use the same row. Then, the command is inserted after the commands already using the row. This creates row reuse where there would not be with the simpler scheme. However, there must now be the facility to insert commands into the queue at arbitrary points, so the queue is no longer strictly FIFO. Additionally, now a comparison of the row of each of the commands must be done, making the search more complex.

Also, there must be a counter with each command in the queue to determine how long it has been waiting. If this counter exceeds a certain number, no other commands are allowed to go in front of it. This prevents starvation of commands. Because a read command may be pulling the next instruction from

memory that a certain thread needs before it can proceed, it is important to limit the amount of time any transaction and its commands can wait. Otherwise, another thread may come along and have commands that reuse the same row over and over and cause the first thread to wait indefinitely long.

6.4.3 Close Page Aggressive

Close page aggressive is a hybrid policy designed to combine some of the benefits of both close and open page. When the memory controller is done with the row, it is closed, either by using CAS with auto-precharge or just a precharge command. Rather than simply inserting the new command at the back of the per-bank queue, the memory controller scans the queue from back to front, looking for other CAS(W) commands that are using the same row. Just like with open page, there are counters to prevent starvation of old commands and the queues will no longer be strictly FIFO.

This policy assumes that there will be a small amount of row reuse. It looks to reuse rows as they are opened, but still closes them afterward, assuming that it is unlikely that a command will arrive that will reuse an open row much after the row is opened. This should yield many of the benefits of open page's row reuse, but without suffering the penalty of having to close out a row before opening a new one.

6.4.4 Open Page Aggressive

This policy is similar to open page, but has several more complex optimizations. Ordinarily in open page, the precharge is a discrete command. This is done to give other commands as long of a chance as possible to reuse the row before it is closed. The same is true for open page aggressive, except that when the queue becomes more than some amount full, it will compress these commands into a CAS(W)+Pre. This reduces the number of commands on the command bus and frees up the row sooner for new commands to open new rows. No matter whether the command was going to be inserted into the middle of the queue or as a new RAS, CAS(W), Pre sequence, the commands are condensed to make more room for new commands. This keeps the per-bank command queues from filling as quickly, so more transactions can have their

commands in the queues at a time. It reduces the degree to which the per-bank queues become backlogged when the memory system is saturated.

Additionally, this policy assumes that when the queues are getting full that it is more important to finish up with one row and move on to the next than it is to try to find open rows and reuse them. So this policy behaves much more like regular open page under light loading but changes to become more like close page aggressive under heavy loading. At the same time, the benefits to keeping a row open when the per-bank command queues are empty is still realized.

6.4.5 Row Buffer Management Policy and Its Effects on Power

In addition to affecting the performance of the system, the row buffer management policy can also affect the power consumed by the DRAMs in a system. As described in the previous chapter on power, the amount of power consumed is dependent on which state the memory system is in. Policies that tend to leave all the banks open will remain in at least active-powerdown mode, using approximately 5x the background power of the precharged mode. If the memory controller is less aggressive about turning off CKE, then the DRAMs will likely be in active-standby mode much of the time, which is 6x as much background power. If there are not enough requests to justify being in this high-power state, then the power is simply being wasted.

Likewise, it is important to reduce the number of activations to reduce the activation power as well. If there are many opportunities to reuse a row, but the row buffer management policy is set to close page then the same rows will end up being closed and reactivated again and again. So not only will the rows be open nearly as often as with open page, thus consuming active-standby background power, but they will also have quite a lot of activation power used as well.

If the memory controller is an embedded device and is using one of the aggressive algorithms and constantly searching to try to improve performance, the memory controller itself will use more power. Many embedded devices only stream data from memory, thus limiting severely the amount of row reuse that occurs. In this case, even though the DRAMs won't use as much power, the memory controller will be over-

engineered for the job it must do and will deplete the batteries on the device. So it is important to know when it is overkill to add too much complexity to the memory controller.

It is also important to know what sort of workload will be running. If there are only desktop applications running, the power and performance benefits of a row reuse rate of 50% may be outweighed by the fact that the request rate is very low, so background power will dominate the power usage. At the same time, a very busy server will be receiving requests from many different address spaces, giving a low row reuse rate, but, when coupled with a close page policy, will end up activating rows very frequently. In this case, open page and close page will yield similar results. However, the server may have periods of high usage and times when the request rate drops very low, but row reuse increases. When this happens, the open page policy will use more power than the close page. Knowing what the intended workload is and what the design criteria are will help to choose between these policies.

6.5 Address Mapping Policies

One of the simplest but most important aspects of the memory controller is the address mapping policy. When a transaction enters the memory controller's transaction queue, it is a read or write to a specific address, often a 32- or 64-bit number. This address is unique, if a CPU writes to a specific address then it will expect to read back the same value at a later time. Should another CPU ask for the same address then it will expect to see the same value. If CPU0 writes to an address and CPU1 writes to it immediately after, the memory controller should enforce this partial ordering of memory requests. However, the way that the memory controller ensures that these conditions are met is left up to the controller. As long as every address is distinct and no values alias to one another then the system functions as expected.

In order to determine where in the DRAMs a particular request should go, the memory controller employs an address mapping policy. This policy dictates how a physical address, which is the value that processors, network interfaces, serial ATA controllers use to refer to memory locations, is mapped onto the channels, ranks, banks, rows and columns. Often the CPU will request a value at a particular memory

location. Because the minimum granularity in a DDR3 memory system is 64 bytes, often the memory controller will prioritize that byte and return it first and then send the other 63 bytes. The same holds true for other DDR variants: the minimum granularity is greater than 1 byte, so the processor should expect to receive more. Often the memory controller is setup to return data in chunks that match the size of the CPU cache block so that the CPU can deal with data a cache block at a time and not worry about data in a granularity other than that.

Because the low order bytes usually imply the critical byte or bytes to send first, the remainder of the physical address is available for the memory controller to use. Of course, there are some locations that the memory controller would never see due to memory mapped I/O, but it is assumed that memory holes are above the physical memory space for the purpose of this discussion. There are many ways to partition these remaining bits into ranks, banks, etc.

Essentially, the address mapping policy takes these remaining bits and divides them up into segments that map to physical locations. As shown in Figure 6.6, there are a number of ways to divide up this remaining number. Each block represents the chunk that will be taken to determine the physical location. None of the blocks tell how many bits are used because this is dependent on the installed amount of memory. If there are 2 channels, each having 4 ranks, each with 16 banks, 1024 rows and 32768 columns, then there will be 1 bit, 2 bits, 4 bits, 10 bits, and 15 bits, respectively. The order is then determined by the policy. In some policies, such as the SDRAM Base policy, there are “column low” and “column high” blocks. These represent the fact that the bits for the column are taken from two separate places, the low-order bits coming from the low-order bits of the address and the high-order bits coming from higher-order bits in the address. In the case of the Intel 845G chipset's policy, there is no block for channel. That is because this memory controller only supported one channel of memory, so there was no need to decode the channel from the address.

Designing a policy properly requires some a priori knowledge about what the data stream will look like. Once it is known which address bits will tend to give a good distribution, one can plan what sort of

Burger Base (BBM)

row	bank	rank	column	channel	Byte addr
-----	------	------	--------	---------	-----------

SDRAM High Performance (OPBAS)

row	rank	bank	Column high	channel	Column low	Byte addr
-----	------	------	-------------	---------	------------	-----------

SDRAM Base (SDBAS)

rank	row	bank	Column high	channel	Column low	Byte addr
------	-----	------	-------------	---------	------------	-----------

Intel 845G (845G)

rank	row	bank	column	Byte addr
------	-----	------	--------	-----------

SDRAM Close Page (CPBAS)

row	Column high	rank	bank	channel	Column low	Byte addr
-----	-------------	------	------	---------	------------	-----------

SDRAM Close Page Low Locality (LOLOC)

Column high	row	Column low	bank	rank	channel	Byte addr
-------------	-----	------------	------	------	---------	-----------

SDRAM Close Page High Locality (HILOC)

rank	bank	channel	Column high	row	Column low	Byte addr
------	------	---------	-------------	-----	------------	-----------

Figure 6.6: Various address mapping policies available for use in the simulator

address mapping policy would work well. For example, it is common for the low order address bits to change quite often from request to request as one thread often has a chunk of memory that it iterates through sequentially. Knowing this, one could put the column bits at the low-order end of the map and attempt to increase the potential for row reuse. If the row is mapped to higher-order bits, then the requests will often map to different columns within a row. This will allow the memory controller to use different columns within a row.

Another strategy is to assign the channel to the low-order portion of the map to improve concurrency. If the requests can be assigned to different channels, they can be completed in parallel and not have added latency by waiting for other requests to complete. The SDRAM High Performance policy does just that. If the low-order bits are mapped to the column and the row is mapped to some of the highest-order bits, then they will reuse the row and not incur precharge and activate penalties. The channel bits are right after just a few of the column bits, so the channels will all be used quite frequently. So the strategy is to use as few rows as possible, as many channels as possible and return the requests more quickly.

The Low and High Locality policies were designed specifically to compare against the other policies as they make different assumptions about the request stream. Low Locality attempts to distribute the requests to as many channels and ranks as possible, improving concurrency. From there, it assumes that there will be requests from several memory spaces, so the column is assigned to the high-order bits to attempt to exploit entropy in the high-order bits. Essentially it assumes that there are requests that appear in clusters, but each of the clusters of requests are widely distributed from one another.

The High Locality policy is another original, which attempts to move the row down to see if there is more locality in the low and upper middle bits. There are many more possibilities available. As long as all parts of the physical system are available through the address mapping policy, then it will work. How well it works really depends on how well it meshes with the address stream and the other policies in the system.

If the row buffer management policy is set to close page, then an address mapping policy that accurately creates opportunities for row reuse is not a good policy. Close page will not exploit row reuse, so by mapping the requests to fewer rows will yield no performance benefits. At the same time, if the policy does not map the requests to enough different parts of the memory system then the requests will start to get backlogged as they wait to use certain resources. If the address mapping policy does not work well with the address stream and maps all requests to one channel and one rank, then the performance of the system will suffer.

Lastly, in systems that are designed to be low-power, one must be very careful about how the address mapping policy is designed. A policy that uses all resources in a balanced fashion will give good performance both in terms of bandwidth and latency, but will suffer higher power usage. Even if only one bank is perpetually open in a rank and it cycles through the banks in such a way as to maximize performance, the memory controller will be unable to power down that rank. As long as one bank is open, the whole rank is in active mode and will consume far more power than if it was precharged. So a policy that fairly uses resources in response to demand will do better. For example, in an embedded system the request rate is likely to be fairly low, needing only a small amount of bandwidth and be very latency-tolerant. If the address mapping policy is likely to use only one channel and one rank at a time as it slowly streams through a large chunk of memory, then this policy would be best. The memory controller can disable CKE to the other channels and ranks until they are needed and use only a small amount of power, but still deliver reasonable performance.

It is important to consider all aspects of the system being designed in order to ensure adequate performance with minimal power usage. A small improvement in row reuse may lead to all rows being left open much longer and more power being used. Knowledge of the address stream is useful as well. Programs often go through memory in small, sequential chunks, so the low-order bits are often used to map to resources that reduce latency and improve concurrency.

6.6 Command Ordering Algorithms

Once the transactions have entered the transaction queue and been decoded into a channel, rank and bank by the address mapping policy and inserted into the per-bank queue by the row buffer management policy, it is then time to choose commands to execute. Each channel is completely independent, so they may issue commands at the same time. All ranks are independent, so except for the rank switching time, they may issue commands whenever the address bus is open. Banks are somewhat independent, but must observe several restrictions, like the four activation window. A good command ordering algorithm will be able to

keep all of these factors in mind when choosing which command to execute to effectively utilize all available resources.

There are at least a couple strategies to choosing commands. The first is to look at all available commands and choose one that has all resources available and can be executed immediately without violating any timing requirements[Rixner 04]. The memory controller will need to track all ongoing activities in all banks to know what restrictions due to timing requirements there are. It will also have to consider many commands at a time and choose from among them.

The other strategy is to follow a prescribed pattern of going through the banks. If the bank you are searching for has an command, then choose it. If not, then search the other banks. When a command has been selected, the memory controller needs only to wait until the timing requirements are met and then issue the command. Before getting into the details of the command ordering algorithms, it is appropriate to look at how timing requirements are kept and how a memory controller can efficiently determine when a command is able to be executed.

6.6.1 Timing Requirements

All of the different command selection algorithms require the memory controller to keep track of what DRAM resources are in use and when they will be free. Some algorithms only require that the selected resource be free and then issue the command at this time. Others want to know which resources are free right now so that they can choose commands that can be issued immediately and if not immediately, then when. This is not only for the purpose of sending commands that will not violate timing constraints, but also to choose commands that can better utilize DRAM resources and improve performance. If the data bus utilization goes up, then the system bandwidth will be closer to the theoretical maximum. So, in order to maximize the resource utilization of the channels, the memory controller must know when resources will be free so it can know when to issue commands to the DRAMs.

6.6.2 Timing Requirements – Channel

The timing restrictions for sending commands to a particular channel are very simple. Because the timing requirements imposed by the rank and bank cover most resources, such as sense amp availability and bus utilization, the memory controller needs only keep track of the command and address bus. To do this, the controller must remember only how many cycles have elapsed since the last command was sent to any rank and wait before sending another command. If 1T style addressing is used, then the address bus will be open for a new command every t_{ck} , usually on the rising edge of the clock. When 2T addressing is used, the controller will need to wait $2 * t_{ck}$ before issuing another command. Likewise for 3T and beyond.

6.6.3 Timing Requirements – Rank

To ensure that the resources in a rank are not utilized too quickly, the memory controller must keep track of more parameters for a rank than for a channel. These parameters restrict the timing of commands between banks in a rank and between ranks. So essentially, these parameters protect the resources of the buses shared by the different ranks as well as between banks within a rank. The parameters are summarized in the table below.

Previous	Next	Which Ranks	Time Constraint
Activate	Activate	Each	$\text{time}(\text{previous activate}) + t_{RRD}$
Activate	Activate	Each	$\text{time}(\text{previous activate}) + t_{FAW}$
Read	Read	Either	$\text{time}(\text{previous read}) + \max(t_{Burst}, t_{CCD})$
Write	Write	Each	$\text{time}(\text{previous write}) + \max(t_{Burst}, t_{CCD})$
Read	Write	Either	$\text{time}(\text{previous read}) + t_{CAS} + t_{Burst} + t_{RTRS} - t_{CWD}$
Read	Read	Different	$\text{time}(\text{previous read}) + t_{Burst} + t_{RTRS}$
Write	Write	Different	$\text{time}(\text{previous write}) + t_{Burst} + t_{OST}$
Write	Read	Either	$\text{time}(\text{previous write}) + t_{CWD} + t_{Burst} + t_{RTRS} + t_{CAS}$
Refresh	Refresh	Each	$\text{time}(\text{previous refresh}) + t_{RFC}$
Precharge	Refresh	Each	$\text{time}(\text{previous precharge}) + t_{RP}$

Table 6.1: Timing requirements for within a rank and between different ranks

The timing requirements are derived from the timing requirements in previous sections. The column entitled “Which Ranks” refers to which ranks are being considered. When this column says “Each,” then it

means that this is a requirement for any bank within a rank, but does not hold for separate ranks. For example, an activate may not follow another activate sooner than would violate t_{FAW} . However, this is a constraint for each rank, so separate ranks are not constrained by the four-activation window. When the column says “Either,” as the read-to-read timing does, this means that this is a requirement for all the different ranks and constrains them all collectively. So, for read-to-read timing, no command may be issued sooner than $\max(t_{BURST}, t_{CCD})$ to avoid having two ranks fighting to send data on the bus at the same time. Finally, when this column has the entry of “Different,” then this means that this constraint applies only to other ranks. The read-to-write and write-to-read timings refer to the time it takes for one rank to give up control of the bus and another rank to take control (t_{RTRS}) or the time it takes to switch the ODT from one rank to another (t_{OST}). When timing constraints are affected by different ranks, every rank is affected. So when rank 0 receives a command, not only are the counters for rank 0 updated, but so are the counters in ranks 1..n. This means that the ranks are not strictly independent and the memory controller logic must account for this explicitly to ensure that the data bus can be used correctly by the different ranks.

In the current implementation, each rank's information has a set of event time logs. These keep track of the most recent time a read happened on this rank, a write happened on another rank, etc. In this way, when the memory controller looks to see when a command can be issued, it needs only identify situation and use the framework of Figure 6.1 and see what is the greatest time amongst all the constraints. This is then the earliest time that a command can be executed with respect to the rank constraints.

6.6.4 Timing Constraints – Bank

Aside from rank and channel constraints, the memory controller must consider the individual banks as well. Many of the constraints imposed on a bank are fundamental limits of the DRAM device. Although resources like the data bus must be shared between the banks, these are already accounted for in the inter- and intra-rank constraints. The following chart illustrates the constraints that the banks must follow to ensure proper operation:

Previous	Next	Which Banks	Time Constraint
Activate	Activate	Same	$\text{time}(\text{previous activate}) + t_{RC}$
Precharge	Activate	Same	$\text{time}(\text{previous precharge}) + t_{RP}$
Refresh	Activate	Same	$\text{time}(\text{previous refresh}) + t_{RFC}$
Activate	Read	Same	$\text{time}(\text{previous activate}) + t_{RCD} - t_{AL}$
Activate	Write	Same	$\text{time}(\text{previous activate}) + t_{RCD} - t_{AL}$
Activate	Precharge	Same	$\text{time}(\text{previous activate}) + t_{RAS}$
Read	Precharge	Same	$\text{time}(\text{previous read}) + t_{AL} + t_{RTP} + t_{Burst} - t_{CCD}$
Write	Precharge	Same	$\text{time}(\text{previous write}) + t_{AL} + t_{CWD} + t_{Burst} + t_{WR}$

Table 6.2: The time constraints for commands to a given bank

One thing to note is that these constraints do ensure that the data bus has only one device transmitting on it at a time. That is done by the rank timing constraints, but could also be done by the bank timing constraints. These constraints have been discussed at length in previous sections, but are grouped together here to show what the memory controller must keep track of for every bank.

In its present form, the memory controller is just a simulation, so it can allocate structures when it starts. Real hardware, however, would need to account for variable configurations. If the channels are populated with 4 DIMMs, each having 4 ranks and 16 banks per rank, then the memory controller would need to have enough slots to track all of these banks. Alternatively, the memory controller could use shared storage and assign the memory as it detects what is installed in the channels. Then, for every command it evaluates as it looks at the per-bank command queues, it can simply look at the channel, rank and bank constraints and see if the counters have reached zero, indicating that a command can be allocated.

6.6.5 Command Ordering Algorithm: Strict

The strict algorithm attempts to make the memory system behave as a FIFO. So requests that arrive and are decoded first are executed and returned first. Because the requests are subdivided into the per-bank queues and divided into one or more commands, the system must track the requests in order to know which came first.

If the row buffer management policy is close page, then ordering of the requests to maintain FIFO order is simple. Simply tag each request with either an event number or the time it came in. Then look at the head of each queue and choose the lowest event number or time. In a real system, this could be accomplished by having a counter tracking how long a request has been in the queue and choosing the oldest command.

However, all of the other row buffer management policies will reorder the requests as they enter the per-bank queues, so it is important to tag all the requests appropriately. In this case, the strict algorithm will go and find the oldest command from the head of each queue. As a result, activates and reads or writes will tend to be grouped together since they were inserted at the same time. If the DRAMs allow it, using additive latency would be a wise choice. Additive latency would allow a CAS to follow immediately after a RAS and minimize the amount of time that the memory controller spends on a particular bank. If there is no ability to use a posted CAS command, then the controller will wait to issue the CAS command and may skip several opportunities to send commands to other banks. Thus, the amount of overlapping between the banks will be reduced.

The primary benefit of this scheme is its simplicity. The per-bank queues can be implemented as a single queue so that the algorithm does not need to actually search through several queues. Instead, the row buffer management policy can insert all new commands into a single, unified queue and the command ordering algorithm can choose the head of a single queue. A great drawback to this algorithm is the fact that there is no reordering to account for resource usage. Some commands could be issued to available ranks or banks while other commands complete, but if they arrived after commands intended for heavily utilized banks then they must wait. A lightly loaded memory system would likely benefit from this algorithm. If there are never more than just a few outstanding transactions at any given time, then it is unlikely that any commands will have to wait long for other commands to finish.

6.6.6 Command Ordering Algorithm: Bank/Rank Round Robin

Two algorithms, rank round robin and bank round robin are actually quite similar, so they will be discussed together. As the name would suggest, each of these algorithms go through the available ranks and banks in a round robin fashion. They look first at the next rank or bank that numerically follows the rank or bank that a command was last issued to.

So if the policy is rank round robin and the last command was issued to rank 6, then rank 7 will be searched first. If there are 8 ranks and nothing was found in rank 7, then rank 0 will be searched next, up to rank 5. Rank 6 will be searched, but not until after every other rank has been searched to attempt to prioritize every other rank above the previously chosen rank. In each of these ranks, the bank that was chosen previously is searched first. When every rank has been searched for a command that can be issue immediately, then the bank number is increased by 1 and the all of the ranks are again searched. Once the search has looked through every rank and every bank, then the search returns to the location where the last command was chosen from. If no command is available to be issued immediately, then the search is repeated to find any command at all. As long as one of the per-bank queues has a command, then the first one found will be chosen to be executed.

Although this command won't be able to be executed immediately, this command represents the next command that will be chosen, so the simulator can determine when the next event would occur and move to this event time. If there are no available commands and all of the per-bank queues are empty, then the algorithm does not return a command and the simulator can know that there are no pending commands in this channel.

Bank round robin works in much the same way. Instead of iterating through every rank on a particular bank, it does the reverse. It attempts to use a given rank as long as possible and switches banks each time in an attempt to choose a command from the same rank. If there are many available commands then this algorithm will tend to group commands to a given rank.

One interesting feature of these algorithms is that they treat activate and affiliated reads or writes as a single command. So if an activate is chosen, the read or write that follows it will be chosen by the next selection. Likewise, as long as there are groups of reads or writes, the round robin algorithms will continue to select from that rank and bank. This is intended to exploit open row buffers. If a row is opened, the row-to-column delay is often short, or zero in the case of additive latency, so the row is opened no longer than is necessary and the CAS command begins as soon as possible. This grouping also has benefits for power when using close page as the row is closed out sooner and less power is used. Without this grouping, there might be a large gap between the RAS and the CAS commands while the algorithm goes and chooses commands from every other bank before returning to the one where the RAS was originally. In the worst case, the banks will all be activated at nearly the same time and then all requested to send data at nearly the same time. By grouping the activates and the reads or writes, the accesses can be staggered and there will be less contention for the data bus.

One option available to either algorithm is known as read/write sweeping. This enables the system to preferentially choose reads or writes and continue to choose the same type until there are no more available. If reads or writes are chosen in long series, the system will tend to issue groups of reads or writes before switching to the other. The point of this is to improve utilization of the data bus. If many banks are available to send data, the delays associated with changing senders, such as t_{RTRS} and t_{OST} , can be avoided.

Bank and rank round robin algorithms are fairly complex to implement. Logic must be added to identify RAS and CAS pairs, keep track of whether a read or write sweep is in progress and which rank and bank was most recently searched. One or more banks must be searched, giving priority to banks earlier in the search. Despite implementation difficulties, rank and bank round robin have significant advantages in terms of performance that make them better choices than strict ordering for a high-performance memory system.

6.6.7 Command Ordering Algorithm: First Available

The First Available algorithms attempt to choose the command that can execute the soonest. Each variant goes through each per-bank queue and calculates, using timing requirements, when this command may next execute. This task is complicated by the fact that there are often several commands that can be executed immediately. To attempt to solve this, there are three different secondary criteria that define the three variants of First Available.

In an attempt to help keep the latency of commands to a minimum, the age of the command can be considered as the secondary criterion. The idea is that if two commands can both be executed now, the older of the two commands probably should be executed sooner so that the average wait time for requests is reduced. If two commands are of equivalent age, then the one with the lower rank and bank ID is chosen.

Another option for a secondary criterion is to choose reads and instruction fetches first (RIFF). Because a CPU or other device does not stall for a write, there is no need to prioritize them. However, since CPUs do wait on reads and especially on instruction fetches, each cycle saved in the memory controller can lead to a reduction in execution time. Thus, prioritizing reads when choosing which commands should be executed next is a good strategy.

Finally, choosing the command from the fullest queue as a secondary criteria can help to reduce congestion and improve performance. If requests tend to be mapped to certain banks more than others due to irregular access patterns, some per-bank queues can fill faster than others. If the queues are full, then the transaction queue will be unable to decode transactions into the queues and eventually the backlog will stall the CPU. Prioritizing the fullest queues will help to alleviate congestion and prevent any backlogs in overused banks. If two queues have a command that can be executed immediately and are equally full, then the queue with the lower rank and bank ID is chosen. This might be improved by making the selection random or by adding a third criterion, but in the current implementation.

All three variations on the first available algorithm attempt to keep the banks as busy as possible and improve performance. They all want to find commands that are ready to be issued and send them as soon as

possible, optimizing locally. Being able to always consider all commands improves parallelism by executing commands as soon as possible, rather than following a pattern. The drawback to this policy is that even though it is locally optimal, it may not be globally optimal. The secondary selection criteria may choose commands in one particular rank repeatedly and leave another rank idle for a time. For example, the algorithm may choose commands until t_{FAW} is hit and then have to move on to another rank. However, if the algorithm had alternated selection between the ranks, it may have been able to avoid hitting t_{FAW} and would not have had to leave one rank idle until the activation window had passed.

6.6.8 Command Ordering Algorithm: Command Pair Rank Hop

The command pair rank hopping algorithm is loosely based on rank or bank hopping as it follows a prescribed pattern. It is a patented algorithm, developed by Wang and Jacob [Jacob 09], to easily allow a memory controller to schedule commands to achieve high bandwidth and low latency.

The column accesses are scheduled sequentially, while the row accesses go to alternating ranks. So the algorithm will attempt to send bank accesses to each of the available banks in sequence, from rank 0, bank 0 to rank n, bank m. The row activates will increase sequentially as well, although cycling through the other ranks before returning to the original rank, next bank. The column access commands are grouped according to the same rank to attempt to get the maximum amount of transfers from a rank before incurring the t_{RTRS} penalty for switching to another rank. In fact, while it is going through all the banks within a rank, the bandwidth will be as good as the theoretical maximum. The benefit is that if there are commands for every rank and bank, then this algorithm can schedule commands in a similar fashion to rank or bank round robin, with row and column access pairs.

By sequentially activating the ranks in a round robin fashion, the algorithm attempts to avoid any stalls due to t_{RRD} . If there are four ranks, it will be at least every eighth command that performs an activate on any rank, assuming there are commands for every rank. For example, in a Micron DDR3-1066 DRAM, the t_{RRD} value is at least $4 * t_{ck}$ or 7.5ns. The clock period is 1.875ns, so even with 1T timing and only 2 ranks, a

second ACT command could not be sent sooner than 7.5ns later than the first. So there is no issue with waiting for t_{RRD} when using command pair rank hopping.

Another feature of command pair rank hopping is that it switches the starting rank after each sweep through all the banks in a rank. Specifically, it issues an activate to the same rank as the previous activate. The reason for this is that because the reads are switching from one rank to another, the time penalty of t_{RTRS} will be incurred. If the controller must wait for t_{RTRS} no matter what, then it is alright to wait for t_{RRD} , as it will likely wait longer on t_{RTRS} than on the remainder of t_{RRD} . So the sequence for the bank of each RAS command remains the same, the first rank changes to vary the rank activation pattern slightly. An example pattern is shown in the table below for 4 ranks and 8 banks.

This scheduling policy works well because it makes it much more likely that the first command chosen can be executed immediately. This means that there is little waiting for resources to become free and latency and bandwidth are improved. Command pair rank hopping also requires fewer resources to choose the next command. Because there is a prescribed sequence to follow, determined as the memory controller detects the memory configuration, the next command is easily chosen. Only when there are empty queues does the algorithm have to skip over a slot and evaluate more than one command. Additionally, the minimum time to execute a command needs to be computed possibly only once and need not be compared with other commands. This makes implementation much easier than with the first available policies.

RAS Rank, Bank		CAS Rank, Bank	
0	4	0	0
1	0	0	1
2	5	0	2
3	1	0	3
0	6	0	4
1	2	0	5
2	7	0	6
3	3	0	7
3	4	1	0
0	0	1	1
1	5	1	2
2	1	1	3
3	6	1	4
0	2	1	5
1	7	1	6
2	3	1	7
2	4	2	0
3	0	2	1
0	5	2	2
1	1	2	3
2	6	2	4
3	2	2	5
0	7	2	6
1	3	2	7
1	4	3	0
2	0	3	1
3	5	3	2
0	1	3	3
1	6	3	4
2	2	3	5
3	7	3	6
0	3	3	7

Table 6.3: Command Pair Rank Hop Sequence for 4 ranks, 8 banks

6.7 Random Address Simulation Mode

DRAMsimII can run in random address generation mode. If the goal is to see how various policies react to a given stream, the simulator may randomly generate its own transactions. The simulator will treat this as a request that arrives from a CPU or other device and respond to it.

The benefit of this is that simulations proceed much faster than they would with a full system attached. There is also no need to maintain the binaries or the system files required to run an operating system. Since only the memory system is simulated, there are far more requests processed per unit time, meaning that one can look at how the memory system responds to this level of loading. If one is trying to build an address mapping policy that scatters requests across more ranks and banks, then this is a faster way to determine if a new policy is better.

The downside to this type of simulation is that there is no feedback to the CPU, so the requests will come at a given time no matter how fast the memory system runs. In a real system, a faster memory system would return requests to the CPU sooner, thus allowing the CPU to send out new requests sooner as well. So there is no concept of execution time when using only a memory system. Even if all the queues are completely full, there is no processor to stall. There is no feedback to limit the rate that requests are sent to the memory system, so there are limited insights to be gained from running with only simulated addresses.

This mode can show the interaction of address mapping policies and command ordering algorithms quickly. However, because the address are random and lack the locality a real address stream would have, this mode does not simulate single-threaded or workstation applications well.

The requests may be grouped according to certain probabilistic models like Gaussian, Poisson, Normal and uniform. The simulator also specifies an average interarrival cycle time, which determines the average amount of time that elapses before another request comes along. If this is set to zero, then the memory system will likely have its queues filled very quickly to emulate saturation from the CPUs.

6.8 Simulation Setup: DRAMsimII and M5

In order to accurately simulate the performance impact that DRAMsimII would have on a given system, it was incorporated into the M5 framework. M5 is a full system simulator developed at the University of Michigan that accurately models Alpha, SPARC, MIPS and x86 ISAs[Binkert 06]. It provides a framework to create simulation and system objects independently and then connect them using ports, similar to real hardware. The simulator is event-driven, so the DRAMsimII module had to define and schedule events related to the memory system. Although M5 supports SMT/CMP and multi-system configurations, they were not simulated and will be saved for future work.

M5 has two different setups, known as syscall-emulation (SE) mode and full-system mode (FS). When running in SE mode, the system calls are handled by the simulator itself. So when the executable asks the operating system to perform a read to a certain location, the simulator handles this request and performs the read, returning the data to the application's memory after a certain period of time. However, in full-system mode, when a read is requested, the operating system validates permissions and translates the physical location of the read before sending a request to the simulated hard drive. After a period of time, the request is returned to the operating system which copies it to the application's memory space. So in SE mode, there is no operating system, several components are not simulated and the application is the only application running on the system at that time. SE mode therefore runs much faster and provides a reasonable approximation of how an application would behave, given a particular system configuration. However, in order to see how the operating system affects the execution of a given application, one must run in FS mode. FS mode adds in paging, process scheduling, realistic hard drive latencies, memory spaces and all the other features that an operating system provides. Also, because there are memory spaces for the application's process, the addresses requested by the process will be in a much different range than with SE mode. Because FS mode is more accurate than SE mode, it is used to test various configurations in this study. The fact that FS mode has address spaces is especially important when studying address mapping policies, as having all memory requests for an application starting at 0 does not make for a very realistic stream.

Aside from allowing FS and SE mode, M5 has two modes of operation: atomic and timing. When the simulator is running in atomic mode, whether in FS or SE mode, every component behaves functionally as it would normally, but without any timing information. So requests from the CPU for data in the L1 cache are returned immediately, just as requests to main memory. The point of atomic mode is to allow a functional simulation without waiting for timing requirements. This allows a simulated program to run faster, but without any behaviors caused by having delays in the system. Often this mode is used to skip through a segment of operation where the behavior is not intended to be studied. For this study, the operating system bootup and benchmark preparation are run in atomic mode because it is only the benchmark behavior that we are interested in.

Because M5 allows for most configurations of CPU caches, a L1 data and L1 instruction cache were used as well as a combined L2 cache. The L2 cache used a prefetcher to help make the system behave as much like a modern, real system as possible. Although M5 supports other ISAs and other CPUs, currently Alpha is the most reliable and developed model, so an Alpha Tsunami (21264) out-of-order CPU was used. Although this CPU does not exactly correspond to a modern x86 processor, it is similar enough to correctly emulate the memory stream that would be produced. The Alpha processor was set to a faster speed than any x86 processor runs at to minimize any effect the CPU would have on program execution. If the CPU is running very fast and the L1/L2 caches are also quite fast, the runtime will be less sensitive to the latency they contribute and more sensitive to the latency from the DRAMs. Additionally, when comparing one simulation result to another, the processor will be the same in either case, so any difference in runtime will be due to memory latency. So long as one simulation setup is relatively better than another, so should be the equivalent setup on real hardware. Although the hard drive is modeled with some accuracy, none of the benchmarks require virtual memory and none of them print to the disk, so any effects the hard drive might have on runtime are minimal. Below are some of the specifications of the simulated system:

Parameter	Value
Processor Speed	6GHz
ISA	Alpha
L1 I-Cache	64kB, 2-way, 64B block, 500ps latency, GHB prefetcher
L1 D-Cache	64kB, 2-way, 64B block, 500ps latency, GHB prefetcher
L2 Unified Cache	8MB, 16-way, 64B block, 6ns latency, 22 MSHRs
CPU ↔ DRAM Bridge	2.6GHz, 5ns delay
IDE Controller	2 disks, system and swap

Table 6.4: Common system setup for all simulations

6.8.1 Benchmarks

To test the changes that policies and algorithms can have on a system, a variety of benchmarks from the SPEC CPU2006 (SPEC06) benchmark suite. SPEC06 was chosen because it is a standard benchmark suite used widely throughout the industry. It has benchmarks chosen to work the core of the system, specifically the CPU and the memory subsystem. Additionally, SPEC06 contains stripped-down versions of many actual applications, so high scores in SPEC06 benchmarks will likely lead to good real-world performance as well.

The first benchmark is 401.bzip2. This is a modified version of the popular bzip2 utility, written by Julian Seward. Normally, bzip2 will take the input file, read it, compress it and write the resulting compressed stream to another file. This version, however, will only compress and decompress the file in memory. No output is written so that only the CPU and memory are tested. 401.bzip2 has several different files that it can use for different workloads, including image files, program binaries and an tar archive of another program's source code. There is also a workload that includes the concatenation of all of these workloads and this was the one used to run the benchmark. This provides a mix of highly and barely compressible files for the application to work on.

Next is 429.mcf, which is used to schedule mass transportation vehicles out of a single depot. The core of this application is solving a minimum-cost flow problem using linear algebra. A network simplex implementation is used, which involves pointer and integer arithmetic. There are test, training and reference

data sets available. For the following results, the test data set was used to reduce the amount of time spent simulating while still creating a realistic run.

445.gobmk simulates artificial intelligence playing the game of Go.

458.sjeng is based on Sjeng 11.2, a program that plays chess and several variants.

462.libquantum simulates a library that can simulate a quantum computer.

483.xalancbmk is an XSLT processor that transforms XML documents into HTML or XML documents.

433.milc is a chromodynamics simulator developed by the MIMD Lattice Corporation (MILC).

459.GemsFDTD is a computational electromagnetics simulator.

470.lbm uses the Lattice Boltzmann Method to simulate incompressible fluids in 3D.

Finally, a benchmark called stream is used to evaluate the raw bandwidth of each simulated system[McCalpin 95].

6.8.2 Methodology

In order to ensure that the memory controller was providing results that were at least realistic, the times of each command sent to the DRAMs was recorded as well as the destination. So there was an event log of every command to every channel, rank, bank row and column. Using these logs, a Verilog simulation was setup using the same timing parameters and configuration as the DRAMsimII setup. A specific speed grade, the SG125E bin, was chosen and both simulations were set to identical values.

The DRAM models were provided by Micron and included IBIS, HSpice and Verilog models. The examples included a trace file for a memory controller communicating with several DRAMs directly. Because they provided a DIMM module, the correct signals were attached to the DIMM. These included the chip select and clock enable as well as some address signals to ensure that the DRAMs behaved as a rank

rather than individual devices. Once this was setup, the trace file generated by running the DRAMsimII simulation was played back for the Verilog simulation.

The devices modeled by the Verilog modules checked to ensure that the bus was never fought for and that no timing constraints were violated. Some aspects that are abstracted away from the DRAMsimII simulations had to be added. These included setting up of the mode registers and performing clock calibration. However, once this was added and the traces were run, the Verilog modules reported no timing violations or incorrect addressing.

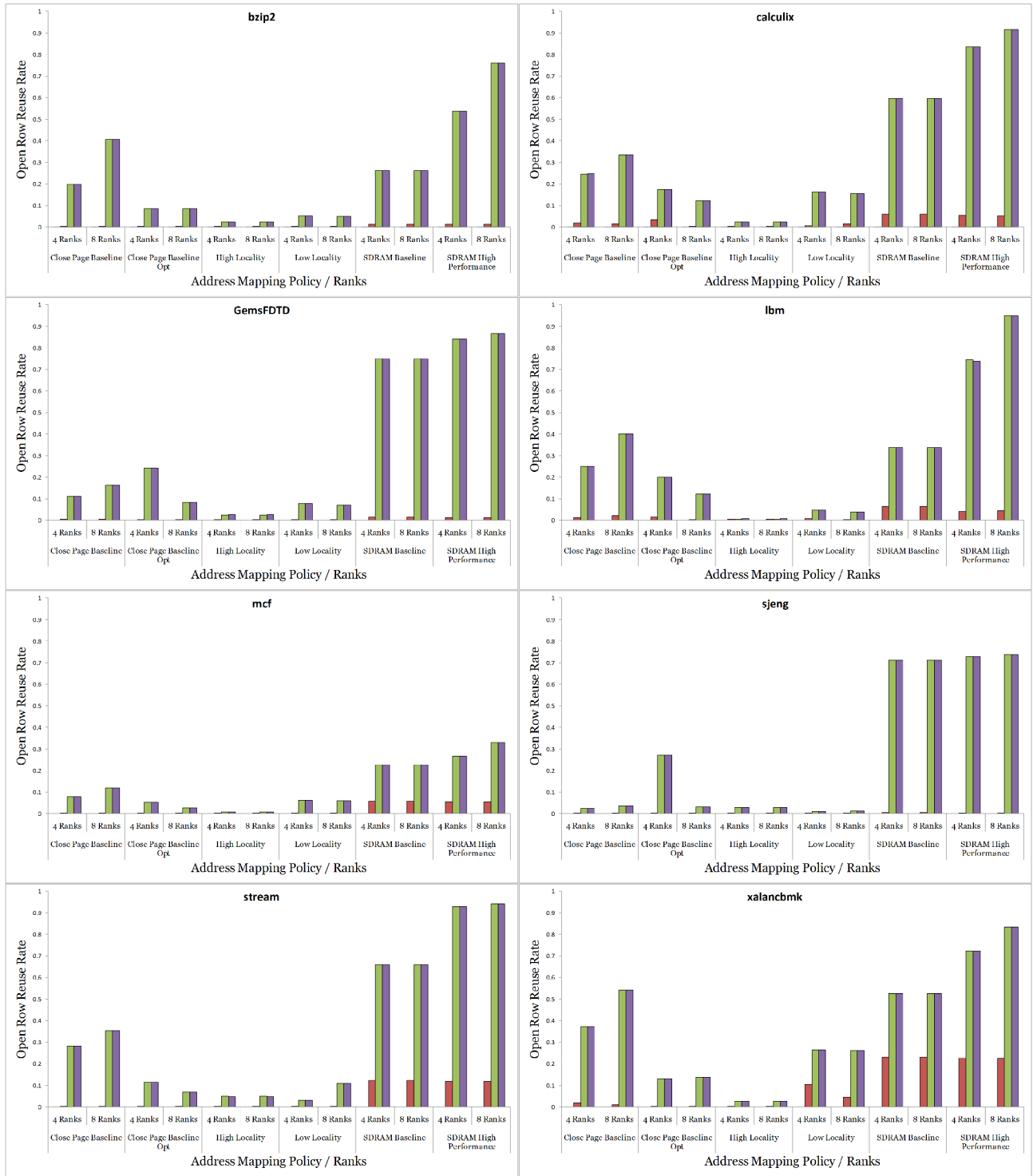
This gives a strong measure of confidence that DRAMsimII gives realistic results. The goal was to show that the simulation was not doing anything that was not possible in a real system, such as violating timing constraints. It does not prove that the simulation runs as well as commercial memory controllers, but it shows that it is playing by the same rules. In the worst case it is possible that DRAMsimII performs worse than other memory controllers and thus never violates any timing constraints. However, it is much more likely that the timing models in DRAMsimII are accurate and the constraints enforced on these simulations are realistic.

CHAPTER 7 RESULTS

To test the effectiveness of the various policies on a system, the results were run in two phases. The first was to look at the address mapping policies and the row buffer management policies while holding the command ordering algorithm constant. The second considered only two address mapping policies while testing all of the command ordering algorithms. In this way, the effects of the command ordering algorithm can be considered independently, while the address mapping policy and row buffer management policy can be considered together or independently, depending how they are analyzed. Several different metrics of performance are considered, including execution time and energy used. These are important metrics to consider because they can be directly interpreted to compare different setups. In addition to these, several other metrics are considered: average IPC, open row reuse rate, average active-standby power and average transaction latency. Although these are somewhat indirect metrics, they can be used to determine how well a given algorithm is working and why a given setup performs better than another.

We will begin by first looking at a comparison of address mapping policies and row buffer management policies. Between these two policies, the commands and locations of the transactions are determined, so this is why they are studied together. If an address mapping policy easily translates spatial and temporal locality in the transaction stream into locations on the same row but the row buffer management policy is not able to reuse the open row, then the two policies do not work effectively together.

The close page policy will never have the chance to reuse rows, but close page aggressive can reuse rows if there is enough temporal locality. Open page and open page aggressive are able to reuse rows if there is enough spatial locality.



Row Buffer Management Policy
■ Close Page ■ Close Page Aggressive ■ Open Page ■ Open Page Aggressive

Figure 7.1: The open row reuse rate as compared against various address mapping policies with either 2 or 4 ranks. Each grouping has 4 different row buffer management policies represented.

Bzip2, as shown in Figure 7.1, has reuse rates ranging from just over 1% all the way up to nearly 80%. Low locality, high locality and close page baseline optimized do not map the requests well and succeed in allowing the row buffer management policies to reuse open rows less than 10% of the time. Because close page baseline optimized moves the lower portion of the row ID to the lower order bits of the address, it tends to spread similar requests further apart, while close page baseline keeps these same requests mapped to the same row.

SDRAM baseline and SDRAM high performance tend to achieve very good performance, likely due to the fact that the column information is contained in the low order bits. If the program iterates through addresses in order, it will change the low order bits much more frequently than the higher order bits. These bits determine the column for the SDRAM baseline and SDRAM high performance. So the channel, rank, bank and row will change infrequently while the column will continue to change. SDRAM high performance likely had more row reuse due to the fact that the row used the highest order bits from the address, while SDRAM baseline used those bits to determine the rank. Since a program sequentially streaming through data arrays will change lower order bits more often, the row was changed more often for SDRAM baseline, while the rank was changed more often for SDRAM high performance, meaning that the row changed less often, allowing for more reuse.

Because the accesses were spaced far enough apart in time, the requests that would have hit in an open page system did not hit in the close page aggressive system. The effect of closing out the row immediately after the access is clearly seen in the calculix benchmark. Only about 5% of the requests in SDRAM baseline and SDRAM high performance were able to be reused.

MCF fares significantly worse than other benchmarks for all address mapping policies. In the best case, just over 30% of the accesses are able to reuse an open row. Because MCF is a scheduling algorithm to attempt to find optimal routes for mass transportation, it does not iterate through data regularly. Instead, it tends to follow pointers and walk through graphs to optimize the routes. This makes the access pattern unpredictable and erratic. Therefore, the access pattern could jump to any point in memory at any time as

pointers are followed, so it is unlikely that any address mapping policy would perform well with respect to reuse rates for MCF.

Xalancbmk has good reuse rates for open page and open page aggressive when using close page baseline, SDRAM baseline or SDRAM high performance, like the other benchmarks. However, close page aggressive has reuse rates approaching 20%. Because xalan parses text and searches for expressions and transforms a document according to a specific set of rules, it often goes over the same data several times. Because it is parsing a large document, it often has to go to memory to read and write very similar values and thus the rows are able to be reused.

Looking closer, in Figure 7.2, we can see that the majority of the row reuse happens in the early stages of the program's execution, when the files are being read and parsed. Later in the program, when the rules are being applied and the transformations are happening, the reuse rate drops to very low levels. Because so many of the requests occurred early in the execution of the program and the reuse rate was high, the average reuse rate was 0.23. However, a longer run of this program would likely fare no better than the

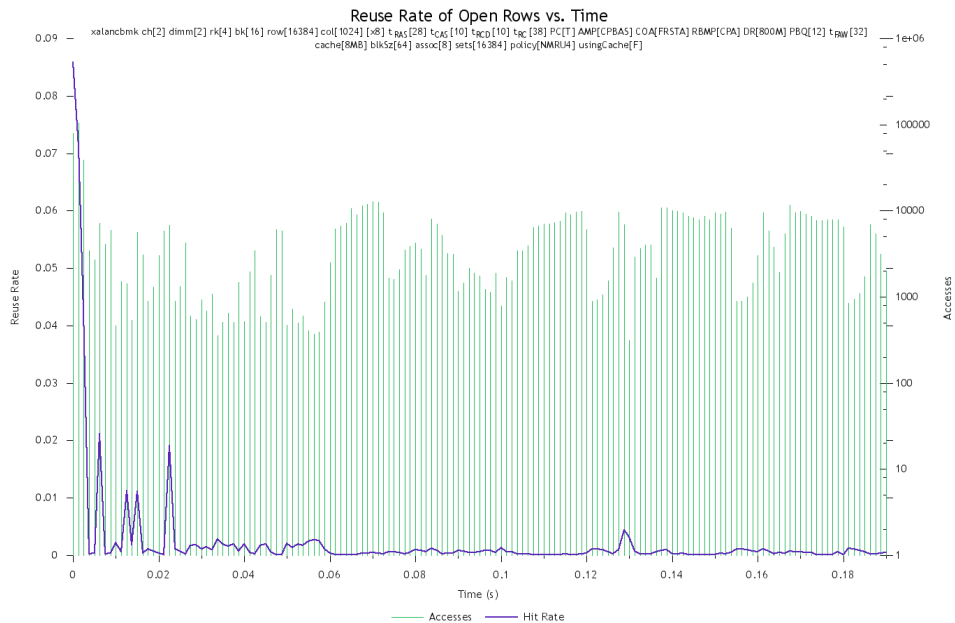


Figure 7.2: Row reuse rate vs. time for xalancbmk using close page baseline address mapping policy and close page aggressive row buffer management policy. Note that the hit rate is very low except for the very beginning of the run.

other row buffer management policies as the startup portion would remain the same length of time but the processing section, with its low reuse rates, would begin to dominate and drive down the average reuse rate.

On the other hand, if only the row buffer management policy is switched from close page aggressive to open page in order to exploit temporal locality for a greater time, we can see that the row reuse rate increases greatly in Figure 7.3. Although the average reuse rate is over 35%, this is because the initial portion of the program has many accesses and a reuse rate near 100%. Although the reuse rate varies from 5% to 40%, the average is much higher than for the close page policies, which will help to reduce average latencies and improve overall system performance.

One thing to note is that if the request stream has a lot of spatial locality and the address mapping policy can help to improve the row reuse rate. If the row buffer management policy is close page, the requests must show up in a much shorter time span to be able to reuse rows, while open page policies tend to allow much more time to reuse the rows. However, because processors are becoming faster at a greater rate than DRAMs, the requests will appear to be closer. Also, as processors incorporate more parallelism, whether by vector instructions or more parallel cores and threads, the requests with spatial locality will arrive closer in time and allow close page aggressive a better chance to reuse rows. So even though close page aggressive never had anywhere near the reuse rates of the open page policies, future systems may allow it a better chance to approach the open page reuse rates.

Also of interest is how well the address mapping policy maps transactions to their respective channel, rank, bank and row. If the row buffer management policy is an open page or close page aggressive policy, then it is advantageous to map requests with spatial locality to adjacent columns within a row so that the system can read or write that value without cycling out the current row and activating another. However, if too many requests are mapped to the same row, then there might be longer delays while waiting to get to the front of the per-bank queue than if the system had simply opened a new row to satisfy the request. It reduces latencies to reuse a row, but if there are always a dozen requests in the queue then the row reuse will be of no benefit. However, some address mapping policies will tend to scatter requests almost uniformly

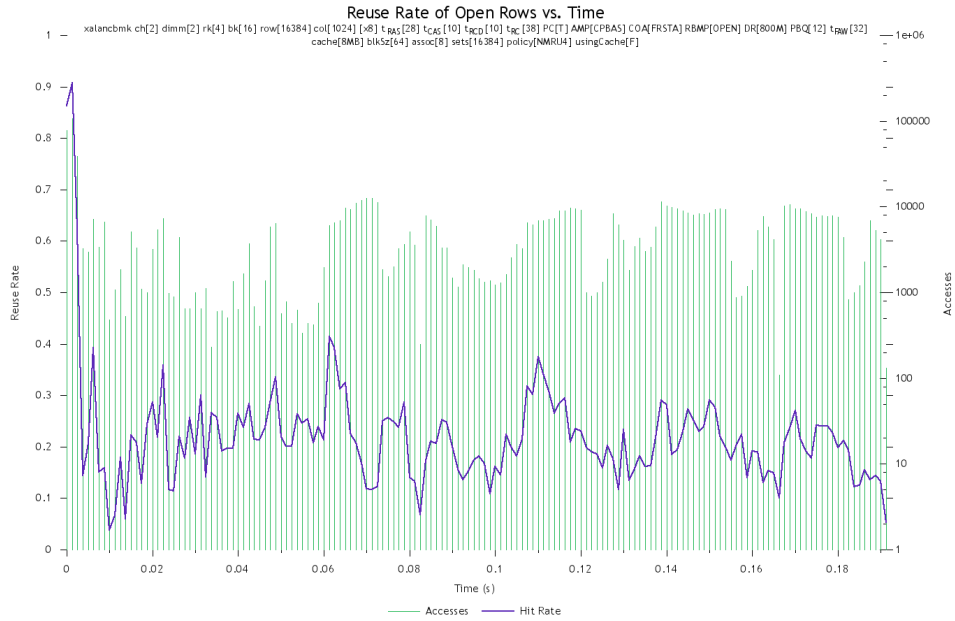


Figure 7.3: Row reuse rate for the same setup as in Figure 7.2, but with the row buffer management policy changed to open page

across the available banks and rows and not allow any potential row reuse. Although this means that there will be a minimal amount of waiting in the queue, each request must wait to open a row in the best case or precharge and open a row in the worst case.

A good address mapping policy has elements of each. It will maintain spatial locality of the stream by mapping requests to the same rows but also tend to distribute the requests across the whole memory system to exploit parallelism in the memory system. The reuse rate does not tell the whole story, as a program requiring very high bandwidth may be sitting idle most of the time while the same few rows are reused and the requests sit in queues waiting for these queues.

In Figure 7.4, the address distribution for a run of lbm using the SDRAM baseline address mapping policy is shown. Both use the open page row buffer management policy, so they should both have a good opportunity for row reuse. The only difference is the address mapping policy. Looking only at row reuse rates, the SDRAM baseline policy would seem better as it had a 34% reuse rate while the close page baseline policy, Figure 7.5, was only 25%. However, the average transaction latency for SDRAM baseline was 58 cycles while the close page baseline policy was 46 cycles. This translates to a reduction in runtime of 18%.

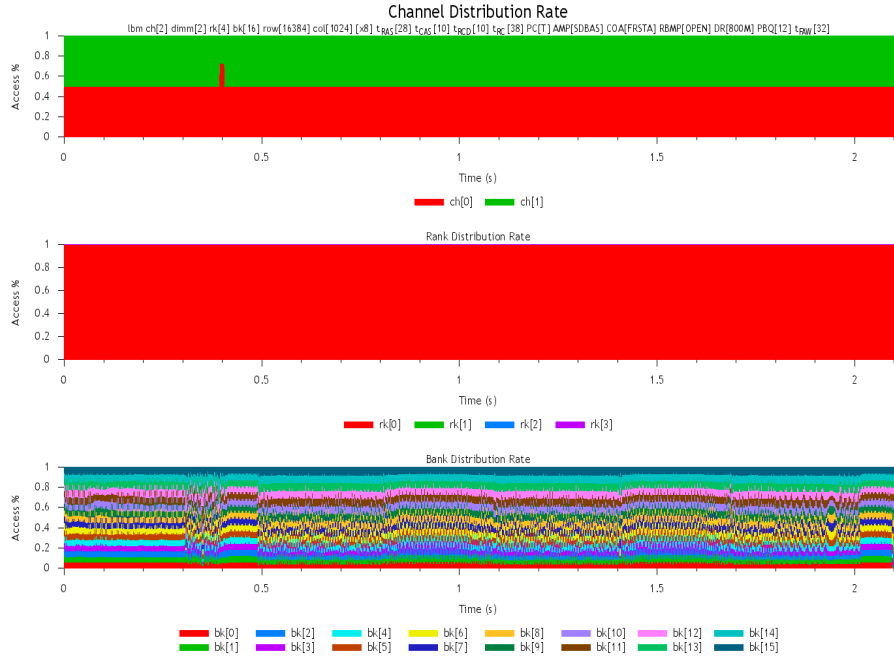


Figure 7.4: The address distribution ratio for all channels, ranks and banks of the lbm benchmark using the SDRAM baseline address mapping policy.

The difference between the two is easily shown by comparing the address distributions. The SDRAM baseline policy achieves better reuse rates by preferentially mapping requests to rank 0, while close page baseline tends to distribute the requests across all ranks equally. It also distributes the requests across the banks fairly equally as well, while SDRAM baseline tends to vary the distribution somewhat with time.

Furthermore, we can look at a histogram of the latencies of the transactions and clearly see a difference. Figure 7.6 clearly shows what the latency distribution of SDRAM baseline was while Figure 7.7 shows the distribution and latency of close page baseline. The x-axis represents the latency of the transaction, while the y-axis shows how many transactions had that latency. So Figure 7.6 shows that many transactions were returned in just over 25ns, while close page baseline shows that less than half as many transactions were returned in that time. However, each graph shows a range from 0 to 12x the standard deviation beyond the median latency, so SDRAM baseline had a deviation of almost twice that of close page baseline. Even though fewer requests were able to be made into row reuses, the increased availability of banks and ranks allowed the transactions to be returned soon after that.

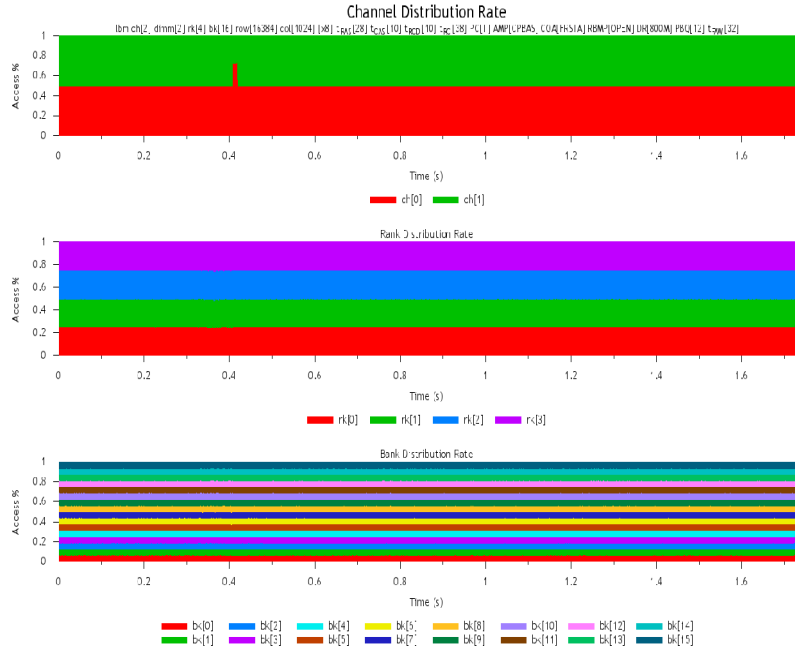


Figure 7.5: The address distribution ratio for all channels, ranks and banks of the lbm benchmark using the close page baseline address mapping policy.

The latencies for the close page baseline policy drop off sooner than SDRAM baseline, but has far more requests in the sub-50ns range. Especially notable is the number of requests after the 34ns peak for close page. The SDRAM baseline graph has very few requests that took between 34ns and 40ns, while close

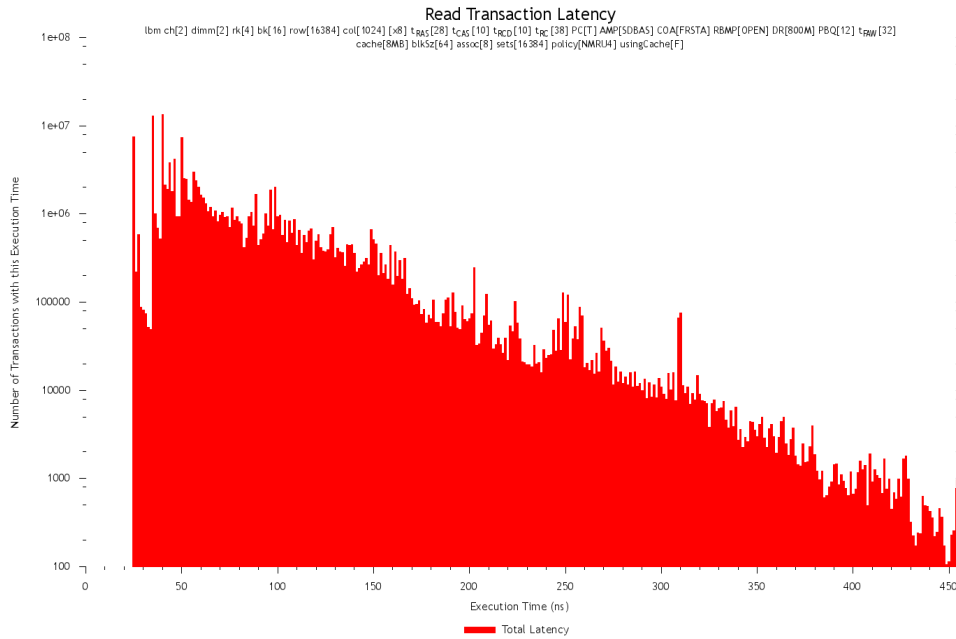


Figure 7.6: A histogram of the transaction latencies of lbm, using an address mapping policy of SDRAM baseline and an open page row buffer management policy

page baseline has several times with nearly a million requests. This is a good indication that requests either hit in an open row immediately and took only a few cycles to be returned or they waited in a queue for other requests to complete and then were returned. Close page baseline, on the other hand, had more requests that did not hit in an open row but were able to precharge and activate a new row without delay.

So when considering benchmarks like lbm, which have high bandwidth requirements, it's best to have an address mapping policy that distributes the requests across all the banks. If the address mapping policy is unable to group requests with similar requests, then there are few opportunities for row reuse and close page row buffer management policies are better because they anticipate this. Because close page policies close the pages just after they are needed, the subsequent requests do not need to wait for a row precharge to happen, they can simply activate a row and read or write from/to it.

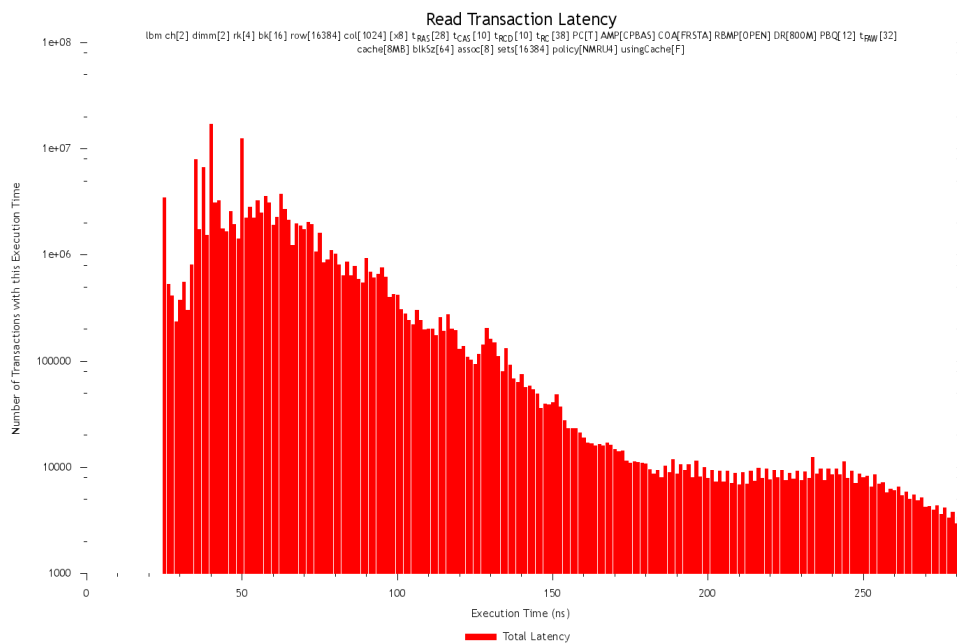


Figure 7.7: A histogram of the transaction latencies of lbm, using an address mapping policy of close page baseline and an open page row buffer management policy

Other benchmarks that are more CPU-bound, like mcf, can use fewer bytes per second since they spend more of their time doing calculations and less time waiting for transactions to return from memory. However, when benchmarks of this type do have read requests, it is important that the requests are returned

quickly because it is more likely that the overall execution time depends on the transaction latencies. Essentially, it is important for the memory controller to return the requests that are part of the critical path of the program's execution sooner.

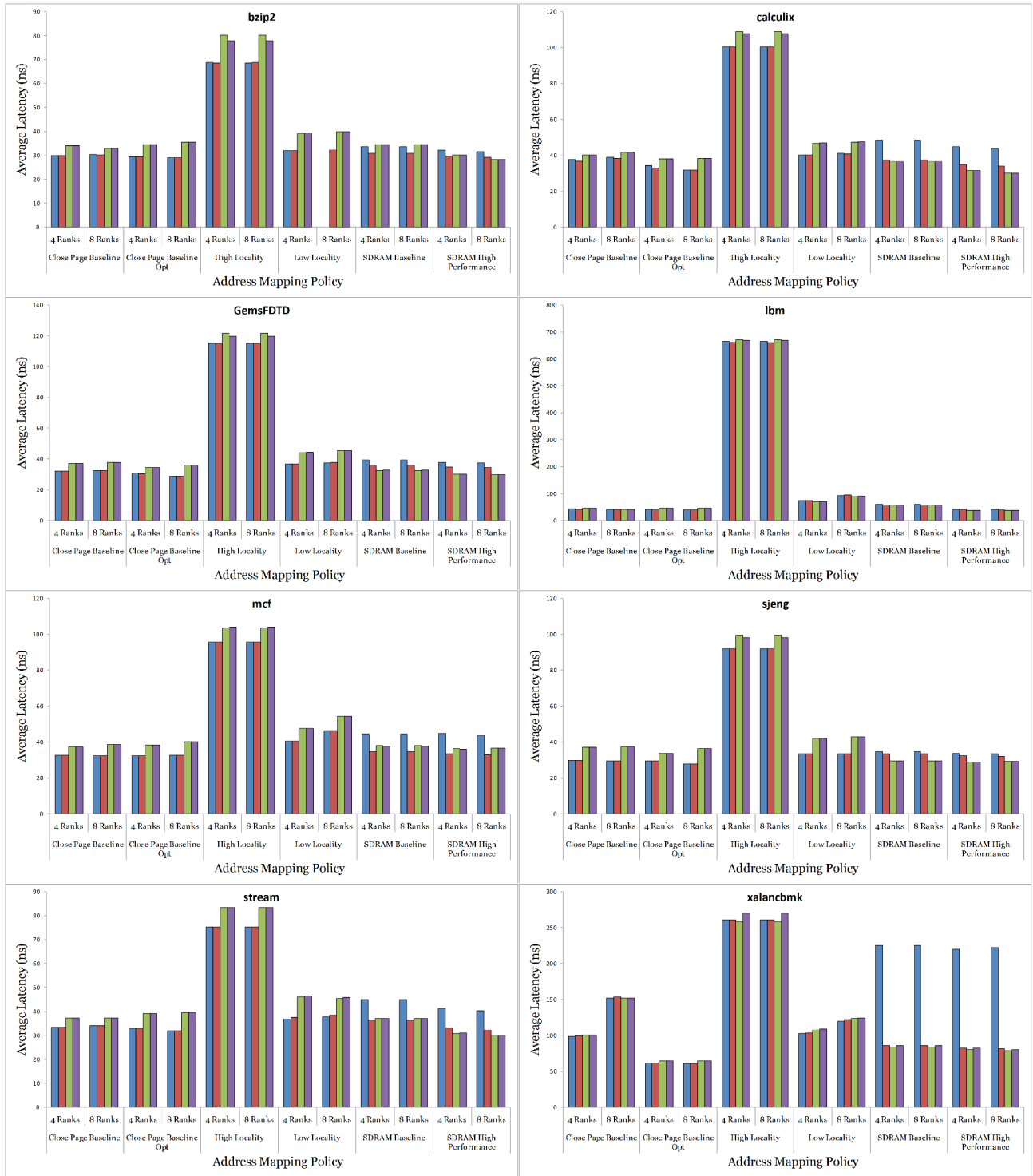
When comparing Figure 7.8 and Figure 7.1, one will notice that when the address mapping policy enabled the row buffer policy to reuse rows more frequently, the open page-based policies have lower average latencies. When the opportunities for row reuse are less abundant, then the close page policies are better. In *xalancbmk*, which has a very high row reuse rate for address mapping policies of SDRAM baseline and SDRAM high performance, the close page policy has a much higher latency than the others due to its inability to reuse open rows and exploit the latency savings. Close page aggressive, however, is able to reuse rows where close page is not and as a result has an average latency that is nearly as good as either of the open page policies. The same is true for the stream benchmark. Although close page aggressive has only a limited window of time in which to insert a request into the per bank queue, there is enough temporal locality in the stream benchmark to reuse the rows about 10% of the time. Although this is far less than the open page policies, which achieve over 70% with SDRAM baseline or SDRAM high performance, it is enough to reduce the average latency by nearly 10ns; which is nearly as good as the open page policies.

If the chosen setup has relatively low row reuse rates, as close page baseline, close page baseline opt, low and high locality do, then the close page row buffer policies achieved lower latencies. Usually the difference in latencies is not great, maybe 7ns, despite the reuse rate of 20-30%. This suggests that there is a break-even point between reusing rows and having rows precharged in anticipation of another row being opened. As seen in Figure 7.9, the bandwidth roughly correlates with the latency of each configuration. In configurations where the row reuse rate was lower, the open page policies performed worse, while the close page policies were better.

Overall, configurations that had lower latencies had higher average bandwidths. This makes sense considering that the CPU was waiting on one request before sending another. Because the memory was not completely saturated, as the system can handle several gigabytes per second, the latency dictated the

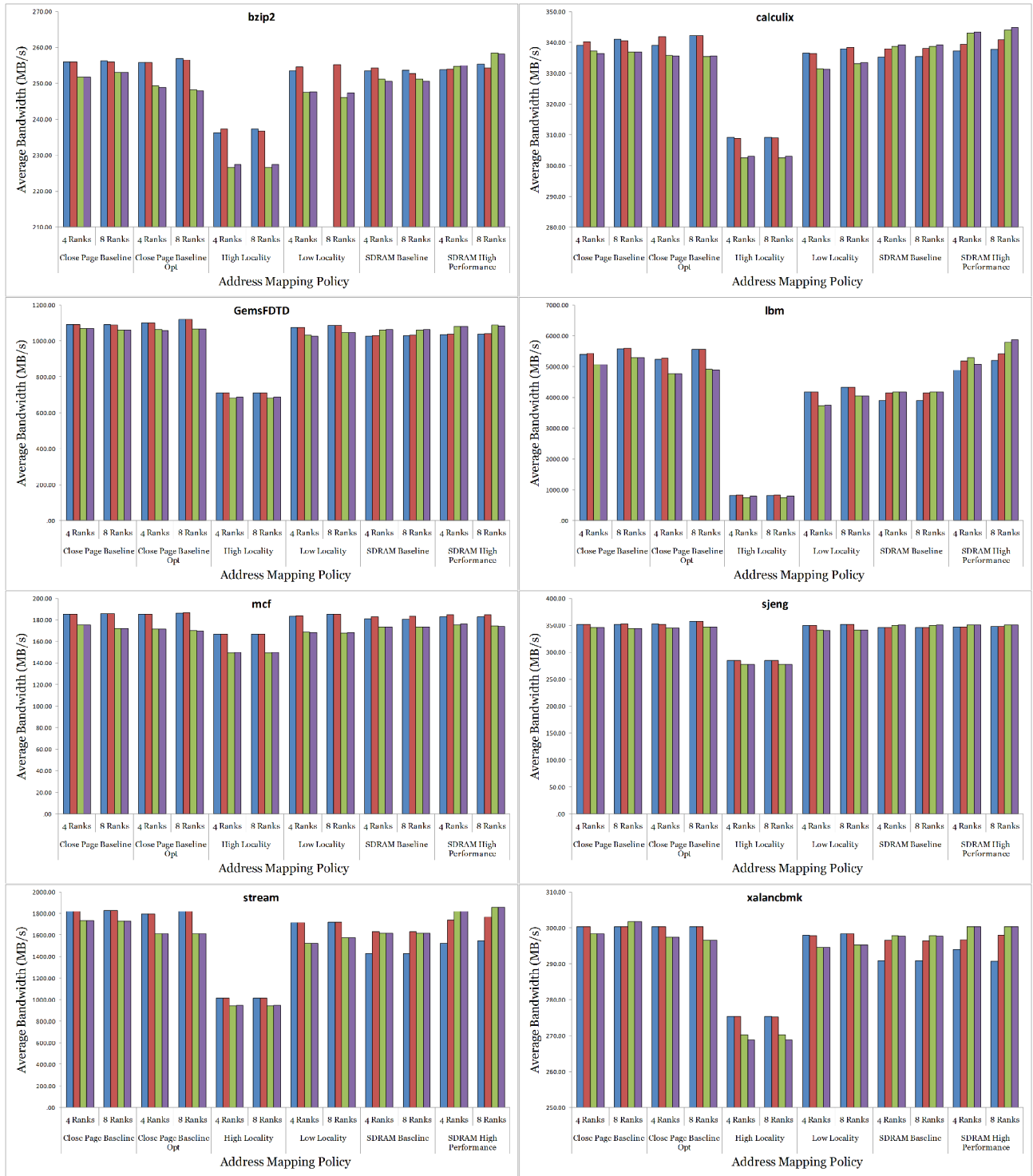
bandwidth. The high locality policy had much greater latency than the others and in turn much lower bandwidth. This is not a fundamental limitation of the memory system, but determined by the CPU serializing some requests and the memory delaying the processor.

It is interesting to note the disparity in bandwidth from one benchmark to another. While some benchmarks, like xalancbmk, can use only about 300MB/s, others, like stream or lbm, can use many gigabytes per second. Workloads vary significantly, so optimizing for bandwidth may not provide any benefit if the application cannot use it.



Row Buffer Management Policy
■ Close Page ■ Close Page Aggressive ■ Open Page ■ Open Page Aggressive

Figure 7.8: A comparison of the average transaction latencies of several benchmarks versus the address mapping policies, number of ranks and row buffer management policies



Row Buffer Management Policy

- Close Page
- Close Page Aggressive
- Open Page
- Open Page Aggressive

Figure 7.9: The average bandwidth of the benchmarks according to the address mapping policy, number of ranks and row buffer management policy

Some applications are more sensitive to bandwidth and some to transaction latency. When comparing which factors contribute to lower runtimes, by looking at Figure 7.7, Figure 7.8, and Figure 7.9, we can begin to determine which applications require more bandwidth and which perform better with lower transaction latencies. Note that these figures do not include the high latency address mapping policy, as it has significantly worse performance than the others and would make it harder to read the charts due to the increased scale. For example, the shortest runtime for lbm was 1.62s using a close page aggressive row buffer management policy and the close page baseline address mapping policy. All subsequent runtimes, in ascending order, also feature increasing bandwidths. Looking at Figure 7.8 and Figure 7.9, one can see that runtimes decrease almost linearly with increases in bandwidth. However, increasing average latencies aren't strongly linked to increases in runtime. Clearly, lower latencies correspond to lower runtimes, but when the average is above 60ns, the runtime can vary significantly, so lbm responds to increasing bandwidth but less so to reduced latency.

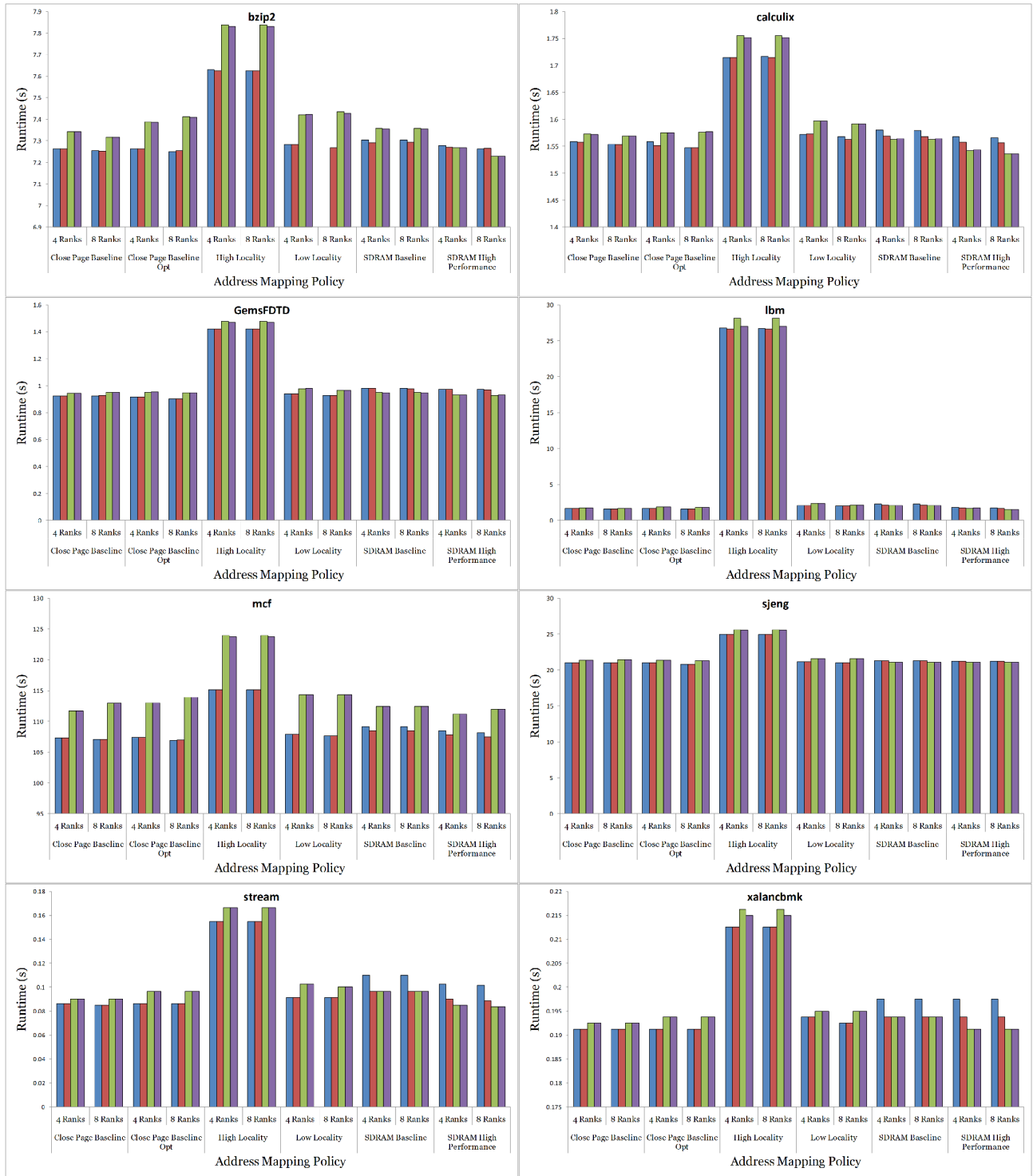
GemsFDTD, stream and sjeng also show strong correlation between bandwidth and runtime. Additionally, none of these show strong correlation between average latency and runtime. Each show a general trend of increasing runtime as the latency increases, but the correlation of runtime and bandwidth is much greater. So these applications are more sensitive to bandwidth than to latency. The correlation between latency and bandwidth means that latency and bandwidth are not necessarily independent values. If a system has an extremely long latency, then the bandwidth will suffer. Theoretically, as long as the system can deliver results at a given rate then it can sustain a given bandwidth. Realistically, if the latency is long enough, the CPU will run out of things to do in parallel after a certain amount of time spent waiting and will simply stop until the results are returned. At this point, the CPU cannot make any more requests to memory and the memory system will not have data to return and the bandwidth will drop off. So the two are certainly connected, but some applications are more sensitive to one or the other.

Bzip2, mcf and calculix correlate roughly with bandwidth but also show a fairly strong correlation with latency. None of the benchmarks exceed 350MB/s, so they are certainly not taxing the memory system's

bandwidth limits. Clearly, decreasing the latency reduces execution time, so there is less time spent waiting for read requests. If the memory is fast enough or the cache is large enough the application will be CPU-bound. This means that the memory system provides latency and bandwidth sufficient to not add a significant amount of delay to the overall execution time. If the limiting factor of an application's runtime is the memory, then the system is said to be “memory bound.” For these three benchmarks, it would appear to be somewhere between CPU bound and memory bound. There are instances for all three benchmarks where an increase in bandwidth or a reduction in latency yields no or a negative improvement in runtime.

This case is even more apparent when looking at `xalancbmk`. Although one might notice a rough correlation between bandwidth or latency and runtime, there is no clear link. It seems likely that `xalancbmk` is affected not by average bandwidth or latency, but by the latency of certain requests. If critical requests are returned sooner, such as non-speculative instruction fetches or reads, then the program will execute faster. In the case of `xalancbmk`, the program parses a file and applies rules from one file onto another file. Because of this nature, the program will be streaming data and also executing a relatively large number of instructions to apply the individual rules. The program will also have a lot of pointer chasing as it walks the document trees. Because of this, there are likely to be capacity and conflict misses for instructions. So if data and instructions are located on the same rows, instruction fetches will compete with data requests and be delayed.

If row reuse is good and the requests are sporadic enough to not cause queue congestion, execution times will be good. If the memory system mapping policy spreads requests around evenly and uses a close page policy, the instruction fetches will not have to wait so long in the queue but will take longer due to the lack of open row reuse. The address mapping policy should work in conjunction with the row buffer management policy. When it is unlikely that the row buffer management policy will reuse rows, then the address mapping policy should distribute requests so as to minimize queuing delays. If the row buffer management policy will reuse rows, requests should go to the same rows more often, but not so often as to create queuing delays. SDRAM high performance seems to do this best when used with an open page row buffer management policy.



Row Buffer Management Policy

■ Close Page
 ■ Close Page Aggressive
 ■ Open Page
 ■ Open Page Aggressive

Figure 7.10: A comparison of several benchmarks and the total execution time of each. They are grouped by row buffer management policy, number of ranks and address mapping policy.

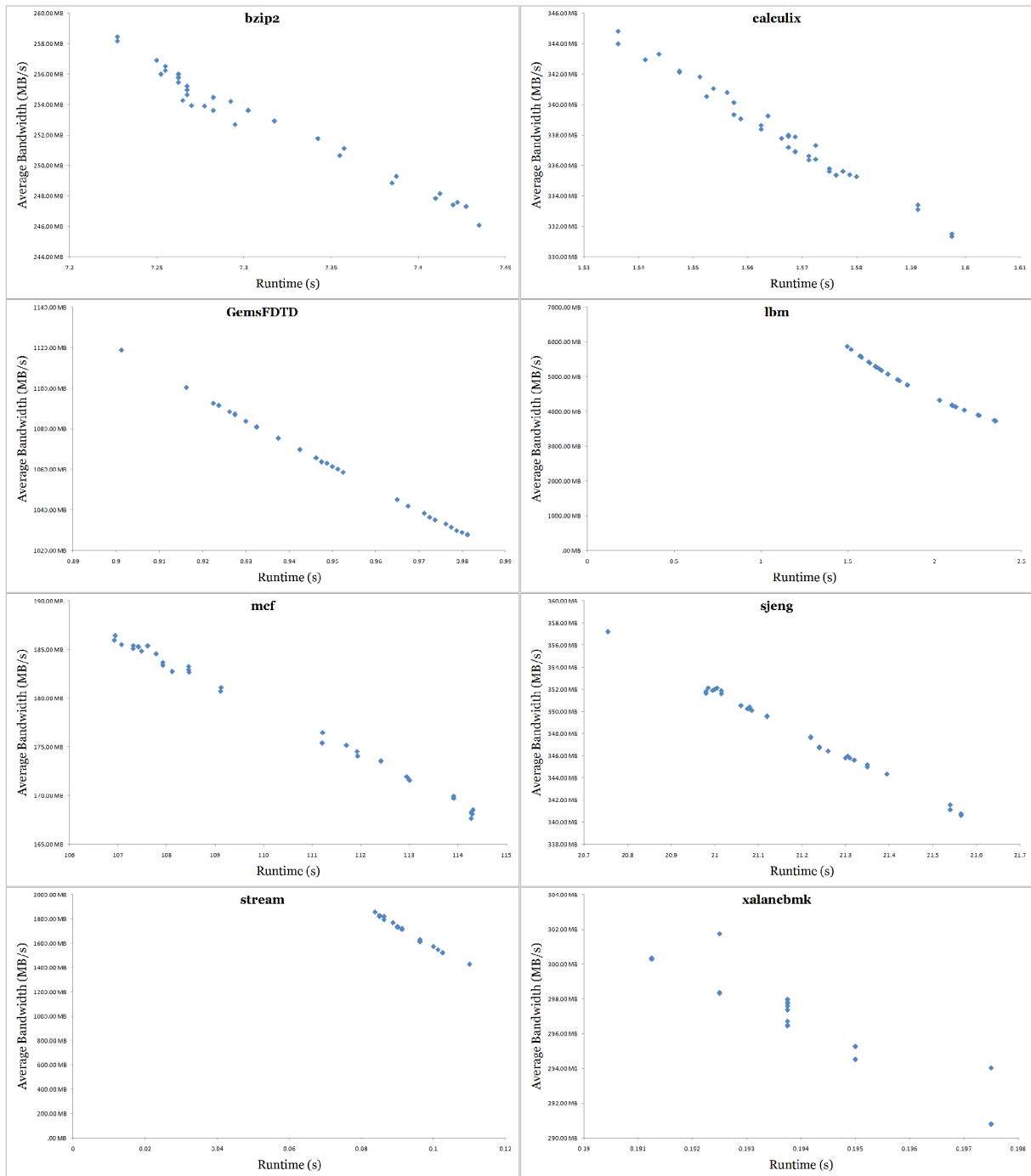


Figure 7.11: Several benchmarks comparing average bandwidth versus runtime. The points represent different configurations including number of ranks, row buffer management policy and address mapping policy. Some benchmarks certainly have shorter runtimes as the memory controller is able to provide more bandwidth, while others have less consistent behaviors.

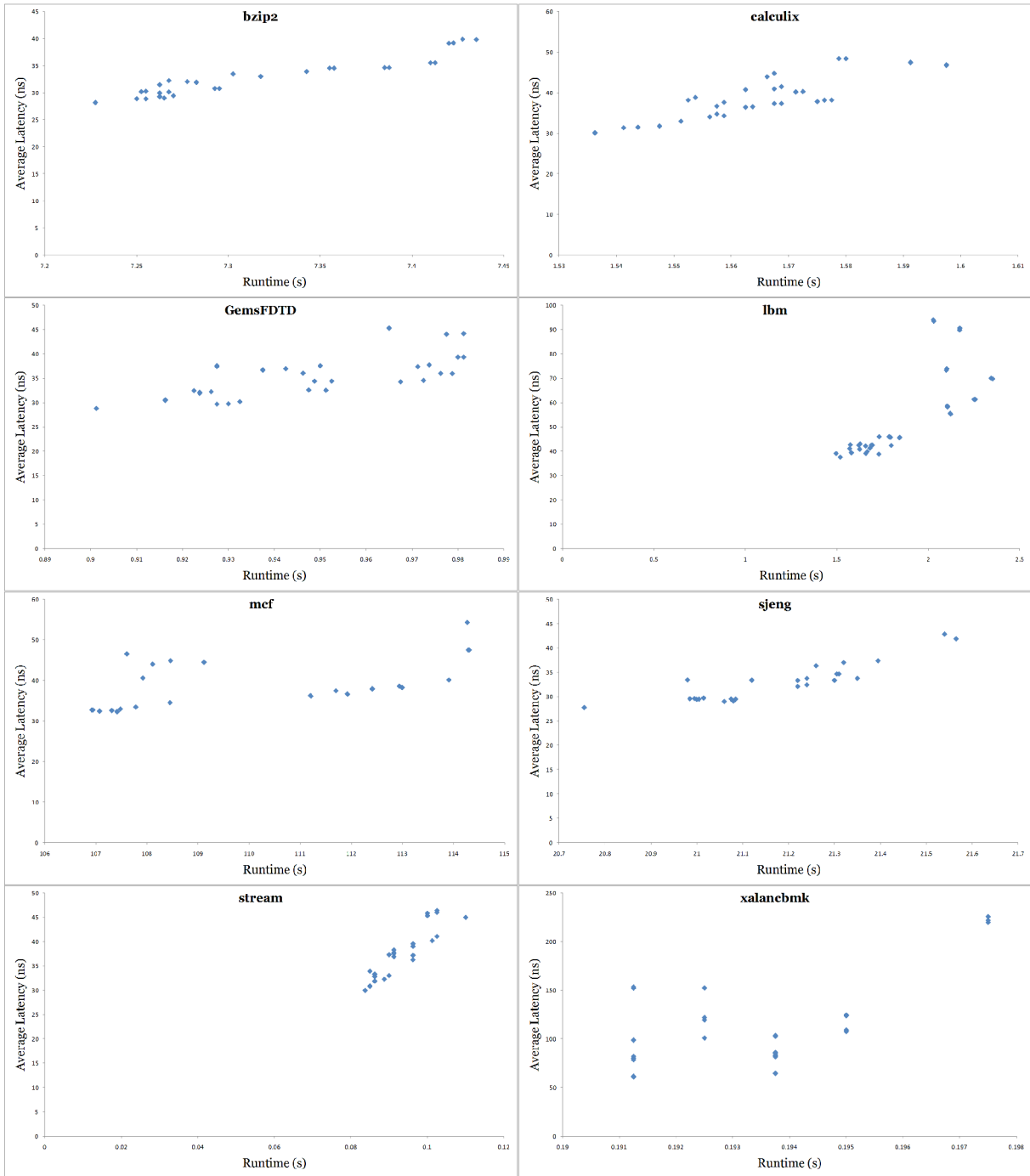


Figure 7.12: Several benchmarks comparing their runtime versus average transaction latency for different configurations involving number of ranks, row buffer management policy and address mapping policy. Note that average latency is a good predictor of relative runtime for only certain benchmarks.

7.1 Power Results

Another thing to consider is the power and energy used to run each application to completion. Using energy as a metric captures the power used by each system as well as the time it took to execute the simulation. So, if one configuration uses far more power than the rest of the simulations, but finishes in a very short period of time, it could potentially use less energy. Conversely, some configurations used only a small amount of power but took far longer to run. For example, high locality used much less power than the other address mapping schemes, but it took many times longer to finish. A convenient of capturing both the execution time and power in one value is by showing the energy used. Energy numbers are calculated by calculating the average DRAM power at each epoch and multiplying by the duration of the epoch and keeping a running total.

As shown in Figure 7.13, for a given row buffer management policy, the energy required to run 8 ranks is nearly twice as much as to run 4 ranks. Although each simulation requires the same number of reads and writes to the memory system, there are twice as many DRAMs consuming energy, regardless of how many are being used.

The power graphs illustrated in Figure 7.14, Figure 7.15 and Figure 7.16 show the breakdown of power vs. time in the 4 and 8 rank simulations. Each segment is broken down into the types of power described previously. The vast majority of the transactions took place at the beginning of the program and near the 3-second mark. Other than these two bursts, sjeng had very little traffic, so the memory system was idle. Even during the periods of high activity, the memory system was not very busy, as evidenced by the fact that there was about 2600mW of background power used by DRAMs that were idle.

Between Figure 7.14 and Figure 7.16, when going from 4 ranks to 8, power essentially doubles for the idle periods, as there are just twice as many devices in the idle state. However, during the busy periods, the power goes from about 3450mW to 5050mW, an increase of 46%. The reason for this is that the requests were able to be effectively spread over twice as many ranks. Because there were more ranks handling the

requests, this increased parallelism allowed the ranks to finish handling the requests and go back to low-power standby mode. The 4-rank system, however, will have one or more banks per rank open, thus making it difficult to get all of the banks closed and go back to standby mode.

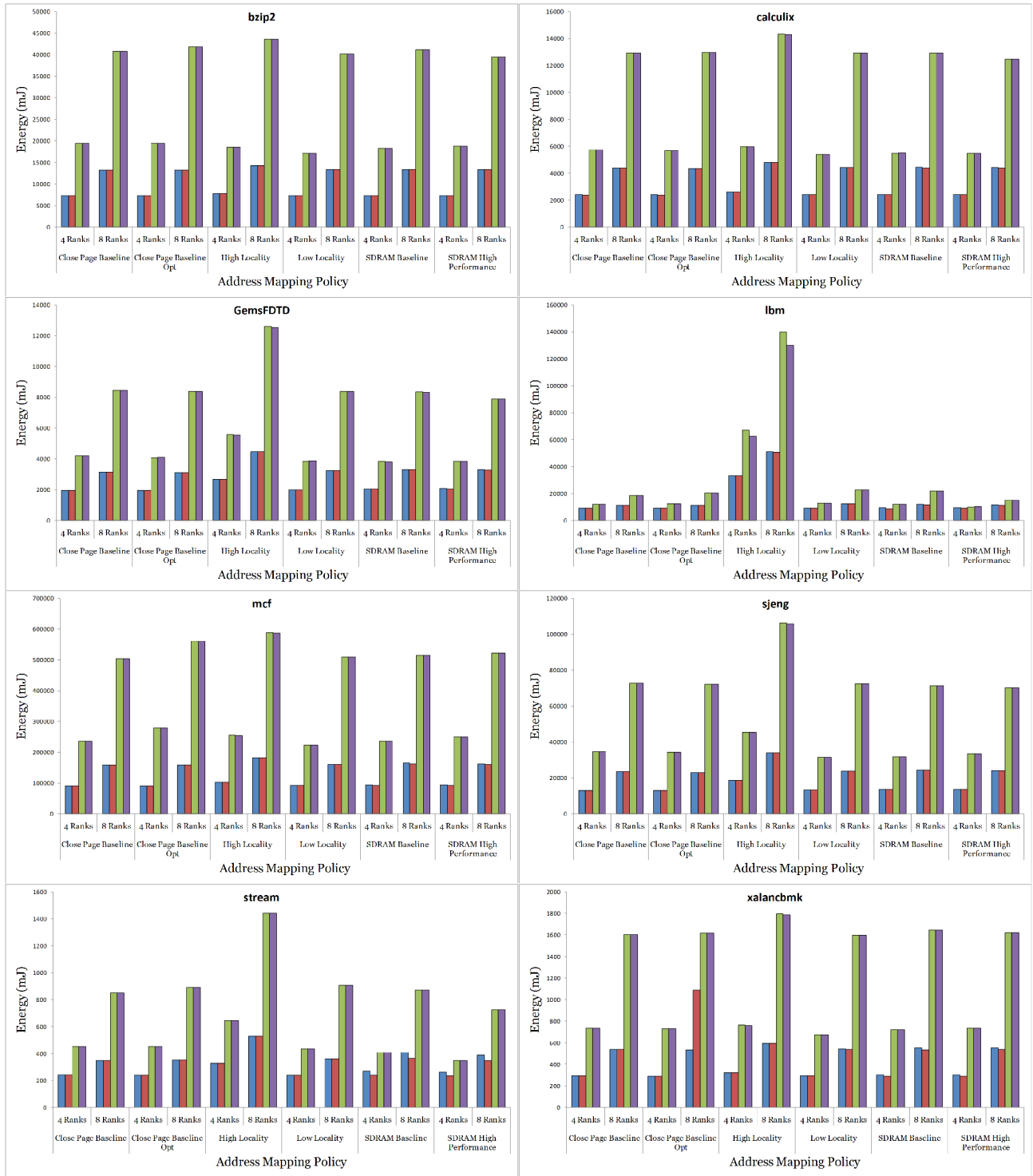
The 4-rank system has 690mW dedicated to activate-standby mode when busy, while the 8-rank system is 950mW, an per-rank average of 172mW and 119mW, respectively. So each rank in the 8-rank system is shouldering less of the load, despite the fact that the overall power usage is higher. Also the 8-rank system was able to finish this phase of the program in a shorter period of time, indicating that the additional ranks reduced the bank conflicts and improved concurrency.

When comparing the power of the two close page policies in Figure 7.14 and Figure 7.16 against the open page aggressive policy in Figure 7.15, the difference is clear. In the effort to keep rows open, the open page aggressive policy keeps the rows in higher-power states at all times. This strategy yields good row reuse rates, which translates to some of the best average transaction latencies and execution times. However, keeping those rows open keeps the rows in the active state, which uses much more power. So while the close page policy takes slightly over 2700mW during the idle phases with 8 ranks, the open page policy takes over 8200mW while idle, over 300% greater. Activation power is reduced slightly as there are fewer row activations when the rows are reused. Unfortunately, since the memory system is not heavily utilized, reductions the activation power are canceled out by the increased background power.

The major disadvantage of open page versus close page when comparing power comes down to the characteristics of the DRAM devices themselves. In this simulation, the DRAM devices were modeled after a Micron 2Gb part running at 800MHz. When at least one bank in the rank is active, the devices all consume 70mA or 50mA, depending whether the device is in standby or powerdown mode. When all banks are precharged, the devices all consume 60mA or 30mA (for fast exit), depending whether they are in standby or powerdown mode. Additionally, if the devices are set to slow exit, the devices will consume only 12mA. Because close page tends to keep the DRAM devices precharged and open page keeps the DRAM devices active, the power will almost always be greater for open page. Assuming the memory controller efficiently

uses CKE to put the devices in a low-power state when they are idle, the open page policy still consumes 20mA more per device. If the memory controller has the logic to use slow exit, then the disparity grows to 38mA.

When the ranks are in active mode, the memory controller knows that a request could come along at any time and is preparing for a row reuse. So it is less likely that the memory controller would want to put the system into the powerdown state. The Micron datasheets estimate that the memory controller would keep the DRAMs in standby state over 90% of the time. Conversely, when all the banks are precharged, it is estimated that an activate could come along at any time, but there will not be any immediate use of the row buffers. So the memory controller does not have any need to keep the DRAMs perpetually in standby. The examples estimate that the DRAMs can be left in the powerdown state 99% of the time. If the memory controller can more aggressively use CKE when using an open page policy, then the gap will be reduced.



Row Buffer Management Policy

■ Close Page ■ Close Page Aggressive ■ Open Page ■ Open Page Aggressive

Figure 7.13: Various benchmarks and the impact that address mapping policy, row buffer management policy and number of ranks affect the energy used in the run.

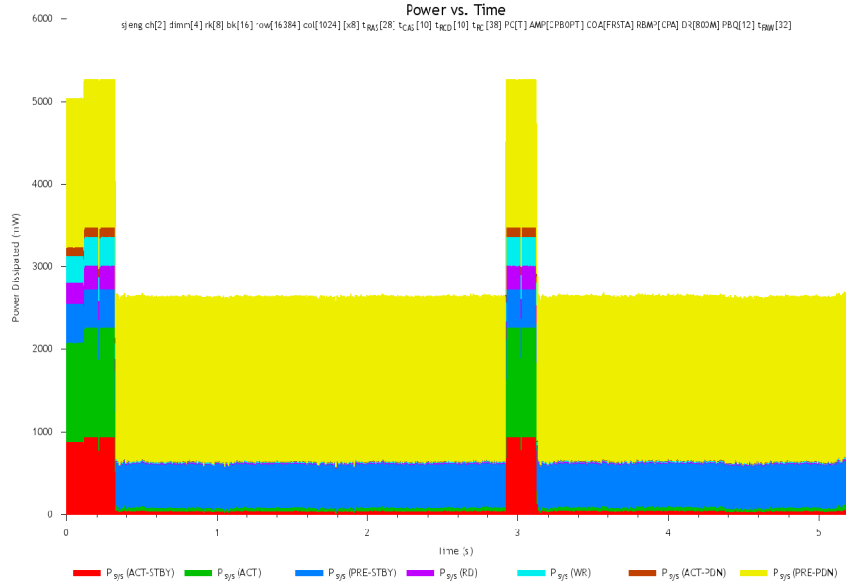


Figure 7.14: Power vs. time for *sjeng* with 8 ranks across two channels, using the close page optimized address mapping policy, and close page aggressive row buffer policy. Note that when active, the power usage is less than twice, but when idle the power is nearly twice.

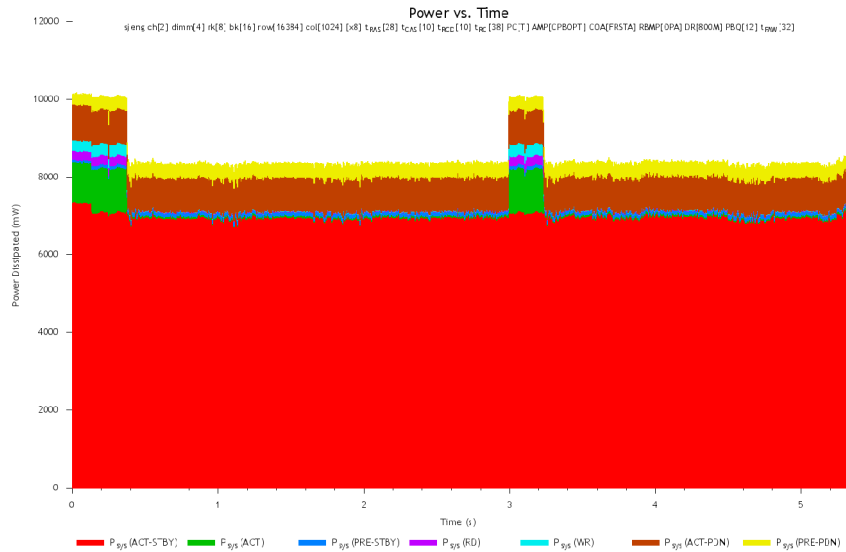


Figure 7.15: The same configuration as Figure 7.14, but using open page aggressive row buffer policy instead. Far more power is consumed while in the active-standby state.

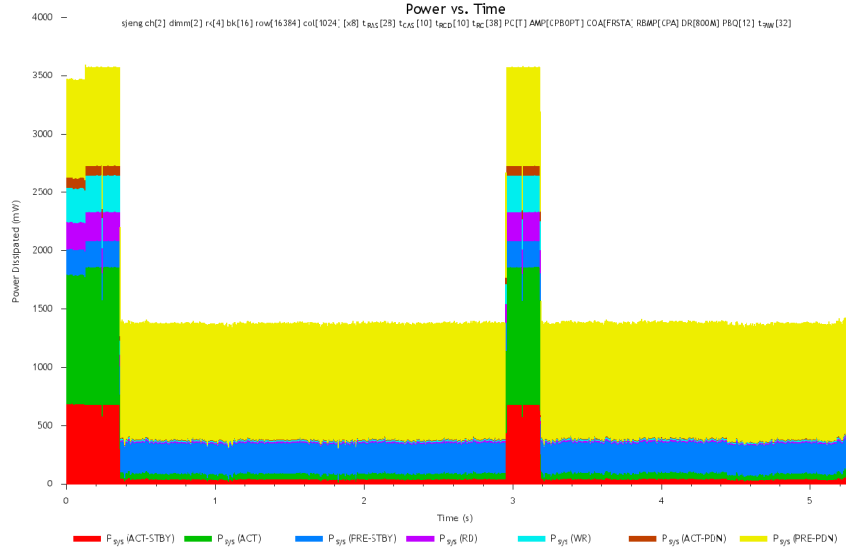


Figure 7.16: The same configuration as Figure 7.14, but with half as many ranks (4). The background power values are halved, but the activate, read and write powers remain the same.

7.2 Detailed Power Comparison – LBM

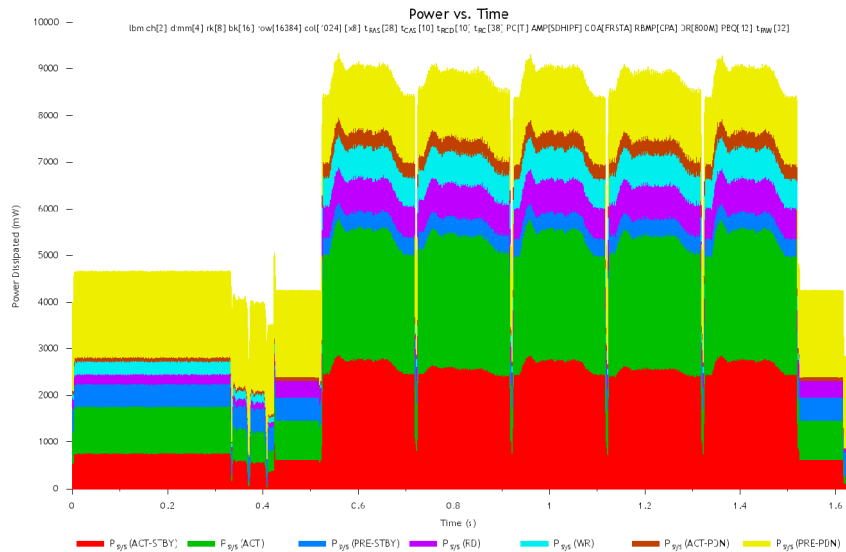


Figure 7.17: A run of lbm, using 4 ranks, SDRAM high performance address mapping policy and a close page aggressive row buffer management policy.

Figure 7.17 Shows that lbm is a much more memory-intensive application than others previously characterized . Although it is not completely saturated, as evidenced by the amount of power used for the DRAMs in precharge-standby, this memory system consumes much more power. The activation power is 20-30% of the total power, indicating that many rows are being opened because the reuse rate is not very good;

5% in this case. Five time steps simulated, which show up as the periods of increased activity. This is a relatively short simulation, so thousands of time steps would be simulated on actual hardware. The duration of the time steps would be very important and would take the majority of the execution time.

Comparing the same simulations of open page aggressive and close page aggressive, one can see that open page aggressive uses the majority of its power for keeping the DRAMs in the active-standby state.

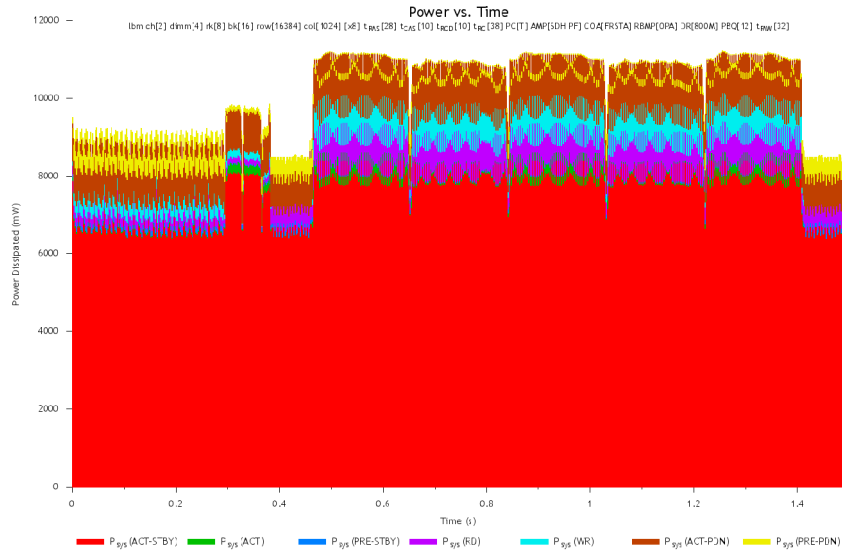


Figure 7.18: A run of lbm, using 4 ranks, SDRAM high performance address mapping policy and a open page aggressive row buffer management policy.

Activation power is minimal, due to the fact that the row reuse rate is 95%. However, for each time step, the average power for the open page aggressive policy is around 11000mW, while the close page policy is only about 9000mW. The tradeoff is an increase in power of over 20% and a runtime reduction of 9% (from 1.62 to 1.49). Each processing step is reduced from .19s to .18s, a decrease of less than 5%. This slight advantage could be useful if there are many thousands of time steps to calculate. However, the designer must know how important power usage is before choosing the row buffer management policy and an address mapping policy. For each DRAM, according to the Micron -187E speed bin, running at 1066MT/s, the active-standby uses 80mA and precharge-standby uses 70mA. Active-powerdown uses 55mA and precharge-powerdown uses 35mA. If slow exit is used then precharge-powerdown is a mere 12mA. Clearly, any policy that can reduce

the current draw for each DRAM by 20mA (active-powerdown to precharge-powerdown, fast exit) or 43mA (active-powerdown to precharge-powerdown, slow exit) will be beneficial in terms of power usage.

7.3 Command Ordering Algorithm Performance

The command ordering algorithm determines the order that commands are sent to the DRAMs, which can determine the maximum system bandwidth. A good command ordering algorithm will choose commands in such a way as to overlap operations, reduce latency, especially for transactions that are critical to the execution of the application, avoid starvation of lower priority commands, execute refresh commands without interruptions and keep the per bank queues from getting full.

Figure 7.19 shows that there is no optimal algorithm for all benchmarks and configurations. For bzip2, the strict ordering policy affords the highest average IPC except when using close page baseline with 4 ranks, where it is among the worst. Bank round robin is the best for bzip2 when using close page baseline opt. For most benchmarks, the first available policies give the best IPC, while bank round robin are next and strict is worst. The order remains the same in Figure 7.20, with the first available policies exhibiting the best latencies. Most often, first available (age) has the lowest latencies. Choosing the oldest command helps to choose the longest-queued commands and return them sooner. Most of the policies have lower latencies than the strict policy, except command pair rank hop. Because strict is considered the baseline, this shows that there is improvement over the typical default implementation.

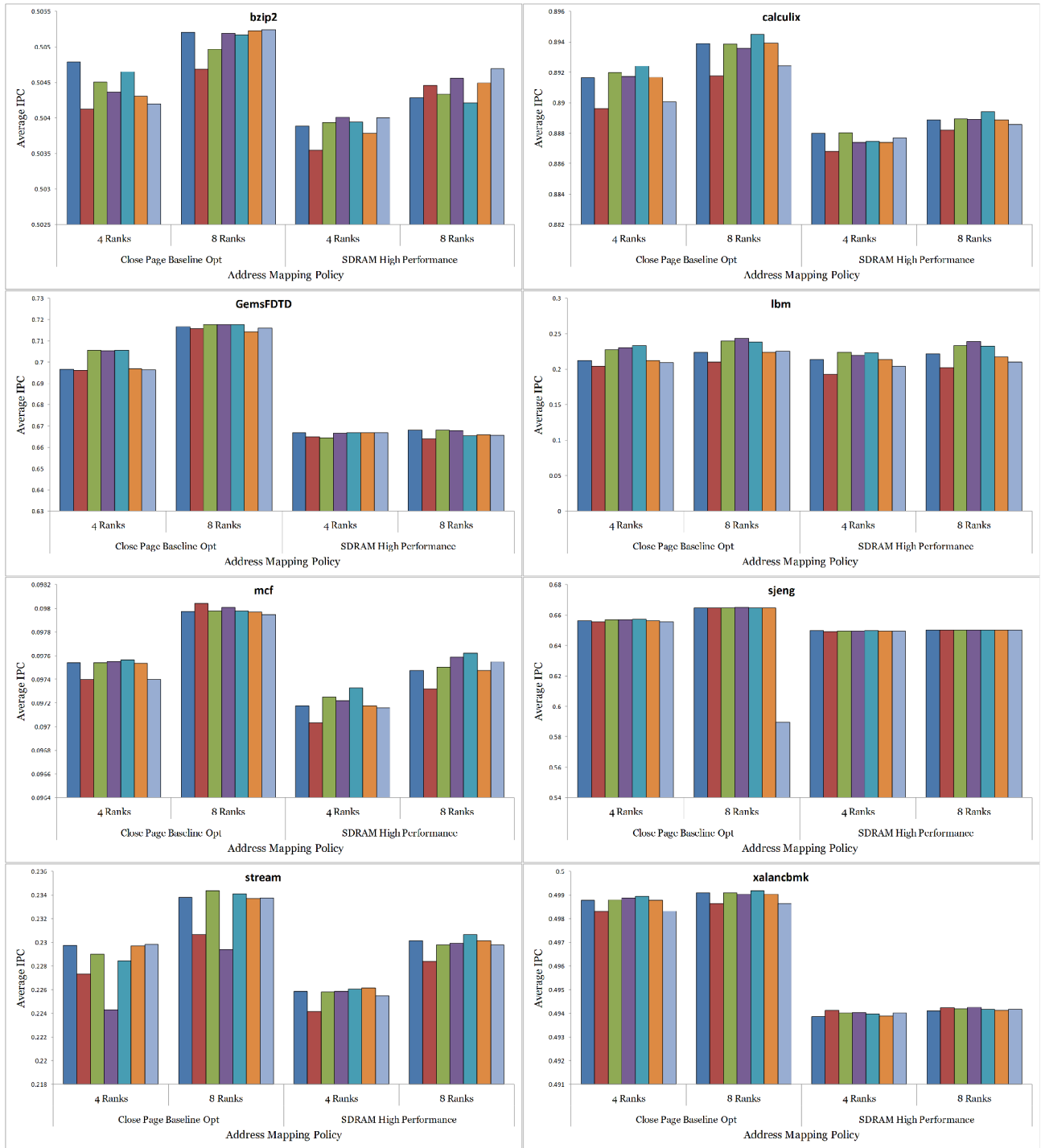
These advantages for the first available policies are evident in the execution times listed in Figure 7.22. Close behind the first available policies is bank round robin and sometimes rank round robin. So the choice of command ordering algorithm involves a tradeoff. The first available policies are resource intensive and require the memory controller to keep track of many items, evaluating every per bank queue at every cycle. Bank round robin is very simple and follows a prescribed pattern, but must evaluate the banks in order. Increased memory controller resource usage and complexity will yield performance improvements, so a designer must decide which is more important.

7.3.1 Saturation Mode

To really see the difference between the command ordering algorithms on an enterprise machine, the per-queue banks should be full. To accomplish this, random transactions were set to arrive every 3 cycles on average, fast enough to ensure the queues would quickly fill. This is quite similar to a large server with dozens of active processes. Consider a web server handling many clients simultaneously. Each of the many simultaneous requests data from a different portion of the database that backs the web server, so there are many unrelated transactions to and from memory at any given time. Any massively parallel data mining or information processing process is likely to have similar characteristics: using all of the banks all of the time with minimal data overlap; i.e. row reuse. In these situations, it is useful to know which command ordering algorithms tend to do well to maximize throughput.

Figure 7.23 shows that first available (age) and first available (RIFF) consistently outperformed all other algorithms. Bank round robin was quite good in most cases, but suffered when used in conjunction with open page or open page aggressive row buffer management policies. Rank round robin also performed quite well in most cases but also suffered when using either open page policy. It is likely that the additional latency from having to first precharge and then open a new row did not work well with the round-robin pattern.

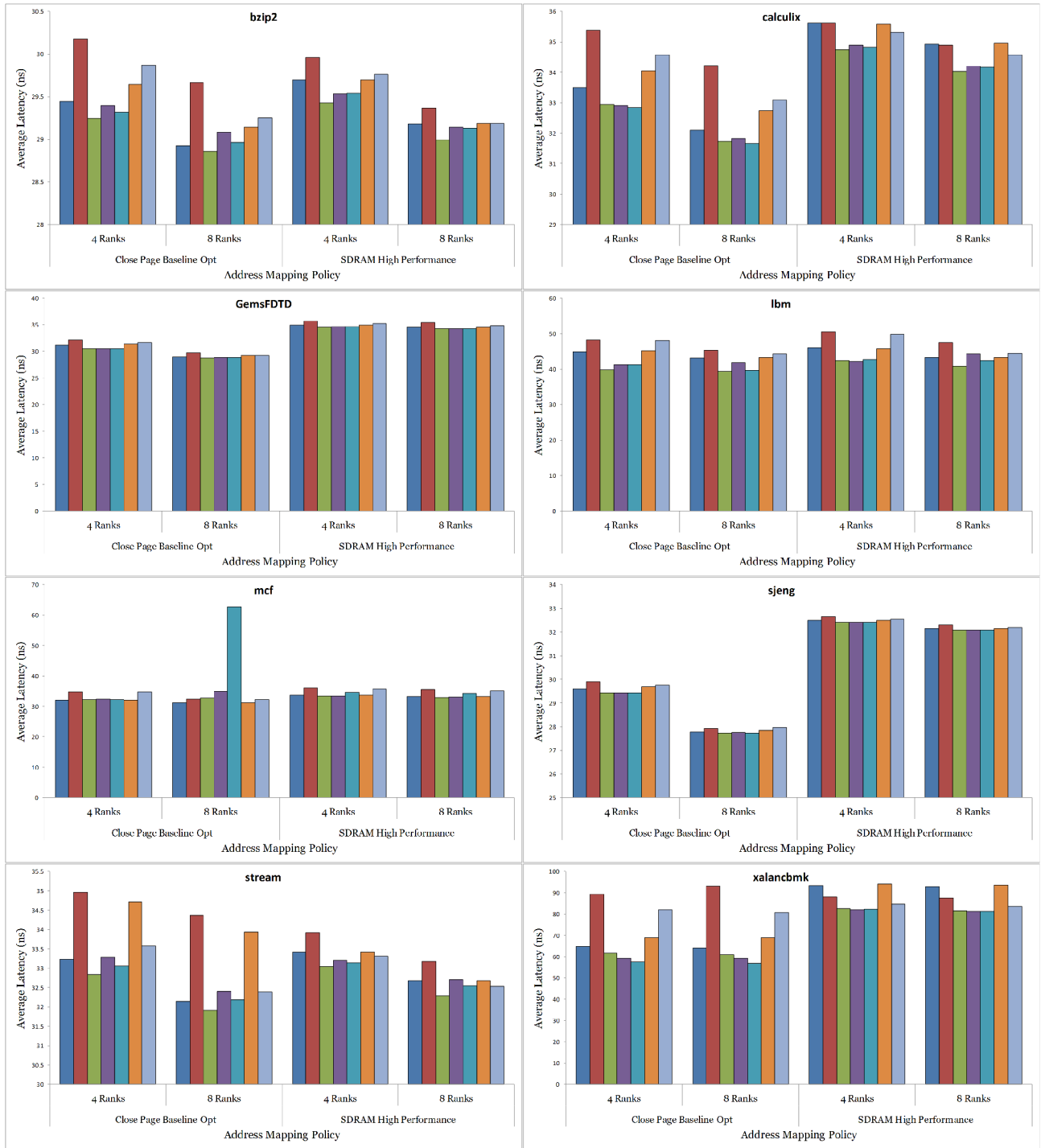
Command pair rank hop was consistently good, although not as good as the best algorithms. For scenarios where memory controller resources are limited and all of the parameters required to implement a first available policy are too much, command pair rank hop will provide consistent performance for a variety of setups. First available (queue) lags behind the other first available policies in all configurations and this is likely because the queues are consistently full. If the queues are full, the secondary criterion of queue utilization provides no help in deciding which queue to use. So essentially, first available (queue) tends to find all queues which can execute a command and then chooses randomly among them.



Command Ordering Algorithm

- Bank Round Robin
- Command Pair Rank Hop
- First Available (Age)
- First Available (Queue)
- First Available (RIFF)
- Rank Round Robin
- Strict

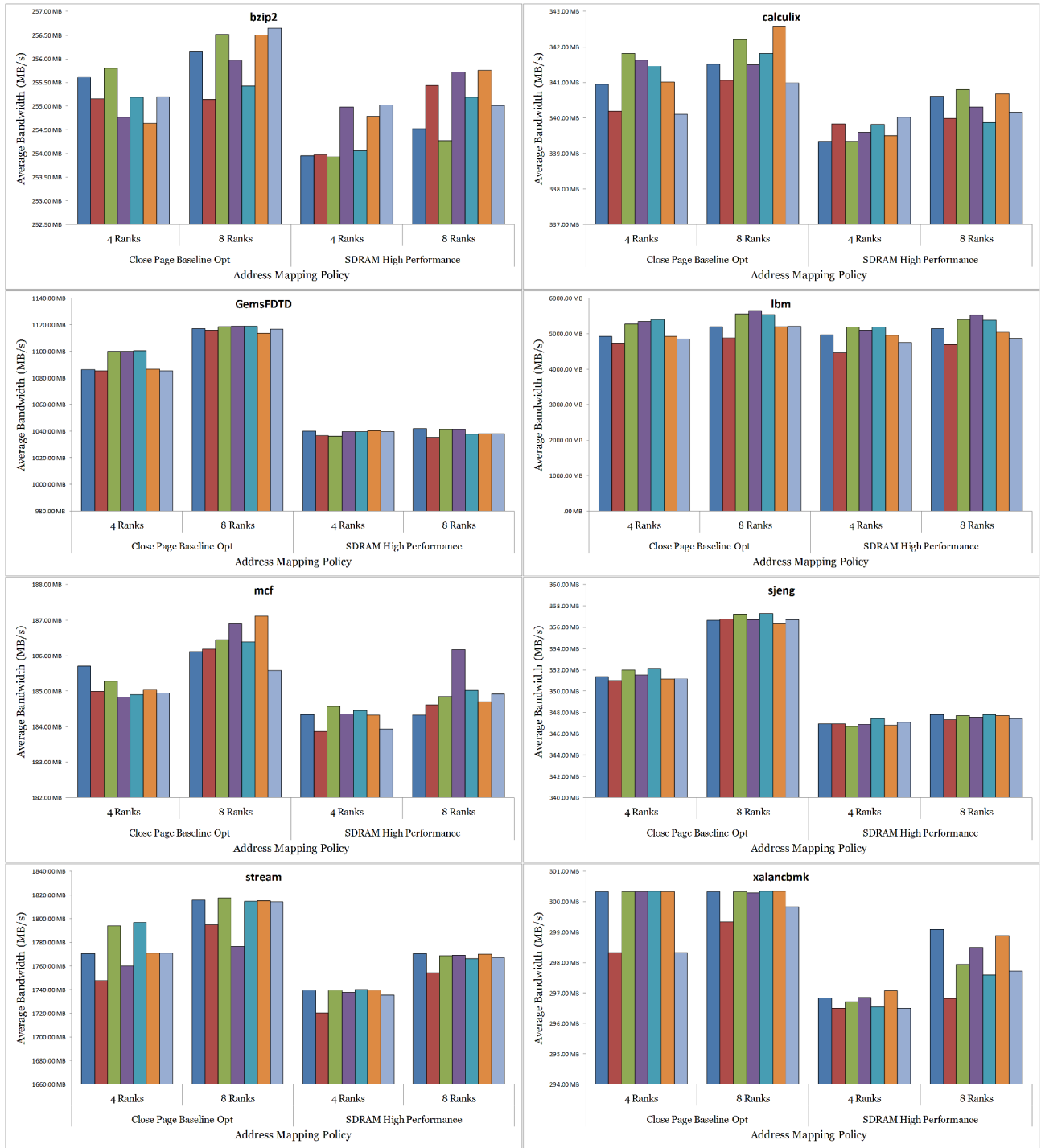
Figure 7.19: The average IPC for the various benchmarks when choosing different address mapping policies, number of ranks and command ordering algorithms.



Command Ordering Algorithm

- Bank Round Robin
- Command Pair Rank Hop
- First Available (Age)
- First Available (Queue)
- First Available (RIFF)
- Rank Round Robin
- Strict

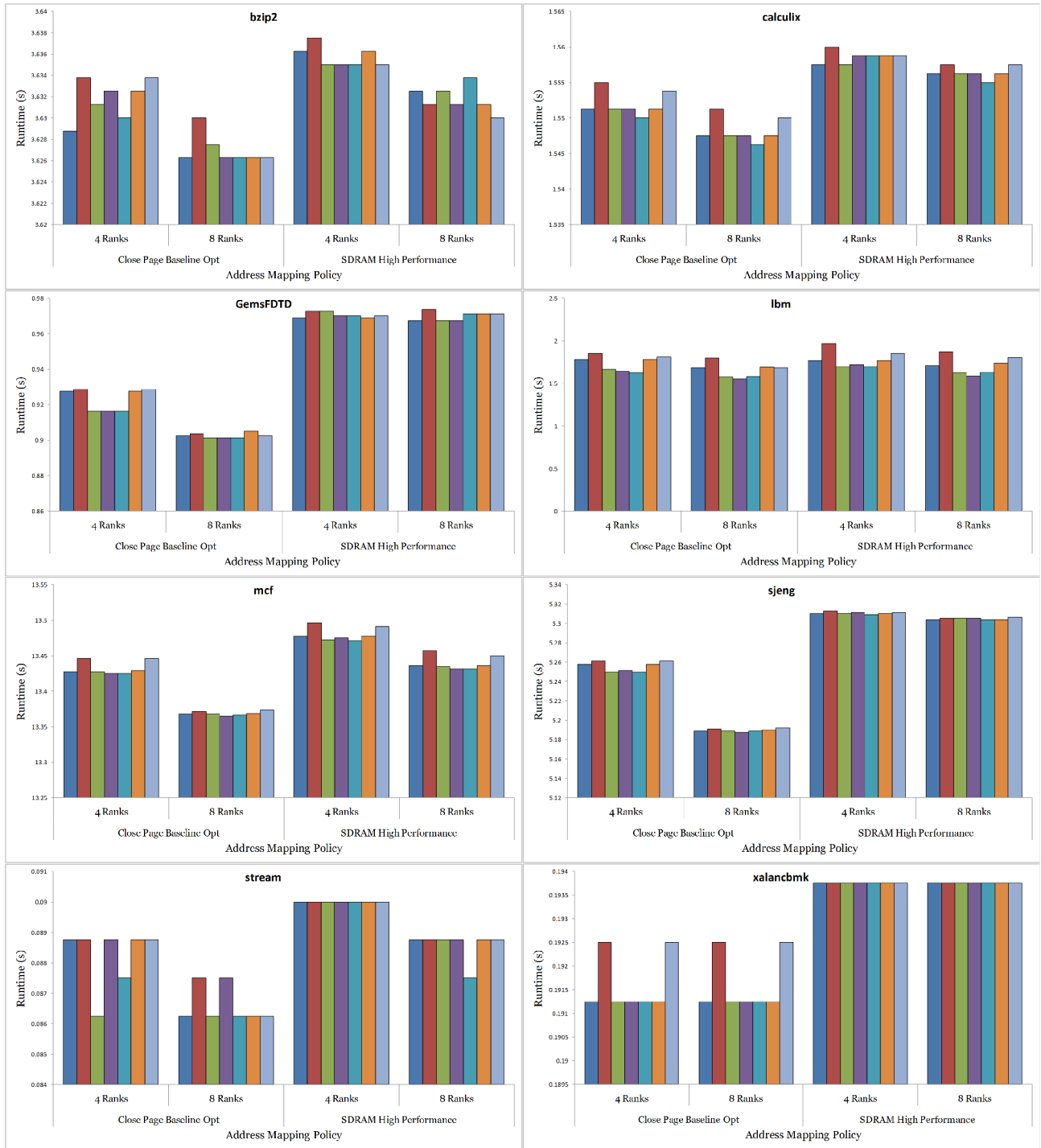
Figure 7.20: The average latency for transactions for various benchmarks when choosing varying numbers of ranks, address mapping policy or command ordering algorithm.



Command Ordering Algorithm

- Bank Round Robin
- Command Pair Rank Hop
- First Available (Age)
- First Available (Queue)
- First Available (RIFF)
- Rank Round Robin
- Strict

Figure 7.21: These are the average bandwidths for the benchmarks while varying the command ordering algorithm, address mapping policy and number of ranks.



Command Ordering Algorithm

- Bank Round Robin
- Command Pair Rank Hop
- First Available (Age)
- First Available (Queue)
- First Available (RIFF)
- Rank Round Robin
- Strict

Figure 7.22: The runtimes for the various benchmarks versus the command ordering algorithm, address mapping policy and number of ranks.

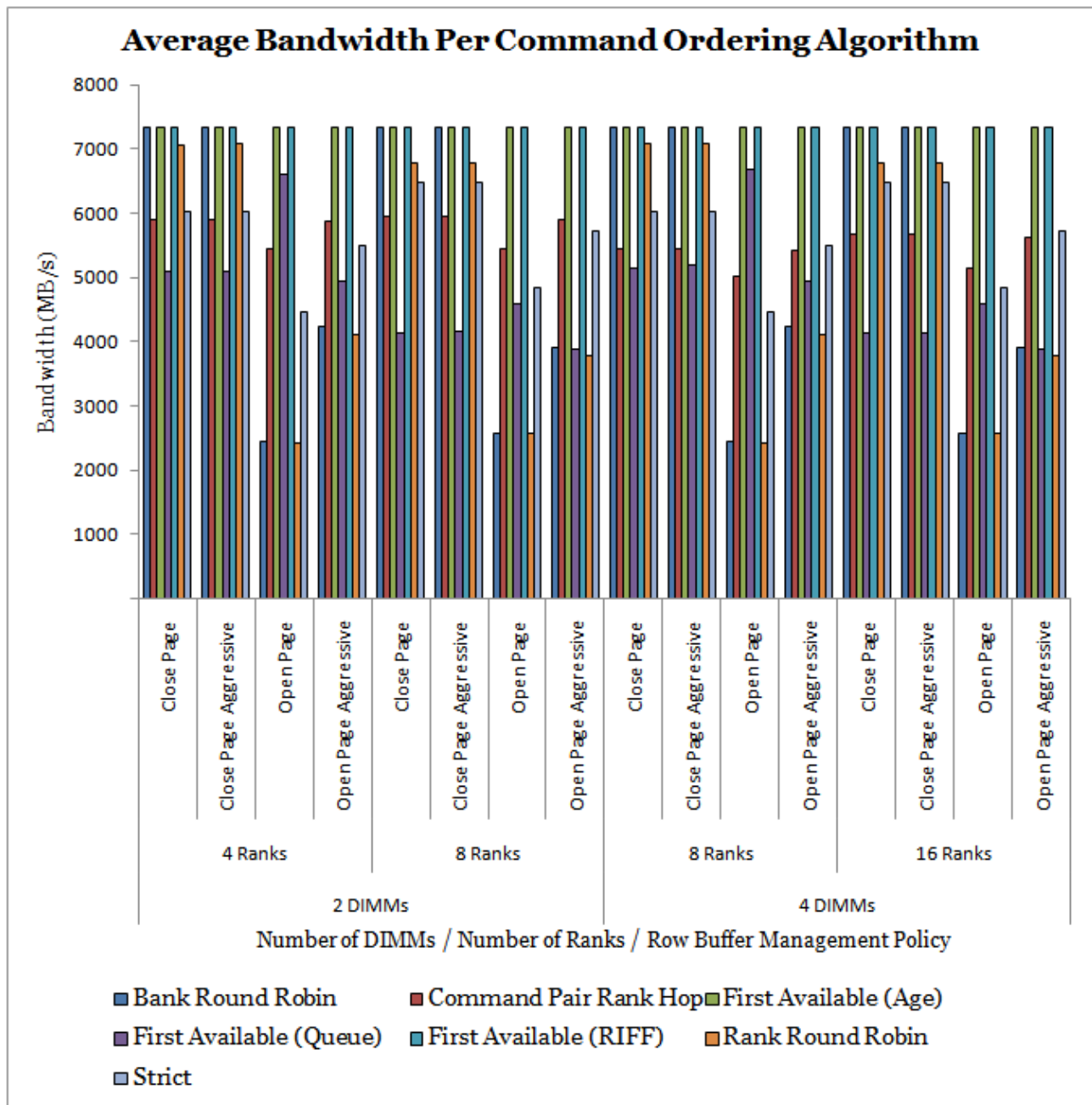


Figure 7.23: A comparison of the average bandwidth maintained using different command ordering algorithms along with the different row buffer management policies and different numbers of ranks and DIMMs.

The strict policy shows how the performance of a FIFO and actually does quite well. Often, strict provides 60-80% the bandwidth of the best available algorithm, so in a memory controller that is critically limited for resources, the strict policy would suffice.

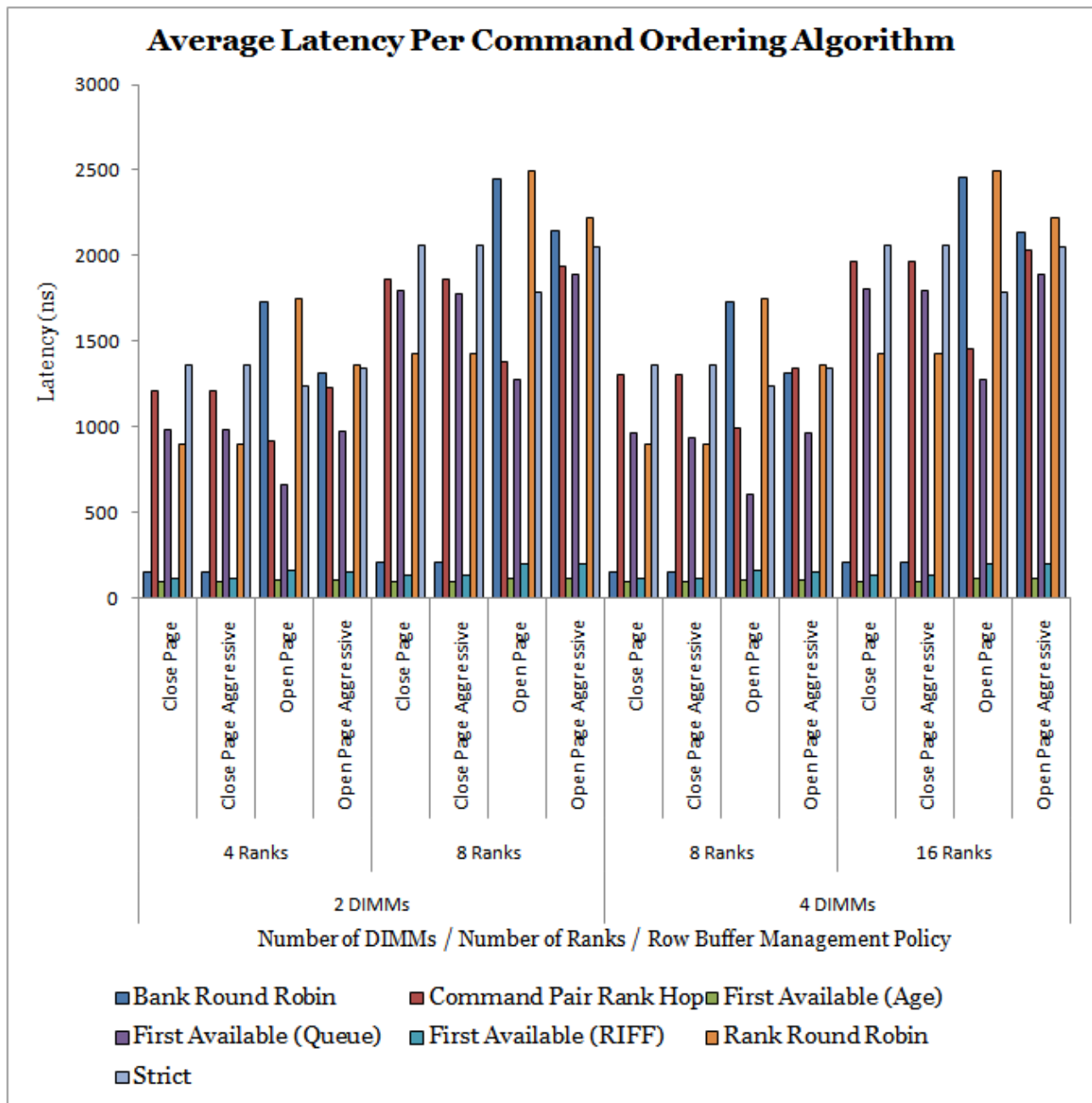


Figure 7.24: The average latency of the various command ordering algorithms as compared against row buffer management policies and different numbers of ranks and DIMMs.

The latency of the different configurations is inversely related to the bandwidth; setups that had higher bandwidth had lower latency. This follows because a system that can return requests sooner has more time to return more requests and thus has higher bandwidth. Again, first available (RIFF) and first available (age) lead the way with the lowest latencies, consistently. First available (age) is always a few cycles faster

than first available (RIFF). For some configurations, bank round robin performs quite well. The best policies have latencies around 100ns and bank round robin sometimes has latencies of 150-200ns. However, in some cases this goes up to nearly 2500ns. Surprisingly, algorithms that were perhaps 20-30% worse in terms of bandwidth were often 10-fold worse in terms of average latency. So even though bandwidth may suffer only a small amount when the command ordering algorithm is changed, the latency may increase drastically. For example, for a 2 DIMM, 8 rank system using close page row buffer management policy, first available (age) uses 7.32GB/s while rank round robin uses 6.77GB/s. This is a difference of 8%. However, rank round robin has an average latency of 1429ns while first available (age) is only 100ns. This is an increase of 1329%. There are many similar cases, showing that this trend is quite common. So comparing only the average bandwidth does not give an accurate description; one must look at the latency to determine the performance of the command ordering policy.

Finally, we must consider that energy used when evaluating which command ordering algorithm to use. Figure 7.25 shows the effects of the row buffer management policies as well as the command ordering algorithms. Often, bank round robin and rank round robin go through all the banks before returning to close out the banks, so when coupled with open page row buffer management policies, they tend to use significantly more energy. First available (queue) sometimes uses more energy than the other first available algorithms. The remaining algorithms have similar performance, with only command pair rank hop lagging somewhat behind. This means that again first available (age) and first available (RIFF) are often the best, so the power required to track these statistics may be offset by the power they save and the performance gains they bring.

Close page and open page are closer in energy use when used in saturation mode. The open page policies use only about 40% more energy in saturation mode and they do not even benefit from row reuse (random access patterns mean row reuse is less than .1%).

When saturated with requests, a close page policy is very similar to an open page policy. Open page aggressive will even begin to use CAS+P commands, so both appear to be a CAS+Pre and ACT with

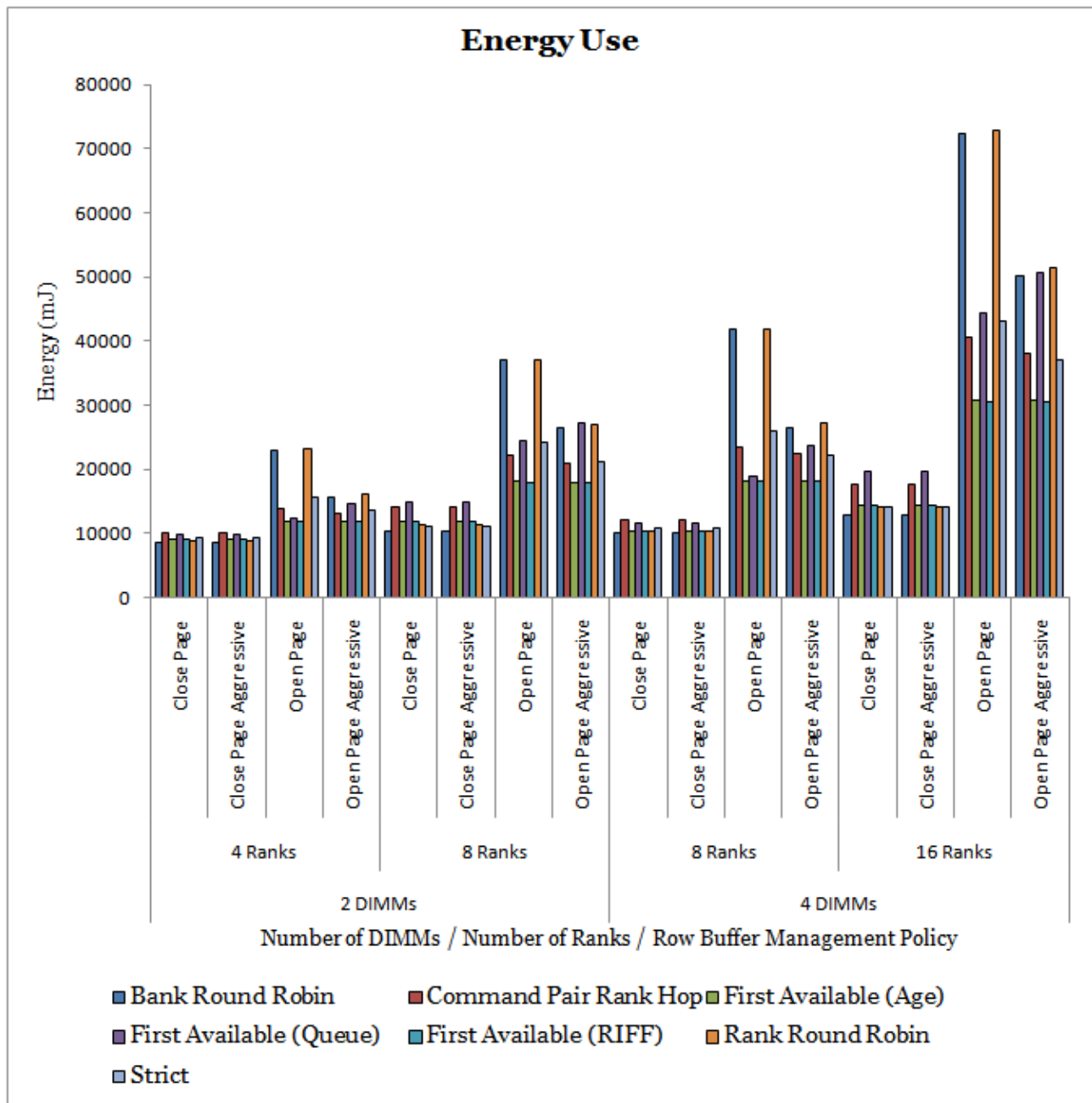


Figure 7.25: Total energy for different command ordering algorithms versus row buffer management policy, number of ranks and number of DIMMs.

minimal time sitting unused. If the server load is very heavy and open page is able to reuse many of the rows, it is possible that open page policies may have similar energy use to close page.

CHAPTER 8 CONTRIBUTIONS AND RELATED WORK

8.1 Summary and Contributions

The purpose of this thesis has been to attempt to understand what factors are important to understand while designing a high-performance memory system. Additionally, the same lessons can be applied for the purpose of designing a low-power, embedded memory system. Essentially, small changes in certain design parameters can cause large differences in application execution time and power usage, so system architects and memory controller designers should be aware of these factors.

DRAMsimII was designed in order to run these studies and provide a framework for evaluating proposed changes to the memory controller. DRAMsimII is a cycle-accurate, DDR/DDR2/DDR3/SDRAM simulator that provides statistics about its resource usage and also emulates the timings a real memory controller could have. It was designed to be flexible and easy to adapt. The various policies are well delineated, so adding a new policy was a fairly simple process. The designer needed only to write a new algorithm and modify the code a little. The DRAM protocol is abstracted from the algorithms to a degree, so it is less complicated to add a new policy. Additionally, all policies and algorithms are separate to the degree that they can be interchanged without affecting correct operation of the memory controller. This allowed the comparison of every combination of policies. These combinations showed that some address mapping policies worked very poorly with some application streams. They also showed that close page baseline and close page baseline opt generally worked better with close page policies and SDRAM baseline and SDRAM high performance worked better with open page row buffer management policies. They showed that in many cases, close page policies used less energy than open page policies. However, when the memory system was running at maximum capacity, the two policies had similar energy use.

Several of the policies are new, including low locality, high locality, and close page baseline opt. Low locality was meant to work with a close page row buffer policy and take advantage of entropy in the low order bits of the address by spreading the requests to the channel, rank, bank as the entropy moved to the higher order bits. High locality was exactly the opposite. It assumed that there would be entropy in the high order bits and attempt to spread the requests to ranks, banks, channels from high to low. This was intended to work well with applications that had very large address spaces and spread the requests to these requests seemingly randomly. Although none of the applications tested took advantage of this policy, it may still prove to be effective in larger applications. Finally, close page baseline optimized is a new policy intended to improve upon the close page baseline policy. By splitting the bits for the row into two parts and drawing half of them from the lower portion of the address, a more even distribution was achieved. Instead of distributing requests to separate ranks, they would be first given different rows and then different ranks, thus improving row reuse over close page baseline.

Several of the address mapping policies already existed and provided a sense for how to adjust and optimize novel policies to improve row reuse or channel/rank/bank distribution. Most were part of the original DRAMsim. Additionally, DRAMsimII includes the concept of a DIMM. Although this plays no part in addressing of individual DRAMs, this concept allows a user to specify the number of channels, DIMMs and ranks per DIMM for simplicity. This is a more realistic representation than simply having a channel with a certain number of ranks attached to it.

The close page and open page policies have been around for a long time and both are widely used. The close page aggressive and open page aggressive policies are based on the original policies, but have optimizations to attempt to exploit a few tendencies. Close page aggressive was created to allow a close page scheme but allow some row reuse when it was convenient. Open page aggressive was meant to attempt to make open page look more like close page when the chances for row reuse were more limited due to increasing congestion. Each of the “aggressive” policies have elements taken from the other and tend to

outperform the policies from which they are derived. In fact, close page aggressive is able to achieve 10% row reuse on some benchmarks.

The first available (age) policy existed previously and was complemented by adding first available (RIFF) and first available (queue). Because using age as a secondary criterion worked well at times, it seemed logical to think that other criteria would also work well, so choosing reads preferentially tends to return requests sooner. Likewise, choosing from the fullest queue tends to make more room in the queues and reduce queuing delays due to the per bank queues being full. Command pair rank hop was previously implemented statically as a 2 rank, 8 bank pattern, but was extended to be a generic algorithm for the purpose of these simulations. Now command pair rank hop will work on any system configuration. The simulations show that some command ordering algorithms work better than others when it comes to maximizing bandwidth. But for a system that does not need so much bandwidth or extremely low latency requests, it is better to use a simpler command ordering algorithm.

These studies also show that choosing the right address mapping policy and row buffer management policy can improve row reuse from 0% to over 80%, depending on the benchmark. They also show that there are limited returns on row reuse. Even though row reuse may be 75% or more, execution time is reduced by only a fraction. This shows that simply improving row reuse may not be the best goal to have.

DRAMsimII also provides a model to calculate availability for banks as command are executed. Previous versions looked at the history of past events to calculate when those resources would again be available. In contrast, DRAMsimII calculates the time when resources will be available when an event happens and then does a lookup to determine when an event can occur. This is useful because when the first available policies are being used, there are about 8000 times as many lookups as executed events. So it is much more efficient to simply do a lookup frequently than to calculate when resources are available.

Reordering of transactions at several levels has been described before[Rixner 00][Hur 04], but this work includes the first example of a decode window for transactions. This arose from the fact that certain transactions were being decoded to just a few banks, so those banks were becoming full very quickly.

Therefore, more transactions arrived but could not be enqueued in the transaction queue and the system appeared to be full even though most of the per bank queues were empty. In an effort to fix this problem, the decode window was devised to allow more transactions to be decoded at once. There is still a mandatory delay in the transaction queue before they can be decoded to simulate propagation delays that would exist in a real system.

Many previous studies use trace files to estimate what sort of speedup they would achieve when running in an actual system. However, this presents some timing inaccuracies. Requests may appear closer in time from a trace file than they would in a real system because they do not have to be returned to the requester first. So a read followed by a write would not appear together in a real system because the write would depend on the read being returned. This simulation uses M5 as a full system simulator and ensure that the simulations are as accurate as possible. Although speed is sacrificed by simulating the other components, the result is more accurate.

Finally, to track the effectiveness of an address mapping policy on a particular benchmark, the simulator tracks the distribution of latencies and requests on a per-bank basis. Then there is a post-processing script that will evaluate the logs and show a graph of what the distribution looked like. These graphs allow the user to immediately see if a given address mapping policy is using all available channels, ranks or banks for a given benchmark or whether the distribution is uneven. Being able to quickly, visually determine the effectiveness of a policy is important when designing a system. There are several other scripts available that generate graphs and many of the graphs in this paper were generated by those scripts and the statistics tracking package in DRAMsimII.

8.2 Related Works

Other groups have looked at memory controllers and the effects of realistic DRAM simulation before, but often with less detail or not in the context of a modern processor. Some looked at older variants of DRAM, often older than SDRAM, which would produce significantly different results.

Zhang, Fang et al. studied the Impulse Memory Controller [Zhang 01], which focuses on how to improve locality by remapping the memory requests as they arrive. So if a program tends to have an erratic access pattern that would normally span many rows, the OS can define a simple algorithm to allow the memory controller to map the requests in such a way as to achieve better locality. They used a significantly older simulator, RSIM [Pai 97], modeling a MIPS processor and simulated various image processing applications. They do not, however, use modern DRAMs or processors and they also do not adjust the row buffer policies or command ordering algorithm. This allows a secondary address mapping policy beyond the initial one that is build in to the memory controller.

Fan, Ellis et al. look at adjusting memory controller policies to improve power management [Fan 01]. They estimate DRAM idle time by using trace-driven simulations of some common workstation applications. This study only estimates idle time by using trace-driven simulation, while this thesis measures idle time using an entire simulated system to ensure much greater accuracy. Also, the paper uses RDRAM and measures only background power as determined by the state is in. This thesis uses more modern DDR3 DRAM and measures not only background power but power from various commands as well.

Hur and Lin propose a memory controller that schedules requests based on its recent read/write history[Hur 04]. They simulate this controller in the context of an IBM Power5 processor. They also simulate actual DRAM commands and look at the length of the history as a variable in choosing whether to choose a read or write next. It is not known what address mapping policy or row buffer management policy was used, but their command ordering algorithm estimated the latency of the available commands to decide which commands were likely to execute soonest. From these, they used the recent read/write ratio to choose between available commands. The delay is estimated by keeping track of the read-to-write and write-to-read turnaround latencies.

Zhang et al. use a permutation-based interleaving scheme to attempt to increase row reuse[Zhang 00]. They model an address mapping scheme for a system with one channel and one rank. Although they recognize that there are close page and open page row buffer management schemes, they use the open page

scheme for all of their tests. They propose to use a portion of the physical address and XOR it with another portion to generate a bank index that will hopefully yield greater row reuse. Several key memory timing parameters are modeled and a variety of SPEC95 benchmarks are simulated to look at the row reuse rates.

Aggarwal et al. create a DRAM controller that works with the processors' cache coherency policy to decide whether or not a request is likely to come from another CPU's cache[Aggarwal 08]. If it is likely, then a speculative read is not performed. It then waits to see if, in fact, the other processor was able to supply the data requested by the snoop. If it was not, then the memory controller performs the operation and returns the data after a longer latency. If it detects a snoop that is not likely to be returned by another CPU, the request is speculatively performed and the result conditionally used. DRAM latency is modeled as a constant value with basic delays due to contention. They report up to 21% reduction in DRAM energy usage by reducing accesses.

Rixner et al. have proposed and done studies on memory access scheduling, looking to reorder memory accesses in order to efficiently schedule memory accesses and improve bandwidth[Rixner 00]. They look at FIFO and prioritized policies to choose the ordering of memory accesses. They also introduce the idea of per-bank queues to accommodate commands for the various banks. Their policies and arbiters reschedule DRAM commands to improve performance of several benchmarks. By using first-ready scheduling, an average bandwidth improvement of about 25% is observed. Application bandwidth is improved by 85-93% over in-order scheduling.

Zhang and McKee looked at using a DRAM prefetcher with DRDRAM to reorder requests, improve row reuse and increase performance[McKee 00]. Many relevant DRAM timing parameters are modeled in a memory controller attached to a SimpleScalar CPU. They model one channel with eight individual devices. Various benchmarks are analyzed and their access patterns described in order to determine what sort of prefetcher would be most effective. They also show that reordering of memory accesses provides a performance benefit in all cases.

CHAPTER 9 CONCLUSIONS AND FUTURE WORK

Many simulation models over the years have simplified the memory subsystem to the point of losing accuracy. They assume a constant latency or a random latency in hopes of having a reasonably accurate result. However, for large applications that put lots of pressure on the memory system, these inaccuracies are multiplied many times to the point where the entire simulation, as shown by these simulations. In fact, simply changing internal policies can affect runtime by up to 66% for good policies and by over 12000% for poor policies.

We have shown that not only is it important to choose good policies for the memory controller, but also to make sure that these policies work well together and with the applications that will be run on this system. Choosing an open page row buffer management policy for a workload with little locality will have worse performance and higher power usage than a close page policy. If the wrong address mapping policy is chosen, then the rows may not get much reuse and any advantage of an open page policy is nullified.

The close page policies were often comparable to the open page policies in terms of execution times, but often yielded better power numbers. Open page policies, when coupled with the right address mapping policies, often had the best execution times and latencies.

Command ordering algorithms determined what sort of sustained bandwidth and latencies were possible. Choosing a more sophisticated first available (age or RIFF) often gave the best bandwidth and latency numbers for any configuration and workload. However, they are far more complex than simpler policies like bank round robin, which require less-complex implementations. The row buffer management policy also determined whether a command ordering algorithm could perform well. Although bank round robin did well when combined with close page policies, it was significantly worse with open page policies.

Not only did the memory system affect overall system performance, but overall power usage as well. Accurately simulating a system's memory system will give more accurate results and choosing appropriate memory controller policies and algorithms can help to further improve performance. Understanding the interactions of the application, CPU and memory subsystem will help to make better choices when choosing or designing a system.

9.1 A Word on Multithreading

All of the simulations were run using only a single threaded benchmark application. Although each of these was run in a Linux operating system with many threads, there was predominantly one thread running at a time. This gave that single thread most of the time on the CPU and most of the access to the memory system. Because it was the only thread accessing the memory system for most of the time, most of the requests had very fast access to the DRAMs, the queues were shorter and requests were generally returned sooner than they would otherwise be. In a multithreaded benchmark, it is likely that the requests will be returned after longer delays. The requests will compete for the same resources and keep the memory system busier, thus having to wait longer than before. However, it is possible that there will be something of a synergistic interaction between the threads. One thread may open a row before another thread asks for it and the second thread will benefit by not having to wait to open the row. The caches in other CPUs may also contain data needed by a thread, so the CPUs can source the data from their caches rather than going to memory. In fact, some threads may perform an ad hoc prefetch of data for other threads and then supply it through snoop requests, improving performance. This will alleviate some of the burden on the memory system, if the caches in the other CPUs running the other threads can supply the data often.

When this synergistic relationship fails to work well, there will be behavior akin to cache thrashing. When a single thread is running, a row may stay open for quite a while to accommodate all the requests, but with multiple threads, subsequent requests may close the rows and destroy the locality of the single thread.

The more cores are running threads, the greater the risk that the threads will interrupt the row locality of the other threads, but of course it depends on the application behavior and the address mapping policy.

The odds of row reuse go down if there are multiple simultaneous copies of a program running as separate threads. Because they have completely separate memory spaces, the chance of finding locality between different processes relies on supplying fetches to common libraries or memory addresses that happen to map to adjacent columns despite being from different processes.

Finally, if the single threaded version of a benchmark is mostly CPU-bound, then there may be little change in performance when switching to a multithreaded or multiprocess version as the memory system is not saturated. The additional load of those threads may not be sufficient to hurt the performance of the other threads.

One way to deal with the burden of additional threads would be through a quality-of-service monitoring/scheduling policy. It could ensure that some threads could not saturate the memory system while leaving other threads with a long wait. Threads waiting for many requests would have the lowest priority when waiting for access to the DRAMs, while threads not using as much bandwidth would jump to the front of the queues. This assumes that threads with few requests place higher priority on these requests because they are often part of the critical path of its execution. Likely these are for instruction misses or for loop variable indices. So giving them higher priority will allow them to run unimpeded while the heavy requesters will continue to wait, but will not really notice the impact of the other threads in the system. By prioritizing in this way, each thread should have a fair chance to get to the memory system in a reasonable time.

9.2 Future Work

In the future, it would be good to attempt to use an adaptive algorithm to look at which address mapping policies are optimal. Being able to look at the execution time of an application versus several different mapping policies and then choose and adapt the best would yield better policies.

It would also be interesting to explore a system that could dynamically switch between row buffer management policies based on load. When the system is lightly loaded, a close page policy would be best for its relatively better power usage. When the system detects that the request rate reaches a certain threshold, it could switch to an open page policy to begin to exploit row reuse better and have power usage roughly equivalent to a close page system, especially when heavily-loaded such as when connected to many CPUs.

When a row has been open for a while but has not been , it would be good to implement a policy that would automatically close the row to save power. This might help to bridge the gap between close page's power and open page's performance and row reuse. The ability to tune the time that it takes for the system to decide that a row is unused and close it would allow the system to improve row reuse rates for rows that are often hit just after they are closed or close rows quickly when they are rarely reused.

Lastly, it would be good to have a hierarchical memory system that could send requests to sub-memory controllers and retrieve the results when they are done to improve concurrency. This would reduce the data bus as a bottleneck and help to improve the capacity of a system. If there are 2 channels per controller and 4 ranks per channel, then each additional sub-controller could handle this and increase the capacity of the system by that amount.

CHAPTER 10 BIBLIOGRAPHY

- [Jacob 07] Memory Systems: Cache, DRAM, Disk. Bruce Jacob, Spencer W. Ng, and David T. Wang ISBN 978-0123797513. Morgan Kaufmann, September 2007.
- [Jacob 03] Bruce Jacob. "A Case for Studying DRAM Issues at The System Level." *IEEE Micro*, vol. 23, no. 4, July/August 2003.
- [Cuppu 2001] Vinodh Cuppu and Bruce Jacob, "Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance?." In *Proc. 28th International Symposium on Computer Architecture (ISCA 2001)*, June 2001, pp. 62-71.
- [Prince 2000] High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function. B. Prince ISBN 978-0471986102. John Wiley & Sons, August 1999.
- [Ganesh 07] Brinda Ganesh, Aamer Jaleel, David Wang and Bruce Jacob, "Fully-Buffered DIMM Memory Architectures: Understanding Mechanisms, Overheads and Scaling." In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, February 2007, pp. 109-120.
- [Ganesh 07-2] Brinda Ganesh. (2007). *Understanding and Optimizing High-Speed Serial Memory-System Protocols* (Doctoral dissertation). University of Maryland, College Park, MD.
- [Cuppu 99] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge, "A Performance Comparison of Contemporary DRAM Architectures." In *Proc. 26th International Symposium on Computer Architecture (ISCA 1999)*, May 1999, pp. 222-233.
- [Micron 07] Micron Technology, Inc..(2007, August). *Calculating Memory System Power for DDR3* (Publication No. 09005aef829559ff). Retrieved from Micron Technology, Inc. Online:
http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3%20Power.pdf

- [Micron 10] Micron Technology, Inc..(2010, April). *DDR3 SDRAM* (Publication No. 09005aef826aaadc). Retrieved from Micron Technology, Inc. Online: http://micron.com/document_download/?documentId=424
- [Wang 05-2] David Wang. (2005). *Modern DRAM Memory Systems: Performance Analysis and a High Performance, Power-Constrained DRAM-Scheduling Algorithm* (Doctoral dissertation). University of Maryland, College Park, MD.
- [Fan 07] Xiaobo Fan, Wolf-Dietrich Weber, Luiz Andre Barroso, "Power Provisioning for a Warehouse-sized Computer." In *Proceedings of the ACM International Symposium on Computer Architecture*, June 2007, pp. 13-23.
- [Wang 05] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. "DRAMsim: A Memory-System Simulator." *SIGARCH Computer Architecture News*, vol. 33, no. 4, September 2005.
- [Alakarhu 02] Juha Alakarhu, "A Comparison of Precharge Policies with Modern DRAM Architectures." In *Proceedings of the 9th International Conference on Electronics, Circuits and Systems*, September 2002, pp. 823-826.
- [Rixner 04] Scott Rixner, "Memory Controller Optimizations for Web Servers." In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2004, pp. .
- [Jacob 09] Bruce Jacob and David Wang. "System and Method for Performing Multi-Rank Command Scheduling in DDR SDRAM Memory Systems." U.S. Patent 7,543,102, filed April 17, 2006 and issued June 2, 2009
- [Binkert 06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi and Steven K. Reinhardt. "The M5 Simulator: Modeling Networked Systems." *IEEE Micro*, vol. vol. 26, no. 4, July/August 2006.
- [McCalpin 95] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers." *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, vol. 1, no. 1, December 1995.
- [Zhang 01] Lixin Zhang, Zhen Fang, Mike Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh and Sally A. McKee. "The Impulse Memory Controller." , vol. 50, no. 11, November 2001.

- [Pai 97] Vijay S. Pai, Parthasarathy Ranganathan and Sarita V. Adve, "RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors." In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997, pp. 1-5.
- [Fan 01] Xiaobo Fan, Carla S. Ellis and Alvin R. Lebeck, "Memory Controller Policies for DRAM Power Management." In *ISPLED '01*, August 2001, pp. 129-134.
- [Hur 04] Ibrahim Hur and Calvin Lin, "Adaptive History-Based Memory Schedulers." In 37th International Symposium on Microarchitecture. December 2004.
- [Zhang 00] Zhao Zhang, "Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality." In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 32-41.
- [Aggarwal 08] Nidhi Aggarwal, Jason F. Cantin, Mikko H. Lipasti, James E. Smith, "Power Efficient DRAM Speculation." In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA-08)*, February 2008, pp. 317-328.
- [Rixner 00] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, J.D. Owens, "Memory Access Scheduling." In International Symposium on Computer Architecture (ISCA). June 2000.
- [McKee 00] C. Zhang, S.A. McKee, "Hardware-Only Stream Prefetching and Dynamic Access Ordering." In *Proc. 14th International Conference on Supercomputing (ICS'00)*, May 2000, pp. 167-175.