UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

FELIPE SALERNO PRADO

Scheduling Mechanisms for DRAM Memory Controllers

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Sergio Bampi
Co-advisor: Dr.-Ing. Matthias Jung

Porto Alegre
July 2017

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Renato Ventura Bayan Henriques
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Wahrlich es ist nicht das Wissen, sondern das Lernen, nicht das Besitzen, sondern das Erwerben, nicht das Da-Seyn, sondern das Hinkommen, was den grössten Genuss gewährt." - Carl Friedrich Gauss

ACKNOWLEDGMENTS

I would like to express my gratitude to my family, especially to my parents, Jacqueline Leal Salerno and Jorge Antonio Tadeu Posada Prado, without whom I could have never graduated.

I would also like to thank Prof. Dr.-Ing. Norbert Wehn and all the staff of the University Kaiserslautern's Microelectronic Systems Design Research Group for the exceptional professional experience in Germany.

Special thanks to Dr.-Ing. Matthias Jung, who shared his experience and knowledge on DRAMs and virtual prototyping, for guiding me in this work since its beginning in Kaiserslautern.

Last but not least, I thank Prof. Dr. Sergio Bampi for being my advisor in this work and Prof. Dr. Taisy Silva Weber for coordinating the Brazilian side of the partnership between UFRGS and University of Kaiserslautern.

Mecanismos de Escalonamento para Controladores de Memória DRAM

RESUMO

Ao longo das últimas décadas, a disparidade de desempenho entre o processador e memória tem crescido cada vez mais, uma questão comumente conhecida como *Memory Wall* (WULF; MCKEE, 1995). Tendo em vista que Memória Dinâminca de Acesso Randômico (DRAM) é a forma mais comum de memória em sistemas computacionais, intensas pesquisas têm sido feitas para melhorar seu desempenho e eficiência energética. Entretanto, a exploração do espaço de projeto de DRAM vem com problemas. Apesar de simuladores de memória *cycle-accurate* poderem simular *timings* de maneira muito precisa, a simulacão pode se tornar extremamente lenta. Movendo-se para um mais alto nível de abstração, através do método de prototipação virtual, a simulação pode ser acelerada com erros desprezíveis. Como o controlador de memoria DRAM é um bloco crítico para o desempenho global do sistema de memória, ele é integrado no mesmo *chip* dos múltiplos núcleos de CPU os quais demandam accesso a memória principal *off-chip*. Portanto, neste trabalho, escalonadores do controlador de DRAM foram modelados com SystemC/TLM 2.0 (IEEE Computer Society, 2012) no DRAMSys (JUNG M.; WEIS; WEHN, 2015), um *framework* flexível de exploração do espaço de projeto do subsistema da DRAM. A reestruturação do DRAMSys utilizando SystemC/TLM 2.0 foi a principal contribuição técnica deste trabalho. Em seguida, o controlador foi simulado com diferentes mecanismos de escalonamento com o objetivo de melhor compreender os gargalos do sistema de memória e de investigar qual escalonador fornece o melhor compromisso entre características como largura de banda, *thread-fairness* e consumo de energia.

Palavras-chave: DRAM, Prototipação Virtual, Controlador de Memória, Escalonamento.

ABSTRACT

Over the last decades, the performance disparity between processor and memory has steadily grown in computer systems, an issue commonly known as Memory Wall (WULF; MCKEE, 1995). Since Dynamic Random-Access Memory (DRAM) is the most common type of main memory in computer systems, intense research efforts have been made to improve its performance and energy efficiency. However, DRAM design space exploration comes with problems. Although cycle-accurate memory simulators can simulate timings in a very precise manner, the simulation can become extremely slow. By moving to a higher abstraction level, through the virtual prototyping method, simulation speedup can be achieved with negligible losses in accuracy. Seeing that the DRAM memory controller is a key block for the overall memory system performance, it is integrated onto the same chip of the multiple CPU cores that require access to the off-chip main memory. Therefore, in this work, schedulers of the DRAM controller were modeled with SystemC/TLM 2.0 (IEEE Computer Society, 2012) in DRAMSys (JUNG M.; WEIS; WEHN, 2015), a flexible DRAM subsystem design space exploration framework. The DRAMSys restructuring was the main technical contribution of this work. Then, the controller was simulated with different scheduling mechanisms in order to better understand the memory system bottlenecks and to investigate which scheduler provides the best compromise between characteristics, such as bandwidth, thread-fairness and energy consumption.

Keywords: DRAM. Virtual Prototyping. Memory Controller. Scheduling.

LIST OF ABBREVIATIONS AND ACRONYMS

1T1C	One-Transistor One-Capacitor
ARM	Advanced RISC Machine
BRC	Bank-Row-Column
CPU	Central Processing Unit
DCS	DRAM Command Scheduler
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random-Access Memory
FCFS	First Come First Served
FIFO	First In First Out
FP	Floating-Point
FR-FCFS	First Ready - First Come First Served
GPU	Graphics Processing Unit
HMC	Hybrid Memory Cube
IEEE	Institute of Electrical and Electronics Engineers
INT	Integer
IPC	Instructions Per Clock
IQ	Instruction Queue
JEDEC	Joint Electron Device Engineering Council
LPDDR	Low Power Double Data Rate Memory
LSQ	Load-Store Queue
MPKC	Miss Per Thousand Cycles
PAR-BS	Parallelism-Aware Batch Scheduling
PDNA	Power-Down Activate

PDNP	Power-Down Precharge
RBC	Row-Bank-Column
ROB	Re-order Buffer
RTL	Register-transfer Level
SJF	Shortest-Job-First
SMS	Staged Memory Scheduling
SRAM	Static Random-Access Memory
SREF	Self-Refresh
SoC	System on Chip
TLM	Transaction Level Modeling
VHDL	VHSIC Hardware Description Language
XOR	Exclusive Disjunction

LIST OF FIGURES

Figure 2.1 Standard vs. Prototyped Project Flow (SMART, 2013) 13
Figure 2.2 SysctemC Core Facilities 15
Figure 2.3 Basic TLM 2.0 Features 16
Figure 2.4 TLM 2.0 Base Protocol (DOULOS, 2013) 17

Figure 3.1 1T1C DRAM Cell 18
Figure 3.2 Basic Structure of a DRAM 19
Figure 3.3 Basic DRAM Bank Operations 20
Figure 3.4 DRAM Memory System 22

Figure 4.1 Base Architecture of DRAMSys (JUNG M.; WEIS; WEHN, 2015) 23
Figure 4.2 Bank-Row-Column and Row-Bank-Column Address Mappings (JUNG, 2017) 25
Figure 4.3 DRAMSys Trace Analyzer (JUNG M.; WEIS; WEHN, 2015) 26

Figure 5.1 PAR-BS Request Batching 31
Figure 5.2 Organization of the SMS (AUSAVARUNGNIRUN et al., 2012) 33
Figure 5.3 New Architecture of DRAMSys 34
Figure 5.4 New Trace Analyzer of DRAMSys 35
Figure 5.5 Trace Capture from a detailed execution-driven CPU and Replay for Memory-system exploration. (JAGTAP et al., 2016) 36
Figure 5.6 Stream Trace 39
Figure 5.7 Random Trace 39
Figure 5.8 Results of Experiment 1 40
Figure 5.9 Results of Experiment 2 42
Figure 5.10 Results of Experiment 3 43
Figure 5.11 PAR-BS Simulation 44

LIST OF TABLES

Table 5.1 CPU parameters....................37
Table 5.2 DRAM parameters, VDD = 1.5V and f = 800 MHz (Micron Technology Inc., 2006)....................38
Table 5.3 Setup of the Experiments....................40

CONTENTS

1 INTRODUCTION...12
2 VIRTUAL PROTOTYPING - SYSTEMC AND TLM 2.0.......................................13
2.1 SystemC 14
2.2 TLM 2.0 15
2.3 Coding Styles 16
3 DRAM BASICS...18
3.1 Structure 19
3.2 Operations 20
3.3 Memory Controller 21
4 DRAMSYS...23
4.1 Initiator 24
4.2 Arbiter...24
4.3 Address Mapping 24
4.4 Channel Controller - Controller Core and Scheduler 25
4.5 TLM Recorder and Trace Analyzer...26
4.6 DRAMPower 27
5 COMPARISON OF SCHEDULING MECHANISMS...28
5.1 FR-FCFS..28
5.2 PAR-BS 29
5.3 SMS 31
5.4 DRAMSys Restructuring 33
5.5 Elastic Traces...35
5.6 Experiments...37
5.7 Results 40
5.8 Concluding Remarks 43
6 CONCLUSIONS AND FUTURE WORK 46
REFERENCES..47

1 INTRODUCTION

Nowadays, in times of ever-increasing number of processors in computer systems, the main memory plays an important role, since the CPU directly interacts with it. The advancement of the silicon technology has provided large improvements in circuits density which are visible both in processors and memories. However, over the last decades, the performance disparity between them has steadily grown, an issue commonly known as Memory Wall (WULF; MCKEE, 1995). Hence, it is increasingly crucial to develop new ideas and concepts in memory design.

The Dynamic Random-access Memory (DRAM) has been widely used in computer systems for many years, due to its low-cost, simple structure and scalability. As the memory system becomes one of the bottlenecks of the overall system, several mechanisms were proposed with the purpose of enhancing its effectiveness. However, DRAM design space exploration comes with problems. Implementing hardware prototypes is expensive and time-consuming and, in spite of the timing precision, cycle-accurate simulations can become extremely slow. Thus, the virtual prototyping method emerged as a novel way of conducting research on computer systems. By modeling systems at a higher abstraction level, this method speeds up the simulation with negligible losses, which allows a fast exploration of various design choices for the lower level implementation.

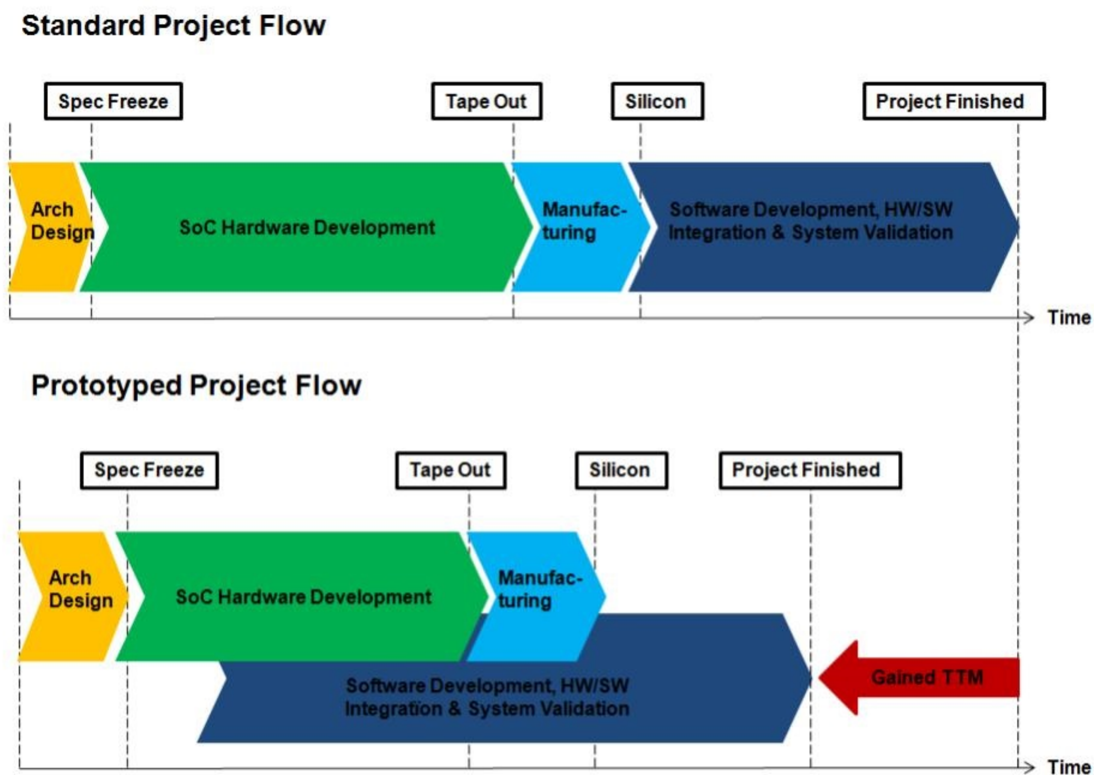
The DRAM controller, which schedules the memory requests and manages the flow of data going to and from memory, is a key block to be explored with the purpose of developing new concepts for DRAMs. It may aim at one of many goals, for example, maximizing throughput, minimizing latency or maximizing fairness. As the number of processor increases, the number of accesses to the memory also increases. Thus, the challenge is to design an effective controller, whose scheduling technique handles various memory accesses from different sources fairly and optimally.

In this work, schedulers of the memory controller were modeled with SystemC/TLM 2.0 (IEEE Computer Society, 2012) in DRAMSys (JUNG M.; WEIS; WEHN, 2015), a flexible DRAM subsystem design space exploration framework. The DRAM controller was then simulated with different scheduling techniques to better understand the memory system bottlenecks and to investigate which scheduler is the most efficient.

2 VIRTUAL PROTOTYPING - SYSTEMC AND TLM 2.0

Hardware manufacturers today are under pressure to reduce time-to-market (TTM) and optimize products. In the standard project flow of developing computer systems, the software development just starts when the hardware is already manufactured and ready for use. The product will be available for the market after integrating the software with the hardware. However, this method is expensive and time-consuming, because the initial physical prototype is highly unlikely to meet expectations and, therefore, engineers usually have to re-design the initial concept multiple times. Hence, virtual prototyping was established as an alternative method in the process of computer systems development: software prototypes of hardware are used to predict performance prior to constructing real prototypes. In other words, the development of the hardware and software is conducted concurrently, shortening the time-to-market and simplifying the development cycle. Figure 2.1 gives a brief preview of the difference between the standard and the prototyped project flows.

Figure 2.1: Standard vs. Prototyped Project Flow (SMART, 2013)



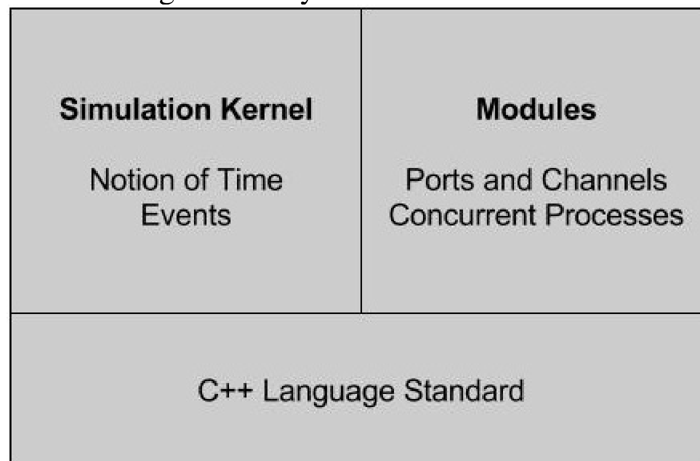
2.1 SystemC

Hardware description languages like Verilog and VHDL allow a register-transfer level (RTL) modeling and cycle-accurate simulation of computer systems and is used to design integrated circuits. However, the modeling and cycle-accurate simulation of a whole system can become very slow, and the abstraction level provided by RTL modeling might be over-detailed, depending on the primary goals of the project. By using SystemC, a system design language that has evolved in response to a pervasive need for a language that improves overall productivity for designers of electronic systems (BLACK; DONOVAN., 2004), simulation speedup and system modeling at a higher level of abstraction can be achieved.

SystemC is a language built in standard C++ by extending the language with a set of class libraries created for design and verification (ACCELLERA, 2013). It provides the facilities to simulate processes and events in C++ through hardware-oriented features, such as notion of time, concurrency, and hardware data types. By providing such attributes, SystemC is able to help engineers design both hardware and software together and, thus, is an ideal tool for the virtual prototyping efforts described previously. SystemC is described and documented in (IEEE Computer Society, 2012).

Figure 2.2 shows the SystemC core facilities, which includes a simulation kernel and modules. The simulation kernel provides the notion of simulation time to the execution of a C++ program and events that can be triggered at specific times or under certain conditions so that hardware modules sensitive to signals such as a system clock signal can be modeled. A module can have several concurrent processes, which may be active or suspended (waiting for an event). The inter-module communication is done via channels and ports and, in contrast to VHDL, modules can also interchange complex data structures in addition to bit vectors.

Figure 2.2: SysctemC Core Facilities

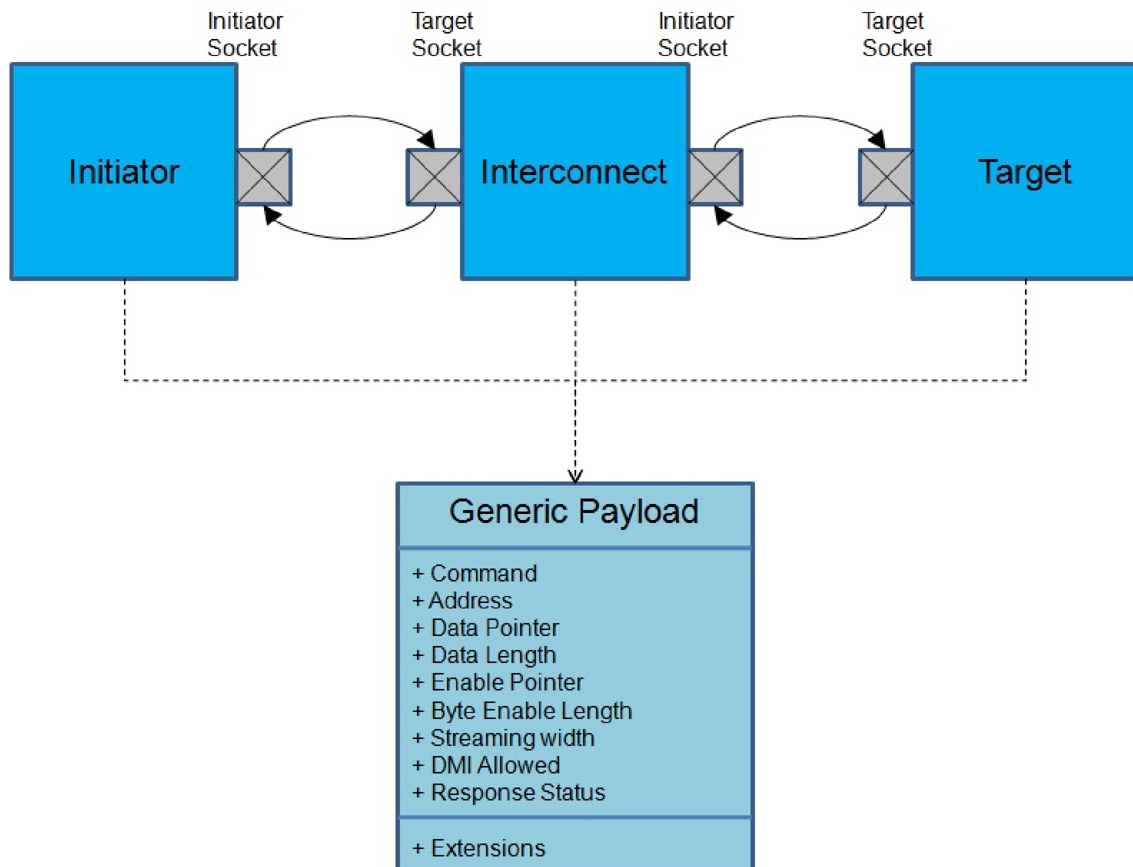


2.2 TLM 2.0

Transaction Level Modelling 2.0 (TLM 2.0) is an extension to SystemC, defining a communication base protocol and interfaces in order to achieve maximum interoperability among hardware models. In other words, TLM-2.0 compliant SystemC modules can easily be connected to and interact with each other. The TLM 2.0 is particularly well-suited for memory-mapped bus modeling (IEEE Computer Society, 2012).

The communication between modules is based on a generic communication object, the generic payload, which is passed for each transaction. The payload is always passed by reference, i. e., once created in the memory, it does not have to be copied to be modified. An initiator is a module which starts a transaction by sending a request to a target. A target is a module which responds to a transaction initiated by another module by processing the request, possibly modifying the communication object and sending back a response. A transaction is finished when an initiator receives a response. The same module can be both initiator and target and, between initiator and target, there can be interconnected modules which manipulate and dispatch requests. For interoperability, via TLM sockets, which define communication interface methods, modules communicate using function calls. A custom payload extension, an arbitrarily complex data structure, can be attached to the generic payload. Since this feature destroys interoperability, it should just be used for the intra-communication of components. Seeing that a function call may model a complete bus cycle, the simulation time is speeded up. Figure 2.3 shows the basic features of the TLM 2.0.

Figure 2.3: Basic TLM 2.0 Features



2.3 Coding Styles

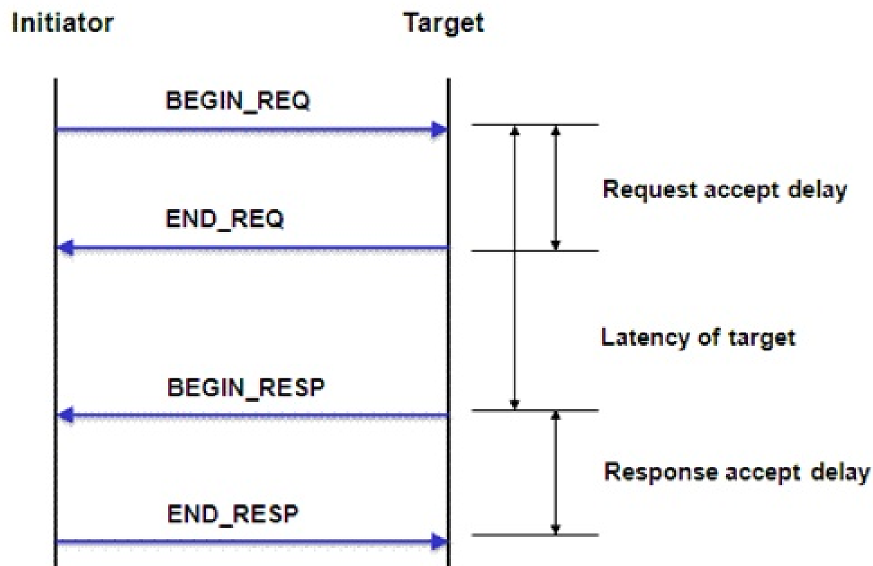
For the inter-communication of modules, TLM 2.0 considers models which can be separated into two coding styles regarding the timing detail: the loosely-timed (LT) and the approximately-timed (AT) coding styles.

The LT coding style has a loose dependency between timing and data and does not depend on the advancement of the time to be able to produce a response, decreasing the time accuracy. A transaction timing is just defined by a start and an end of the request, for instance, a memory request modeled with the elapsing time, where the initiator has to wait for the target response before sending the next request. Due to the limited dependencies and minimal context switches, this coding style runs faster simulations, and thus, it is particularly useful for doing software development on virtual platforms.

The AT coding style has a much stronger dependency between timing and data and depends on internal/external events firing and time advancing before they can provide a response. Different from the LT coding style, resource contention and arbitra-

tion can be easily modeled with this style. In order to achieve the required accuracy to model hardware components such as a memory controller properly, a transaction is split into multiple phases. The TLM 2.0 base protocol comprises four phases: BEGIN_REQ, END_REQ, BEGIN_RESP and END_RESP. The initiator is allowed to start a new transaction, i.e., to send a new BEGIN_REQ, after receiving the END_REQ of the previous transaction. Likewise, the target is allowed to send a new BEGIN_RESP after receiving the END_RESP of the prior transaction. However, the initiator can start the next transaction before receiving the BEGIN_RESP of the previous transaction. Therefore, several transactions can be in the system at the same time. Figure 2.4 shows the TLM 2.0 base protocol behavior. Special cases such as early completion are not considered in the diagram. For more details refer to (IEEE Computer Society, 2012). Since a request and its processing data can be separated, pipelining and out-of-order execution can be modeled. As long as the base protocol rules are respected, it is possible to extend the base protocol by defining custom phases. As the payload extensions, custom phases conflict with the interoperability efforts and, hence, it should just be used for the intra-communication of components.

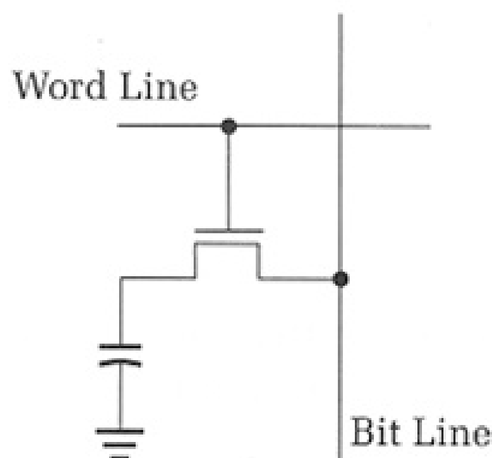
Figure 2.4: TLM 2.0 Base Protocol (DOULOS, 2013)



3 DRAM BASICS

DRAMs (Dynamic Random-Access Memory) belong to the class of volatile memories, which require power to maintain the stored information and are based on a capacitor, which stores a single bit of data. The two states of the capacitor, charged and discharged, represent the two values of a bit, 0 and 1. The DRAM cell consists of just one transistor and one capacitor. Figure 3.1 shows the One-Transistor One-Capacitor (1T1C) DRAM cell. It is a dynamic memory, i. e., the information stored in a cell eventually fades. Since the information (charge) leaks from/into the capacitor due to the imperfection of the components, in order to retain the stored data, the DRAM cells should be periodically refreshed, as opposed to the SRAM (Static Random-Access Memory) cells. The time a cell can hold information is called retention time. Because of the regenerative feedback present in SRAM memory cells and due to the refresh requirement of DRAMs, SRAMs are much faster than DRAMs. However, the structural simplicity of DRAMs allows reaching higher densities and lower cost per bit. A DRAM Memory System is composed of memory modules, the memory bus, and the memory controller. The predominant type of memory module is the dual in-line memory module (DIMM), which comprises a series of DRAM chips and it is designed for personal computers, workstations, and servers. The memory bus includes the address, data, and control buses. Hence, DRAMs are very common in current computer systems. For an in-depth explanation of memories refer to (JACOB B.; WANG, 2008).

Figure 3.1: 1T1C DRAM Cell

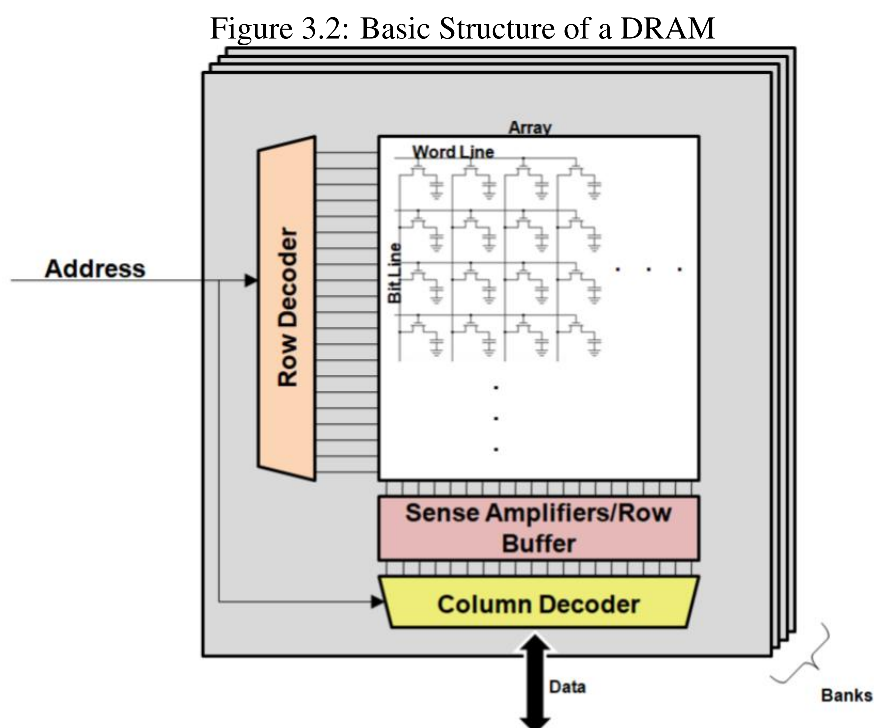


3.1 Structure

The hierarchical structure of DRAMs has basically been the same since the first generations. The set of minimum requirements for each DRAM technology, e.g., timing constraints and communication interfaces, is regulated by the Joint Electron Device Engineering Council (JEDEC) (JEDEC, 1958), a trade organization and standardization body. Since internals are specified by each DRAM manufacturer, the JEDEC standards allow interoperability between components from diverse manufacturers.

The DRAM cells are arranged in a grid-like pattern, an array, in which they are accessed via word lines and bit lines. Memory arrays are organized into rows and columns. At each column, all cells are connected to the same bit line and at each row, also called page, all cells are connected to the same word line. Thereby, at every intersection of a row and a column, a cell is connected to a word line and a bit line. A set of memory arrays comprises a bank, whose number of cell arrays is also called the data width of the DRAM. Each bank has a row buffer, which can hold exactly one page and is physically an array of sense amplifiers acting as buffers (JACOB B.; WANG, 2008). A DRAM chip consists of multiple independently operating banks and is described as xN, being N the number of output pins. Figure 3.2 shows the basic organization of a DRAM.

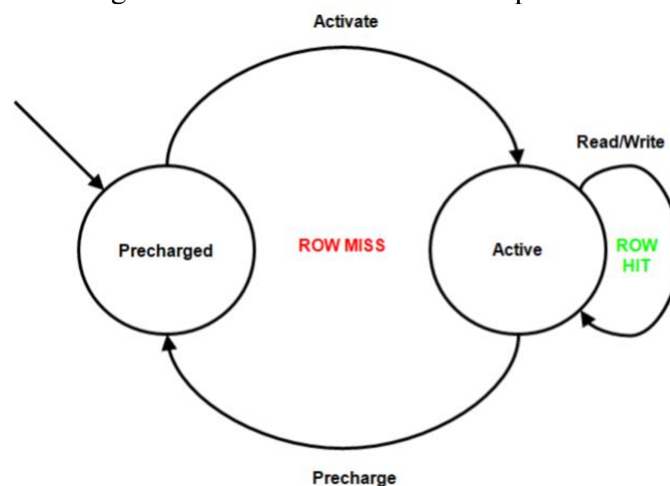
Figure 3.2: Basic Structure of a DRAM



3.2 Operations

When a request enters the DRAM system, its address is used to determine which bank, row and column are accessed. The operation to be performed depends on the current state of the row buffer. If the row buffer is loaded with a different row than the requested row, the bit lines have to be precharged to a reference voltage to begin the activity of the sense amplifiers and get the data from the target row cells. This operation is called precharge, and it sets the bank to a precharged state. If the row buffer has the content of the target row, also known as a row hit, or the bank is already in the precharged state, this operation is not necessary. A request which does not result in a row hit is called a row miss. If the bank is initially in the precharged state, through the sense amplifiers activity, the target row is accessed and loaded into the row buffer. This activity is called activate, and it sets the bank to an active state. While the bank is in the active state, read or write commands can be performed. By selecting the memory cells according to the target column, data are read or written. Hence, a row-hit request can omit the precharge and activate operations and directly perform the requested operation. A row-miss request that requires an activate command is also known as a row conflict. Figure 3.3 shows the state diagram of the operations described above. Seeing that activate and precharge operations consume time and energy, it is desirable to avoid row misses whenever possible. Thus, modern DRAMs can operate in bursts with the purpose of enhancing the bandwidth, i. e., more than one column is read/written after the execution of a command. The number of accessed columns is called burst length, and this effectively reduces latency when accessing consecutive memory locations.

Figure 3.3: Basic DRAM Bank Operations



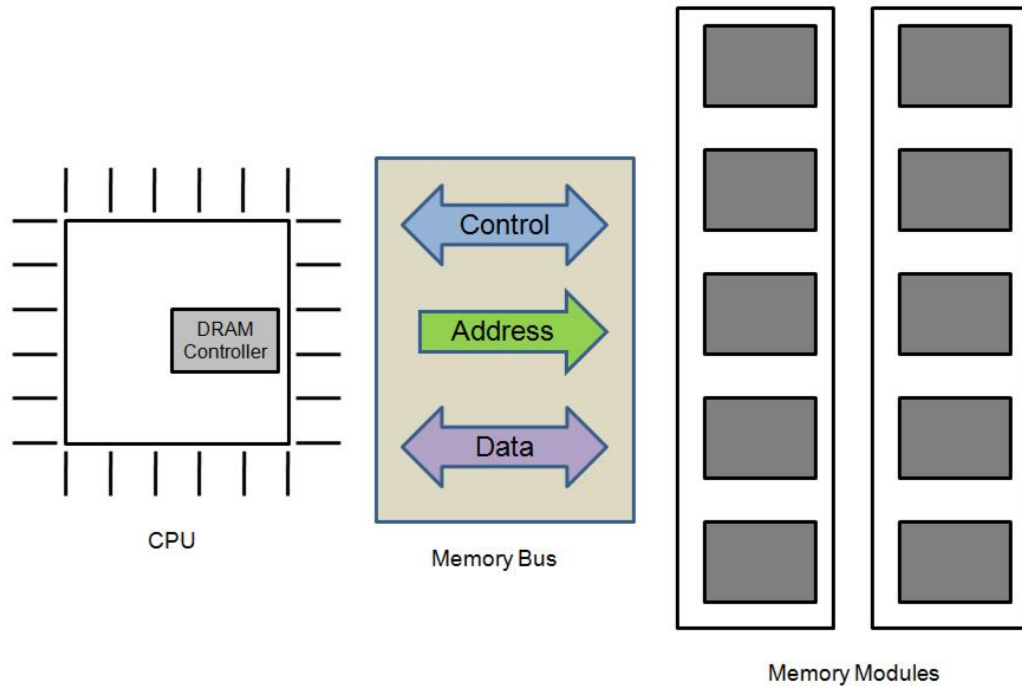
As mentioned before, a refresh operation is needed because DRAMs are dynamic memories. The refresh operation, which rewrites the data into the memory cells, is done by activating and precharging a row. Every time a refresh operation is required, the DRAM system has to firstly precharge all active banks and, once all banks are precharged, perform the refresh command. The refresh rate of a DRAM depends on the technology and also on the temperature of the memory.

Since energy efficiency has increasingly been a key goal of the electronics industry, there are also commands capable of saving energy in the DRAM system. When it is in an idle state, in other words, when there are no requests, the following commands are ready to be performed: Power-Down Activate (PDNA), Power-Down Precharge (PDNP) and Self-Refresh (SREF) (JUNG M.; WEIS; WEHN, 2015). The PDNA can be executed if one or more banks are still in active mode and it sets the DRAM to the power-down active state. The PDNP can be performed if all banks are precharged and it sets the DRAM to the power-down precharged state. In power-down active and precharged states, explicit refreshes need to be issued. The SREF can be executed if all banks are precharged and it sets the DRAM to the self-refresh mode. Since internal refreshes are executed during this state, it is the most energy saving state. However, a long wake-up time is required to return to a nonidle state. In current DRAM systems, these operations work at the chip level, so all banks on a chip enter and exit a power-down mode at the same time.

3.3 Memory Controller

The memory controller can be a separate chip or integrated into another chip, such as being placed on the same die or as integrated part of a microprocessor. Since today's memory controllers are integrated into a System On Chip (SoC) which comprises multiples cores, the DRAM controller communicates internally with the CPU and externally via the memory bus with the memory modules. Figure 3.4 illustrates this scenario.

Figure 3.4: DRAM Memory System



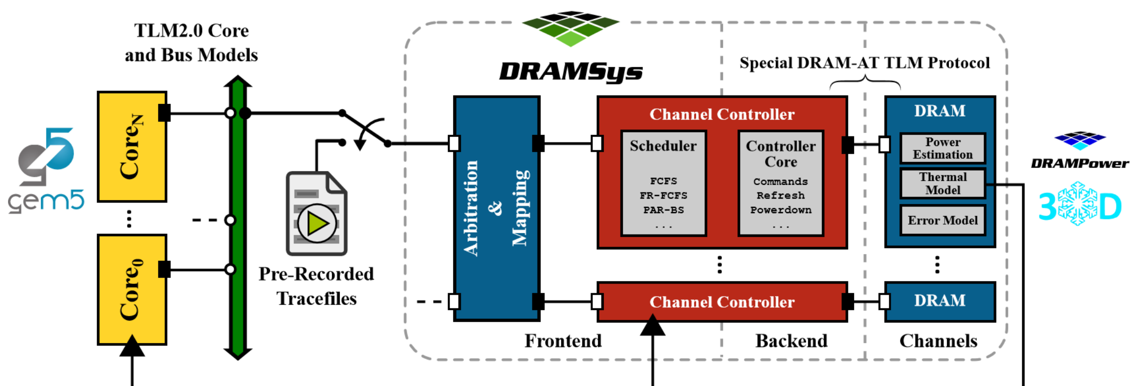
The DRAM controller is responsible for issuing requests, which includes refresh and power-down commands, to a DRAM module, in such a way that resources and timing constraints are met and the performance is maximized. It is also the module which connects the memory modules to other components. Thus, it is the brain of the memory system, and through many mechanisms, it is possible to optimize many aspects of the system. For example, by maximizing the number of row hits and exploiting the bank parallelism, the system throughput is increased, and the access latency is minimized. In order to achieve improvements like the described above, the scheduler in the memory controller, which manages the requests and decides the next one to be issued, have to be properly designed. Hence, the scheduler is a key mechanism for enhancements in the DRAM system. Seeing that there are other mechanisms besides the scheduler to be exploited, the design space exploration of DRAM systems is noticeably huge.

4 DRAMSYS

DRAMSys is a DRAM subsystem design space exploration framework (JUNG M.; WEIS; WEHN, 2015) which supports a wide range of standard and emerging DRAM subsystems, e. g., DDR3, DDR4, LPDDR3, Wide I/O and Hybrid Memory Cube (HMC). Hence, it consists of flexible and extensible models that are designed in a modular fashion and reflect the DRAM functionality, power, temperature and retention time errors. It provides several analysis tools that assist the designer in order to enable him to optimize the DRAM subsystem concerning the controller architecture, power, and thermal management as well as device selection and channel configuration for a specific application.

The framework is implemented in C++ using SystemC/TLM 2.0, which provides simulation speedup, interoperability and easy integration of other TLM components. In order to achieve enough accuracy to reflect realistic behaviors, i. e., negligible losses in accuracy compared to an RTL simulation (JUNG M.; WEIS; WEHN, 2015), all connections are implemented in the TLM 2.0 AT coding style, except the connection between controller and channel, which is implemented in DRAM-AT (JUNG et al., 2013), an extended TLM 2.0 protocol with DRAM specific phases. Figure 4.1 shows the base architecture of DRAMSys. The modules communicate with each other using TLM 2.0 sockets, the arbiter can be connected to multiple channels, which are basically independent DRAM systems, and generally, an arbitrary number of initiators and DRAM channels can be instantiated. The tool can be used in professional virtual platforms environments like Synopsys Platform Architect (Synopsys Inc., 2015) or as a standalone simulator with native SystemC/TLM 2.0.

Figure 4.1: Base Architecture of DRAMSys (JUNG M.; WEIS; WEHN, 2015)



4.1 Initiator

The Initiator can be any TLM 2.0 compliant module such as trace players or even complete CPU models like gem5 (BINKERT et al., 2011). A trace player sends memory requests to the DRAM system by processing trace files. The native trace file of DRAMSys describes each transaction with one line, in which three separated columns represent the initial clock cycle of the transaction, the requested operation (read or write) and the address to be accessed. In order to facilitate efficient analysis and explorations, the framework supports trace files from other simulation such as gem5 (BINKERT et al., 2011) and Simplescalar (BURGER; AUSTIN, 1997). To achieve faster simulations, it is also possible to replay pre-recorded transaction traces with elastic trace players (KOGEL, 2010) (JAGTAP et al., 2016). By instantiating multiple initiators, the memory usage of multicore systems can be simulated as well.

4.2 Arbiter

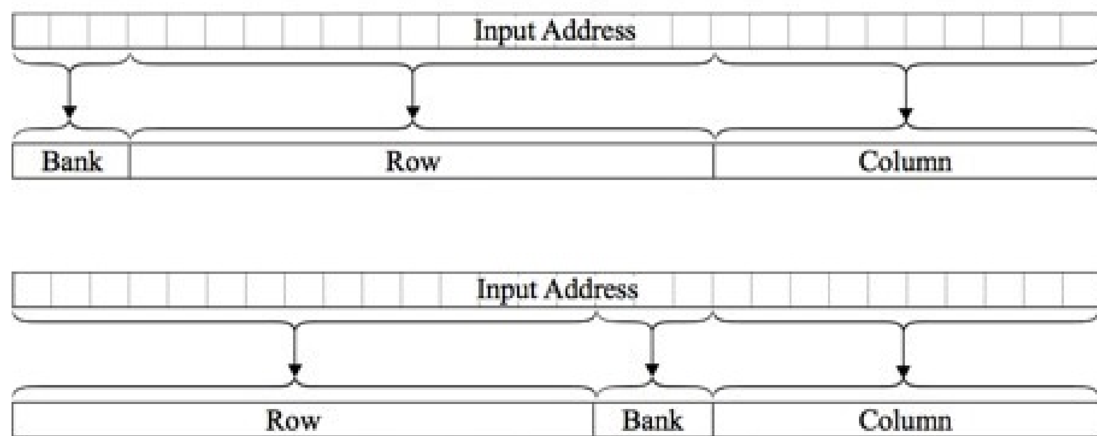
The Arbiter is a front-end module of DRAMSys, which is responsible for forwarding requests from an initiator to a memory channel and responses from a channel to an initiator. Once a transaction enters the system, it is stored in a buffer according to its initiator. A transaction remains in a buffer until the target initiator or channel is free, and the arbiter selects it as the next one to be sent. Since the speed and the number of initiators and channels might be different, the arbiter has to manage transactions in such a way that they are stalled as minimum as possible. Hence, it is essential to model the arbiter correctly to get realistic results.

4.3 Address Mapping

The front-end module of DRAMSys arbitrates the several inputs to the memory subsystems and maps transactions to the different channels by a defined address mapping. The Bank-Row-Column (BRC) and the interleaved Row-Bank-Column (RBC), as shown in figure 4.2, are the most common addressing schemes. The second one, also called Page Mode or Bank Interleaving (JACOB B.; WANG, 2008), brings the upper bank bits of the address down between the row and the column bit to ensure a better bank utilization.

Besides, there are also sophisticated mapping schemes such as the Permutation-Based Page Interleaving Technique (ZHANG et al., 2000), in which accesses to different rows in the same bank are transformed into accesses to different banks by XORing the bank bits with selected row bits. Therefore, the address mapping has a great impact on the performance of a memory system.

Figure 4.2: Bank-Row-Column and Row-Bank-Column Address Mappings (JUNG, 2017)



4.4 Channel Controller - Controller Core and Scheduler

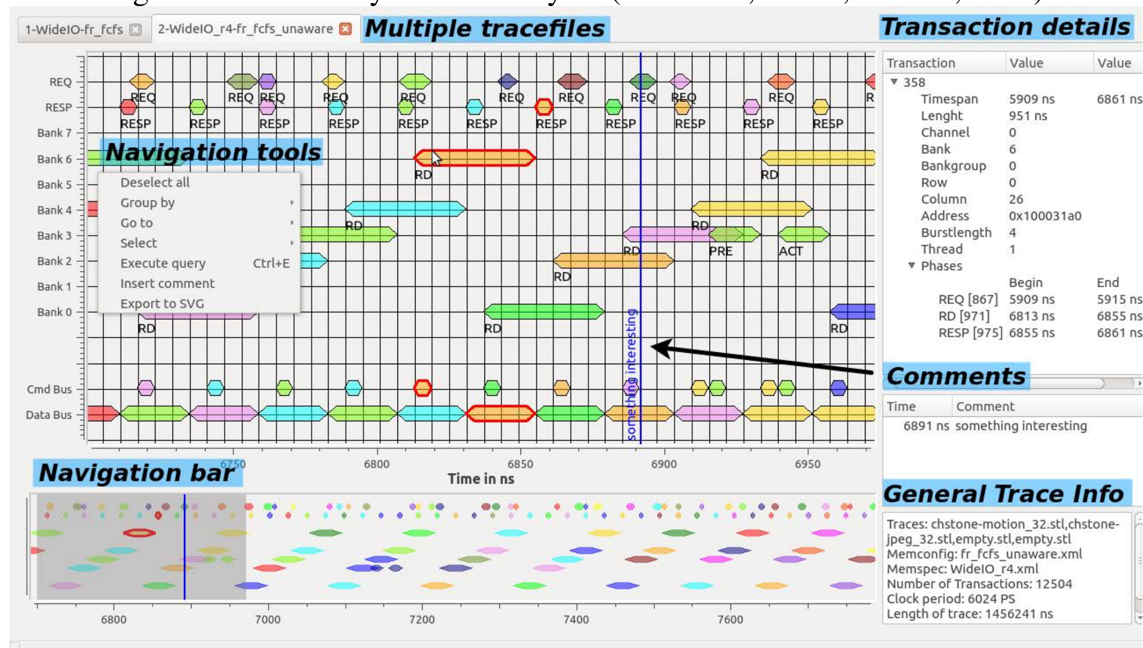
In DRAMSys, the channel controller is the module which issues requests to the memory module and responses to the arbiter. It acts as a TLM 2.0 wrapper which implements the TLM 2.0 base protocol at the target side, for the communication with the arbiter, and the DRAM-AT protocol, mentioned previously, for the communication with its associated DRAM module at the initiator side. In a similar way to the arbiter, the transactions are also stored in buffers. A request is issued after being scheduled by the controller scheduler and processed by the controller core. The scheduler is the module that manages the issue order of the requests for the purpose of achieving better performance. DRAMSys supports state-of-the-art scheduling techniques, such as FR-FCFS (First Ready - First Come First Served) (RIXNER et al., 2000) , PAR-BS (Parallelism-aware Batch Scheduling) (MUTLU; MOSCIBRODA., 2008) and SMS (Staged Memory Scheduling) (AUSAVARUNGNIRUN et al., 2012). The controller core is in charge of transforming scheduled read and write requests into a sequence of necessary commands and determining when the execution of these commands have to start according to the cur-

rent state and timing constraints of the target DRAM module. Besides, it is the module which issues refreshes, when it is required and can issue power down commands depending on the current state of the memory. After the execution of all requested commands, the memory module sends a response to the channel controller, which sends it to the arbiter and, that way, a transaction is finished.

4.5 TLM Recorder and Trace Analyzer

The TLM Recorder is the component which uses the SQLite Database Engine (SQLITE., 2000) to record memory traces into one or more database files, according to the number of memory channels, during a simulation in DRAMSys. By monitoring the communication between the components, it stores much information, such as TLM phases, requested addresses, operations and timings, which comprises a file. The recorded data can be visualized in the trace analyzer tool, which is implemented in C++ and Python and provides a graphic view of transactions and mechanisms for analysis and verification. These mechanisms include extraction of metrics, such as average latency and bandwidth, evaluation tests, like verification of timing constraints, and plots generation. It allows a quick navigation through the memory traces, and it is also useful for debugging purposes. Figure 4.3 shows the DRAMSys trace analyzer and some of its features.

Figure 4.3: DRAMSys Trace Analyzer (JUNG M.; WEIS; WEHN, 2015)



4.6 DRAMPower

Since DRAMs contribute significantly to the power consumption of today's systems, the need for an accurate power modeling is visible. In DRAMSys, this is possible through DRAMPower (CHANDRASEKAR et al., 2012), a power and energy estimation tool. This tool was modified to be used as a library, which was integrated into DRAMSys (JUNG M.; WEIS; WEHN, 2015). Therefore, it enables an online energy consumption calculation during a simulation.

5 COMPARISON OF SCHEDULING MECHANISMS

In recent years, due to multi-core environments and the continuous demands of memory, the scheduling mechanism of the DRAM controller has become increasingly important. When multiple cores share the same memory system, new aspects such as thread-fairness have to be considered. As pointed out in (MUTLU; MOSCIBRODA., 2008), in a multi-core scenario high memory throughput (number of served memory requests per second) cannot equal high performance of the overall system. Therefore, different memory scheduling techniques have been proposed to increase the overall system performance, i. e., to optimally achieve several goals, such as maximizing row-buffer hit rates, bank-level parallelism and fairness. The schedulers FR-FCFS (RIXNER et al., 2000), PAR-BS (MUTLU; MOSCIBRODA., 2008) and SMS (AUSAVARUNGNIRUN et al., 2012) nicely reflect the crescent complexity of proposed memory schedulers over the last decades. Hence, it is very important to compare the effects of these scheduling techniques on the DRAM memory system and identify whether the complexity of a mechanism is directly related to its efficiency.

In this work, the aforesaid scheduling mechanisms, proposed by other authors, were implemented in the DRAMSys framework. Afterward, this work presents results obtained through simulations with multi-core CPU models in the framework.

5.1 FR-FCFS

The First Ready - First Come First Served (FR-FCFS) is the memory scheduling mechanism proposed in (RIXNER et al., 2000) in which out-of-order DRAM operations are executed to enhance the performance of the overall system. This approach is the most common online DRAM scheduler and exploits row-buffer locality through two distinctive reordering mechanisms, conservative and aggressive reordering.

In conservative reordering, the first-ready reference is scheduled. If there are many pending references, the oldest one is chosen. This mechanism uses bank-first scheme in which accesses to different banks are prioritized over same bank accesses. If a bank is performing a precharge or an activate command, accesses from different banks can be performed. Hence, bank parallelism and concurrency are exploited, i.e., throughput is increased. However, when the oldest pending reference targets a different row than the active one in a particular bank, it would be precharged even if this bank would still

have row-hit references. Thus, a potential performance gain would be lost. On the other hand, the aggressive reordering would avoid this scenario because it takes the performance enhancement into consideration. It focuses on memory bandwidth optimization by further reordering references in such a way that maximizes the row-buffer hit ratio.

The FR-FCFS policy, as presented in (RIXNER et al., 2000), has three priority rules:

1 - Ready-First: prioritizes ready commands over unready ones.

2 - Column-First: prioritizes column accesses, i.e., read and write commands, over row accesses, i.e., activate and precharge commands. This policy maximizes the row-buffer hit rate.

3 - Oldest-First: prioritizes commands from older requests over those from more recent ones.

FR-FCFS aims to minimize the average service latency of requests and maximize the data throughput. For single-threaded systems, this scheduling mechanism provides the best average performance (MUTLU; MOSCIBRODA., 2008). Nevertheless, when multiple threads share the DRAM system, the command prioritization policy employed by the FR-FCFS can be unfair to different threads. This can be illustrated when imagining a scenario in which a thread generates accesses to a random address, while another thread acts as a stream. The streaming thread would keep generating a lot of row hits, and it would be served almost exclusively, whilst the other would be stalled for almost all the time. Besides, the oldest-policy implicitly gives higher priority to threads which generate memory requests at a faster rate than others, i.e., requests from less memory-intensive threads are not served until all earlier arriving requests from memory-intensive ones are served. Hence, FR-FCFS is thread-unware and, even though it maximizes memory throughput, threads running together on the same chip can experience different memory performance.

5.2 PAR-BS

The Parallelism-Aware Batch Scheduling (PAR-BS) is a thread-aware scheduler proposed in (MUTLU; MOSCIBRODA., 2008) that tries to establish high performance of the overall system by forming batches of execution, in which each thread has a limited number of requests per bank and a rank according to its workload. It consists of two components: request batching and parallelism-aware within-batch scheduling.

The idea of batching is to consecutively group outstanding requests in the memory

request buffer into units called batches. The maximum number of requests per bank for each thread in a batch is defined by the scheduler parameter Marking-Cap. Once all requests of a batch are serviced, i.e., when the batch is finished, a new batch is formed consisting of requests in the memory request buffer that were not included in the last batch (MUTLU; MOSCIBRODA., 2008).

The formation of batches allows for the flexibility to re-order requests within a batch in order to exploit row-buffer locality and bank parallelism without significantly disturbing thread-fairness. The first one is important because if a row-hit rate is maintained within a batch, bank accesses incur smaller latencies on average, which increases the throughput of the system. The second one is also important because scheduling multiples requests from a thread to different banks in parallel effectively reduces that thread's experienced stall-time (MUTLU; MOSCIBRODA., 2008). Hence, the parallelism-aware within-batch scheduling uses the following prioritization rules:

1 - Within-batch requests: within-batch requests are prioritized over the others.

2 - Row-hit-first: Row-hit requests are prioritized over row-miss requests.

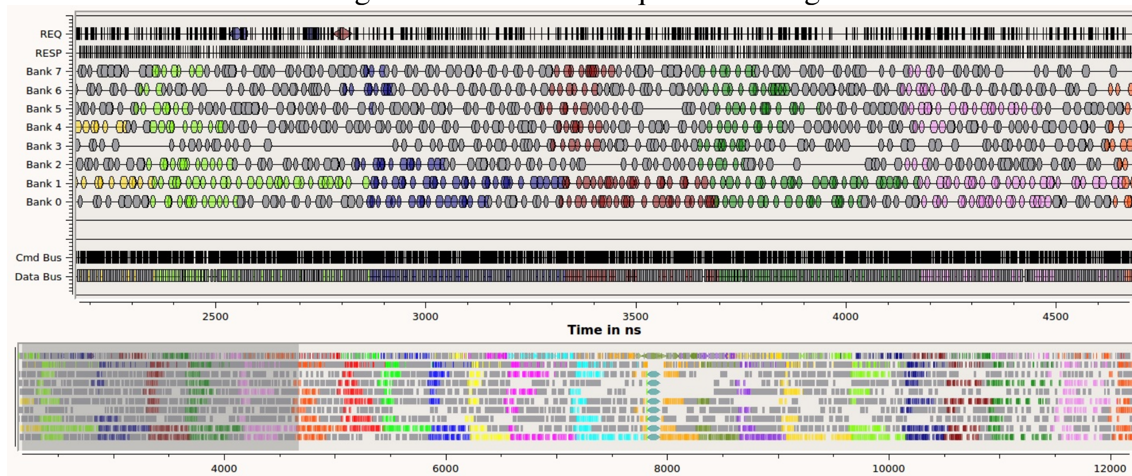
3 - Higher-rank-first: Requests from threads with higher rank are prioritized over requests from lower-ranked threads.

4 - Oldest-first: Older requests are prioritized over younger requests.

By prioritizing requests belonging to the current batch, the scheduler prevents request starvation and enforces a fair progress across all threads. When a bank is free for requests and there is no request left in the batch for that bank, a request which does not belong to the current batch is serviced, so that neither bandwidth is wasted nor unnecessary requests are delayed. Within a batch, row-hit requests are prioritized in order to increase row-buffer locality. Since a thread ranking remains the same while a batch is processed, different threads are prioritized in the same order across all banks and thus, each thread's requests are more likely to be serviced in parallel by all banks. Besides, seeing that non-intensive threads are usually ranked higher than intensive ones, PAR-BS ensures that non-intensive threads make fast progress and are not delayed unnecessarily long (MUTLU; MOSCIBRODA., 2008). Finally, all else being equal, an older request is prioritized with the purpose of minimizing the average service latency of requests. Figure 5.1 shows data extracted from a DRAMSys simulation using the PAR-BS scheduler, where each request has a specific color according to its batch, with the exception of the gray ones, which do not belong to any batch. Therefore, in contrast to FR-FCFS, which might sacrifice thread-fairness in lieu of maximizing memory throughput, the PAR-BS

tries to maximize both of them simultaneously.

Figure 5.1: PAR-BS Request Batching



5.3 SMS

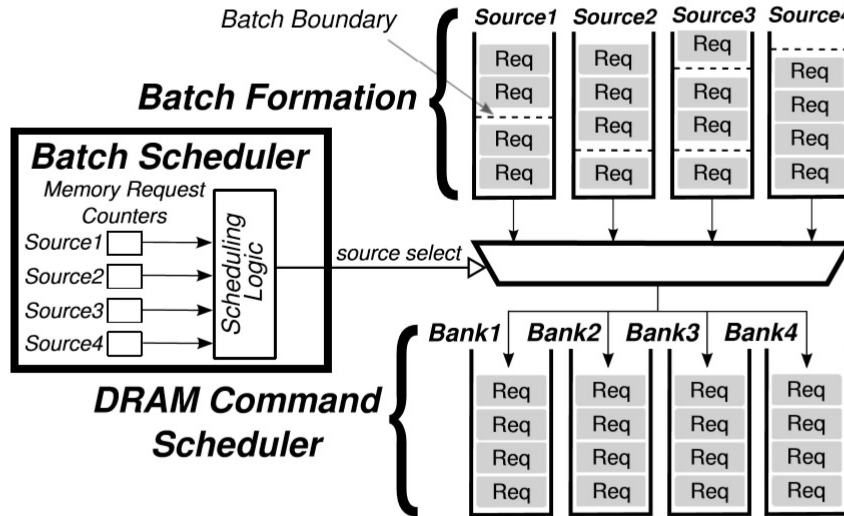
The Staged Memory Scheduling (SMS) is an application-aware scheduler proposed in (AUSAVARUNGNIRUN et al., 2012) that decouples memory controller tasks in three structures, which together try to improve the overall system performance and fairness, especially in integrated CPU-GPU systems. Therefore, the SMS composes three separate stages: batch formation, batch scheduler and DRAM command scheduler.

The goal of the first stage of the scheduler, the batch formation, is to combine individual requests from the same application based on the row-buffer locality, i.e., requests that access the same row. It consists of several First In First Out (FIFO) structures, one per source, e. g., a CPU or a GPU. Each request is initially inserted into its respective FIFO upon arrival at the memory controller. A batch is simply one or more memory requests from the same source that access the same DRAM row. Hence, except perhaps for the first one, the requests within a batch would be row-buffer hits if scheduled consecutively. A batch is deemed ready or complete when an incoming request accesses a different row, when the oldest request in the batch has exceeded a threshold age, or when the FIFO is full (AUSAVARUNGNIRUN et al., 2012). The threshold age of an application is determined by the dynamic parameter Miss per Thousand Cycles (MPKC), which changes over the application execution. The lower the memory intensity of an application the lower is its threshold age. Only ready batches are considered by the second stage of the SMS, the batch scheduler.

The second stage, the batch scheduler, deals directly with batches and therefore does not need to worry about scheduling to optimize for row-buffer locality. Instead, it focuses on higher-level policies regarding inter-application interference and fairness. The goal of this stage is to prioritize batches from applications that are latency critical, while making sure that bandwidth-intensive applications such as the GPU still make reasonable progress (AUSAVARUNGNIRUN et al., 2012). It schedules the batches using either a Shortest Job First or a round-robin policy. By using the SJF policy, the scheduler chooses the oldest ready batch from the source with the fewest total in-flight memory requests across all the three stages of the SMS. By using the round-robin policy, it simply picks the next ready batch in a round-robin manner across the source FIFOs. The first one reduces the average requests service latency, which tends to favor the latency-sensitive applications, while the second one ensures that high memory-intensity applications receive adequate service (AUSAVARUNGNIRUN et al., 2012). The configurable parameter Shortest-Job-First Probability defines how often the SJF policy is used, i.e., how frequent non-intensive applications are prioritized over intensive ones. After picking a batch, the batch scheduler forwards the requests from the selected batch to the final stage of the SMS. It simply dequeues the requests one at a time until all requests of the batch have been removed from the source FIFO. It then selects the next batch to be forwarded.

The third and final stage of the SMS, the DRAM command scheduler (DCS), handles low-level DRAM commands, e.g., activate and precharge, and consists of one FIFO queue per DRAM bank. After dequeuing a request from a batch, the batch scheduler moves it to one of these queues. Row-buffer locality within a batch is preserved into a DCS FIFO because batches are moved one batch at a time. Moreover, seeing that previous stages have already made application-aware scheduling decisions, at this point, the scheduler simply services the requests in the order they arrived. Figure 5.2 shows the three stages of the SMS.

Figure 5.2: Organization of the SMS (AUSAVARUNGNIRUN et al., 2012)



Different from the previous schedulers, SMS considers heterogeneous systems and uses additional structures and separate stages. Like the PAR-BS, it also tries to deliver high throughput and fairness simultaneously. However, due to its own arbitration mechanism and more complex design, it tries to better identify and exploit the different characteristics of applications.

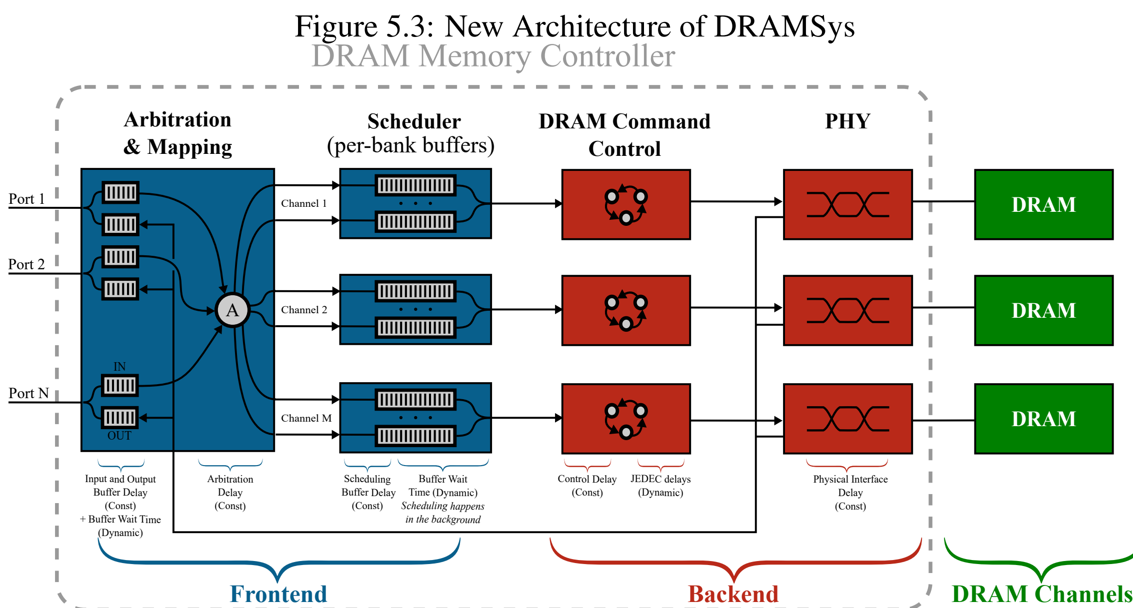
5.4 DRAMSys Restructuring

A multi-threaded simulation is fundamental for simulating DRAM schedulers since one of the goals of a scheduler can be maximizing the thread-fairness. Although DRAMSys supported multi-threaded simulation, in order to perform realistic simulations, it was required to modify some of its components such as models which did not reflect the real hardware behavior suitably. In other words, the main modules, arbiter, controller, and trace analyzer had to be restructured. Once restructured, DRAMSys satisfied the requirements to be the main environment of this work.

The restructuring of the arbiter and the controller, performed in this work, comprised the implementation of size-limited buffers, additional inter- and intra-communication delays, a new TLM 2.0 communication protocol between the two components and a new arbitration scheme. The implementation of size-limited buffers includes the definition of a size limit for every buffer of the arbiter and controller and a backpressure mechanism. The implementation of new inter- and intra- communication delays consists of the buffer delay, the delay for putting data into a buffer; the arbitration delay, the time spent by the arbiter to perform the arbitration scheme once; the control delay, the time

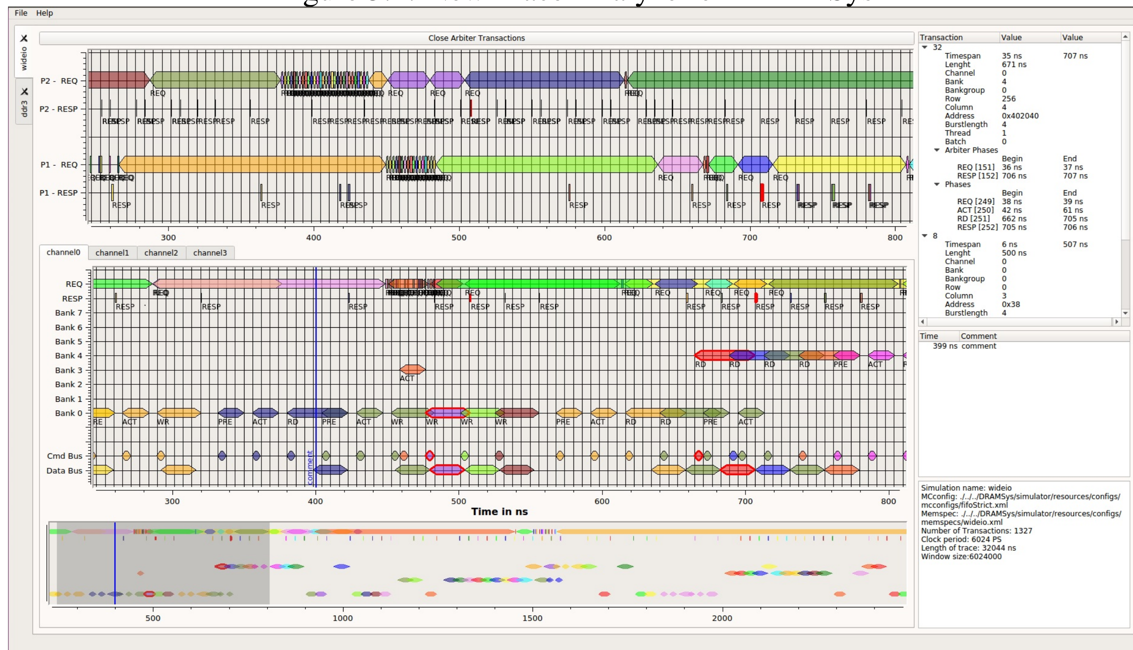
spent by the controller for timing verification and for splitting a new request into one or more commands; and the physical interface delay, the delay for translating digital data into an analog DRAM command. The new TLM 2.0 communication protocol allows a simultaneous dispatch of requests to the same channel controller, which enables the new arbitration scheme, which uses the oldest-first policy, to select requests from all the input buffers and send them to the controller at once. Figure 5.3 presents these changes.

Figure 5.3: New Architecture of DRAMSys



With the purpose of exploiting the aforementioned modifications, changes in the database schema, and consequently, in the TLM Recorder were required. The new database schema and TLM Recorder allow a whole multi-threaded simulation to be recorded in just one database file, in contrast to the previous version which recorded one file per channel. That way, the trace analyzer provides a complete view of the simulation, including transactions between initiators and the arbiter, instead of only those between controller and arbiter. Figure 5.4 shows the new trace analyzer and some of its new features. This restructuring does not just increase the simulation and analysis potential of the tool but also its debugging capabilities.

Figure 5.4: New Trace Analyzer of DRAMSys



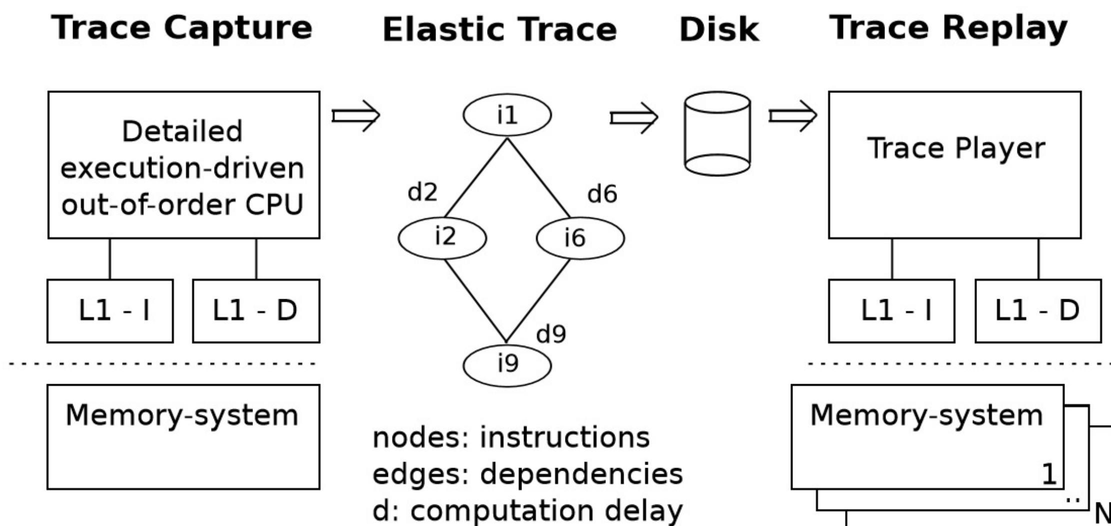
5.5 Elastic Traces

In addition to a multi-threaded simulation environment, it is essential that the initiator modules in DRAMSys mimic the execution of out-of-order processors, issuing requests according to some aspects that might delay or create dependencies between requests. These aspects are tied to the memory system performance and must be taken into account. Detailed execution-driven CPU models offer high accuracy but at the cost of simulation speed. Therefore, for the purpose of alleviating this problem, trace-driven simulation is widely adopted. However, in absolute time traces such as the native traces of DRAMSys, memory access timestamps are inferred offline, ignoring part of these aspects, which may lead to incorrect results. Hence, elastic traces (JAGTAP et al., 2016) were used in this work.

Elastic traces accurately capture memory dependencies by instrumenting a detailed out-of-order execution-driven CPU model. During the run time of a detailed out-of-order pipeline in the simulation tool gem5 (BINKERT et al., 2011), the elastic trace is formed by tracking instruction dependencies and their relative delays inside the processor pipeline, and thus they do not rely on offline analysis of timestamps. Besides, it also records effects of branch misprediction, which have a non-negligible impact on the memory system (JAGTAP et al., 2016). The execution of instructions is modeled as a

directed acyclic graph where instructions are nodes and dependencies are edges. Seeing that processor resources, such as the Issue Queue and the Reorder Buffer, affect the system performance, they are also taken into consideration. Thus, data dependencies such as read-after-write ones, order dependencies for loads and stores and speculative and wrong path loads are captured. After the elastic traces generation, the elastic trace player, implemented in the simulator gem5, enables playing them back by connecting it to different memory system models. The player respects the recorded dependencies and adapts its execution time to memory-system changes, as the actual CPU would do. Figure 5.5 illustrates the trace capture and replay methodology.

Figure 5.5: Trace Capture from a detailed execution-driven CPU and Replay for Memory-system exploration. (JAGTAP et al., 2016)



The elastic traces are application- and microarchitecture-specific but memory-system independent and achieve faster simulation speed than detailed out-of-order execution-driven processor models with negligible errors. Since the elastic trace player is integrated with a SystemC TLM 2.0 interface, elastic traces can be replayed in DRAMSys. Therefore, elastic traces enable accurate memory-system exploration, and consequently a realist analysis of memory scheduling mechanisms in the framework.

5.6 Experiments

The scheduling mechanisms were evaluated using the DRAMSys and elastic traces. The traces modeled multi-core CPUs, including per-core L1 and shared L2 caches. Each elastic trace modeled one core executing a single thread. The parameters for the CPUs are shown in table 5.1.

Table 5.1: CPU parameters

Instruction Set Architecture	ARMv7-A
Cores	1-8
Core clock frequency	4 GHz
L2 Cache	2 MB 16-way associative 12-cycle latency
L1-I Cache	32 KB 2-way associative 1-cycle latency
L1-D Cache	32 KB 2-way associative 2-cycle latency
Issue width	8
Pipeline stages	8
Physical INT/FP registers	128/192
IQ entries	32
LSQ entries	16 each
ROB entries	40

The parameters for the memory system modeled in DRAMSys are presented in table 5.2. Although the DDR4 standard is coming now slowly into the market, the DDR3-1600 (Micron Technology Inc., 2006) standard keeps being widely used in computer systems and, as mentioned previously, the RBC is one of the most common address mappings used nowadays. Scheduler-specific parameters and the size of the arbiter and controller buffers were defined based on (AUSAVARUNGNIRUN et al., 2012) and (MUTLU; MOSCIBRODA., 2008).

The experiments of this work used elastic traces simulating the execution of the

Table 5.2: DRAM parameters, VDD = 1.5V and f = 800 MHz (Micron Technology Inc., 2006)

DRAM Chip	Micron DDR3-1600	Timing	[clk]	Current	[mA/channel]
Number of Channels	1	t_{RRD}	5	I_{DD0}	70.00
Number of Ranks	1	t_{RCD}	10	I_{DD2N}	45.00
Number of Banks	8	t_{RL}	10	I_{DD3N}	45.00
DRAM Row-buffer Size	1 KB	t_{WL}	8	I_{DD4R}	140.00
DRAM Controller Clock Rate	1 GHz	t_{WTR}	6	I_{DD4W}	145.00
Input Buffer Size	8	t_{RP}	10	I_{DD5}	170.00
Output Buffer Size	8	t_{RAS}	28	I_{DD2P}	12.00
Scheduling Buffer Size	16	t_{WR}	12	I_{DD3P}	30.00
Address Mapping	RBC	t_{RFC}	59	I_{DD6}	8.00
Marking-Cap (PAR-BS)	5	t_{XP}	6		
Per-Source Queue Size (SMS)	8	t_{CKE}	3		
Per-Bank Queue Size (SMS)	16	t_{XSR}	96		
SJF Probability (SMS)	0.8	t_{CKESR}	4		
		t_{RC}	38		

stream and random applications presented in (MOSCIBRODA; MUTLU, 2007). Each elastic trace simulates around 500 thousand read and write requests. The stream trace accesses each memory column several times in a row and thus has high row-buffer hit ratio and low bank-parallelism, and the random trace issues memory accesses randomly and hence has low row-buffer hit ratio and high bank-parallelism. The figures 5.6 and 5.7 illustrate the behavior of the stream and random traces respectively. Since they have opposite characteristics, simulations with them enabled a suitable investigation regarding the thread-fairness. To analyze the overall system performance, bandwidth utilization ratio, which is directly related to the throughput, energy consumption and the number of accesses per activate command were also measured.

The fairness was measured based on the unfairness metric used in (AUSAVARUNG-NIRUN et al., 2012), which uses the slowdown of CPU cores sharing the same DRAM system. The slowdown of a core is the number of instructions per clock (IPC) it issues when running together with other cores divided by the number of IPC it issues when running alone on the same system:

$$Slowdown_i = \frac{IPC^{alone}_{Core_i}}{IPC^{shared}_{Core_i}}. \tag{5.1}$$

Hence, the fairness ratio is the the minimum slowdown over all the cores divided by the

Figure 5.6: Stream Trace

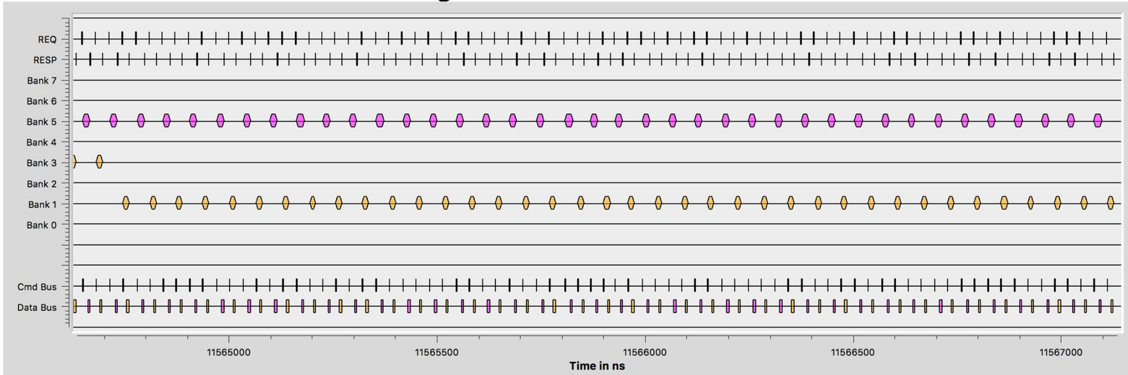
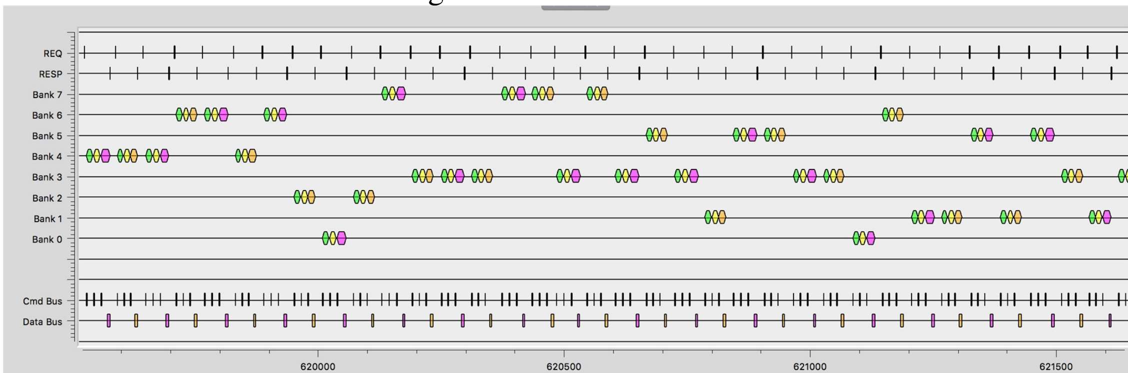


Figure 5.7: Random Trace



maximum slowdown over all cores:

$$Fairness = \frac{min_i Slowdown_i}{max_j Slowdown_j}. \tag{5.2}$$

The bandwidth utilization ratio and the number of accesses per activate command were measured through the DRAMSys trace analyzer, and the energy consumption was measured using the DRAMPower energy estimation.

In addition to the scheduling mechanisms described previously, the First In First Out (FIFO) technique, also known as First Come First Served (FCFS), was also simulated. This mechanism just services the memory requests exactly in the order they arrived in the memory controller. Since it is no scheduler at all, it was used as the baseline of the experiments for accurately perceiving the effectiveness of schedulers.

Based on the behavior of the stream and random traces, a set of combinations of these traces modeling 2, 4 and 8-core systems was replayed in DRAMSys. These combinations exploited scenarios, in which the memory system is led to service requests according to the characteristics of the majority or half of the cores. Table 5.3 presents the setup of the experiments.

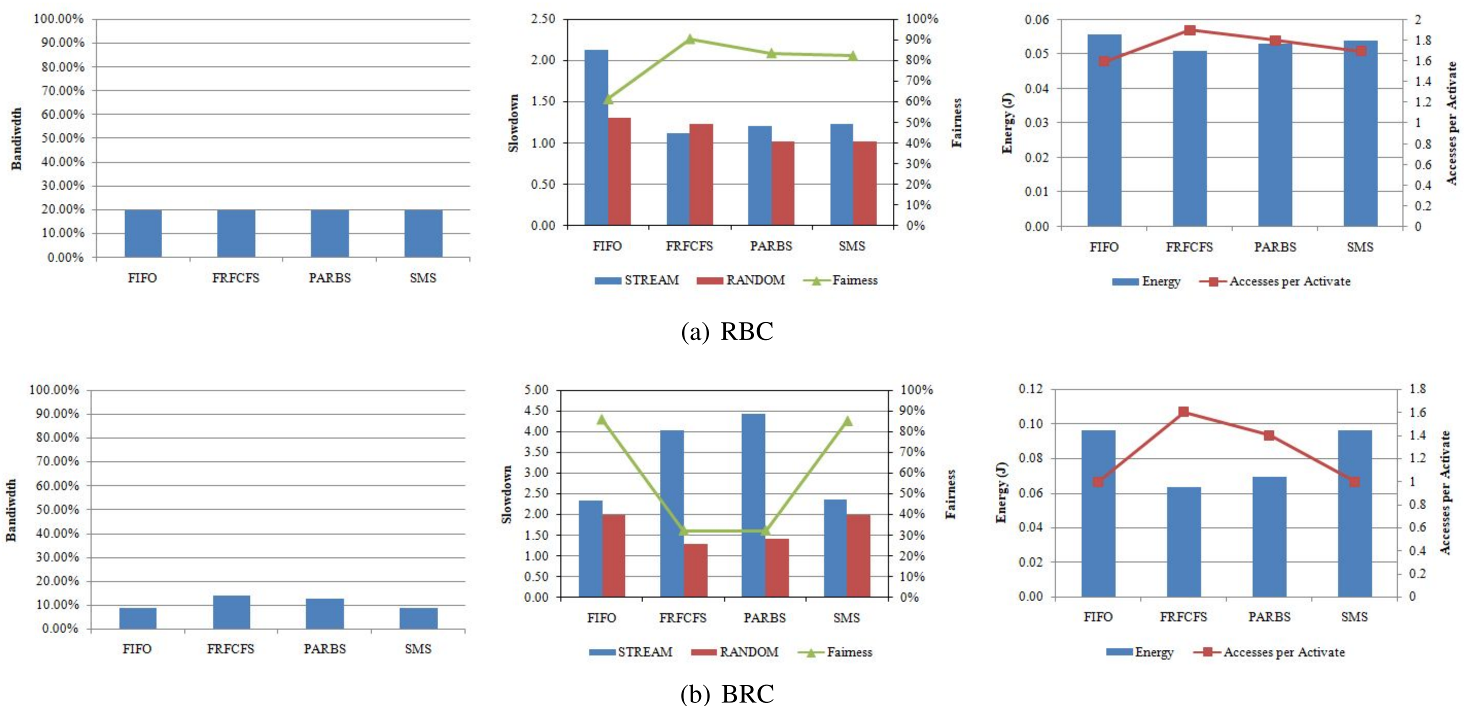
Table 5.3: Setup of the Experiments

Experiment	Number of Cores	Number of Traces	
		Stream	Random
1	2	1	1
2	4	1	3
		3	1
		2	2
3	8	1	7
		7	1
		4	4

5.7 Results

Initially, the efficiency of the schedulers and the baseline in experiment 1 are compared. Its results are shown in figure 5.8(a). As expected, even for 2-core systems, the baseline produced the worst results. Seeing that PAR-BS and SMS are more appropriate for systems with many cores, it was expected that their performance would not be as good as that of the FR-FCFS in this experiment. Even though the FR-FCFS provided the highest fairness, it is possible to observe that it slowed both applications down by 18%

Figure 5.8: Results of Experiment 1



(a) RBC

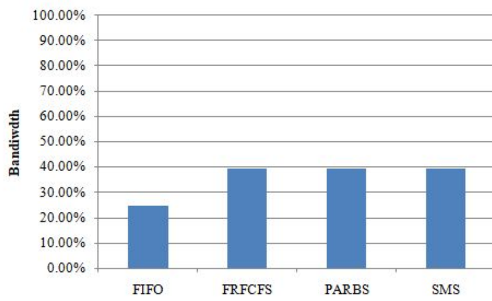
(b) BRC

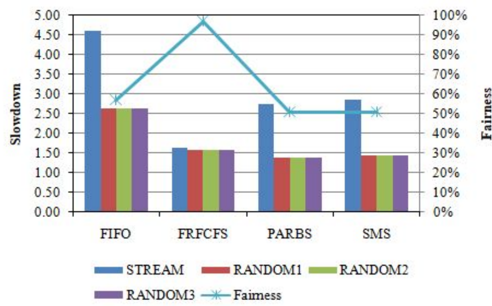
on average while PAR-BS and SMS slowed just the stream trace down by around 20%. The thread-aware schedulers prioritized row-miss requests from the random trace over row-hit requests from the stream trace sometimes while the FR-FCFS always prioritized the row-hits, stalling the random trace more. On the other hand, due to the size-limited buffers of the memory controller, the FR-FCFS was led to service at least one row-miss request every time a buffer was full of requests from the random trace, which kept the scheduler from being unfair even with a high bandwidth. The bandwidth utilization was almost the same regardless of the technique applied, due to the small number of cores. Since the FR-FCFS is well suited to achieve the maximum possible number of row-buffer hits and consequently the maximum possible number of accesses per activate command, it was expected that it would consume less energy than the others. It is important to notice that the other mechanisms still consumed less energy than the baseline. In order to present the effect of the address mapping, experiment 1 was repeated using the BRC addressing scheme. The results of this experiment are shown in figure 5.8(b). It is noticeable a big increase in the slowdown and energy consumption, besides a big decrease in the bandwidth in comparison to the results using the RBC scheme. Therefore, regardless of the scheduler, a bad address mapping will give bad results, and the better the addressing scheme, the better a scheduler can perform.

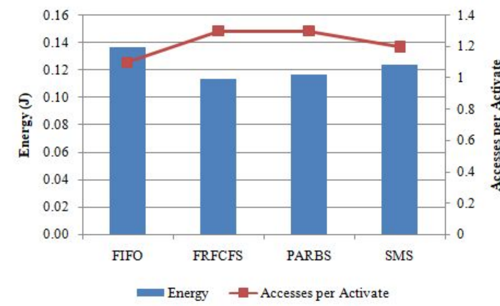
Figures 5.9 and 5.10 show the results of experiments 2 and 3 respectively. Remarkable differences can be noted, mainly regarding the fairness. When the majority of the cores were stream traces, the thread-aware schedulers delivered a higher average bandwidth than the FR-FCFS, because the single random trace was stalled and its simple scheduling scheme ended up not being clever enough to exploit bank-level parallelism. In the other cases, since there was more than one random trace, bank-level parallelism could be easier explored, and thus the schedulers produced similar bandwidth utilization ratios, with differences less than 5% on average. On the other hand, FR-FCFS provided the highest fairness. In experiment 2, the FR-FCFS kept stalling the random traces more and the stream ones less than the thread-aware schedulers did, but, unlike in experiment 1, it also presented an average slowdown less than those presented by the other mechanisms. In experiment 3, when the majority of the cores were random traces, the fairness ratios produced by the schedulers were similar. In view of the huge number of row misses, even the FR-FCFS was not able to avoid slowing the single stream trace down overly. In the other cases, it delivered the highest fairness, as in the 4-core simulations. However, in all cases, it slowed the stream traces down more than the random ones, which resulted

in lower fairness rates than those presented in experiment 2, indicating that the higher the number of cores, the less the FR-FCFS is able to be fair. Moreover, the SMS stalled the stream and random traces more than the PAR-BS did, and, therefore, it produced lower fairness ratios. Although the fairness ratios presented by the baseline were similar to those presented by the thread-aware schedulers, it is fundamental to observe that the baseline slowed all cores down much more than the others did. Therefore, a high fairness ratio does not always entail an efficient scheme. Even for 4- and 8-core systems, the FR-FCFS scheduler kept being the most energy-efficient due to its simple prioritization scheme which optimizes the row-buffer hit ratio.

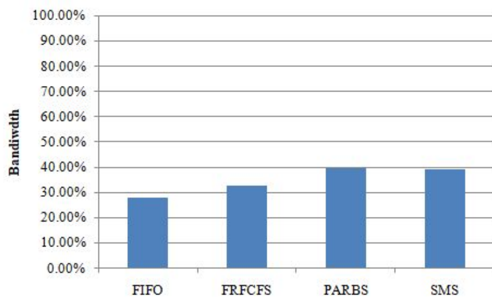
Figure 5.9: Results of Experiment 2

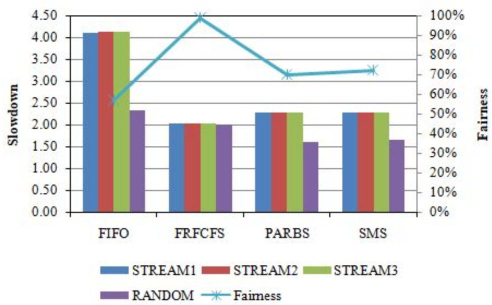


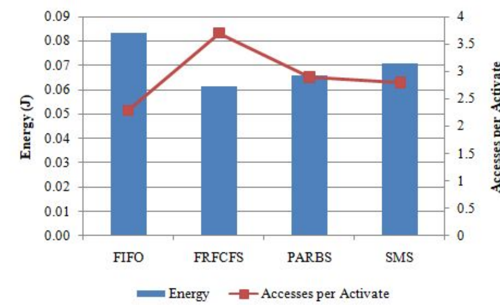




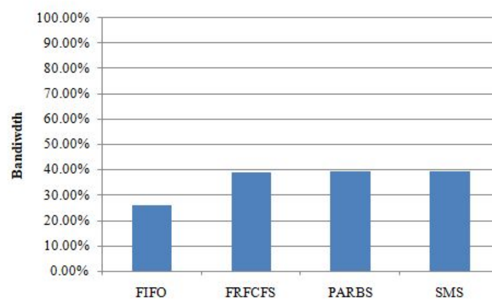
(a) 1 Stream and 3 Random

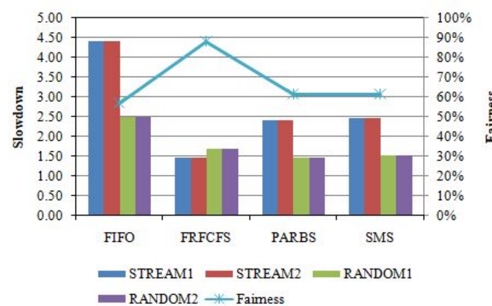


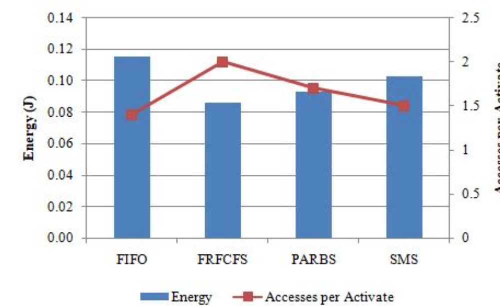




(b) 3 Stream and 1 Random

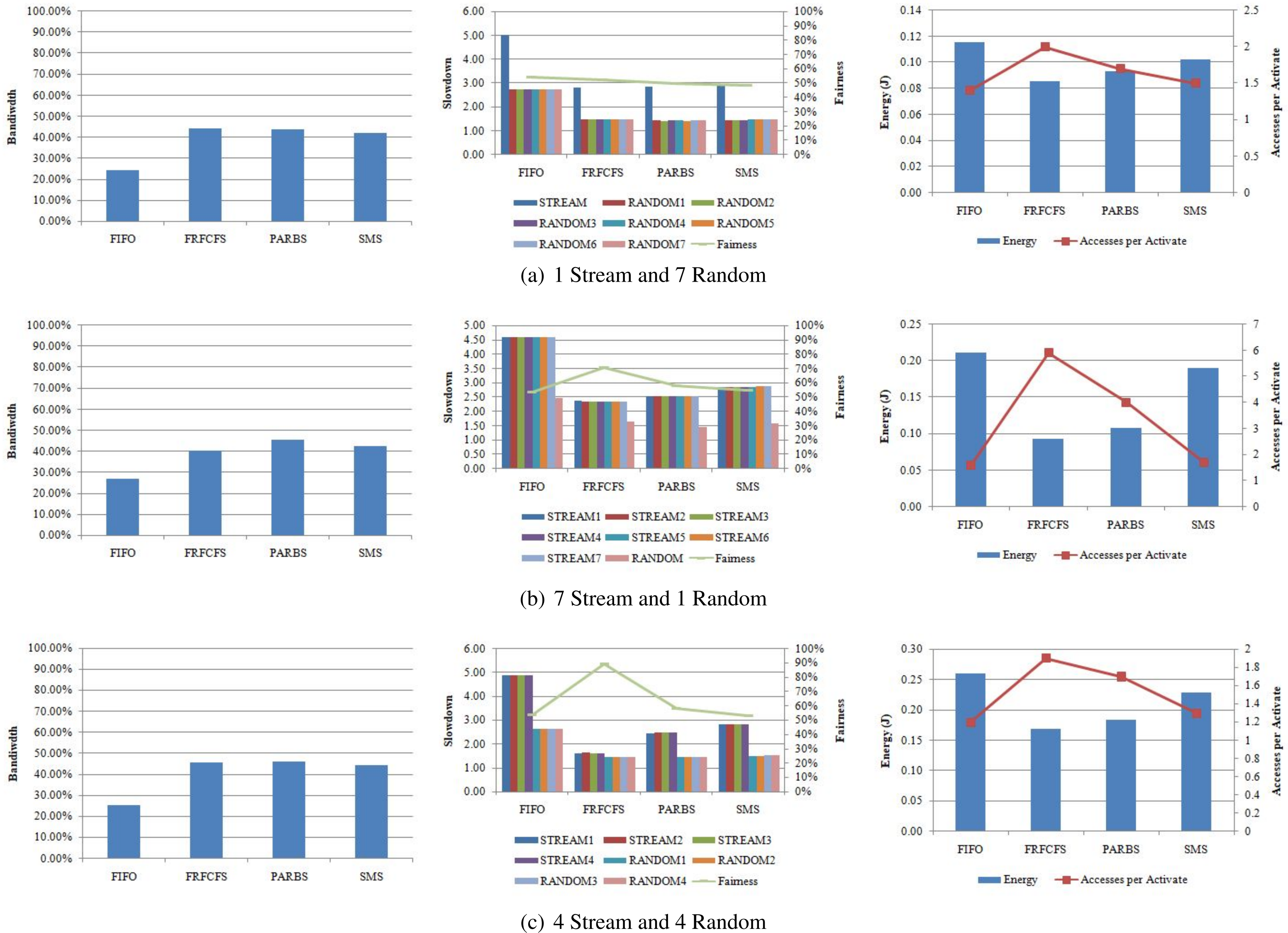






(c) 2 Stream and 2 Random

Figure 5.10: Results of Experiment 3



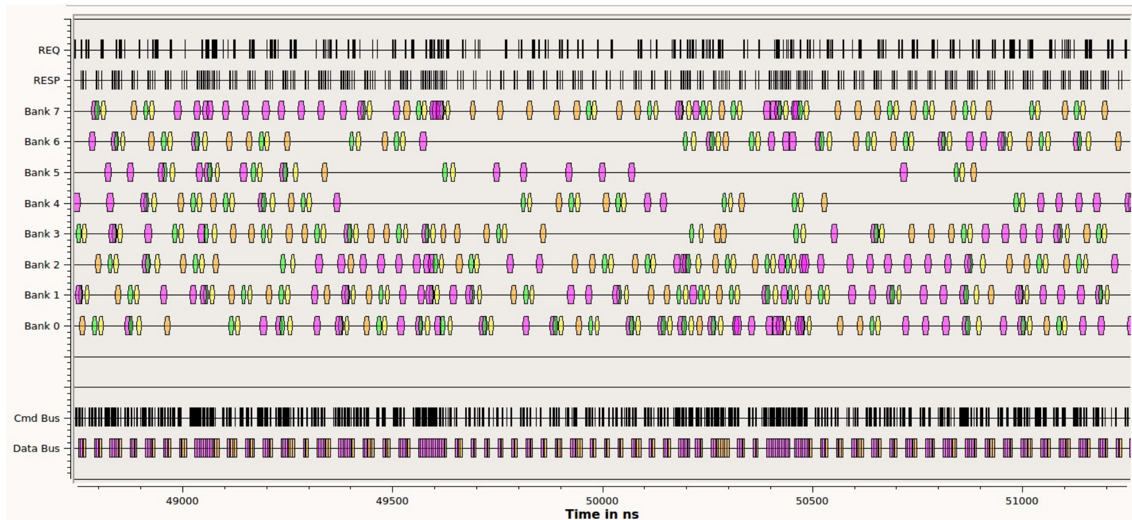
5.8 Concluding Remarks

The results of the experiments evinced the importance of the scheduling mechanisms, which can improve not only the throughput but also the energy efficiency and thread-fairness of the DRAM memory system. Although the FR-FCFS does not aim at maximizing fairness, it presented an automatic fair scheduling, and, owing to its simple prioritization scheme, it was the most energy-efficient scheduler. The PAR-BS proved to be efficient as well because it demonstrated a good trade-off between bandwidth and energy consumption in the experiments, in which it delivered the highest average bandwidth and presented an average energy consumption less than 10% greater than that presented by the FR-FCFS. As expected, the SMS provided a reasonable high but not the highest

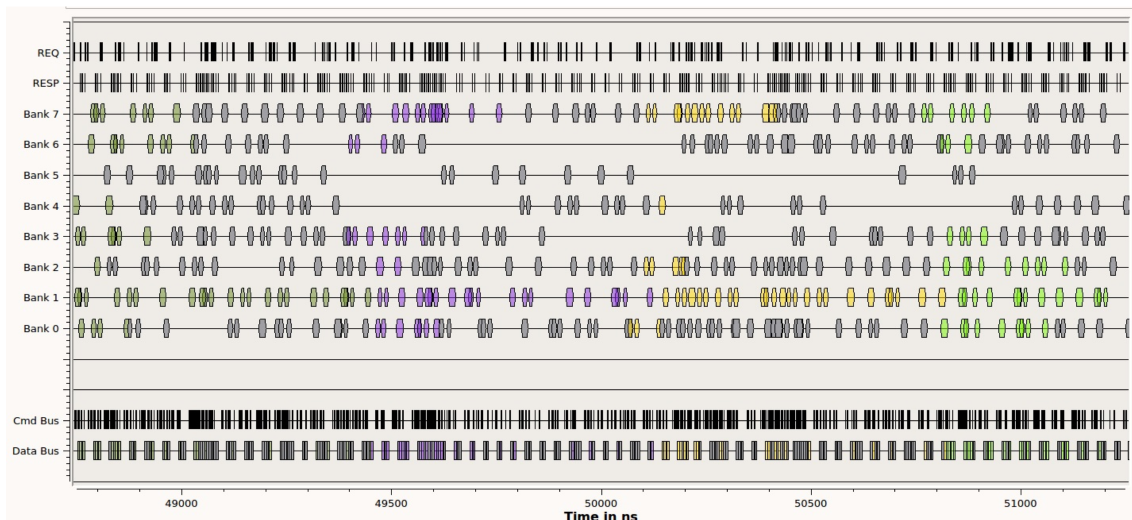
overall system performance, because, as mentioned before, it is well suited for CPU-GPU systems, and, as pointed out in (AUSAVARUNGNIRUN et al., 2012), it can be employed in CPU-only systems, although it might not be as effective as CPU-only schedulers.

Knowing that there is a timing penalty every time a write is issued right after a read, and vice versa, through the simulations performed in this work, it was possible to note the effect of read-write switches, i.e., consecutive executions of read and write commands, on the DRAM system performance. The figures 5.11(a) and 5.11(b) present the same time interval of a simulation, in which operations are grouped by phase and by batch respectively. In the first one, precharge, activate, write and read operations are colored green, yellow, orange and purple respectively. In the second one, requests which do not belong to any batch are colored gray while the others have specific colors according

Figure 5.11: PAR-BS Simulation



(a) Grouped By Phase



(b) Grouped By Batch

to their batches. Although these figures show data extracted from a simulation using the PAR-BS, simulations using other techniques repeated the data bus scenario. In the first figure, it is possible to observe multiple gaps between the data on the data bus. Seeing that reads were not always prioritized over writes, these gaps were formed due to frequent read-write switches, and thus bandwidth may have been wasted. Moreover, the figures show why the thread-fairness provided by the PAR-BS in the experiments was not as high as expected. Because of read-write switches, a PAR-BS batch took longer to be serviced and then many times the scheduler ended up servicing requests which do not belong to any batch. Therefore, since the batch boundary and the thread-ranking were often ignored, its thread-aware scheduling scheme was ruined. Hence, a scheduling mechanism should also manage the order of execution of read and write commands in such a way that read-write switches were minimized.

6 CONCLUSIONS AND FUTURE WORK

In this work, DRAM design space was exploited using virtual prototyping with SystemC/TLM 2.0. The benefits of this method allowed the implementation of several DRAM scheduling mechanisms and a realistic modeling and fast simulation of the DRAM memory system. Despite simulating multi-core systems, the results of the experiments showed that FR-FCFS is the most efficient scheduler, owing to its simple prioritization policy combined with other DRAM controller mechanisms. However, since the experiments comprised just applications with particular characteristics and some memory system mechanisms still can be better modeled, this does not imply that this scheduler is necessarily the best option among the schedulers studied. As shown in (JUNG, 2017), there are other aspects to be explored on DRAMs, such as the refresh retention time, thermal behavior, bank-wise power-down modes and advanced addressing schemes, which can contribute to the efficiency of the scheduling mechanisms and even to the design of new ones. Hence, the next step is the modeling of an efficient scheme, which prioritizes reads over writes being aware of data hazards and groups the execution of these commands in such a way that read-write switches are minimized and bandwidth is minimally wasted. As mentioned in (MUTLU; MOSCIBRODA., 2008), read requests must be prioritized because they can directly block forward progress in processing cores and are therefore more performance critical. Furthermore, experiments using applications with different memory-intensity levels would better exploit the application awareness of the SMS and the thread-ranking of PAR-BS, which would culminate in a refined analysis of the thread-fairness provided by the different schedulers. Therefore, the research on scheduling mechanisms for DRAMs must go forward.

REFERENCES

ACCELLERA. **Open SystemC Initiative (OSCI).** 2013. <accellera.org/community/systemc>. [Online; accessed April-2017].

AUSAVARUNGNIRUN, R. et al. Staged memory scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In: **The 39th Annual International Symposium on Computer Architecture, ISCA '12**. [S.l.: s.n.], 2012.

BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, v. 39, n. 2, p. 1–7, 2011.

BLACK, D. C.; DONOVAN., J. **SystemC: From The Ground Up.** [S.l.]: Academic Publishers, 2004.

BURGER, D.; AUSTIN, T. M. The simplescalar tool set, version 2.0. **SIGARCH Comput. Archit. News**, v. 25, n. 3, p. 13–25, 1997.

CHANDRASEKAR, K. et al. **DRAMPower: Open-source DRAM Power and Energy Estimation Tool.** 2012. <http://www.drampower.info>. [Online; accessed April-2017].

DOULOS. **Open SystemC Initiative (OSCI).** 2013. <https://www.doulos.com/knowhow/systemc/tlm2/at_example>. [Online; accessed April-2017].

IEEE Computer Society. **IEEE Standard for Standard SystemC Language Reference Manual. IEEE Standard 1666-2011 (Revision of IEEE Std 1666-2005).** [S.l.], 2012.

JACOB B., N. S.; WANG, D. **Memory systems: Cache, DRAM, Disk. Morgan Kaufmann.** [S.l.]: Morgan Kaufmann, 2008.

JAGTAP, R. et al. Exploring system performance using elastic traces: Fast, Accurate and Portable. In: **IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation, SAMOS**. Samos Island, Greece.: [s.n.], 2016.

JEDEC. **JEDEC Homepage.** 1958. <https://www.jedec.org/>. [Online; accessed April-2017].

JUNG, M. **System Level Modeling, Analysis and Optimization of DRAM Memories and Controller Architectures**. Thesis (PhD) — University of Kaiserslautern, 2017.

JUNG, M. et al. Tlm modelling of 3d stacked wide i/o dram subsystems: A Virtual Platform for Memory Controller Design Space Exploration. In: **The 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '13**. New York, NY, USA.: ACM, 2013. p. 5:1–5:6.

JUNG M.; WEIS, C.; WEHN, N. Dramsys: A Flexible Dram Subsystem Design Space Exploration Framework. **IPSJ Transactions on System LSI Design Methodology**, v. 8, p. 63–74, 2015.

KOGEL, T. **Generating Workload Models from TLM-2.0-based Virtual Prototypes for Efcient Architecture Performance Analysis.** 2010. <http://www.nascug.org/events/13th/tlm20_workload_models.pdf>. [Online; accessed April-2017].

Micron Technology Inc. **1Gb: x4, x8, x16 DDR3 SDRAM Features.** 2006. <https://www.micron.com/resource-details/31780d73-ea30-4400-af00-86e22c6c22ea>. [Online; accessed July-2017].

MOSCIBRODA, T.; MUTLU, O. **Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems**. [S.l.], 2007.

MUTLU, O.; MOSCIBRODA., T. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. **SIGARCH Comput. Archit. News**, v. 36, n. 3, p. 63–74, 2008.

RIXNER, S. et al. Memory access scheduling. **SIGARCH Comput. Archit. News**, v. 28, n. 2, p. 128–138, 2000.

SMART, K. **ISCUG: Virtual Prototyping From Evolution to Revolution. Synopsys**. 2013.

SQLITE. **SQLite Project Hompage.** 2000. <http://www.sqlite.org/>. [Online; accessed April-2017].

Synopsys Inc. **Synopsys Platform Architect MCO.** 2015. <https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html>. [Online; accessed April-2017].

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the Obvious. **SIGARCH Comput. Archit. News**, v. 23, n. 1, p. 20–24, 1995.

ZHANG, Z. et al. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In: **The 33rd Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 33**. New York, NY, USA.: ACM, 2000. p. 32–41.