Embedded Systems
Den Dolech 2,
5600 MB Eindhoven
The Netherlands
http://w3.win.tue.nl/en/

2009

MSc THESIS

Memory Pattern Generation based on Specification and Environment

Williston Sterchi Hayes Jr.

ES-MS-2009-0614000

Abstract

Guaranteeing hard real-time requirements in systems with multiple requestors accessing a single memory is a difficult task due to variable access times of SDRAMs. This problem has been solved with the use of the Predator memory controller, which is able to put a bound on worst-case latency and worst-case bandwidth. This controller uses precomputed sequences of SDRAM commands, called memory access patterns, in order to interact with SDRAMs. However, these patterns are difficult to construct due the complexity of SDRAM timing parameters and constraints. Thus, three heuristic-based pattern generation algorithms are developed that explore the trade-offs between run time and bandwidth offered. In the end, the approach selected provides adequate bandwidth while still offering a low run time. This pattern generator has been integrated into the Predator memory controller design flow.

TU/e Technische Universiteit Eindhoven University of Technology

Faculty of Mathematics and Computer Science

Memory Pattern Generation based on Specification and Environment

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Williston Sterchi Hayes Jr.
born in San Francisco, The United States of America

Embedded Systems
Department of Computer Science and Engineering
Faculty of Mathematics and Computer Science
Eindhoven University of Technology

Memory Pattern Generation based on Specification and Environment

by Williston Sterchi Hayes Jr.

Abstract

Guaranteeing hard real-time requirements in systems with multiple requestors accessing a single memory is a difficult task due to variable access times of SDRAMs. This problem has been solved with the use of the Predator memory controller, which is able to put a bound on worst-case latency and worst-case bandwidth. This controller uses precomputed sequences of SDRAM commands, called memory access patterns, in order to interact with SDRAMs. However, these patterns are difficult to construct due the complexity of SDRAM timing parameters and constraints. Thus, three heuristic-based pattern generation algorithms are developed that explore the trade-offs between run time and bandwidth offered. In the end, the approach selected provides adequate bandwidth while still offering a low run time. This pattern generator has been integrated into the Predator memory controller design flow.

Laboratory : Embedded Systems

Committee Members :

Advisor: Henk Corporaal, Electronic Systems, TU\e

Member: Benny Åkesson, Electronic Systems, TU\e

Member: Kees Goossens, SAI, NXP

Contents

List of Figures

List of Tables

Acknowledgements

This project has been offered through NXP Semiconductors with cooperation from the Eindhoven University of Technology. I would like to thank Kees Goossens and Benny Åkesson for allowing me to join their team in the System-On-Chip Architecture & Infrastructure group (SAI). I am very grateful to Benny Åkesson for his excellent guidance and help during the course of this project. Without him this project would not have been possible. I am also grateful to Kees Goossens as he provided a lot of insight into the project that allowed me to continually gain a new outlook on the thesis.

I would also like to thank Henk Corporaal for his valuable remarks and considerations made on the project. He always provided an interesting perspective, and encouraged the project's movement.

Additionally, Ad Siereveld and Roelof Salters at NXP are held in gratitude for sharing their deep knowledge of SDRAMs and SDRAM controllers with me. The information received from them helped clarify many problems I encountered in this project.

I would like to thank all of my colleagues at NXP for the various discussions we held and coffees we drank. Matteo Scordino, Ulf Winberg, Frank Ophelders, Dongrui She, Getachew Teshome, Pim Ritzen, Anna Kosek, Tion Kusumo, Jing Jing, Fabio Pania, and Adriano Sanches, you made this a great experience for me.

Williston Sterchi Hayes Jr.
Eindhoven, The Netherlands
August 31, 2009

1 Introduction

In this thesis we present a problem in the domain of real-time embedded systems that utilize SDRAM devices. In Section 1.1, we introduce requirements that applications with real-time requirements running on System-on-Chips (SoCs) have. This will be followed by Section 1.2, where we discuss the platform considered for this thesis. Section 1.3 details the problem of using SDRAM devices with the requirements specified in Section 1.1. Section 1.4 presents our solution to the problem. This is followed by sections detailing our contributions as well as the outline of this thesis.

1.1 Application Requirements

As transistors have gotten smaller due to advances in technology, now entire systems can be implemented on a single chip [13]. These SoCs typically have multiple IP blocks, and the gain offered by using SoCs is that the interconnect distance between IP blocks is typically much smaller than that of a traditional system. These shorter interconnects reduce time needed to transfer information from one IP block to another. Additionally, these systems use less power, which is also of great importance to embedded systems as power is a limited resource. We will now discuss the requirements of applications running on SoCs.

Applications running on real-time systems are used in a variety of situations, with each situation requiring different behavior. The first type of application requirements is soft requirements, meaning that if a deadline is not met, there still might be some useful information to be gained [5]. An example of an application with soft real-time requirements is an MPEG-2 decoder. This type of application would require good average-case bandwidth. If frames are not fully processed in time then the picture may appear pixilated or blocky. Yet, one can imagine that a viewer would still prefer to see a blurry picture over no picture at all.

The second type of requirements is hard-real time requirements. Applications which have hard real-time requirements cannot miss their deadlines under any circumstances [10,16]. An example of a system with hard real-time requirements is an anti-lock braking system found in automobiles. These systems require a low worst-case latency. If this latency requirement is not met, then the automobile's wheels may lock up resulting in skidding, and loss of control of the vehicle.

Another aspect of applications running on SoCs is their latency requirements. Some applications have very tight latency deadlines that must be respected, while other types of applications have more lenient deadlines. Both must be accounted for.

In this thesis we focus on applications which have hard real-time requirements. Additionally, we assume that we know nothing about the traffic that is generated, except for bandwidth and latency requirements. Finally, we state that we want to give guarantees

on worst-case latency and worst-case bandwidth that are analytically provable.

1.2 Considered Systems - Platform

This section details the system that is modeled. We have chosen to represent our system as multiple processing elements utilizing a single memory controller. We define the requesting service from these processing elements to the memory controller as requestors. We consider the case where the memory controller is controlling a DDR-SDRAM [2]. The entire overview can be seen in Figure 1.1.

The reason we represent our system as multiple processing elements using a single memory controller is because this is often required in practical cases. Having multiple memories is not a cost effective way to store data, because it increases power consumption and may require more expensive packaging. Thus, sharing memory decreases chip real estate and power usage, and results in a low cost-per-bit.

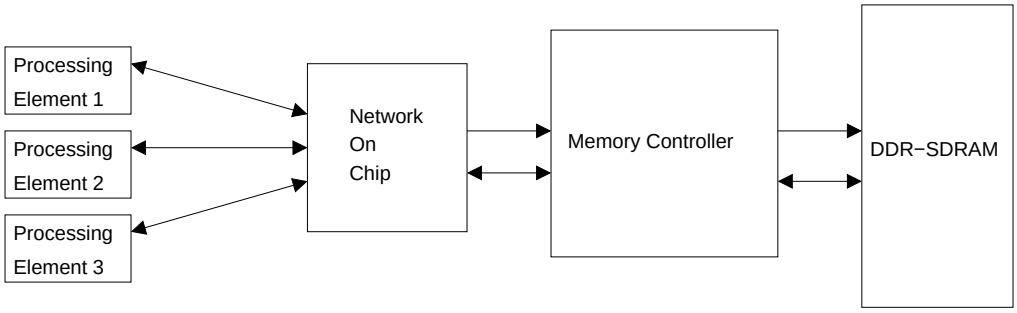

Figure 1.1: General overview of the considered system.

1.2.1 Predictability

All SDRAMs are predictable in the sense that they have ranges of times in which data is accessed, however, these ranges are not indicative of normal performance. We hope to place bounds on bandwidth and latency such that we can prove the exact value and make the value useful, i.e. the bounds we place are improved over the absolute worst-case values listed in the specification. The reason why we are concerned about improving worst-case values is that bandwidth is a shared, scarce resource, and has been proven to be a main bottleneck in SoCs [6, 12]. Thus, any improvement that can be determined in the worst-case is very useful.

Definition 1.1 (Predictability) *A predictable resource is one that has a known, useful, worst-case bound.*

The way a controller interacts with an SDRAM is by sending commands. These commands take the form of a read command, a write command, and a few other auxiliary commands. These read and write commands return a few words of data, the amount of which depends on a memory parameter.

The reason that placing bounds on latency and bandwidth is difficult is due to the fact that SDRAMs have varying access times depending on the sequence of commands. The amount of time it takes to access a word of data from an SDRAM depends on the current state of the memory. This results in variable bandwidth and latency.

Most memory controllers handle the problem of variable bandwidth and latency in two ways. *Statically scheduled controllers* work by using a fixed, precomputed schedule, in which the bandwidth and latency can be computed at design time, as long as the system is rigid. *Dynamically scheduled controllers* attempt to improve the average-case performance as much as possible, and they do this by creating their schedules at run time, and thus allow for a more flexible environment, at the cost of not being able to place analytic bounds on worst-case latency and bandwidth. Therefore in order to guarantee bounds that are known and useful in the case when we only know the bandwidth and latency requirements of applications, we need a new type of memory controller.

The memory controller we use is called Predator. It is a hybrid memory controller and as such it shares qualities of both statically scheduled and dynamically scheduled controllers, and is being developed at NXP [2]. The benefits of this controller are that it offers a predictable arbitration scheme which allows the bounding of latency. Additionally, it uses memory patterns that allow for the net bandwidth to be bounded. The details of this controller are discussed in Section 4.4.3. An important aspect of this memory controller is the way it interacts with the memory. It does not send individual commands to the memory, but rather it sends fixed-length sequences of commands, called *memory patterns*, to the memory. There exists one type of pattern for reading, one for writing, and 3 others that are introduced in Section 5. The read pattern is used whenever a requestor wishes to retrieve data from the memory, and the write pattern is used when a requestor wishes to store data to the memory.

1.3 Problem Statement

The problem that arises from the use of the memory controller mentioned above is that the patterns it uses to interact with the memory are difficult to compute, for numerous reasons.

There are many timing constraints and parameters that must be observed when creating a pattern [8, 9]. Some of these constraints are independent of each other, and some are dependent. Therefore, due to the complexity of the constraints, creating these patterns by hand is error prone and time consuming.

A second problem that arises is that we would like to use the same controller on a variety of different SDRAM devices. Incidentally, the timing parameters and constraints listed above are different for every device type. Additionally, the timing parameters are based on the speed of clock being used to time the device. To further complicate the issue, every device has multiple configurations, which also influences the parameters. Therefore this problem further reinforces the idea that creating these patterns by hand is time consuming.

Another issue that occurs is that we must take into account requestor requirements. As we shall see in Section 3, the amount of data being requested by a requestor compared

to the amount of data returned by a memory pattern has a significant effect on the overall efficiency, and thus the bandwidth provided by the memory controller.

To address the problems detailed above, we present our goal for this thesis in the following section.

1.4 Goal

The goal of this thesis is to create an algorithm that runs at design time, which produces memory patterns that are utilized by Predator. This algorithm should allow for the generation of multiple patterns such that different combinations of patterns can be evaluated for efficiency. The algorithm generates patterns based on a memory specification and requestor bandwidth and latency requirements.

The requirements of the algorithm are that it should take as input a memory specification and requestor requirements. The algorithm should produce memory patterns that provide as much bandwidth as possible, and that the production time of these patterns should not exceed 48 hours.

The benefits of this approach are that it removes the time-consumption and mistake factors out of the pattern generation. Furthermore, having an algorithm instead of performing the computations by hand allows us to apply the generator to future iterations of SDRAM, such as DDR4 and beyond.

1.5 Contributions

The contributions offered are that of creating a memory pattern generator which provides high-bandwidth patterns for many different memory types. This generator is integrated into the Predator configuration flow and is used by the hardware implementation.

The work of Eelke Strooisma on the Predator architecture is extended to include the notion of pattern dominance [17].

Three heuristic based approaches were developed for the memory pattern generator and compared against eachother. These approaches explore trade-offs between run time and bandwidth provided.

This pattern generator was integrated into the existing design flow of Predator, and new algorithms were developed to allow the configuration tools to work with any pattern set for any memory.

1.6 Outline

This thesis begins by discussing the architecture and run time operation of SDRAM devices in Section 2. In Section 3, concepts for measuring the efficiency of memories are discussed. This is followed by a general discussion of memory controllers, which includes their architecture, operation, and the state of the art in Section 4. Section 5 provides an in-depth look at the definition of memory patterns, how they are used by Predator, and how they may be used to determine the bandwidth provided by the memory controller. Section 6 details the 3 heuristic-based approaches that were created. The Predator design

flow, and how the generator is integrated into that flow, is described in Section 7. In the remaining two sections, conclusions derived from the thesis are discussed, and possible future work is detailed.

2 SDRAM

Prior to SDRAMs, data was stored in magnetic rings, which stored bits as the polarity of the magnetic field. This technology was slow and expensive, and thus Robert Dennard began the search for a faster memory type at IBM in 1967 [1].

His idea for improving this older design was to develop a memory cell which was composed of a capacitor and transistor, where the value of a bit is stored as charge in the capacitor. Today, SDRAMs are used in a huge variety of systems such as personal computers, automobiles, and mobile phones. They are used in so many places due to their cost-effectiveness in storing volatile data, as well as their speed.

Over time more modifications were made to the SDRAM to improve its effectiveness. The most significant of these was the introduction of DDR SDRAM. DDR memories provide a word of information twice per clock cycle, thus doubling the maximum rate of data transfer.

2.1 Architecture

An SDRAM device is composed of multiple banks. Each bank contains an array of memory cells, along with a row buffer, as shown in Figure 2.1. When an incoming request is sent to the SDRAM, the address is decoded into the bank address, row address, and column address based on a memory map. If the request is a read, then the row of data cells specified by the address is loaded into the row buffer. Once it is there, the data can then be read by an external resource. Conversely, if the request is a write, then the address is decoded as before, but the data is sent into the row buffer. After the write has finished loading data into the buffer, it is committed into the actual memory cells.

Using a bank architecture allows for a high degree of parallelism when accessing the SDRAM. Each bank can be considered as a separate memory that happens to share output pins. Each bank can be prepared for reading and writing in a pipelined fashion, which increases bandwidth. Additionally, due to the sharing of pins, the overall output pin count is lower than that of an equivalent capacity memory that does not share pins. Also, as a result of sharing output pins, less power is consumed.

There are two buses in an SDRAM that allow for interaction with other components. The first is the command bus, and it is an input into the SDRAM that is used to specify which type of action the SDRAM should perform. The second bus is a shared, bidirectional data bus. Any data that is transferred to or from the SDRAM must go through this bus.

Typical numbers for current DDR SDRAM memories are shown in Table 2.1.

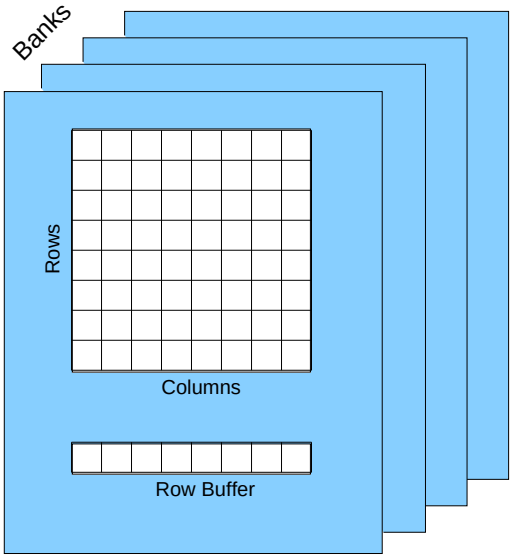

Figure 2.1: SDRAM Bank Architecture

Table 2.1: Typical characteristics of a DDR SDRAM device.

Number of banks	4 or 8
Capacity	256MB - 8GB
Word widths	4, 8, 16 bits
Column bits	∼10
Row bits	∼15

2.2 Commands

In order to interact with the SDRAM, some atomic commands are used. Most importantly, these commands allow for data to be read from the memory and written to the memory, as well some auxiliary commands that are required for proper operation.

The *activate* command takes a row and bank address as its argument, and then loads the specified row at the specified bank into the row buffer. Once data has been loaded into the row buffer it is allowed to be modified or read from.

The *read* command retrieves one burst of data from an activated row, and a *write* command sends one burst of data to the activated row. This use of these two commands is the only way data may be transferred to or from the memory.

The *precharge* command is the converse of the activate command. It accepts a row and bank address, and commits the row buffer in the specified bank into the specified row in the cell array. There is a special case concerning the read and write commands above, and that is they may be combined with an *auto-precharge*. The way auto-precharge works is that after a read or write command has been submitted to the memory, the bank will be precharged automatically at the earliest opportunity without any explicit command. An example of the activate-precharge loop is shown in Figure 2.2.

Due to the fact that bits in an SDRAM are represented by the charge in a capacitor,

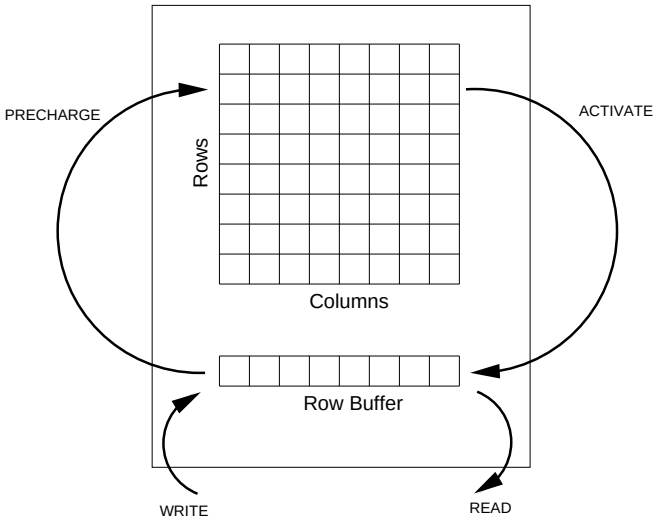

Figure 2.2: SDRAM Activate and Precharge Loop.

and that capacitors are not perfect and leak charge over time, a mechanism for keeping the data valid had to be invented. Hence, a *refresh* command was designed. This command 'refreshes' the data stored in the memory cell arrays of bank by recharging the capacitors that are storing data. This command may only be issued when all banks in the memory have had their row buffers precharged.

There are many timing constraints, as we shall see in the next section, that prevent two commands from being executed directly after each other. This space is then taken up by NOP commands, which stand for *no operation*. These commands essentially let the memory idle while waiting for another command to be scheduled. For clarification in figures, NOP commands are shown as blanks. The remaining commands are abbreviated as follows:

- Activate(Bank, Row) = ACT-Bank
- Read(Bank, Row, Column) = RD-Bank
- Write(Bank, Row, Column) = WR-Bank
- Refresh = REF
- Precharge(Bank, Row) = PRE-Bank

ACT 0			RD 0	ACT 1		PRE 0	RD 1		PRE 1

Figure 2.3: Example of a command sequence.

Also noteworthy is that row and column addresses are dropped from commands in their abbreviated form. This is due to the fact that the row and column addresses are

not significant with respect to timings and constraints. Only the bank has any effect in this regard.

2.3 Timing Constraints

There are many timing constraints and parameters that restrict the commands allowed to be scheduled at a given time. A *memory specification* lists all constraints for a given SDRAM device. The constraints specify the minimum amount of time that must pass between two commands. These constraints are listed in Appendix A. A small example is outlined below.

Four of the constraints and parameters are demonstrated in Figure 2.4. When we would like to send an activate command to two different banks, *tRRD* specifies the minimum amount of time that must pass after the first activate has been executed until the second one may be executed. *tRCD* specifies the minimum amount of time between an activate command and a read or write command. *BurstSize* refers to the number of words produced or consumed by the memory per read or write command, and thus is used to determine the granularity of a memory access. *DataRate* is the number of words on the data bus per clock cycle.

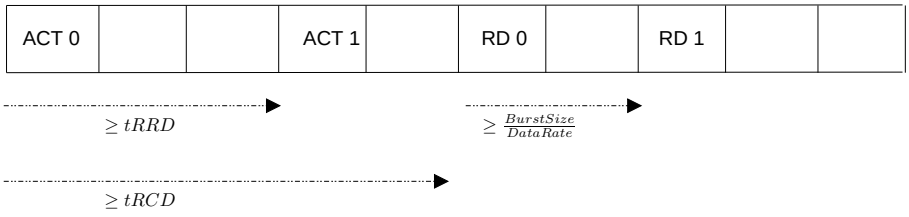

Figure 2.4: Example of *tRRD*, *tRCD*, *BurstSize*, and *DataRate* requirements and parameters.

As we see in Figure 2.4, the second activate may not be scheduled until *tRRD* cycles after the first activate, and the read command to bank 0 must be *tRCD* cycles after the activate to bank 0. The minimal delay between read commands is specified as such as we must make sure that a new read command does not interrupt data already being put on the data bus by a previous read command.

Table 2.2: Comparison of timing constraints in nanoseconds and clock cycles for two SDRAM devices.

	DDR2-400		DDR3-1600	
Constraint	ns	cc	ns	cc
tCK	5	1	1.25	1
tRC	55	11	45	36
tRCD	15	3	10	8
tRRD	7.5	2	6	5
tRP	15	3	10	8

An interesting point to take away from these parameters is that they are specified in nanoseconds, and that they do not change very much with successively newer SDRAM devices. This means that as the clock speed increases, the number of cycles needed to satisfy constraints also increases. This is demonstrated in Table 2.2 by listing a small sample of actual constraint values for different memory devices. The DDR2-400 device is assumed to be clocked at its maximum frequency of 200MHz, and the DDR3-1600 device is also assumed to be clocked at its maximum of frequency of 800MHz, which yields a clock cycle of 5ns and 1.25ns respectively.

3 Memory Efficiency

To make sensible calculations regarding latency and bandwidth, we need to establish a method for determining how efficiently a memory behaves. In this section the efficiency model of Woltjer [18] is presented. This model is used to differentiate types of efficiencies. Once these efficiencies are known, they may be used to calculate the amount of bandwidth provided by an SDRAM memory controller.

3.1 Peak Bandwidth

Definition 3.1 (Peak Bandwidth) *Peak bandwidth is the maximum number of bytes per second that can be transferred on the data bus.*

Computing the peak bandwidth is important as this number represents the maximum bandwidth offered by a memory. Inefficiencies introduced by physical characteristics of an SDRAM can significantly lower the actual amount of bandwidth provided to a resource.

If we let $DataBusWidth$ equal the width in bits of the data bus, and $ClockFrequency$ be the frequency of the clock, then we have

$$bandwidth_{peak} = \frac{ClockFrequency \text{ cycles}}{\text{second}} * \frac{2 \text{ bursts}}{\text{cycle}} * \frac{DataBusWidth \text{ bits}}{\text{burst}} * \frac{1 \text{ byte}}{8 \text{ bits}} \tag{3.1}$$

As an example, if we had a DDR2-400 SDRAM device running at 200MHz with a 16 bit data bus we have

$$bandwidth_{peak} = \frac{200M \text{ cycles}}{\text{second}} * \frac{2 \text{ bursts}}{\text{cycle}} * \frac{16 \text{ bits}}{\text{burst}} * \frac{1 \text{ byte}}{8 \text{ bits}} = 800 \text{ MB/s}$$

Most memory controllers are unable to provide this amount of bandwidth due to physical characteristics of an SDRAM, such as refresh. In order to properly calculate the worst-case bandwidth offered by a given memory, we must examine areas of SDRAM operation where efficiency is lost.

3.2 Data Efficiency

Definition 3.2 (Data Efficiency) *Data efficiency is the amount of data requested by a component compared to the actual amount of data returned by the memory controller.*

Data efficiency is traffic dependent, and therefore cannot be computed at design time unless we know some information about the requestors using the memory controller. Specifically, we need to know the granularity of the memory, as well as the sizes of requests of the resources.

Often, the amount of data requested does not exactly match the amount of data actually returned by a memory. This occurs because there can be many types of processing elements performing different actions, that share one memory. Because the requestors' application requirements may be very different, it is possible that the amount of data being requested may also be very different. Thus, frequently a requestor is given more data than it requires, and must throw away the excess. This data that is thrown away is representative of the data efficiency.

As seen in Equation (3.2), the severity of the loss in efficiency is dependent on the request size. Requestors with a small request size using a memory with a large granularity will have extremely poor data efficiency. As we see later in Section 3.6, this translates to poor actual bandwidth as well. Thus, data efficiency can change significantly depending on the resources using the memory controller. Experimentally, it has been observed that a data efficiency of 75% is expected for an MPEG-2 stream [18].

$$e_{data} = \frac{size_{request}}{granularity_{memory}} \tag{3.2}$$

As an example, we examine Figure 3.1 and assume a requestor wants to read 6 words. However, the granularity of this memory is such that it provides 4 words per read burst. Thus, in order to provide 6 words to the requestor, 2 read commands must be sent. Therefore e_{data} in this case is 6/8, or 75%.

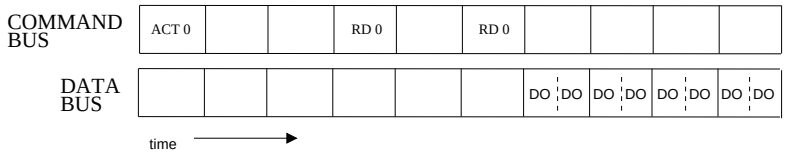

Figure 3.1: Visual representation of bank efficiency. Blank slots in the command bus represent NOP commands.

3.3 Bank Efficiency

The time taken to access a memory cell is highly variable for SDRAMs. If a read command is sent to a bank which currently has a different row already activated, the controller must first precharge the current row, activate the new row, and then send the read command. This penalty is captured in the bank efficiency. By examining the amount of cycles that data is on the bus, compared to the total number of cycles taken to get that data on the bus, we are able to compute this value. Bank efficiency is dependent on the memory map chosen, as we shall see in Section 4.2. As the bank efficiency is dependent on the addresses of requests, which are not known at design time, we cannot place a useful bound on bank efficiency.

If we examine Figure 3.1, we are also able to compute the bank efficiency of this pattern. In this particular period, we see 4 cycles where there is data on the bus, and that the entire length of this group of commands is 10 cycles long. Therefore the bank efficiency is 40%.

$$e_{bank} = \frac{DataCycles}{TotalCycles} \tag{3.3}$$

3.4 Switching Efficiency

SDRAMs require a time buffer between a read and write command, such that the direction of the data bus can be reversed. This often results in cycles where no data can be on the bus. Thus, we must take this drop in efficiency into account by examining the difference between the time taken for a read and write command, compared to the time taken for a read and write commands, plus their respective switching costs. According to Woltjer [18], traffic consisting of 70% reads and 30% writes yields a switching efficiency of 93.8%.

In general, the switching efficiency is not possible to compute at design time as the read to write switching frequency is not known at design time. However, in Section 5 we see how some design decisions allow us to compute this value.

3.5 Refresh Efficiency

Every SDRAM must be refreshed periodically in order to maintain the integrity of the stored data. However, in order for a refresh to occur, all activity must cease, and all banks must be precharged. Refresh efficiency is used to determine how much the effectiveness of the memory is changed due to the lost time introduced by the refresh command.

Therefore we compute $e_{refresh}$ as the amount of time in a period where the memory is not busy with refresh commands. t_{ref} refers to the amount of time taken for a refresh command to be executed, and t_{period} is the period in which refreshes are executed.

$$e_{refresh} = 1 - \frac{t_{ref}}{t_{period}} \tag{3.4}$$

Refresh efficiency is not traffic dependent and therefore can be calculated at design time. Due to the very long refresh period (7.8 μs at normal operating temperatures) compared to the time taken to execute a refresh command (on the order of 100 ns), the refresh efficiency is usually around 99%.

3.6 Net Bandwidth

Definition 3.3 (Net Bandwidth) *Net bandwidth is the actual amount of bandwidth provided by the memory to requestors.*

The concern of applications using this memory controller is directed at the net bandwidth, as this is the amount of bandwidth they are actually provided. It is calculated by simply finding the product of the above efficiencies and the peak bandwidth, with one exception. e_{data} is not taken into account due to the way bandwidth allocation and requestor requirements are handled by our memory controller. As we shall see in

Section 7.1.1, e_{data} is factored into the requestor's requirements. Thus, the equation for calculating $bandwidth_{net}$ becomes,

$$bandwidth_{net} = e_{bank} * e_{switch} * e_{refresh} * bandwidth_{peak} \tag{3.5}$$

4 Memory Controllers

A memory controller is the interface between an SDRAM and the system. By having it as a separate unit, the system utilizing the SDRAM does not need to worry about timing constraints and other details of a specific SDRAM, and can simply treat the device as a general memory.

Specifically, a memory controller is responsible for the following tasks:

- Memory Mapping
- Command Generation
- Command Scheduling

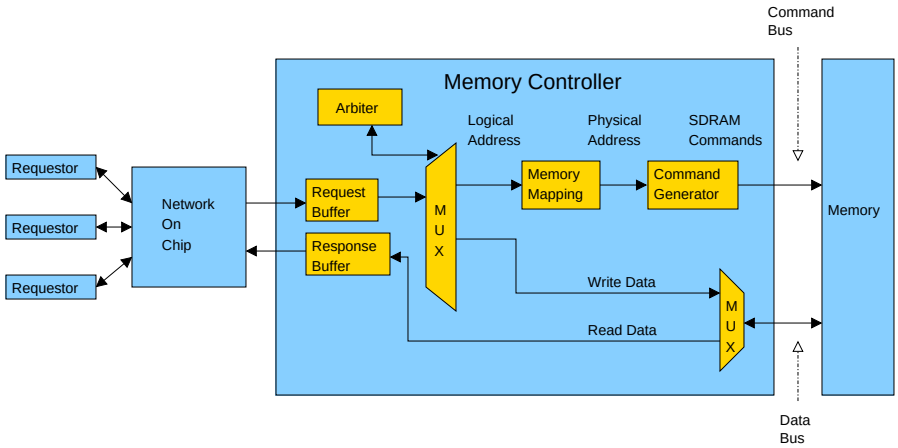

Figure 4.1: Architecture of a memory controller.

The controller works by first accepting some requests from external IP blocks, and then storing them into request buffers. The addresses are then translated into the physical address space by means of a *memory map*, after which they are scheduled by the *arbiter*, and converted to memory commands by the *command generator*. The response from the memory is then sent back to the response buffer, and from there it will be sent back to the requestor.

4.1 Arbiter

The arbiter is responsible for determining which outstanding request is allowed to be sent to the memory controller. The logic for choosing which request ought to be scheduled

depends on the type of system in which the controller is used. Some systems value low power use, some value high efficiency, and some value soft or hard real-time requirements. The characteristics of requests that the arbiter looks at are the requirements of the requestor, and optionally, the current state of the memory device.

4.2 Memory Mapping

A memory map provides a translation between logical memory and physical memory addresses. Thus, to a requestor, a memory looks like one large continuous array, while in reality the memory is organized into separate banks, rows, and columns. The choice of memory mapping is important as it has a large effect on latency and bandwidth, as seen in the following sections.

4.2.1 Continuous Memory Map

One type of translation used is called the continuous memory map. It is called such because successive addresses in the logical space are mapped to successive addresses in a single row in a single bank. Thus, the same row is accessed over and over again until the end of the row is met. At this point the mapping may switch to a new bank or to a new row in the same bank. As a result, row activation cost needs only to be incurred once in order to access a significant amount of successive data.

A continuously mapped address space can be seen in Figure 4.2. The memory presented is a highly simplified design for clarity. The memory has 4 banks, each with a 2 by 2 array of memory cells. Therefore 1 bit is needed to represent the column and row respectively, and 2 bits for the bank address. Thus, in total 4 bits are required for the logical address. As can be seen, if the least significant bit of the logical address is mapped to the column address, the next 2 significant bits are mapped to the bank address, and the final bit is mapped to the row address, we achieve a continuous memory map.

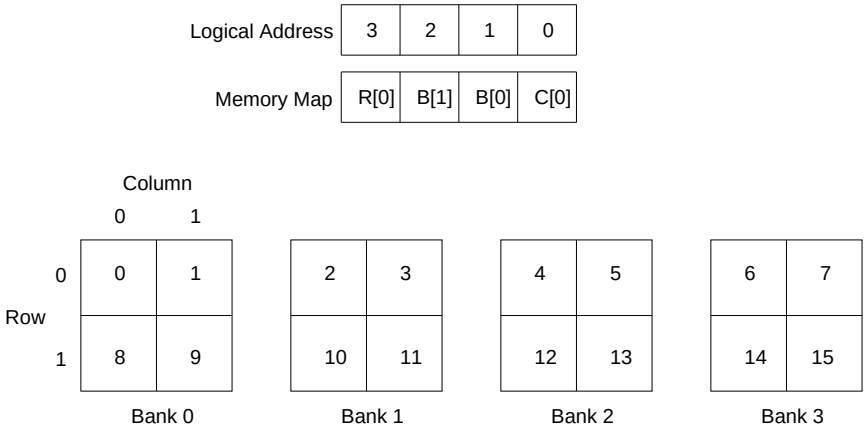

Figure 4.2: Illustration of a continuous memory map.

This type of mapping can be very beneficial to bandwidth and latency when the system has a low number of requestors, and the data being considered exhibits data

locality. Consider a system with a single requestor, who requests data from consecutive logical addresses, as seen in Figure 4.2. If this requestor wants to send 4 read commands starting at bank 0, row 0, column 0, then we see that the memory controller needs to send an activate command to bank 0, and an activate to bank 1, and 2 reads to each bank. The resulting group of commands is shown in the top graphic in Figure 4.3.

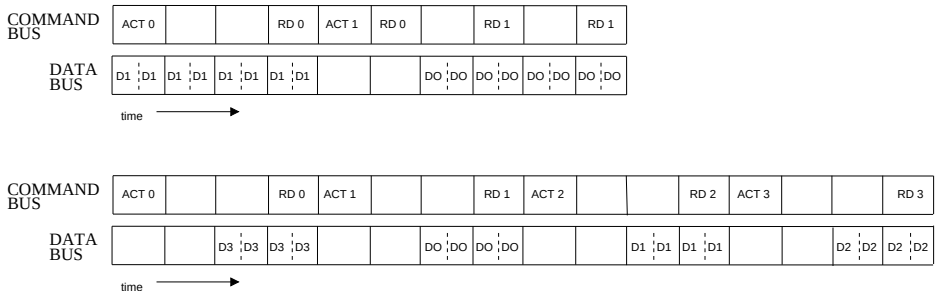

Figure 4.3: Example of commands generated for a continuous memory map (top) and an interleaving map (bottom). The data bus has been wrapped back onto itself.

A significant problem occurs when we examine the sequence of commands generated in the worst-case scenario. We see in the top graphic of Figure 4.6 that when a requestor is trying to access different rows in the same bank, the efficiency of the memory controller drops significantly. This occurs because the minimum time between sending an activate command to the same bank is quite large.

Another issue occurs when the number of requestors begins to grow. The effectiveness drops significantly because of different requestors all sharing the same row buffer in a bank. With high numbers of requestors, the frequency of overwriting a given bank's row buffer increases, and this type of mapping becomes less viable.

As mentioned above, the use of continuous memory maps are limited to use cases where there are a low number of requestors. However, there is a method that may be employed to alleviate this problem, called bank partitioning. Bank partitioning is achieved by funneling requests from different requestors statically into different banks. In this way, the row buffers are not scrambled by interfering requestors assuming there are fewer requestors than the number of banks. However, to do this we must make assumptions regarding the number of requestors. A design goal of this thesis is to not make any assumptions regarding the number of requestors in the system, and thus we have chosen to not employ bank partitioning.

4.2.2 Interleaving Memory Map

The alternative type of translation to a continuous memory map is called an interleaving memory map, and is shown in Figure 4.4. This style of translation is called interleaving because successive bursts of accesses in the logical address space are mapped to different banks in the physical address space. Initially, we might expect a performance reduction as every bank must now be activated before it is accessed. However, there is a memory requirement that prevents a read or write command from being issued to the memory immediately after an activate command. Thus, there are some wasted cycles. In an

interleaving memory map we can insert an activate command to a different bank while waiting to be allowed to schedule a read or write command. If the requested size of data is large enough, the cost of activating all banks can be ameliorated. Thus, the effectiveness of the interleaving memory map increases as the granularity of the accesses increases.

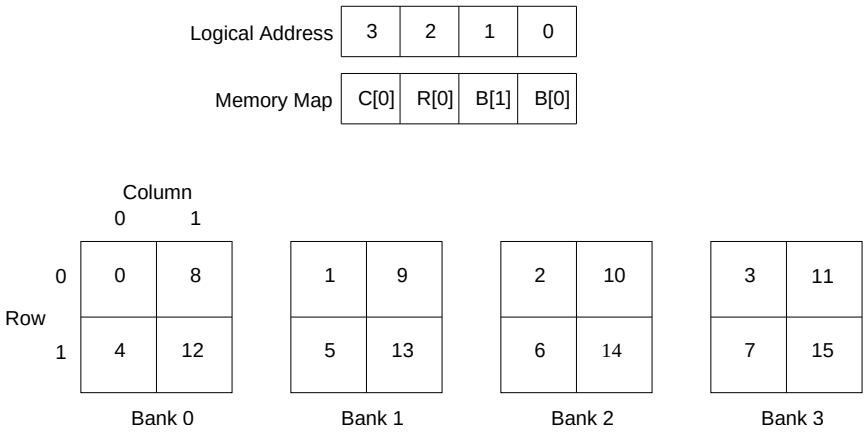

Figure 4.4: Illustration of an interleaving memory map

Another benefit of great importance to the goals of this thesis is that we want to place a useful bound on worst-case bandwidth. The problem that the continuous memory map suffered from in the worst-case scenario was that the activate to activate constraint on the same bank creates large gaps between the access commands. With the interleaving memory map, we are not concerned about activating the same bank repeatedly, but rather activate different banks. The constraints governing the minimum time between activates to different banks is much less strict than its same bank counterpart, therefore the number of cycles to produce the same amount of data in the worst-case is significantly less when using an interleaving pattern.

Therefore the benefits of this approach are that the latency to access a word is not dependent on the state of the SDRAM, and activate command costs can be reduced. Because this mapping makes no assumptions about the type of data being used, or about the number of requestors in the system, and for the benefits detailed above, the interleaving map has been chosen as the type of memory map to be used.

To demonstrate this benefit we examine two examples. If we compare the length of time for sending 4 read commands with a continuous memory map in the best-case scenario, compared to an interleaving memory map, we see that the continuous map produces the same amount of data with many fewer clock cycles. This is seen in Figure 4.5. This shorter pattern sequence translates to a higher bandwidth for the continuous memory map due to a higher bank efficiency, as shown in Equation (3.3).

However, in Figure 4.6 we compare the length of time for executing 4 read commands with a continuous memory map in the worst-case scenario, against an interleaving memory map, we see a much different result. The worst-case scenario occurs when requests coming into the memory controller wish to read from different rows in the same bank. As we stated in our goals for the thesis, we want to place useful bounds on worst-case

bandwidth, therefore we have chosen to use the interleaving memory map, as it provides better worst-case performance.

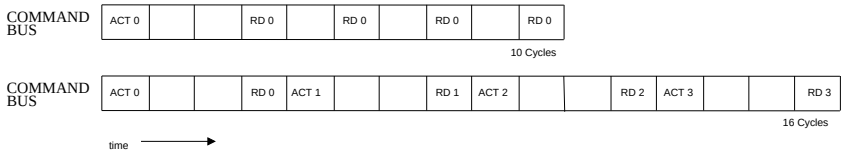

Figure 4.5: Illustration of best-case scenario when using a continuous memory map (top) compared to an interleaving memory map (bottom)

.

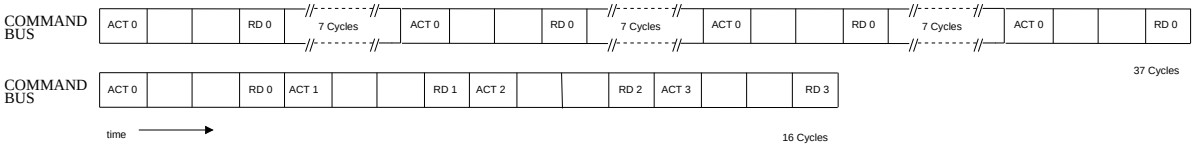

Figure 4.6: Illustration of worst-case scenario when using a continuous memory map (top) compared to an interleaving memory map (bottom)

.

4.3 Command Generator

The command generator is responsible for producing SDRAM memory commands based on requests in the request buffer, depending on the memory map chosen. The generator may need to be aware of the current state of the SDRAM device. If a requestor is trying to read a word of data from a bank, often the row in the bank is not already in the row buffer, therefore the controller must first activate the row in the bank, and then it may send the actual read command to retrieve the data. The generator must be configured to type of SDRAM device being used, as each type requires different amounts of time between commands.

4.4 Types of Memory Controllers

Based on the memory controller components defined previously, this section gives an overview of the various types of memory controllers used today.

4.4.1 Statically Scheduled Memory Controllers

Statically scheduled controllers work by having a predefined order of commands programmed at design time. These controllers are used in systems where there is no unpredictability in the traffic, and thus everything is static. If there is any variation in the read to write ratio of commands, or if the bandwidth requirements change, then the current schedule becomes invalid. In order to remedy this, one can consider creating a static

schedule for every use case that the system may encounter, however this number may become very large. To overcome these problems, a new type of controller was developed, and we examine it in the next section.

4.4.2 Dynamically Scheduled Memory Controllers

In the above section, we saw that statically scheduled controllers cannot perform well when the traffic in question varies over time. Thus, a new type of controller was invented; the dynamically scheduled memory controller. This controller works by generating commands at run time depending on requests. It then attempts to optimize the actual command schedules based on the current outstanding requests, and this can be done with command reordering [14], a self-optimizing controller [7], and other techniques [15].

Command reordering is mechanism by which a memory controller attempts to improve average-case efficiency. It can accomplish this with command grouping, where the controller groups similar commands together to reduce the number of data bus direction changes, and bank grouping, which means the controller separates requests by bank. Then, each bank determines when it should schedule an outstanding request depending on a selectable policy. Bank reordering has been shown to improve sustained bandwidth by up to 144% over a system without reordering when applied to realistic synthetic benchmarks [14]. An example of command grouping in shown in Figure 4.7. This example demonstrates how two commands can be reordered to improve efficiency. In the upper figure we see that if the controller were to execute these requests in order, that there would be 3 changes in data bus direction, whereas after commands are reordered, there is only one change in direction. This offers a significant increase in bandwidth, however, the amount of time until command 2 is executed has now been increased.

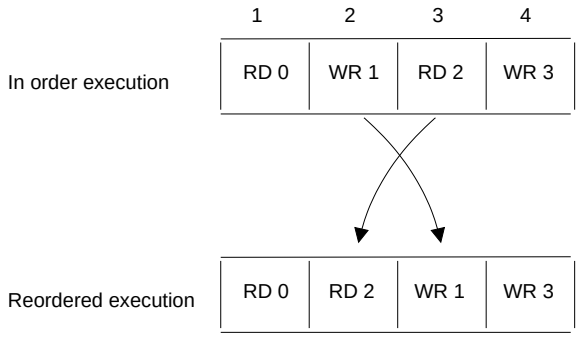

Figure 4.7: Illustration showing reordering of commands to reduce data bus direction changes. The top figure shows the requests in order, the bottom figure shows the requests reordered.

Another method of improvement mentioned above was a self-optimizing controller. During the operation of this controller, it is constantly examining the long term impacts its schedules have on memory efficiency. If it begins to see a shift in the incoming requests, it optimizes the schedules to produce the best possible patterns. It has been shown to increase the net bandwidth of a multiprocessor system by 30% over a system with in-order scheduling [7].

These optimizations can greatly increase the average-case latency and bandwidth at the cost of the worst-case latency and bandwidth. Because commands are possibly reordered during runtime, it becomes very difficult to analytically determine the absolute worst-case latency and net bandwidth. As such, simulations are the only way to try and verify if bounds are met. As a result, only ranges and averages are returned as verification for system traces that have been simulated. In a system with hard real-time deadlines this type of verification could be unacceptable, and therefore these dynamically scheduled memory controllers are not suitable for this thesis.

4.4.3 Hybrid Memory Controllers

To provide useful performance to a system with variable traffic without sacrificing hard real-time requirements, a new memory controller has been proposed [2].

This controller works by combining aspects of both the statically scheduled controller and dynamically scheduled controllers. A set of groups of commands are utilized by the controller to interact with the memory. Each one of these groups of commands, referred to as a pattern, is responsible for one type of interaction. There is a read pattern, a write pattern, a read to write switching pattern, a write to read switching pattern, and a refresh pattern. These patterns are generated at design time, and their construction is governed by the constraints in the memory specification and limited by requestor requirements.

The idea that separates this controller from a statically scheduled controller, is that this controller schedules these precomputed patterns dynamically, using a predictable arbiter.

Dynamic scheduling with a predictable arbiter allows us to bound the worst-case latency, while knowing the patterns in advance lets us bound the worst-case bandwidth. Another benefit of this approach is that by placing the commands into larger groups, we simplify the amount of constraints present. The patterns generated at design time are explored in Section 5. A demonstration of the differences among the memory controllers introduced above is shown in Table 4.1.

Table 4.1: Comparison of characteristics of different memory controllers.

Controller	Commands	Arbiter	Predictability	Cost	Complexity
Dynamic	Dynamic	Dynamic	No	Bad Worst-Case	High
Predator	Static	Dynamic	Yes	Dependent on e_{data}	Medium
Static	Static	Static	Yes	Rigid Design	Low

5 Memory Patterns

Memory patterns are sequences of SDRAM commands, and are used by the hybrid memory controller proposed in Section 4.4.3. In this section, we present an overview of the patterns, followed by descriptions of the individual patterns themselves. Later, we discuss how these patterns can be evaluated in terms of their efficiency, latency, and bandwidth.

5.1 Pattern Overview

Patterns provide the only mode of interaction between a memory controller and the memory itself. We have divided the types of patterns into two groups: access patterns and auxiliary patterns.

Read patterns and *write patterns* are grouped under access patterns as they are the only patterns that can actually access the contents of the memory. Once the access patterns have been determined, they are used to compute the lengths of the auxiliary patterns.

The remaining patterns are auxiliary patterns. These patterns consist of the *read to write switching pattern*, the *write to read switching pattern*, and the *refresh pattern*. These patterns perform more of a support role, and are responsible for giving the data bus time to switch direction, and keeping the data cells in the memory charged.

5.1.1 Scheduling Rules

The patterns are constructed in such a way that a read pattern may be immediately scheduled after itself. Similarly, write patterns are also constructed in such a manner. The reason this is done is to reduce the number of patterns required to interact with a memory device.

When a switch occurs from a write pattern to a read pattern, a write to read switching pattern must be executed. This provides the SDRAM with the required time to alter the direction of the data bus. Similarly, a read to write switching pattern must be executed when the memory controller wants to execute a write pattern when a read pattern has just finished. A refresh pattern may be executed after read and write patterns. Switching patterns are not required after a refresh. An example of a sequence of patterns can be observed in Figure 5.1.

Figure 5.1: Sequence of various patterns.

5.2 Types of Patterns

Detailed in this section are the types of patterns used by the Predator memory controller.

5.2.1 Access Patterns

A read pattern is used to retrieve data from the SDRAM. There is one activate command per bank. Each activate is followed by *BurstCount* read commands to each bank, in an interleaved fashion. *BurstCount* is defined in Definition 5.2. Wherever there are gaps remaining, they are filled with NOP commands. Similarly, a write pattern is constructed the same way with write commands instead of read commands.

In Figures 5.2 and 5.3, we see two examples of valid patterns for a DDR2-400 SDRAM device. The definition of a valid pattern is seen in Definition 5.1.

Definition 5.1 (Valid Pattern) *Valid patterns are defined as patterns that do not violate any of the constraint timings of a given SDRAM.*

Definition 5.2 (BurstCount) *BurstCount is the number of read or write commands per bank present in a read or write pattern.*

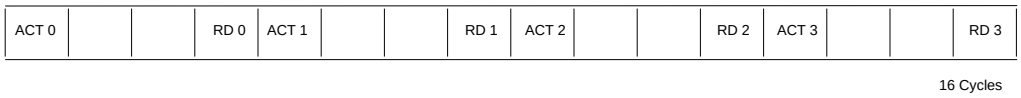

Figure 5.2: Read pattern for DDR2-400 SDRAM. Blank schedule slots indicate NOP commands.

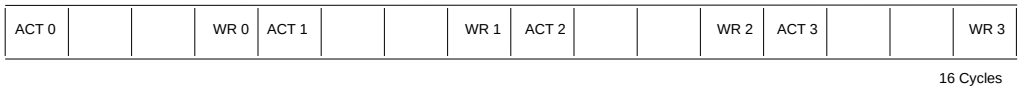

Figure 5.3: Write pattern for DDR2-400 SDRAM.

5.2.2 Auxiliary Patterns

The read to write switching pattern is used to provide time to the SDRAM so that it may change the direction of its data bus. This pattern only contains NOP commands, the number of which is dependent on the required distance between a read and write command as stated by the memory specification. The write to read switching pattern is constructed in the same way, except it is used in the transition between a write pattern and a read pattern.

As a side note, sometimes this switching time may be completely mitigated depending on the SDRAM specification and the pattern chosen.

The refresh pattern contains one refresh command, along with a number of NOP commands. The number of NOP commands, and the exact location of the REF command, are governed by the read and write pattern construction.

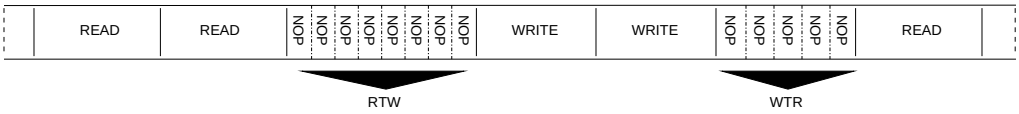

Figure 5.4: Switching patterns being used between read and write patterns.

Definition 5.3 (Pattern Set) *A pattern set is the collection of read, write, read to write switching, write to read switching, and refresh patterns that are generated for a particular memory specification and BurstCount.*

5.3 Pattern Dominance

When computing the various types of efficiencies and latencies for a given pattern set, it is important to determine which sequence of patterns will produce the worst-case latency and bandwidth. There are 4 categories into which a pattern set may fall. These are, read dominance, write dominance, mix read dominance, and mix write dominance. Below, we have outlined how to calculate the different dominance types. t_{read}, t_{write}, t_{rtw}, t_{wtr}, and t_{ref} refer to the amount of time taken to execute the respective pattern in a set.

5.3.1 Read Dominance

Read dominance is said to occur when the length of a read pattern is greater than the sum of the read to write switch, write, and write to read switch patterns. Thus, when attempting to compute the worst-case latency of a pattern, one should assume that only read patterns are being executed by the controller. An example of a read dominant pattern set is shown in Figure 5.5

$$t_{read} > t_{write} + t_{wtr} + t_{rtw} \tag{5.1}$$

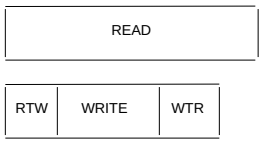

Figure 5.5: Illustration of read dominance.

5.3.2 Write Dominance

Write dominance is the converse of read dominance. When calculating worst-case latency or bandwidth, one should assume that only write patterns are being executed. An example of this can be seen in Figure 5.6.

$$t_{write} > t_{read} + t_{wtr} + t_{rtw} \tag{5.2}$$

This may be rewritten as

$$t_{read} < t_{write} - t_{wtr} - t_{rtw} \tag{5.3}$$

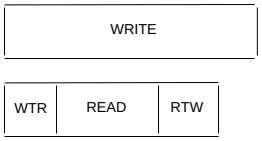

Figure 5.6: Illustration of write dominance.

5.3.3 Mix Dominance

When a set of patterns does not fall into the read or write dominance areas, it must fall into the mixed dominance area. This area is reached when alternating between read and write patterns provides the worst-case scenario.

$$\begin{aligned} &t_{read} + t_{rtw} + t_{write} + t_{wtr} \geq 2 * t_{read} \quad \wedge \\ &t_{read} + t_{rtw} + t_{write} + t_{wtr} \geq 2 * t_{write} \end{aligned}$$

This may be rewritten as

$$t_{write} - t_{wtr} - t_{rtw} \leq t_{read} \leq t_{write} + t_{wtr} + t_{rtw} \tag{5.4}$$

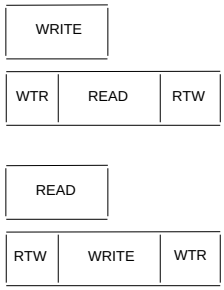

Figure 5.7: Illustration of mix dominance.

Mixed dominance is further subdivided into two more areas; mixed read and mixed write dominance, in order to further simplify the calculations needed to determine worst-case latency and bandwidth. Mix read dominance means that the sum of lengths of read and write to read switch patterns is larger than the sum of write and read to write switch patterns. The bandwidth offered by a pattern set of mix read or mix write dominance does not change, as the sum of the lengths of the patterns remains constant, however the latency calculations may produce different results depending on which dominance type the pattern exhibits.

$$t_{wtr} + t_{read} \geq t_{rtw} + t_{write}$$
$$t_{read} \geq t_{write} - t_{wtr} + t_{rtw} \tag{5.5}$$

Mix write dominance means that the sum of write and read to write switch patterns is larger than the sum of read and write to read switch patterns.

$$t_{rtw} + t_{write} > t_{wtr} + t_{read}$$
$$t_{read} < t_{write} - t_{wtr} + t_{rtw} \tag{5.6}$$

Figure 5.8 shows the range of dominance types on a line. In this figure t_{read} is scaled up and down while keeping t_{write}, t_{wtr}, and t_{rtw} fixed.

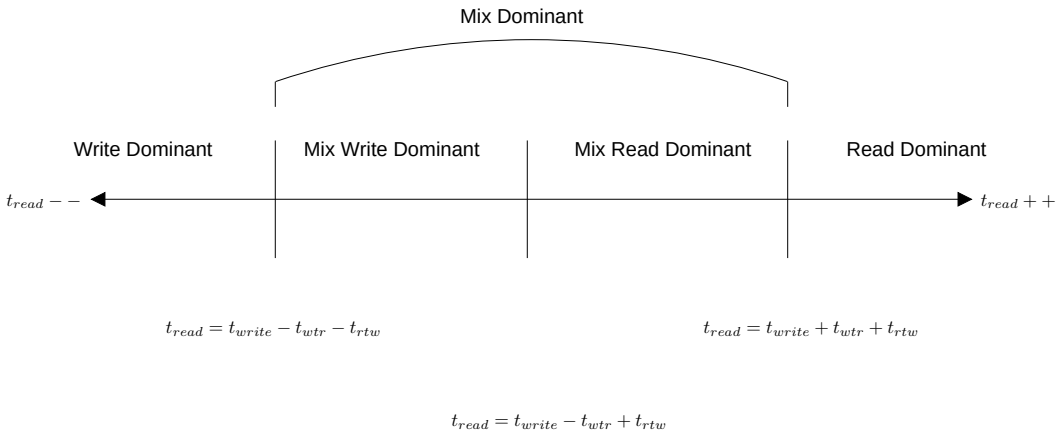

Figure 5.8: Dominance scale viewed on a line.

5.4 Efficiency Calculations

As mentioned in Section 3, the efficiency of a memory controller is very important. It is a major component of the net bandwidth equation shown in Equation (3.5). Therefore, in order to accurately calculate the bandwidth provided by the memory controller, we need to know the efficiency values. Now that we have defined the tools used by a memory controller to interact with an SDRAM, we can precisely compute the various efficiencies. Additionally, the efficiency model of Woltjer that is used by Predator is extended to accommodate pattern sets where the read and write patterns are of different lengths. In Section 3, we concluded that it was difficult to compute some of the efficiencies at design time due to not knowing the traffic. We bypass this restriction now because we have interleaving patterns with fixed access granularity, and shift all uncertainty into data efficiency.

5.4.1 Bank Efficiency

From Section 3.3, we saw that the bank efficiency was a way to incorporate the loss in cycles incurred as a result of SDRAM access time variability.

The bank efficiency is now computed as the number of cycles that there is data on the bus (*TransferCycles*), as defined in Equation (5.7), divided by a pattern length determined by the pattern set dominance type as seen in Equation (5.8). *DataRate* is the number of words per clock cycle, $BurstSize$ is the number of words per read or write command, and $BurstCount$ is the number of read or write bursts sent to a particular bank. The reason that we divide by the average pattern length in Equation (5.8) in the case of mix dominance is because in the worst-case we will have equal amounts of both types of patterns being executed.

$$TransferCycles = \frac{BurstCount * BurstSize * NumberOfBanks}{DataRate} \tag{5.7}$$

$$e_{bank} = \begin{cases} \frac{TransferCycles}{t_{read}} & \text{if read dominant} \\ \frac{TransferCycles}{t_{write}} & \text{if write dominant} \\ \frac{TransferCycles}{\frac{t_{read}+t_{write}}{2}} & \text{if mix read or mix write dominant} \end{cases} \tag{5.8}$$

The calculations are now possible because the addresses of requests coming into the memory controller do not affect access times. Due to the interleaving nature of our patterns, every access will activate all banks before data is read, and all banks are in an idle state when a pattern begins to execute. Thus, access times become static per pattern.

5.4.2 Data Efficiency

As shown in Section 3.2, the equation for determining the data efficiency is dependent on the request size of a requestor and the granularity of the memory access pattern. Now that the read and write pattern has been defined, and considering that it is the only way in which a requestor may access data, we can compute specifically the data efficiency.

$$\begin{aligned} granularity_{pattern} = {} & BurstCount * NumberOfBanks \\ & * BurstSize * DataBusWidth \end{aligned} \tag{5.9}$$

Therefore Equation (3.2) becomes

$$e_{data} = \frac{size_{request}}{granularity_{pattern}} \tag{5.10}$$

5.4.3 Switching Efficiency

When a pattern set exhibits read dominance or write dominance, it implies that the worst-case scenario occurs when only read patterns or only write patterns are being executed. This means that there will be no switches in the worst-case, and therefore the switching efficiency will be 100%.

If the pattern is mix dominant, the switching efficiency is computed as the time it takes to execute a read and write pattern divided by the time taken to execute a read, read to write switch, write, and write to read switch patterns. This is due to the fact that in the worst-case read and write patterns are alternated at every opportunity.

$$e_{switch} = \begin{cases} 1 & \text{if read or write dominant} \\ \frac{t_{read}+t_{write}}{t_{read}+t_{write}+t_{wtr}+t_{rtw}} & \text{if mix read or mix write dominant} \end{cases} \tag{5.11}$$

5.4.4 Refresh Efficiency

In order to compute the refresh efficiency, we need to know the worst-case request time that can occur. Now that this value is easily computable, we can determine the refresh efficiency by taking the amount of time it takes for a refresh pattern to execute, and dividing it by how frequently that refresh actually occurs. All current DDR-SDRAMs have a refresh interval ($tREFI$) of 7800ns at normal operating temperatures. However, to compute the refresh efficiency, we need to know the worst-case request time that can occur. The reason we need to calculate $LongestRequestTime$ is that a refresh pattern may need to be scheduled right after a read or write pattern has been scheduled. $LongestRequestTime$ is defined to be the sum of a switching pattern and an access pattern because in the worst-case, the opposite access pattern has just been executed and thus be must switched. To make sure that the refresh interval is not violated, we rewrite the refresh interval to be $LongestRequestTime$ nanoseconds shorter, as shown in Equation (5.12).

$$\begin{aligned} LongestRequestTime &= \max(t_{read} + t_{wtr}, t_{write} + t_{rtw}) \\ RefreshPeriod &= tREFI - LongestRequestTime + t_{ref} \end{aligned} \tag{5.12}$$

$$e_{refresh} = 1 - \frac{t_{ref}}{RefreshPeriod} \tag{5.13}$$

5.5 Latency Calculation

As mentioned in the problem statement in Section 1.3, we also want to guarantee a bound on worst-case latency. Now that we have precisely defined what the memory patterns are, we may now use them to analytically derive the worst-case latency.

The first latency equations that will be shown will compute the worst-case latency assuming that there are α other requests interfering.

$$latency(\alpha) = \begin{cases} t_{read} * \alpha & \text{if read dominant} \\ t_{write} * \alpha & \text{if write dominant} \\ \lceil \frac{\alpha+1}{2} \rceil * t_{wtr} + \lceil \frac{\alpha}{2} \rceil * t_{read} + \lceil \frac{\alpha}{2} \rceil * t_{rtw} + \lfloor \frac{\alpha}{2} \rfloor * t_{write} & \text{if mix read dominant} \\ \lceil \frac{\alpha+1}{2} \rceil * t_{rtw} + \lceil \frac{\alpha}{2} \rceil * t_{write} + \lceil \frac{\alpha}{2} \rceil * t_{wtr} + \lfloor \frac{\alpha}{2} \rfloor * t_{read} & \text{if mix write dominant} \end{cases} \tag{5.14}$$

However, the above equations assume that no refreshes will occur, and this is not a valid assumption while using DRAMs. Therefore the equations are transformed below to account for refreshes, with ϕ defined in Equation (5.18) for clarity.

Equation (5.17) specifies the maximum number of refreshes that can occur with α requestors interfering. It is broken up into 4 pieces as the worst-case scenario changes with dominance. Once this equation is obtained, we can multiply it with t_{ref} as shown in Equation (5.16) to determine how much time refreshes by themselves can take. We then add this to the latency calculated in Equation (5.14) to obtain the final latency equation, shown in Equation (5.15).

$$TotalLatency(\alpha) = RefreshTime(\alpha) + latency(\alpha) \tag{5.15}$$

$$RefreshTime(\alpha) = t_{ref} * NumberRefreshes(\alpha) \tag{5.16}$$

$$NumberRefreshes(\alpha) = \left\lceil \frac{latency(\alpha)}{\phi} \right\rceil \tag{5.17}$$

$$\phi = \begin{cases} RefreshPeriod - t_{read} & \text{if read dominant} \\ RefreshPeriod - t_{write} & \text{if write dominant} \\ RefreshPeriod - t_{read} - t_{wtr} & \text{if mix read dominant} \\ RefreshPeriod - t_{write} - t_{rtw} & \text{if mix write dominant} \end{cases} \tag{5.18}$$

5.6 Optimality

Now that we have defined how to calculate various efficiencies, latencies, and net bandwidth, we need to define what an optimal pattern set is.

Definition 5.4 (Optimality) *An optimal pattern set for a given memory specification is the set of valid patterns that provide the maximum net bandwidth.*

The reason why net bandwidth is chosen is two-fold. The first benefit of using this is that the system can support the addition of new requestors more easily, as there is more

net bandwidth to distribute. Alternatively, the extra net bandwidth can be redistributed to the requestors with the idea that with more bandwidth allotted to them, they can complete their tasks sooner. In Section 6.2, we see precisely how these patterns are determined.

6 Algorithm Approaches

Three heuristic algorithm approaches are explored in this section: branch and bound, as-soon-as-possible scheduling (ASAP), and bank scheduling. These approaches explore the trade-offs between execution time of the algorithm and amount of net bandwidth offered by the resulting patterns. They are all heuristics as computing optimal patterns is too time intensive. The usefulness of the algorithms is measured by the following aspects.

- Speed of the algorithm
- Net bandwidth offered by the resulting patterns

6.1 Pattern Generation Design Decisions

When generating the patterns as shown in the following sections, we made 4 design decisions regarding pattern construction. First, we assume that shorter access patterns offer more bandwidth than longer patterns. Secondly, the algorithms will only focus on creating access patterns without regard for auxiliary patterns. Thirdly, we always place an activate command in the first pattern slot. Lastly, we let the algorithms treat all banks equally.

1. We justify the first decision by examining how pattern dominance is related to worst-case scenarios.

   (a) Most importantly, shorter access patterns are always better in terms of latency and bandwidth, when a pattern set exhibits read or write dominance. For latency, a shorter pattern means that we can have more interfering requestors without missing deadlines. For the bandwidth aspect, we see that shorter patterns are better due to the effect of bank efficiency on net bandwidth as shown in Equations (5.7) and (3.5). When we have a fixed $BurstCount$, there is always the same amount of transfer cycles, and therefore having a shorter pattern length always results in higher bank efficiency, and thus a higher net bandwidth.

   (b) In the case where a pattern exhibits mix dominance, we use the fact that shorter patterns are better as a heuristic. The reason why we cannot guarantee that the shortest access patterns provide the most bandwidth in mix dominance pattern sets is that it is theoretically possible for a pattern set to have an increase in access pattern lengths that results in a larger decrease in the switching pattern lengths. In the experimental cases where this phenomenon has been observed, the pattern set has always been read or write dominant.

2. Due to the above decision regarding shortest access patterns offering more bandwidth, the algorithms will focus only on creating shortest possible access patterns without regard for auxiliary patterns. In order to create optimal pattern sets for mix dominant patterns we would need to compute the auxiliary patterns for every combination of access patterns at all pattern lengths. This would take an inordinate amount of time, which violates our requirement for algorithms to produce pattern sets within 48 hours.

3. The third decision we motivate with the following.

   (a) First, we consider the case where we have NOP commands before any activate commands present in a pattern, and the NOP commands are not necessary for the pattern to be valid. In this case, we would remove the NOP commands with the justification given above regarding shorter patterns always being better.

   (b) In the second case, we consider the situation where we have NOP commands before any activate commands in a pattern, and without the NOP commands present the pattern is no longer considered valid. In this case the only possible reason why the removal of NOPs causes pattern invalidity is because the tRC requirement is no longer met, or the precharge to activate constraint is no longer met.

      i. In the case where tRC is no longer met, by shifting these NOP commands to the end of pattern, we have not altered the length of the pattern, while still maintaining the requirement of tRC.

      ii. If the pattern invalidity would be caused by the precharge to activate constraint being violated, we see that by shifting NOPs to the end of the pattern also does not violate the constraint for the same reason as the case for tRC.

The effect of moving NOP commands from the back of an access pattern to the front of an access pattern can change the size of individual switching patterns as seen in Figure 6.1. Here, we enforce a minimum delay of 12 NOP cycles between a read pattern and a write pattern, and a delay of 7 NOP cycles between a write pattern and a read pattern. In the upper illustration, the access patterns have their NOP commands at the front of the pattern while the lower illustration has its NOP commands at the end. When determining the length of a switching pattern, we must enforce the minimum number of cycles between the last access of an access pattern and the first access of the following As shown in the figure, the difference in lengths between the two switching patterns is shifted from one switching pattern to the other, in order to maintain the minimum number of cycles between accesses. Because the overall sum of the two switching patterns does not change, we see that moving NOPs does not influence the bandwidth of a given pattern set. It may have a small effect on latency however, due to the changes in t_{rtw} and t_{wtr} lengths. These lengths affect latency as shown in Equation (5.14).

The refresh pattern is also affected by the location of NOP commands in the access patterns. The length of the refresh pattern is partially determined by the number

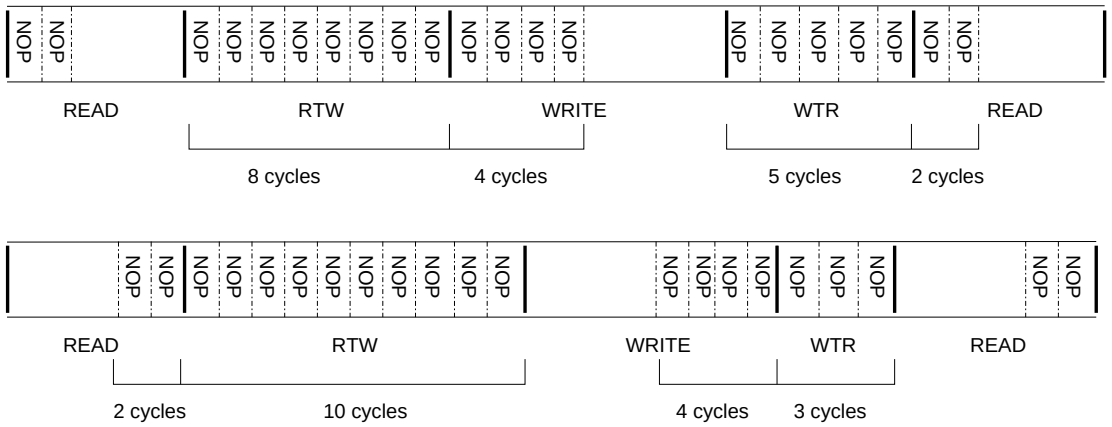

Figure 6.1: Illustration of moving NOPs from back to front of pattern.

of cycles from the last precharge of the previous access pattern. An access pattern with its NOPs at the beginning has its precharges earlier compared to a pattern with its NOPs at the end, thus a few cycles can be saved on the refresh pattern.

Thus, by having NOP commands at the end of the pattern, we do not increase the length of the access patterns, we give the algorithms a base case to start from, we save cycles in the refresh pattern, and we significantly reduce the design space of the branch and bound algorithm.

4. A final design assumption was that all banks are treated as equal by all algorithms. This means that when the algorithm is looking at available commands to be scheduled at a certain cycle, it does not consider commands to individual banks, but only the commands themselves. When a command is selected then the algorithm determines the proper target bank for the command. This was done in order to reduce the design space.

6.1.1 Access Pattern Termination

As we have mentioned above we search for the shortest access patterns, which means that it becomes critical to know when an access pattern is complete. We determine access pattern completeness with the following criteria.

- All commands have been scheduled
- tRC constraint satisfied
- Data bus constraints satisfied
- Precharge constraints satisfied

The first criterion is stems from the obvious fact that we do not want to end a pattern before all of the commands we must insert have been scheduled. The set of commands to be scheduled is determined with the following equations.

$$
\begin{aligned}
numActivates &= numBanks \\
numReads &= BurstCount * numBanks \quad \text{For Read Patterns} \\
numWrites &= BurstCount * numBanks \quad \text{For Write Patterns}
\end{aligned}
$$

The *tRC* constraint specifies the minimum amount of cycles that must occur between successive activate commands to the same bank. Due to the fact that we allow an access pattern to be scheduled after itself, this constraint is effectively the minimum theoretical access pattern length. As we saw in Section 5.1.1, we allow a read pattern to be scheduled directly after itself, as well as a write pattern to be scheduled directly after itself. Therefore, we must make sure that in these cases where a pattern is scheduled twice in a row, that the minimum number of cycles between their activate commands to each bank has been satisfied.

Data bus constraints are that we must ensure that data that is produced on the data bus does not conflict with data being produced on a data bus from a future access pattern. To prevent this from happening we must examine the *burstSize* parameter, and the *dataRate* parameter. These parameters refer to the number of words produced per read or write command, and the number of words per clock cycle, respectively. By knowing these, we can determine how many clock cycles must separate the last access command of one pattern, and the first access command of the following pattern.

The last criterion refers to the fact that before a bank can be activated, it must have been precharged already. Therefore it becomes important to know where the precharge actually occurs in an access pattern. We determine this in two ways, as the read and write constraints on a precharge are different.

Under a read pattern, the cycle of precharge for a particular bank is computed by finding the cycle of the last read command to that bank, *cycleLastRead*, and the cycle of the activate to the bank, *cycleACT*, and evaluating Equation (6.1), which comes from [8, 9].

$$
\begin{aligned}
prechargeCycle = \max(&cycleLastRead + tRTP, \\
&cycleACT + tRAS, \\
&cycleLastRead + \frac{burstSize}{dataRate} + \max(tRTP, 2) - 2)
\end{aligned} \tag{6.1}
$$

We say that for read and write patterns, a bank has its precharge constraint met if the cycle of its precharge is at least *tRP* cycles before the next activate occurs. Under a write pattern, the idea remains the same yet the parameters involved change. The precharge cycle for a write pattern is defined in Equation (6.2).

$$
prechargeCycle = cycleLastWrite + tWL + \frac{burstSize}{dataRate} + tWR \tag{6.2}
$$

Then we say the precharge requirement for write patterns is satisfied if precharge cycle is at least *tRP* cycles before the next activate command occurs.

6.1.2 Auxiliary Pattern Computation

As we saw in the previous section, the heuristics of the pattern generation approaches focus on access patterns. We treat the auxiliary patterns as being dependent on the access patterns, which allows us to reduce the size of the design space.

The way to compute the refresh pattern length is as follows. First we determine which access pattern has its precharge cycle happen the latest, as the refresh command is not allowed to occur until a minimum number of cycles has passed since the last precharge. The size of the refresh pattern is defined in Equation (6.3).

$$t_{ref} = \begin{cases} tRP + tRFC + (t_{read} - prechargeCycle) & \text{if } t_{read} - prechargeCycle < \\ & \quad t_{write} - prechargeCycle \\ tRP + tRFC + (t_{write} - prechargeCycle) & \text{if } t_{read} - prechargeCycle \geq \\ & \quad t_{write} - prechargeCycle \end{cases} \tag{6.3}$$

The switching patterns have constraints similar to that of the refresh pattern in that there is a minimum number of cycles that must occur between two commands. In the case of switching patterns, there are two constraints, one for each type of switch. This number of cycles is defined in the memory specification [8, 9]. However, as the cycle of the last non-NOP command in an access pattern is not necessarily equal to the pattern's length, we use Equations (6.4), and (6.5) to determine the size of the switching patterns. *delay* is the minimum number of cycles that must occur between the last access of one pattern and the first access of the following pattern.

$$t_{rtw} = \max(delay - (cycleFirstWrite + t_{read} - cycleLastRead), 0) \tag{6.4}$$

$$t_{wtr} = \max(delay - (cycleFirstRead + t_{write} - cycleLastWrite), 0) \tag{6.5}$$

6.2 Branch and Bound

This algorithm is used to find the minimum length access patterns by searching over all possible access patterns according to the design decisions listed above. When the pattern set found exhibits read or write pattern dominance, this algorithm will produce optimal results. If the pattern set generated is mix dominant, then we cannot guarantee optimality. The reason we guarantee optimality in the case of read and write dominance is because switching patterns do not affect the worst-case bandwidth offered. Therefore only access patterns are of concern, and shorter access patterns will always offer higher e_{bank} than longer options, resulting in a higher net bandwidth, as shown in Equation (3.5). When switching patterns do affect the scenario, we must compute the auxiliary patterns for every combination of access patterns in order to guarantee optimality, because the shortest access patterns do not guarantee the shortest switching patterns. The amount of time taken to compute this is inordinate. Therefore we use the heuristic.

The branch and bound algorithm works by starting a read pattern at cycle 0, and then looking to see what are the possible commands that could be scheduled at cycle 1. It then creates one copy of the schedule for each of the available commands respecting the memory constraints, and then appends to the end of the schedule the available command in question. The algorithm repeats this process until the pattern is complete, at which point it stores the pattern into a container and continues its search in remaining directions. The same process is repeated for the write pattern. In essence, it is a depth first tree traversal. An example of the tree being searched is shown in Figure 6.2.

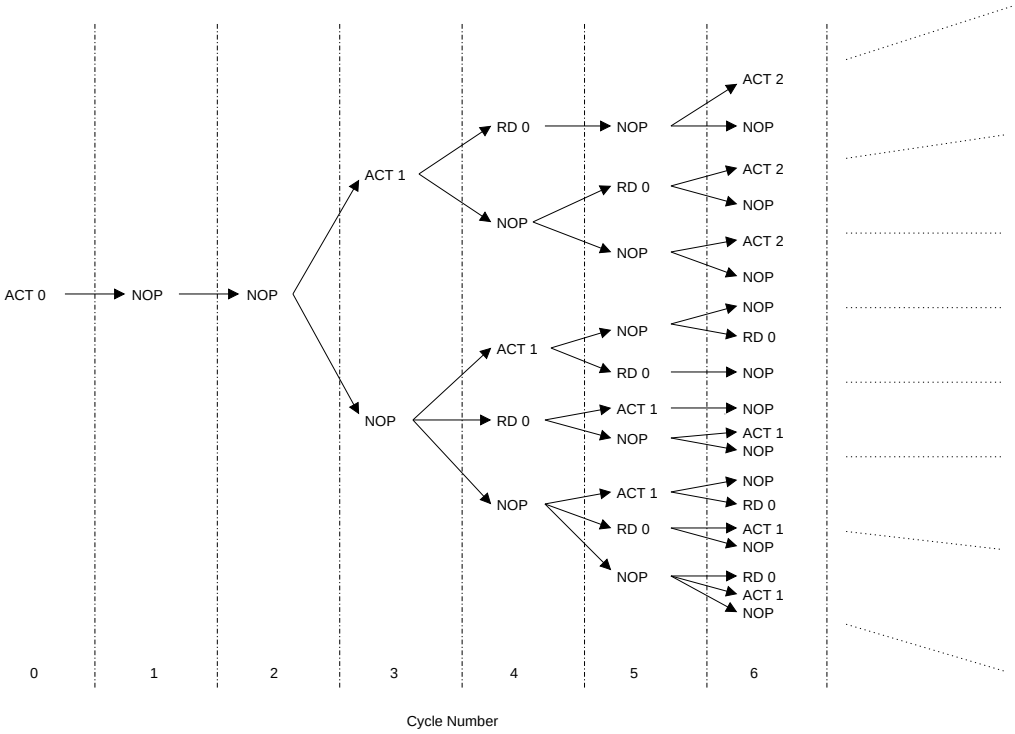

Figure 6.2: Example of command tree.

When each of the optimizations below are implemented, there is at least 1 pattern of a fixed length found for the read and write patterns. In order to select the best read and write pattern, the read patterns and write patterns in the set returned by the algorithm must be searched. The access patterns that contain their last scheduled non-NOP command at the earliest point in the pattern are selected. Once the read and write patterns have been selected, they are used to compute the switching patterns and the refresh pattern. The reason why the patterns with their last commands scheduled the earliest are used is because we are able to possibly take advantage of NOP cycles at the end of patterns to reduce the size of switching patterns. This results in a better switch efficiency as shown in Equation (5.11), which in turn provides a higher net bandwidth, shown in Equation (3.5).

Now that we have defined how to arrive at a pattern set with a fixed $BurstCount$, we need a way to determine which $BurstCount$ should be chosen. This problem is addressed

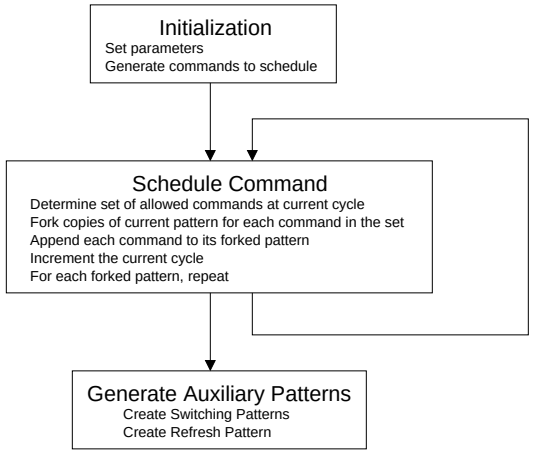

Figure 6.3: Flow diagram of the branch and bound algorithm.

in Section 7.

6.2.1 Hard Ceiling

To reduce the run time of the algorithm, some optimizations had to be made. The most important one was to put a hard ceiling on the pattern length. This prevents the algorithm from searching for infinite length patterns, and at the same time significantly cuts down on the exploration space. These changes result in significant decreases in memory and run time.

6.2.2 Sliding Ceiling

This optimization starts where the hard ceiling leaves off. Once a valid pattern is found, its length is recorded and then used as the current maximum ceiling. Thus, when the algorithm resumes exploring the other branches of the design tree, it first checks to see how long the current path length is. If it is already over the current ceiling, it stops searching the current path. This change also results in significant decreases in run time and memory, as well as a dynamic exploration space.

During the course of this optimization, it is possible that an optimal pattern set is missed. Previously in Section 6.1, it is stated that shortest access patterns do not always offer optimal bandwidth.

6.2.3 Sanity Check

This last optimization works in tandem with the sliding ceiling optimization. Whenever the algorithm wishes to create a copy of a pattern so that it may append a command, it first checks to see if even in the best-case scenario, is it possible for this current pattern's length to be smaller than the current ceiling, based on the remaining commands to be scheduled into the pattern. As with the other two optimizations, this reduces memory use and run time. Pseudo-code is shown for this optimization in Figure 6.4. *numActs*

refers to the remaining number of activate commands to be scheduled, *numAcc* refers to the number of remaining read or write commands to be scheduled.

The first check is based on how many activates are remaining. For example, if there are 3 activate commands remaining at the current cycle, we can guarantee that this pattern cannot be completed until 2 * *tRRD* + *tRCD* cycles have occurred. The 2 * *tRRD* comes from the fact that activates cannot be scheduled within *tRRD* cycles of each other, and there are at least 2 full delays between activates remaining. The second part of the sum is *tRCD*. This is included into the delay because we know if there is at least one activate left, there is also at least one read or write access, and *tRCD* is the minimum delay between an activate command and a read or write command.

The second check looks purely at how many read or write commands are left. If there are 3 read commands remaining in the pattern, and we know that it there is a minimum delay of $burstSize/dataRate$ between read commands, then we know that there must be 2 full delays of $burstSize/dataRate$ cycles.

if $numActs = 0$ **then**
  $len1 \leftarrow currentCycle$
else
  $len1 \leftarrow currentCycle + (numActs - 1) * tRRD + tRCD$
end if
if $numAcc = 0$ **then**
  $len2 \leftarrow currentCycle$
else
  $len2 \leftarrow currentCycle + (numAcc - 1) * burstSize/dataRate$
end if
if $\max(len1, len2) < currentCeiling$ **then**
  $SchedulePossible$
else
  $ScheduleNotPossible$
end if

Figure 6.4: Pseudo-code of sanity check optimization.

6.2.4 Conclusions

The benefit of this algorithm is that it always finds the shortest access patterns. This translates to optimal net bandwidth in the case of read or write dominant pattern sets, but not necessarily optimal net bandwidth in the case of mix dominance. These benefits come at the cost of run time.

Despite the optimizations that reduce the run time of the algorithm, with high frequency SDRAMs, the complexity begins to show itself. Timing constraints are larger, and thus there is more space in between non-NOP commands, which creates many more options when the algorithm is making copies of the current schedule. Furthermore, as the *BurstCount* increases, there are more commands to be scheduled, which also significantly increases the complexity of the design space. In Figure 6.5 the growth in numbers of valid patterns starting with an activate command at different lengths is demonstrated.

A read and write pattern length of 32 is shortest provided by the branch and bound algorithm, while 37 is the length provided by the ASAP algorithm.

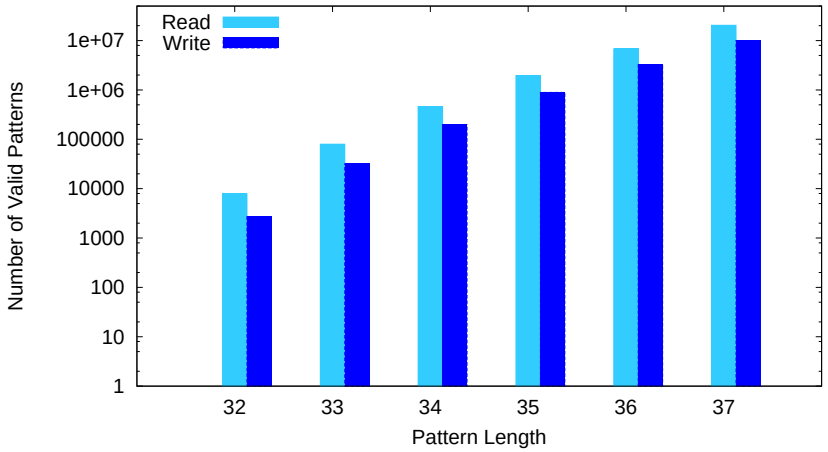

Figure 6.5: Number of valid patterns at *BurstCount* 2 for a DDR2-400 SDRAM device

For practical purposes, the algorithm becomes unusable for online use whenever the *BurstCount* is greater than 2. Offline use is still practical until a DDR3-1600 is used with *BurstCount* equal to 2. After this point, the run time moves into months and years. As a result, this algorithm may be used to produce a limited database offline, for use as a comparator with new algorithms being developed. However, the problem of not being able to generate more complicated patterns remains, and thus a new algorithm must be developed.

6.3 As Soon As Possible Scheduling

This algorithm works by linearly creating a read pattern starting from cycle 0. The algorithm looks at the current cycle to see all of the commands allowed to scheduled at this cycle based on the timing constraints imposed by previous commands. If there is more than one available command, a priority scheme is used to pick one. The priority scheme is that read and write commands are selected over activate and NOP commands, and activate commands are chosen over NOP commands. The pseudo-code for this algorithm is shown in Figure 6.6, and a flowchart is shown in Figure 6.7.

6.3.1 Conclusions

This major benefit achieved from this algorithm is that it runs extremely fast (less than 1 second) for any value of *BurstCount*. However, this comes at the cost of net bandwidth. As seen in Figures 6.11, 6.12, 6.13, 6.14, the patterns provide around 80% of the bandwidth that is offered by branch and bound algorithm.

The reason this occurs is that it is not always beneficial to schedule a command as soon as it becomes available. After a write command has been issued to the SDRAM,

```
currentCycle ← 0
pattern[currentCycle] ← {ACT − 0}
while !isValid(pattern) do
    availableCmds ← getAllowedCmds(pattern, currentCycle)
    cmdToSchedule ← pickBestCmd(availableCmds)
    pattern[currentCycle] ← cmdToSchedule
    currentCycle + +
end while
```

Figure 6.6: Pseudo-code of ASAP algorithm

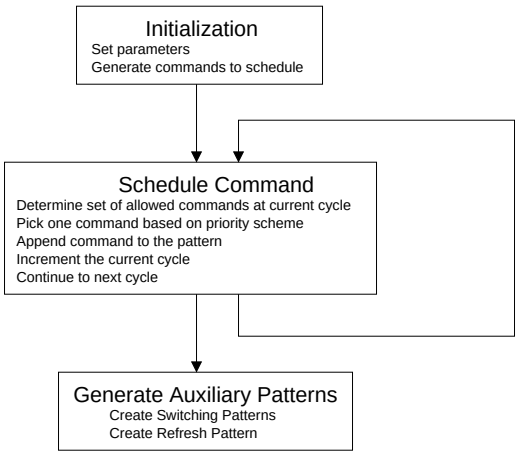

Figure 6.7: Flowchart of the ASAP algorithm

time is required after the completion of the write to precharge the data in the row buffer. If the algorithm does not account for this, cycles can be wasted.

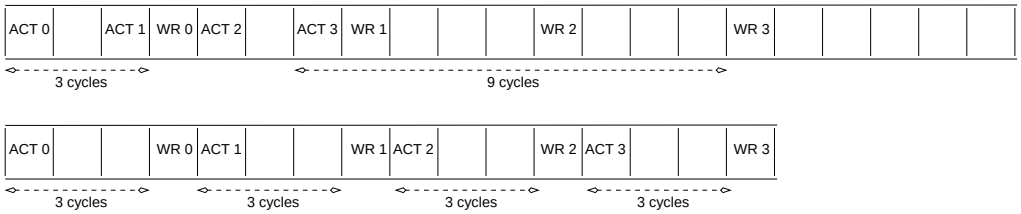

Figure 6.8: Example of a pattern generated by the ASAP algorithm (top), and a pattern generated by the branch and bound algorithm (bottom).

Examine the situation where two write patterns are to be scheduled after each other and a bank requires 10 cycles after a write command has been issued before it is allowed to be activated again. We also say that the length of the pattern is 16 cycles, the last command is a write command, and that we have the option of scheduling the last activate command at cycle 8 or at cycle 12 of the pattern without changing the overall length of the pattern.

In this situation, we see that there is potential conflict with the following pattern

when the activate is scheduled at cycle 8, thus two extra NOP commands are required to be appended to the pattern to satisfy the timing requirement, thereby increasing the length, decreasing bandwidth, and increasing latency.

In conclusion, this algorithm cut the run time drastically compared to the branch and bound algorithm, yet the difference in net bandwidth provided was too significant to ignore. A goal of these algorithm approaches is to provide a high amount of bandwidth and this approach's net bandwidth is too low. Therefore a new algorithm was developed with the idea that we were willing to sacrifice a bit in terms of run time for gains in net bandwidth.

6.4 Bank Scheduling

This algorithm was born from the idea that a bank's activate command should be kept as close as possible to its respective read/write commands. This was done in order to avoid the problems that the ASAP algorithm ran into, shown in Figure 6.8.

The procedure for this algorithm is to first plot an entire schedule for only the first bank, with all commands placed as closely to each other as possible. Once this is completed the algorithm then plots the schedule for the next bank. The way this part works is that the algorithm looks for the last read or write command scheduled in the previous bank. The algorithm then places the current bank's read or write command in the next allowed slot, assuming that the slot *tRCD* cycles back in the schedule is allowed to have an activate command placed. The timing constraint *tRCD* is the minimum number of cycles between an activate command to a bank and a read or write command to the same bank. If the pattern does not allow an activate command to be scheduled there, then the read/write command is moved forward one cycle, and the algorithm tries again. This process is then repeated until all banks have been scheduled. The pseudo-code for this algorithm is shown in Figure 6.9.

```
pattern[0] ← {ACT − 0}
pattern[tRCD] ← {RD − 0}
currentBank ← 1
while !allCmdsScheduled() do
   cycleLastRead ← getLastRead(pattern)
   targetCycleRead ← cycleLastRead + BL/2
   while !activateAllowed(targetCycleRead, tRCD) do
      targetCycleRead + +
   end while
   pattern[targetCycleRead] ← {RD − currentBank}
   pattern[targetCycleRead − tRCD] ← {ACT − currentBank}
   currentBank + +
end while
```

Figure 6.9: Pseudo-code of bank scheduling algorithm

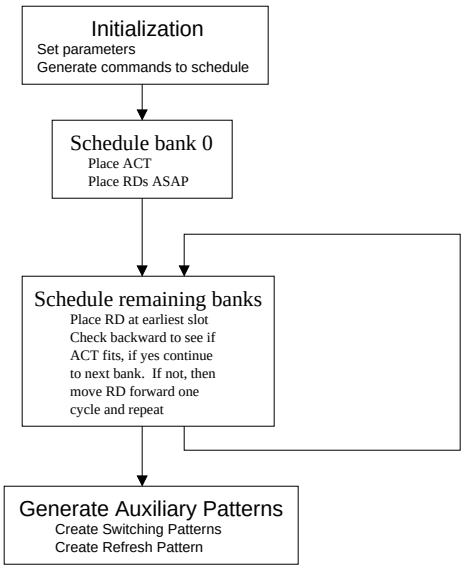

Figure 6.10: Flow diagram of the bank scheduling algorithm

6.4.1 Sliding Activate

An alternative approach attempted was to let the ACT command slide backwards until an empty slot is found, as opposed to the READ/WRITE command sliding forwards immediately if the slot *tRCD* cycles behind is not empty. However, the results of this implementation were at best the same, and with some memory specifications this implementation proved to yield poorer results than the original implementation.

6.4.2 Conclusions

After running this algorithm, we find that the run time is comparable to that of the ASAP algorithm, and therefore suitably fast. More importantly, the closeness to the branch and bound bandwidth offerings are much more reasonable. As seen in Figures 6.11, 6.12, 6.13, 6.14, the maximal difference between the two is very small. As a result, this algorithm has been selected for use with Predator. We discuss the integration of this algorithm in Section 7.

6.5 Results

Plotted in Figures 6.11, 6.12, 6.13, and 6.14, we see comparisons of guaranteed bandwidth offered by pattern sets generated by the branch and bound, ASAP, and bank scheduling algorithms. The charts are based on the net bandwidth defined in Equation (3.5).

The bank scheduling approach and the branch and bound approach guarantee similar net bandwidths over different memory specifications. Also observed is the lower net bandwidth guaranteed by the ASAP approach. For all memories tested, branch and bound always provided the best results in terms of offered bandwidth. Bank scheduling always matched, or provided slightly less bandwidth than the branch and bound

approach. The ASAP approach always provided the worst results of the three. The lack of data for *BurstCount* 4 for the branch and bound approach is due the algorithm not being able to calculate such complex patterns. A *BurstCount* of 3 is included in these graphs for demonstration purposes only. In practical applications, only powers of 2 are used for *BurstCount* values as they are the only numbers that may be translated into a memory map. An interesting observation drawn from these results is that offered bandwidth saturates at *BurstCount* 3 for the memory types tested.

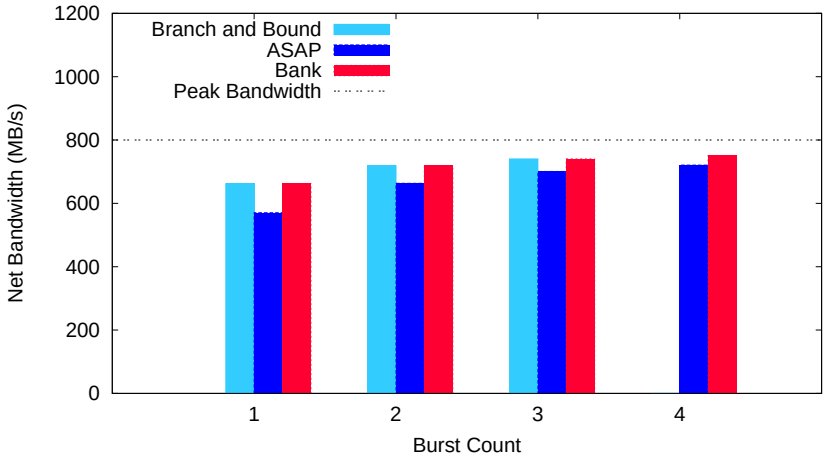

Figure 6.11: Comparison of net bandwidth guaranteed by algorithms for a DDR2-400 SDRAM.

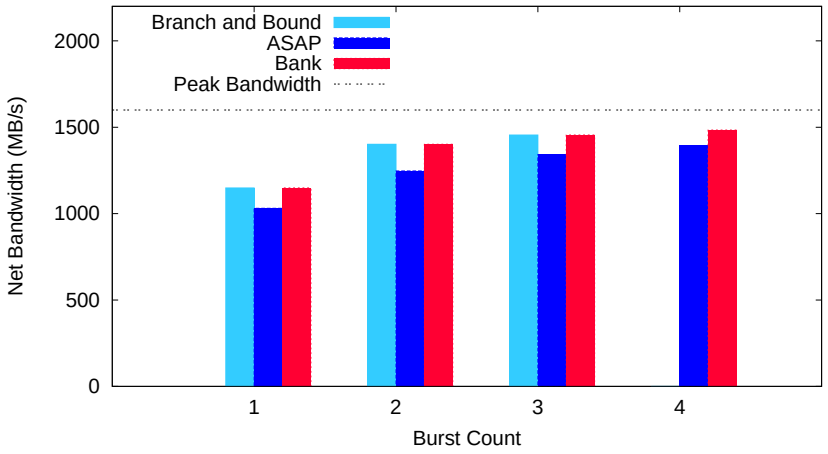

Figure 6.12: Comparison of net bandwidth guaranteed by algorithms for a DDR2-800 SDRAM.

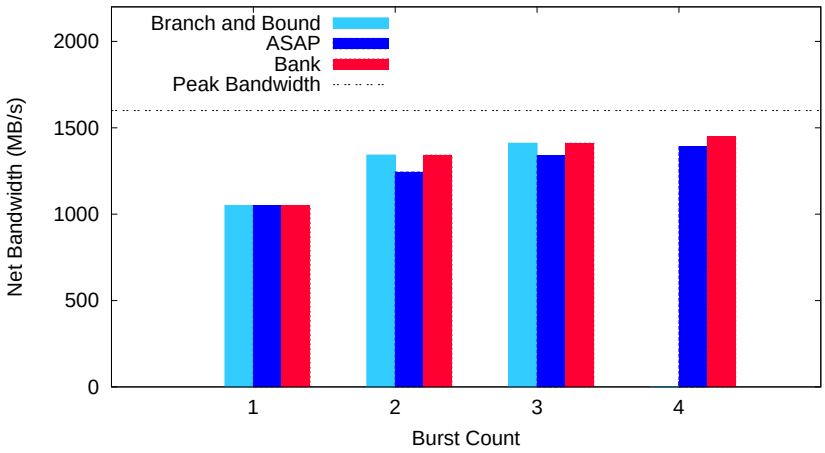

Figure 6.13: Comparison of net bandwidth guaranteed by algorithms for a DDR3-800 SDRAM.

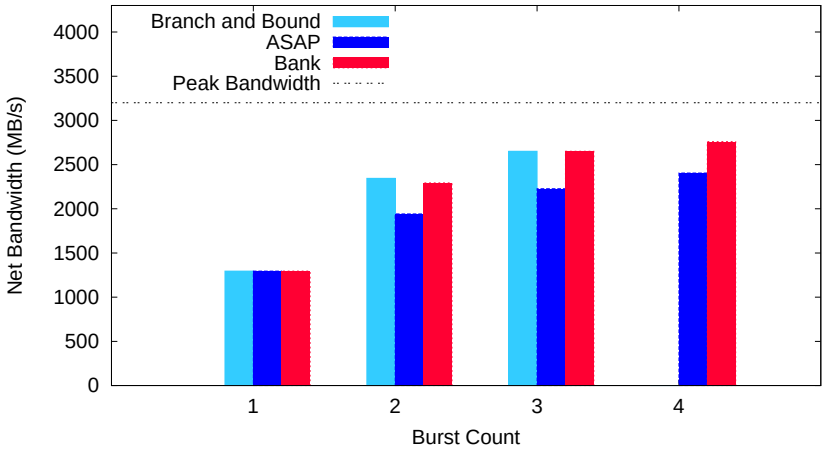

Figure 6.14: Comparison of net bandwidth guaranteed by algorithms for a DDR3-1600 SDRAM.

7 Algorithm Context

In this section, the overall design flow of the memory controller architecture is examined. Specific attention is given to the integration of the pattern generation algorithm into the Predator configuration flow.

7.1 Tooling Flow Overview

This section provides an overview of the various stages in the offline tool flow. First, the initialization of requestors is considered, followed by a description of the pattern generator. Lastly, the bandwidth allocator and priority assigner will be discussed.

7.1.1 Requestors

Requestors have their request sizes, latency requirements, and bandwidth requirements initialized. The bandwidth requirements require special attention as they are not initialized to the requestor's specification, but are initialized to a normalized value. By examining the pattern granularity and the request size of the requestor, we are able to determine e_{data} as shown in Equation (3.2). Because we now know which percentage of bandwidth the requestor actually uses, we scale up its requested bandwidth, such that the amount it receives meets its requirements. As an example, if a requestor has a bandwidth requirement of 200 MB/s, and a request size of 8 bytes, and the pattern set being used has a granularity of 16 bytes, then the e_{data} will be 50%. Therefore, we scale the bandwidth requirement up by a factor of 2, to 400 MB/s. We call this value the *normalized bandwidth* requirement. A requestor specification can be seen in Appendix D.

7.1.2 Pattern Generator

The pattern generator is responsible for creating pattern sets. It takes as inputs the following:

- Memory Specification
- $BurstCount$

The memory specification refers to the timings of the constraints detailed in Appendix A, and an example can be seen in Appendix E. These constraints are very significant in determining how the patterns are allowed to be built. As detailed below, $BurstCount$ is iterated over, and is therefore needed as an input into the pattern generator so it knows which pattern to produce.

The outputs of the pattern generator are as follows:

- Read Pattern
- Write Pattern
- Read To Write Switch Pattern
- Write To Read Switch Pattern
- Refresh Pattern

These patterns comprise a pattern set.

7.1.3 Bandwidth Allocator

We consider the case where a credit-controlled static-priority arbiter is used [3]. This arbiter's configuration has two steps: bandwidth allocation and priority assignment.

In the bandwidth reservation mechanism of the arbiter, the bandwidth of the memory is allocated to the various requestors. The amount of bandwidth requested is approximated by the allocator as closely as possible. However, due to finite precision, discretization of the bandwidth occurs, which results in the possibility of the allocated bandwidth exceeding the net bandwidth offered by the memory controller. A second issue that is worthy of concern is that the sum of the requested bandwidth of the requestors may simply be more than the net bandwidth offered by the memory controller. Both of these issues result in negative non-allocated bandwidth, and are taken into account as shown in the pseudo-code of the algorithm, in Figure 7.1.

7.1.4 Priority Assigner

The priority assignment stage is responsible for finding a priority scheme that allows for all requestors latency requirements to be met. An optimal priority assignment algorithm is used to accomplish this, and it runs in polynomial time [4]. A revision of the latency equation detailed in Equation (5.15) becomes very useful in this stage.

This equation is needed because it allows us to use a priority assignment algorithm that is independent of the resource. In essence, the equation is a conversion from actual latency requirements, to using more abstract requirements, such that the tooling can be ignorant of the details of the memory being used.

Equation (7.2) is called the reverse latency equation and its purpose is to find the maximum number of read and write patterns that can interfere before a deadline is missed in the worst-case. Here, α is defined as the number of interfering patterns, and $\hat{\Theta}$ is defined as the maximum latency allowed by a requestor specification, and ϕ is defined in Equation (5.18).

$$
\begin{aligned}
\hat{\Theta} &\geq TotalLatency(\alpha) \\
\hat{\Theta} &\geq latency(\alpha) + NumberRefreshes(\alpha) \\
\hat{\Theta} &\geq latency(\alpha) + \left(\frac{latency(\alpha)}{\phi} + 1\right) * t_{ref} \\
\hat{\Theta} &\geq latency(\alpha) + \frac{t_{ref} * latency(\alpha)}{\phi} + t_{ref} \\
\hat{\Theta} &\geq latency(\alpha) * \left(1 + \frac{t_{ref}}{\phi}\right) + t_{ref} \\
\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} &\geq latency(\alpha)
\end{aligned} \tag{7.1}
$$

Now, solving for α we get,

$$
\alpha \leq \begin{cases}
\frac{\hat{\Theta} - t_{ref}}{\left(1 + \frac{t_{ref}}{\phi}\right) * t_{read}} & \text{if read dominant} \\
\frac{\hat{\Theta} - t_{ref}}{\left(1 + \frac{t_{ref}}{\phi}\right) * t_{write}} & \text{if write dominant} \\
\frac{\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{wtr}}{2} - t_{read} - t_{rtw}}{\frac{t_{wtr}}{2} + \frac{t_{read}}{2} + \frac{t_{rtw}}{2} + \frac{t_{write}}{2}} & \text{if mix read dominant} \\
\frac{\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{rtw}}{2} - t_{write} - t_{wtr}}{\frac{t_{rtw}}{2} + \frac{t_{write}}{2} + \frac{t_{wtr}}{2} + \frac{t_{read}}{2}} & \text{if mix write dominant}
\end{cases} \tag{7.2}
$$

The derivations of these equations can be found in Appendix C.

7.2 Integration of Pattern Generator

Now that the algorithm for use in run time pattern generation has been established, the actual integration into the memory controller flow is explained. Figure 7.1 illustrates the pseudo-code of the tooling algorithm, and Figure 7.4 shows the architecture of the memory controller.

As we saw in Section 5.6, there is a pattern set for each value of *BurstCount*. The problem for the generator now becomes which *BurstCount* should be chosen. This is accomplished by running the pattern generator multiple times with an incrementing *BurstCount*, and using a new metric that is defined below, to evaluate which generated pattern set is ideal for a given use case.

We iterate over different values of *BurstCount* because the pattern generator cannot know if the pattern set it has created is ideal until bandwidth allocation and priority assignment have taken place at other values of *BurstCount*. Therefore, depending on

```
BurstCount ← 1
NonAllocBW ← 0
while isNonAllocBWBetter() do
   LatencyFlag ← 1
   patternSet ← generatePatternSet(memSpec, BurstCount)
   allocateBandwidth()
   assignPriorities()
   for requestor in Requestors do
      if requestor.AllocatedLatency > requestor.RequiredLatency then
         LatencyFlag ← 0 // Latency requirement for requestor not met
      end if
   end for
   if LatencyFlag = 0 then
      break
   end if
   BurstCount ← 2 * BurstCount
   NonAllocBW ← computeNonAllocBW()
end while
usePatternSet(patternSet)
```

Figure 7.1: Pseudo-code of integrated pattern generator.

the result of the output of the allocation and priority assignment, the algorithm may need to generate a new pattern at a higher $BurstCount$.

The reason why the value of $BurstCount$ affects the bandwidth is due to the nature of the patterns. Increasing $BurstCount$ can result in a higher net bandwidth offered by the pattern because of better bank efficiency. The number of transfer cycles grows by a factor of the $BurstCount$, while due to the parallelism offered by a bank architecture SDRAM, the length of the pattern typically increases at half that rate. Therefore the bank efficiency increases as shown in Equation (5.7).

However, using a higher $BurstCount$ can also prove detrimental to the net bandwidth, due to its effect on data efficiency. If we imagine a situation where a requestor only wishes to read 4 bytes of memory, and a pattern set has an access granularity of 8 bytes, we can see that by increasing the $BurstCount$ we increase the access granularity even more, which results in a worse data efficiency. As shown in Equation (3.5), a decreasing data efficiency results in lower net bandwidth offered.

Therefore we must iterate over different values of $BurstCount$ in order to find the pattern set that maximizes bank efficiency and data efficiency. In order to do this over different values of $BurstCount$, we need to establish a loop between the pattern generator and the flow into which it is set. The flowchart can be seen in Figure 7.3. To demonstrate the relationship between e_{data} and e_{bank}, a plot of a DDR3-1600 SDRAM memory device has been created in Figure 7.2. In this graph we see how increasing $BurstCount$ improves e_{bank} at the cost of e_{data}. Also plotted is the non-allocated bandwidth. The goal of the integrated pattern generator is to find the value of $BurstCount$ that produces the peak non-allocated bandwidth.

The method that has been implemented is to generate an initial pattern with a

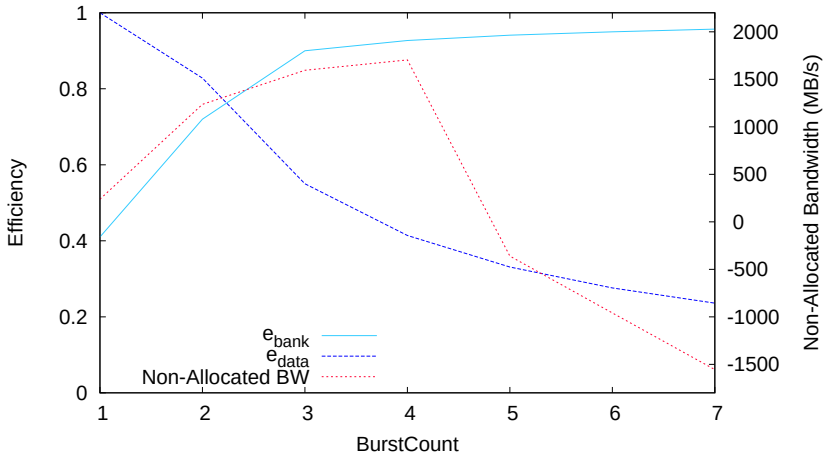

Figure 7.2: Trade-off between e_{data} and e_{bank}.

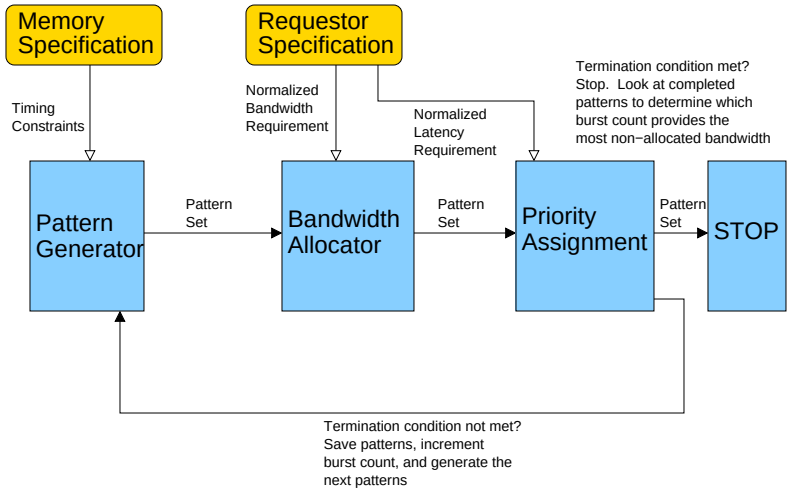

Figure 7.3: Flow of integrated pattern generator

BurstCount of 1. If the requestors' bandwidth requirements are met, then the pattern set generated is sent to the service allocation stage, where the bandwidth is discretized to the various requestors. The patterns are then to the priority assignment stage of the flow.

At the priority assignment stage the arbiter attempts to find a priority scheme for the requestors such that the latency requirements of no requestor are violated. If the arbiter is able to successfully find an assignment where none of the latency requirements are violated, then it means that the generator has found a valid pattern set. Then, the non-allocated bandwidth is computed for the pattern set. If it is less than the value found for the previous *BurstCount*, then the loop is broken. Additionally, if the arbiter is not able to find a priority scheme to fit the pattern set, the loop is broken. Otherwise, the pattern is saved for later consideration, and the loop continues with an incremented

BurstCount.

The loop termination condition is always reached. The non-allocated bandwidth can only decrease when e_{data} has become less than 100% for all requestors, as shown by Equation (7.3). As e_{data} must decrease below 100% as the *BurstCount* rises as shown by Equation (5.10), the loop must be broken.

When the arbiter is unable to find a priority scheme for the pattern set, the loop is broken for the following reason. A pattern with a higher *BurstCount* is always longer than a pattern with a lower *BurstCount*, which also means that higher *BurstCount* patterns take longer to execute. Thus, increasing *BurstCount* can only exacerbate the problems an arbiter may have creating a priority assignment. A second termination condition is implemented to prevent the *BurstCount* from becoming so large that the refresh constraint is violated. Once the termination condition has been met, the pattern set that has the highest offered non-allocated bandwidth is selected.

In Figure 7.4 we see the architecture of the Predator memory controller. Now that we have a proper tool to generate memory pattern sets, we show how the tool is related to the architecture. Once the patterns have been generated, they are loaded into the command generator. This allows the command generator to know which sequence of patterns should be sent to the SDRAM. The priority assignment stage is responsible for creating proper priorities for the requestors, and these priorities are loaded into the scheduler. Finally, the bandwidth allocator is responsible for determining how to divide the available bandwidth for the requestors. This information is used by the rate regulator, which is responsible for enforcing the budgets derived by the bandwidth allocator.

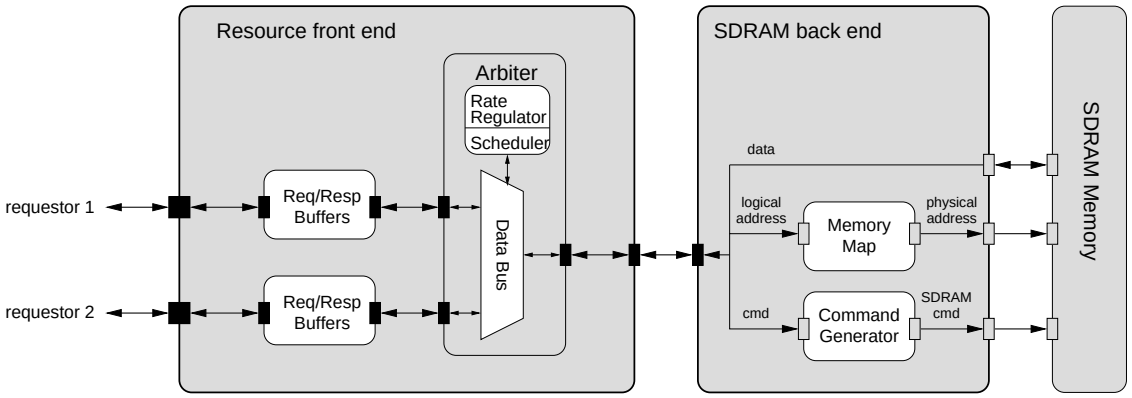

Figure 7.4: Illustration of Predator architecture.

7.3 Non-allocated Bandwidth Calculations

As mentioned above, the criteria for determining whether a given generated pattern set is better than other pattern sets depends on the amount of non-allocated bandwidth. As such, it is important to be able to calculate this value.

The first thing that needs to be computed is the net bandwidth, which is comprised of the product of the efficiencies presented in Section 5.4 and the peak bandwidth,

presented in Section 3.1. Now that we have the total amount of bandwidth offered to the requestors, we need to calculate how much bandwidth does each requestor take.

For each requestor at a specified $BurstCount$, we subtract the normalized bandwidth requirement from the net bandwidth being offered by the memory controller. This value left over is the non-allocated bandwidth. An example is shown in Table 7.3 with hypothetical values of net bandwidth.

$$NonAllocatedBandwidth = NetBandwidth - \sum_{k=1}^{NumRequestors} NormalizedBandwidth_k \tag{7.3}$$

Table 7.1: Example of normalized bandwidth changing with $BurstCount$.

Requestor	*Burst Size*	*Access Granularity*	*Normalized Bandwidth*	*Burst Count*	*Net Bandwidth*
Req. 1 Req. 2	64 128	128	3200MB/s 400MB/s	1	5000MB/s
Req. 1 Req. 2	64 128	256	6400MB/s 800MB/s	2	9000MB/s
Req. 1 Req. 2	64 128	512	12800MB/s 1600MB/s	4	16000MB/s

Thus with this particular example at $BurstCount = 2$, we have improved the non-allocated bandwidth over the previous iteration. Therefore we increment $BurstCount$ and try again. We see at $BurstCount = 4$ that the non-allocated bandwidth is lower than the previous iteration, and thus terminate the loop and choose the pattern set with $BurstCount = 2$.

$$NonAllocatedBandwidth = 5000 - 3200 - 400 = 1400 \quad (BurstCount = 1)$$
$$NonAllocatedBandwidth = 9000 - 6400 - 800 = 1800 \quad (BurstCount = 2)$$
$$NonAllocatedBandwidth = 16000 - 12800 - 1600 = 1600 \quad (BurstCount = 4)$$

7.4 Experimental Results

This section presents two types of results. The first results demonstrate how iterating over different values of $BurstCount$ is useful. The second section presents simulations of the Predator memory controller with requestors, detailing how the actual offered bandwidth compares to the guaranteed values computed in Section 6.

7.4.1 Rates of Satisfaction

The integrated pattern generator iterates over different values of $BurstCount$ to get the highest rates of latency and bandwidth requirement satisfaction for different use cases. Figures 7.5, 7.6, 7.7, and 7.8 demonstrate the rates of success for meeting a use case's bandwidth or latency requirements for fixed $BurstCount$ compared to an iterating $BurstCount$.

In each of the following figures, 5000 use cases are used. Each use case has 6 requestors, and each requestor has randomized requirements based on the following variables. *load* refers to the percentage of peak bandwidth requested by the requestors. $ReqSize$ is a parameter used to determine the amount of data requested by a requestor. $ReqMod$ is a modifier used in conjunction with $ReqSize$. The request size of a requestor is determined by randomly selecting a value from 1 to $ReqMod$, and multiplying it by $ReqSize$. Latency requirements are generated in a similar fashion, with $LatReq$ being a maximum latency requirement, and $LatMod$ being its modifier. These figures are useful as they prove that an iterating $BurstCount$ has a higher rate of diverse use case requirement satisfaction. The values used in the figures are defined in Table 7.2.

Table 7.2: Values of load, request size, and latency requirement.

load	82.6%
$ReqSize$:$ReqMod$	512:8
$LatReq$:$LatMod$	135:100

As seen in Figure 7.5, at a low $BurstCount$ it is difficult to provide enough bandwidth to match the bandwidth requirements of the requestors. Due to the short pattern lengths created with a low $BurstCount$, the priority assigner has more space to work with and thus has an easier task of finding a valid assignment to meet latency requirements. As $BurstCount$ increases, we see a corresponding increase in bandwidth satisfaction. This is expected because in the environment that this graph comes from, the request sizes of the requestors are larger than the granularity of the patterns produced for $BurstCount$ 1, 2, and 4. This means that the e_{data} will always be 1, and thus increasing the $BurstCount$ does not have a negative effect on bandwidth satisfaction at these values.

Figure 7.6 displays the combined latency and bandwidth satisfaction rate. The combined rate of satisfaction is highest with the iterating approach as we predicted in Section 7, thus verifying that an iterative method provides higher requirement satisfaction over different use cases.

In Figures 7.7 and 7.8, we have decreased $ReqSize$:$ReqMod$ to 64:8 in order to demonstrate a decrease in bandwidth requirement satisfaction. As seen in Figure 7.7, the bandwidth satisfaction rate drops sharply at $BurstCount$ 4. This occurs because the granularity of the patterns is now much larger than the request size of the requestors, thus negatively influencing e_{data} as $BurstCount$ increases, and as a result, the bandwidth offered is reduced.

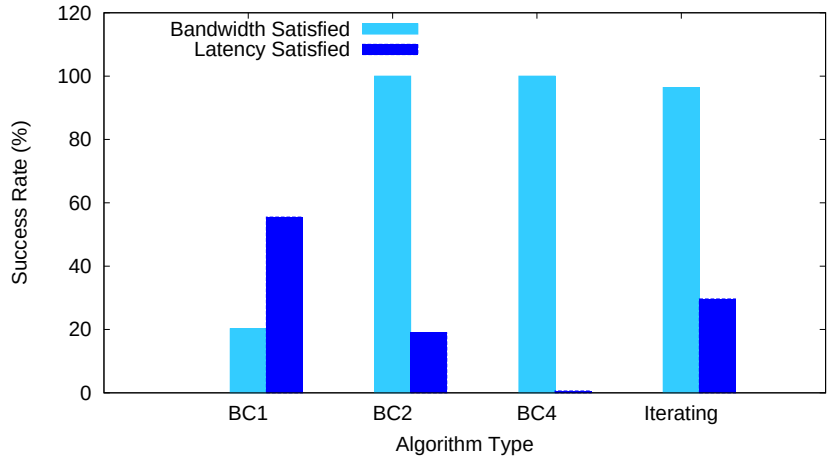

Figure 7.5: Comparison of fixed BurstCount generator and an iterating generator with large request size. The memory specification used is DDR2-400.

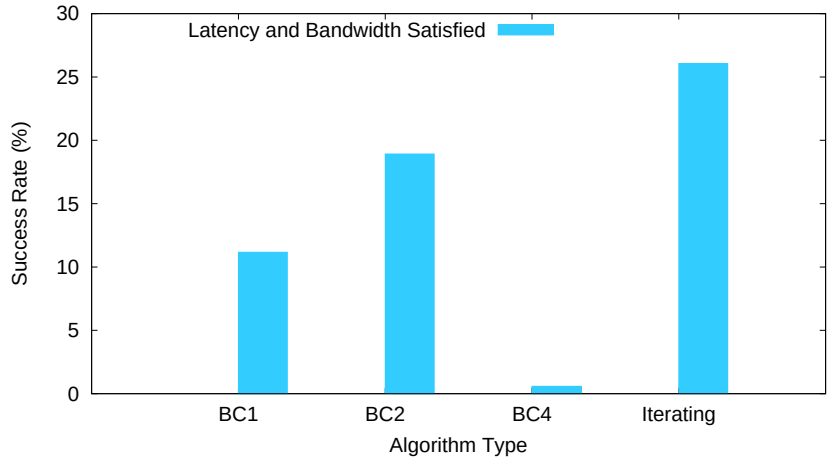

Figure 7.6: Comparison of fixed BurstCount generator and an iterating generator with large request size. The memory specification used is DDR2-400.

7.4.2 Actual Bandwidth

The results in this section are based on simulations of the Predator memory controller with 4 requestors, with a constant backlog of requests. Plotted in Figures 7.9, 7.10, 7.11, and 7.12 are actual measured bandwidth compared against the worst-case bounds guaranteed by the bank scheduling algorithm. In each figure, the simulation is run twice. In the first simulation, the patterns being executed are forced to switch from read writes at every opportunity. In the second simulation, switches are not forced, but allowed to switch at every opportunity. The length of time for all plots is 16.6 μs. This value was chosen because it is just over the amount of time required to execute 2 refresh commands. We see at 7.8 μs and at 15.6 μs a small dip in the average

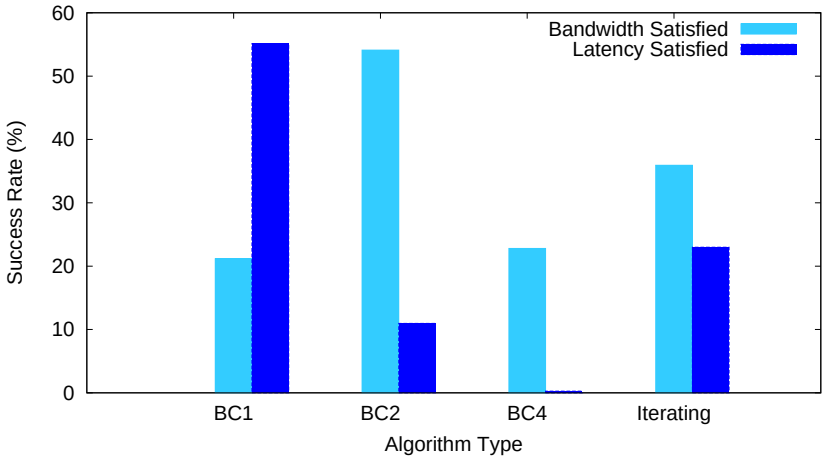

Figure 7.7: Comparison of fixed BurstCount generator and an iterating generator with small request size. The memory specification used is DDR2-400.

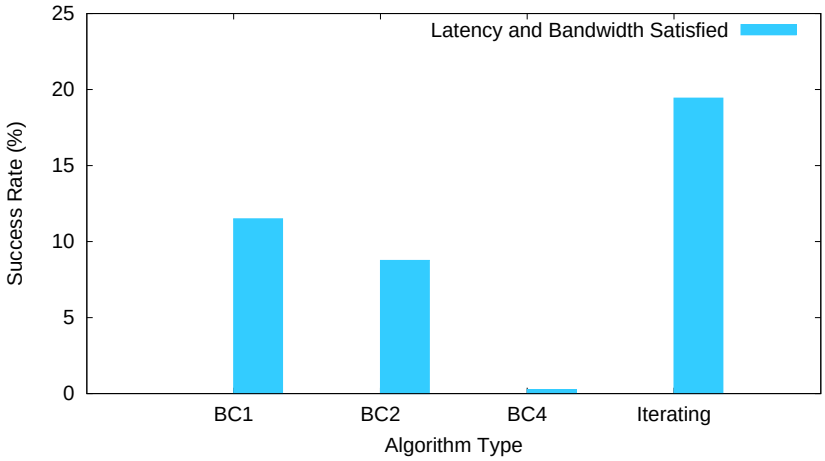

Figure 7.8: Comparison of fixed BurstCount generator and an iterating generator with small request size. The memory specification used is DDR2-400.

bandwidth due to the refresh pattern executing.

As we see in Figure 7.9, the plot of bandwidth when switching is not forced is slightly higher because the pattern set produced for this memory device is mix dominant. This means that the worst-case scenario in terms of bandwidth offered occurs when there is constant switching between read and write patterns.

In the remaining figures the forced switch and unforced switch plots are the same because the patterns generated for each figure write dominant or mix dominant with switching patterns of length 0. This implies that the worst-case occurs for forced switching use cases and non-forced switching use cases.

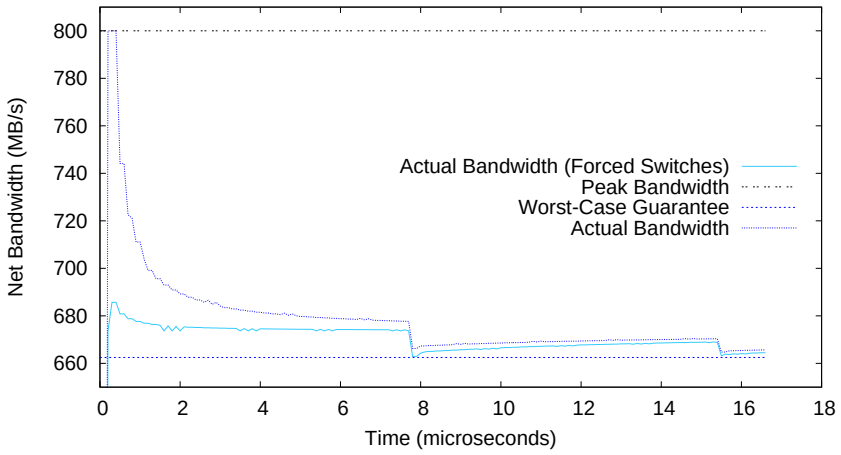

Figure 7.9: Average bandwidth over time for a DDR2-400 device by simulation.

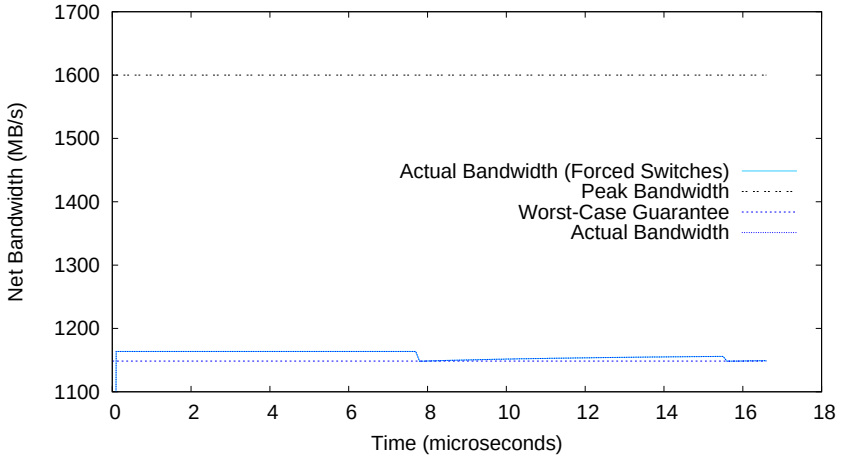

Figure 7.10: Average bandwidth over time for a DDR2-800 device by simulation.

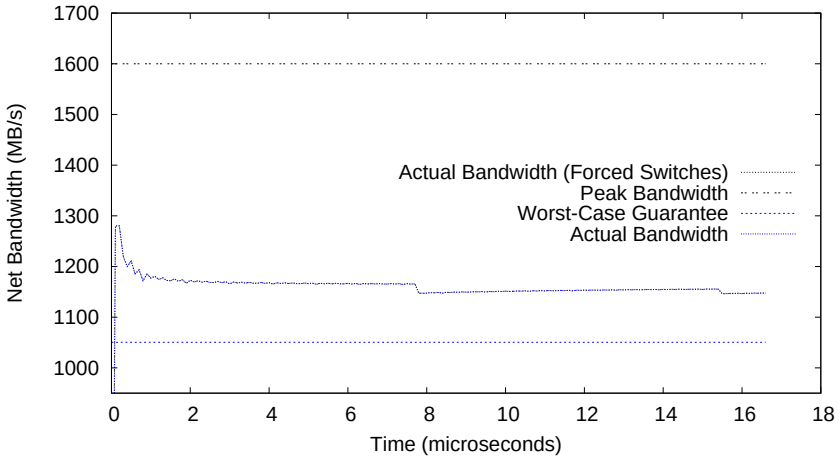

Figure 7.11: Average bandwidth over time for a DDR3-800 device by simulation.

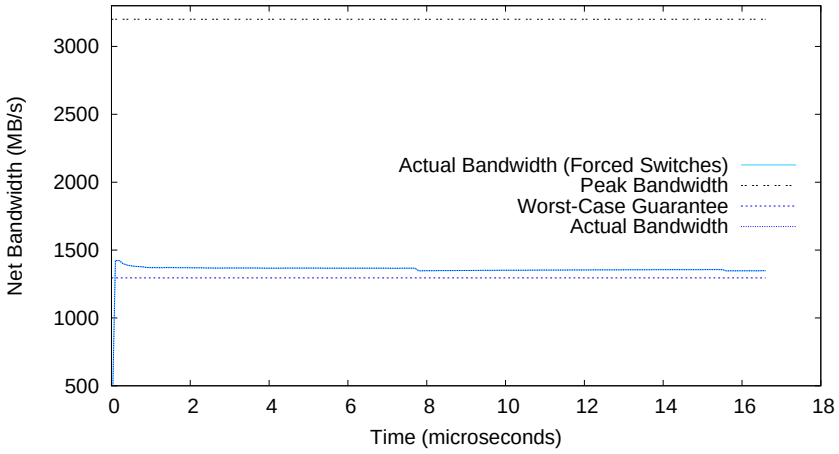

Figure 7.12: Average bandwidth over time for a DDR3-1600 device by simulation.

8 Conclusion

This thesis describes a problem associated with applications being run on real-time systems with hard requirements, namely, the bounding of worst-case latency and bandwidth. It introduces the idea of a hybrid memory controller, Predator, which uses memory patterns to interact with an SDRAM device, in order to bound worst-case latency and bandwidth. The problem of creating these memory patterns is described, along with the goal of this thesis, which is to create a method for producing patterns.

Following this, the thesis describes the architecture and operation of various basic components of the considered platform. These are the SDRAM memory device, and memory controllers. Current solutions are presented, along with why these solutions do not conform to our requests. The Predator memory controller is detailed with more information, and explanations are given on why this controller is suitable.

Basic pattern construction and metrics for evaluating pattern sets are illustrated. The metrics detailed involve memory efficiency, which is an extension of an existing model, as well as latency calculations.

Three heuristic-based pattern generation approaches are created. They explore the trade-offs between run time and net bandwidth offered. Based on experimental results, the bank scheduling algorithm is chosen as it provides a huge decrease in run time compared to the branch and bound algorithm, while providing much more bandwidth than the ASAP algorithm.

Once the pattern generation algorithm is determined, the generator is integrated into the design flow of the Predator memory controller. The integration consists of creating a loop between the generator, bandwidth allocator, and priority assigner, such that highest satisfaction rate of required bandwidth and latency are met. We prove this is the case by running the Predator design flow with multiple requestors, and showing that the iterating $BurstCount$ provides higher satisfaction rates than having a fixed $BurstCount$.

9 Future Work

This section contains ideas for possible future directions of the project.

9.1 3D Stacking

An emerging technology is that of 3D stacking. This technology reduces the cost-per-pin of a package, and therefore allows for some interesting areas of optimization. One theory to consider would be to have separate buses for reading and writing to a memory device. This could lead to the elimination for the need of switching patterns, and thus increase net bandwidth offered by mix dominant pattern sets.

A second area of improvement could be in consideration of the memory interface. With reduced cost-per-pin, it could be feasible to widen existing bus widths, or to even implement multiple interfaces for a device. These improvements would provide a higher bandwidth to requestors.

9.2 Optimal Pattern Generation

One area for improvement is in the pattern generator. Currently, in the branch and bound algorithm we only search for the shortest access patterns. This idea behind it is that with shorter access patterns, you increase e_{bank}. However, it has also been shown that in certain conditions a one or two cycle increase in access pattern length results in a 4 or 5 cycle decrease in the switching patterns. In a case observed experimentally, the pattern set was write dominant, so in fact due to the definition of e_{switch} the switching patterns were irrelevant, thus the shorter access pattern still provided higher bandwidth. But the case remains where the resulting pattern set is mix dominant, in which case the switching patterns make a difference. Therefore a proposal for making the generator produce optimal patterns could be an interesting future direction.

9.3 Low Power Considerations

Currently the generator produces patterns without any consideration for power use. As such, interleaving memory map-based patterns are the ideal choice because we are able to exploit the parallelism of the SDRAM's bank architecture, and thus provide a higher net bandwidth.

However, this approach also consumes a lot of power as all banks are activated on every access pattern execution. Activate commands take significantly more power than the other SDRAM commands. Therefore, it could be interesting to research how pattern construction choices could be modified to be aware of power consumption [11].

9.4 Future SDRAM Iterations

Another area where there may require future improvement is with next generation SDRAM devices. The transition from DDR1 to DDR2 memories saw minor changes in device operation, and the same thing happened again in the transition from DDR2 to DDR3. The algorithms detailed above were explicitly designed with those memories in mind. It is expected that when DDR4 arrives on the market that in order to fully support it some minor revisions must be made to account for new differences in timing constraints and parameters.

Bibliography

[1] S. Adee. Thanks for the Memories. *IEEE Spectrum*, 46(5), 2009.

[2] B. Akesson *et al.* Predator: a predictable SDRAM memory controller. In *Proc. CODES+ISSS*, 2007.

[3] B. Akesson *et al.* Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Proc. RTCSA*, Aug. 2008.

[4] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Real-Time Systems*, 1991.

[5] S. Dutta *et al.* Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Des. Test. Comput.*, 2001.

[6] K. Goossens *et al.* Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends. In *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15. Kluwer, 2004.

[7] E. İpek *et al.* Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[8] JEDEC Solid State Technology Association, JEDEC Solid State Technology Association 2004, 2500 Wilson Boulevard, Arlington, VA 22201-3834. *DDR3 SDRAM Specification*, jesd79-3 proposal edition, Oct 2005.

[9] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, JESD79-2C edition, May 2006.

[10] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1), 2003.

[11] S. Liu *et al.* A power and temperature aware dram architecture. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, 2008.

[12] S. A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing Frontiers*, 2004.

[13] G. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, 1975.

[14] S. Rixner *et al.* Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[15] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA 13: Symposium proceedings on High Performance Computer Architecture*, 2007.

[16] L. Steffens *et al.* Real-time analysis for memory access in media processing socs: A practical approach. *Proc. ECRTS*, 2008.

[17] E. Strooisma. A predictable and composable front-end for system on chip memory controllers. Master's thesis, Delft University of Technology, May 2008.

[18] L. Woltjer. Optimal DDR controller. Master's thesis, University of Twente, Jan. 2005.

List of relevant DDR timing constraints and parameters

A

Constraint	Description	DDR2-400 Values (cycles)
tRC	Minimum time between successive activate commands to the same bank	11
tRCD	Minimum time between activate and read/write commands on the same bank	3
CL	Latency in cycles after a read command until data is available on the bus	3
tWL	Latency in cycles after a write command until data is available on the bus	2
tAL	Additive latency, used to artificially inflate latency to provide better command bus utilization	0
tRP	Minimum time between a precharge command on a bank and a successive activate command	3
tRFC	Minimum time between a refresh command and a successive refresh or activate command	15
tRAS	Minimum time after an activate command to a bank until that bank is allowed to be precharged	8
tRTP	Minimum time between a read and precharge command	2
tWR	Minimum time after the last data has been written to a bank until a precharge may be issued	3
tFAW	Within this time frame at most 4 banks may be activated	8
tRRD	Minimum time between activates to different banks	2
tCCD	Used to compute the amount of time needed for a switching pattern	2
tWTR	Used to compute the amount of time needed for a switching pattern	2
BurstSize	Number of words produced/consumed per read/write command	Configurable
DataRate	Number of words on the data bus per clock cycle	2
BurstCount	Number of read/write commands per bank per pattern	Configurable

Glossary

B

Term	Definition	Page
BurstCount	Number of read or write commands per bank in a memory pattern	26
BurstSize	Number of words per read or write command	10
DDR	Double Data Rate	7
SDRAM	Synchronous Dynamic Random Access Memory	7
SoC	System-On-Chip	1
t_{read}	Length of read pattern	26
t_{ref}	Length of refresh pattern	26
t_{rtw}	Length of read to write switching pattern	26
t_{write}	Length of write pattern	26
t_{wtr}	Length of write to read switching patern	26

Latency Equation Derivations

C

For the case of read dominance,

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \alpha * t_{read}$$

$$\alpha \leq \frac{\hat{\Theta} - t_{ref}}{\left(1 + \frac{t_{ref}}{\phi}\right) * t_{read}} \tag{C.1}$$

For the case of write dominance,

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \alpha * t_{write}$$

$$\alpha \leq \frac{\hat{\Theta} - t_{ref}}{\left(1 + \frac{t_{ref}}{\phi}\right) * t_{write}} \tag{C.2}$$

For the case of mix read dominance,

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \left\lceil \frac{\alpha + 1}{2} \right\rceil * t_{wtr} + \left\lceil \frac{\alpha}{2} \right\rceil * t_{read} + \left\lceil \frac{\alpha}{2} \right\rceil * t_{rtw} + \left\lfloor \frac{\alpha}{2} \right\rfloor * t_{write}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \left(\frac{\alpha + 3}{2}\right) * t_{wtr} + \left(\frac{\alpha + 2}{2}\right) * t_{read} + \left(\frac{\alpha + 2}{2}\right) * t_{rtw} + \left(\frac{\alpha}{2}\right) * t_{write}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \frac{\alpha * t_{wtr}}{2} + \frac{3 * t_{wtr}}{2} + \frac{\alpha * t_{read}}{2} + t_{read} + \frac{\alpha * t_{rtw}}{2} + t_{rtw} + \frac{\alpha * t_{write}}{2}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{wtr}}{2} - t_{read} - t_{rtw} \geq \alpha * \left(\frac{t_{wtr}}{2} + \frac{t_{read}}{2} + \frac{t_{rtw}}{2} + \frac{t_{write}}{2}\right)$$

$$\alpha \leq \frac{\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{wtr}}{2} - t_{read} - t_{rtw}}{\frac{t_{wtr}}{2} + \frac{t_{read}}{2} + \frac{t_{rtw}}{2} + \frac{t_{write}}{2}} \tag{C.3}$$

For the case of mix write dominance,

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \left\lceil \frac{\alpha + 1}{2} \right\rceil * t_{rtw} + \left\lceil \frac{\alpha}{2} \right\rceil * t_{write} + \left\lceil \frac{\alpha}{2} \right\rceil * t_{wtr} + \left\lfloor \frac{\alpha}{2} \right\rfloor * t_{read}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \left(\frac{\alpha + 3}{2} \right) * t_{rtw} + \left(\frac{\alpha + 2}{2} \right) * t_{write} + \left(\frac{\alpha + 2}{2} \right) * t_{wtr} + \left(\frac{\alpha}{2} \right) * t_{read}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} \geq \frac{\alpha * t_{rtw}}{2} + \frac{3 * t_{rtw}}{2} + \frac{\alpha * t_{write}}{2} + t_{write} + \frac{\alpha * t_{wtr}}{2} + t_{wtr} + \frac{\alpha * t_{read}}{2}$$

$$\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{rtw}}{2} - t_{write} - t_{wtr} \geq \alpha * \left(\frac{t_{rtw}}{2} + \frac{t_{write}}{2} + \frac{t_{wtr}}{2} + \frac{t_{read}}{2} \right)$$

$$\alpha \leq \frac{\frac{\hat{\Theta} - t_{ref}}{1 + \frac{t_{ref}}{\phi}} - \frac{3 * t_{rtw}}{2} - t_{write} - t_{wtr}}{\frac{t_{rtw}}{2} + \frac{t_{write}}{2} + \frac{t_{wtr}}{2} + \frac{t_{read}}{2}} \tag{C.4}$$

Requestor Specification

D

The specification of four requestors are defined below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE communication SYSTEM "../../../etc/dtd/communicationgrm.dtd">
<communication>
  <application id="Application">
    <connection qos="GT" id="0">
      <initiator ip="input" port="p1"/>
      <target ip="memory_controller" port="p1"/>
      <read latency="0" bw="164" burstsize="128"/>
      <parameter id="delay" type="bool" value="1"/>
    </connection>
    <connection qos="GT" id="1">
      <initiator ip="filter1" port="p1"/>
      <target ip="memory_controller" port="p1"/>
      <write latency="0" bw="164" burstsize="128"/>
      <parameter id="delay" type="bool" value="1"/>
     </connection>
    <connection qos="GT" id="2">
      <initiator ip="filter1" port="p2"/>
      <target ip="memory_controller" port="p1"/>
      <read latency="0" bw="164" burstsize="128"/>
      <parameter id="delay" type="bool" value="1"/>
    </connection>
    <connection qos="GT" id="3">
      <initiator ip="output" port="p1"/>
      <target ip="memory_controller" port="p1"/>
      <write latency="0" bw="164" burstsize="128"/>
      <parameter id="delay" type="bool" value="1"/>
    </connection>
  </application>
</communication>
```

DDR Memory Specification

E

```
<!DOCTYPE architecture SYSTEM "../../../etc/dtd/architectureregrm.dtd">
<architecture  id="isss">

   ...........
   ...........

   <ip id="memory_controller" type="MemoryController">
      <port id="p1" type="Target" protocol="MMIO_DTL">

         <!-- Memory Specification -->
         <parameter id="memoryId" type="string" value="64MB_DDR3-1600_16bit" />
         <parameter id="capacity" type="uint" value="65536" />
         <parameter id="nbrOfBanks" type="uint" value="4" />
         <parameter id="clk" type="uint" value="800" />
         <parameter id="dataRate" type="uint" value="2" />
         <parameter id="tREFI" type="double" value="7800" />
         <parameter id="burstSize" type="uint" value="8" />
         <parameter id="wordSize" type="uint" value="2" />
         <parameter id="RC" type="uint" value="36" />
         <parameter id="RCD" type="uint" value="8" />
         <parameter id="CL" type="uint" value="8" />
         <parameter id="WL" type="uint" value="7" />
         <parameter id="AL" type="uint" value="0" />
         <parameter id="RP" type="uint" value="8" />
         <parameter id="RFC" type="uint" value="72" />
         <parameter id="RAS" type="uint" value="28" />
         <parameter id="RTP" type="uint" value="6" />
         <parameter id="WR" type="uint" value="12" />
         <parameter id="FAW4" type="uint" value="24" />
         <parameter id="RRD" type="uint" value="5" />
         <parameter id="CCD" type="uint" value="4" />
         <parameter id="WTR" type="uint" value="6" />

         ..........
         ..........
      </port>
   </ip>
</architecture>
```