Design and Formal Analysis of Real-Time Memory Controllers

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen
op maandag 26 september 2016 om 16:00 uur

door

Yonghui Li

geboren te Shaanxi, China

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.B. Smolders
1^e promotor:	prof.dr. K.G.W. Goossens
copromotor:	dr. K.B. Akesson (CISTER INESC TEC and ISEP)
leden:	prof.dr. Y. Wang (Uppsala University)
	prof.dr.ir. J.P. Katoen (RWTH Aachen)
	prof.dr.ir. C.H. van Berkel
	dr.ir. R.J. Bril

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Design and Formal Analysis of Real-Time Memory Controllers

Yonghui Li

Committee:

prof.dr. K.G.W. Goossens	Eindhoven University of Technology, *promotor*
dr. K.B. Akesson	CISTER INESC TEC and ISEP, *copromotor*
prof.dr.ir. A.B. Smolders	Eindhoven University of Technology, *chairman*
prof.dr. Y. Wang	Uppsala University
prof.dr.ir. J.P. Katoen	RWTH Aachen University
prof.dr.ir. C.H. van Berkel	Eindhoven University of Technology
dr.ir. R.J. Bril	Eindhoven University of Technology

This thesis was typeset in LaTeX, written in *Sublime Text* and built with *SCons*. The figures were created with Viso and the graphs were plotted using python *matplotlib*.

The cover of this thesis was designed by Mr. Juan Manuel Martelli.

Printed by CPI Koninklijke Wöhrmann – The Netherlands.

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-4140-9

ACKNOWLEDGMENTS

The PhD study has resulted in this thesis after four-year hard work, which involved many bright minds and productive collaborations with many people. I foremost thank my promotor Prof. Kees Goossens for his continuous encouragement, and for his sharp insights and valuable suggestions that helped to make progresses in my research and also engineering skills. For his broaden knowledge and wisdom, I would like to say that he is the one whom I can always believe. My special thank goes to my co-promotor Dr. Benny Akesson for his comprehensive assistance in the whole procedure of my research work. I extremely appreciate all the enjoyable and detailed discussions with him, which helped me to make my work more concrete. In addition to the technical work, I am also very grateful to him for taking care of me in life and for always giving me valuable advices. I would also like to thank Prof. Yi Wang, Prof. Kees van Berkel, Prof. Joost-Pieter Katoen, Dr. Reinder J. Bril, and Prof. Bart Smolders for being my thesis committee and giving their valuable feedback.

Without productive and supportive collaborations, it is hard to come up with this thesis. In particular, I want to thank Dr. Orlando Moreira, Hrishikesh Salunkhe, and João Bastos for their help on the dataflow modeling of the memory controller. Without their persistent support, it was hard to complete this work fast and was impossible to win the best paper award from ESTIMEdia' 15. I also thank Dr. Marc Geilen for the valuable discussions of dataflow modeling. His deep insight helped me to correct my model and make the work more concrete. My special thank also goes to Dr. Kai Lampka for the generous discussions about timed automata model and for making my stay in Uppsala enjoyable. I also appreciate his help in completing a paper published in a top conference. Mladen Skelin has given very valuable feedback on a chapter of this thesis. Moreover, the discussions with him were very enjoyable and useful. He deserves my gratitude. I would also like to thank Valeriu Codreanu especially for his help in applying for the permission to use the SURFsara servers. Acknowledgment also goes to SURFsara for supporting the experiments of model checking.

In the past four years, I have had the opportunity to work with a large number of people. I am especially grateful to the Memory team, where I comprehensively learned the memory subsystem. I enjoyed the discussions during the regular meetings. I also appreciate the critical feedback from the team members on my work as well as their great help on solving (non-)technical issues. I was fortunate to be a part of the memory team, and I am thankful to the team members Sven Goossens, Manil Dev Gomony, Karthik Chandrasekar, and Jasper Kuijsten. The memory team was involved in a larger CompSoC team, where I learned more about multicore embedded systems than anywhere else. I am grateful to the CompSocers Andrew Nelson, Martijn Koedam, Gabriela

Breaban, Reinier van Kampenhout, Shubhendu Sinha, Davit Mirzoyan, Juan Valencia, Rasool Tavakoli, and Hadi Ahmadi Balef for the valuable and interesting discussions.

I am very grateful to the friendship of all members of the Electronic Systems group. Many thanks to Firew Siyoum, Shakith Fernando, Erkan Diken, S. Rehan Afzal, Francesco Comaschi, Luc Vosters, Mark Wijtvliet, Gert-Jan van den Braak, Maurice Peemen, Amir Behrouzian, Hadi Ara, Joost van Pinxten, Andreia Moço, Robinson Medina Sanchez, Sander Stuijk, for the wonderful discussions during internal/external meetings/conferences, parties, and group events. My sincere thanks give to our group secretaries Marja de Mol, Rian van Gaalen, and Margot Gordon for their special care and supportive assistances that made my stay in the office very comfortable. I am extremely thankful to these Chinese in our group, including Dongrui She, Yifan He, Hailong Jiao, Xin Chen, Jun Zhu, Bo Liu, Ang Li, Wenjin Wang, Wenfeng (Sean) Wang, Tong Geng, Qi Tang, Zechuan Li, Hubiao Yang, for their help in work and life and also the great moments that we shared together inside and outside work.

The PhD life might be not colorful without many friends for the daily life. I would like to thank my running partners Jiong Zhang, Hefeng Zhou, Zizheng Cao, Qing Wang, Jie Xie, for sharing many great moments outside. I am also grateful to Bilin Han, Caixia Liu and Roy Berkeveld, Shenhai Ran, Wang Miao and Shuli Wang, Yuqing Jiao, Pei Tang, Jun Xia, Guanna Li and Dapeng Sun, Jiquan Wang, Lin Xu and Yang Zhang, Huapeng Sui, Shengnan Lu, Yaping Luo and many others who made my stay in Eindhoven filled with lots of joy and fun.

Finally, I would like to thank my family for supporting me through all these years. Their encouragement and expectation drive me to move on. No matter how far away we are from each other, our hearts are always close. In particular, I would like to thank my farther who left us three years ago. May you find peace in heaven and you are always in my heart. Last but not least, I am very grateful to my wife YingChao Cui for supporting me to overcome the difficulties in life and work, and for sharing the exciting moments together. I cannot imagine how miserable my life might be without her accompanying in the past years. I really look forward to the future life being with her.

Yonghui Li, August 17, 2016

ABSTRACT

Design and Formal Analysis of Real-Time Memory Controllers

Modern multi-core embedded systems integrate an increasing number of heterogeneous resources to provide the necessary computational capacity for executing a variety of applications simultaneously. To reduce the cost, resources are shared by real-time and non-real-time applications. The former have timing requirements, such as a maximum response time or a minimum throughput, which must always be satisfied. The latter must be responsive and require good average performance. However, it is difficult to design these systems because of the complex sharing of resources between applications, making it challenging to meet their timing requirements.

SDRAM is typically used as a main data storage device in embedded systems to store the data for the executed applications. It is shared by memory requestors, such as processors, DMAs, and hardware accelerators, which generate diverse memory traffic in terms of arbitrary read/write transactions with variable sizes on behalf of the applications they run. As a result, the memory controller coordinating between the requestors and the SDRAM has to provide guaranteed performance for real-time applications, while still giving good average performance to the rest, in the context of this diverse memory traffic. It is difficult for a memory controller to meet this requirement, because SDRAM is a complex resource, resulting in interference between requestors when they compete for the memory. Existing real-time memory controllers execute transactions by scheduling commands to the SDRAM either (semi-)statically based on pre-computed command schedules or dynamically. Static scheduling eases the analysis by sacrificing average performance, since run-time information cannot be exploited, and it does not efficiently support variable transaction sizes. On the other hand, dynamic scheduling achieves better average performance, while the analysis becomes harder. Therefore, existing dynamically-scheduled memory controllers and/or their analyses only support fixed transaction sizes.

The goal of this thesis is to overcome the problem of designing an efficient real-time memory controller for increasingly complex systems that feature a mix of real-time and non-real-time applications. The main contributions of this thesis are 1) a memory controller, named Run-DMC, designed to efficiently deal with the diverse traffic by dynamically scheduling commands for each transaction at run-time, and 2) three analysis approaches proposed to provide the worst-case response time (WCRT) and worst-case bandwidth (WCBW) based on a formalization, a dataflow model, and a timed automata model, respectively.

The architecture of Run-DMC consists of a front-end and a back-end. The front-end connects to the requestors through a bus or Network-on-Chip (NoC). It receives transactions from different requestors and then uses a novel work-conserving *time-division multiplexing (TDM)* arbiter to send transactions to the back-end. The TDM arbiter uses time slots with variable lengths to cope with the variable-sized transactions. Moreover, the requirements of the requestors can be satisfied by allocating a different number of slots to them. The back-end executes transactions with variable sizes by dynamically scheduling commands to the SDRAM using a new command scheduling algorithm, which exploits pipelining within and between transactions.

Run-DMC is analyzed using three proposed analysis approaches. Our first approach is based on a formalization that accurately computes the time when a command is scheduled. The formalization is implemented as an open-source tool called *RTMemController*. Based on the formalization, the WCRT and WCBW can be computed, and they are guaranteed to be conservative using manual proofs, which are time-consuming to make. Our second approach switches the effort from formal analysis to modeling the timing behavior of the memory controller, which is much easier and faster. The worst-case bounds can be derived by analyzing the model with existing techniques and tools, which can automatically handle the complex interferences between transactions. The second approach is based on a mode-controlled dataflow (MCDF) model. An existing tool called Heracles is used to automatically derive the WCBW bound. However, Heracles cannot analyze the WCRT. The third approach uses a timed automata (TA) model to accurately capture the timing behavior of Run-DMC, and the bounds on WCBW and WCRT are obtained by verifying properties of the TA model via model checking. Finally, since the same memory controller is analyzed with these three analysis approaches, we investigate their strengths and weaknesses with respect to performance, portability, exploitation of static information, simulation, validation, and verification.

The proposed memory controller and the analysis approaches are experimentally evaluated. The results demonstrate that Run-DMC significantly outperforms a state-of-the-art semi-static memory controller in the average case by achieving 44.9% smaller response time and 16.7% larger bandwidth, while they are comparable in the worst-case. Moreover, we compare the performance of the three analysis approaches and quantify the impact of their underlying assumptions. The results show that the TA model outperforms the MCDF model that in turn is better than the formal analysis approach.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

In the 20th century, one of the greatest inventions was the Internet [9], which connects people and delivers information worldwide. Now, we are in 2016 and what can be expected for the future is that a massively connected world beyond the Internet will be built, where everything will be connected, leading to the *Internet of Things (IoT)* [41]. The connected world will be much more cooperative, productive, and intelligent. Similarly to the Internet where computers are the fundamental components, embedded platforms will be the "heart" of "things", such as our digital watch, mobile phones, navigation systems, factory controllers, and the computers inside cars and aircraft. Embedded platforms are designed for a particular purpose and interact with physical mechanical or electrical systems by running their applications, which often have *real-time requirements* [19], such as a time deadline or a processing throughput requirement. These requirements may be associated with the safety or mission of the system and must hence be satisfied.

To satisfy the requirements of the real-time applications, the embedded platform has to provide guaranteed performance [11], such as a worst-case execution time and/or a minimum throughput. However, it is challenging to achieve this goal because of the complexity of embedded platforms, which are composed of an increasing number of resources, such as processing cores, hardware accelerators, memories, I/O interfaces, and peripherals. Moreover, the platform supports both real-time and non-real-time applications [2, 16], and the resources in the platform are shared between all the applications. Guaranteed performance must be given to real-time applications, such that their requirements are always satisfied, while good average performance is needed by non-real-time applications to feel responsive.

Synchronous Dynamic Random-Access Memory (SDRAM) is one of the most shared resources in an embedded platform and has great impact on satisfying the requirements of applications [60]. It is accessed by memory *requestors*, such as cores, hardware accelerators, and *direct memory access (DMA)* modules, via a memory controller. Since multiple applications execute concurrently in the platform, the requestors generate *diverse traffic* for the memory controller, which receives arbitrarily-mixed read/write transactions with variable sizes. Moreover, the memory addresses of transactions correspond to different internal locations of the SDRAM, resulting in complex interferences between

transactions. *This thesis focuses on design and formal analysis of real-time memory controllers, which efficiently deal with the diverse memory traffic.*

This chapter starts with Section 1.1, which introduces real-time embedded systems. The requirements of applications are discussed, followed by a brief introduction of the main memory sub-system in an embedded hardware platform. The problems of designing and analyzing of real-time memory controllers are given in Section 1.2. In Section 1.3, we briefly discuss how our contributions address the raised issues.

1.1 REAL-TIME EMBEDDED SYSTEMS

This section investigates some general trends in the application requirements, modern multi-core hardware platforms, and main memory subsystems. In particular, the main memory subsystem is presented in more detail, since it is the basis of the work in this thesis.

1.1.1 *Application Requirements*

There is a variety of applications running on modern multi-core embedded platforms, including hard/soft real-time and non-real-time applications [61]. The hard real-time applications have deadlines that must always be respected. For example, a longitudinal flight controller adjusts the longitude of an aircraft by changing its speed within a given time period [83]. It is unacceptable to miss a deadline, since it can cause catastrophic consequences, such as an aircraft crash and the loss of lives. In contrast, soft real-time applications are not safety critical, and certain deadline misses are tolerable, though they are highly undesirable. One or more overrun deadlines for soft real-time applications may result in temporary quality/service degradation, but will not lead to a catastrophe. For example, displaying a video stream on mobile devices can accept occasional dropped frames, which have a little negative effect on the *Quality of Service (QoS)* for the users [106]. Finally, non-real-time applications, such as web browsing [89], do not have any timing constraints and can tolerate occasional slow response times. However, they have to be fast enough to keep the user happy and have a good average performance.

This thesis focuses on how to provide guaranteed performance for the main memory, such as the *worst-case response time* and *bandwidth*. The worst-case response time is the maximum latency experienced by a transaction in the memory controller. The worst-case bandwidth represents the minimum data transfer rate of the memory controller over a long time period. They are used to satisfy the requirements imposed by the hard real-time applications on the memory subsystem. In this thesis, we do not distinguish soft real-time and non-real-time applications. The reason is that soft real-time applications are typically supported by just providing good enough average performance. Note that this thesis does not address the issues of how to efficiently allocate resources (e.g.,

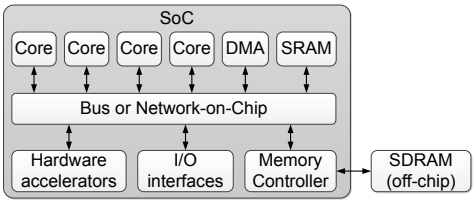


Figure 1.1: An example of a multi-core hardware platform.

memory space, bandwidth) to the applications, such that their requirements are satisfied. There exist solutions in [8, 46, 77] to solve these issues.

1.1.2 *Multi-Core Hardware Platforms*

With the advances in semiconductor technology with respect to shrinking transistor dimensions, an increasing number of transistors are integrated on a single chip [57]. This high density of transistors makes it possible to integrate multiple cores in a *System-on-Chip (SoC)*. As a result, a large amount of off-chip circuitry can be moved from printed circuit boards to integrated circuits. This allows manufactures to produce smaller boards, while simplifying the board layout and routing, reducing power consumption and cost [54]. It also reduces the complexities involved in high-speed board design. Due to the abundant resources on a chip, it allows multiple applications to be executed concurrently.

Multi-core platforms have been widely used in almost all present electronic systems, such as consumer electronics [60, 103], telecommunication systems [14], and automotive systems [17]. They are also promising for use in avionics [82]. Multi-core platforms executing multiple applications usually consist of various heterogeneous hardware resources. Figure 1.1 presents an example of a general hardware platform, which can be used for smart phones [86, 104] for instance. It contains a number of processing cores, on-chip *static random-access memory (SRAM)*, hardware accelerators, DMAs, I/O interfaces, and also the off-chip memory (i.e., SDRAM), etc. The cores can be either used to perform general-purpose computation [49] or *digital signal processor (DSP)* cores for numerical manipulation of signals [78]. Hardware accelerators are specialized for accelerating specific functions [102]. For example, video or audio engines are often implemented using hardware accelerators [76]. The on-chip SRAM is fast and used for the cache or scratchpad. However, the capacity of the SRAM cannot be large, e.g., maximally a few megabytes, because of the relatively high cost per bit (i.e., 6 transistors for one bit). The off-chip SDRAM is used as main memory. It has much larger capacity up to gigabytes, since only one transistor and one capacitor are needed for one bit, thus consuming less area. However, SDRAM is slower than SRAM and has to be periodically refreshed to prevent data loss, as the capacitor suffers from leakage. The SDRAM is shared by other resources to read or write data via a memory controller integrated

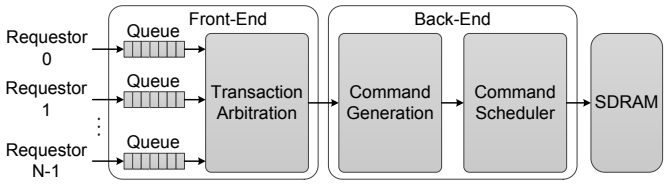


Figure 1.2: An abstracted view on a typical memory controller.

in the SoC. The on-chip interfaces for I/O devices manage video streams or data from Ethernet network. These on-chip resources are interconnected by a communication infrastructure, such as a bus or a *network-on-chip (NoC)* [33].

The hardware resources in a platform allow an application to be parallelized by running tasks on multiple processors simultaneously [94, 99]. Multiple applications can be deployed in the platform at the same time by sharing the resources. As a result, complex interferences between applications rise due to shared resources in the platform. When one application accesses a resource, other applications requiring the same resource have to wait. This impacts their ability to satisfy the timing constraints. In particular, SDRAM is a very commonly shared resource by the requestors, such as cores, hardware accelerators, DMAs, which generate diverse memory traffic. This makes the design and analysis of memory controllers challenging.

Next, we will focus on the memory subsystem and introduce how a memory controller serves transactions generated by different requestors.

1.1.3 *Main Memory Subsystem*

The memory subsystem is typically hierarchically organized in a SoC. The on-chip caches (e.g., L1/L2/L3 caches) are closer to the processor, such that it operates without suffering the long latency of reading data from the main memory, i.e., off-chip SDRAM. When the required data is not stored in the cache, e.g., a last-level cache miss, the processor generates a transaction (i.e., memory request) to read data from the SDRAM via the memory controller. When data must be stored, a write transaction is generated by the processor and sent to the memory controller, which forwards the data to the SDRAM. Similarly, other processing elements (e.g., hardware accelerators) may use DMA or specific circuit logic to manage their memory transactions and read/write data from/into the SDRAM via the memory controller.

SDRAM is a very popular volatile memory and is used for temporary data storage. It is accessed via the on-chip memory controller, which is typically partitioned into a *front-end* and a *back-end*, as shown in Figure 1.2. The front-end receives transactions from different requestors, and decides in which order to serve them. Its architecture is composed of *queues* for transactions per requestor and an arbiter to schedule each transaction to the back-end. The transaction is then executed by sending instructions (i.e., memory

commands) to the SDRAM, such that internal actions are triggered, e.g., reading or writing. In essence, the back-end translates the transaction into commands with the *command generator*, as presented in Figure 1.2. Once commands are generated, they are scheduled for execution in the SDRAM by a *command scheduler*, also shown in Figure 1.2. Note that commands are scheduled through a command bus to the SDRAM, where only one command can be transferred per clock cycle. Although the front-end and back-end are conceptually separate, interaction exists between them. For example, the front-end arbitration between requestors can be triggered by the back-end when a specific command is scheduled. Moreover, depending on the implementation, the transaction-level arbitration in the front-end can be combined with the command-level scheduling in the back-end.

The external SDRAM is typically viewed as a black-box to simplify its internal complexities [21, 29, 91, 109]. However, SDRAM is structured with multiple banks (i.e., memory arrays), which are controlled by memory commands. Commands can be executed on multiple banks simultaneously. This results in so-called bank parallelism [59], which supports the pipelining between transactions by executing their commands corresponding to different banks at the same time. The command scheduling is also complex. The reasons include: 1) it has to respect an internal *finite-state machine* that specifies the valid orders of executing commands, and 2) commands are executed subject to the *timing constraints*, such that the SDRAM can work properly. 3) When multiple commands are available for execution, e.g., all time constraints are satisfied for them, the scheduler has to choose one of them and sends to the SDRAM via the command bus. This is called a *collision* between these executable commands. The occurrence of collisions is unpredictable, because it is hard to know when the timing constraints are satisfied for multiple commands at the same time. Therefore, it is hard to predict the scheduling of a command, and thus bounding the memory performance is a challenging problem.

1.2 PROBLEM STATEMENT

This section discusses the two main problems solved in this thesis. The first problem is *how to design* memory controllers to efficiently deal with the diverse memory traffic in a heterogeneous multi-core hardware platform. The following problem is *how to analyze* the timing behavior of the memory controller and provide guaranteed performance in terms of worst-case response time and bandwidth. These worst-case results can be further integrated into system-level analysis of the application. For example, the worst-case response time for the SDRAM can be integrated as the cache miss penalty into the worst-case execution time estimation tools of applications [11]. In addition, they can be also integrated into a system-level analysis using dataflow formalism [80]. Although this is an important topic, the integration of worst-case results into a high-level analysis of real-time applications is outside the scope of this thesis.

1.2.1 *Problem I: Real-Time Memory Controller Design*

In modern multi-core systems, we see two relevant trends. 1) Both real-time and non-real-time applications are deployed at the same time in the same SoC [2, 16]. As discussed in Section 1.1.1, the former require guaranteed performance, while the latter should be given good average performance to feel responsive. 2) An increasing number of heterogeneous hardware resources are integrated in a system. The external SDRAM is shared by all resources, which result in diverse memory traffic in terms of arbitrarily-mixed read and write transactions with variable sizes and different physical addresses, as discussed in Section 1.1.2. These trends pose the following requirements on a memory controller: *it has to efficiently deal with the diverse traffic, while being analyzable to bound its performance, such that the requirements of real-time applications can be satisfied. Moreover, the memory controller should also give good average performance to non-real-time applications.*

A memory controller faces diverse memory traffic from different requestors executing tasks of real-time and/or non-real-time applications. The front-end of the memory controller needs an arbiter to schedule transactions from different requestors. However, *it is hard to choose a proper arbiter*. First, requestors have different requirements in terms of worst-case response time and/or bandwidth because of the applications being executed. The arbiter has to distinguish these differences between requestors. As a result, the widely used *round-robin (RR)* arbiter [85] is not always applicable, since requestors are served in a cycle, and each of them is given an equal opportunity to access the memory. A TDM arbiter [37] can allocate different number of time slots to requestors based on their requirements. However, a requestor needs many slots to satisfy its tight response time requirements, while the allocated bandwidth is wasted if the requestor has a low bandwidth requirement [77]. This problem can be overcome by fixed-priority based arbiters by giving different priorities to requestors according to their response time requirements [55, 56]. However, this may result in starvation when a requestor is always overtaken by a prioritized requestor.

The back-end of a memory controller executes transactions by generating and scheduling commands to SDRAM subject to the timing constraints. There exists state-of-the-art real-time memory controllers [3, 25, 39, 88], which use pre-computed static command schedules to execute transactions, such that their analyses are easy. Intuitively, each transaction is executed by picking up commands from the static schedules and sequentially sending them to the SDRAM. Since the schedules are statically designed, they only support a fixed transaction size and cannot exploit the run-time state of the SDRAM. However, transactions in the diverse traffic need to be *dynamically* executed, since they have variable sizes. Moreover, the run-time state of the SDRAM can be exploited to achieve good performance. We can imagine that different numbers of commands are needed by the variable-sized transactions, since a read/write command executed by SDRAM triggers a data burst of fixed size. Moreover, other commands are also

needed to manage the banks of SDRAM. Therefore, *the question is what kind of commands and how many of them should be generated for each transaction.* This question relies on the mechanisms used by the back-end. For example, a memory transaction can read/write data bursts from/into multiple banks rather than a single one, resulting in parallel accesses to different banks.

The scheduling of commands has to enable correct execution of commands based on the internal finite-state machine of SDRAM, while satisfying their timing constraints. It is difficult for the scheduler to make a decision, because 1) the execution of a command can be fast or slow depending on the relevant timing constraints and the current state of the SDRAM. 2) The execution of the current command influences future commands, resulting in command-level interferences. 3) The interferences between transactions are complex, because their commands to different banks can be scheduled in a pipelined manner. Therefore, it is challenging to design an efficient memory controller to deal with the diverse memory traffic. Finally, the memory controller must be analyzable, such that the worst-case response time and bandwidth can be derived. The analysis challenges will be discussed in the next section.

1.2.2 *Problem II: Real-Time Memory Controller Analysis*

This thesis focuses on analyzing the worst-case response time and bandwidth for the SDRAM. However, the main difficulty is the complex interferences between requestors, transactions, and commands. The response time of a transaction in the memory controller starts when it arrives at the front-end. Then it may experience a delay caused by other requestors, whose transactions may be executed first. The interference between requestors is highly dependent on the arbitration mechanism used in the front-end of the memory controller. It can be RR, TDM, or priority-based arbitrations, such as *credit-controlled static-priority arbitration (CCSP)* [5] and *frame-based static priority (FBSP)* [6]. When the transaction is scheduled and is sent to the back-end, it is executed in pipelining with the previous transaction, resulting in interferences between transactions. Finally, the data transmission is triggered when the read/write commands are executed by SDRAM. Hence, it is hard to analyze for how long time a transaction is executed. Within this time, a fixed amount of data corresponding to the transaction size is transferred. It is even more difficult to extend this analysis to a sequence of arbitrary transactions corresponding to a larger data volume. As a result, the long-term bandwidth measured by the execution times of transactions divided by the transferred data is more difficult to obtain.

1.3 THESIS CONTRIBUTIONS

This section introduces our solutions to solve the two critical issues of design and analysis of real-time memory controllers.

1. We design a **d**ynamically-scheduled **m**emory **c**ontroller, which efficiently deals with the diverse memory traffic at **run**-time and offers guarantees as well as good average performance. We refer to this memory controller as *Run-DMC* and it is introduced in Chapter 3.

2. To analyze the worst-case response time and bandwidth of Run-DMC, three analysis approaches are proposed.
   - A formal analysis approach uses a mathematical model to compute the time, at which each command is scheduled by Run-DMC. To address the complex interferences between transactions, it applies two simplifying assumptions to provide the worst-case initial bank state for an arbitrary transaction. Then the bounds on the worst-case response time and bandwidth are computed, and they are formally proved to be conservative. This formal analysis approach is given in Chapter 4.
   - A dataflow model is proposed to provide better bounds in an easier way than the formal analysis approach. It naturally captures the dependencies of scheduling commands by Run-DMC, and an existing analysis tool is used to automatically derive the bound on worst-case bandwidth. This approach eliminates one of the assumptions used in the formal analysis approach, resulting in a better bound. Moreover, the bound is easier to obtain, since it does not rely on complex manual proofs used by the previous approach. This approach is presented in Chapter 5.
   - The third approach continues modeling Run-DMC in Chapter 6, where a timed automata model is proposed to accurately describe the timing behavior of Run-DMC without any simplifying assumptions. The bounds on both worst-case response time and bandwidth are derived using model checking with an existing tool. This approach performs equally well as or better than the previous approaches to derive the bounds.

3. All the three analysis approaches are used to analyze the same memory controller (i.e., Run-DMC). This allows us to investigate their strengths and weaknesses in Chapter 7.

4. Finally, the formal analysis approach has been implemented as an open-source tool *RTMemController* [70] to evaluate both the average-case and worst-case performance of Run-DMC. Moreover, the TA model is also publicly available on-line [73].

Figure 1.3 shows an overview of the development of Run-DMC, including its implementation, simulation, validation, and verification, by using different models and tools. We proceed by explaining each contribution and Figure 1.3 in more detail in the following sections.

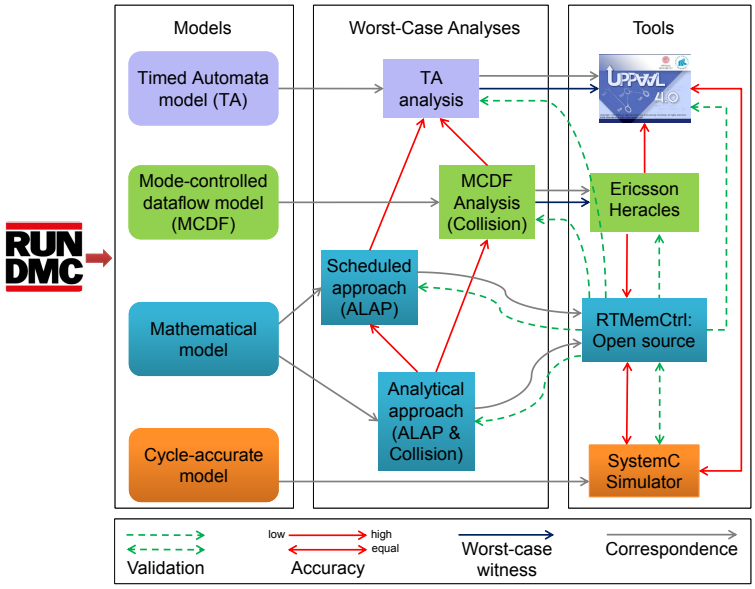


Figure 1.3: The development of our dynamically-scheduled memory controller Run-DMC.

1.3.1 *Run-DMC: a Dynamically-Scheduled Real-Time Memory Controller*

To efficiently deal with the diverse memory traffic, we design a memory controller to execute transactions with variable sizes by dynamically scheduling commands to the SDRAM according to the run-time SDRAM state and timing constraints. Its architecture is composed of a front-end and a back-end. The front-end uses a novel TDM arbiter to serve requestors with variable transaction sizes. The TDM slots hence have variable lengths. Moreover, it is flexible to meet the requirements of requestors by allocating different number of time slots to each requestor, while being easy to analyze. The two novelties to reduce the worst-case response time are that 1) idle slots (i.e., no transactions from the requestor) are skipped rather than being reallocated. This reduces the interference delay between requestors, since the requestor receiving the slot may have larger transaction sizes and consume more time. 2) Requestors are served in descending order of their transaction sizes. The reason is that transactions pipeline better following a larger transaction than a smaller one.

To avoid complexity while still being efficient, the back-end dynamically generates commands based on the transaction type (i.e., either read or write), size, and physical address. It schedules commands based on an algorithm that uses a *first-come first-serve (FCFS)* scheme to execute transactions in-order, while it supports pipelining between successive transactions. Moreover, scheduling collisions occurring when multiple commands are executable at the same time are avoided by prioritizing the read/write

commands over others. The reason is that the read/write commands trigger data transmission, and giving higher priorities to them intuitively provides good performance. The proposed memory controller will be introduced in Chapter 3. It has been implemented as a cycle-accurate SystemC model, as shown in Figure 1.3. It is the basis of validating the analysis models of Run-DMC.

1.3.2 *Formal Analysis of Run-DMC*

As discussed in Section 1.2.2, the analysis of a memory controller that pipelines commands across transactions is difficult because of the complex interferences between requestors, transactions, and commands. We propose a formal analysis approach to overcome this problem by partitioning it into two parts, where 1) interferences between requestors are analyzed in the front-end of Run-DMC that uses our novel TDM arbiter, and 2) the analysis of the back-end covers the interferences between commands. In particular, the back-end analysis bounds the maximum length of each time slot used by the TDM arbiter. The formal analysis approach will be presented in Chapter 4.

To carry out the worst-case analysis, we firstly propose a formalization to accurately capture the scheduling times of commands. This formalization has been implemented as an open-source C++ tool, named *RTMemController* [70]. It is validated by the SystemC simulator, ensuring accurate command scheduling times (see Figure 1.3). Therefore, the timing behavior of *RTMemController* is equivalent to the SystemC model of the memory controller. Due to the equivalence, *RTMemController* will be used to validate the other analysis models of Run-DMC.

Based on the formalization, the worst-case execution time experienced by a transaction in the back-end is analyzed. The difficulty is that the execution of the current transaction depends on the previously scheduled commands corresponding to earlier transactions. We employ two conservative assumptions to eliminate the impact of these earlier transactions. The first assumption is that the commands of the previous transactions were scheduled *as-late-as-possible (ALAP)*, resulting in the maximum possible scheduling times of the previous commands. This provides the worst-case initial SDRAM state for the current transaction. Moreover, Run-DMC prioritizes read/write commands over others. When there is a collision between them, these commands with lower priority are delayed. Since the collisions are unpredictable, the analysis has to conservatively assume that they always occur for the low-priority commands. Then the formalization can compute the maximum scheduling times of commands for the current transaction, resulting in the worst-case execution time in the back-end. This is the so-called *analytical approach*, as shown in Figure 1.3. To achieve lower worst-case execution time, *RTMemController* is used to actually detect the command collisions for the current transaction, while the scheduling times of the previous commands are still given using ALAP scheduling. As a result, this *scheduled approach* hence outperforms the *analytical approach*. Both of them are included in *RTMemController*. Finally, we compute the worst-case band-

width based on the worst-case execution time, with which a fixed amount of data is transferred. In addition, the worst-case response time in the front-end is calculated according to the static TDM slot allocation, where the period of each slot equals to the relevant worst-case execution time.

1.3.3 *Dataflow Modeling of Run-DMC*

Run-DMC schedules commands subject to the SDRAM finite-state machine and the timing constraints, which result in dependencies between commands. These dependencies have already been captured by the formalization. However, its worst-case analysis provides pessimistic results because of the conservative assumptions. Moreover, the analysis is based on manual proofs that are very time-consuming to make. As a result, it is difficult to extend the formal analysis approach to different memory controllers or SDRAM devices. Dataflow models naturally capture dependencies and existing tools can be used to derive the worst-case results by automatically analyzing the model. Therefore, *we switch the effort from formal analysis to modeling the timing behavior of the memory controller and derive the worst-case results by analyzing the model with existing techniques and tools.*

We propose a *mode-controlled dataflow (MCDF)* model that captures the dependencies of dynamic command scheduling for Run-DMC. Existing dataflow analysis techniques implemented in the Ericsson's Heracles tool [79] are used to derive the worst-case results. The MCDF model will be introduced in Chapter 5. In the MCDF model, memory commands are represented by actors. The SDRAM timing constraints are captured by the execution time of actors, while the dependencies are described by edges between actors. A mode corresponds to a subset of the dataflow graph and the dynamism in the memory traffic in terms of different transactions can be captured by dynamically selecting different modes.

The MCDF model is executable and supports simulation of the memory controller. We use this to validate the MCDF model with the open-source tool *RTMemController*, which provides identical scheduling times of commands for the same transaction traces. The analysis of the MCDF model provides the minimum throughput of executing transactions after determining the critical sequence of transactions. The minimum throughput can be converted into the worst-case/minimum bandwidth (WCBW). The WCBW is determined by analyzing sequences of transactions rather than a single transaction. The former supports exploiting the pipelining between transactions, while the latter used by the formal analysis approach does not. Therefore, the MCDF model can provide better WCBW results in a more convenient way using existing analysis tool. The experimental results demonstrate that the MCDF model outperforms the formal analysis approach. However, similarly to the analytical approach in the formal analysis, the MCDF model conservatively assumes command collisions to avoid non-deterministic variations in the dataflow model. It uses the actual scheduling times of command rather

than the ALAP scheduling, as shown in Figure 1.3. Moreover, the Heracles tool is only capable of proving the WCBW, as analysis of worst-case response time is not supported.

1.3.4 *Timed Automata Modeling of Run-DMC*

To overcome the drawbacks of the previous MCDF model, we continue modeling Run-DMC using a different model, which is based on *timed automata (TA)* [15]. The open-source tool Uppaal [13] is used to carry out the modeling, simulation, and verification. In particular, the worst-case response time and bandwidth can be automatically obtained by exhaustively exploring the state space via model checking with Uppaal.

We have developed a modular TA model of our dynamically-scheduled memory controller without any simplifying assumptions, as shown in Figure 1.3. The TA model will be given in Chapter 6. The timing behavior of each component (e.g., TDM arbiter, command generator, command scheduling) is accurately described by a TA. The accuracy has been validated by simulating the TA model with given transaction traces using Uppaal, and also feeding these traces to the open-source *RTMemController*. Identical scheduling times of commands are obtained in these two ways, which suggests that the TA model accurately captures the timing behavior of Run-DMC.

Model checking exhaustively explores the state space, which automatically includes the complexities of interferences between requestors, transactions, or commands, whether they occur in the front-end or back-end. Therefore, the proposed TA model is capable of 1) providing tight worst-case results, which are validated easily. The reason is that Uppaal provides a diagnostic transaction trace (i.e., witness) for each result. The experimental results demonstrate that the TA model gives better worst-case results than these previous approaches. 2) It can be easy to extend to other memory controllers, since the TA of the common components can be reused.

2

BACKGROUND & TERMINOLOGY

This chapter sets the stage on which the following chapters will play out. The SDRAM is introduced in Section 2.1, including its architecture and memory commands as well as the timing constraints between commands. SDRAM is connected to the memory controller with the command, address, and data buses. It executes commands scheduled by the memory controller, which coordinates between the memory requestors (e.g., processors, GPU, DMA, hardware accelerators) and the SDRAM. Section 2.2 will introduce the general functionalities of a memory controller, which are partitioned into a front-end and a back-end. The front-end receives transactions from different requestors, which are served by an arbiter. When a transaction is sent to the back-end, a number of commands are generated, and are sequentially scheduled to the SDRAM via the command bus without violating any timing constraints. Finally, to formalize the command scheduling for an individual transaction, the relevant timings are defined in Section 2.3, followed by the metrics to evaluate the performance of the memory controller. The metrics include the execution time in the back-end, the response time, and the bandwidth provided by the memory controller.

2.1 SDRAM ARCHITECTURE AND OPERATION

2.1.1 *SDRAM Architecture*

The off-chip SDRAM is a very popular volatile memory and used as temporary data storage. An SDRAM chip comprises a set of banks, e.g., a contemporary DDR3 SDRAM chip typically has 8 banks. A bank contains a memory array consisting of elements arranged in rows and columns [51], as shown in Figure 2.1. The banks can work in parallel. However, they share the same interface consisting of command, address, and data buses. As a result, only one command or data word can be sent to one bank at a time. The command bus transfers a single command per clock cycle, while the data bus transfers two data words per cycle for a *double data rate (DDR)* memory. Moreover, the data bus is bidirectional and used to both read and write data. The address bus transfers the physical address in terms of bank, row, and column for each command. To issue a command, several timing constraints have to be satisfied, as specified by the JEDEC

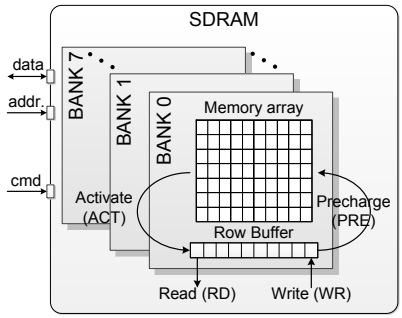


Figure 2.1: SDRAM architecture

DDR3 standard [53]. Timing constraints guarantee that the SDRAM can properly execute the received commands. Note that although this thesis focuses on DDR3 SDRAMs, it requires only minor adaptations to work with other types of SDRAMs, such as DDRx, LPDDRx and Wide I/O.

2.1.2 *SDRAM Commands and Timing Constraints*

An SDRAM executes commands, which mainly include *Activate (ACT)*, *read (RD)*, *write (WR)*, *precharge (PRE)*, *refresh (REF)*, and *no operation (NOP)*. The execution of commands has to satisfy SDRAM timing constraints, as summarized in Table 2.1. The commands work as follows:

- An *ACT* command opens a row in a bank, i.e., moves the data in the row into the row buffer of the bank (see Figure 2.1). It makes the data available for subsequent *RD* or *WR* command(s). This activation takes *tRCD* cycles, as indicated in Table 2.1. The *ACT* command is accompanied by the addresses of the required bank and row.

- When data is available in the row buffer, a *RD* or *WR* command triggers reading or writing a burst of data from a range of columns in the open row. The *burst length (BL)* is 8 words for DDR3 SDRAMs. The first bit of the required data appears on the data bus *tRL* cycles after issuing a *RD* command, while it is after *tWL* cycles for a *WR* command. The DDR3 SDRAM transfers two words per cycle. As a result, a data burst occupies the data bus for $BL/2$ cycles. When more data bursts are needed, a number of *RD* or *WR* commands can be issued to SDRAM. Each *RD* or *WR* command is accompanied by the address of the first column of the data burst. The address is aligned with *BL*, i.e., address (in words) modulo $BL = 0$.

- Once reading/writing is finished, a PRE command is issued to close the open row, i.e., the data in the row buffer is stored back to the original row in the bank. Subsequently, a different row in the bank may be opened. The timing constraints to

issue a *PRE* command include *tRAS* after an *ACT* command, *tRTP* after a *RD* command, and *tWTP* and *tWR* after a *WR* command. They are shown in Table 2.1. A *PRE* command can be either explicitly issued via the command bus or by adding an auto-precharge flag to the previous *RD* or *WR* command, such that precharging is automatically triggered when all timing constraints are satisfied. The latter is called an auto-precharge policy. The precharging duration is *tRP* cycles.

- Since SDRAM is volatile, it has to be periodically refreshed every *tREFI* cycles to retain the data. A *REF* command is issued after all open rows are closed. The period of refreshing (*tRFC*) depends on the capacity of the SDRAM and the operating temperature. Table 2.1 assumes a capacity of 2 Gb and a fixed operating temperature range, i.e., 0 °C to 85 °C [53].
- Finally, a NOP command is issued when waiting until timing constraints are satisfied, or when no commands have to be executed. NOP does nothing.

Table 2.1: Timing constraints (TC) for DDR3-1600G SDRAM [53].

TC	*Description*	*Cycles*
tCK	Clock period	1
tRCD	Minimum time between *ACT* and *RD* or *WR* commands to the same bank	8
tRRD	Minimum time between *ACT* commands to different banks	6
tRAS	Minimum time between *ACT* and *PRE* commands to the same bank	28
tFAW	Time window in which at most four banks may be activated	32
tCCD	Minimum time between two *RD* or two *WR* commands	4
tWL	Write latency. Time after a *WR* command until first data is available on the bus	8
tRL	Read latency. Time after a *RD* command until first data is available on the bus	8
tRTP	Minimum time between a *RD* and a *PRE* command to the same bank	6
tRP	Precharge duration time	8
tWTR	Internal *WR* command to *RD* command delay	6
tWR	Write recovery time. Minimum time after the last data word has been written to a bank until a precharge may be issued	12
tRFC	Refresh period time	128
tREFI	Refresh interval	6240

2.1.3 *Command Scheduling*

An SDRAM operates on commands, which can be scheduled in various ways by the memory controller. This section firstly provides a simplified state diagram, which gives

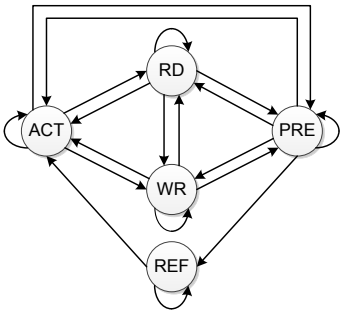


Figure 2.2: Simplified state diagram of scheduling commands.

an overview of the possible orders of scheduling commands to a single bank or multiple banks, respectively. Then, we introduce several concepts for command scheduling, including 1) page policy, 2) command pipelining, and 3) bank interleaving.

2.1.3.1 *State Diagram of Scheduling Commands*

Following the descriptions of how commands work in the previous section, we can straightforwardly draw Figure 2.2 that presents a state diagram to show the valid orders of scheduling commands. To access a single bank, it implies the following situations:

- SDRAM can read/write one or more data bursts only if the row is open. This requires an *ACT* command followed by one or multiple *RD/WR* commands. It is captured by the transition from the ACT state to the RD/WR state in Figure 2.2. In addition, RD/WR state has a self-transition, which means multiple *RD/WR* commands can be issued in a sequence.
- Precharging is performed after reading or writing is complete, resulting in the transition from the RD or WR state to the PRE state. The transition from PRE to ACT in Figure 2.2 shows that opening a new row in the bank is after closing the previous row.
- Finally, the SDRAM can be refreshed when the row is closed, and thus there is a transition from PRE to REF in Figure 2.2. Since SDRAM is allowed to execute a group of refreshes, the REF state consists of a self-transition. It implies that a new *REF* command can be scheduled after another one subject to the timing constraints.

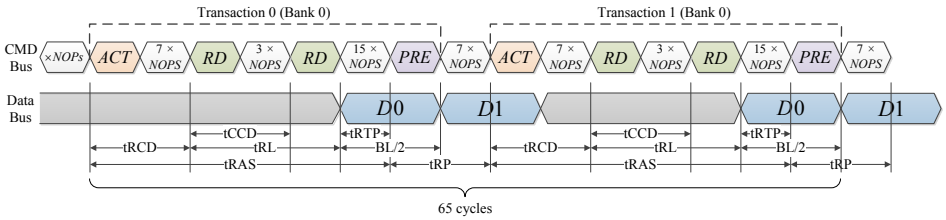
In addition to the transitions of scheduling commands to a single bank, additional transitions in Figure 2.2 illustrate the possible orders of scheduling commands to different banks. For example, two *ACT* commands can be scheduled to two different banks successively, since banks work in parallel. This is depicted by the self-transition of the ACT state in Figure 2.2.

2.1.3.2 *Page Policies*

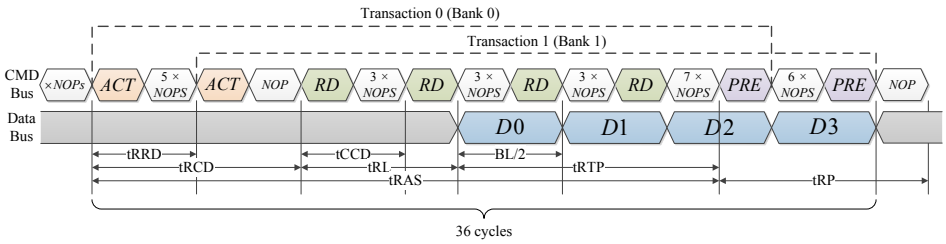
The *ACT* and *PRE* commands are scheduled according to the *page policy* of the memory controller. With an *open-page policy*, a row is left open when a *RD* or *WR* command has completed. In contrast, with a *close-page policy*, an open row is closed as soon as possible after a *RD* or *WR* command has finished. When an open row is required by a transaction, a so-called *row-hit* or *page-hit* occurs, while a *row-miss* or *page-miss* is caused when a closed row is needed. With a *page-hit*, only *RD* and *WR* commands are scheduled by the memory controller that uses an open-page policy. When the transaction encounters a *page-miss*, the memory controller with an open-page policy has to schedule a *PRE* command to close the current row, followed by an *ACT* command to open the required row. Then, *RD* or *WR* commands can be scheduled. When a close-page policy is used, the row of a transaction is always closed. As a result, the memory controller has to schedule an *ACT* command, followed by a number of *RD* or *WR* commands, and finally a *PRE* command. Transactions resulting in a *page-hit* benefit from an open-page policy, while a close-page policy is more efficient for *page-misses*. It only needs to open the required row. In the worst-case, most analyses of real-time memory controllers have to assume transactions always experience *page-misses*. The reason is that it is hard to statically show they will be hits [108]. To achieve better worst-case performance, we use a close-page policy in this thesis.

2.1.3.3 *Command Scheduling Pipelining*

SDRAM banks can work in parallel, where one bank is activating or precharging while another bank is simultaneously reading or writing data. This is the so-called *bank parallelism*. As a result, *ACT* and *PRE* commands for one bank can be pipelined with the *RD* or *WR* commands to another bank. To illustrate the benefit of pipelining, Figure 2.3 shows the command schedules for two transactions with or without pipelining, where a DDR3-1600G SDRAM is taken as an example. The timing constraints are given in Table 2.1. Both transactions need to read two data bursts. Figure 2.3(a) presents the case where the two transactions access the same bank, e.g., Bank 0. With a close-page policy, the *ACT* command for the second transaction has to be scheduled after the *PRE* command of the first transaction has completed, i.e., a new row in Bank 0 can be opened after the current row is closed. This takes a long time, i.e., 65 cycles to schedule all the commands. If the second transaction needs a different bank, e.g., Bank 1, Figure 2.3(b) shows the pipelined command schedule. The *ACT* command for the second transaction is pipelined with the commands of the first transaction. As a result, the command schedule for these two transactions is shorter, requiring only 36 cycles. However, due to the command scheduling pipelining, a *collision* may occur on the command bus when timing constraints are satisfied for more than one command at the same time. The reason is that the command bus only transfers one command per cycle. The memory controller needs to resolve command collisions.



(a) No pipelining between transactions (65 cycles)



(b) Pipelining between transactions (36 cycles)

Figure 2.3: Command scheduling: no pipelining vs. pipelining

2.1.3.4 *Bank Interleaving*

The previous section discussed pipelining between transactions. This section introduces the pipelining *within* a transaction, which is interleaved over multiple banks while several data bursts are transferred per bank. Two parameters have been introduced to flexibly exploit the command pipelining for a transaction [7, 36]:

1. *Bank interleaving (BI)*: the number of consecutive banks that are accessed to read or write the required data of a single transaction;
2. *Burst count (BC)*: the number of continuous data bursts per bank for a single transaction.

As a result, the data size of a transaction equals $BI \times BC \times BL$ words, where BL is the burst length. The width of a word is determined by the width of the data bus.

2.2 REAL-TIME MEMORY CONTROLLERS

In modern multi-core platforms, memory requestors, such as processors, DMAs, and hardware accelerators, access the off-chip SDRAM via a memory controller. A general real-time memory controller architecture is shown in Figure 2.4. Its front-end and back-end will be introduced in Section 2.2.1 and Section 2.2.2, respectively.

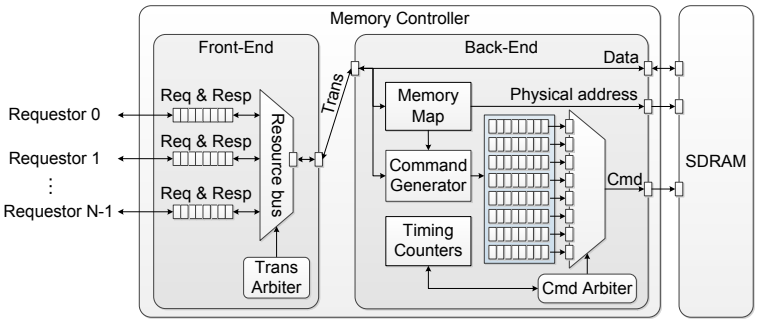


Figure 2.4: The general architecture of a memory controller.

2.2.1 *Memory Controller Front-End*

The front-end receives transactions from the requestors through a bus or a NoC. In real-time systems, the interconnect must offer performance guarantees, e.g., as done by the dAElite NoC [96]. The received transactions are buffered in a queue per requestor, as shown in Figure 2.4. One of these transactions is then selected by the arbiter according to an arbitration policy, such as TDM [37], RR [85] or CCSP [5], and is sent to the back-end. Note that a write transaction is sent to the back-end without transferring its associated data, which still stays in the buffer in the front-end. The data will be transferred to the SDRAM via the data bus based on scheduling the *WR* in the back-end. Figure 2.4 shows a general front-end architecture, which is suitable for contemporary multi-core platforms. For many-core platforms with a very large number of requestors, techniques such as coupling NoC and memory controller [23], distributed arbitration [32] and multiple memory channels [30] can be used. However, they are outside the scope of this thesis.

2.2.2 *Memory Controller Back-End*

The back-end receives each transaction sent by the front-end and *translates the transaction into a sequence of commands, which are scheduled to the SDRAM via the command bus.* To achieve this goal, the logical address of a transaction is translated into a physical address in terms of bank, row, and column according to the memory map, which determines the location of data in the SDRAM. The memory map also specifies how a transaction is split over the memory banks and thus the degree of bank parallelism used when serving it. This is captured by the two parameters: *BI* and *BC*, as previously described.

BI and *BC* are limited to powers of two for efficient address decoding. Thus, the *BI* consecutive banks of a transaction must start on a bank aligned with *BI* [39]. Memory mapping with BI and BC is a trade-off between execution time, bandwidth, and power consumption, as shown in [36, 39, 40]. This thesis focuses on the lowest possible execu-

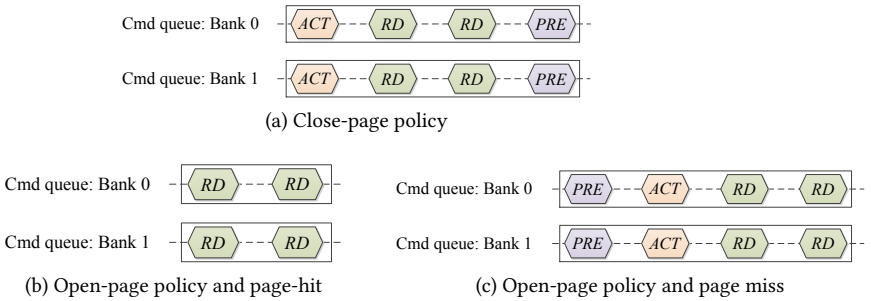


Figure 2.5: Command generation for a read transaction with *BI* = 2 and *BC* = 2.

tion time. As a result, the largest possible *BI* is chosen to increase the exploited bank parallelism. However, interleaving over all 8 banks (i.e., *BI* = 8) of a DDR3 SDRAM cannot be helpful to achieve the lowest execution time because of the *tFAW* timing constraint (see Table 2.1) [39]. Moreover, this configuration increases the power consumption, since there are more banks working in parallel. In fact, various memory mapping strategies can be supported by specifying different *BI* and *BC* combinations. For example, a small *BI* and a large *BC* support a block-oriented memory mapping that increases the row hit rate by mapping consecutive data bursts to the same row of a bank [43]. In contrast, stripe-oriented mapping with a relatively large *BI* and a small *BC* allocates data bursts to different banks and exploits bank parallelism [74].

The command generator translates a transaction into a sequence of commands that are stored in command queues for each bank. The required commands of a transaction depend on *BI*, *BC*, and also the page-policy, i.e., either open-page or close-page, as previously introduced in Section 2.1.3. A transaction is interleaved over *BI* number of banks, each with *BC* number of data bursts. With a close-page policy, it requires an *ACT* command followed by *BC* number of *RD* or *WR* commands and a *PRE* command at the end for each bank. Figure 2.5(a) shows the generated commands for a read transaction to access each bank, where the starting bank is assumed to be Bank 0. Note that if an auto-precharge policy is used, no explicit *PRE* commands are needed, but instead a flag is attached to the last *RD* or *WR* command of a bank to trigger the precharging when timing constraints are satisfied. With an open-page policy, Figure 2.5(b) and Figure 2.5(c) present the required commands, when a transaction experiences page-hit and page-miss, respectively.

Finally, the command arbiter schedules commands to SDRAM, subject to the timing constraints, and data transmission on the data bus automatically follows a *RD* or *WR* command after a fixed time. The data is transferred directly between the buffer in the front-end and the SDRAM via the data bus. Consequently, the analysis of a real-time memory controller only needs to focus on scheduling commands for each transaction.

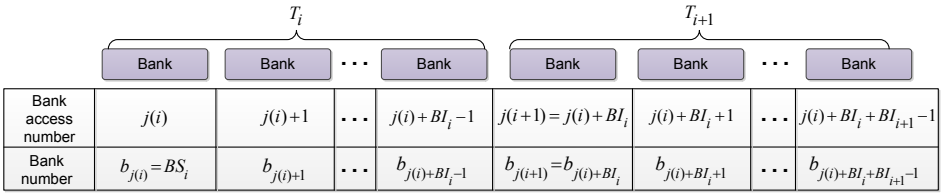


Figure 2.6: *Bank access number* and *bank number* for T_i and T_{i+1}.

2.3 ANALYSIS OF REAL-TIME MEMORY CONTROLLERS

This section introduces the terminology and definitions to formalize the execution of transactions by a memory controller. They will be used throughout this thesis.

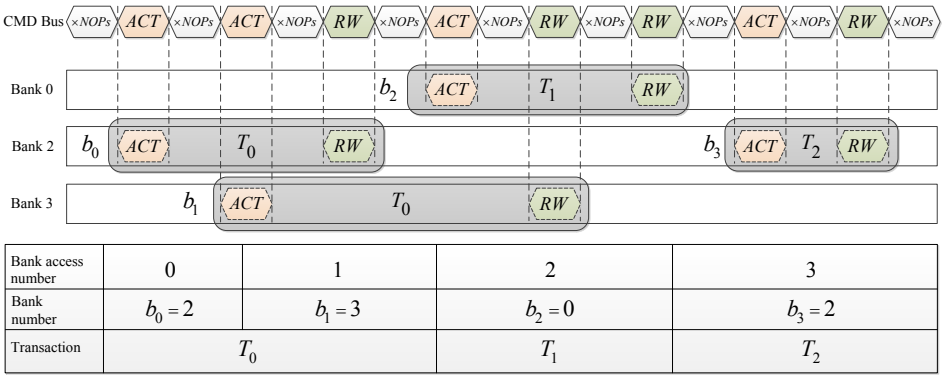
2.3.1 *Bank Accesses for Transactions*

A transaction is defined in Definition 1 based on its characteristics in terms of size, type, and the parameters (i.e., *BI* and *BC*) for memory mapping. An arbitrary transaction is denoted by T_i and it uses BI_i and BC_i. Since T_i interleaves over consecutive banks, we only need to know its *starting bank number (BS)* for the analysis, while the physical addresses of row and column can be ignored for analysis purposes. Note that i is the arrival number of the transaction in the back-end rather than in the front-end, since the analysis of the front-end in this thesis does not require the arrival number.

Definition 1 (Transaction)**.** *A transaction is defined as a tuple T_i = ($S(T_i)$, $Type(T_i)$, BI_i, BC_i, BS_i), where:*

1. *i represents the arrival number of the transaction in the back-end and $\forall i \geq 0$.*
2. *$S(T_i)$ is the size of T_i in bytes.*
3. *$Type(T_i)$ denotes the type of T_i and is either read or write.*
4. *BI_i is the number of banks that T_i interleaves over.*
5. *The number of the read or write bursts per bank for T_i is BC_i.*
6. *The starting bank number of T_i is denoted by BS_i.*

Each transaction accesses one or more banks. From the analysis perspective, we often only care about successive bank accesses, but not which transactions they belong to. For example, the first transaction T_0 has the bank access number from 0 to $BI_0 - 1$, and T_1 continues with the bank access number from BI_0 to $BI_0 + BI_1 - 1$, and so on. Generally, the $j^{th}(\forall j \geq 0)$ bank access uses bank b_j, which is the *bank number*. We assume the first bank access number for an arbitrary transaction T_i is j, which is a function of i as



Bank access number	0	1	2	3
Bank number	$b_0 = 2$	$b_1 = 3$	$b_2 = 0$	$b_3 = 2$
Transaction	T_0		T_1	T_2

Figure 2.7: An example of illustrating the bank accesses for transactions T_0, T_1, and T_2.

denoted by $j(i)$. It can be calculated with Eq. (2.1) based on the BI used by all previous transactions. Figure 2.6 illustrates the execution of two successive transactions T_i and T_{i+1} accessing BI_i and BI_{i+1} banks, respectively, where the bank access number and bank number are explicitly shown. *For legibility, we use j instead of j(i) and b_j instead of $b_{j(i)}$ throughout this thesis. Therefore, j should be syntactically replaced by j(i) and b_j by $b_{j(i)}$ everywhere.* Note that b_j is also the starting bank of T_i, thus $b_j = BS_i$.

$$\forall i \geq 0, j(i) = \sum_{k=0}^{i-1} BI_k \tag{2.1}$$

Using the numbering method of bank accesses, Figure 2.7 presents an example of executing three successive transactions T_0, T_1, and T_2 by sequentially scheduling commands through the command bus subject to the SDRAM timing constraints. Note that a *RD* or *WR* command is denoted by *RW* in Figure 2.7, where an auto-precharge policy is assumed and there are no explicit *PRE* commands. T_0 uses $BI_0 = 2$ and $BC_0 = 1$, while both T_1 and T_2 only need one data burst, leading to $BI_1 = BI_2 = 1$ and $BC_1 = BC_2 = 1$. T_0 starts with Bank 2, i.e., $BS_0 = 2$. T_1 and T_2 start with Bank 0 and Bank 2, respectively, and hence $BS_1 = 0$ and $BS_2 = 2$. T_0 is the first transaction executed in the memory controller back-end. Its bank access number starts with 0 (i.e., $j(0) = 0$) and increases for the following transactions. The corresponding bank number is Bank 2, resulting in $b_{j(0)} = b_0 = 2$. In the same way, $b_1 = 3$, $b_{j(1)} = b_2 = 0$, and $b_{j(2)} = b_3 = 2$.

Next, we formalize the general scheduling dependencies of commands to any two successive banks in Section 2.3.2. Then, the timings of commands and transactions are defined in Section 2.3.3, followed by definitions of the performance metrics (worst-case) response/execution time and bandwidth.

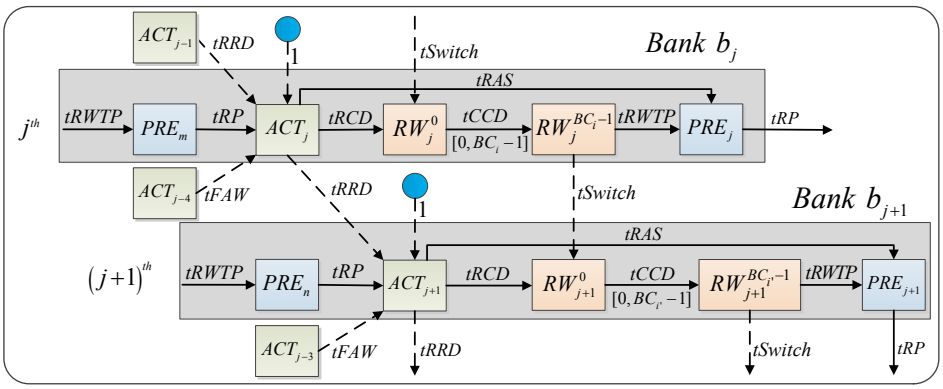


Figure 2.8: Command scheduling dependencies between any two successive bank accesses corresponding to T_i and $T_{i'}$.

2.3.2 *Command Scheduling Dependencies*

This section focuses on formalizing the scheduling of commands to banks. Dependencies exist between commands when they are scheduled to the SDRAM. For example, after an *ACT* command is scheduled to open a row in a bank, the memory controller has to wait at least *tRCD* cycles (see Table 2.1) to schedule the following *RD* or *WR* command. As a result, the scheduling of the *RD* or *WR* command depends on the *ACT* command. The dependencies are also affected by the command scheduling algorithm, which specifies the order of commands. The timing constraints can be classified into *inter-bank* and *intra-bank*. The former apply for scheduling commands to different banks, while the latter are used for the same bank. This section generally formalizes the scheduling dependencies of commands to two successively accessed banks.

Figure 2.8 illustrates the scheduling dependencies of commands to two banks b_j and b_{j+1} for one or two transactions. j and j+1 are the bank access numbers. It is possible to have $b_j = b_{j+1}$, representing that the same bank is successively accessed twice. The j^{th} bank access needs an *ACT* command followed by BC_i *RD* or *WR* (i.e., *RW*) commands and a *PRE* command at the end (or an auto-precharge flag). These commands are represented by ACT_j, RW_j^k ($k \in [0, BC_i - 1]$) , and PRE_j, respectively, where BC_i is the burst count of transaction T_i that bank access j belongs to. In particular, RW_j^k is the k^{th} *RW* command to the bank.

Figure 2.8 uses dotted and solid arrows to depict the command scheduling dependencies, which are caused by the inter- and intra-bank timing constraints, respectively. The scheduling of a command depends on previous commands, which are specified by the input arrows. The labels near the arrows denote the timing constraints, i.e., the number of cycles that the following command has to wait before it can be scheduled. For example, the timing constraints for scheduling an *ACT* command include *tRRD*, *tRP* and

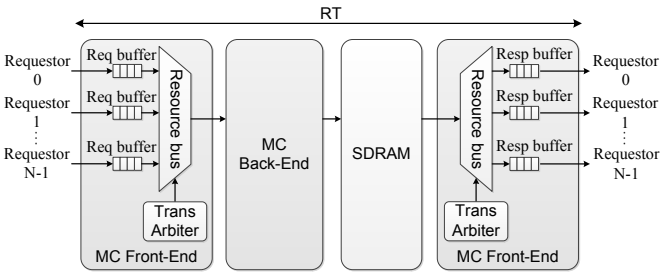
Figure 2.9: Dependencies between transactions.

tFAW, previously described in Table 2.1. Therefore, the *ACT* command (see Figure 2.8) has three input arrows. Moreover, a collision may occur on the command bus. To resolve the collision, only one command is scheduled while others have to be postponed and re-arbitrated in the next cycle. Figure 2.8 uses the solid circles to represent the collisions. As an example, the collisions delay *ACT* commands by one cycle. The scheduling of the first *RD* or *WR* command for a bank access has to satisfy the timing constraints *tRCD* and *tSwitch*. The following *RD* or *WR* commands to the same bank only need to satisfy the *tCCD* timing constraint. Note that *tSwitch* represents the timing constraints from *RD* to *WR* and vice versa, and it will be later defined by Eq. (3.2) based on the JEDEC-specified timing constraints [53]. Finally, an (auto-)precharge has to satisfy the timing constraints *tRAS* and *tRWTP*, where *tRAS* is given in Table 2.1 while *tRWTP* is later defined by Eq. (3.1). Note that refresh commands are not depicted because their impact on WCET can be easily analyzed, as presented in Section 4.5. Moreover, the effect of *REF* is small (approximately 3%) in terms of bandwidth or the WCET of an application, and it is not a main concern in this thesis.

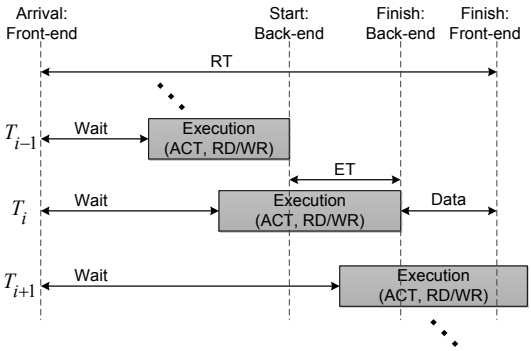
Due to the dependencies between commands and hence the bank accesses, there exist dependencies between transactions that the bank accesses belong to. As illustrated in Figure 2.9, the execution of the current transaction T_i relies on the previous transaction T_{i-1}. Intuitively, to evaluate the maximum time to execute T_i by scheduling commands, all the previous transactions have to be taken into account. Therefore, it is difficult to analyze the worst-case bounds of serving transactions, especially with the arbitrary read or write transactions with variable sizes and requiring different sets of SDRAM banks.

2.3.3 *Performance Metrics & Definitions*

To evaluate the performance of a memory controller, two metrics are used in this thesis: the *response time (RT)* of an individual transaction and the bandwidth. The response time measures the total time experienced by a transaction, while the bandwidth is the long-term rate of transferring data. Figure 2.10(a) provides the architecture view of the RT, where the front-end of a memory controller is split into transactions arriving in the request (i.e., req) buffer and the responses returning to the response (i.e., resp) buffer. The RT covers all the processes experienced by a transaction in the memory controller, including transaction arrival and scheduling in the front-end, commands scheduling in the back-end, executing commands in the SDRAM, and finally returning response



(a) Architecture view of response time



(b) Temporal view on execution time in the back-end and response time in the front-end for transaction T_i.

Figure 2.10: Terminology of performance metrics

to the front-end. A transaction experiences the execution time (ET) in the back-end. Figure 2.10(b) gives a temporal view on RT and ET of a transaction T_i, which is pipelined with other transactions. The unusual definition of ET is explained next.

The ET is the time spent by the memory controller back-end *exclusively* on behalf of the current transaction, though the commands (i.e., *ACT* or *PRE*) of other transaction(s) are pipelined. The purpose of defining ET in this way is that when adding the execution times of a sequence of pipelined transactions, we never count any cycle twice. This is helpful when computing the response time and bandwidth. The ET includes waiting for timing constraints to be satisfied (e.g. an *ACT* of transaction T_i may have to wait after commands of T_{i-1}), and executing *ACT* and *RW* commands. *PRE* commands are ignored because auto-precharge is used. Any *ACT* commands that can be pipelined with the previous transaction do not count towards the ET. Note that *RD* and *WR* commands of different transactions can never be pipelined, because they trigger data transmission on the shared data bus. Moreover, if the memory controller is idle when transaction T_i arrives, then all of its *ACT* commands *do* count toward its ET, because there is no previous transaction to overlap with, and to hide the time spent on executing them. As

illustrated in Figure 2.10(b), the execution of *ACT* commands of a transaction in the back-end may start before the previous transaction finishes.

With this definition of ET, the bandwidth is the transaction size divided by the ET, since a fixed amount of data is transferred. The RT of a transaction is the time from its arrival in the front-end until the response is returned, as shown in Figure 2.10(b). It covers the interference delay of executing other transactions and its own ET. In addition, the response of a read transaction to the front-end is finished when the last data word is returned through the data bus. This is later than the finishing of the execution in the back-end, i.e., scheduling the last *RD* or *WR* command. The *worst-case response time (WCRT)* is the maximum RT of a transaction, while the *worst-case bandwidth (WCBW)* is the minimum long-term data transmission rate guaranteed by the memory controller. To define these two metrics, we begin with the following definitions.

An arbitrary transaction T_i from a requestor arrives at the interface of a memory controller front-end and its arrival time is given by Definition 2. Then, it experiences interferences from other requestors. When the transaction is selected by the arbiter and sent to the back-end, as shown in Figure 2.4, its arrival time in the back-end is defined by Definition 3. In the back-end, T_i is executed by scheduling its commands to the BI_i consecutive banks, where each of them receives BC_i number of *RD* or *WR* commands.

Definition 2 (Arrival time of transaction T_i in the front-end)**.** *$t_a^{fe}(T_i)$ is defined as the time at which T_i has arrived at the request queue of the front-end. In case of a write transaction, all data must have arrived.*

Definition 3 (Arrival time of transaction T_i in the back-end)**.** *$t_a(T_i)$ is defined as the time at which T_i has arrived at the interface of the back-end. In case T_i is write, $t_a(T_i)$ does not relate to the data, which is stored in the front-end.*

The execution of T_i finishes when its last *RW* command is scheduled, which is denoted by $RW_{j+BI_i-1}^{BC_i-1}$. It represents the $(BC_i-1)^{th}$ *RW* command scheduled for the $(j+BI_i-1)^{th}$ bank access. The scheduling time of a command is defined by Definition 4. For the last *RW* command, its scheduling time is denoted by $t(RW_{j+BI_i\text{-}1}^{BC_i\text{-}1})$. The finishing time of T_i in the back-end is defined as the scheduling time of its last command, as given by Definition 5. *The back-end starts executing T_i either immediately when it arrives or after finishing the previous transaction $T_{i\text{-}1}$.* The starting time of T_i is given by Definition 6, where $t_f(T_{\text{-}1}) = -\infty$ indicating the initial transaction $T_{\text{-}1}$ finished a long time ago, leaving all banks closed and imposing no timing constraints on T_0. Finally, the execution time of T_i in the back-end is defined as the time between its starting time and its finishing time, as given by Definition 7. The *worst-case execution time (WCET)* is defined as the maximum execution time, as denoted by $\hat{t}_{ET}(T_i)$. It is worth noting that this ET does not cover the time of the data transmission for the purpose of giving a convenient analysis. RT does need to cover the data transmission, since it accounts for the total delay from arrival to finish of a transaction in the memory controller.

Definition 4 (Scheduling time of a command). *t(CMD) is defined as the time, at which the command CMD is put on the command bus of the SDRAM.*

Definition 5 (Finishing time of transaction T_i in the back-end).

$$\forall i \geq 0, t_f(T_i) = t(RW_{j+BI_i-1}^{BC_i-1})$$

Definition 6 (Starting time of transaction T_i in the back-end).

$$\forall i \geq 0, t_s(T_i) = max\{t_a(T_i), t_f(T_{i-1}) + 1\}, \text{ where } t_f(T_{-1}) = -\infty$$

Definition 7 (Execution time of transaction T_i in the back-end).

$$\forall i \geq 0, t_{ET}(T_i) = t_f(T_i) - t_s(T_i) + 1$$

Figure 2.11 shows an example of the command scheduling for three successive transactions T_{i-1}, T_i, and T_{i+1}, where $BI_{i-1} = BI_i = BI_{i+1} = 2$ and $BC_{i-1} = BC_i = BC_{i+1} = 1$. We also assume the starting bank of T_{i-1} and T_{i+1} is Bank 0, while T_i starts with Bank 2, i.e., $BS_{i-1} = BS_{i+1} = 0$ and $BS_i = 2$. The arrival time of T_i in the front-end is $t_a^{fe}(T_i)$. In this example, T_i arrives at the back-end at $t_a(T_i)$, which is after the scheduling of the last (i.e., second) *ACT* command of T_{i-1}. This allows pipelining between T_i and T_{i-1}, as shown in Figure 2.11. As a result, the back-end starts executing T_i after T_{i-1} finishes when the last *RD* or *WR* command is scheduled to Bank 1, i.e., $t_s(T_i) = t_f(T_{i-1}) + 1$ (see Case 1 in Figure 2.11). In this case, T_i starts just after $t_f(T_{i-1})$ and not with $t_a(T_i)$. T_i finishes in the back-end when its last *RD* or *WR* command is scheduled to Bank 3. Finally, the execution time of T_i is computed, which is from $t_s(T_i)$ to $t_f(T_i)$. T_{i+1} arrives when T_i has already finished, and there is no pipelining between them. In this case, $t_s(T_{i+1}) = t_a(T_{i+1})$, as shown in Figure 2.11.

Transaction T_i is finished in the front-end when all data is returned from the SDRAM if T_i is a read or the last *WR* command is scheduled in case T_i is a write. The time between a *RD* command and the first corresponding data word is constant (i.e., the JEDEC-specified read latency *tRL*). Each data burst consumes *BL*/2 cycles on the data bus, because of the double data rate. We can define the finishing time of T_i in the front-end by Definition 8 on the basis of its finishing time in the back-end. Note that each *RD* or *WR* command triggers the transfer of a data burst. The response time of T_i is hence defined as the time between its arrival time and the finishing time in the front-end, as given by Definition 9. Figure 2.11 shows the response time $t_{RT}(T_i)$ for a read and a write, respectively. The longest response time represents the worst-case response time of a transaction, as denoted by $\hat{t}_{RT}(T_i)$.

Definition 8 (Finishing time of transaction T_i in the front-end).

$$t_f^{fe}(T_i) = \begin{cases} t_f(T_i) + tRL + BL/2 & \textit{Read transaction} \\ t_f(T_i) & \textit{Write transaction} \end{cases}$$

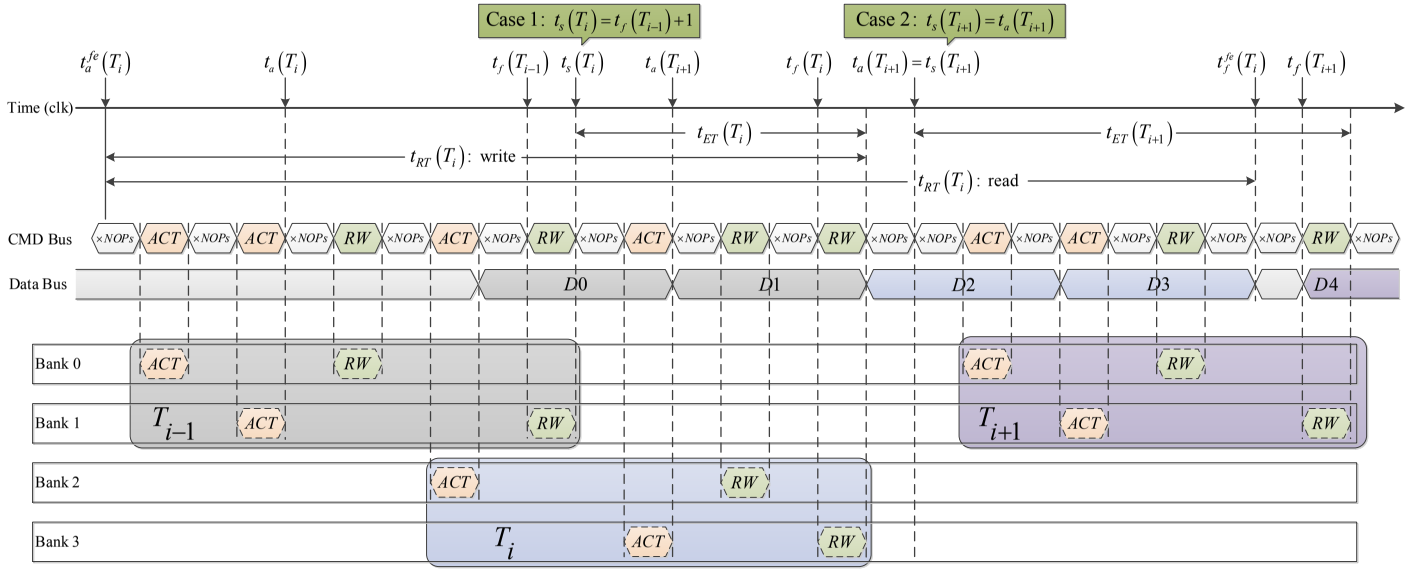


Figure 2.11: Timing definitions for accessing SDRAM with pipelining.

Definition 9 (Response time of transaction T_i)**.** $t_{RT}(T_i) = t_f^{fe}(T_i) - t_a^{fe}(T_i) + 1$

The memory bandwidth is the long-term rate of data transmission from/into the SDRAM. It is mainly determined by the execution of transactions, i.e., the scheduling of their memory commands. In addition, the refresh of SDRAM can interrupt the command scheduling of transactions. A refresh is regularly needed every *tREFI* cycles, i.e., a relatively long time period of $7.8\mu s$ for DDR3 SDRAMs [53]. When a refresh is needed, it has to wait for the end of the current transaction. The refresh time t_{ref} consists of the time to precharge all the currently open banks and the time to complete the refresh itself [72]. Specifically, t_{ref} is given by Eq. (2.2), consisting of the time *tRWTP* between the last *RD* or *WR* command of the current transaction and the associated *PRE* and the precharge period *tRP*, as well as the refresh period *tRFC*.

$$t_{ref} = tRWTP + tRP + tRFC \tag{2.2}$$

The bandwidth can be evaluated based on executing a transaction trace $\overline{T}$ that consists of a sequence of transactions. Definition 10 defines the bandwidth of a transaction trace based on the total transferred data and the total execution time of the trace. For an arbitrary transaction $T_i \in \overline{T}$ ($\forall i \geq 0$), its size is $S(T_i)$ that denotes the amount of transferred data, while $t_{ET}(T_i)$ is the execution time of T_i. Since the total execution times are measured in cycles, the SDRAM clock frequency denoted by f_{mem} is applied to compute the bandwidth in bytes per second (i.e., B/s). Moreover, the refresh efficiency e^{ref} given by Eq. (2.3) causes a reduction of the bandwidth. As a result, it is included to capture the impact of refresh on the bandwidth. Finally, the bandwidth representing the long-term data rate is achieved when the transaction trace is infinitely long, i.e., $|\overline{T}| = \infty$.

Definition 10 (Bandwidth of a transaction trace $\overline{T}$)**.**

$$bw(\overline{T}) = \frac{\sum_{\forall T_i \in \overline{T}} S(T_i)}{\sum_{\forall T_i \in \overline{T}} t_{ET}(T_i)} \times f_{mem} \times e^{ref}$$

$$e^{ref} = 1 - \frac{t_{ref}}{tREFI} \tag{2.3}$$

The worst-case bandwidth (WCBW) denoted by $\hat{bw}$ is the minimum bandwidth of all infinite transaction traces. It is defined by Definition 11.

Definition 11 (Worst-Case Bandwidth)**.**

$$\hat{bw} = \min_{\forall \overline{T}, |\overline{T}| = \infty} bw(\overline{T})$$

3

RUN-DMC: A REAL-TIME MEMORY CONTROLLER WITH DYNAMIC COMMAND SCHEDULING

The previous chapter introduced the background and design space of real-time memory controllers in general, which are required as the basis for understanding the memory controller presented in this chapter. Our memory controller is designed to deal with the diverse memory traffic generated in heterogeneous multi-core systems, which features transactions with variable sizes. On the other hand, the memory controller must be analyzable, such that the worst-case response time (WCRT) of transactions and/or the worst-case bandwidth (WCBW) are provided to meet the requirements of real-time applications. Meanwhile, the lowest average response time and maximum average bandwidth should be given to non-real-time applications. This chapter focuses on designing a real-time memory controller to achieve these goals, while its worst-case analysis will be achieved by three means, i.e., a formal analysis approach in Chapter 4 and two modeling methods in terms of mode-controlled dataflow [79] and timed automata [15] in Chapter 5 and Chapter 6, respectively.

This chapter summarizes the existing real-time memory controllers in Section 3.1, followed by introducing the design and evaluation of a real-time memory controller, named Run-DMC, which executes transactions with variable sizes by dynamically scheduling commands at run-time. Run-DMC achieves good performance by pipelining successive transactions and exploiting different bank parallelisms for variable transaction sizes, while being analyzable. Following the general memory controller architecture, Run-DMC is composed of a *front-end* and a *back-end*. The front-end shown in Section 3.2.1 provides arbitration between different requestors with variable transaction sizes using a novel work-conserving TDM arbiter. This TDM arbiter can reduce the worst-case response time of transactions. In Section 3.3, the back-end generates appropriate commands for transactions based on the memory mapping and dynamically schedules them to the SDRAM according to a priority-based algorithm. This enables pipelining between successive transactions, leading to smaller average response time and hence higher bandwidth. Finally, a cycle-accurate SystemC model of Run-DMC is built in Section 3.4, leading to a simulator to evaluate its performance in terms of average and measured worst-case response time and bandwidth. In addition, these results are compared to a state-of-the-art semi-static real-time memory controller [4] in Section 3.5. The results demonstrate that Run-DMC has much better average performance,

where non-real-time applications can benefit, while the measured maximum execution/response times are comparable between these approaches for systems with fixed transaction sizes. However, for variable transaction sizes, Run-DMC provides smaller measured maximum execution/response times.

3.1 RELATED WORK

Several types of real-time memory controller designs have been proposed in the past decade. Static [12] or semi-static [3, 25, 88] controller designs are used to achieve a bounded execution time of memory transactions. In [12], an application-specific static command schedule is constructed using a local search method. However, it requires a known static sequence of transactions, which is not available in a system with multiple applications. A semi-static method is proposed in [3] that generates static memory patterns, which are shorter sub-schedules of SDRAM commands computed at design time, and schedules them dynamically based on incoming transactions at run time. The drawback of this solution is that it cannot efficiently handle variable transaction sizes as the patterns are statically computed for a particular size. When it is employed by transactions with variable sizes in a system, larger transactions use the pattern multiple times, but smaller transactions use the pattern and discard unneeded data. This problem also applies to the semi-static controller in [37, 39], which uses a conservative open-page policy to improve the average performance of [3]. [88] presents a semi-static predictable DRAM controller that partitions sets of banks into virtual private resources with independent repeatable actual timing behavior. However, it requires constant duration for accessing the virtual resources. Thus, the actual or average case execution time is equal to the worst-case execution time.

Dynamic command scheduling is used because it more flexibly copes with variable transaction sizes and it does not require schedules or patterns to be stored in hardware. Several dynamically scheduled memory controllers have been proposed in the context of high-performance computing, e.g., [48, 50, 58]. These controllers aim at maximizing average performance and do not provide any bounds on execution times, making them unsuitable for real-time systems. Paolieri et al. [85] propose an analyzable memory controller, which uses dynamic command scheduling based on a modified version of the DRAMSim memory controller simulator [105], although the modifications to the original scheduling algorithm are not specified. However, it is limited to a fixed transaction size and a single memory map configuration. This also applies to [92], where transactions with fixed size are executed on an FPGA instance of a dynamically scheduled Altera SDRAM controller and analyzed using an on-chip logic analyzer. In [63, 107], a dynamically scheduled controller is presented that combines the notion of bank privatization with an open-page policy, which results in both low worst-case and average-case execution times. The memory controller in [26] employs several private banks to create a virtual device, which is shared by a single client requiring *guaranteed throughput (GT)*

and multiple clients with *best-effort (BE)* service. The GT client is prioritized over the BE client. However, these controllers cannot directly support variable transaction sizes and different memory map configurations. In addition, privatization assumes that the number of memory requestors is not greater than the number of memory banks. The number of banks is at most 32 supported by a DIMM with maximally 4 ranks, where each of them is composed of a DDR3 SDRAM with typically 8 banks. However, complex heterogeneous systems, such as [60], have more memory requestors. This limitation also applies to [25, 52] that employ bank privatization. The memory controller in [55] employs a *First-Ready First-come First-Serve (FR-FCFS)* policy to dynamically schedule commands for transactions with different priorities. This policy is also analyzed by [110], which considers the prioritization of read over write and multiple outstanding transactions. However, their worst-case analysis is pessimistic because of the conservative interference delay among different memory commands. For example, the maximum switching delay between write and read is always taken as the maximum delay for a read or write command. The analysis is also limited to a single transaction size and memory map configuration. This limitation also applies to [24], where the memory controller reorders *RD* and *WR* commands to reduce the number of data bus turn around.

In short, current real-time memory controllers do not efficiently address the dynamic memory traffic in complex heterogeneous systems because of the limitations either in architecture or in analysis with respect to variable transaction sizes and memory map configurations, or both. To fill this gap, this chapter presents a dynamically scheduled memory controller architecture supporting different transaction sizes and memory map configurations and the corresponding analyses will later be given in the following three chapters.

3.2 MEMORY CONTROLLER FRONT-END

This section introduces the hardware architecture of the Run-DMC front-end, which receives transactions with variable sizes from memory requestors and schedules them to the back-end according to a novel work-conserving TDM arbiter, offering lower interference delay. The back-end that executes each transaction by dynamically scheduling commands to the SDRAM is later introduced in Section 3.3.

3.2.1 *Front-End Architecture*

The front-end receives memory transactions on its ports from different requestors either directly or via a bus or a network-on-chip. Transactions are queued in the "trans" buffers per requestor, as illustrated in Figure 3.1, while the data read from or written into the SDRAM is stored in separate queues (i.e., read/write data queues). Typically, requestors generate memory transactions with fixed size, e.g., CPU cache misses [97]. Therefore, each requestor is assumed to have a fixed transaction size, while it varies

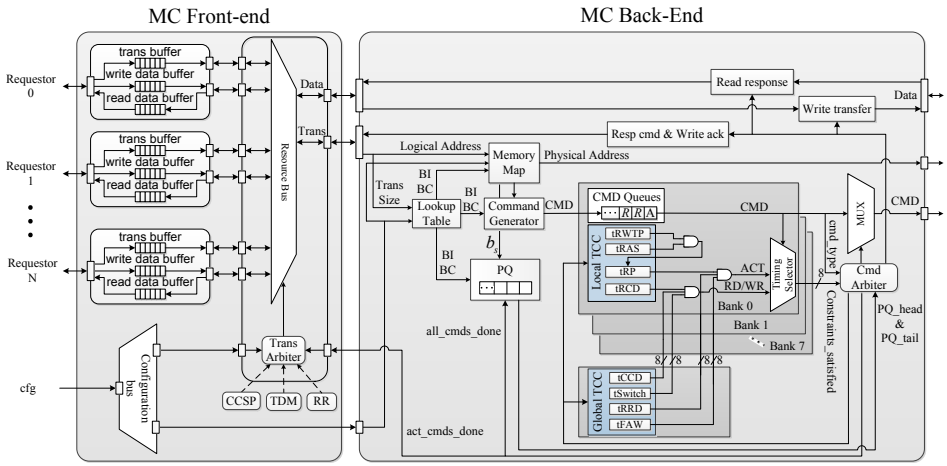


Figure 3.1: The architecture of the front-end and back-end of Run-DMC.

between different requestors. Note that this assumption is relaxed later in Section 4.4.5, where a safe WCRT is guaranteed by using the largest transaction size of a requestor when it actually generates variable transaction sizes. The size of a trans buffer is determined by the maximum number of outstanding transactions from the requestor. To avoid arbitrarily high self-interference when computing the WCRT, we assume each requestor has at most one outstanding transaction. This is common in a fully timing-compositional architecture [11]. The delay caused by the shared SDRAM is additive to the application execution time. After sending a read transaction, a requestor cannot schedule the next transaction before receiving the response, i.e., the data is returned. For a write transaction, an acknowledgment is sent back to the requestor after the data is written into the SDRAM memory, and then the requestor sends the next transaction. The read data buffer per requestor receives the data from the memory for a read transaction. Note that the read data buffer must have enough space when a read transaction is executed.

The front-end in Figure 3.1 supports any predictable arbiter chosen at design time. We have implemented CCSP [5], TDM [37, 72], and Round Robin [1], which can be chosen at design time and be configured via the configuration bus. The arbiter selects one transaction queue and sends the first transaction to the back-end. A novel TDM arbiter introduced in the next section is designed to efficiently deal with diverse memory traffic with variable transaction sizes. It exploits static information of the TDM schedule of requestors with different transactions sizes, resulting in lower worst-case response

time. The arbiter makes a scheduling decision when triggered by the back-end via the arbitration signal *act_cmds_done* in Figure 3.1.

3.2.2 *Work-Conserving TDM Arbitration for Variable-Sized Transactions*

We proceed by introducing a new work-conserving TDM arbiter for transactions with variable sizes. By exploiting the order of requestors based on their (largest) transaction size, the work-conserving TDM has a lower WCRT than traditional work-conserving TDM. We first discuss the issues of supporting variable transaction sizes and then specify the algorithm, before illustrating its operation with an example.

3.2.2.1 *TDM Arbitration Issues for Variable-Sized Transactions*

TDM arbiters employ time slots to serve requestors, each of which only receives the service within its allocated slots durations (i.e., the time period of the slots). All the slots constitute a TDM frame, which is periodically used by the TDM arbiter. Requestors are allocated either continuous or distributed slots in the frame. To simplify the WCRT analysis latter in Section 4.5, our TDM arbiter allocates continuous slots to each requestor. The non-preemptive TDM arbiter, shown in the front-end in Figure 3.1, serves requestors with different transaction sizes, which results in variable execution time for transactions and hence different time slot durations. The execution time is defined by Definition 7 as the scheduling time of the last command of the transaction minus the starting time of the transaction, and it depends on both the size of the transaction and the initial bank states when it arrives at the back-end. In particular, the size of the previous transaction affects the bank states, and a smaller previous transaction results in a larger WCET of the current transaction. The reason is that larger successive transactions pipeline more efficiently, as discussed later in Section 4.4. It hence follows that the order of serving requestors with different transaction sizes, i.e., their order in the TDM table influences the WCET of their transactions. From this discussion, we conclude that the duration of TDM slots varies and depends on several different factors. This is an issue that should be considered when using TDM arbiter for variable transaction sizes.

For a non-work-conserving non-preemptive TDM arbiter, each slot is statically allocated to a requestor. The slot therefore has the maximum duration equal to the WCET of transactions of that fixed size. Traditional work-conserving non-preemptive TDM dynamically reallocates unused slots to a requestor with pending transactions, according to some slack management policy. Unfortunately, this may increase the worst-case slot duration from the WCET of the (smallest) transactions of the idle slot owner, to the WCET of the transactions of any requestor receiving the slot (which may be the requestor with the largest transactions). Traditional work-conservation therefore has a negative effect on the WCRT in presence of variable-sized transactions by increasing the worst-case slot duration, which is another issue that needs to be addressed.

To solve these two issues, we firstly propose a new work-conserving policy for non-preemptive TDM arbiters used by requestors with variable transaction sizes. This policy has two innovations, which will be latter experimentally validated in Section 4.8.

1. When a requestor *r* that is allocated the current TDM slot has no pending transactions, the current slot becomes idle and we specify that *this idle slot and the following continuous slots belonging to r in the frame are skipped by the arbiter*. Instead, the next requestor with pending transaction(s) is served. As a result, idle slots are skipped, instead of being reallocated to another requestor (with larger transactions perhaps). Therefore, the maximum interference experienced by a requestor is always smaller than when its slots would have been reused by another requestor. Moreover, skipped slots reduce the waiting time for all other requestors.

2. We configure the TDM arbiter to *serve requestors in descending order of their transaction sizes*, such that their WCET and hence slot durations are smaller. This takes advantage of the fact that a transaction has a smaller WCET when preceded by a larger transaction. Note that the largest transaction is preceded by the smallest one because the TDM schedule repeats periodically. However, the approach still results in the minimum total length of all slots.

3.2.2.2 *Transaction Scheduling Algorithm*

Algorithm 1 presents the proposed work-conserving TDM arbitration. The inputs of Algorithm 1 include the arbitration signal *act_cmds_done* in Figure 3.1, which triggers the front-end to arbitrate as the back-end is ready to accept a new transaction. Another input is the information whether or not a transaction queue corresponding to a requestor has a pending transaction, and it is denoted by *RQueues*[ ], e.g., *RQueues*[*r*] == true implies that requestor *r* has a pending transaction. The third input is the TDM slot allocation, which is configured in a table *TDM_Table*[ ]. It specifies the order of serving requestors and the number of continuous slots per requestor. For example, *TDM_Table*[*r*] represents the number of slots allocated to requestor *r*. The TDM arbiter is configured to serve requestors in descending order of their transaction sizes, which is the second innovation presented previously. The transaction sizes of requestors hence decrease from requestor 0 to requestors *N*-1, and the requestors are served in this order. In a word, requestor 0 has the largest transaction size while requestor *N*-1 has the smallest size. Finally, the output of Algorithm 1 is the number of the transaction queue, denoted by *Q_ID*, whose head transaction is scheduled to the back-end.

To obtain *Q_ID*, Algorithm 1 uses two internal variables *r_index* and *s_index* that are the index of a requestor (associated with a transaction queue) and the index of its allocated slots, respectively. Algorithm 1 begins with initializing *Q_ID* to be invalid (line 5), and it ends when *Q_ID* becomes valid (line 16). However, the exploration of a valid *Q_ID* (between line 7 and 17) starts only if *act_cmds_done* is true and there exists at least one transaction queue with a pending transaction, as shown on line 6. If there are no queues

Algorithm 1 Transaction scheduling with work-conserving TDM

```
1:  Inputs: act_cmds_done, RQueues[ ], TDM_Table[ ]
2:  Internal state: r_index, Q_ID, s_index
3:  Initialization: act_cmds_done ← true; r_index ← 0; s_index ← 0;
4:  Begin:
5:  Q_ID← invalid; /*No requestor is selected.*/
6:  if act_cmds_done = true && ∃i, RQueues[i] has a transaction then
7:    Repeat:
8:      if RQueues[r_index] has a transaction then
9:        Q_ID ← r_index; /*Serve requestor r_index, and update the index of its slots.*/
10:       s_index ← (s_index + 1) mod TDM_Table[r];
11:       if s_index = 0 then /*The final slot is taken by requestor r_index.*/
12:         r_index ← (r_index + 1) mod N; /*Update the requestor index.*/
13:     else /*The requestor has no transaction, skip forward to the next one. */
14:       r_index ← (r_index + 1) mod N; /*Update the index for next requestor.*/
15:       s_index ← 0; /*Initialize the slot index for the next requestor r_index.*/
16:   Until Q_ID is valid.
17: End
18: Output: Q_ID
```

with pending transactions, this algorithm restarts the following clock cycle until a new transaction arrives at the front-end. The algorithm firstly checks whether the request queue indexed by *r_index* has a transaction (line 8). If not, then its allocated slots become idle and are *skipped* by setting *r_index* to the next transaction queue, and *s_index* to 0 (line 14 to 15). This is different from traditional work-conserving TDM arbitration, where the idle slots are reallocated to another arbitrary requestor with pending transaction(s). This is the first innovation as discussed previously. If the transaction queue *r_index* has a pending transaction (line 8), the algorithm behaves the same as normal TDM arbitration.

Note that round robin is a special case of TDM when each requestor is only allocated a single slot in the table. In TDM, requestors can have more than one slot and the allocated slots can be placed in any order in the TDM table [8]. In our case, we allocate continuous slots to a requestor, and the slots of different requestors are placed in descending order of their transaction sizes.

3.2.2.3 *Example*

Take four requestors r_0, r_1, r_2, and r_3 with different transaction sizes as an example to illustrate the benefits of the proposed work-conserving TDM arbitration. As previously stated, we assume the transaction sizes decrease from r_0 to r_3. Each of them is allocated one slot in the TDM table, as shown in Figure 3.2 (a). The slot duration of each requestor is the WCET of its transactions experienced in the back-end. The larger transactions

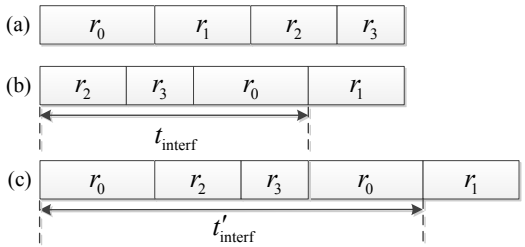


Figure 3.2: The worst-case interference delay for requestor r_1: (a) TDM slot allocation; (b) the proposed work-conserving TDM arbiter; (c) traditional work-conserving TDM arbiter.

have larger WCET, as later shown in Section 4.4.5. As a result, the slot duration of r_0 is the largest while it is the smallest for r_3 (see Figure 3.2 (a)). Moreover, the TDM arbiter is configured to serve requestors in descending order of their transaction sizes. As a result, the requestors are served in the order from r_0 to r_3.

Take requestor r_1 as an example. In the worst case, a transaction from r_1 arrives just as it misses its slot. This idle slot is hence skipped according to the proposed work-conserving TDM arbitration, and the following requestors r_2, r_3 and r_0 use their allocated slots. This leads to the maximum interference delay t_{interf} for r_1, as shown in Figure 3.2 (b). A traditional work-conserving TDM arbiter could reallocate this idle slot to another requestor, e.g., r_0 as a bonus, according to some slack-management policies. Then, the following requestors r_2, r_3, and r_0 consume their allocated slots (Figure 3.2(c)), leading to interference delay $t'_{interf} > t_{interf}$. The difference between them is actually the duration of the idle slot that was given as a bonus to requestor r_0. Hence, the proposed work-conserving TDM arbiter is capable of providing smaller WCRT for transactions with variable sizes. Moreover, this benefit also applies to transactions with fixed size, since it is a special case when all these four requestors in the example have the same transaction size.

3.3 MEMORY CONTROLLER BACK-END

The memory controller back-end receives scheduled transactions from the front-end, as shown in Figure 3.1. However, it is a general component that could be used without the front-end, for example by connecting to a memory tree NoC [32] that plays the arbitration role to schedule transactions from different requestors to the back-end. Each arrived transaction is translated into a number of memory commands that are scheduled to a number of consecutive banks subject to the timing constraints of the memory. The basic idea underlying the back-end architecture (Section 3.3.1) and command arbiter (Section 3.3.2) is that each transaction (i.e., T_i, $\forall i \geq 0$) generates an *ACT* command followed by BC_i times *RD* or *WR* commands, the last one with an auto-precharge. This commences with the starting bank BS_i and is repeated BI_i times for all the required

banks. Commands are generated one per cycle, but are usually scheduled more slowly, due to timing constraints. Commands are therefore buffered per bank (see Figure 3.1, and discussed below). To limit the size of the command queues per bank while still enabling pipelining between transactions, a new transaction is sent by the front-end and hence new commands are admitted to the queues only when all the *ACT* commands of the current transaction have been issued to the memory. This is enforced by the command arbiter via the act_cmds_done signal in Figure 3.1 that triggers a new scheduling decision in the front-end. To avoid read/write hazards or read-response reorder buffers, the *RD*/*WR* commands of T_i are scheduled before those of the next transaction T_{i+1}. This order (as a (BI_i, BC_i, BS_i) tuple) of each transaction is stored in the parameter queue (PQ), and used by the command arbiter to guarantee in-order execution of transactions. This results in an efficient pipelined back-end.

3.3.1 *Back-End Architecture*

We proceed by introducing the main components in the back-end, which include the **Lookup Table**, parameter queue (**PQ**), **Command Generator** and the **Cmd Arbiter**, as shown in Figure 3.1. In addition, other common components used by existing memory controllers are also briefly introduced to show how these components constitute a dynamically scheduled back-end.

As shown in Figure 3.1, 1) the **Lookup table** translates the transaction size to the bank interleaving number (*BI*) and burst count (*BC*), which are needed by the command generation. They are determined at design time when the memory map configuration is chosen and are programmed via a configuration interface (cfg) when the system is initialized. If there is no (*BI*, *BC*) corresponding to a transaction size in the Lookup Table, the (*BI*, *BC*) related to the next larger size is used with the additional data being masked out. An important (usually unstated) assumption on the translation from size to (*BI*, *BC*) is that it must be monotone, as given by Definition 12, where $S(T_m)$ and $S(T_n)$ are the sizes of transaction T_m and T_n, respectively, where $\forall m \geq 0$ and $\forall n \geq 0$. A methodology to choose the memory map configuration based on the requirements of bandwidth, execution time and power consumption has been presented in [36]. 2) With the *BI* and *BC*, the widely used **Memory Map** module in Figure 3.1 translates the logical address of the transaction into the starting physical address that consists of the starting bank *BS*, row, and column. 3) Then (*BI*, *BC*, *BS*) of the transaction is inserted at the back of the parameter queue (PQ). This queue keeps track of the order of transactions in the back-end and is used by the command scheduling algorithm in Section 3.3.2.

Definition 12 (Monotone memory mapping)**.** *For* $\forall m \geq 0$ *and* $\forall n \geq 0$, $S(T_m) \leq S(T_n) \implies BI_m \leq BI_n \wedge BC_m \leq BC_n$.

Based on (*BI*, *BC*) and the physical address, 4) the **Command Generator** generates memory commands for each bank according to the rules introduced at the beginning

of this section. It generates an *ACT* command followed by *BC RD* or *WR* commands, which are sequentially inserted into the command queue (i.e., FIFO) per bank. The last command attaches an auto-precharge flag. This is repeated for each of the *BI* banks. Note that a new transaction can be sent by the front-end when the arbitration signal act_cmds_done is true, which happens only if all the *ACT* commands of the currently executed transaction are no longer in the command queue, i.e., have been issued to the memory.

To keep track of the timing constraints of the commands, 5) **timing counters** are commonly used by dynamically scheduled memory controllers. Each counter tracks one timing constraint specified by the JEDEC DDR3 standard [53]. We classify the *timing constraint counters (TCC)* into *local TCC* and *global TCC*, which constrain the command scheduling for the same bank and different banks, respectively. Most timing constraints shown in Figure 3.1 are directly provided by JEDEC, while *tRWTP* and *tSwitch* are derived from the JEDEC specification and are given by Eq. (3.1) and (3.2), respectively. *tRWTP* is the time between a *RD* or *WR* command and the precharging to the same bank, while *tSwitch* limits the time between two successive *RD* and/or *WR* commands. Due to the double data rate of DDR SDRAM, *BL*/2 is the time consumed transferring a burst of data associated with a *RD* or *WR* command.

$$tRWTP = \begin{cases} tRTP & \text{PRE } \textit{follows} \text{ RD} \\ tWL + BL/2 + tWR & \text{PRE } \textit{follows} \text{ WR} \end{cases} \tag{3.1}$$

$$tSwitch = \begin{cases} tRL + tCCD + 2tCK - tWL & \text{WR } \textit{follows} \text{ RD} \\ tWL + BL/2 + tWTR & \text{RD } \textit{follows} \text{ WR} \\ tCCD & \textit{otherwise} \end{cases} \tag{3.2}$$

A command that is at the head of the command queue can be issued only if its timing constraints are satisfied in the current cycle. It is then called a *valid command*. As shown in Figure 3.1, the 6) **Timing Selector** of the bank shows whether the timing constraints for the head command are satisfied. Multiple command queues may have a valid command simultaneously. This implies *command scheduling collisions*, since only one command can be issued per cycle on the command bus. Therefore, an arbiter is required to select a valid command, which is the 7) **Cmd Arbiter** shown in Figure 3.1. It has to guarantee in-order execution of transactions to avoid the architectural and analysis complexity of re-ordering. Moreover, it provides the valid arbitration signal act_cmds_done to the front-end when all the *ACT* commands of the current transaction have been scheduled, such that the front-end schedules a new transaction to enable pipelining of transactions. To achieve these goals, it uses the command scheduling algorithm presented in Section 3.3.2. Finally, the chosen command is removed from the command queue and is passed to the memory. When a *RD* or *WR* command is scheduled, the Read response and Write transferring modules in Figure 3.1 are enabled to transfer data

from/into the memory via the data bus. Moreover, both the local and global TCC associated with the scheduled command are reset. This is shown by the feedback wires from the output of the arbiter to the TCC in Figure 3.1. Lastly, a refresh command needs to be scheduled every *tREFI* cycles. Once triggered, it is scheduled after the data transmission of the currently executing transaction to prevent unnecessary interference, while still ensuring that no refresh command is delayed more than $9 \times tREFI$ clock cycles, as specified by the DDR3 standard [53]. Refresh is also implemented by timing counters, which are not depicted in Figure 3.1 for simplicity.

3.3.2 *Dynamic Command Scheduling Algorithm*

After memory commands are generated and stored in the command queues by the Command Generator in Figure 3.1, the arbiter has to decide which command to schedule every clock cycle for transactions in the back-end. It has to solve three critical issues, namely:

1. a single command must be chosen from the set of valid commands;
2. transactions must be executed in first-come-first-serve (FCFS) order to avoid reorder buffers for the responses;
3. to simplify logical-to-physical address translation [39], successive banks of a single transaction have to be accessed in ascending order.

These issues are not independent from each other, and we proceed by explaining how they are addressed by the arbiter. To guarantee the FCFS, the valid commands of a transaction have higher priority than the valid commands of the following transactions. Moreover, to transfer data as quickly as possible to/from the memory, valid *RD*/*WR* commands have higher priority than *ACT* commands, resulting in lower execution time. Within a transaction, the command queue corresponding to a bank with a lower number has higher priority, forcing banks to be served in ascending order. Though these priorities cannot guarantee an optimal command scheduling algorithm, they solve the three critical issues.

These priorities form the basis of Algorithm 2 that is used by the arbiter to select a command from the multiple valid commands in every cycle. Note that a NOP is scheduled when there is no valid command in a cycle. As shown in Figure 3.1, the inputs of the arbiter include the outputs of the Timing Selectors, the type (*ACT*, *RD* or *WR*) of each command at the head of the command queues, and the head and tail elements of the parameter queue. These inputs are taken by Algorithm 2 and represented by constraint_satisfied, cmd_type, and PQ_head and PQ_tail, respectively. constraint_satisfied and cmd_type are arrays with sizes equal to the number of command queues. The outputs of Algorithm 2 are bank_id, all_cmds_done and act_cmds_done, where bank_id indicates the command queue whose head command can be scheduled to bank bank_id.

all_cmds_done is true when all commands of the current transaction have been issued to the memory. The (*BI*, *BC*, *BS*) triple at the head of the parameter queue is then removed. act_cmds_done indicates whether all *ACT* commands of the current transaction have been sent to the memory. When true, this triggers the front-end to arbitrate for a new transaction, even though *RD*/*WR* commands of current and past transactions are (likely to be) pending.

Algorithm 2 Dynamic command scheduling

```
1:  Inputs: PQ, constraint_satisfied, cmd_type
2:  Internal state: rw_bank, act_bank
3:  Initialization: bank_id ← null; act_bank ← null; rw_bank ← null;
    act_cmds_done ← true; all_cmds_done ← false;
4:  if act_bank = null then act_bank ← PQ_tail.b_s; act_cmds_done ← false;
5:  if rw_bank = null then rw_bank ← PQ_head.b_s;
6:  if cmd_type[rw_bank] = RD/WR and constraint_satisfied[rw_bank] = true then
7:    bank_id ← rw_bank;
8:    if last RD/WR of PQ_head transaction then
9:        rw_bank ← null;
10:       all_cmds_done ← true;
11:   else if last RD/WR of PQ_head transaction to bank bank_id
12:       then rw_bank ← rw_bank+1;
13: else if act_bank != null and then
14:   if cmd_type[act_bank] = ACT and constraint_satisfied[act_bank] = true then
15:       bank_id ← act_bank;
16:       if last ACT of PQ_tail transaction then
17:           act_bank ← null; act_cmds_done ← true;
18:       else act_bank ← act_bank+1;
19: Outputs: bank_id, act_cmds_done, all_cmds_done
```

In Algorithm 2, line 6 checks whether there is a valid *RD*/*WR* command for the current bank (rw_bank) for reading/writing. Otherwise, line 14 checks whether there is a valid *ACT* command. This guarantees that a valid *RD* or *WR* command has higher priority than a valid *ACT* command. *act_bank* and *rw_bank* indicate the number of the bank to which an *ACT* or a *RD*/*WR* command can be scheduled, respectively. *act_bank* is increased by one after an *ACT* command has been selected (line 18), while *rw_bank* increases by one when *BC* number of *RD*/*WR* commands of the current transaction have been scheduled to bank *bank_id* (line 12). This update scheme ensures the banks are accessed in ascending order for each transaction. *act_bank* and *rw_bank* are initialized with the starting bank *BS* of the transactions associated with the tail and head of the parameter queue, respectively (line 4, 5). A new transaction can only be sent to the back-end if all the *ACT* commands of the current transaction have been issued, as indicated by act_cmds_done (line 17). As a result, only one transaction has *ACT* commands in the command queues, namely the one at the tail of the parameter queue (PQ_tail). Hence,

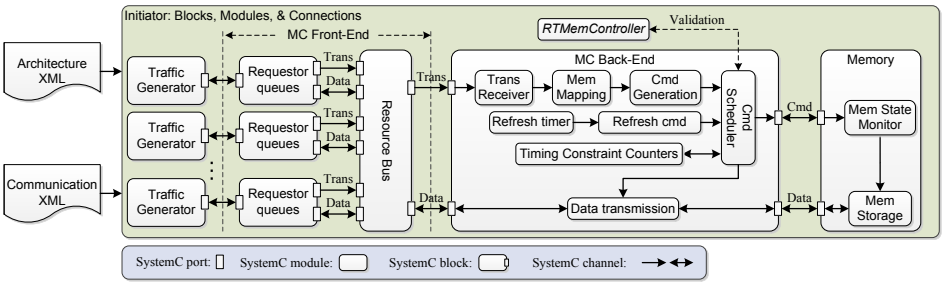


Figure 3.3: The structure of the cycle-accurate SystemC simulator of Run-DMC.

transactions are served in FCFS order, the banks of each transaction are served in ascending order, and command priorities ensure that only a single command is scheduled per cycle. Algorithm 2 thus addresses all three critical issues mentioned previously. Although command priorities are used, there is no livelock or starvation since transactions are executed in FCFS order.

Regarding the hardware cost, our memory controller is comparable to existing memory controllers, such as the one based on First-Ready First-Come First-Serve (FR-FCFS) policy [55], the ROC [63] and the cadence DDR controller [20]. Our memory controller has common components with these existing memory controllers, such as the request/response buffers in the front-end and the memory map, command queues, command generator, and the timing constraint counters in the back-end. The additional components of our memory controller are the lookup table and the parameter queue, which have a limited number of entries. Moreover, the arbiters in the front-end and back-end use Algorithms 1 and 2, respectively. They are similar to existing arbiters, such as the work-conserving TDM [38] and the 3-level arbitrations of ROC [63]. We therefore expect our memory controller to be similar in area and speed to existing designs.

3.4 CYCLE-ACCURATE SYSTEMC MODEL OF RUN-DMC

Run-DMC is implemented as a cycle-accurate SystemC model, where the functional components shown in Figure 3.1 are captured as SystemC blocks with ports and the connections between different components are modeled by SystemC channels. This cycle-accurate model has been implemented as a SystemC simulator and its structure is shown in Figure 3.3. The simulator takes the XML specifications of the architecture and communication of the memory controller as its inputs. These specifications provide abstract descriptions of the memory controller, such as the arbiter in the front-end and the requestors with different requirements in terms of latency and bandwidth, etc. Then, the particular blocks, ports, and connections are instantiated (see Figure 3.3).

The SystemC simulator adds traffic generators, which behave as the sources to provide the arrived traffic generated by requestors. The traffic are generated either in a

synthetic way, e.g., the transaction arrival follows a normal distribution, or according to the memory traces of benchmark applications. The traffic consists of read and/or write transactions along with the data from/into the memory. They are enqueued into the Requestor queues, which are included in a SystemC block, as shown in Figure 3.3. Each of these blocks has two ports for transferring transactions and data to the Resource Bus block, respectively. Note that the data port can also receive data from the Resource Bus for read transactions. The Resource Bus presented in Figure 3.3 implements the arbitration between different requestors and also forwards the transactions to the back-end of Run-DMC or transfers data from/into the memory via the back-end. Therefore, the Requestor queues and the Resource Bus in Figure 3.3 model the front-end of Run-DMC, which has been illustrated in Figure 3.1.

The back-end of Run-DMC is captured by a single SystemC block, as shown in Figure 3.3. It receives transactions from the Resource Bus via one of its input ports, while the other port is used to transfer data. Transactions are received by the TransReceiver SystemC thread, followed by translating the logical address of a transaction to the physical address by the MemMapping function. The CmdGeneration thread generates commands for each transaction and enqueues them into the corresponding command queues. Next, the CmdScheduler thread schedules each command based on Algorithm 2. Its inputs include the pending commands of the transaction and the refresh command. The latter is generated by the RefreshCmd function according to the RefreshTimer thread that tracks the refresh period timing constraint *tREFI* (see Table 2.1). Moreover, CmdScheduler has the input from the thread Timing Constraint Counters, which tracks the timing constraints for scheduling commands of transactions, while feedback is given in the other direction to reset the counters when a command is scheduled. Finally, when a command is scheduled by the CmdScheduler, it is also validated by our open-source tool RTMemController [70], which computes the scheduling time of each command based on the scheduling dependencies. RTMemController is later presented in Section 4.7. In addition, the scheduling of a *RD*/*WR* command also triggers the Data transmission thread, resulting in data transfer from/into the memory via the back-end.

The SDRAM device has been modeled by the Memory block shown in Figure 3.3. When it receives a command from the back-end, the memory state monitor that is a SystemC thread checks whether all timing constraints are satisfied for the received command. It ensures that no timing constraint is violated for the SDRAM device. Note that this should never happen and it is used to debug the memory controller. If a command is successfully received, memory state monitor is updated to track the timing constraints for a new command. In addition, the memory storage thread is triggered to receive or send a burst of data for a *WR* or *RD* command, respectively. At the end, data is transferred between the Requestor queues in the front-end and the memory through the back-end, as indicated in Figure 3.3.

3.5 EXPERIMENTAL RESULTS

This section experimentally evaluates our memory controller Run-DMC, which is implemented as a cycle-accurate SystemC simulation model. The experimental setup is presented and it will be used throughout this thesis to validate the bounds derived with different techniques in the following chapters. In addition, our SystemC simulator is debugged using an open-source tool RTMemController, which formally captures the timing behavior of Run-DMC and will be latter introduced in Chapter 4. As a result, the SystemC is ensured to provide accurate command scheduling results, which are the prerequisite to derive accurate response/execution time and bandwidth. Experiments are carried out for requestors with the same transaction sizes or variable sizes, respectively. We collect the average and maximum measured results in terms of response/execution time and bandwidth. Moreover, these results are compared to a state-of-the-art semi-static approach [3], which is the only existing approach supporting different memory map configurations.

3.5.1 *Experimental Setup*

The cycle-accurate SystemC simulation model of Run-DMC runs on a 64-bit Ubuntu 12.04.5 LTS system with 8 Intel(R) Core(TM) i7 CPU running at 3.07 GHz and with 24 GB RAM. The experiments use a combination of independent real application traces and/or synthetic traffic. Each of them generates one transaction stream and they result in a mixed stream in the memory controller back-end after the arbitration in the front-end, which is assumed to have four requestors for most experiments in this thesis. Note that the TDM arbiter in the front-end uses continuously allocated slots and most experiments assume one slot per requestor for simplicity. The allocation of TDM slots per requestor to meet the latency and/or bandwidth requirements is out of the scope of this thesis. Please refer to [8, 77] for this issue. We use application traces generated by running applications from the MediaBench benchmark suite [65] on the SimpleScalar 3.0 processor simulator [10], which uses separate L1 data and instruction caches, each with a size of 16 KB. The L2 caches are private unified 128 KB caches where the cache-line size varies depending on the experiments. Synthetic traffic is generated using a normal distribution with very low variance, resulting in near-periodic traffic inspired by e.g., some hardware accelerators and display controllers in the multimedia domain. For each transaction size in the experiments, we have chosen the memory map configuration that provides the lowest execution time for transactions by interleaving more banks to exploit bank parallelism. The configured (*BI*, *BC*) for transaction sizes of 16 bytes, 32 bytes, 64 bytes 128 bytes, and 256 bytes are hence (1, 1), (2, 1), (4, 1), (4, 2), and (4, 4) respectively [36]. (4, 2) and (4, 4) are used by 128 byte and 256 bytes transactions instead of (8, 1) and (8, 2) because of *tFAW* that causes a longer execution time with (8, 1) and (8, 2). Experiments have been done with three JEDEC-compliant DDR3 SDRAMs,

DDR3-800D, DDR3-1600G, DDR3-2133K, all with interface widths of 16 bits and a capacity of 2 Gb [53]. Refresh is manually scheduled according to the schema described in Section 3.3.1, where a refresh starts to be scheduled after the finishing of the currently executed transaction, i.e., the last *RD* or *WR* is scheduled. Since refresh is periodically needed with a relatively long time period *tREFI*, its effect on the application results in a slight increase in the overall execution time. Therefore, we collect the measured maximum execution/response times of a transaction without taking refresh into account. However, when collecting average execution/response times, the effect of refresh is included.

3.5.2 *Fixed Transaction Size*

This experiment evaluates our approach for systems with fixed transaction size, and compares the results to a semi-static approach [3]. Moreover, this experiment also collects the measured worst-case results, which are the maximum results obtained from the simulation. Note that the analytical worst-case bounds will be derived in latter chapters using different approaches. Four memory requestors are used, corresponding to four processors executing different MediaBench applications (*gsmdecode*, *epic*, *unepic* and *jpegencode*), which generate a large number of transactions. The TDM arbiter in the front-end allocates one slot per requestor. For each application, the total number of transactions (*TransN*) and the ratio (or percentage) of read transactions (*RRatio*) are shown in Table 3.1. The processors access the memory through their L2 caches and have the same cache-line size. The experiment is executed for four different cache-line sizes of 32 bytes, 64 bytes, 128 bytes, and 256 bytes with different memory map configurations, respectively.

Table 3.1: Characterization of memory traffic with fixed transaction size.

Size (bytes)	*gsmdecode*		*epic*		*unepic*		*jpegencode*	
	TransN	*RRatio*	*TransN*	*RRatio*	*TransN*	*RRatio*	*TransN*	*RRatio*
32	19734	64.4%	182957	69.7%	129145	61.0%	173995	87.4%
64	10104	64.3%	96984	69.3%	67664	61.0%	92905	87.8%
128	5216	64.1%	55644	69.8%	36540	60.9%	55192	89.1%
256	2626	64.5%	24577	70.5%	12675	61.2%	25159	88.9%

3.5.2.1 *Execution Time*

The execution time of a transaction is the time required by the back-end to schedule commands to the memory. This experiment hence only evaluates the dynamically scheduled back-end of Run-DMC. The front-end will be included later when evaluating response

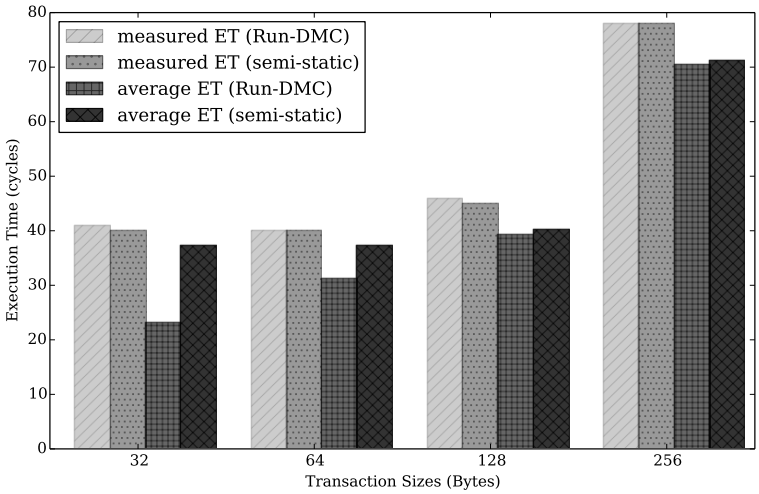


Figure 3.4: The maximum measured WCET and average execution time (ET) for DDR3-1600G SDRAM with fixed transaction sizes.

times. Note that the back-end can be independently used in a system or can be used with different front-ends. In addition, the back-end is a key component when determining response times. As a result, it is necessary to provide the execution time in the back-end. The maximum measured WCET and the average execution times of transactions with fixed size accessing a DDR3-1600G memory are presented in Figure 3.4. The results for other memories are similar and not shown. We can observe that:

1. The maximum measured WCET of our Run-DMC is the same as the semi-static approach [3] with only a few exceptions, where the WCET of 32 bytes and 128 bytes are 1 cycle larger, as shown in Figure 3.4. However, these exceptions rarely occur for all the DDR3 SDRAMs, because they rely on the particular JEDEC timing constraints and the specific sequence of previously executed transactions.

2. Run-DMC achieves significantly better average execution time than the semi-static approach for all these transaction sizes with different DDR3 memories, as shown in Figure 3.4, where DDR3-1600G is taken as an example. This is because dynamic command scheduling monitors the actual state of the required banks and issues commands earlier for a transaction that requires a different set of banks from that of the previous transaction. In contrast, the semi-static scheduling [3] uses pre-computed schedules that always assume worst-case initial bank state for every transaction. Figure 3.5 shows the improvement in average ET, which is defined as $100\% \times (1 - \bar{t}^d_{ET}/\bar{t}^s_{ET})$. $\bar{t}^d_{ET}$ and $\bar{t}^s_{ET}$ denote the average ET of our dynamic approach and the semi-static approach, respectively.

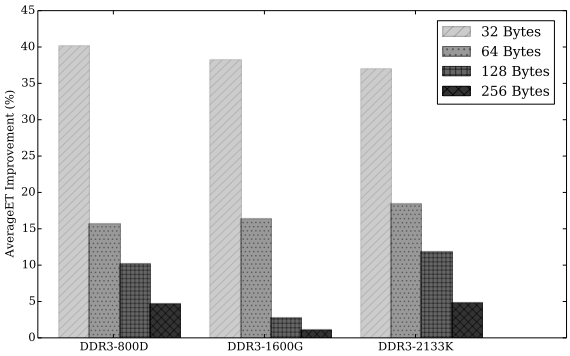


Figure 3.5: The improvement of the average execution time (ET) for different DDR3 SDRAMs with fixed transaction sizes. The results of Run-DMC are compared to the semi-static approach [3]

3. Moreover, we see that smaller transactions benefit more from dynamic command scheduling. For example with DDR3-800D, 32 byte transactions gain 40.2% while 256 byte transactions gain 4.7%. The reason is that smaller transactions require fewer banks, increasing the chance that the next transaction accesses different banks and thus be scheduled earlier.

3.5.2.2 *Bandwidth*

Bandwidth is another metric to evaluate the performance of a memory controller. It represents the long-term rate of transferring data from/into the SDRAM. It is a main concern for memory-intensive applications with a high throughput requirement. As defined by Definition 10 in Chapter 2, bandwidth is computed based on the transaction size and its execution time. As a result, we can obtain the measured minimum (i.e., worst-case) and the average bandwidth based on the measured WCET and average ET. For example, the bandwidth results for DDR3-1600G SDRAM with fixed transaction sizes are illustrated in Figure 3.6. We can draw the same conclusions as for execution time, where 1) the measured worst-case bandwidth provided by our Run-DMC is comparable with the semi-static approach, while 2) significantly better average bandwidth is achieved, as shown in Figure 3.6. Moreover, 3) smaller transaction sizes benefit more from the dynamic command scheduling to achieve better average bandwidth. These conclusions are expected because the bandwidth is calculated based on the execution time according to Definition 10. We can also observe that 4) larger bandwidth is given using larger transaction sizes. It is for the reason that more consecutive data bursts are transferred for each bank activation, resulting in higher efficiency of transferring data. These conclusions are also observed from the results of other DDR3 SDRAM memories, although they are not shown for brevity.

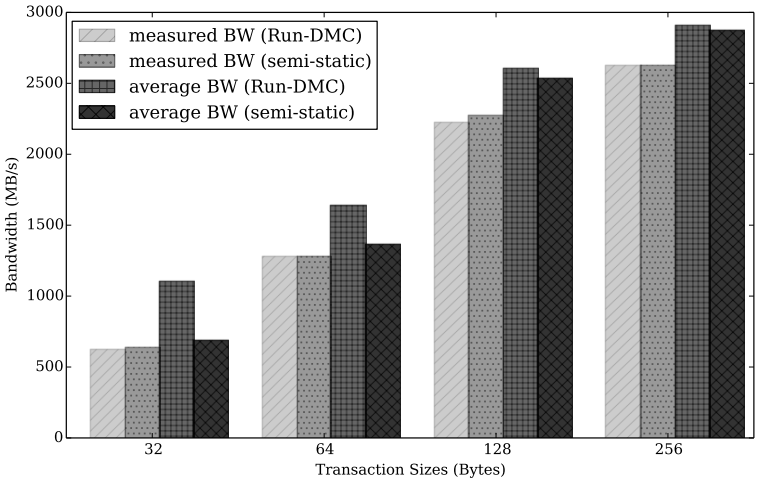


Figure 3.6: The measured minimum and average bandwidth for DDR3-1600G SDRAM with fixed transaction sizes.

3.5.2.3 *Response Time*

The response time of a requestor is essentially determined by accumulating the execution time of transactions from each requestor, which are executed within their allocated TDM slots. However, the random arrival of transactions makes the response time of a requestor varying, which depends on the number of interfering transactions from other requestors and the initial bank states when executing its own transaction. This experiment collects the maximum measured response time (RT) of read and write transactions, respectively. Figure 3.7 presents the measured RT obtained from both our dynamically-scheduled memory controller and its semi-static counterpart for fixed transaction sizes, where DDR3-1600G SDRAM is taken as an example. We can see that 1) the response time of read transactions is larger than that of write transactions. The reason is that a read finishes when its last data word is returned, while a write transaction is done when its last *WR* command is scheduled, as given by Definition 9 determining the response time of the memory controller. 2) Comparing to the semi-static approach, Run-DMC achieves smaller RT for all fixed transaction sizes except 256 bytes, as shown in Figure 3.7.

For small transaction sizes (i.e., 32 bytes, 64 bytes, and 128 bytes), Run-DMC provides smaller measured RT because it dynamically exploits bank parallelism, since small transactions typically access different sets of banks. However, the semi-static approach cannot exploit this bank parallelism across transactions requiring different banks. For large transaction size of 256 byte, the measured RT of semi-static approach is smaller than that given by Run-DMC, as shown in Figure 3.7. On one hand, 256-byte transactions have a fewer sets of banks to use. As a result, these two approaches perform closely. On

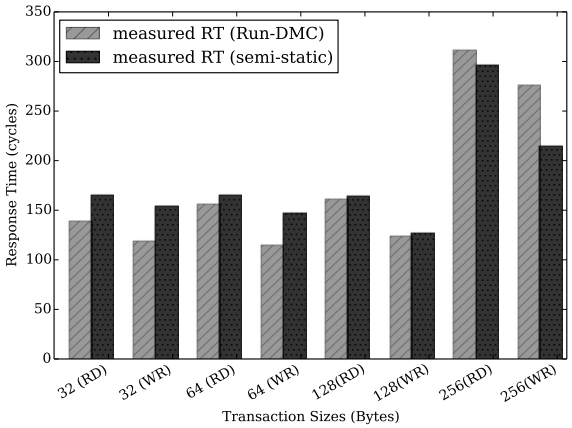


Figure 3.7: The maximum measured response time (RT) for DDR3-1600G SDRAM with fixed transaction sizes.

the other hand, to support the pipelining between successive transactions, Run-DMC specifies that a new transaction can be sent to the back-end when all the *ACT* commands of the current transaction are scheduled. This further enables the front-end to receive the next transaction. In contrast, the semi-static approach does not support pipelining between successive transactions. As a result, a transaction is sent to the back-end when the current transaction is finished, i.e., the last *RD* or *WR* command is scheduled. As a result, the front-end becomes available to receive the next transaction latter than that of Run-DMC, since *ACT* commands of the current transaction are scheduled earlier than the *RD* or *WR* commands. Therefore, the next transaction received by the front-end of Run-DMC may experience longer response time than that of the semi-static approach. To conclude, it is hard to say whether our Run-DMC always outperforms the semi-static approach in the worst-case according to these experimental results (e.g., measured ET/RT). The reason is that the worst-case scenario may not be covered by running these MediaBench application traces. We will later formally compare these two approaches in Chapter 4, 5, 6.

Our dynamically-scheduled memory controller achieves better average performance, as demonstrated by the execution time in Figure 3.4. We further explore this benefit by deriving the improvement in average RT for each MediaBench application trace when comparing Run-DMC to the semi-static approach. The improvement is shown in Figure 3.8, and its definition is similar to that of the average ET in Figure 3.5. Figure 3.8 demonstrates that the average RT of each MediaBench application trace is greatly improved, e.g., 43.7% improvement is achieved to access a DDR3-1600G SDRAM for *jpegencode* with 32-byte transactions. Moreover, smaller transaction sizes can benefit more from our dynamically-scheduled memory controller.

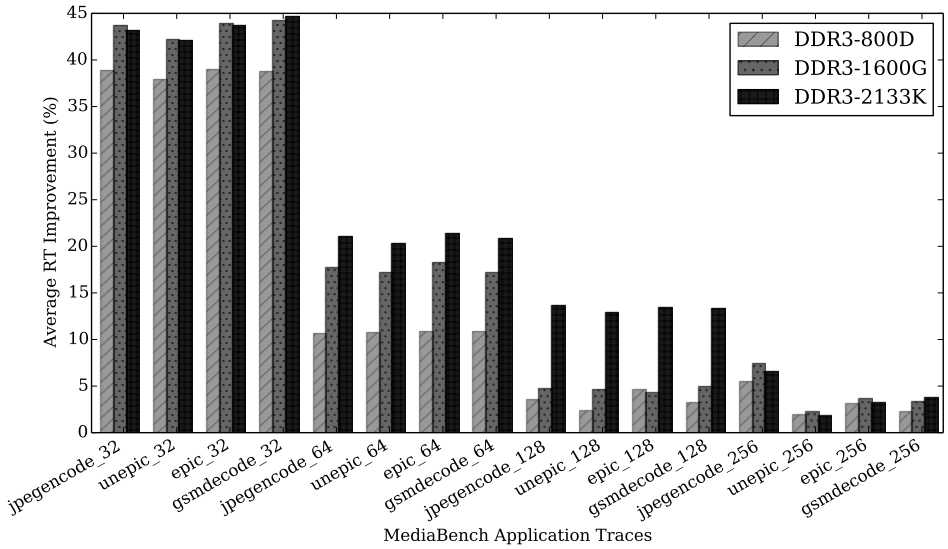


Figure 3.8: Comparison to a semi-static approach [3] in average response time of Mediabench application traces for different DDR3 SDRAMs with fixed transaction size.

3.5.3 *Variable Transaction Sizes*

The last experiment evaluates Run-DMC with variable transaction sizes. The setup is loosely inspired by a High-Definition video and graphics processing system [31] featuring a number of CPU, GPU, hardware accelerators and peripherals with variable transaction sizes. This system has 4 requestors with the transaction sizes of 16 bytes, 32 bytes, 64 bytes and 128 bytes, respectively. The first requestor Req_1 represents a GPU with 128 byte cache line size, executing a Mediabench application *jpegdecode*. A video engine corresponding to the second requestor, Req_2, runs *mpeg2decode* and generates memory transactions of 64 bytes. The Mediabench application *epic* is executed by a processor denoted Req_3 with a cache-line size of 32 bytes. A synthetic memory trace is used by a CPU with a 16 byte cache-line size, resulting in read and write transactions of 16 bytes. This is requestor Req_4. The characterization of these memory traces is given by Table 3.2. Note that a requestor stops when the whole trace is executed. The TDM arbiter in the front-end of our Run-DMC allocates one slot per requestor and serves these requestors from Req_1 to Req_4 in descending order of their transaction sizes. In contrast, the semi-static approach uses memory patterns (i.e., static schedules of commands) to serve transactions with a fixed size. When it is used for variable transaction sizes, larger transactions have to be split into several pieces served by multiple patterns, while smaller transactions are directly served by the pattern, discarding the unneeded data. Therefore, we have to investigate the best transaction size of the patterns, i.e., pro-

viding the lowest execution/response times of all requestors or the maximum overall bandwidth. In this way, a fair comparison can be carried out between Run-DMC and the semi-static approach.

Table 3.2: Characterization of memory traffic with variable transaction sizes.

Traffic	*jpegdecode*	*mpeg2decode*	*epic*	*synthetic*
Size (bytes)	128	64	32	16
TransN	11385	37320	182957	990151
RRatio	63.1%	77.0%	69.7%	50.0%

3.5.3.1 *Execution Time*

The execution time of a transaction executed by Run-DMC starts either from the finishing time of the preceding transaction or its own arrival time, whichever is larger, according to Definition 7. Due to the command scheduling dependencies on the SDRAM banks, the measured WCET must be obtained when the transaction starts immediately when the preceding transaction finishes. In contrast, the execution time given by the semi-static approach [3] equals to the length of the pattern used by the transaction. Since the semi-static approach only uses static command scheduling patterns with a particular transaction size at run-time, we have done separate experiments by configuring patterns with four different transaction sizes at design time, respectively. Note that smaller transactions are executed using the pattern and the unneeded data is masked out, resulting in low data efficiency. While larger transactions are executed by continuously employing multiple patterns.

The measured maximum execution times of the variable transaction sizes are illustrated in Figure 3.9 for our dynamically-scheduled and the semi-static approaches, where DDR3-1600G SDRAM device is taken as an example. We can see that the measured ET has great differences for the semi-static approach using patterns with different transaction sizes. Since the semi-static approach only uses patterns with a particular size at run-time, we can observe from Figure 3.9 that the best pattern size is 128 bytes, such that the total (measured) execution time is minimized when the semi-static approach executes transactions with 16 bytes, 32 bytes, 64 bytes, and 128 bytes for DDR3-1600G. This best pattern size ensures that the semi-static approach achieves the smallest worst-case response time. Therefore, we can fairly compare Run-DMC with the semi-static approach. Experiments also show that the best pattern size for DDR3-800D and DDR3-2133K is 64 bytes and 128 bytes, respectively.

The purpose of this experiment is to find the best pattern size, such that Run-DMC can fairly compare its response time and bandwidth in the following sections rather than compare the measured execution time. The reason is that the definitions of ET are

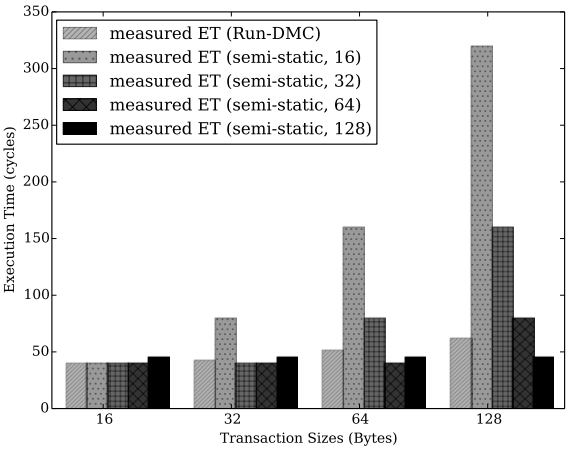


Figure 3.9: The maximum measured WCET of both Run-DMC and the semi-static approach [4] for DDR3-1600G SDRAM with variable transaction sizes.

different for these two approaches. We find from the experiments that the measured ET of Run-DMC is achieved when the preceding transaction is always the smallest 16-byte transaction, which uses only a single bank. Therefore, the measured (maximum) ET is caused for a transaction that has to reactivate the same bank. This reactivation takes long time. However, the semi-static approach executes 16-byte transactions with the pattern of 128 bytes, which interleaves over more banks. As a result, the reactivation time taken by Run-DMC for the new transaction can be pipelined in the pattern for the preceding 16-byte transaction with the semi-static approach, although the unneeded 112-byte data has to be discarded. In other words, Run-DMC computes the ET of a transaction by taking the reactivation time into account, while the semi-static approach puts this time to the preceding transaction instead of the current one.

3.5.3.2 *Bandwidth*

The bandwidth obtained by executing transactions with variable sizes can be computed based on the execution time and the corresponding transaction size (see Definition 10). For the semi-static approach applying for variable transaction sizes, the data efficiency has to be considered by the bandwidth. For example, to achieve the lowest execution time of transactions for DDR3-1600G SDRAM, the semi-static approach uses patterns with 128 bytes for all transaction sizes. As a result, the data efficiencies for transactions with 16-bytes, 32 bytes and 64 bytes are 12.5%, 25% and 50%, respectively. Figure 3.10 shows the average bandwidth of Run-DMC and the semi-static approach with different patterns. These results are computed based on the average ET obtained by different approaches. *We can see that Run-DMC can always provide more bandwidth than the semi-static approach using different patterns.* The reasons are that 1) Run-DMC efficiently

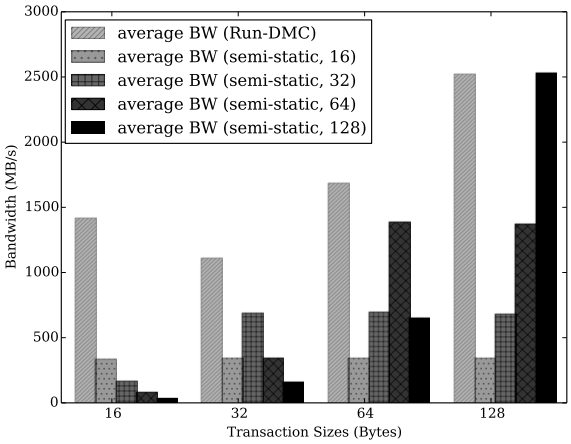


Figure 3.10: The average bandwidth (BW) of both Run-DMC and the semi-static approach [4] for DDR3-1600G SDRAM with variable transaction sizes.

deals with variable transaction sizes by dynamically exploiting the bank states, and 2) the static patterns used by the semi-static approach result in low data efficiency when applying for variable transaction sizes. Note that this observation also holds for other DDR3 SDRAMs.

3.5.3.3 *Response Time*

The response times of the four requestors in our experiments can be fairly compared between our dynamically-scheduled memory controller and the semi-static approach. As shown in Figure 3.11, *Run-DMC always gives lower measured maximum response times to requestors with variable sizes compared to the semi-static approach using patterns with different transaction sizes.* In particular, the latter with 16-byte patterns provides much larger response time, e.g., 587 cycles for read transactions with 128 bytes, and the results are too large to be included in Figure 3.11. The reasons that Run-DMC outperforms the semi-static approach for variable transaction sizes include 1) Run-DMC exploits the dynamism of the memory traffic that the semi-static approach cannot, i.e., variable-sized transactions to different banks, and 2) even using the best patterns to achieve the lowest response time, the semi-static approach is still not efficient for all the transaction sizes, e.g., discarding data for smaller sizes or being used multiple times for large ones. The former results in low data efficiency. The latter repeats the same pattern for a large transaction multiple times. With close-page policy, the same bank required by the large transaction is therefore reactivated/precharged multiple times, leading to longer execution time.

Our dynamically-scheduled memory controller provides significantly better average response time for the application traces, which are used in our experiments. Figure 3.12

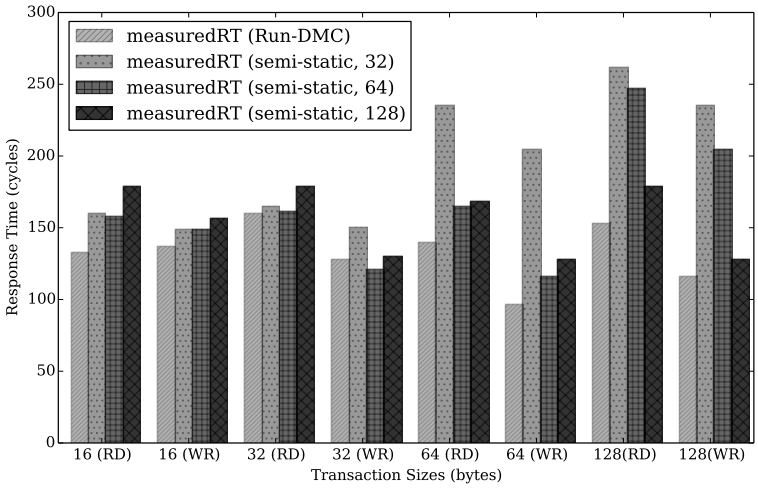


Figure 3.11: The measured response time (RT) of both Run-DMC and the semi-static approach [4] for DDR3-1600G SDRAM with variable transaction sizes.

presents the improvement of the average RT, which is gained by Run-DMC when comparing to the semi-static approach with the best patterns for DDR3-800D, DDR3-1600G and DDR3-2133K SDRAM memories, respectively. We can see that the average RT is improved from 16.2% for *jpegdecode_128* with DDR-800D to 79% for *synthetic_16* with DDR3-1600G. Again, it shows that smaller transaction sizes benefit more from our dynamically-scheduled memory controller.

3.6 SUMMARY

Design of real-time memory controller can use various mechanisms (e.g., priority-based arbitrations and re-ordering) to achieve different functionalities and performance. Moreover, the memory controller must be analyzable, i.e., being capable of providing the bounds on the worst-case response time and bandwidth. This chapter introduces the architecture and algorithms of our memory controller, Run-DMC, which is capable of efficiently dealing with diverse memory traffic with variable transaction sizes. Moreover, Run-DMC serves requestors with a novel work-conserving TDM arbiter in the front-end, and the back-end executes each transaction by dynamically scheduling commands to the required SDRAM banks in a pipelined manner according to a priority-based algorithm. The novel TDM arbiter is designed to achieve better/smaller worst-case response time by skipping idle slots for reducing the interference and configuring the service order of requestors in the descending of their transaction sizes, where the latter results in smaller execution time. Therefore, it can efficiently deal with the variable transaction sizes.

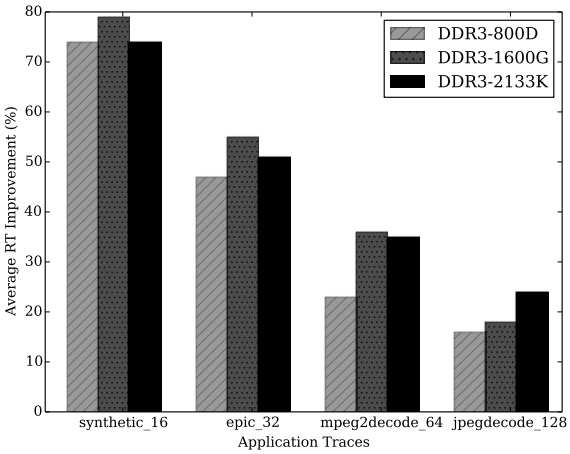


Figure 3.12: The average response time (RT) improvement gained by Run-DMC versus the semi-static approach [4] with the best patterns for DDR3 SDRAMs with variable transaction sizes.

Run-DMC is implemented as a cycle-accurate SystemC model, which is used to evaluate its performance in terms of measured maximum and average execution/response time as well as the bandwidth. The experimental results demonstrate that our dynamically-scheduled memory controller significantly outperforms a semi-static approach in the average case, while they are comparable in the worst-case according to the measured maximum/minimum response time/bandwidth for fixed transaction sizes. For variable transaction sizes, Run-DMC provides smaller measured maximum RT and larger average bandwidth than the semi-static approach using the best command scheduling patterns for different DDR3 SDRAMs. Moreover, smaller transaction sizes benefit more from the dynamically-scheduled memory controller in terms of average performance, which is beneficial for non-real-time applications.

4

FORMAL ANALYSIS OF RUN-DMC

The analysis of real-time (RT) memory controllers provides the worst-case execution/response time for each memory transaction and/or derives the worst-case (i.e., guaranteed) bandwidth to the memory requestors. These analysis results can be further integrated into a system-level analysis that ensures application requirements in terms of latency or throughput are satisfied [11]. However, the worst-case analysis of RT memory controllers is challenging because of 1) the interferences between different memory requestors, 2) the complex dependencies between SDRAM commands subject to the inter- and intra-bank timing constraints, and finally 3) the diverse memory traffic of arbitrarily mixed read and write transactions with variable sizes that require different SDRAM banks.

This chapter proposes a formal analysis of our dynamically-scheduled memory controller, Run-DMC, which was previously introduced in Chapter 3. Run-DMC can efficiently address the diverse memory traffic with variable transaction sizes. Its front-end uses a novel TDM arbiter to serve requestors, while the back-end executes each transaction via dynamically scheduling commands to the required banks according to their run-time states. The formal analysis in this chapter follows the same way used by existing analyses, which are based on analyzing the command scheduling dependencies and figuring out which ones dominate in the scheduling process and determine the worst-case results. The advantage of these formal analyses is that they are easy to use when the formal worst-case bounds are derived. However, the analysis of the complex scheduling dependencies is difficult. When applying for a new memory controller using different mechanisms, the analysis has to be repeated, which is time-consuming. Moreover, the analyzed worst-case results may be pessimistic, since the analysis has to use conservative assumptions, such that the complexity is manageable.

To overcome the shortcomings of formal analysis, we switch the effort from analyzing the complex command scheduling dependencies to modeling the memory controller and obtaining the worst-case results by analyzing the models with existing techniques. Toward this goal, Chapter 5 will later introduce a dataflow model of our Run-DMC, where commands are represented by the nodes and dependencies are captured by the edges between nodes in a dataflow graph. The analysis of dataflow model is easy to derive the minimum throughput of executing the graph, which is converted into the worst-case/minimum bandwidth (WCBW). However, obtaining the worst-case response

time (WCRT) is difficult due to the dynamism of executing the dataflow graph. Finally, Chapter 6 will give a timed automata (TA) model of our dynamically-scheduled memory controller. The behavior of the dynamic command scheduling can be precisely captured by the states and the transitions of the TA model. With model checking of the TA model, we derive the bounds of both WCBW and WCRT.

This chapter introduces a formal analysis of our Run-DMC, which consists of three steps. First, the command scheduling dependencies of an arbitrary transaction are formalized, followed a derivation of the worst-case initial bank states for the transactions, such that the worst-case execution time (WCET) in the back-end can be computed. The worst-case initial bank states are derived by scheduling commands for the preceding transactions as-late-as-possible. This results in the maximum scheduling times of commands for the current transaction subject to the constant timing constraints, leading to a bound on the WCET. Thirdly, with the bounded WCET of each transaction in the back-end, the WCBW is obtained based on Definition 10. Moreover, the WCRT of a transaction experienced in the front-end is therefore obtained by accumulating the WCET of the interfering transactions executed within their allocated slots and the WCET of the transaction itself. The formalization of the generic command scheduling dependencies and the corresponding WCET analysis are implemented in an open-source tool, named RTMemController [70]. Finally, the worst-case bounds given by the proposed formal analysis are experimentally validated. We also compare the worst-case results to those given by the semi-static approach [4], which is the only other memory controller supporting different memory configurations.

In the remainder of this chapter, Section 4.1 summarizes the related work of worst-case analysis of RT memory controllers. The formalization of the generic command scheduling dependencies is given in Section 4.2, followed by determining the worst-case initial bank states in Section 4.3. The WCET is bounded in Section 4.4 and a bound on the WCRT is derived in Section 4.5. The bound on the WCBW is given in Section 4.6 and it is based on the WCET. Section 4.7 introduces the open-source tool. Finally, the results are presented in Section 4.8, before the summary of this chapter in Section 4.9.

4.1 RELATED WORK

The analysis of real-time memory controllers is challenging because of the complex scheduling dependencies between commands of memory transactions. The scheduling dependencies are caused by the intra- and inter-bank timing constraints of SDRAM memories. To ease the analysis, static and semi-static memory controllers are designed, where the complex dependencies are resolved off-line when generating static command schedules of an application or transactions at design time. During run-time, commands are scheduled according to these static schedules. Therefore, the worst-case execution/response time and bandwidth can be statically calculated based on inspecting these command schedules. For example, an application-specific SDRAM memory controller [12]

uses a static command schedule for a particular application, and the total WCET of all the memory transactions is hence statically known. However, this SDRAM controller is not scalable to support other applications running in a different platform, e.g., a multi-core system. To overcome this limitation, semi-static memory controllers [4, 25, 37, 46, 84, 85, 88] use pre-computed short command schedules corresponding to transactions with a fixed size instead of an entire application. As a result, they can dynamically select the static schedules for particular transactions, such that different applications are supported. Since the number of these static schedules is limited, it is easy to determine the valid combination of schedules, which leads to the worst-case results. However, these semi-static memory controllers cannot efficiently execute transactions with variable sizes due to the limited number of schedules. The reason is that the variable-sized transactions access different sets of the SDRAM banks, which require more static schedules. However, these schedules will consume more hardware resources of the memory controller and also result in the complexity of worst-case analysis. This conflicts the whole point of the semi-static approach.

Dynamic command scheduling is promising to efficiently address transactions with variable sizes, where commands are generated and scheduled to the required sets of banks at run-time. As a result, it can dynamically exploit the SDRAM bank parallelism. However, its worst-case analysis is much more difficult than those of static and semi-static memory controllers. The formal analyses of dynamic command scheduling are based on analyzing the complex scheduling dependencies of commands. Therefore, conservative assumptions are employed by the state-of-the-art analyses to compute the time interval between any two commands. The assumptions include that 1) the switching from write to read is always assumed to be the interval between two *RD* and/or *WR* commands, and 2) the tFAW constraint dominates in the scheduling of each ACT command. For example, these consumptions have been used by [52, 55, 63, 107] to derive conservative worst-case results. Though these memory controllers schedule command dynamically, they only support a fixed transaction size, such that their analysis is simplified. Our dynamically-scheduled memory controller Run-DMC [69, 72] supports variable sizes directly and the formal analysis presented in this chapter will provide the worst-case results in terms of WCET, WCBW, and WCRT. The two assumptions mentioned above also apply to our analysis technique. However, the switching timing constraint from write to read is only used to compute the scheduling time of the first *RD* command of a transaction and not for all the *RD* commands. Similarly, when the *tFAW* constraint applies to an *ACT* command of the transaction, it cannot dominate in the scheduling of the next consecutive *ACT* command. These are captured by our analysis technique. Moreover, existing analyses of real-time memory controllers always assume scheduling collisions on the command bus. However, our analysis shows that this is not always true. Therefore, we can derive better worst-case results.

4.2 FORMALIZATION OF DYNAMIC COMMAND SCHEDULING

In this section, we introduce standard notation and definitions to formalize the timing behavior of the back-end architecture and the dynamic command scheduling of transactions, as specified by Algorithm 2 in Section 3.3.2. As introduced in Section 3.3.1, a transaction is translated into a series of *BI bank accesses*. Each bank access activates a bank, and then reads or writes *BC* times, the last time with auto-precharge. A command can only be scheduled and executed at its scheduling time when the timing constraints from previous commands are satisfied. Timing constraints therefore result in scheduling dependencies. A bank access is a natural self-contained group of commands, and each transaction is made up of one or more bank accesses. As discussed in Section 2.3.2, the command scheduling dependencies of two successively accessed banks b_j and b_{j+1} are depicted by Figure 2.8. Our analysis in this section is based on the scheduling dependencies of an individual transaction T_i that generates BI_i successive bank accesses. The notation used in this section is summarized in Table 4.1. Note that the formalization of dynamic command scheduling and the analysis in this chapter are the novelties and not the mathematical analysis techniques.

4.2.1 *Formalization*

In dynamic command scheduling, the order of command execution is decided at run time on the basis of the timing constraints between commands. In addition, the command scheduling in Algorithm 2 specifies that 1) *RD*/*WR* is prioritized over *ACT* commands of different banks, resulting in that an *ACT* can be blocked; 2) Commands are scheduled in ascending order of their targeted banks, and 3) the same kind of commands (i.e., *RD*/*WR* or *ACT*) of a later transaction cannot be scheduled earlier, such that transactions are served in FCFS manner. Recall that the commands to the same bank are enqueued in the FIFO queue, where the *ACT* command is enqueued before the *RD*/*WR* commands for the same transaction. We therefore obtain the command scheduling dependencies of a transaction by extending the dependencies between commands of successively accessed banks, as depicted in Figure 2.8. We analyze the execution time of a transaction by computing the actual scheduling time of commands under our dynamic scheduling algorithm. We later compute the worst-case execution time in Section 4.4.

When an arbitrary transaction T_i ($\forall i \geq 0$) arrives at the back-end with the arrival time defined by Definition 3, it is executed by scheduling commands to a number of banks. Note that T_{-1} is a transaction assumed to be finished long time ago, i.e., $t_f(T_{-1}) = -\infty$. T_{-1} is only used to provide the initial bank states for analyzing the following arbitrary transactions. We assume T_i uses BI_i and BC_i, and the starting bank is b_j. j is the number of the first bank access of T_i and it is a function of i, as given by Eq. (2.1). It is the total number of bank accesses by previous transactions. Expanding Figure 2.8 to an entire transaction T_i, Figure 4.1 illustrates the scheduling dependencies between all its

Table 4.1: Summary of notation.

Variables	*Descriptions*
i	The number of an arbitrary transaction arrived at the back-end. $i \geq 0$.
T_i	The i^{th} transaction received by the back-end
$S(T_i)$	The size of transaction T_i
j(i)	The first bank access number for the current transaction T_i. $j(i) \geq 0$ is defined by Eq. (2.1).
j	The shorthand for j(i)
BI_i, BC_i	The bank interleaving number (BI) and burst count (BC) of T_i
$b_{j(i)}$	The bank number that is targeted by the j^{th} bank access, which is also the starting bank of T_i. $b_j \in [0, 7]$ is one of the 8 banks in DDR3 SDRAMs.
b_j	Shorthand for $b_{j(i)}$
ACT_j	The ACT command of the j^{th} bank access
$t(ACT_j)$	The scheduling time of ACT_j
$C(j)$	The delay in scheduling ACT_j due to a collision; and it is either 1 or 0 cycle depending on whether the collision exists or not.
RW_j^k	The k^{th} ($\forall k \in [0, BC_i - 1]$) RD or WR command of the j^{th} bank access
$t(RW_j^k)$	The scheduling time of RW_j^k
PRE_j	The PRE command for the j^{th} bank access
$t(PRE_j)$	The scheduling time of PRE_j
$t_s(T_i)$	The starting time of T_i in the back-end
$\hat{t}_s(T_i)$	The worst-case starting time of T_i in the back-end
$t_f(T_i)$	The finishing time of T_i in the back-end
$\hat{t}_f(T_i)$	The worst-case finishing time of T_i in the back-end
$t_{ET}(T_i)$	The execution time of T_i in the back-end
l	Used to index the banks of a transaction T_i and $\forall l \in [0, BI_i - 1]$
k	Used to index the bursts of a bank for T_i, and $\forall k \in [0, BC_i - 1]$

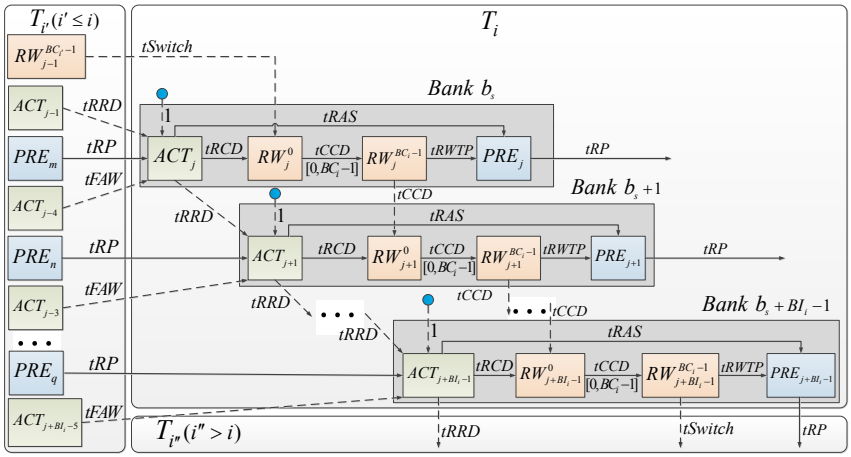


Figure 4.1: The timing dependencies of command scheduling for transaction T_i.

commands. The command scheduling for T_i depends on zero or more previous transaction(s) $T_{i'}$. For $\forall l \in [0, BI_i - 1]$, the $(j + l)^{th}$ bank access comprises ACT_{j+l} and several *RD* or *WR* commands to bank b_{j+l}. The *RD* or *WR* commands are denoted by RW^k_{j+l}, where $\forall k \in [0, BC_i - 1]$. Moreover, an auto-precharge PRE_{j+l} is issued after the access of bank b_{j+l}, and it is specified by an auto-precharge flag issued together with $RW^{BC_i-1}_{j+l}$. For $BI_i > 4$, the scheduling of some *ACT* commands also depends on the previous *ACT* commands of the current transaction T_i because of the four-activate window (*tFAW*).

For T_i, Eq. (4.1) computes the scheduling time of ACT_{j+l} where $m = max_{k<j}\{k|b_k = b_{j+l}\}$ is the previous bank access to bank b_{j+l}. The *max* function in Eq. (4.1) guarantees that all the timing constraints for scheduling ACT_{j+l} are satisfied. In addition, the scheduling time of ACT_{j+l} is after T_i arrives. In case of a command scheduling collision where ACT_{j+l} is blocked by a *RD* or *WR* command, $C(j+l)$ is equal to 1 and 0 otherwise. Similarly, the scheduling time of RW^k_{j+l} is given by Eq. (4.2) and (4.3). Eq. (4.2) provides the scheduling time of the first *RD* or *WR* command of T_i to bank b_{j+l}. It depends on $t(RW^{BC_i-1}_{j+l-1})$, which is the scheduling time of the last *RD* or *WR* to b_{j+l-1}, and the scheduling time of ACT_{j+l}. Note that for $l = 0$, $t(RW^{BC_i-1}_{j-1})$ is defined as the finishing time of T_{i-1}. The scheduling time of the remaining *RD* or *WR* commands ($k \in [1, BC_i - 1]$) to bank b_{j+l} only depend on the previous *RD* or *WR* command, and is given by Eq. (4.3). Finally, the precharging time of the auto-precharge for bank b_{j+l} is given by Eq. (4.4). This is the time at which the precharge actually happens, although it was issued earlier as an auto-precharge flag appended to the last *RD* or *WR* command to the same bank. We define the finish time of the initial transaction as $t_f(T_{-1}) = -\infty$, such that the *ACT* of the first

transaction T_0 can be scheduled at time 0. These equations have been implemented in our open source tool [70] to provide the scheduling time of commands.

$$t(ACT_{j+l}) = max\{t(ACT_{j+l-1}) + tRRD, t(PRE_m) + tRP, \\ t(ACT_{j+l-4}) + tFAW, t_a(T_i)\} + C(j+l) \tag{4.1}$$

$$t(RW^0_{j+l}) = max\{t(RW^{BC_i-1}_{j+l-1}) + tSwitch, t(ACT_{j+l}) + tRCD\} \tag{4.2}$$

$$t(RW^k_{j+l}) = t(RW^0_{j+l}) + k \times tCCD \tag{4.3}$$

$$t(PRE_{j+l}) = max\{t(ACT_{j+l}) + tRAS, t(RW^{BC_i-1}_{j+l}) + tRWTP\} \tag{4.4}$$

Based on Eq. (4.1) to (4.4), it is possible to determine the finishing time of T_i by only looking at the finishing time of T_{i-1} and the scheduling time of its *ACT* commands. As shown in Figure 4.1, only the first *RD* or *WR* commands and the *ACT* to each bank have dependencies on previous transactions. The other *RD* or *WR* commands can be scheduled with the dependencies directly or indirectly originating from those commands. *Intuitively, the finishing time of T_i is determined only by the scheduling time of all its ACT commands, the finishing time of the previous transaction and JEDEC-defined timing constraints.* This intuition is formalized by Lemma 1 and the proof is included in Appendix A.1.

Lemma 1. *For $\forall i \geq 0$ and $t_f(T_{-1}) = -\infty$,*

$$t_f(T_i) = \max_{0 \leq l \leq BI_i - 1} \{t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD, \\ t(ACT_{j+l}) + tRCD + \lceil (BI_i - l) \times BC_i - 1 \rceil \times tCCD\}$$

4.3 WORST-CASE INITIAL BANK STATES

The command scheduling for the current transaction T_i is highly dependent on the initial bank states resulting from when the commands of the previous transactions (e.g., T_{i-1} and T_{i-2}) were scheduled. Intuitively, given that the minimum starting time of T_i is fixed by the finishing time of T_{i-1}, the worst-case finishing time of T_i occurs when all the commands of T_{i-1} were scheduled as late as possible (*ALAP*), because this maximizes the timing dependencies. In this section, we formalize the *ALAP* scheduling of T_{i-1}, which defines the worst-case initial bank states for T_i. Later Section 4.4.2 computes the worst-case finishing time of T_i based on these worst-case initial bank states. The WCET of T_i is finally given in Section 4.4.3.

4.3.1 *Worst-Case Starting Time*

From Definition 7, it follows that the execution time, $t_{ET}(T_i)$, is maximized if *the starting time is minimum while the finishing time is maximum.* According to Definition 6, the starting time $t_s(T_i)$ is determined by its arrival time $t_a(T_i)$ and the finishing time $t_f(T_{i-1})$ of the previous transaction T_{i-1}. In the worst-case situation, T_i has arrived before the finishing of T_{i-1}, such that the commands for T_i have to wait longer time for their timing constraints to be satisfied. Therefore, the worst-case starting time of T_i is only one cycle after the finishing time of T_{i-1} and is given by Eq. (4.5).

$$\hat{t}_s(T_i) = t_f(T_{i-1}) + 1 = t(RW_{j-1}^{BC_{i-1}-1}) + 1 \tag{4.5}$$

To derive the maximum finishing time of T_i, denoted by $\hat{t}_f(T_i)$, the scheduling time of its *ACT* commands should be maximized according to Lemma 1. Eq. (4.1) indicates that the scheduling of an *ACT* command depends on the previous *PRE* to the same bank, the previous *ACT* commands and the possible collision caused by a *RD* or *WR* command. Therefore, the worst-case finishing time of T_i is achieved by maximizing the scheduling time of the previous *PRE* and *ACT* commands as well as assuming there is always a command collision for every *ACT* command.

The preceding transaction T_{i-1} has many possibilities, since it is not statically known. For example, it may be a read or a write with variable sizes and requiring different sets of banks, and its commands were scheduled based on its initial bank states that were determined by even earlier transactions. Therefore, it is hard to statically know which T_{i-1} provides the worst-case initial bank states for T_i. However, the worst-case starting time given by Eq. (4.5) defines the finishing time $t(RW_{j-1}^{BC_{i-1}-1})$ of T_{i-1} and *we can conservatively assume all the commands of T_{i-1} were scheduled as late as possible (ALAP) with respect to the fixed finishing time of T_{i-1}, subject to the timing constraints of the memory.* This *ALAP* scheduling ensures the latest (i.e., maximum) possible scheduling time of the previous commands, which are the worst-case initial bank states for T_i.

4.3.2 *ALAP Scheduling*

This section shows how to formalize the *ALAP* scheduling by computing the worst-case (latest possible) scheduling time of all the commands for the previous transaction. According to *ALAP* scheduling, the scheduling time of the previous *ACT*, *RD* or *WR* commands and *PRE* can be obtained by calculating backwards from the scheduling time of the last *RD* or *WR* command at $t(RW_{j-1}^{BC_{i-1}-1})$. Specifically, the time between any successive commands must be minimal while satisfying the timing constraints, thereby ensuring an *ALAP* schedule of the previous commands. Therefore, the *minimum time interval* between any two commands is significant to formalize the *ALAP* scheduling. Recall that T_{i-1} has BI_{i-1} and BC_{i-1}. First, as stated in Table 2.1, the minimum time be-

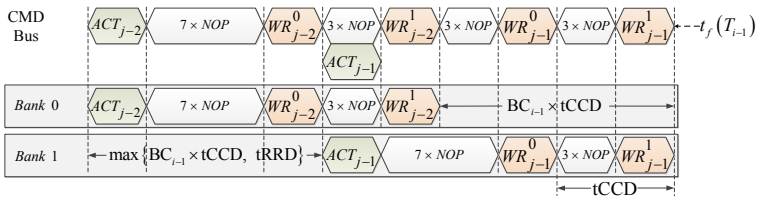


Figure 4.2: An example of As-Late-As-Possible (*ALAP*) scheduling with DDR3-1600G SDRAM for T_i which has $BI_i = 4$ and $BC_i = 2$. The previous transaction T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. The starting bank for both T_{i-1} and T_i is *Bank* 0.

tween two *RD* or *WR* commands is *tCCD*. Since *RD* or *WR* commands targeting the same bank are scheduled sequentially, the minimum time between the first *RD* or *WR* commands to consecutive banks is $BC_{i-1} \times tCCD$. Second, an *ACT* command is followed by a *RD* or *WR* command to the same bank, and their minimum time interval is *tRCD* (see Table 2.1). This implies that an ACT command must be scheduled at least *tRCD* cycles before the first *RD* or *WR* command to the same bank. To calculate backwards, the time interval between two successive ACT commands to different banks has to be at least $BC_{i-1} \times tCCD$. In addition, Table 2.1 also states that the minimum time between two *ACT* commands to different banks is *tRRD*. Hence, the minimum time interval between two successive *ACT* commands to different banks without violating any timing constraints is $max\{tRRD, BC_{i-1} \times tCCD\}$.

Figure 4.2 illustrates an example of *ALAP* scheduling for a DDR3-1600G SDRAM. This example assumes the current transaction T_i and the previous write transaction T_{i-1} have the same starting bank *Bank* 0. T_i has $BI_i = 4$ and $BC_i = 2$, while T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. With the fixed finishing time $t(RW^1_{j-1})$ of T_{i-1}, the scheduling time of all the previous commands is computed backwards with the minimum time interval between them. Figure 4.2 shows the *ALAP* scheduling of the previous commands for T_{i-1} to *Banks* 0 and 1. In this way, some *ACT* commands have the same scheduling time as some *WR* commands, which indicates command scheduling collisions. However, we conservatively ignore these collisions so that larger scheduling time of the previous *ACT* and *WR* commands for T_{i-1} is achieved, which provide the initial bank states for the new transaction T_i.

Next, *ALAP* scheduling is formalized to provide the scheduling time of previous commands. First, the preceding transaction T_{i-1} must have banks in common with T_i, because the reactivation of a bank for T_i needs more time if it was accessed by T_{i-1}. To obtain larger finishing time, the starting bank b_j of T_i must have been accessed by T_{i-1}, and the last bank b_{j-1} of T_{i-1} is also required by T_i. Since the banks of a transaction are accessed in ascending order according to Algorithm 2, there must be $b_j \leq b_{j-1}$. As a result, the set of common banks is $[b_j, b_{j-1}]$. We introduce the short hand notation $b_{com} = b_{j-1} - b_j$ and the number of common banks is hence $b_{com} + 1$. For example, the set of common banks between T_{i-1} and T_i in Figure 4.2 is $[0, 1]$, and the number of com-

mon banks is 2. With the minimum time interval between commands, for $\forall l \in [0, b_{com}]$ and $\forall k \in [0, BC_{i-1} - 1]$, the scheduling time of the *RD* or *WR* commands to a common bank $b_j + l$ is given by Eq. (4.6). Note that $\hat{t}_s(T_i) - 1$ is the finishing time of T_{i-1} according to Eq. (4.5). Eq. (4.6) can be used to conservatively determine the *ALAP* scheduling time of all *RD* or *WR* commands of T_{i-1}.

$$\begin{aligned}\hat{t}(RW^{k}_{j-1-(b_{com}-l)}) =& \hat{t}_s(T_i) - 1 - (BC_{i-1} - 1 - k) \times tCCD \\ & - (b_{com} - l) \times BC_{i-1} \times tCCD\end{aligned} \tag{4.6}$$

Given a finishing time of T_{i-1}, the scheduling time of its last *ACT* command is obtained since the minimum time interval between an *ACT* command and the first *RD* or *WR* command to the same bank is *tRCD* (see Table 2.1). Thus, with the minimum time interval between *ACT* commands, the scheduling time of the previous *ACT* commands is calculated by Eq. (4.7). Based on Eq. (4.4), the time of the previous *PRE* is obtained by using the worst-case scheduling time for *RD* or *WR* and *ACT* commands from Eq. (4.6) and (4.7), respectively. It is given by Eq. (4.8) based on two observations of the timing constraints in JEDEC DDR3 standard [53], namely: 1) *tRWTP* is larger for a write transaction than for a read transaction, and 2) there is $tRWTP > tRAS - tRCD$ for a write transaction. In a word, the second term in the *max* of Eq. (4.8) dominates in the worst-case. *Therefore, the worst-case initial states for T_i is that T_{i-1} is write rather than read*, and Eq. (4.8) is further simplified.

$$\begin{aligned}\hat{t}(ACT_{j-1-(b_{com}-l)}) =& \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ & - (b_{com} - l) \times max\{tRRD, BC_{i-1} \times tCCD\}\end{aligned} \tag{4.7}$$

$$\begin{aligned}&\hat{t}(PRE_{j-1-(b_{com}-l)}) \\ &= max\{\hat{t}(ACT_{j-1-(b_{com}-l)}) + tRAS, \hat{t}(RW^{BC_{i-1}-1}_{j-1-(b_{com}-l)}) + tRWTP\} \\ &= \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD\end{aligned} \tag{4.8}$$

Note that Eq. (4.6), (4.7) and (4.8) formalize *ALAP* command scheduling for T_{i-1}, leading to the worst-case initial bank states for T_i. This formalization is parameterized to BI_{i-1} and BC_{i-1} used by T_{i-1}. If more is known about T_{i-1}, e.g., its accessed banks, the *ALAP* scheduling can be specialized to obtain better analysis results. For example, bank privatization is employed by the memory controllers in [24–26, 52, 56, 63, 88, 107] for different requestors. We leave the exploitation of this static knowledge to obtain a tighter WCET as future work.

4.4 WORST-CASE EXECUTION TIME

Based on the worst-case initial bank states given by the *ALAP* scheduling in Section 4.3, we can compute the maximum scheduling time of commands for T_i, resulting in the

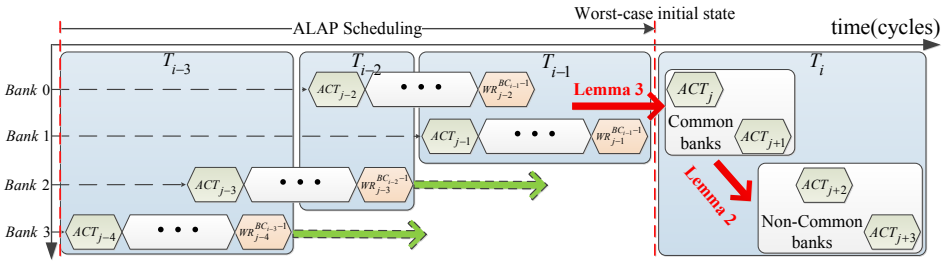


Figure 4.3: An illustration of the *ALAP* scheduling that provides worst-case initial bank states for the current transaction T_i.

worst-case finishing time $\hat{t}_f(T_i)$. Before deriving $\hat{t}_f(T_i)$, we firstly prove that the off-line *ALAP* scheduling of the preceding transaction T_{i-1} indeed guarantees a conservative $\hat{t}_f(T_i)$. This is achieved by introducing Lemma 2 and Lemma 3 that demonstrate the maximum scheduling times of the *ACT* commands for T_i only rely on the *ALAP* command scheduling of T_{i-1}. Lemma 4 gives $\hat{t}_f(T_i)$ that is computed based on Lemma 1. After deriving the worst-case finishing time, the WCET defined as the time between the worst-case starting time and the worst-case finishing time is computed in Section 4.4.3. A generic parameterized WCET is first derived (see Theorem 1), followed by two interesting special cases, fixed transaction size and variable transaction sizes, respectively, which are given by Corollary 1 and Corollary 2.

4.4.1 *Conservative $\hat{t}_f(T_i)$ Based on ALAP Scheduling*

Intuitively, *ALAP* scheduling of commands for the previous write transaction T_{i-1} provides the worst-case initial bank states for T_i. However, the actual command scheduling for T_i may not only depend on T_{i-1} but also earlier transactions. Figure 4.3 shows an example where T_i uses 4 banks from *Bank* 0 to *Bank* 3. T_{i-1} has the common banks *Bank* 0 and *Bank* 1 with T_i. T_{i-2} and T_{i-3} accessed *Bank* 2 and *Bank* 3, respectively. We can see that the *ACT* commands for T_i to the common banks *Bank* 0 and *Bank* 1 have to follow the constraints from the previous *WR* commands of T_{i-1}. For the non-common banks *Bank* 2 and *Bank* 3, the *ACT* commands of T_i may be scheduled according to the *WR* commands of earlier transactions T_{i-2} and T_{i-3} to the same bank. Moreover, *tFAW* must be satisfied between ACT_j of T_i and ACT_{j-4} of T_{i-3}.

We proceed by formally proving that the *ALAP* command scheduling of T_{i-1} guarantees a conservative $\hat{t}_f(T_i)$, even though earlier transactions (e.g., T_{i-2}, T_{i-3}) may actually have constraints that dominate in the command scheduling for T_i. Note that the *ALAP* command scheduling of T_{i-1} provides conservative worst-case initial bank states for T_i, and it is therefore not necessary to consider these earlier transactions in the worst-case analysis. This proof is achieved by three steps. As shown in Figure 4.3, the first step is

given by Lemma 2 stating that the scheduling of *ACT* commands of T_i to *non-common banks* with T_{i-1} is only determined by the *ACT* commands to the common banks in the worst-case. This indicates that the constraints from earlier transactions T_{i-2} and T_{i-3}, as depicted by the green arrows in Figure 4.3, cannot dominate in the scheduling of these *ACT* commands when using the *ALAP* command scheduling of T_{i-1}. The second step given by Lemma 3 guarantees that the *ACT* commands of T_i to *common banks* with T_{i-1} can be scheduled only dependent on the *ALAP* scheduling of commands of T_{i-1}, as shown in Figure 4.3. As a result of Lemma 2 and Lemma 3, the scheduling of *ACT* commands of T_i only depends on T_{i-1} in the worst-case. Finally, the third step computes the $\hat{t}_f(T_i)$ based on the *ALAP* command scheduling of T_{i-1} in Lemma 4. All the proofs are included in Appendix A.

The idea of Lemma 2 and Lemma 3 is to eliminate all the dependencies that cannot dominate in the scheduling of the *ACT* commands of T_i according to the worst-case initial bank states formalized by the *ALAP* scheduling. Lemma 2 states that the worst-case scheduling times of the *ACT* command to any non-common bank b_{j+l} ($\forall l \in (b_{com}, BI_i - 1]$) are determined by $\hat{t}(ACT_{j+b_{com}})$, which is the worst-case scheduling time of the *ACT* command to the last common bank $b_{j+b_{com}}$. Note that $\hat{t}(ACT_{j+b_{com}})$ essentially depends on the previous transaction T_{i-1}. We can observe that a smaller b_{com} provides larger $\hat{t}(ACT_{j+l})$ for the particular non-common bank b_{j+l} (i.e., fixed l). Since $b_{com} = b_{j-1} - b_j$, the smallest b_{com} is achieved only if b_{j-1} is as close as possible to b_j, implying that T_i starts with a bank b_j that is very close to the finishing bank b_{j-1} of T_{i-1}. Note that this gap is determined by the size of T_{i-1} or T_i, whichever is smaller. As a result, $b_{com} = \min\{BI_{i-1}, BI_i\} - 1$ leads to the worst-case scheduling times of these *ACT* commands of T_i to non-common banks.

Lemma 2. *For* $\forall l \in (b_{com}, BI_i - 1]$,

$$\hat{t}(ACT_{j+l}) = \hat{t}(ACT_{j+b_{com}}) + [l - b_{com}] \times tRRD + \sum_{l'=b_{com}+1}^{l} C(j + l')$$

Lemma 3 states that the worst-case scheduling of an *ACT* command to a common bank b_{j+l} ($\forall l \in [0, b_{com}]$) is either dominated by $\hat{t}(ACT_{j-1})$ (l=0) or the *ALAP* scheduling time of the *PRE* commands to the common banks of T_{i-1}. Note that ACT_{j-1} is the last *ACT* command of T_{i-1}.

Lemma 3. *For* $\forall l \in [0, b_{com}]$,

$$\hat{t}(ACT_{j+l}) = max\{\hat{t}(ACT_{j+l-1}) + tRRD, \hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP\} + C(j + l)$$

From Lemma 2 and Lemma 3, we can therefore conclude that *all the ACT commands of* T_i *are scheduled based on the ALAP scheduling of commands for* T_{i-1} *in the worst-case.*

4.4.2 *Worst-Case Finishing Time*

We proceed by deriving the worst-case finishing time based on the worst-case initial bank states provided by the *ALAP* scheduling. Lemma 1 states that the finishing time of T_i is determined by the finishing time of the previous transaction T_{i-1} and the scheduling time $t(ACT_{j+l})$ ($\forall l \in [0, BI_i - 1]$) of each *ACT* command for T_i. Therefore, the worst-case finishing time $\hat{t}_f(T_i)$ is obtained by using $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$ and $\hat{t}(ACT_{j+l})$ that is obtained by substituting the *ALAP* formalization into Lemma 2 and Lemma 3.

Lemma 3 shows that $\hat{t}(ACT_{j+l})$ is determined by either the scheduling time $\hat{t}(ACT_{j+l-1})$ of the previous *ACT* command or the last precharge time $\hat{t}(PRE_{j-1-(b_{com}-l)})$ to the same bank, where $l \in [0, b_{com}]$. $\hat{t}(PRE_{j-1-(b_{com}-l)})$ is given by Eq. (4.8) according to the *ALAP* command scheduling for the previous write transaction T_{i-1}. Moreover, Lemma 2 shows that the scheduling time $\hat{t}(ACT_{j+l})(l \in (b_{com}, BI_i - 1])$ of *ACT* commands to non-common banks is determined by $\hat{t}(ACT_{j+b_{com}})$, which is the scheduling time of the *ACT* to the last common bank and can be computed with Lemma 3. As a result, $\hat{t}(ACT_{j+l})$ can be obtained by iteratively using Eq. (4.8), Lemma 2 and Lemma 3. We proceed by introducing Lemma 4, which gives the worst-case finishing time of T_i. The proof is presented in Appendix A.4. Intuitively, the worst-case finishing time of T_i is the worst-case starting time of T_i plus the maximum of all relevant timing dependencies (assuming an *ALAP* schedule for T_{i-1}).

Lemma 4. *For* $\forall i \geq 0, \forall l' \in [0, b_{com}]$ *and* $\forall l \in [l', BI_i - 1]$, $b_{com} = b_{j-1} - b_j$,

$$
\begin{aligned}
\hat{t}_f(T_i) = \hat{t}_s(T_i) - 1 + \max\{ & (l+1) \times tRRD - (BC_{i-1} - 1) \times tCCD \\
& + \left[(BI_i - l) \times BC_i - 1\right] \times tCCD + \sum_{h=0}^{l} C(j+h), \\
& tRWTP + tRP + tRCD - (b_{com} - l') \times BC_{i-1} \times tCCD \\
& + (l - l') \times tRRD + \left[(BI_i - l) \times BC_i - 1\right] \times tCCD \\
& + \sum_{h=l'}^{l} C(j+h), \\
& tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned}
$$

4.4.3 *Generic Worst-Case Execution Time*

According to Definition 7, the WCET is the difference between the worst-case starting time and the worst-case finishing time, which are both included in Lemma 4. Therefore, the WCET is obtained by rewriting Lemma 4. We observe that the expressions in the max{} of Lemma 4 either linearly increase or decrease with l and l'. As a result, these expressions can be simplified to give the worst-case finishing time $\hat{t}_f(T_i)$ and hence the WCET $\hat{t}_{ET}(T_i)$. We proceed by introducing Theorem 1, which shows that $\hat{t}_{ET}(T_i)$ is

only determined by the JEDEC DDR3 timing constraints [53], and the sizes of T_i and T_{i-1} via (BI_{i-1}, BC_{i-1}) and (BI_i, BC_i) according to the chosen memory map configurations. Therefore, Theorem 1 provides a WCET parameterized by the sizes of T_i and T_{i-1}. The proof of Theorem 1 is presented in Appendix A.5.

Theorem 1. *(Generic Worst-case execution time) For* $\forall i \geq 0$,

$$
\begin{aligned}
\hat{t}_{ET}(T_i) = \max\{&(BC_i - BC_{i-1}) \times tCCD + BI_i \times (tRRD + 1),\\
&tRWTP + tRP + tRCD\\
&+ \left[BI_i \times BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}\right] \times tCCD + 1,\\
&tRWTP + tRP + tRCD\\
&+ \left[(BI_i - (\min\{BI_{i-1}, BI_i\} - 1)) \times BC_i - 1\right] \times tCCD + 1,\\
&tRWTP + tRP + tRCD + (BI_i - 1) \times (tRRD + 1) + 1\\
&+ \left[BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}\right] \times tCCD,\\
&tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD\\
&+ \left[BI_i - \min\{BI_{i-1}, BI_i\}\right] \times (tRRD + 1) + 1,\\
&tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned}
$$

In general systems with variable transaction sizes, the specific size of the previous transaction that leads to WCET is unknown. We have found that the smallest previous transaction size must be assumed to derive a conservative WCET. This is later captured by Corollary 1. However, in the special case of the TDM arbitration presented in Section 3.2.2, the previous transaction size is known in the worst-case due to the static mapping of requestors to TDM slots. Therefore, less pessimistic WCET is obtained based on the known size of the previous transaction. A special case is that a system has a single fixed transaction size, such as 64-byte cache lines. As a result, the previous transaction size is statically known. The WCET for this special case is given by Corollary 2 in the next section. Note that the analysis of these two special cases only needs to instantiate Theorem 1 that is generic to any preceding and current transaction sizes.

Theorem 1 defines that $\hat{t}_{ET}(T_i)$ is parameterized by the sizes of T_i and T_{i-1}. We can observe that $\hat{t}_{ET}(T_i)$ increases when BI_{i-1} and BC_{i-1} decrease. By taking both of them to be 1, i.e., T_{i-1} is the smallest transaction, we obtain Corollary 1, which is conservative for any (unknown) preceding transaction. Intuitively, T_i experiences the WCET when the previous transaction is a small write that has only one burst to the starting bank of T_i. Moreover, it is not necessary to assume a collision for the first *ACT* command of T_i. The reason is that the finishing bank of T_{i-1} is the starting bank of T_i, and no *WR* commands of T_{i-1} collide with the first *ACT* of T_i. Therefore, $\hat{t}_{ET}(T_i)$ given by Corollary 1 is tighter than Theorem 1.

Corollary 1. *(ANALYTICAL WCET FOR VARIABLE TRANSACTION SIZES)*
For $\forall i \geq 0$,

$$\begin{aligned}\hat{t}_{ET}(T_i) = \max\{&tRWTP + tRP + tRCD + (BI_i \times BC_i - 1) \times tCCD,\\ &tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD\\ &+ (BI_i - 1) \times (tRRD + 1)\}\end{aligned}$$

Another common situation is that all transactions have the same size. For example, a homogeneous multi-core system may have a single memory transaction size, since the cache-line size of all the cores is the same. Transactions with the same size use the same BI and BC. So, $BI_{i-1} = BI_i = BI$ and $BC_{i-1} = BC_i = BC$. According to Theorem 1, we can derive Corollary 2 that provides the WCET to transactions with the same size. The intuition of Corollary 2 is that a transaction suffers the WCET when its previous transaction is a write that accessed the same set of banks.

Corollary 2. *(ANALYTICAL WCET FOR FIXED TRANSACTION SIZE)*
For $\forall i \geq 0$, $BI_{i-1} = BI_i = BI$ *and* $BC_{i-1} = BC_i = BC$,

$$\begin{aligned}\hat{t}_{ET}(T_i) = \max\{&tRWTP + tRP + tRCD + (BC - 1) \times tCCD + 1,\\ &tRWTP + tRP + tRCD + (BC - 1) \times tCCD\\ &+ (BI - 1) \times (tRRD + 1 - BC \times tCCD) + 1,\\ &tSwitch + (BI \times BC - 1) \times tCCD\}\end{aligned}$$

The WCET given by Corollary 1 and Corollary 2 for variable and fixed transaction sizes are parameterized with BI and BC. Many existing real-time memory controllers [24–26, 52, 56, 63, 107] execute transactions with a single data burst, i.e., $BI = BC = 1$. We explicitly provide the analytical WCET for this particular case by introducing Corollary 3. It can be obtained by using either Corollary 1 or Corollary 2 with $BI = BC = 1$. However, Corollary 1 is used to derive Corollary 3, because the collision assumption to the first *ACT* command is not needed in this case. The reason has been discussed when deriving Corollary 1.

Corollary 3. *(ANALYTICAL WCET FOR A SINGLE DATA BURST)*
For $\forall i \geq 0$, $BI_{i-1} = BI_i = 1$ *and* $BC_{i-1} = BC_i = 1$,

$$\hat{t}_{ET}(T_i) = tRWTP + tRP + tRCD$$

4.4.4 *Scheduled Worst-Case Execution Time*

The analytical WCET given by Corollaries 1 and 2 have the benefit of being simple equations that bound the WCET by just inserting the timings of the particular memory device and the chosen memory map configuration for a transaction. However, they are somewhat pessimistic, since they conservatively assume that there is a command

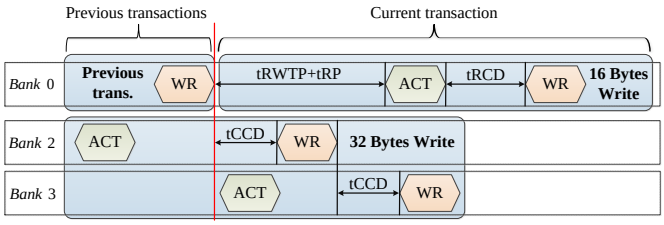


Figure 4.4: An example illustrating that the actual execution time of a larger transaction (32 Bytes write) can be less than that of a smaller transaction (16 Bytes write).

collision for every *ACT* command. Here, we present a second approach that builds on the presented formalism and ALAP schedule to overcome this limitation and derive a tighter bound.

The idea is to derive the worst-case initial bank state for a transaction based on the ALAP schedule as presented in Section 4.3.2, followed by actually scheduling the commands of the transaction off-line. *This has the advantage of only accounting for the actual number of command collisions and knowing exactly how many cycles the WCET increases due to these collisions.* The drawback of the approach is that it is no longer a simple equation, but requires a software implementation of the scheduling algorithm. To this end, the formalization of the timing behavior of the proposed scheduling algorithm, previously presented in Section 4.2, has been implemented as an open-source off-line scheduling tool [70]. For the remainder of this article, we will refer to this approach as the *scheduled WCET* and the bounds obtained from Corollaries 1 and 2 as the *analytical WCET*. Both of them can be obtained from our open-source tool [70].

4.4.5 *Monotonicity of Worst-Case Execution Time*

Intuitively, a transaction with a smaller size should have lower execution time than a larger one. However, it is not always true in the actual execution of transactions. The reason is that the execution time is highly dependent on the initial bank states for the current transaction, i.e., the bank accesses by previous transactions. Figure 4.4 shows a counter example. The 32-Byte write transaction uses bank 2 and bank 3 and the corresponding ACT commands can be scheduled in a pipelined manner with the previous write transaction that uses bank 0. As a result, the scheduling of the *WR* commands is dominated by the *tCCD* constraint. In contrast, the 16-Byte write transaction accesses bank 0. It has to wait longer (*tRWTP+tRP*) to precharge bank 0 and then activate it. This shows that a smaller transaction may have a longer actual execution time.

However, the *WCET* of a smaller transaction cannot be larger than that of a larger transaction. This is guaranteed by Theorem 1 that shows the WCET of an arbitrary transaction T_i monotonically increases with BI_i and BC_i. Moreover, Definition 12 states that

the transaction size is monotone with its *BI* and *BC*. Theorem 2 states that the WCET monotonically increases with transaction size. The proof is included in Appendix A.6.

Theorem 2. *For* $\forall T, T', S(T) \leq S(T') \implies \hat{t}_{ET}(T) \leq \hat{t}_{ET}(T')$.

Theorem 2 allows us to use the WCET of the largest transaction that a requestor can issue as an upper bound for all its transactions. This is especially useful to relax the requirement of fixed transaction size per requestor in the front-end (see Section 3.2.1) by conservatively using the largest transaction size from the requestor.

4.5 WORST-CASE RESPONSE TIME

The worst-case response time (WCRT) of a transaction represents the maximum time consumed to access the shared memory, including time spent in both front-end and back-end. It is based on the WCET computed in Section 4.4. This section introduces the analysis of the WCRT based on the proposed front-end that uses a work-conserving TDM arbiter for requestors with variable transaction sizes, previously presented in Section 3.2.1.

As defined by Definition 9 in Section 2.3.3, the response time of a transaction is the time from it arrives at the front-end of the memory controller until it is finished, i.e., the last data word is returned for a read transaction or the last *WR* command is issued to the SDRAM for a write transaction. The front-end of the memory controller shown in Figure 3.1 uses a TDM arbiter to serve transactions from different requestors. We assume the number of requestors is N. For an arbitrary requestor $r \in [0, N-1]$, the TDM arbiter allocates N_r consecutive TDM slots to it. Moreover, the TDM arbiter is configured to serve requestors in descending order of their transaction sizes to achieve smaller WCET, as discussed in Section 3.2.2. We assume the TDM arbiter serves requestors in the order from Requestor 0 to Requestor $N-1$, where Requestor 0 has the largest and Requestor $N-1$ has the smallest transactions.

The response time of a transaction from a requestor r actually consists of the *interference delay* that is caused by other requestors in the front-end, its own *execution time* in the back-end, and the *time* to return read data. As a result, the transaction experiences the WCRT only if its interference delay is maximum, after which it suffers its WCET in the back-end. With the proposed work-conserving TDM arbitration in Section 3.2.2, the maximum interference delay for a transaction occurs only if it misses any of its slots, causing all its following consecutive slots to be skipped by the arbiter, while the following requestors use all their allocated slots. Moreover, we have to conservatively assume that each transaction from requestor r is executed with the worst-case execution time $\hat{t}^r_{ET}$ in the back-end. Since the size of the previous transaction size is known when using TDM arbitration, we use Theorem 1 to compute $\hat{t}^r_{ET}$. As a result, $\hat{t}^r_{ET}$ is less pessimistic than using Corollary 1, leading to a shorter TDM slot length. The TDM *frame size (FS)*, which is the sum of all slot lengths in the TDM table (given by Definition 13),

is hence smaller. We will later experimentally show the benefits of this approach in Section 4.8.4.4.

Definition 13 (Frame size of the TDM table)**.** *The frame size* $FS = \sum_{r=0}^{N-1} N_r \times \hat{t}_{ET}^{r}$.

The worst-case response time $\hat{t}_{RP}^{r}$ of a transaction from requestor r comprises three parts, as shown in Eq. (4.9). $\hat{t}_{interf}^{r}$ is the maximum interference delay for requestor r, which is given by Eq. (4.10). It is the sum of the WCET of transactions from all other requestors that are executed within their slots. The WCET results of these transactions are given by Theorem 1 with known previous transaction sizes. For the first interfering transaction, its WCET is computed assuming its preceding transaction has the minimum size in the TDM table. This results in conservative WCET of the first interfering transaction, since its previous transaction may be from any requestor and is hence unknown. The second part of Eq. (4.9) is the worst-case execution time of the transaction. Since the execution time of a transaction finishes when the last *RD* or *WR* command is scheduled, the Δt (the third part of Eq. (4.9)) represents the extra time spent on returning the data of the last RD command to the response buffer and is given by Eq. (4.11) that only comprises JEDEC-specified timings.

$$\hat{t}_{RP}^{r} = \hat{t}_{interf}^{r} + \hat{t}_{ET}^{r} + \Delta t \tag{4.9}$$

$$\hat{t}_{interf}^{r} = \sum_{\forall r' \in [0,N-1], r' \neq r} \hat{t}_{ET}^{r'} \times N_{r'} \tag{4.10}$$

$$\Delta t = \begin{cases} tRL + BL/2, & \text{Read transaction} \\ 0, & \text{Write transaction} \end{cases} \tag{4.11}$$

Finally, the transaction may be delayed by a refresh. The maximum refresh delay can be obtained from Eq. (2.2) by assuming the preceeding transaction is write rather than read. It consists of the time between the last *WR* command of the previous transaction and the associated *PRE* and the precharge period as well as the refresh period. However, a refresh is regularly needed every *tREFI* cycles, i.e. a relatively long period of $7.8\mu s$. Therefore, the penalty caused by refreshing depends on the refresh efficiency, as given by Eq. (2.3). For example, the refresh leads to only about 3% increase in the total delay of accessing DDR3-1600G SDRAM for an application. As a result, it is not added to the WCRT of each transaction to avoid pessimism, but added as an overall cost in the system-level analysis of the application.

4.6 WORST-CASE BANDWIDTH

The bandwidth provided by a memory controller represents the long-term average data transfer rate of executing transactions by dynamically scheduling commands to the SDRAM. The worst-case bandwidth (WCBW) is the minimum long-term bandwidth. However, it is difficult to manually analyze the exact sequence of transactions, which leads to the minimum bandwidth. This is because transactions with variable sizes arrive at the front-end of the memory controller randomly, resulting in huge number of transaction sequences. To derive a conservative bound, we can compute the WCBW based on the WCET of an individual transaction rather than a sequence of transactions. Based on Definition 10 that defines the bandwidth, we derive Eq. (4.12) to compute a conservative WCBW, which is the minimum one achieved by different transaction sizes. We assume the total number of different transaction sizes in the system is K and S_k represents one of the transaction sizes, where $\forall k \in [1, K]$. In Eq. (4.12), the WCBW is denoted by $\hat{bw}$. $S(T)$ represents the size of the transaction T while its WCET is $\hat{t}_{ET}(T)$. In addition, f_{mem} is the frequency of the memory and e^{ref} denotes the refresh efficiency as defined by Eq. (2.3). Note that the WCBW is always achieved by the smallest transaction size according to the experimental results that will be later shown in Section 4.8. The reason is that the smallest transaction cannot exploit the bank parallelism as well as a larger transaction that interleaves over more banks. For a system with fixed transaction size, K equals 1, and the WCBW is computed based on this fixed size and its WCET.

$$\hat{bw} = \min_{\forall k \in [1,K],\ S(T)=S_k} \frac{S(T)}{\hat{t}_{ET}(T)} \times f_{mem} \times e^{ref} \tag{4.12}$$

4.7 *RTMEMCONTROLLER* TOOL

The formalization of dynamic command scheduling presented in Section 4.2 can be used to precisely compute the scheduling times of memory transactions. As a result, the formalization is implemented as a C++ tool, named *RTMemController*. It is capable of validating the scheduling times provided by the cycle-accurate SystemC simulator, as described in Section 3.4. Moreover, this tool can also collect the statistic results, such as the average-case execution time (ACET) of transactions and the maximum measured execution time. By integrating Theorem 1, Corollary 1 and Corollary 2, *RTMemController* provides the analytical WCET for fixed and variable transaction sizes, respectively. The scheduled approach given in Section 4.4.4 is also included in this tool and the scheduled WCET is provided. Finally, both the scheduled and analytical WCET bounds are validated by the maximum measured WCET, i.e., the latter cannot be larger than the former. Note that *RTMemController* is an open-source tool [70].

Figure 4.5 shows the design flow of *RTMemController*. Its inputs consist of the memory specifications and the memory transaction traces. The former describe the targeted

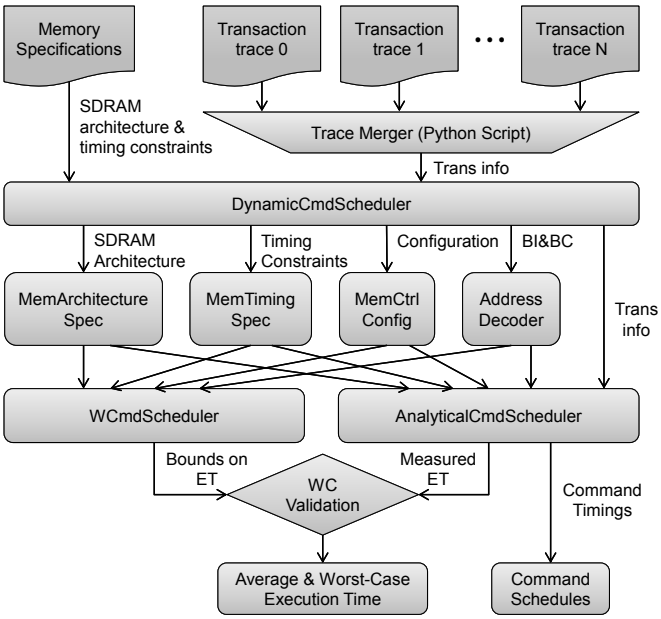


Figure 4.5: The design flow of *RTMemController*, an open-source WCET and ACET analysis tool for real-time memory controllers [70].

SDRAM devices in terms of their architectures (e.g., number of banks, rows, and columns) and the timing constraints corresponding to inter-/intra-bank and refresh. Each memory transaction trace provides the traffic from a requestor. We provide the transaction traces generated by running applications from the MediaBench benchmark suite [65] on the SimpleScalar 3.0 processor simulator [10], which uses separate data and instruction caches, each with a size of 16 KB. The L2 caches are private unified 128 KB caches where the cache-line size varies depending on the experiments. These traces are the same as used by the experiments in Section 3.5. Therefore, the Trace Merger shown in Figure 4.5 is used to combine the traces from different requestors. Note that the Trace Merger is implemented as Python Scripts, which are convenient to read/write data from/into a file. The combined trace contains a sequence of transactions that arrive sequentially and they are later executed in order.

The DynamicCmdScheduler shown in Figure 4.5 is enabled when both the memory specification is chosen and the combined trace is generated. Next, the memory specification is extracted by MemArchitectureSpec and MemTimingSpec for the architecture and the timing constraints of the selected SDRAM, respectively. Moreover, the MemCtrlConfig provides the configuration (i.e., BI and BC per transaction size) of the memory controller when it executes fixed or variable transaction sizes. The AddressDecoder takes the memory map configuration in terms of *BI* and *BC* per transaction size as the

input and translates the logical address of each transaction into the physical address in terms of bank, row, and column. The next step of this design flow is the command scheduling. The WCmdScheduler is only triggered once and it computes the analytical based on Theorem 1, Corollary 1 and Corollary 2. Moreover, the scheduled WCET is also calculated by WCmdScheduler for all possible transaction sizes when the memory controller receives transactions with fixed size or variable sizes, respectively. These results are stored in a table and are later validated by the AnalyticalCmdScheduler. For each transaction, the AnalyticalCmdScheduler computes the scheduling times of all its commands based on the proposed formalization in Section 4.2. These scheduling times constitute the Command Schedules (see Figure 4.5), which are used to validate the cycle-accurate SystemC simulator when it uses the same traces. Moreover, the execution time of each transaction is collected, where the maximum ET is recorded as the measured WCET while the ACET is obtained by averaging the ET of all transactions. The measured WCET validates our analytical and scheduled WCET and guarantees that our WCET bound is conservative. Finally, the WCET, ACET and the scheduling times of commands constitute the output of this tool.

4.8 EXPERIMENTAL RESULTS

This section experimentally evaluates the formal analysis and presents the analytical and scheduled worst-case execution/response time and worst-case bandwidth. These worst-case results are validated by comparing to the measured ones given in Section 3.5, where our memory controller Run-DMC has been evaluated. Therefore, we carry out experiments with the same setup as given in Section 3.5. Three experiments are presented in this section. The first experiment shows that the formalization accurately describes the timing behavior of the memory controller back-end. The last two experiments evaluate our analysis for fixed transaction size and variable transaction sizes, respectively. The results in terms of WCET, WCRT, and WCBW are analyzed and also compared to a state-of-the-art semi-static approach [3].

4.8.1 *Experimental Setup*

Our open-source tool *RTMemController* is implemented with C++ and it has been tested on a 64-bit Ubuntu 14.04.1 LTS system with 2 Intel(R) Core(TM) i7 CPU running at 2.8GHz and with 2 GB RAM. Please refer to [70] for the detailed usage of *RTMemController*. We reuse the experimental setup given in Section 3.5.1, where four requestors are served by the front-end of Run-DMC using a novel TDM arbiter. Each requestor is allocated a TDM slot and produces a memory trace. The characterizations of these MediaBench or synthesis traces used in the experiments are given in Table 3.1 and Table 3.2. The transaction sizes include 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes, and their memory configurations in terms of (*BI*, *BC*) are (1, 1,), (2, 1), (4, 1), (4, 2), and (4,

4), respectively. These configurations ensure the lowest execution time of transactions with each size. The order of serving requestors with variable sizes will also be investigated to show that the descending service order of transaction sizes can achieve the lowest TDM frame size, resulting in the lowest WCRT. Toward this goal, we investigate the cases of 4 and 8 requestors, respectively. Note that experiments have been done with three JEDEC-compliant DDR3 SDRAMs, DDR3-800D, DDR3-1600G, DDR3-2133K, all with interface widths of 16 bits and a capacity of 2 Gb [53].

4.8.2 *Experimental Validation of the Formalization*

The purpose of our first experiment is to validate the formalization of the timing behavior of the dynamic command scheduling in Algorithm 2 by verifying that the scheduling time of each command is the same as given by the cycle-accurate SystemC simulator. To this end, the open-source off-line scheduling tool *RTMemController* [70] that implements the formalism has been provided with the same inputs as the SystemC implementation for all experiments in this chapter, covering a wide range of read and write transactions with different sizes and inter-arrival time under different memory map configurations. The results of this experiment are that all commands of all transactions are scheduled *identically*, indicating that the formalization accurately captures the implementation. This is important since the formalization forms the base for both the analytical and the scheduled WCET bounds. Moreover, it suggests the SystemC implementation is correct. *This proven relation between the formal model and the implementation is an important result of our work and a distinguishing feature compared to the related work.*

4.8.3 *Fixed Transaction Sizes*

This experiment evaluates our formal analysis approach for the dynamically-scheduled memory controller Run-DMC with fixed transaction sizes. The worst-case bounds on execution time, response time and bandwidth are obtained using Corollary 2, Eq. (4.9) and Eq. (4.12), respectively. With the same experiment setup used in Section 3.5.2, those worst-case bounds are validated by comparing to the measured worst-case results. Recall that this experiment tests four memory requestors corresponding to four processors, which execute different Mediabench applications (*gsmdecode*, *epic*, *unepic* and *jpegencode*). The TDM arbiter in the front-end of Run-DMC allocates one slot per requestor. Moreover, our formal analysis of Run-DMC is also compared to the semi-static approach [3], the only other approach that supports different memory map configurations.

4.8.3.1 *Worst-Case Execution Time*

The execution time of a transaction is spent on scheduling commands to the SDRAM in the back-end of our memory controller. This section evaluates the back-end in terms

of the worst-case execution time (WCET). Figure 4.6 shows the WCET results of our dynamically-scheduled memory controller Run-DMC given by the formal analysis and the measured maximum execution time by running the MediaBench application traces. These results are compared to the semi-static approach [3]. The results in Figure 4.6 demonstrate that

1. The maximum measured WCET from the experiments is equal to or slightly smaller than the scheduled WCET. This indicates that the proposed analysis provides a tight WCET bound. The scheduled WCET is a little too conservative for some transaction sizes, e.g., 32 bytes and 64 bytes for DDR3-1600G that use $BC = 1$. This is caused by the worst-case initial states determined by the *ALAP* scheduling in Section 4.3.2, which is pessimistic since *tCCD* is always used as the time interval between two *RD* or *WR* commands. However, for $BC = 1$, the actual interval is larger than *tCCD* because *ACT* command dominates in the scheduling of a *RD* or *WR* command. This pessimism is eliminated for 128-byte and 256-byte transactions that use ($BI = 4$, $BC = 2$) and ($BI = 4$, $BC = 4$), where the *ALAP* scheduling accurately determines the worst-case initial states.

2. The analytical WCET derived from Corollary 2 is equal to or slightly larger than the scheduled WCET. The difference is because Theorem 1 conservatively assumes a collision per *ACT* command, which may not actually be the case and all collisions do not necessarily lead to an increased execution time, since the *ACT* command does not always dominate in the computation of the finishing time (see Lemma 1). The maximum difference is *BI* cycles. However, the analytical WCET is much easier to obtain, since it can be computed based on an equation. In contrast, a tool (i.e., *RTMemController*) is needed to derive the scheduled WCET.

3. The WCET given by the semi-static approach is identical to the measured WCET of our approach. There is a single exception for 32 byte transactions, where the measured WCET given by Run-DMC is 41 cycles, while it is 40 cycles for the semi-static approach. Since this exception is highly dependent on the values of timing constraints, it does not occur for most DDR3 SDRAMs. For example, there is no such exception for DDR3-800D whose results are not presented here for brevity. The interested reader can refer to [69]. It is worth noting that Run-DMC achieves significantly better average ET than the semi-static approach, as previously discussed in Section 3.5.2.1. The improvement on the average ET benefits the non-real-time applications, which are simultaneously supported with real-time applications [72].

4.8.3.2 *Worst-Case Bandwidth*

The worst-case bandwidth (WCBW) represents the minimum long-term data transferring rate, and it is the lower bound on bandwidth. In Section 4.6, Eq. (4.12) is given to

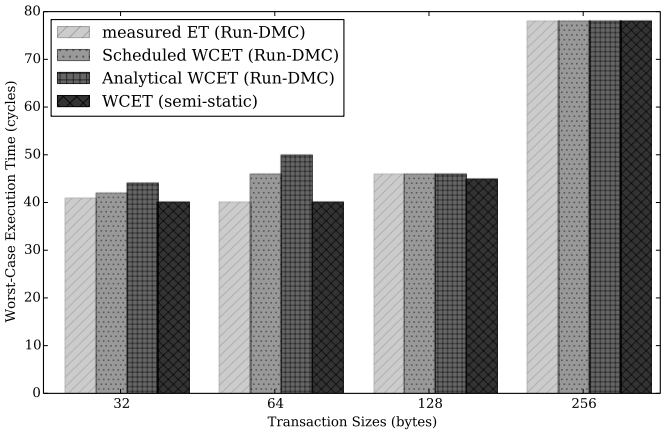


Figure 4.6: The WCET of fixed transaction sizes with DDR3-1600G SDRAM. Results are compared to a semi-static approach [3].

compute the WCBW based on the transaction size and the corresponding WCET. Therefore, we can derive scheduled and analytical WCBW based on the scheduled and analytical WCET. First, these WCBW bounds are conservative for bandwidth provided by Run-DMC, since they are not larger than the measured minimum bandwidth, as shown in Figure 4.7, where DDR3-1600G SDRAM is taken as an example. When comparing to the semi-static approach (see Figure 4.7), we can draw similar conclusions as the WCET in the previous Section 4.8.3.1 that 1) the measured WCBW of our Run-DMC is identical to the WCBW of the semi-static approach for all these transaction sizes except 32 bytes, where Run-DMC achieves slightly smaller measured WCBW. The reason is that the WCET of 32-byte transactions is slightly larger than that of the semi-static approach because of the particular timing constraints. 2) The scheduled and analytical WCBW are the same as the semi-static approach for large transaction sizes (e.g., 256 bytes) but not the smaller ones, such as 32 bytes, 64 bytes, and 128 bytes. It is because these transactions use smaller BC ($BC = 1$ or $BC = 2$), i.e., a fewer RD or WR data bursts per bank, and the formal analysis has to assume conservative time interval between two successive RD or WR commands, resulting in pessimistic initial bank states and hence larger WCET and smaller WCBW. However, our formal analysis only provides slightly less WCBW for these small sizes than the semi-static approach. As shown in Figure 4.7, it has maximally 20% less WCBW than the semi-static approach for 64-byte transactions, while it is only 2.2% for 128-byte transactions. Moreover, Figure 4.7 shows that higher WCBW is achieved with larger transaction sizes. The reason is that larger transactions transfer more data bursts from an individual bank, while it pays the activation penalty only once, i.e., the required bank is opened by issuing an ACT command. As a result, it is more efficient.

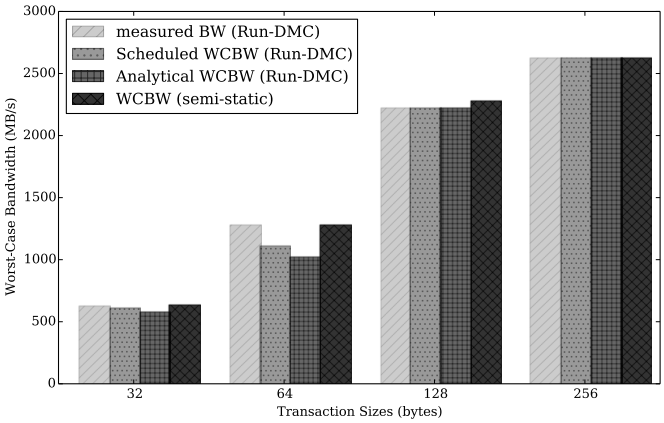


Figure 4.7: The worst-case bandwidth (WCBW) for a DDR3-1600G SDRAM using our dynamically-scheduled Run-DMC and the semi-static approach [3] with fixed transaction sizes.

4.8.3.3 *Worst-Case Response Time*

The WCRT of a transaction is given by Eq. (4.9). It is essentially determined by accumulating the WCET of transactions from each requestor. This experiment shows the worst-case response time of transactions with fixed sizes. Figure 4.8 presents the WCRT for DDR3-1600G with fixed transaction sizes. The results are derived on the basis of the WCET shown in Figure 4.6, and new observations from Figure 4.8 include: *1)* the response times of transactions are bounded. The measured WCRT is smaller than the bound in terms of scheduled and analytical WCRT. The difference between them is because the worst-case situation is unlikely to occur in both the front-end and back-end simultaneously, which requires transactions from all requestors competing in the front-end, while each transaction in the back-end experiences worst-case initial bank state. *2)* The analytical WCRT is more pessimistic than the scheduled WCRT, because the analytical WCRT is derived by accumulating the analytical WCET of transactions from each requestor. This exaggerates the conservative assumption of a collision per ACT command for computing the analytical WCET. These observations also hold for other DDR3 SDRAMs, although their WCRT results are not presented for brevity.

4.8.4 *Variable Transaction Sizes*

The last experiment evaluates our approach with variable transaction size. First, the WCET of a transaction is evaluated without any a priori information, e.g., the size of the previous transaction, where the worst-case is assumed. Second, we experiment with the case when the size of the previous transaction is statically known, which is guaran-

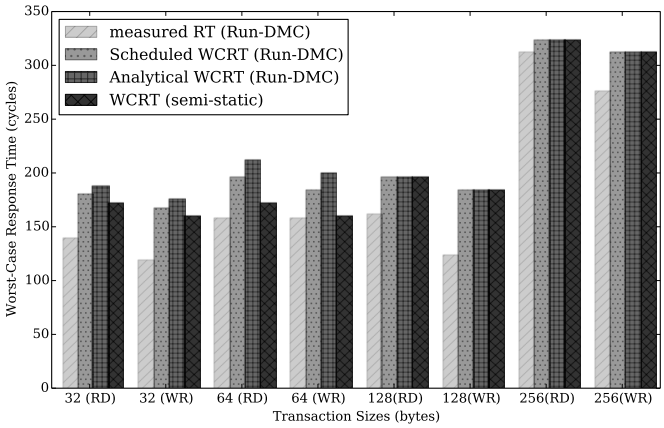


Figure 4.8: The worst-case response time for DDR3-1600G SDRAM with fixed transaction sizes

teed by the TDM arbiter. Then, the impact of the service order of requestors with their transaction sizes on the WCET is evaluated, based on which the WCRT are evaluated. The setup is loosely inspired by a High-Definition video and graphics processing system featuring a number of CPU, GPU, hardware accelerators and peripherals with variable transaction sizes. This setup has been used in Section 3.5.3. So, recall that the system has 4 requestors with transaction sizes of 16 bytes, 32 bytes, 64 bytes and 128 bytes, respectively. The first requestor Req_1 represents a GPU with 128 byte cache line size, executing a Mediabench application *jpegdecode*. A video engine corresponding to requestor, Req_2, is used for *mpeg2decode* and it generates memory transactions of 64 bytes. The Mediabench application *epic* is executed by a processor with a cache-line size of 32 bytes, which is denoted Req_3. A synthetic memory trace is used by a CPU which has a 16 byte cache-line size, resulting in read and write transactions with 16 bytes. This is requestor Req_4. The characterizations of these memory traces are given in Table 3.2. The TDM arbiter in the front-end allocates one slot per requestor and it serves these requestors from Req_1 to Req_4 in descending order of their transaction sizes.

4.8.4.1 *Worst-Case Execution Time*

Corollary 1 is used to compute the WCET of transactions with variable size, and the results for DDR3-1600G are shown in Figure 4.9. It also shows the WCET results given by the semi-static approach for particular sizes, including 16 bytes, 32 bytes, 64 bytes and 128 bytes, respectively. Note that the static command schedules (also named patterns) used by the semi-static approach are computed at design time for a particular transaction size, and are configured before the system is running. We get similar conclusions as previously presented in Section 4.8.3.1. New interesting observations are:

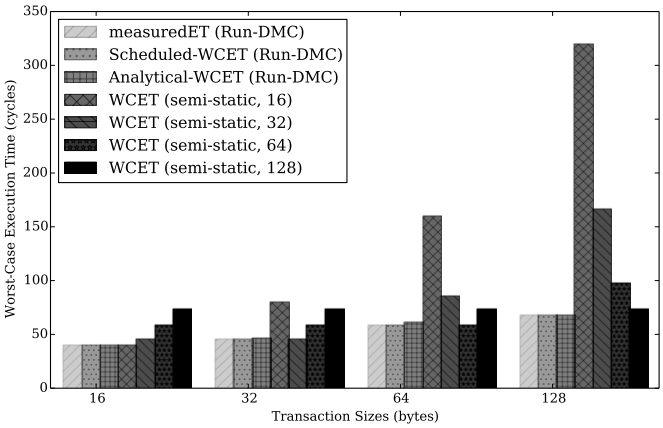


Figure 4.9: WCET for DDR3-1600G with variable transaction sizes.

1. the scheduled WCET bound is perfectly tight, since the worst-case situation for a transaction is accurately captured by Corollary 1 for variable sizes, and actually occurs during simulation. The situation is that the previous transaction is a write and its finishing bank is the starting bank of the new transaction.

2. when the semi-static approach is used for variable transaction sizes, it has to choose a particular pattern size such that the total WCET of all requestors is minimum, leading to smaller WCRT. For a particular pattern size, transactions with larger sizes have to be split into several pieces that are served in consecutive TDM slots. If the transaction size is smaller than the pattern size, it will fetch the data and throw the unnecessary part away. This has two consequences. First, the WCET of transactions with variable sizes highly depend on the chosen pattern size. For example, the 16-byte pattern provides very high WCET for larger transaction sizes, as shown in Figure 4.9. Second, since data is discarded, it wastes power and reduces the bandwidth provided by the SDRAM, which is a scarce resource. The best pattern size depends on the mix of the transaction sizes and the timing constraints of the memory. For example, the best pattern size used in our experiments for DDR3-1600G is 128 bytes, while it is 64 bytes and 128 bytes for DDR3-800D and DDR3-2133K, respectively.

3. the WCET for each transaction obtained from our approach is less than or equal to that of the semi-static approach. This demonstrates that our dynamically scheduled memory controller outperforms the semi-static approach in the worst case with variable sizes.

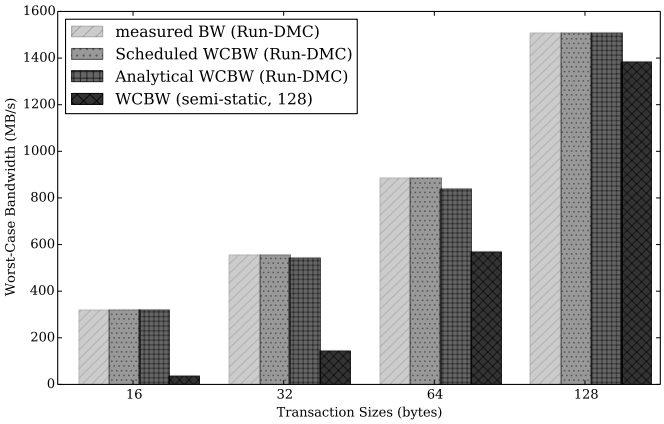


Figure 4.10: Worst-Case Bandwidth for DDR3-1600G with variable transaction sizes.

4.8.4.2 *Worst-Case Bandwidth*

The worst-case bandwidth (WCBW) is computed based on the transaction size and the corresponding worst-case execution time (WCET), as given by Eq. (4.12). For variable transaction sizes, the WCET per size was previously presented in Section 4.8.4.1. In particular, to achieve the lowest WCRT with variable sizes, the semi-static approach can use the best patterns designed for 128-byte transactions for DDR3-1600G SDRAM. Therefore, we can derive the WCBW, as shown in Figure 4.10. We can draw similar conclusions as in Section 4.8.4.1 for the WCET of variable transaction sizes. For example, both the scheduled and analytical WCBW bounds are valid, since they are not larger than the measured minimum bandwidth. Moreover, the scheduled WCBW are tight bounds because they are identical to the measured bandwidth, while the analytical WCBW are conservative. In addition to these conclusions, Figure 4.10 shows that our dynamically-scheduled memory controller Run-DMC always provides more WCBW with variable transaction sizes than the semi-static approach. The reason is that the semi-static approach has poor data efficiency for small transaction sizes (e.g., 16 bytes, 32 bytes, and 64 bytes) when it uses the best pattern (i.e., for 128-byte transactions) to achieve the lowest WCRT. Though the data efficiency is 100% for 128-byte using the semi-static approach, its WCET is larger than that given by Run-DMC, as previously discussed in Section 4.8.4.1.

4.8.4.3 *WCET with Known/Unknown Previous Transaction Size*

The WCET of a transaction is given by Corollary 1 for unknown previous transaction size, denoted as pre-size, while it is provided by Theorem 1 for known pre-size. As discussed in Section 4.4, if there is no static information about the size of the previous transaction, we have to assume the worst case, i.e., that its starting bank was the finishing

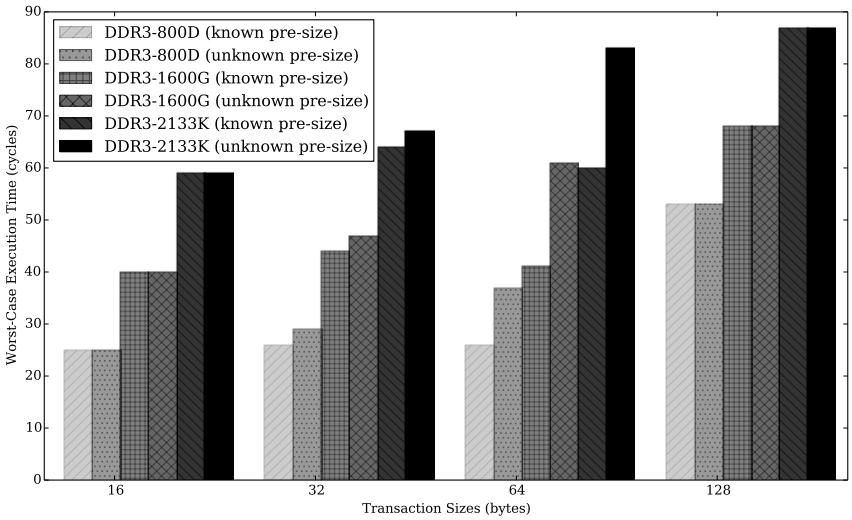


Figure 4.11: WCET with known/unknown previous transaction size. Requestors are allocated to TDM slots in descending order of their transaction sizes.

bank of the previous write transaction. This results in pessimism for the WCET given by Corollary 1. The TDM arbiter in the front-end provides static information about the slot allocation per requestor. Therefore, the size of the previous transaction is statically known in the worst case. In this experiment, four requestors have transaction sizes of 128 bytes, 64 bytes, 32 bytes and 16 bytes, respectively. The TDM arbiter allocates one slot per requestor and serves them in descending order of sizes, e.g., from 128 bytes to 16 bytes. Figure 4.11 shows the WCET of a transaction with known and unknown size of the previous transaction for DDR3 SDRAMs, respectively. We can see that the WCET with unknown previous transaction size is greater than or equal to the case with known size. For example, a 128 byte transaction is preceded by a 16 byte transaction consisting of one burst, leading to no difference for its WCET if the previous size is known or unknown. In contrast, a 64 byte transaction is preceded by a 128 byte transaction. Its starting bank cannot be the finishing bank of the 128 byte transaction for aligned transactions, resulting in much better WCET with known previous transaction size (see Figure 4.11). Therefore, shorter worst-case frame size is obtained if the size of previous transaction is known. This leads to smaller WCRT, as presented in the following section.

4.8.4.4 *TDM Service Order of Requestors*

Besides known size of the previous transaction, lower WCET is obtained if transactions are executed in descending order of their sizes because of improved pipelining between successive transactions, as previously discussed in Section 3.2.2. This results in a shorter

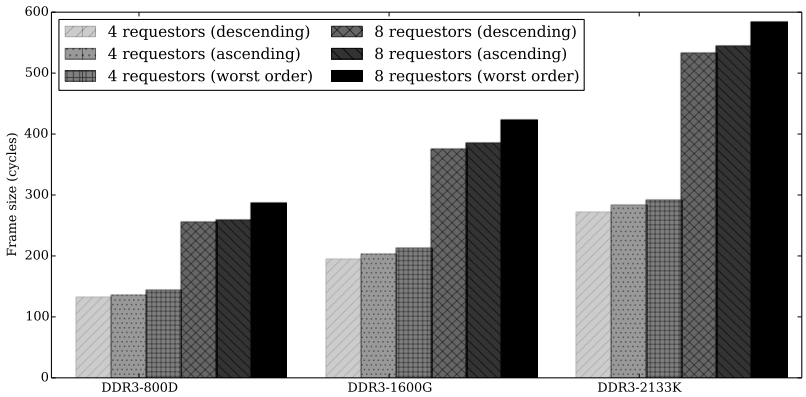
frame size. An experiment is carried out to explore all the possible orders of serving 4 and 8 requestors with transaction sizes of 16 byte, 32 byte, 64 byte and 128 byte, respectively. For the case of 8 requestors, there are two requestors with each transaction size. Each requestor has one slot in the TDM table. All the possible orders of serving these requestors have been evaluated, although only frame sizes for descending, ascending and the worst possible order are shown in Figure 4.12 (a). The best way to serve requestors is descending order of their transaction sizes. The worst order is the one that results in the maximum frame size. The experiment shows that the minimum frame size is always obtained using the descending order. Compared to the worst order, the improved percentage of frame size by using descending order is given by Figure 4.12 (b). It indicates that a system with a larger number of requrestors benefits more from the descending order, e.g., 13.4% is gained for 8 requestors with DDR3-800D. Note that this is a free improvement by using our analysis in Section 4.4.3, which provides the WCET by exploiting more detailed information about the bank state when the size of the previous transaction is known. This has not been considered by existing work.

4.8.4.5 *Worst-Case Response Time*

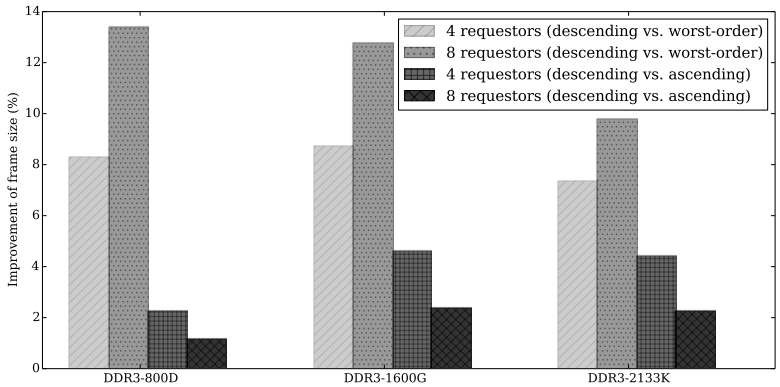
The WCRT for the four requestors is derived from Eq. (4.9) and the results for DDR3-1600G are shown in Figure 4.13. They are obtained on the basis of the WCET by using Theorem 1. In addition, to fairly compare with the semi-static approach, we choose the best pattern size, e.g., 128 byte for DDR3-1600G. As can be seen from Figure 4.13, it also supports the conclusion given by Figure 4.9 that our dynamically scheduled memory controller outperforms the semi-static approach in the worst case, where our scheduled approach is always better or equal and the analytical approach is worse than the semi-static approach only for 128-byte transactions. As the observation also holds for the other DDR3 SDRAMs, their results are not shown. It is worth to recall that our approach significantly reduces the total time for each application to access the memory, which has been presented in Section 3.5.3.3. For example, compared to the semi-static approach, 53.8% reduction of the average response time for accessing DDR3-1600G is achieved by the Mediabench application *epic* that has 32 byte memory transactions. In addition, the average improvement is 47.6% for all the Mediabench application traces with DDR3-1600G.

4.8.5 *Monotonicity of Worst-Case Execution Time*

Theorem 2 states that the analytical WCET monotonically increases with the transaction size, and it is based on the WCET given by Theorem 1. However, we cannot prove this for the scheduled WCET, as mentioned in Section 4.4.4. We proceed by providing experimental evidence to show that the monotonicity property also holds for the scheduled approach.



(a) The worst-case frame size



(b) The reduction of frame size

Figure 4.12: The worst-case frame size of a TDM table for different number of requestors, and the improvement by serving requestors in descending order of their transaction sizes.

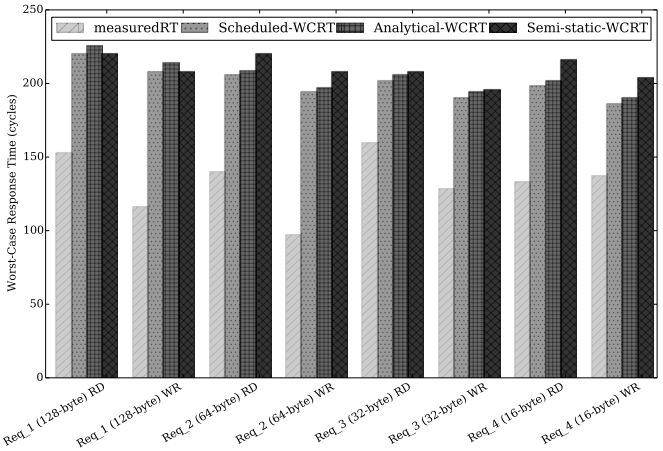


Figure 4.13: WCRT for DDR3-1600G with variable transaction sizes.

Experiments have been done with DDR3-800D, DDR3-1600G and DDR3-2133K to collect the scheduled WCET of transactions. All pair-wise combinations of 16, 32, 64, 128, and 256 bytes transactions have been tested. Figure 4.14 shows the scheduled WCET results of transactions with different sizes under different preceding transaction sizes for DDR3-1600G. The results show that the scheduled WCET appear to be monotonic with the transaction size. This experimental observation also holds for the other memories, and the results are not presented for brevity. We conclude that *the scheduled WCET monotonically increases with the transaction size for DDR3-800D/1600G/2133K memories.*

4.9 SUMMARY

The chapter proposes a formal analysis approach to analyze the worst-case execution/response time and the worst-case bandwidth of our dynamically-scheduled memory controller, previously presented in Chapter 3. This formal analysis approach is based on the formalization of the dynamic command scheduling. The scheduling times of commands can be precisely calculated based on the formalization, which is implemented to be an open-source C++ tool *RTMemController*. On one hand, this tool is validated by the cycle-accurate SystemC simulator (see Section 3.4), where identical scheduling times of commands for the same transaction traces are obtained. On the other hand, *RTMemController* is used to debug the SystemC simulator. With this formalization, the worst-case execution time of a transaction is analyzed based on the worst-case initial bank states, which are derived by scheduling the commands of the previous transaction as-late-as-possible (ALAP). The ALAP scheduling maximizes the scheduling times of these previous commands. Due to the constant JEDEC timing constraints between commands, the commands of the current transaction are scheduled at their maximum

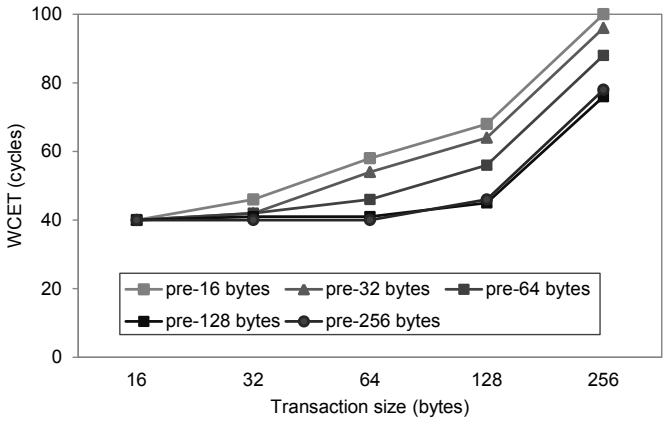


Figure 4.14: The monotonicity of scheduled WCET with transaction size for a requestor. DDR3-1600G is taken as an example.

times. As a result, the maximum execution time (i.e., WCET) is guaranteed. Based on the analysis, two techniques are presented to bound the WCET. The first technique is an equation that computes the WCET for a given transaction size and memory map configuration, while the second technique provides a tighter bound by using an off-line implementation of the dynamic command scheduling to compute actual command collisions. Both of these techniques are included in *RTMemController*. We formally prove that the analytical WCET monotonically increases with the transaction size, and we provide experimental evidence for DDR3-800D/1600G/2133K SDRAMs that this also holds for the scheduled approach. With the WCET of transactions, the lowest WCRT is derived based on the new work-conserving TDM arbiter that schedules transactions from different requestors in the descending order of transaction sizes. Comparison with a state-of-the-art semi-static scheduling approach shows that our approach performs equally well or better in the worst-case with only a few exceptions. Note that our approach significantly reduces the average response times by 79% at most while 44.9% on average, implying shorter time for each application to access the memory. This was previously concluded in Chapter 3.

5

MODE-CONTROLLED DATAFLOW (MCDF) MODELING OF RUN-DMC

The analysis of real-time memory controllers is difficult, and the reasons have been previously discussed in Chapter 4, including: i) the interferences between memory requestors, ii) the complex dependencies between SDRAM commands due to the inter- and intra-bank timing constraints, and iii) the diverse memory traffic with variable transaction sizes. These difficulties have been solved by the formal analysis approach in Chapter 4 by assuming that 1) the worst-case bank states for a transaction are given by as-late-as-possible (*ALAP*) scheduling, and 2) each *ACT* command is always collided with a *RD* or *RD* command. These assumptions make the worst-case bounds pessimistic. Moreover, the formal analysis approach provides the bounds based on analyzing each individual transaction rather than a sequence of transactions, where the pipelining cannot be exploited. In particular, the formal analysis approach cannot provide tight bounds on worst-case bandwidth. The reason is that the bandwidth evaluates the long-term average data transferring rate of executing an infinite number of transactions according to Definition 10. On the other hand, the formal analysis approach is time-consuming and huge effort is needed when analyzing a memory controller with different mechanisms, implying a portability issue.

This chapter introduces a novel approach to derive the lower bound on the worst-case bandwidth (WCBW) by exploiting the pipelining between successive transactions, which are executed by our dynamically-scheduled memory controller (i.e., Run-DMC). As previously presented in Chapter 3, Run-DMC is capable of efficiently dealing with the diverse memory traffic with variable transaction sizes using dynamic command scheduling. The proposed approach captures the complex command scheduling dependencies of transactions with a dataflow model, where SDRAM commands and inter-/intra-bank timing constraints are represented by nodes with specified execution times in a dataflow graph, while the dependencies between commands are represented by the edges of the graph. By using existing analysis tools of dataflow models, such as SDF3 [98] and Heracles [79], the WCBW bounds can be automatically obtained based on iteratively executing the dataflow graph. This corresponds to the execution of a sequence of transactions. Therefore, the WCBW bounds are derived based on exploiting the pipelining between transactions. Comparing to the formal analysis approach in Chapter 4, the analysis of

the dataflow model does not need to assume the worst-case initial bank states given by the *ALAP* scheduling, ensuring tighter bounds.

A MCDF model [68, 79, 90] is used to capture the command scheduling dependencies of Run-DMC. It supports dynamism by selecting different sub-graphs, which correspond to different modes. As a result, the dynamism caused by executing the transactions is captured by creating modes and specifying the mode transitions. Finally, the Heracles [79] tool is used to analyze the WCBW. The advantages of the MCDF model include: 1) it leverages standard dataflow analysis techniques and tools to analyze the bound on the worst-case bandwidth of memory command scheduling without the need to manually develop new complex static analyses. 2) It can easily exploit static information, such as the transaction sequence given by the application or static arbitration of memory requestors (e.g., time-division multiplexing), through generating proper mode sequences. In contrast, the formal analysis approach in Chapter 4 can only exploit the static order of transaction sizes rather than the types and physical addresses contained in the static transaction sequence. 3) The analysis of the MCDF model returns the sequence of commands (corresponding to transactions) that limit the worst-case bandwidth, which is beyond the capability of existing analyses. This information is useful when designing scheduling algorithms, such that the critical sequence of transactions is avoided and hence a better worst-case bandwidth is obtained. 4) The validation of the MCDF model is easier than existing analyses because the formal model is also executable. 5) The MCDF model can be easily adapted to cover other memory controllers with different scheduling policies, which can be captured by mode sequences. 6) Finally, the worst-case bandwidth bounds are better than both the scheduled and analytical approaches introduced in Section 4.6 for Run-DMC and the Predator controller using a semi-static approach [3]. The maximum improvement is 22% while the average improvement is 6.3%. We also experimentally show that exploiting static sequences of transactions achieves up to 77% higher worst-case bandwidth bound, while the average improvement is around 63.2%.

In the remainder of this chapter, Section 5.1 summarizes the related work of analyzing the worst-case bandwidth. The background of dataflow model and mode-controlled dataflow (MCDF) model is given in Section 5.2. The proposed MCDF model of Run-DMC is presented in Section 5.3, followed by introducing the method of deriving the WCBW based on analyzing the MCDF model in Section 5.4. Finally, experimental results are shown in Section 5.5, before summarizing this chapter in Section 5.6.

5.1 RELATED WORK

The worst-case memory bandwidth is challenging to analyze because of the command scheduling dependencies based on the complex internal states of SDRAM [53] and the diverse memory traffic. Most existing approaches for computing memory bandwidth abstract away the complexity of SDRAM internal states. A memory access control ap-

proach has been proposed in [109] to allocate enough bandwidth to a critical core that runs a real-time application. However, it uses constant memory access time to compute the bandwidth, which is pessimistic for variable transaction sizes with different execution time. This drawback also applies to the bandwidth sharing scheme in [87] that treats every memory access as a constant delay. The worst-case bandwidth can be analyzed when a memory controller has statically periodic behavior of command scheduling. For example, the mixed-criticality memory controller in [25] repeats a fixed TDM schedule of command scheduling. This is similar to the memory controllers in [3, 46] that use static command schedules. The PRET DRAM controller [88] does not directly use static command schedules, while it provides conservative periodic cycles of issuing commands. A similar memory controller presented in [26] uses virtual devices, each of which is composed of several private banks and uses a fixed sequence of commands to serve a read/write transaction with fixed size. The virtual devices are periodically accessed in a TDM manner. However, these memory controllers only directly support transactions with fixed sizes to ease their worst-case analysis. They cannot efficiently deal with the variable sizes in diverse memory traffic. In contrast, our memory controller (i.e., Run-DMC [69, 72]) introduced in Chapter 3 dynamically schedules commands for transactions with variable sizes. It is capable of exploiting the run-time SDRAM states. However, the formal analysis approach of Run-DMC given Chapter 4 is complicated and its WCBW bounds are pessimistic due to the assumptions. In this chapter, we tackle this complexity by modeling the command scheduling of Run-DMC with a dataflow model [79], where existing analysis techniques and tools can be used to analyze a tighter WCBW bounds.

Dataflow models have been widely used to model shared resources in modern multi-core systems and provide guaranteed performance. For example, the behavior of an on-chip network is captured by a dataflow model [45] that is used to compute the required buffer size of a network interface, such that the performance of an application is guaranteed. Another example is the dataflow modeling of TDM arbitration [67], that enables an optimized TDM slot allocation to meet the requirements of concurrent applications. The dataflow models of these two examples actually describe the dependencies of resource sharing, and existing dataflow analysis techniques are employed to provide the worst-case results. Nelson et al. shows in [80] how a streaming application mapped on a multi-core platform with several shared resources can be modeled using dataflow. However, the modeling of the resource sharing is quite abstract and does not capture the internals of the resources in detail. *This chapter proposes to capture the complex dependencies of dynamic command scheduling by a dataflow model, where we extend an analysis tool to address these complexities and provide the WCBW. To the best of our knowledge, this work provides the first dataflow model of a memory controller in detail.*

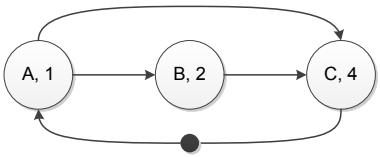


Figure 5.1: A single-rate dataflow graph.

5.2 BACKGROUND OF DATAFLOW MODELS

This section introduces background information about dataflow models, such as the single-rate dataflow (SRDF), followed by presenting the basic concepts and structures of mode-controlled dataflow model in details.

5.2.1 *Single-Rate Dataflow Model*

Dataflow models are popular to describe concurrent processes with unidirectional graphs, where a process is represented by a node (i.e., *actor*), while an edge between two nodes is a FIFO communication *channel* between the corresponding processes [66]. An actor *fires* or executes immediately when all its input *tokens* (i.e., data) are available. The firing of an actor consumes tokens from all inputs and produces a number of output tokens that are transfered to the next actors. The number of these consumed/produced tokens is the *consumption/production rate*. For *single rate dataflow (SRDF)*, each actor has a fixed execution time and communicates with other actors using a single token through each channel. Initial tokens are specified on some edges of the SRDF graph, such that the graph starts firing with particular actor(s). Since the firing of an actor is triggered once all its input tokens arrive, the dependencies between concurrent processes are captured by transmitting tokens between actors. Moreover, the execution of the processes is captured by the iterative firings of the SRDF graph. An iteration of an SRDF graph is defined as a set of actor firings, such that all the initial tokens return to their initial edges, i.e., the SRDF graph returns to the initial state. SRDF model expresses the dependencies between concurrent processes while provides good analytical properties that guarantee the performance in terms of latency and throughput [27].

Figure 5.1 shows an SRDF that has three actors A, B and C with the execution time of 1, 2, and 4, respectively. Their consumption and production rates are 1, which is not shown in Figure 5.1 for brevity. The edges between the actors denote the dependencies. There is one initial token on the edge from actor C to actor A, such that A starts at the beginning, followed by actor B and C once their input tokens become available. For example, actor C starts immediately when both the input tokens from actor A and B are produced. When the dataflow graph depicted by Figure 5.1 executes iteratively with each actor running in a pipelining manner, it is obvious that the critical cycle that

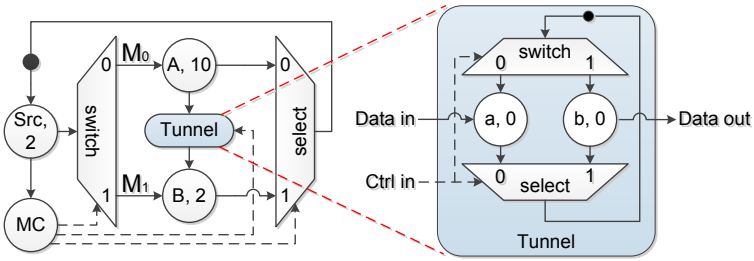


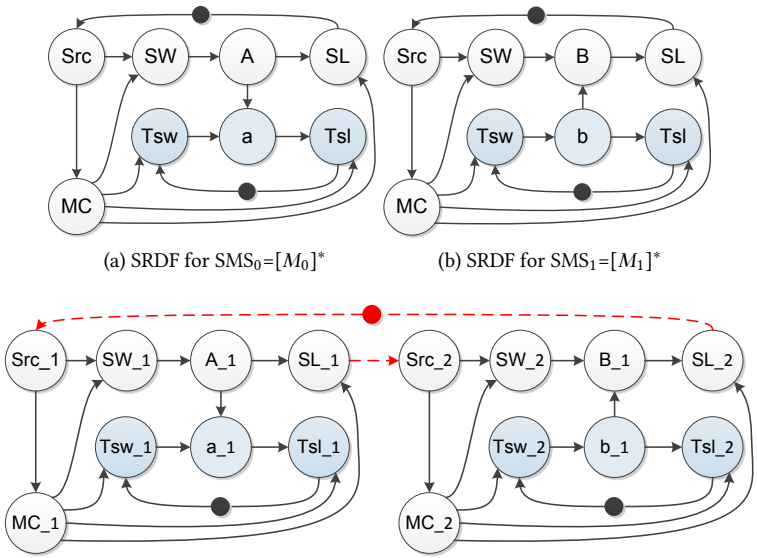
Figure 5.2: An MCDF graph and a basic tunnel.

determines the execution speed of the graph is from A to B to C. This cycle repeats with the time interval of 7 that is the total execution times of the actors on the critical cycle divided by the number of initial tokens on its edges, which is defined as the *maximum cycle mean (MCM)* in tokens/second.

5.2.2 *Mode-controlled Dataflow Model*

Mode-controlled dataflow (MCDF) [79] is a restricted variant of Boolean dataflow [18] that supports dynamism by selecting different sub-graphs of the MCDF graph to fire for each graph iteration, where Figure 5.2 shows a simple example of an MCDF model. These sub-graphs, called modes, are actually smaller dataflow graphs. MCDF features single rate dataflow (SRDF) actors and two types of special actors, named select and switch, which conditionally produce/consume tokens on/from specific edges depending on the mode selected for that firing, which is defined by the value of the token consumed from its *mode control input*. In addition, a special single-rate actor is marked as *model controller (MC)* and produces all tokens consumed through the control ports of all switches and selects in the MCDF graph. For each firing of MC, one token with the same mode value is produced by MC on all control inputs of all switches and selects, which are enabled to fire exactly once in an iteration. The actors of switches, selects, and mode controller fire once per iteration, while the rest actors only fire if their mode is chosen. This means that the actors corresponding to the non-selected modes do not fire. The control tokens drive all the switch and select actors via their control input ports to select different modes. A tunnel constructed with switch and select provides a convenient way to communicate tokens between two modes. It behaves as a *register* and is also driven by control tokens.

The *construction rules* of an MCDF model are that 1) it uses a single MC actor and an arbitrary number of switch, select and tunnel actors. These actors always fire for any chosen mode. 2) MC selects a mode by sending a control token. The switch and select activate the actors of the selected mode to fire. The actors of unselected modes cannot fire. 3) An actor is not allowed to belong to more than one mode. With these



(a) SRDF for $SMS_0=[M_0]^*$ (b) SRDF for $SMS_1=[M_1]^*$

(c) SRDF for $SMS_2=[M_0, M_1]^*$ by unrolling the MCDF graph

Figure 5.3: The equivalent SRDF of recurring SMS for the MCDF in Figure 5.2.

rules, the MCDF model has strong expressiveness to capture the dynamism of a system by dynamically choosing modes.

Figure 5.2 shows a simple MCDF graph that consists of two modes, M_0 and M_1, where the former consists of actor A and latter actor B. All other actors do not belong to any mode, as they fire once per iteration, independently from the values of the mode control tokens. The MC produces control tokens that are sent to the control input port of the *switch (SW)*, *select (SL)*, and tunnel. Tunnel actors encapsulate an MCDF construct enabling well-defined communication between different modes, as explained below. Besides the control input port, a SW has a data input port and a number of output ports that connect to actors belonging to different modes. The SW consumes both the data token sent by the source (Src) actor and the control token given by MC, and produces the same data token on the output port that connects to the mode specified by the control token. Conversely, a SL consumes the control token and the input data token associated with the mode indicated by its received control token, and produces the same data token on the output port. Figure 5.2 also shows a tunnel constructed by a switch (Tsw) and a select (Tsl). It has an internal (initial) token that is always replaced by its input data token. This is achieved when Tsw and Tsl receive the same control token indicating a mode that connects to the data input of the tunnel, e.g., M_0 in Figure 5.2. As a result, it always passes the latest token from the input mode to the output mode via the data out port.

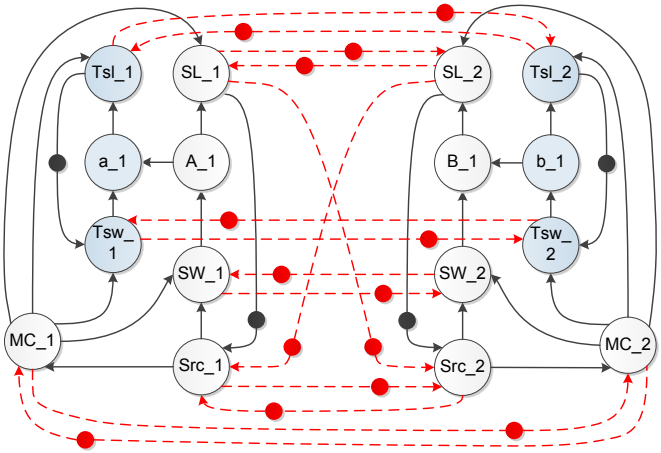


Figure 5.4: Merging the equivalent SRDF graphs of SMS_0 and SMS_1. This results in the equivalent SRDF graph of $[SMS_0 \mid SMS_1]^*$.

The execution of the MCDF graph is highly dependent on how modes are chosen. To capture the static behavior of a system, *a pre-defined static mode sequence (SMS) that specifies a static order of modes to fire can be used.* Moreover, multiple SMSs can be used dynamically in any random order. In addition, MC can repeat an SMS, resulting in *recurring SMS.* By choosing modes according to pre-defined SMSs, a *worst-case throughput analysis* of the MCDF model can be based on the SMSs. Each SMS specifies a static firing order of modes. *The firing dependencies of a recurring SMS are hence equivalently described by a static dataflow graph, which is obtained by eliminating the actors and edges (i.e., dependencies) of the modes that are not chosen by the SMS.* We simply assume that SMS_0 only contains mode M_0 and SMS_1 has mode M_1 in our example in Figure 5.2, i.e., SMS_0=$[M_0]$ and SMS_1=$[M_1]$. When SMS_0 or SMS_1 is repeatedly used by MC, recurring mode sequences are brought and are represented by $[M_0]^*$ and $[M_1]^*$ for SMS_0 and SMS_1, respectively. The equivalent SRDF graphs are shown in 5.3(a) and 5.3(b) for SMS_0=$[M_0]^*$ and SMS_1=$[M_1]^*$, respectively. For a recurring SMS_3=$[M_0, M_1]^*$, its equivalent SRDF is shown in 5.3(c) that is obtained by unrolling the MCDF model in Figure 5.2, where M_0 is always followed by M_1 and the transitions are denoted by the red dashed edges. Therefore, to analyze the worst-case throughput of a given SMS, we only need to analyze its equivalent static dataflow graph with existing dataflow analysis techniques [79].

The transitions across multiple SMSs are usually not known apriori, since they are dynamically executed. All the possible transitions can be described by a finite-state machine (FSM), where each SMS is represented by a state that is able to transit to any states including itself. By assuming the total number of SMSs to be NS ($NS > 0$), this case is described by $[SMS_0 \mid SMS_1 \mid ... \mid SMS_{NS-1}]^*$, where the transitions are given by the FSM.

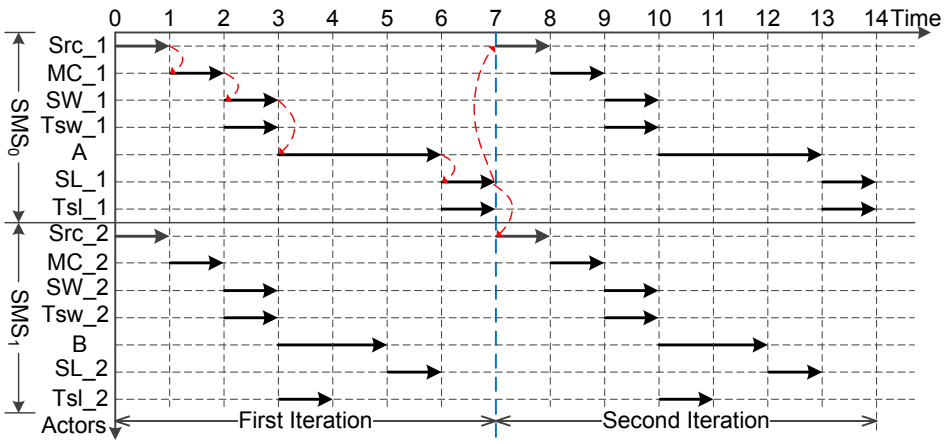


Figure 5.5: The execution of the merged equivalent SRDF graph shown in Figure 5.4.

The worst-case analysis approach in [68] actually does not need to explore all the transitions of the FSM to obtain the worst-case results of $[SMS_0 \mid SMS_1 \mid ... \mid SMS_{NS-1}]^*$ for an MCDF model. Instead, it merges all the equivalent static dataflow graphs of each individual recurring SMS (e.g., SMS_i, $\forall i \in [0, NS-1]$), resulting in a larger equivalent static dataflow graph that captures all the dependencies of dynamically executing an arbitrary SMS_i. As a result, the worst-case throughput analysis only needs to focus on a single giant/merged graph, where the existing dataflow analysis techniques can be efficiently employed. This merging is achieved by adding all the dependencies (i.e., edges with initial token(s)) between the actors chosen by different SMSs. For example, Figure 5.4 shows the merging of the SRDF graphs of $SMS_0=[M_0]^*$ and $SMS_1=[M_1]^*$, which are shown in 5.3(a) and 5.3(b). The added dependencies are denoted by red dashed edges, where each of them is given an initial token. The merged graph is the equivalent SRDF of executing $[SMS_0 \mid SMS_1]^*$. Therefore, it only requires to equivalently analyze the merged static dataflow graph to derive the worst-case results.

Figure 5.5 illustrates the execution of each actor in Figure 5.4 during two iterations. The execution trace in Figure 5.5 shows that the actor firings of each SMS in a new iteration depend on the slowest SMS executed in the previous iteration. *Therefore, the worst-case situation is guaranteed for any SMS that is dynamically executed.* As highlighted by the red dashed arrows in Figure 5.5, the firing of both SMS_0 and SMS_1 in the second iteration starts after the finishing of SMS_0 in the first iteration. The reason is that actor A has the longest execution time (i.e., 3) in the MCDF graph in Figure 5.2, resulting in the critical path (shown by the red dashed arrows) of executing SMS_0 in the first iteration. Therefore, in this case, the repeated SMS_0, i.e., $[SMS_0]^*$, is the dominate mode sequence that leads to the worst-case results.

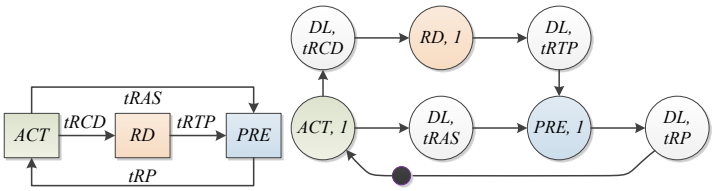
5.3 MCDF MODEL OF RUN-DMC

This section firstly discusses the general principles of modeling memory command scheduling in dataflow, followed by introducing the MCDF model of command scheduling for DDR3 SDRAMs, which includes a generalization of the tunnels used by the MCDF model and a description of how memory transactions are supported by using static mode sequences.

5.3.1 *Dataflow Modeling of Command Scheduling*

The timing dependencies of command scheduling are essentially the same as the data dependencies described by dataflow graphs. A command can be scheduled only if all its timing constraints are satisfied, while a dataflow actor fires when all its input tokens are available. For example, the timing dependencies between commands are depicted in Figure 2.8, where a command can be scheduled only if all its inputs become valid, i.e., the timing constraints from previous commands are satisfied. The scheduling time of a command is computed based on the relevant timing constraints, whichever is the largest. For example, Eq. (4.1) computes the scheduling time of an arbitrary *ACT* command. Principally, Eq. (4.1) represents a form of max-plus algebra [47], which has been used in the dataflow throughput analysis [22, 28]. Essentially, we can model the memory controller with dataflow, and the existing analysis tools can be used to analyze the worst-case results. The dataflow modeling of a memory controller actually captures the command scheduling dependencies by means of 1) modeling each command as an actor and setting its execution time as the time spent on the command bus; 2) tracking the timing constraints by using delay (DL) actors, whose execution times are equal to the constant values of the DDR3 JEDEC-specified timing constraints [53]. Note that the model can be easily used for different generations of DDR SDRAM and also for different devices of the same generation [39] by replacing the constant values of the timing constraints; 3) capturing the command scheduling dependencies by adding edges between the actors of commands and timing constraints. For example, Figure 5.6 illustrates an example of modeling a simple periodic schedule of an *ACT*, *RD*, and *PRE* to a bank (Figure 5.6(a)) with a static dataflow graph (Figure 5.6(b)) according to the above scheme.

A memory transaction is executed by interleaving it over BI banks, each of which requires an ACT, followed by BC times of RD or WR and finally a PRE, according to the close-page policy. The command scheduling of a particular transaction in terms of specific type (read or write), BI, BC, and physical address (starting bank) can be modeled by a specific static dataflow graph, such as the simple example shown in Figure 5.6. However, different static dataflow graphs are needed to capture the command scheduling of transactions with different types, sizes, and physical addresses. *We propose an MCDF model capturing the command scheduling dependencies of various transactions by specifying static mode sequences (SMSs), where a mode sequence is equivalent to a static*



(a) Cmd scheduling of a bank (b) The dataflow graph

Figure 5.6: An example of dataflow modeling of commands to a bank.

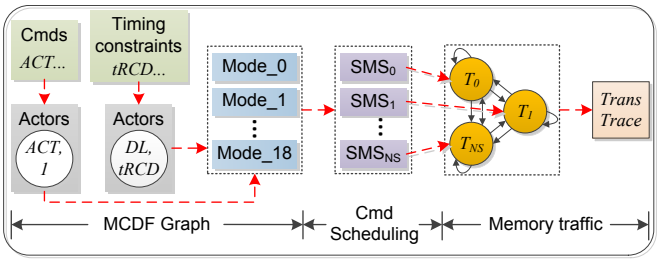


Figure 5.7: An overview of the MCDF modeling of memory controllers.

dataflow graph representing the execution of a transaction with the given type, size, and starting bank. Finally, the MCDF graph must be strongly connected to restrict the firing rate of actors, such that the timing behavior of the memory controller can be correctly captured. In fact, the strongly connected MCDF graph captures the limited speed of command scheduling because of the timing constraints.

Figure 5.7 shows a high-level overview of the MCDF modeling of memory controllers. As previously mentioned, memory commands and the SDRAM timing constraints are modeled by actors, which further constitute each individual mode (shown in Figure 5.8). The scheduling of commands per transaction is captured by using a proper pre-computed static mode sequence (SMS), which has been discussed in the previous Section 5.2.2. The generation of proper SMSs for transactions will be later presented in Section 5.3.2.1. These SMSs are dynamically employed to model the memory traffic. A fully connected FSM can describe the transitions among all kinds of transactions associated with the SMSs (NS denotes the total number) if there is no apriori information about the traffic (discussed in Section 5.4). Overall, the MCDF graph naturally captures commands and timing constraints of SDRAM and can be generally used by various memory controllers and memory devices by computing their corresponding SMSs. When static knowledge about the traffic is provided, e.g., the known transaction sequence given by the application or by memory arbiter (e.g., TDM), the FSM is simply restricted to keep the known sequence.

5.3.2 *MCDF Modeling of Command Scheduling*

We proceed by generally modeling command scheduling of transactions with an MCDF graph based on the previously mentioned principles. There are four different commands that are used to execute each transaction, *ACT*, *RD*, *WR*, and *PRE*. Note that the refresh (*REF*) command is not modeled explicitly because it is only needed for a large interval of tREFI and reduces the bandwidth by approximately 3% [55]. Its effect will be taken into account later when computing the worst-case bandwidth in Section 5.4. The commands can be dynamically scheduled to the required banks according to various scheduling algorithms subject to the inter/intra-bank timing constraints. To generally support any command scheduling algorithm for transactions, *the scheduling of a command to each bank can be modeled by creating a mode that has actors representing the command and the inter/intra-bank timing constraints.* In particular, the actors of inter-bank timing constraints are used to support the transitions across modes. Finally, *the execution of a transaction is modeled by using a mode sequence that specifies the order of modes corresponding to the required commands.* In this way, various command scheduling algorithms for transactions can be supported by specifying their mode sequences.

Figure 5.8 shows the MCDF model of command scheduling. It consists of 18 modes, representing the scheduling of different memory commands to any of the 8 banks. Each mode consists of a command (*ACT*, *RD*, *WR*, or *PRE*) actor and several delay (DL) actors that track the relevant timing constraints. The edges between actors in Figure 5.8 show the dependencies. Since the command bus transfers one command per cycle, the execution time of all the command actors is 1 cycle except for *ACT* actors where it is 2 cycles. This is because an *ACT* has lower priority than a *RD* or a *WR* as stated in Section 3.3.2. We hence conservatively assume an *ACT* always collides with a *RD* or *WR*, resulting in one cycle additional delay. The execution times of DL actors in Figure 5.8 are configured to be the values of the JEDEC timing constraints in Table 2.1.

The *ACT* and *PRE* commands to different banks have to be modeled by different modes because of the intra-bank timing constraints. For example, the scheduling of an *ACT* has to satisfy the *tRP* constraint from the previous *PRE* to the same bank, as shown in Figure 2.8. Mode_0 to Mode_7 in Figure 5.8 model the *ACT* to a bank from Bank 0 to Bank 7, respectively. While Mode_8 to Mode_15 capture the *PRE* to a bank from Bank 0 to Bank 7, respectively. For $\forall i \in [0, 7]$, the transition between Mode_i and Mode_(i+8) captures the timing constraints (e.g., tRAS and tRP) between the *ACT* and *PRE* to the same bank. Note that the transition between modes is supported by tunnels. A basic tunnel is shown in Figure 5.2 that only supports transition from one mode to another. The proposed MCDF model in Figure 5.8 requires a general $M \times N$ tunnel that supports the transition from M modes to N modes, where $M > 0$ and $N > 0$. In the same way, the timing constraints between ACT commands are also captured, such as the *tRRD* and *tFAW*. Finally, *RD* or *WR* commands are sequentially scheduled due to the the shared data bus of transferring their associated data, and it is hence not nec-

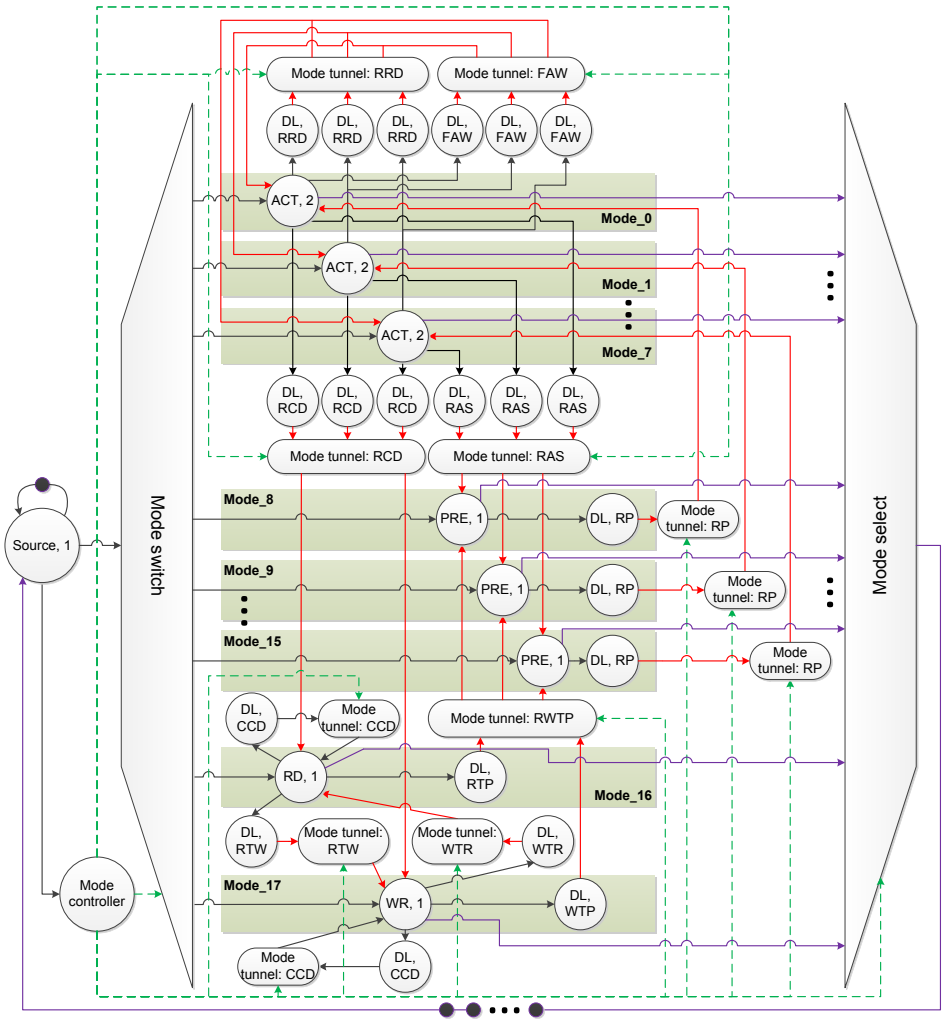


Figure 5.8: Mode-controlled dataflow model of memory command scheduling.

essary to distinguish different banks. Therefore, *RD* and *WR* are modeled by Mode_16 and Mode_17, respectively, where all the relevant timing constraints are captured by tunnels. For example, the RCD Tunnel keeps the *tRCD* constraint from an ACT to a RD or WR, as shown in Figure 5.8. The general $M \times N$ tunnel will be detailed later in Section 5.3.2.2.

The Source actor in Figure 5.8 triggers the command scheduling each clock cycle by producing a token on the input port of the Mode switch, and its execution time is 1. This models the worst-case behavior, where pending transactions are backlogged, i.e., enough commands ensure the scheduler is always busy. The Mode controller (MC) determines which mode to choose, i.e., which command to schedule, by specifying the mode sequence corresponding to a transaction. Therefore, when the Source actor produces a token to trigger a mode, it also gives a token to MC that produces a control token based on the mode sequence for all the switch, select, and tunnel actors to choose the mode. The translation from transactions to mode sequences will be detailed later in Section 5.3.2.1.

The memory controller schedules commands with limited speed due to the timing constraints. This is captured by a feedback edge from the Mode select to the Source actor (see Figure 5.8), which makes the proposed MCDF model strongly connected. The token on this edge is produced by the Mode select that is triggered after the firing of each command actor per mode, and the token is then consumed by the Source to produce a new token to trigger the next command actor, i.e., schedule a new command. Note that the initial tokens on this edge must guarantee that commands are scheduled as soon as all timing constraints are satisfied. The proper number of the initial tokens will be obtained from experiments.

5.3.2.1 *Mode Sequences*

The MCDF model in Figure 5.8 is able to capture the dependencies of different command scheduling mechanisms, e.g., close/open-page policy, bank privatization/interleaving, and priorities. The reason is that the proposed MCDF model captures the JEDEC-specified timing constraints of an SDRAM. In addition, *the execution of transactions by a memory controller using particular mechanisms is captured by creating the appropriate mode sequences, which specify the firing order of modes, and hence the order of commands.* To create a mode sequence for executing a transaction with specific size, type, and physical address, we firstly create a mode sequence per bank, and then combine these per-bank mode sequences for the transaction according to its required banks.

Take the scheduling algorithm (i.e., Algorithm 2) of dynamic command scheduling as an example. We show next how mode sequences are created for it. The dynamic command scheduling is discussed in Section 3.3, where a transaction interleaves over BI banks and there are BC data bursts per bank. This requires commands to be scheduled to all BI banks, where each of them receives an *ACT*, followed by BC number of *RD* or *WR* commands and finally a *PRE* (see Figure 2.8). Therefore, *the mode sequence for*

each bank must be an ACT mode, BC times of RD or WR mode and a PRE mode. This is given by Definition 14 that defines the mode sequence *ms*(k, BC) for an arbitrary bank k ($\forall k \in [0, 7]$). As shown in Figure 5.8, Mode_k captures the *ACT* command to bank k and the mode for the *PRE* command to the same bank is Mode_(k+8). The mode number for the BC number of *RD* or *WR* commands is Mode_16 or Mode_17, as given by Eq. (5.1).

Definition 14 (Mode sequence per bank)**.** *For* $\forall k \in [0, 7]$ *and* $\forall l \in [0, BC-1]$, *ms(k, BC) = [Mode_k,* RW_0*, ...,* RW_l*, ...,* $RW_{BC\text{-}1}$*, Mode_(k+8)].*

$$RW_l = \begin{cases} \text{Mode\_16}, & \textit{RD command} \\ \text{Mode\_17}, & \textit{WR command} \end{cases} \tag{5.1}$$

For an arbitrary transaction T_i ($\forall i \geq 0$) that uses BI_i and BC_i, its corresponding mode sequence $MS(T_i)$ is given by Definition 15, which is a sequential combination of the BI_i number of mode sequences per bank.

Definition 15 (Mode sequence per transaction)**.** *For* $\forall i \geq 0$ *and* $\forall j \in [0, BI_i - 1]$, $MS(T_i)$ *= [ms(bs,* BC_i*), ..., ms(bs+j,* BC_i*), ..., ms(bs+*$BI_i - 1$*,* BC_i*)], where bs is the starting bank of* T_i*.*

For example, a read transaction has BI=2 and BC=1, and its starting bank is Bank 0, i.e., *bs*=0. The mode sequences for the two banks Bank 0 and Bank 1 are [Mode_0, Mode_16, Mode_8] and [Mode_1, Mode_16, Mode_9], respectively. Therefore, the mode sequence for the transaction is the combination of these two mode sequences per bank, which is [Mode_0, Mode_16, Mode_8, Mode_1, Mode_16, Mode_9]. Note that the mode sequence is only used by the MC to trigger different modes sequentially, while the actual firings of the command actors may occur in a different order, since the firings rely on the dependencies between actors. Therefore, this enables command scheduling pipelining.

5.3.2.2 *General Tunnels*

The tunnels of the MCDF model, previously shown in Figure 5.8, are used to support the transitions between modes, and they need multiple data inputs and data outputs. For example, tRCD is the timing constraint from an *ACT* to a *RD* or *WR* command to the same bank. So, the RCD tunnel has to support transition from one of the ACT modes (i.e., Mode_0 to Mode_7) to either Mode_16 or Mode_17 corresponding to *RD* or *WR* commands. As a result, the basic tunnel shown in Figure 5.2 has to be extended, since it only has a single data input and output. We generalize these tunnels to an $M \times N$ tunnel that has M data inputs and N data outputs, as depicted in Figure 5.9. In addition, it consists of a single internal token. This generic tunnel is instantiated to support all the tunnels in Figure 5.8 except the FAW tunnel that captures the tFAW constraint to restrict the scheduling of at most four *ACT* commands within the time window. A single internal

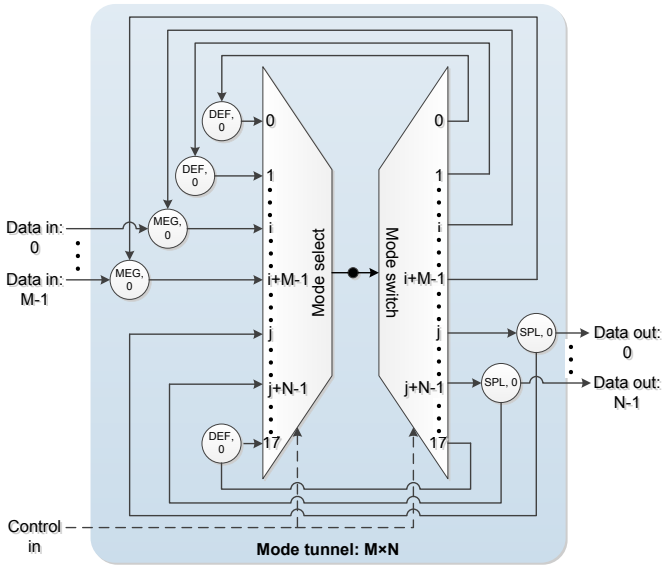


Figure 5.9: A generic mode tunnel with M inputs and N outputs.

token cannot support tFAW. The FAW tunnel is designed with a cascade structure of four internal tokens, as shown in Figure 5.10.

The generic tunnel presented in Figure 5.9 consists of a switch and a select, and the edge between them has an initial token (i.e., internal state). The switch has 18 inputs while the select contains 18 outputs corresponding to all the 18 modes in the MCDF model. For an arbitrary input/output of the select/switch m ($\forall m \in [0, 17]$), the corresponding mode is Mode_m in Figure 5.8. The tunnel is instantiated to support M data inputs and N data outputs, where $\forall M, N \in [1, 18]$. It also has one control input that delivers control tokens sent by the MC to the switch and select. The M inputs correspond to the modes from Mode_i to Mode_(i+M-1), while the N outputs are associated with Mode_j to Mode_(j+N-1), where $\forall i \in [0, 18 - M]$ and $\forall j \in [0, 18 - N]$.

Each data input of the generic tunnel firstly connects to an actor with 0 execution time, called *merger* (MEG), and it consumes both the input data token and the internal token and produces the same data token. Note that the internal token is forwarded by the switch (see Figure 5.9). The token produced by the MEG is consumed by the select when the control token indicates the mode corresponding to this data input, and the select produces the same token that becomes the new internal token. In this way, the internal state is updated. For a data output of the generic tunnel in Figure 5.9, the output token is provided by an actor, namely *splitter* (SPL) that is connected by the output of the switch corresponding to the same mode as the data output. The execution time of a

SPL is 0. The output of the switch forwards the internal token to the SPL that produces the data output token and also returns it to the internal state via the select.

The select and switch of a tunnel fire by consuming both the input token and the control token. They may receive a control token that is not associated with any data input or output of the tunnel, since the MC produces each control token for all select and switch actors. A default actor (DEF) with the mode indicated by this control token is used to connect the output of the switch to the input of the select, which correspond to the same mode. The execution time of DEF is 0. The DEF enables both the switch and select to consume the control tokens not associated with the M inputs and N outputs.

We proceed by introducing the connections of the M data inputs and the N data outputs. For an arbitrary data input k ($\forall k \in [0, M-1]$) with the corresponding Mode_(i+k), it connects to a MEG that further connects to the $(i+k)^{th}$ input of the select in Figure 5.9. An output of the select connects to a SPL that connects to an arbitrary data output h ($\forall h \in [0, N-1]$) corresponding to Mode_(j+h). If $\exists h$ such that j+h = i+k (i.e., the same mode), an output of the SPL connects to the input of the MEG. Otherwise, the $(i+k)^{th}$ output of the switch connects to the input of the MEG. In addition, one of the outputs of the SPL goes back to the $(j+h)^{th}$ input of the select.

5.3.2.3 *Cascade FAW Tunnel*

The FAW tunnel in Figure 5.8 captures the tFAW constraint (in Table 2.1) that allows maximally 4 *ACT* commands to be scheduled within the rolling time window. It supports any transitions amongst Mode_0 to Mode_7. As a result, the FAW tunnel consists of 8 data inputs and 8 data outputs, which connect Mode_0 to Mode_7. Note that we cannot simply add four internal tokens to the generic tunnel in Figure 5.9 to support tFAW. It is because all the four internal tokens may be consumed when the modes indicated by control tokens are not between Mode_0 to Mode_7, which correspond to the scheduling of *ACT* commands. Therefore, when a control token for a mode between Mode_0 to Mode_7 arrives, it cannot be consumed by the select since the four internal tokens have already been consumed. To overcome this problem, a cascade tunnel with four pairs of select and switch is designed, as shown in Figure 5.10, where the internal state of each pair contains an initial token. These four initial tokens allow at most 4 different *ACT* modes execute within the tFAW time window. When one of them is triggered, an initial token of the FAW tunnel is consumed by its *ACT* command actor and a new token will be produced by its DL actor with the execution time of tFAW. This new token goes to one of the data inputs of the FAW tunnel to update the internal state.

The execution of an *ACT* mode requires one internal token of the FAW tunnel (see Figure 5.8). After tFAW cycles, the FAW DL actor of the *ACT* mode produces a token to update the internal state, such that new *ACT* mode can be triggered. The four initial tokens of the FAW tunnel are able to trigger four *ACT* modes, while the fifth one has to wait for an internal token that is updated by the first *ACT* mode after tFAW cycles. In this way, the rolling tFAW constraint is captured. When the FAW tunnel receives control

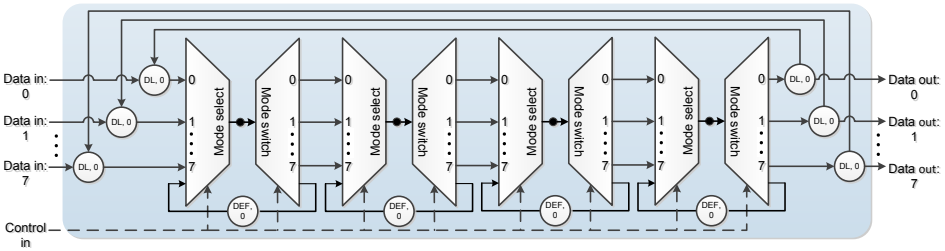


Figure 5.10: A cascade tunnel structure to support multiple initial tokens for a specific set of modes.

tokens for modes from Mode 8 to Mode 17, they are consumed by each pair of switch and select through the default connection, i.e., the edge with a DEF actor, as shown in Figure 5.10. Note that an internal token can be transferred to the next pair of select and switch or the data output of the FAW tunnel only if a control token for Mode 0 to Mode 7 is received. So, the firing of an *ACT* mode either gets an internal token or waits for the update of the internal state when the tFAW constraint is met.

The tunnels previously discussed in Section 5.3.2.2 and Section 5.3.2.3 are composed of normal actors, switch and select, as shown in Figure 5.9 and Figure 5.10. It is time-consuming when manually specifying all the tunnels in the MCDF model shown in Figure 5.8. As a part of this work, a tool is developed to automatically decompose a tunnel to normal actors, switch, and select, which are connected by relevant edges. Consequently, a MCDF graph can be easily described with nodes and edges, where each tunnel is represented by a node. The MCDF graph is then decomposed by the tool into a graph with only normal actors, switch, and select.

5.4 WORST-CASE BANDWIDTH

Bandwidth is defined by Definition 10 to be the long-term data transferring rate of executing transactions. The lower bound on the worst-case (minimum) bandwidth (WCBW) has been computed based on the worst-case execution time of an individual transaction, as given in Section 4.6. However, this bound is pessimistic, because it cannot exploit the pipelining between transactions. This section analyzes the WCBW bound using the MCDF analysis technique [68], which essentially transfers the MCDF model with the specific mode sequences to equivalent SRDF model, as briefly introduced in Section 5.2.2. Note that the analysis of an SRDF model is algorithmically easy. The typical analyses of dataflow models provide the MCM (tokens/second) of iteratively executing the graphs. Instead, we use worst-case bandwidth (bytes/second) to define the critical cycle path of executing the MCDF graph, resulting in a WCBW bound rather than the typical MCM. Note that MCM is defined as the maximum of the total execution time of the critical cycle divided by the total number of initial tokens on the critical cycle. Hence,

the minimum throughput (transactions per second) of the memory controller is 1/MCM. However, the minimum throughput is not always equivalent to the WCBW (bytes/second). For example, the critical cycle can be obtained based on large transactions that consume more time than small ones, but carry more data. Since the critical cycle can be repeated for an infinitely long time, i.e., executing an infinite number of transactions, the obtained WCBW bound is obtained exploiting the pipelining between a sequence of transactions. In addition, it is the bound on the long-term worst-case bandwidth.

The proposed MCDF model can capture the command scheduling behavior of the memory controller by specifying static mode sequences (SMSs) for all kinds of transactions in terms of read or write, sizes and different sets of banks. To analyze the WCBW of $[SMS_1 \mid SMS_2 \mid ... \mid SMS_{NS}]^*$ by using the analysis technique introduced in Section 5.2.2, the key issue is to obtain all these *NS* number of SMSs.

Definition 15 previously defined the method to derive the mode sequence for a transaction, which requires information about the transaction type, size, and physical address. The type determines whether *RD* or *WR* commands are needed, and hence the corresponding Mode_16 or Mode_17 in Figure 5.8. The size is mapped to *BI* and *BC*, while the physical address gives the starting bank (b_s). For example, when a system only generates 64-byte read and write transactions, e.g., the L2 cache line size is 64 bytes for all cores, the most efficient configuration of BI=4 and BC=1 is used to access a DDR3 SDRAM with a 16-bit data bus [39]. Since DDR3 SDRAMs have 8 banks, the transactions may either interleave consecutively over Bank 0 to Bank 3 or Bank 4 to Bank 7 for alignment reasons [39]. Therefore, four SMSs ($[SMS_1 \mid SMS_2 \mid SMS_3 \mid SMS_4]^*$) are needed, two for reads and write, respectively, to each possible starting bank. Take a read transaction interleaving over Bank 0 to Bank 3 as an example. The SMS_1 is [Mode_0, Mode_16, Mode_8, Mode_1, Mode_16, Mode_9, Mode_2, Mode_16, Mode_10, Mode_3, Mode_16, Mode_11]. Similarly, the rest of the SMSs can be obtained. When a static transaction sequence is known, e.g., by using a TDM arbiter in the front-end (c.f., Section 3.2.2), a larger SMS can be obtained by sequentially concatenating the SMS of each transaction with the sequence. The WCBW is hence analyzed based on the combined SMS that guarantees the static transaction sequence. Note that a tool is developed to automatically generate all the mode sequences for transactions served in specified orders.

As introduced in Section 5.2.2, the analysis of the MCDF model is performed by merging the equivalent static dataflow graphs of each SMS, resulting in a larger static dataflow graph that captures the dependencies of executing $[SMS_1 \mid SMS_2 \mid ... \mid SMS_{NS}]^*$. The critical cycle defined by the MCM is obtained when executing the merged static dataflow graph, and it consists of a number of actors belonging to a single mode or different modes, which are specified in one or more SMS(s). As a result, these SMSs lead to the worst-case situation, i.e., the corresponding transactions experience a maximum average time (i.e., MCM) to execute each of them. It is worth noting that these critical transactions are automatically obtained from the analysis of the MCDF model, while the formal analysis approach in Chapter 4 is unable to manually figure them out.

According to Definition 10, the bandwidth of the memory controller depends on the sizes of the transactions and their execution times. The critical cycle of the merged static dataflow graph must be defined as the cycle that provides WCBW rather than MCM. Similarly to the definition of MCM, the WCBW ($\hat{bw}$) is defined by Eq. (5.2). For every cycle C of the MCDF graph G, the total execution time of the actors on C is denoted by $|C|$, while the total number of initial tokens (or delays) on the edges of C is $\omega(C)$. In addition, the total number of SMSs associated with C is $NS(C)$. Each SMS is used by a transaction and its size is S_i ($\forall i \in [1, NS(C)]$). However, it is not guaranteed that the Heracles analysis tool [79] of MCDF model can handle the WCBW defined by Eq. (5.2), since both $\omega(C)$ and S_i vary with C. As a result, we simply assume $\omega(C) = 1$, such that conservative WCBW can be provided by Heracles, which will be later used in our experiments.

$$\hat{bw} = \min_{\forall C \in G} \frac{\omega(C) \times \sum_{i=1}^{NS(C)} S_i}{|C|} \times f_{mem} \times e^{ref} \tag{5.2}$$

The WCBW given by Eq. (5.2) is a new notion for defining the critical cycle of the merged static dataflow graph to provide the WCBW. We can extract the worst-case order of transactions from the critical cycle, which can be used to design better scheduling algorithms to eliminate this bottleneck and obtain a better WCBW bound.

5.5 EXPERIMENTAL RESULTS

This section proceeds by experimentally showing the WCBW of a dynamically scheduled memory controller, analyzed based on the proposed MCDF model. The experimental setup is given, followed by validating the MCDF model for fixed transaction size and variable sizes, respectively. The results are compared to state-of-the-art analysis approaches.

5.5.1 *Experimental Setup*

The proposed MCDF model has been verified and analyzed with Heracles [79], a temporary analysis tool developed at Ericsson. It runs on a 64-bit Ubuntu 14.04.3 LTS system with 8 Intel(R) Core(TM) i7 CPUs running at 1.6 GHz and with 24 GB RAM. We use similar experimental setups as previously in Chapter 3 and Chapter 4. The transaction sizes used by the experiments include 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes. We have chosen the memory map configuration (i.e., *BI* and *BC*) for each size that provides the lowest execution time (i.e., higher memory bandwidth) by interleaving over more banks to exploit bank parallelism. The configured (*BI*, *BC*) for these transaction sizes are hence (1, 1), (2, 1), (4, 1), (4, 2), and (4, 4) [39]. Note that (4, 2) and (4, 4) are used by 128 Byte and 256 Byte transactions instead of (8, 1) and (8, 2), because of *tFAW*

that causes a larger execution time with *BI*=8. We do not specify the number of requestors in the front-end of Run-DMC, since the total bandwidth is determined by the execution of transactions in the back-end. The allocation of bandwidth per requestor depends on the arbitration in the front-end and is not a main concern throughout this thesis. Experiments have been done with JEDEC-compliant, DDR3-800D, DDR3-1600G, and DDR3-2133K, all with interface widths of 16 bit and a capacity of 2 Gb [53].

5.5.2 *Validation of MCDF Model*

This experiment validates that the proposed MCDF model conservatively captures the command scheduling behavior of a dynamically scheduled memory controller. This is achieved by comparing the scheduling time of each command obtained by executing the MCDF model to that given by the open-source scheduling tool *RTMemController* previously presented in Section 4.7, which implements the timing behavior of Run-DMC. First, we have to find the proper number of initial tokens on the feedback edge of the MCDF model in Figure 5.8. This is achieved by experimentally increasing the initial tokens until the feedback edge cannot dominate in any command scheduling. This is achieved by simulating the MCDF graph for the given SMSs. In addition, the feedback edge is also ensured to exclude from the critical cycle. This is achieved by manually increasing the number of initial tokens until the feedback edge is not included in the critical cycle. The experimental results show that 20 initial tokens are enough for DDR3 SDRAMs and they are used by the following experiments. This experimental method is a quick, safe, and easy way to derive the proper number of initial tokens, such that the proposed MCDF model accurately captures the command scheduling of the memory controller. For new memory devices, we can derive the proper number of initial tokens on the feedback edge in the same way.

The five transaction sizes have been tested by specifying all possible mode sequences. The MCDF model executes every mode sequence independently during 40,000 cycles and all the command scheduling times are obtained. This experiment repeats the mode sequence, i.e., simulates the execution of the same transactions using the scheduling tool. Note that we also apply the collision assumption for each *ACT* command to the scheduling tool, such that it runs under the same assumption as the MCDF model. The scheduling times given by these two approaches *are identical* for all commands. This observation also holds for other experiments, where we have mixed the mode sequences corresponding to different transactions in terms of different sizes, read or write, and different banks. *We hence conclude that the proposed MCDF model conservatively captures the timing behavior of the memory controller.*

5.5.3 *Worst-Case Bandwidth*

This section presents the WCBW given by the analysis of the MCDF model. The results are also compared to those given by the scheduled and analytical approaches of dynamic command scheduling in Chapter 4 and the semi-static approach in [3]. Those approaches compute the WCBW based on their WCET of transactions. Remember that the collisions for *ACT* commands are actually detected by the scheduled approach, while collisions are always assumed by the analytical approach. The semi-static approach uses pre-computed command schedules with fixed lengths in cycles, and the WCBW is obtained based on them.

5.5.3.1 *Fixed Transaction Size*

This experiment is carried out to evaluate the WCBW provided by the memory controller when it only executes transactions with fixed size, such as when all cores have the same cache-line size. The experiment is executed for five different cache-line sizes of 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes with different memory map configurations, respectively.

Figure 5.11 gives the WCBW results obtained from the MCDF model. They are compared to that given by the analytical, scheduled, and semi-static approaches, respectively. We can observe that 1) the MCDF model always outperforms the analytical approach, where the maximum improvement is 22.0% for 64 byte transactions and the average improvement reaches 6.3% for all the sizes. The improvement is achieved because the MCDF analysis technique provides WCBW results without assuming the worst-case initial bank states that are needed by the analytical approach. 2) It is also better than the scheduled approach with a single exception for 16 bytes, where it is 2.4% less. The reason is that the MCDF model only conservatively assumes a collision per *ACT* command. In contrast, the scheduled approach has to assume the worst-case initial bank states, leading to pessimistic WCBW bounds. However, it can actually detect the collisions for *ACT* commands. As a result, the scheduled approach provides slightly better bound for 16-byte transactions, which only use a bank and there is no collision for the *ACT* command in the worst-case situation. 3) This exception also applies when comparing to the semi-static approach that statically resolves command collisions at design time. However, for large transaction sizes (e.g., 256 bytes), the MCDF model provides higher WCBW. The reason is that the semi-static approach uses pre-computed static command schedules, which have to be repeatedly used for transactions, resulting in inefficient pipelining of *ACT* commands across transactions. In contrast, the dynamically-scheduled memory controller efficiently hides the latency of issuing *ACT* commands in pipelining with large number of *RD* or *WR* commands of large transactions. These observations also hold for other DDR3 SDRAMs, although the results are not shown for brevity.

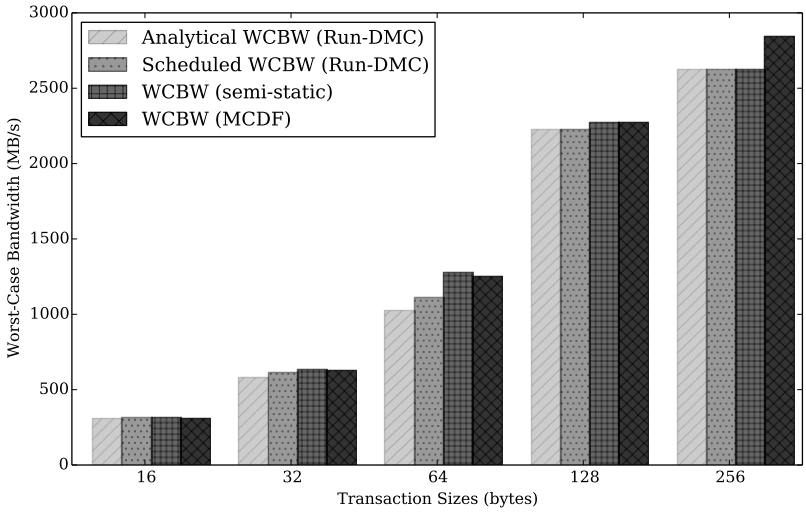


Figure 5.11: The WCBW given by different analysis approaches for DDR3-1600G SDRAM with fixed transaction size.

5.5.3.2 *Variable Transaction Sizes*

In this experiment, the memory controller receives transactions with variable sizes, which are generated by different requestors in a heterogeneous system, such as a High-Definition video and graphics processing system featuring a CPU, hardware accelerators and peripherals with variable transaction sizes [31]. If there is no static information about the transactions, e.g., the execution order of different transaction sizes, we have to conservatively analyze the WCBW results by assuming any possible transaction order. However, when requestors with different transaction sizes are served by an arbiter using static schedules, such as the time-division multiplexing (TDM) proposed in Chapter 3, the static order of transactions with variable sizes is known. This static order of transaction sizes can be exploited to give less pessimistic (but still conservative) WCBW results.

This experiment considers mixed transactions with sizes of 64 bytes and 128 bytes, arriving at the memory controller with statically known or unknown order, respectively. The static order used in this experiment is that a 128 byte transaction is always followed by a 64 byte transaction and they are alternately executed by the memory controller. For instance, this can be enforced by the TDM arbiter in Chapter 3. For unknown transaction order, transactions with these two sizes may be executed in any possible order. Figure 5.12 shows the WCBW results for DDR3-1600G SDRAM given by different analysis approaches. We can see that the WCBW given by the MCDF model is always better than other approaches for both known and unknown transaction order. This indicates that the MCDF model outperforms these existing approaches because the scheduled

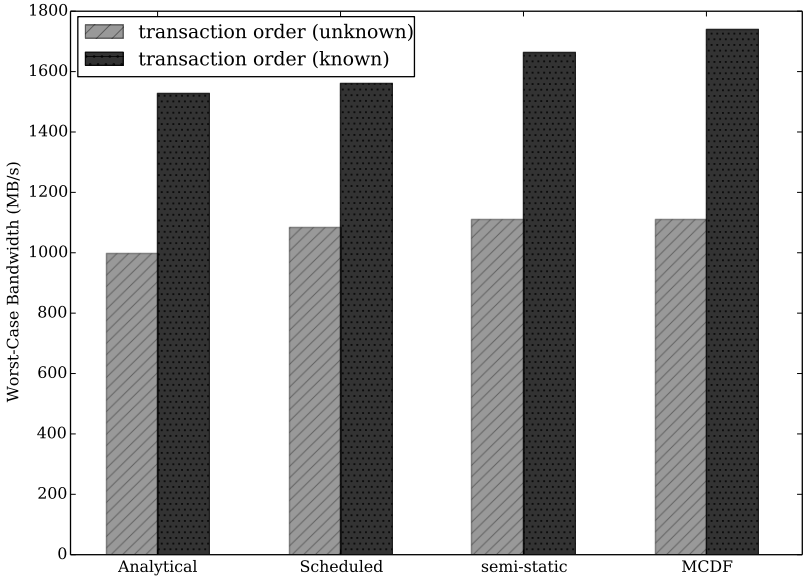


Figure 5.12: The WCBW given by different analysis approaches for DDR3-1600G SDRAM with known/unknown static order of variable transaction sizes.

and analytical approaches use conservative assumptions while the semi-static approach cannot efficiently deal with variable transaction sizes. The maximum improvement is 14% compared to the analytical approach with known transaction order. The average improvement is around 10.1% and 4.4% for known and unknown transaction order, respectively, when comparing to all these approaches. These observations also hold for other DDR3 SDRAMs.

Another experiment compares the WCBW results with known and unknown transaction order. They are analyzed by the MCDF model when applied to different DDR3 SDRAMs. The results are shown in Figure 5.13, which demonstrate that much better WCBW is consistently obtained by exploiting the static order of transactions. It achieves maximally 77.2% improvement of WCBW for DDR3-800D SDRAM, while the average improvement is 63.2% for these three tested memories.

Besides these WCBW results provided by the MCDF model, we can also obtain the worst-case situation that causes the WCBW from the critical cycle. Take DDR3-800D as an example, without knowing the static order of 64-byte and 128-byte transactions, the WCBW is provided when the memory controller repeatedly executes the transaction in the order of a 128 byte read, 64 byte write, 128 byte write, and 64 byte write. In addition, all these transactions access the same set of consecutive banks from bank 0 to bank 3. When scheduling decision is made to avoid this transaction order, better WCBW can be obtained.

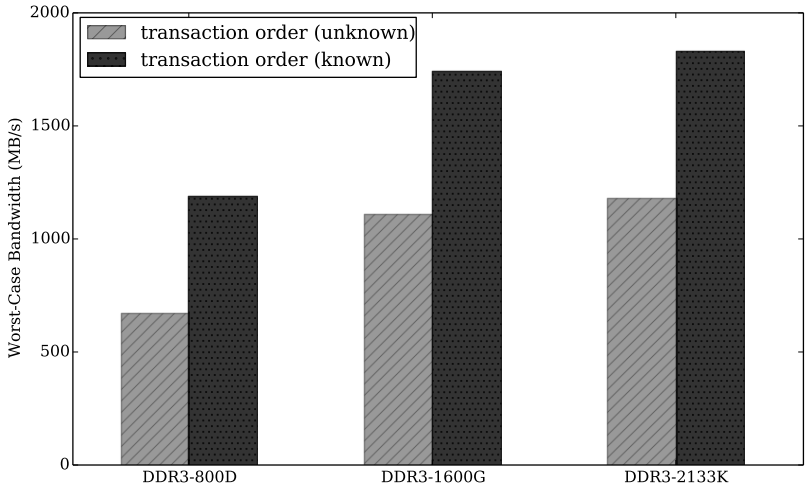


Figure 5.13: The WCBW achieved by MCDF model for DDR3 SDRAMs with known/unknown static order of variable transaction sizes.

5.6 SUMMARY

The worst-case memory bandwidth is critical to satisfy the requirements of memory-intensive real-time streaming applications in modern multi-core systems. This chapter introduces a new mode-controlled dataflow (MCDF) model to capture the scheduling dependencies of commands due to the JEDEC-specified timing constraints. The command scheduling algorithm of Run-DMC in Chapter 3 is modeled by specifying the mode sequences corresponding to transactions with different types, sizes, and addresses. Finally, the worst-case bandwidth (WCBW) is analyzed using an existing analysis technique, where a new notion (i.e., the WCBW) of the critical cycle is introduced for iteratively executing the MCDF graph.

The advantages of the proposed MCDF model are that: 1) the proposed MCDF model is general for DDR3 SDRAMs, and it is also easy to adapt to other SDRAMs, e.g., DDR4. 2) The MCDF model supports other memory controllers using different command scheduling algorithms by creating the corresponding mode sequences. The analysis is achieved by automatically using existing tools. 3) Moreover, the proposed MCDF model can easily exploit the static order of different transaction sizes based on the TDM arbitration of Run-DMC in Chapter 3. This is achieved by creating the specific mode sequences capturing the static order. As a result, much better WCBW results are obtained compared to the case with unknown order. 4) The analysis of the MCDF model provides the WCBW bounds based on the critical cycle of iteratively executing the MCDF graph. The critical cycle corresponds to a sequence of transactions, where the pipelining between them is exploited. As discussed in Section 4.6, the exploitation of pipelining is difficult

for the formal analysis approach. Moreover, the critical cycle can be repeated infinitely, resulting in a long-term lower bound on WCBW. 5) It also provides information about how transactions are executed/scheduled in the worst case, because the critical cycle corresponds to a sequence of transactions. This static information is useful for making a scheduling decision to avoid the worst-case transaction sequence, such that better WCBW is obtained. 6) Experiments have been carried out to validate the MCDF model using the open-source tool RTMemController given in Section 4.7. The experimental results demonstrate that the MCDF model outperforms the scheduled and analytical approaches given in Chapter 4 and also the semi-static approach [3]. Finally, it provides larger/tighter bounds on WCBW.

The proposed MCDF model has the limitations that 1) existing analysis technique of the dataflow model cannot accept the unpredictable collisions on the command bus. As a result, the dataflow model conservatively assumes that each *ACT* command collides with a *RD*/*WR* command, resulting in pessimism in the worst-case results. 2) It is capable of providing the long-term WCBW bounds. However, it is difficult to derive the worst-case execution/response time of individual transactions. The reason is that the analysis technique of MCDF models computes the worst-case latency based on a stable reference actor of executing the MCDF graph [68, 90]. However, it is difficult to find the reference actor when executing the proposed MCDF model for dynamic command scheduling. As a result, the existing analysis technique must be extended to deal with latency (e.g., WCET, WCRT) aspects of dynamic command scheduling.

6

TIMED AUTOMATA (TA) MODELING OF RUN-DMC

This chapter continues the modeling of real-time memory controllers, where the worst-case bounds on the response time (WCRT) of transactions and the bandwidth (WCBW) can be derived using existing techniques to automatically analyze the model. Chapter 4 presented a formal analysis approach to derive the bounds on WCRT and WCBW based on analyzing each individual transaction executed by the dynamically-scheduled memory controller Run-DMC. However, it is a approach with complex manual proofs and the bounds are pessimistic because of the collision assumption for each *ACT* command and the initial bank states obtained from as-late-as-possible (*ALAP*) command scheduling. Moreover, the formal analysis approach cannot be easily adapted to memory controller with different mechanisms. Chapter 5 previously introduced a mode-controlled dataflow (MCDF) model of Run-DMC, and the worst-case bandwidth (WCBW) bounds have been analyzed. However, the MCDF model has two limitations, where i) it also assumes a collision for each *ACT* command, resulting in pessimistic WCBW bounds, and ii) it is only capable of providing the bounds on WCBW rather than WCRT. Moreover, the MCDF model cannot be directly integrated into a system model, such as the dataflow formalization [80] of streaming applications running on a multi-processor system-on-chip (CompSOC) [35]. The reason is that the MCDF model of Run-DMC interacts with other resources (e.g., NoC, SRAM, DMA, and processor) in the system by specifying static mode sequences (SMS). However, the behavior of these resource may not be accurately captured by static mode sequences. To overcome these limitations of the formal analysis approach and the mode-controlled dataflow model, this chapter proceeds by modeling Run-DMC using Timed Automata, where model checking is applied for analysis.

Timed automata (TA) are a theory for modeling and verification of real-time systems [15]. A timed automaton is essentially a non-deterministic finite-state machine extended with real-valued variables that model the logical clocks in the system. Therefore, the timing behavior of a hardware resource can be described by a timed automaton represented by a graph containing a finite set of nodes or locations and a finite set of labeled edges. Our TA model is modular and accurately captures the behavior of Run-DMC. It models the behavior of each functional component of Run-DMC or the timing constraints of DDR3 SDRAM device with an automaton. Our TA model does not employ any simplifying abstractions as the formal analysis in Chapter 4 and the MCDF model

in Chapter 5, resulting in tighter worst-case bounds. The highlights of the TA model include: 1) *a modular publicly available TA model* [1] *of the DDR3 SDRAM device and the memory controller*. The former captures all SDRAM timing constraints, while the latter models the timing behavior of the memory controller architecture. The TA model can be easily extended or reused for different memory controllers or different DDR3 SDRAM devices. 2) We *validate our TA model with the open-source RTMemController tool* [70], previously introduced in Section 4.7. We execute random transaction traces with around 1200 commands using both the TA model and RTMemController, resulting in *identical* scheduling times of each command. This gives evidence that our TA model accurately captures the command timings of the memory controller with dynamic command scheduling. 3) *The bounds on worst-case response time and worst-case bandwidth* are derived by verifying properties of the TA model with the Uppaal model checker [13]. Uppaal provides diagnostic traces that lead to the worst-case bounds. Executing these traces with the cycle-accurate simulator provides *identical* WCRT and WCBW results as Uppaal, which speaks for the accuracy of our model. 4) Finally, the proposed TA analysis of Run-DMC reduce the WCRT bound by up to 20% and improve the WCBW bound by up to 25% compared to the analytical and scheduled approaches in Chapter 4 and the MCDF model in Chapter 5. The average improvements of the bounds on WCRT and WCBW are 7.7% and 13.6%, respectively.

In the remainder of this chapter, the background of TA is given in Section 6.1. Section 6.2 presents our TA model of the memory controller with dynamic command scheduling. Section 6.3 shows how the WCRT and WCBW bounds can be derived by verifying properties of the TA model. We then review related work in Section 6.4 before the experiments in Section 6.5. Finally, we conclude in Section 6.6.

6.1 BACKGROUND OF TIMED AUTOMATA

Timed Automata (TA) are a formal way of modeling and reasoning about the behavior of timed systems. They have been successfully used for a variety of tasks, such as conformance testing of automatic implementation of systems via code synthesis [62], and quantitative analysis of timed systems via model checking through explicit state space exploration [81]. TA essentially model a timed system based on non-deterministic state machines extended with variables. The logical clocks in the system are captured by variables, which synchronously increase. A timed automaton is described by a graph, where its nodes or *locations* represent the states and the edges between locations capture the transitions between states based on the conditions denoted by *labels* on the edges.

The Uppaal toolbox [13] implements TA as finite state machines extended with clocks and variables. Uppaal models a system as a network of several TA in parallel. States are denoted by locations. In this sense, a state of a TA is a set of active locations and a value for each clock and variable. Figure 6.1 (a) shows a TA that periodically produces memory

1 The TA model of Run-DMC is publicly available at http://www.es.ele.tue.nl/rtmemcontroller/TA.zip

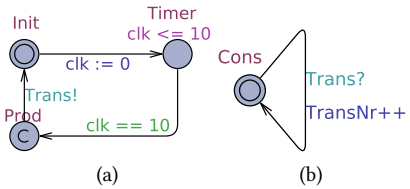


Figure 6.1: A Timed Automata model of producing and consuming transactions.

transactions. It consists of an initial location (Init), and two other locations labeled Timer and Prod that are used to guard a minimal time between any two successive memory transactions of 10 cycles.

An edge connecting two locations can be traversed only if its guard evaluates to true. For example, the edge from location Timer to location Prod in Figure 6.1 (a) is traversed when the guard $clk == 10$ becomes true. Similarly, locations have invariants that have to be true while the location is marked active. Otherwise, the TA cannot reside in this location. As shown in Figure 6.1 (a), the invariant $clk \leq 10$ of location Timer guarantees that the clock variable clk does not exceed 10 when location Timer is marked active. All clocks in the TA are real-valued and increase their value at the same rate. Clocks and variables can only be inspected and reset upon the traversal of an edge. In our model clocks are only reset to 0 or 1.

In this chapter, we heavily exploit the concepts of *urgent* and *committed* locations offered by Uppaal. Urgent locations are marked with U and committed locations are marked with C. For example, location Prod in Figure 6.1 (a) is a committed location. Urgent and committed locations need to be left without time progress, i.e, clocks do not progress when urgent or committed locations are marked active. Contrary to urgent locations, any of the active committed location has to be left immediately on the next transition. This gives their outgoing edges a higher priority and thereby reduces the non-determinism in the model. As a result, using committed locations greatly reduces the state space for model checking.

In networks of TA, the component TA interact via shared variables and synchronization labels. The communication through synchronization labels is realized as a synchronized edge traversal of a sending edge (label with !) and receiving edge (label with ?). This atomic step includes the manipulation of associated variables and reset of associated clocks. Updates on sending edges are performed before the receiving edges. Since pairs of sending and receiving edges that synchronize are selected non-deterministically, all synchronization pairs emanating from active locations need to be generated when analyzing a TA. Besides binary synchronization, Uppaal also features 1:n synchronization for modeling broadcasts. Please refer to the Uppaal manual [13] for more details.

The originally infinite transition system generated from a network of TA can be reduced to a finite quotient system. Instead of tracking individual values of clocks, the domain of each clock is partitioned into finitely many intervals, denoted as clock re-

gions or their conjunction denoted as clock zones. Therefore, timed reachability queries formulated in a temporary logic, e.g., Timed CTL can be verified with a TA in a finite number of steps, as only finitely many combinations of clock regions need to be traversed. However, this number can be huge in practice.

6.2 MODULAR TA MODEL OF RUN-DMC

This section presents the TA model of the real-time memory controller with dynamic command scheduling, i.e., Run-DMC introduced in Chapter 3. A high-level overview of the model is shown, followed by introducing an intuitive model in Section 6.2.2 and an optimized model in Section 6.2.3, respectively. The former is easy to understand. However, it consists of too many locations, clocks, variables, and synchronizations, resulting in a large state space to be explored. Therefore, the latter simplifies the intuitive model using optimizations observed from the SDRAM timing constraints. These two TA models are available as open-source software on-line [73].

6.2.1 *An Overview of the TA Model*

A memory controller arbitrates between requestors. The selected transaction from a requestor is executed by dynamically scheduling its commands to consecutive banks of the SDRAM. To capture the behavior of the memory controller, we model the components shown in Figure 3.1, including the source of memory traffic, the TDM arbiter in the front-end, and the back-end including the memory mapping, command scheduler, and timing-constraint counters.

Figure 6.2 presents the components in our TA model of the RT memory controller together with their communication dependencies. Each of the components in Figure 6.2 is implemented by its own template TA. Communication between them is realized by synchronization labels. Our TA model: 1) Accurately describes the functionalities of the memory controller, *without any simplifying over-approximations*, cf. Section 6.4. This is a key to derive tight WCRT and WCBW bounds. 2) *Scalably* models transactions with different sizes and starting banks, in the sense that the size of the model is independent of the number of sizes and starting banks. 3) Is *modular*, i.e., each memory-controller component is modeled by a corresponding TA. Since memory controllers have common components, e.g., the timing constraint counters and command bus, the corresponding TA can be reused when modeling other memory controllers. 4) Is *easily adapted* to different memory generations (e.g., DDR3 and LPDDR3) by replacing the timing constraint values for the specific memory device [39].

The Source in Figure 6.2 generates read and write transactions for the requestors sharing a bus with a TDM arbiter (i.e., TDM Bus) that decides which memory requestor is served. TDM Bus also specifies the transaction size corresponding to the requestor and sends it to the back-end via the TDMArb synchronization label. As previously stated

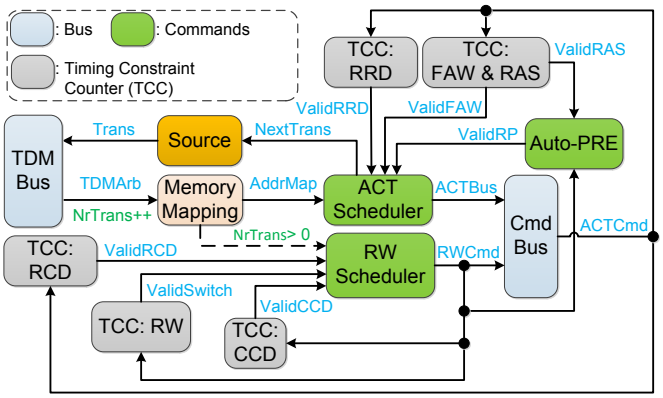


Figure 6.2: Abstracted overview of TA model for the dynamically-scheduled memory controller Run-DMC.

in Section 3.3.1, to capture the pipelining between successive transactions, the back-end accepts the next transaction when all *ACT* commands of the current transaction are scheduled [69, 72]. This is accurately captured by the Source that generates a new transaction when it is notified by the ACT Scheduler in the back-end via the NextTrans synchronization label. Note that this Source component makes the memory controller busy, i.e., there is always a transaction ready when the back-end can accept a new transaction. This ensures that each requestor has pending transactions to be executed within its allocated slots [72]. This results in maximum interference between requestors, leading to the worst-case scenario. Remember that the work-conserving TDM arbitration previously stated in Section 3.2.2 skips idle slots rather than reallocates them. As a result, a transaction experiences the worst-case response time when all other requestors have pending transactions.

The Memory Mapping in Figure 6.2 specifies the *BI*, *BC* and the starting bank address (*BS*) for a transaction sent by the TDM Bus through the TDMArb synchronization label. These parameters are required by the ACT Scheduler and RW Scheduler, which accurately capture the dynamic command scheduling algorithm in Section 3.3.2 for *ACT* and *RD*/*WR* commands, respectively. In particular, the ACT Scheduler issues an *ACT* command for the *BI* consecutive banks on the command bus (Cmd Bus), subject to the timing constraints of the memory. Timing counters (TCC) are used to track these constraints. Since the scheduling of an *ACT* command has to satisfy the *tRRD*, *tFAW*, and *tRP* constraints shown in Figure 2.8, the ACT Scheduler is notified via the ValidRRD, ValidFAW, and ValidRP synchronization labels when the timing constraints are satisfied, allowing for a new *ACT* command to be scheduled. Then, the ACT Scheduler synchronizes with the Cmd Bus using the ACTBus label. In the same way, *RD* and *WR* commands are scheduled by RW Scheduler according to the relevant timing constraints, e.g., *tCCD*, *tRCD*, and the read and write switching constraint captured by RW Counter. It also syn-

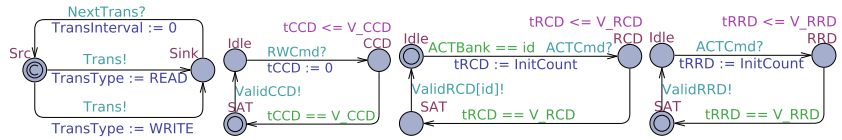
chronizes with the Cmd Bus using the RWCmd label when all these timing constraints are satisfied. Note that the RW Scheduler starts when NrTrans > 0, indicating there is at least one transaction in the back-end. However, the *RD*/*WR* command is issued by the RW Scheduler subject to the relevant timing constraints, such as the *tRCD* from the *ACT* command to the same bank. The ACT Scheduler is triggered by Memory Mapping through synchronization via the AddrMap label. Since the RW Scheduler and the ACT Scheduler work in parallel, the TA model captures the pipelining between scheduling *ACT* commands for the next transaction and scheduling *RD* or *WR* commands for the current transaction.

The Cmd Bus accurately models collisions between *ACT* and *RD* or *WR* commands by prioritizing the latter. As a result, the *ACT* command is delayed by 1 cycle when a collision occurs. After scheduling an *ACT* command, the relevant timing constraint counters are reset through broadcast synchronization using the ACTCmd label. Similarly, the timing constraint counters related to *RD* and *WR* commands are reset via broadcast synchronization labeled as RWCmd. Finally, the Auto-PRE describes the behavior of auto-precharging, which is triggered by RWCmd. These timing constraints are explicitly shown in Table 2.1 except *tRTW* and *tWTP*, which have been defined by Eq. (3.1). When the precharging of a bank is finished, the ACT Scheduler is notified by synchronizing with the ValidRP label. Note that all the synchronizations of our TA model are *urgent*, since commands are scheduled as soon as timing constraints are met.

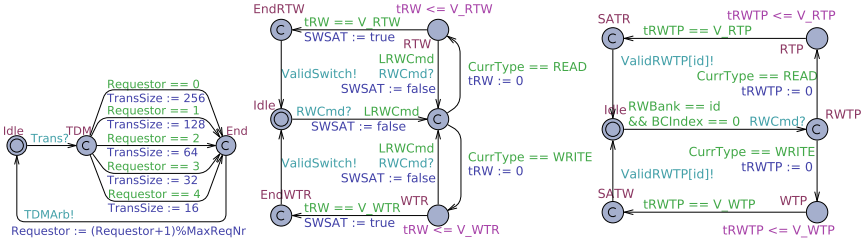
Our TA model does not include the scheduling of refresh, which is required periodically with a relatively large time interval *tREFI*. The reason is that the WCRT bound of each transaction is too pessimistic if the refresh period is included. Alternatively, the refresh penalty can be taken into the analysis of the application rather than individual transactions [93]. Moreover, the elimination of refresh also simplifies the TA model and reduces the state space for model checking. However, it is not difficult to model the refresh mechanism. Our model only needs to use an extra TA template to trigger the refresh every *tREFI* cycles, while it also interacts with the ACT/RW scheduler, such that the *ACT*, *RD*, and *WR* commands can be scheduled subject to all timing constraints.

6.2.2 *Intuitive TA Model of Command Scheduling*

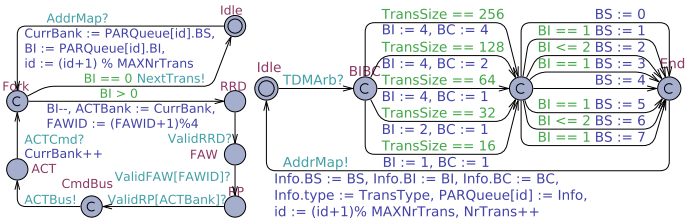
This section introduces how to intuitively model the behavior of Run-DMC, where each of its components and the intra-/inter-bank timing constraints is described by an automaton. The intuitive model is shown in Figure 6.3 and its system declaration in Uppaal is given in Appendix B.1. Note that the system declaration specifies the instances of these TA templates, resulting in a TA network capturing the behavior of a specific memory controller. The system declaration for the intuitive TA model is given in Appendix B.1.



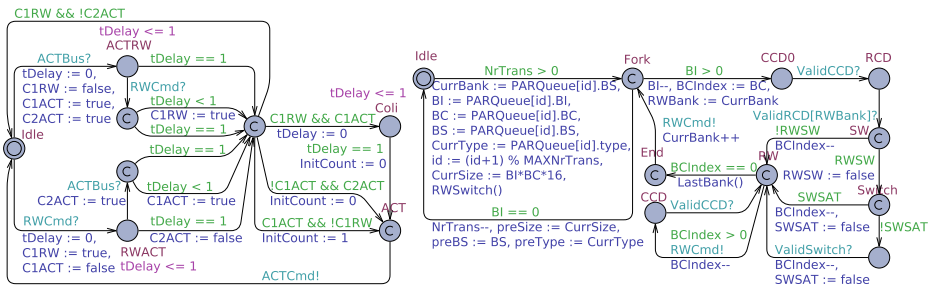
(a) Source of traffic (b) Counter of *tCCD* (c) Counter of *tRCD* (d) Counter of *tRRD*



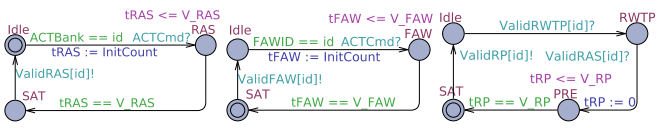
(e) TDM Bus (f) RW Switch (g) Counter of *tRWTP*



(h) ACT Scheduler (i) Memory Mapping



(j) Cmd Bus (k) RW Scheduler



(l) Counter of *tRAS* (m) Counter of *tFAW* (n) Precharging

Figure 6.3: The TA templates for intuitively modeling the behavior of dynamic command scheduling within the Uppaal toolbox.

6.2.2.1 *Automata Templates of the Requestors and Front-End*

Requestors are modeled by the Source TA, shown in Figure 6.3(a). It non-deterministically generates an infinite sequence of read and write transactions. Note that this Source TA always makes the memory controller busy, leading to maximum interference between requestors and hence resulting in the worst-case results. It generates a transaction when the back-end of the memory controller is ready to accept the next transaction. This is notified using the label NextTrans. The type of each transaction is defined by the global variable TransType, which is either READ or WRITE. The Source TA synchronizes with the TDM Bus TA using the Trans label. Note that the TDM bus uses Algorithm 1 for variable transaction sizes. The TDM Bus TA, shown in Figure 6.3(e), models a TDM bus instance with, as an example, five requestors, each of which has one TDM slot. Note that this TDM behaves the same as the round-robin (RR) scheme, since they have the same worst-case behavior when each requestor has only one slot. The guard on each edge specifies the slot and the requestor, and the corresponding transaction size TransSize is specified upon the edge traversal. The TDM Bus TA then synchronizes with the Memory Mapping TA (see Figure 6.3(i)) using the TDMArb label. Finally, only one instance of the Source TA and TDM Bus TA is needed, which is declared in the system declaration.

6.2.2.2 *Automata Template of the Memory Mapping*

The Memory Mapping TA in Figure 6.3(i) determines the memory mapping configuration in terms of *BI* and *BC* based on the TransSize, while the starting bank (*BS*) is given by non-deterministically selecting one outgoing edge. For each transaction size, *BI* and *BC* are configured to achieve the lowest execution time [72]. Recall that *BS* is aligned with *BI* to simplify the physical address decoding [39]. In addition, a transaction may use multiple *BS*, as shown in Figure 6.3(i). The reason is that we do not specify particular memory address allocations to requestors, which are out of the scope of this thesis. When the memory mapping is finished, the number (i.e., NrTrans) of transactions in the back-end is increased and command scheduling is triggered. PARQueue contains the information (i.e., TransType, BI, BC, BS) of the active transactions, cf. Figure 3.1 and Algorithm 2. The Memory Mapping TA is instantiated in the system declaration.

6.2.2.3 *Automata Template of the ACT Scheduler*

The ACT Scheduler TA, shown in Figure 6.3(h), starts scheduling *ACT* commands of a transaction after synchronizing with the Memory Mapping TA through the AddrMap label. An *ACT* command is scheduled to each of the *BI* consecutive banks, starting at bank *BS*. This is achieved by repeatedly scheduling each *ACT* command subject to the timing constraints *tRRD*, *tFAW*, and *tRP*, as given in Table 2.1. These constraints are tracked by TA, which are illustrated in Figure 6.3(d), Figure 6.3(m) and Figure 6.3(g) for the *tRRD*, *tFAW*, and *tRP* constraints, respectively. The ACT Scheduler TA waits for each

timing constraint to be met. It advances through locations RRD, FAW, and RP when the relevant TA indicates that timing constraint is met through the ValidRRD, ValidFAW, and ValidRP labels, respectively. The ACT Scheduler TA and Cmd Bus TA synchronize using the ACTBus and ACTCmd labels. The former notifies the Cmd Bus TA to schedule the *ACT* command, while the latter triggers the ACT Scheduler TA to schedule the next *ACT* command once the previous one has been scheduled by the Cmd Bus TA. Finally, we only need a single instance of the ACT Scheduler TA to schedule *ACT* commands to each bank.

6.2.2.4 *Automata Template of the RW Scheduler*

The RW Scheduler TA, shown in Figure 6.3(k), always schedules *RD* or *WR* commands when there are transactions in the back-end, i.e., NrTrans > 0. It works similarly to the ACT Scheduler, where *BC RD* or *WR* commands are repeatedly scheduled in sequence to each of the *BI* consecutive banks subject to the timing constraints. The relevant timing constraints are *tCCD* and *tRCD*, which are tracked by the TA shown in Figure 6.3(b) and Figure 6.3(c), respectively. Note that the first *RD*/*WR* command of a transaction additionally has to satisfy the switching timing constraint when the previous transaction is write/read. This is identified by the boolean variable RWSW that is determined by RWSwitch() when RW Scheduler starts a new transaction. The switching timing constraints *tRTW* and *tWTR* are captured by TA shown in Figure 6.3(f).

When a *RD*/*WR* command is scheduled, the RW Scheduler TA synchronizes with the Cmd Bus TA using the RWCmd label, such that collisions are resolved in the command bus. Recall that a *RD* or *WR* command has higher priority than an *ACT* command that may have its timing constraints satisfied at the same time. As explained below, the Cmd Bus TA then ensures that the colliding *ACT* command is scheduled one cycle later, thus solving the command collision. Finally, the broadcast synchronization using the RWCmd label tells the relevant timing constraint TA to reset their counters. Note that the RW Scheduler TA is instantiated with a single instance when being declared in the system declaration.

6.2.2.5 *Automata Templates of the Command Bus*

The command bus is modeled by the TA shown in Figure 6.3(j), which detects and solves command bus collisions. The Cmd Bus TA synchronizes with the ACT Scheduler and RW Scheduler TA using the ACTBus and RWCmd labels, respectively. Although these two TA run in parallel, the synchronizations labeled RWCmd and ACTBus are received sequentially. The ACTRW and RWACT locations in Figure 6.3(j) ensure that these actions can be received in either order. This is achieved by the transitions from the Idle location to either ACTRW or RWACT depending on the signals from the synchronization channels labeled as ACTBus or RWCmd. After synchronizing through either a RWCmd label or an ACTBus label, the Cmd Bus TA waits until the end of the cycle to see if

the other synchronization arrives within this time. If so, a command collision has to be resolved by postponing the *ACT* to the next cycle.

A collision is identified by the Boolean variables C1RW, C1ACT, and C2ACT that indicate the presence of a *RD*/*WR* or an *ACT* command within the same cycle (i.e., C1) or in the second cycle (C2). If there is a collision, i.e., both C1RW and C1ACT are true, the *ACT* has to be delayed by one cycle. When there is no collision, the *ACT* command can be scheduled immediately. This includes two cases where 1) the *ACT* command arrives in the second cycle (i.e., both C1RW and C2ACT are true) or 2) it arrives in the first cycle when there is no *RD*/*WR* command in the same cycle, i.e., C1ACT is true while C1RW is false. For the former, the *ACT* command has to be scheduled immediately by broadcasting synchronization using the ACTCmd label, and the corresponding timing constraint counters are reset, e.g., *tRCD*, *tRAS*, *tRRD*, *tFAW*. For the latter, it has already waited for one cycle when broadcasting the ACTCmd label. So, the *ACT* related counters are reset to start counting from 1 instead of 0 as normal. This is achieved by setting the global variable InitCount to be 1, which is the initial value for all the relevant timing constraint counters. Since there is only one command bus, a single instance of Cmd Bus TA is declared in the system declaration.

6.2.2.6 *Automata Templates of Timing Constraint Counters and Precharging*

Timing constraints are tracked by counters, where each of them counts from zero to the JEDEC-specified value [53]. For example, the timing constraint *tRAS* (see Table 2.1) is modeled by the TA shown in Figure 6.3(l). It uses a clock variable tRAS, which is initialized to InitCount (0 or 1, see the previous paragraph) after an *ACT* command was scheduled, as indicated by the ACTCmd label. This counter counts to the constant V_RAS of *tRAS* provided by JEDEC DDR3 standard [53]. When the timing constraint is satisfied, i.e., tRAS == V_RAS, it immediately synchronizes using the ValidRAS label. Other timing constraints are tracked in the same way, where *tCCD*, *tRCD*, *tRRD*, *tRTW*/*tWTR*, *tRWTP*, and *tFAW* are modeled by the TA shown in Figure 6.3(b), Figure 6.3(c), Figure 6.3(d), Figure 6.3(f), Figure 6.3(g) and Figure 6.3(m), respectively.

In particular, the RW Switch TA in Figure 6.3(f) tracks both the *tRTW* and *tWTR* timing constraint for switching between a read and a write transaction, where a single clock variable tRW is used. It is reset when a transaction finishes the command scheduling, i.e., the last *RD* or *WR* command is scheduled. As a result, the RW Switch TA is synchronized using label RWCmd on the condition that LRWCmd is true. The boolean variable LRWCmd becomes true when the last *RD* or *WR* command is scheduled by the RW Scheduler TA. Note that LRWCmd is included in LastBank() in Figure 6.3(k). The RW Switch TA chooses to wait in location RTW or WTR depending on the type of the current transaction (see Figure 6.3(f)). As a result, both *tRTW* and *tWTR* are tracked. Similarly, the timing constraints *tRTP* and *tWTP* are tracked by the *tRWTP* TA given by Figure 6.3(g). It uses a single clock variable tRWTP and is reset when the last *RD* or *WR* command is scheduled to a bank, i.e., BCIndex == 0. Note that BCIndex is updated by

the RW Scheduler TA. Finally, the auto-precharging scheme is captured by a TA shown in Figure 6.3(n). It is synchronized using the labels ValidRWTP and ValidRAS, since the precharging of a bank is enabled when both *tRWTP* and *tRAS* are met. Moreover, the precharging time period *tRP* is tracked using the clock variable tRP.

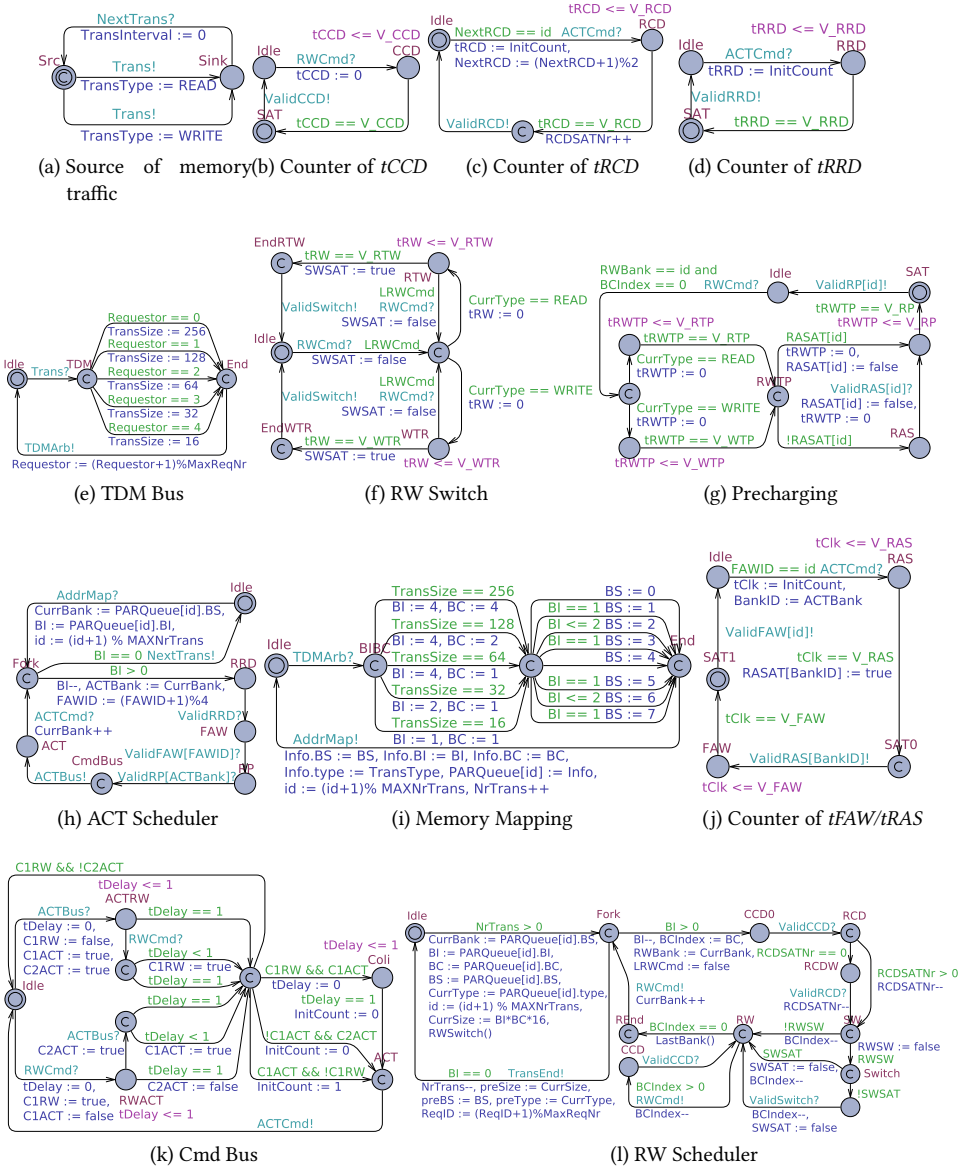
The SDRAM timing constraints are classified into inter- and intra-bank. Intuitively, an inter-bank constraint can be tracked by a single counter while an intra-bank constraint should be tracked by a counter per bank. For example, the intra-bank timing constraint *tRAS* is captured by 8 instances of the TA shown in Figure 6.3(l), since a DDR3 SDRAM device typically consists of 8 banks. This also applies to *tRCD*, *tRWTP*, and *tRP*. For the inter-bank timing constraint *tCCD*, *tRRD* and *tRTW*/*tWTR*, only a single instance of the corresponding TA is needed. The *tFAW* is captured by four instances of the *FAW* TA, shown in Figure 6.3(m), such that mostly 4 *ACT* commands can be scheduled within *tFAW*. All these instances are presented in the system declaration in Appendix B.1. It is worth noting that different memory devices can be supported by using their timing constraint values in these counters [39].

6.2.3 *Optimized TA Model of Command Scheduling*

The intuitive TA model of Run-DMC presented in the previous Section 6.2.2 is easy to understand, since it models each functional component and the timing constraint counter by an automaton. However, it has a large state space, resulting in long time and high memory usage when verifying properties of the TA model with model checking. To reduce the state space, the number of states, clocks, variables, and edges must be reduced. Therefore, we introduce two optimizations to derive an optimized TA model. Whenever possible, 1) we model multiple timing constraints with a single TA instead of modeling them with separate TA. 2) We reuse counters for different timing constraints. The first optimization enables a clock variable being shared when tracking multiple timing constraints with a single TA. In addition, the number of locations and synchronizations of the single TA can be reduced compared to the intuitive model that uses multiple TA. The second optimization reduces the number of TA instances, leading to fewer locations, variables, edges, and synchronizations.

The above optimizations are used to obtain an optimized TA model, which uses several different TA templates (see Figure 6.4) based on the intuitive model shown in Figure 6.3. Most of the TA templates in Figure 6.4 are the same as the Figure 6.3. The differences are given below.

i) The TA template shown in Figure 6.4(j) tracks both the *tRAS* and *tFAW* constraints. Note that an instance of the TA template models a counter tracking the relevant timing constraint(s). The JEDEC [53] standard guarantees that $tFAW \geq tRAS$, and thus the tClk in Figure 6.4(j) counts first until *tRAS*, and then until *tFAW*. The inter-bank four activate window constraint *tFAW* can be tracked by using four counters (i.e., instances) for four *ACT* commands. The intra-bank *tRAS* constraint can be implemented with 8 counters,



(a) Source of traffic (b) Counter of *tCCD* (c) Counter of *tRCD* (d) Counter of *tRRD*

(e) TDM Bus (f) RW Switch (g) Precharging

(h) ACT Scheduler (i) Memory Mapping (j) Counter of *tFAW/tRAS*

(k) Cmd Bus (l) RW Scheduler

Figure 6.4: The optimized TA templates for modeling the behavior of dynamic command scheduling within the Uppaal toolbox.

Table 6.1: Comparison between the intuitive and optimized TA model.

Models	*TA instances*	*clocks*	*variables*	*locations*	*synchronizations*	*edges*
Intuitive TA	45	40	41	186	46	226
Optimized TA	23	18	55	137	39	186

one for each bank of DDR3 SDRAM. However, using the aforementioned inequality, four counters suffice to track *tRAS* for all 8 banks. The fifth *ACT* has to wait until *tFAW* before it can be scheduled. By then, it is guaranteed that at least one counter has passed *tRAS* because $tFAW \geq tRAS$. Therefore, the counter for the fifth *ACT* command can be eliminated. It is also the same case for the next three *ACT* commands. Four instances of the TA shown in Figure 6.4(j) are declared in the system declaration of the optimized TA model, which is given in Appendix B.2.
ii) The TA shown in Figure 6.4(g) captures both the constraints from a *RD*/*WR* command to an auto-precharge, and the precharge period *tRP*. It still requires 8 instances of this TA capturing the timing constraints and precharging of each bank. These instances are declared in Appendix B.2.
iii) Finally, we can observe from JEDEC-specified DDR3 timing constraints that $tRCD \leq 2 \times tRRD$. *tRRD* is the minimum time between two successive *ACT* commands. Within $2 \times tRRD$ cycles, at most three *ACT* commands can be scheduled. When the third *ACT* command is scheduled, the counter triggered by the first *ACT* command is guaranteed to be larger than *tRCD* and hence can be reused. Therefore, only two counters are needed to track the intra-bank timing constraint *tRCD*, shown in Figure 6.4(c), resulting in two instances of the *tRCD* TA. Moreover, the RW Scheduler TA is updated to deal with the changes to the *tRCD* TA. The new RW Scheduler TA is given by Figure 6.4(l). These optimizations rely on particular relations between timing constraints, but they hold for all DDR3 and LPDDR3 devices.

The optimized TA model of Run-DMC, shown in Figure 6.4, has a smaller state space than the intuitive TA model in Figure 6.4. This is helpful when verifying properties via model checking. Table 6.1 summarizes the number of TA instances, clocks, variables, locations, edges, and synchronizations used by these two models. It shows that the optimized TA model uses less of these elements, which have big impact on the memory usage and time consumption for model checking. The only exception is the number of variables, where the optimized TA model uses more. The reason is that the optimized TA model uses variables to indicate whether a timing constraint (e.g., *tRAS*) is satisfied. In contrast, the intuitive TA model uses synchronization labels to indicate that timing constraints are satisfied. However, the synchronizations cause dependencies and result in longer time of verifying a property. As a result, it is easier to derive the worst-case

bounds by verifying properties of the optimized TA model. The verification with model checking will be discussed in the next section.

6.2.4 *Reflection*

Although the TA model of the RT memory controller is involved, its structure mirrors that of the memory controller hardware architecture and algorithm (see Chapter 3). We accurately model all timing constraints, without having to resort to conservative, but pessimistic assumptions, such as a worst-case initial state used by the formal analysis in Chapter 4. Moreover, the collision on the command bus is resolved when happened, rather than conservatively assumed to always happen for each *ACT* command. So, the TA model is able to provide better worst-case bounds than existing analyses, which employ these pessimistic assumptions. Finally, to speed up verification of the model, timing counters were eliminated, although at the cost of model simplicity.

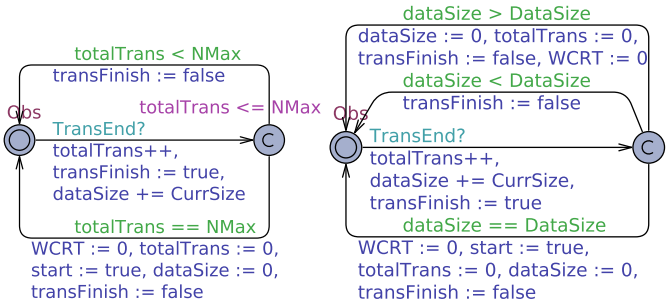
6.3 VERIFICATION WITH MODEL CHECKING

This section proceeds by showing how to derive bounds on the worst-case response time (WCRT) and worst-case bandwidth (WCBW) by verifying properties of our TA model with model checking.

6.3.1 *Verification of Worst-Case Response Time*

According to Definition 9, a transaction experiences the worst-case response time (WCRT) when the maximum number of interfering transactions must be executed before the execution of this new transaction. Moreover, it assumes that the execution of all these transactions needs the maximum time to schedule their commands in a pipelining manner. Essentially, the WCRT bound is the summation of the maximum total execution times of these transactions caused by scheduling commands and the offset of transferring a burst of data for the last *RD*/*WR* command. Note that Run-DMC uses a TDM arbiter in the front-end to serve requestors with pending transactions, as shown in Figure 3.1. Recall that each requestor is assumed to have at most one outstanding transaction to avoid arbitrarily high self-interference in the WCRT [11]. Therefore, *to derive the bound on the WCRT of a requestor, it is equivalent to verify the longest time of executing these maximum number of interfering transactions and the transaction from the requestor itself.* This is achieved by using model checking of the proposed TA model.

An observer TA is designed to track the response time of each transaction from a particular requestor, as shown in Figure 6.5(a). Note that *NMax* denotes the maximum number of interfering transactions and the transaction from the requestor itself. It can be computed based on the TDM slot configuration. This observer counts the number of executed transactions (totalTrans), and *the end of a transaction is signaled by the*



(a) The observer of WCRT (b) The observer of WCBW

Figure 6.5: The TA to verify the WCRT and WCBW bounds.

TransEnd synchronization label when its last RD or WR command is scheduled. The end of a transaction indicates the finishing time given by Definition 5 in the back-end, and it is further used to compute the execution time defined by Definition 7. Meanwhile, the clock variable WCRT in Figure 6.5(a) tracks the total time of executing transactions. By specifying the maximum number (i.e., *NMax*) of transactions to be observed, we use standard reachability queries to verify the clock WCRT with the verifier module of the Uppaal tool suite. By manually executing a binary search on the bound of the clock WCRT, it directly translates into the maximum time of executing these *NMax* transactions. The observer records this maximum time when the last *RD* or *WR* command of a transaction is scheduled. For read transactions, we need to add the constant data offset, which is $tRL + BL/dataRate$ as given by the JEDEC timing constraints [53]. It is worth noting that the Uppaal model checker explores the full state space for all the possible *NMax* successive transactions and derives the WCRT bound. As a consequence, our observer only needs to cycle through every *NMax* successive transactions rather than uses a sliding window for any *NMax* transactions.

To derive the bound on the WCRT, we have to verify the maximum value of the clock variable WCRT in Figure 6.5(a) via binary searching. Uppaal allows users to specify a query for verifying a property. For example, Query 1 and Query 2 are used to verify the maximum WCRT in the Observer for a read or a write transaction, respectively.

Query 1 (WCRT property for write transaction). *A[] (Observer.start == true and CurrType == WRITE) imply Observer.WCRT <= Estimate_Bound*

Query 2 (WCRT property for read transaction). *A[] (Observer.start == true and CurrType == READ) imply Observer.WCRT <= Estimate_Bound*

A[] indicates that it searches all paths in the state space to verify whether the latter expression (i.e., the property) is true. The expression states that the WCRT collected by the Observer cannot be larger than a manually specified value, i.e., *Estimate_Bound*,

when the current transaction is a write/read and the Observer has started the observation. By increasing/decreasing *Estimate_Bound*, the maximum WCRT is obtained when the verification of the query becomes true from false or vice versa. Therefore, a proper value of *Estimate_Bound* makes this process faster. Practically, we can obtain a proper *Estimate_Bound* by simulating the TA model for NMax transactions, since the simulation provides an actual execution time of these transactions. However, this actual execution time may be far from the worst-case bound, leading to a long manual procedure in the binary search. A more efficient way of predicting *Estimate_Bound* is based on the worst-case execution time (WCET) of an individual transaction. Note that the WCET of a transaction equals the maximum WCRT in the Observer when NMax == 1. Then *Estimate_Bound* for any NMax is set to be the summation of the WCET of these NMax transactions. This computed *Estimate_Bound* actually overestimates the bound, because the pipelining between the NMax transactions is not exploited. However, it is an efficient experimental approach to derive a relatively proper *Estimate_Bound* to start the query verification.

6.3.2 *Verification of Worst-Case Bandwidth*

With Definition 11, the worst-case bandwidth (WCBW) is the minimum bandwidth for all infinitely-long transaction traces. However, practically, we can only compute the time to transfer a finite number of bytes. We therefore use an observer TA (see Figure 6.5(b)) to verify the maximum time for transferring *DataSize* data for any possible traces that can generate *DataSize* bytes. In a word, for $\forall\overline{T}$, $\sum_{\forall T\in\overline{T}} S(T) = DataSize$. The WCBW bound can be computed based on this maximum time, as given by Eq. (6.1). This bound is conservative for the long-term WCBW given by Definition 11. The reason is that it is observed based on transferring a fixed amount of data, and the pipelining between a limited number of transactions is exploited. In contrast, the long-term WCBW exploits the pipelining between infinite transactions and it cannot be smaller than the derived bound in Eq. (6.1). Intuitively, the minimum rate observed in a long time period of repeatedly transferring a fixed amount of data cannot be larger than the average rate of transferring the total amount of data in the whole time period, where the former corresponds to repeating a limited number of transactions while the latter is achieved by executing an infinite number of any transactions. Lemma 5 captures this intuition and states that any $\hat{bw}(DataSize)$ is a conservative lower bound for the *WCBW* (i.e., $\hat{bw}$) defined by Definition 11. The former is the minimum rate of transferring *DataSize* data,

while the latter is the long-term (minimum) average rate of transferring data from/into the SDRAM for infinitely-long traces. The proof is given in Appendix A.7.

$$\hat{bw}(DataSize) = \frac{DataSize}{\text{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)} \times f_{mem} \times e^{ref}, \quad where\ DataSize = \sum_{\forall T \in \overline{T}} S(T) \tag{6.1}$$

Lemma 5.

$$\forall DataSize > 0, \hat{bw}(DataSize) \leq \hat{bw}$$

Figure 6.5(b) shows the observer TA that tracks the total time for transferring *DataSize* data. Time is tracked by reusing the clock variable WCRT, while dataSize accumulates the transferred data when a transaction is finished, as notified by the TransEnd synchronization label. We manually execute a binary search on the bound of the clock WCRT with Uppaal. This bound is the maximum time of transferring *DataSize* data. Multiplying the result by $f_{mem} \times e^{ref}$ returns a conservative lower bound for the WCBW, as described above by Eq. (6.1). Moreover, Lemma 5 implies that the WCBW bound is always conservative for any finite *DataSize*.

To verify the maximum time of transferring *DataSize* data with the Observer in Figure 6.5(b), we use Query 3 with Uppaal. By manually increasing/decreasing the value of *Estimate_Bound*, the bound on the clock variable WCRT for the specified *DataSize* is obtained when the expression of Query 3 becomes true from false or vice versa. This bound is a function of *DataSize* and is denoted by ET_Bound(*DataSize*). The WCBW bound is then given by Eq. (6.2).

$$\hat{bw}(DataSize) = \frac{DataSize}{ET\_Bound(DataSize)} \tag{6.2}$$

Query 3 (WCBW property). *A[] (Observer.start == true and Observer.dataSize == Observer.DataSize) imply Observer.WCRT <= Estimate_Bound*

By specifying larger *DataSize* in the Observer (see Figure 6.5(b)), better (i.e., larger) lower bound on WCBW can be obtained, because of exploiting more pipelining between successive transactions. However, we can only practically verify properties with a finite number of different *DataSize*, where Lemma 5 guarantees that the derived bound is conservative for the actual long-term WCBW. On one hand, larger *DataSize* corresponding to longer transaction traces may result in longer time and higher memory usage when verifying the property of the TA model. On the other hand, it is also not necessary to use a very large *DataSize* to derive a tighter WCBW bound. The reason is that an increasing *DataSize* cannot improve the WCBW bound dramatically. Therefore, we have to

manually increase *DataSize* until *WCBW*(*DataSize*) cannot improve significantly. This poses a trade-off between the tightness of the bound and the time/memory consumed to successfully verify the property.

It is critical to specify a proper *Estimate_Bound* for Query 3, such that we can quickly derive ET_Bound(*DataSize*). Since we start with the smallest *DataSize* (i.e., $DataSize_0$) corresponding to the smallest size of a transaction, ET_Bound($DataSize_0$) equals the WCET of this transaction. The WCET can be obtained in the way of deriving the WCRT bound, as previously discussed in Section 6.3.1. For an arbitrary large $DataSize_i$ ($\forall i > 0$), it is composed of smaller *DataSizes*, denoted by $DataSize_m$ and $DataSize_n$, such that $DataSize_i = DataSize_m + DataSize_n$. Therefore, a good upper *Estimate_Bound* for $DataSize_i$ is given by Eq.(6.3), where $ET\_Bound(DataSize_m)$ and $ET\_Bound(DataSize_n)$ are the maximum times of transferring $DataSize_m$ and $DataSize_n$ bytes data, respectively. As a result, the summation cannot be smaller than $ET\_Bound(DataSize_i)$, leading to conservative $Estimate\_Bound(DataSize_i)$. The reason is that a larger $DataSize_i$ ensures that more pipelining between transactions can be exploited and hence $ET\_Bound(DataSize_i) \leq Estimate\_Bound(DataSize_i)$, resulting in a larger bound $\hat{bw}(DataSize_i)$.

$$Estimate\_Bound(DataSize_i) = \min_{\forall m,n, DataSize_i = DataSize_m + DataSize_n} (ET\_Bound(DataSize_m) + ET\_Bound(DataSize_n)) \quad (6.3)$$

6.4 RELATED WORK

Existing analyses of semi-static RT memory controllers [3, 39, 46, 88] provide WCRT and/or WCBW by dynamically using a set of pre-computed static command schedules for transactions. As discussed in Section 4.1 and Section 5.1 and experimentally shown in previous chapters, the drawbacks of using static command schedules are that 1) they cannot exploit dynamic information about the SDRAM state caused by timing constraints and the exact SDRAM banks required by individual transactions. 2) These command schedules transfer a fixed amount of data. When the transaction size varies, unwanted data is discarded, resulting in low data efficiency.

To overcome the inefficiency of semi-static command schedules, dynamic command scheduling can be used, where commands are scheduled by some dynamic algorithms when the SDRAM timing constraints are satisfied. However, analysis of dynamic scheduling constrained by timing dependencies is difficult. The analytical approach [72] presented in Chapter 4 abstracts the state of previous transactions to a worst-case initial state, by pessimistically assuming that their commands were scheduled as late as possible (ALAP). This results in conservative command scheduling times for the current transaction. In addition, it assumes that every *ACT* command collides on the command bus. The dataflow model [71] presented in Chapter 5 provides the WCBW of dynamic command scheduling. It also assumes that *ACT* commands always have command-bus

collisions, but does not require the ALAP assumption. Conversely, the scheduled approach [72], given in Chapter 4, accurately models command-bus conflicts, but assumes ALAP schedules. The analyses in [55, 56, 63] assume that the *RW* to *RD* switching timing constraint and the four-activate window constraint are always incurred, even though these constraints do not always dominate in the command schedule. These analysis approaches may not be easy to manually adapt to memory controllers with different mechanisms or memories. Similarly to the dataflow model in Chapter 5, we shift the manual effort to derive performance bounds from analysis to modeling. In other words, rather than providing a specialized WCRT/WCBW analysis of a memory controller, a specialized model of a memory controller is defined, which is analyzed automatically using state-of-the-art tools. Chapter 5 uses dataflow, while this chapter uses TA, which is more expressive and results in better bounds.

TA have been extensively used to address the complexity of sharing resources. Yi et al. [81] were the first to use TA to represent a system resource (CPU or communication element) as a scheduler model together with a notion of discrete events that trigger the execution of RT tasks on this scheduler. In [75], the basic idea has been extended to analyze multi-core architectures and different memory access policies. A similar approach is also presented by Gustavsson et al. [42]. Lampka et al. [64] present an approach that abstracts from individual core-local workloads by modeling access requests to a shared resource with an aggregated access request curve fed into a network of TA. These works intend to bound the worst-case execution time of applications rather than individual memory accesses/transactions. In contrast, we focus on the effect of sharing on the timing of individual memory transactions in order to find a tight bound. Our WCRT and WCBW bounds can be used by the cited works for more accurate modeling and analysis.

6.5 EXPERIMENTAL RESULTS

This section experimentally validates the proposed intuitive and optimized TA models of dynamic command scheduling, and then analyzes the WCRT and WCBW bounds for fixed and variable transaction sizes, respectively. First, the command scheduling results obtained with Uppaal are validated with the open-source *RTMemController* tool [70], which has been introduced in Section 4.7 and has been proven to be equivalent to a cycle-accurate SystemC simulator in Section 4.8.2 for capturing the timing behavior of the memory controller. Next, the results are compared to the analysis results of the same memory controller design using a) the analytical and scheduled approaches presented in Chapter 4, and b) the mode-controlled dataflow (MCDF) model introduced in Chapter 5. These techniques were discussed in Section 6.4. Note that the WCRT/WCBW bounds are obtained using the optimized TA model, since the verification of the intuitive model takes more time and consumes more memory on the host server.

6.5.1 *Experimental Setup*

The proposed TA model is simulated and verified with Uppaal v4.1.19 on a 64-bit CentOS 6.6 system with 24 Intel Xeon(R) CPUs running at 2.10 GHz and with 125 GB usable RAM. Experiments have been done with a JEDEC-compliant DDR3-1600G SDRAM memory with interface width of 16 bits and a capacity of 2 Gb [53]. The memory controller front-end uses a TDM arbiter with one slot per memory requestor, which is assumed to have one outstanding transaction to avoid self-interference. The transaction sizes used by the experiments are 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes, respectively. Similarly to the previous experimental setup in Section 3.5.1, Section 4.8.1, and Section 5.5.1, the memory map configuration (i.e., BI and BC) of each transaction size is chosen to achieve the lowest execution time and the highest memory bandwidth, where more banks are interleaved when possible to exploit bank parallelism. The configured (BI, BC) for these sizes are hence (1, 1), (2, 1), (4, 1), (4, 2), and (4, 4) [39], respectively. Note that transactions with 128 bytes and 256 bytes use (4, 2) and (4, 4) instead of (8, 1) and (8, 2) because of the tFAW constraint that leads to larger execution time with BI = 8. Finally, we always verify the property "*A[] not deadlock*" to be true for the TA model. This guarantees that there is no deadlock in the TA model, e.g., an automaton cannot resign in a committed location without an immediate transition.

6.5.2 *Validation of TA Model*

The first experiment shows that the proposed TA model can accurately capture the timing behavior of the memory controller with dynamic command scheduling for selected traces. We compare the scheduling time of each command in every trace obtained with Uppaal to that of the *RTMemController* tool [70], which is equivalent to a cycle-accurate SystemC simulator of the memory controller under consideration [69]. We simulate the TA model for 1200 commands corresponding to random read and write transactions with any possible physical addresses that are reflected by using all the possible starting banks. The transactions are generated by the Source and TDM Bus TA shown in Figure 6.4(a) and Figure 6.4(e). We execute *RTMemController* with the same transactions to obtain the scheduling time of each command. The same experiment is repeated for the 5 transaction sizes and for a random mix of them. From the experimental results, we observe that *the scheduling time is always identical for each command.* This suggests that our TA model correctly and accurately captures the timing behavior of the dynamic command scheduling. For the given traces, *the TA model is equivalent to the cycle-accurate implementation of the dynamic command scheduling.* In the same way, the intuitive model is also validated.

For all experiments in the following sections, Uppaal generates a witness that leads to the WCRT/WCBW bound, i.e., the diagnostic trace of transactions. Again, we have fed all diagnostic traces to the RTMemController tool, and it always shows exactly the

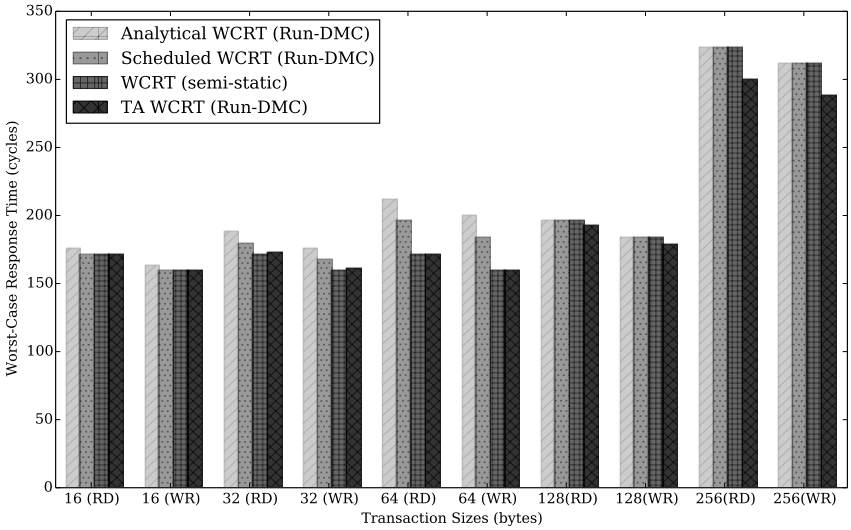


Figure 6.6: The WCRT for 4 requestors accessing DDR3-1600G with fixed transaction sizes.

same results as Uppaal. *This validates the correctness of the TA model, and gives strong reason to believe that the analysis results derived from our TA model are tight.*

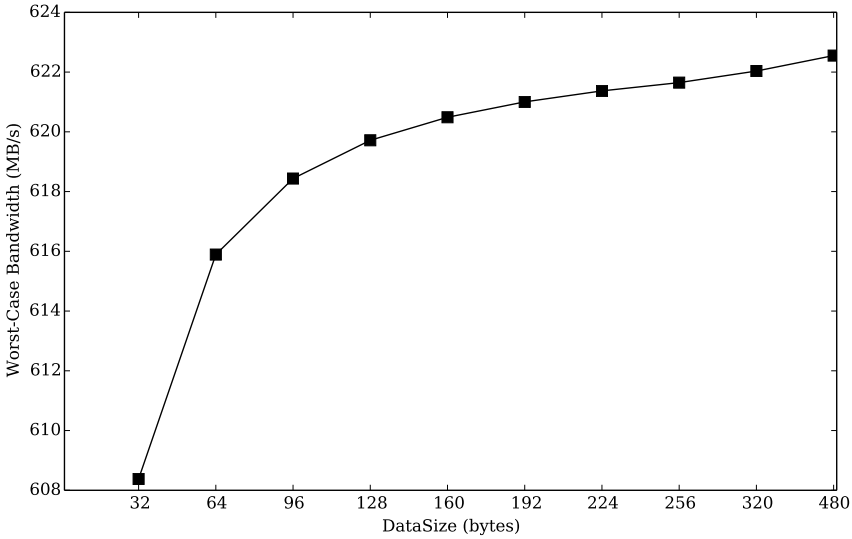
6.5.3 *Fixed Transaction Size*

This experiment uses Uppaal to test the optimized TA model and to obtain the WCRT and WCBW Bounds for fixed transaction sizes. Four memory requestors are employed, corresponding to, e.g., four cores that have the same cache-line size. This experiment tests an arbitrary read/write mix, for the five different transaction sizes. The model checker explores the full state space for each size, except for 16 bytes that uses BI=1. For the purpose of worst-case analysis, it is not necessary to explore all 8 banks. Transactions with 16 bytes only access a single bank. It hence only matters if transactions access the same bank or a different bank. As a result, we arbitrarily select 2 banks (e.g., Bank 0, Bank 1) to be tested by Uppaal. The trace is an arbitrary read/write mix, but we show the WCRT for read (RD) and write (WR) transactions separately in Figure 6.6. They are also compared to those given by the existing analytical and scheduled approaches in Chapter 4 for Run-DMC and the results of the semi-static approach [3]. Note that we do not compare with the MCDF model previously presented in Chapter 5 as it does not analyze the WCRT. Moreover, each result is validated by executing the diagnostic trace from Uppaal with RTMemController, where the same results are obtained from these two tools.

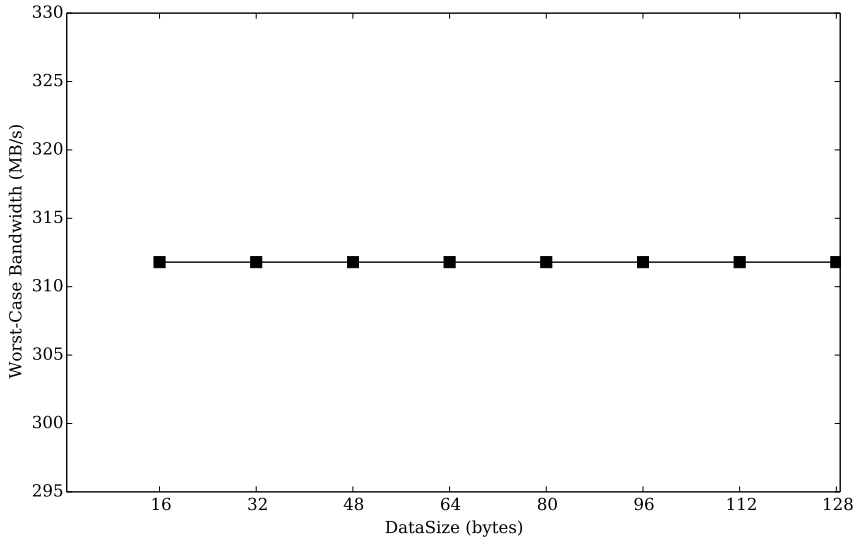
We observe that *the WCRT results from the TA model are better than or equal to those given by either the scheduled or analytical approaches.* This also holds when comparing to the semi-static approach except for 32-byte transactions, where the WCRT of the semi-static approach is only 1 cycle smaller than the results of the TA model. This is because Run-DMC schedules commands dynamically and may lead to larger WCRT than the semi-static approach that uses the pre-computed command schedules. However, this exception only occurs when a particular (and rare) sequence of transactions are executed by Run-DMC. The maximum improvement is 20% for write transactions with 64 bytes, while the average improvement over all these experiments is 7.7%. The improvement is achieved for the following reasons. 1) The TA model accurately models scheduling collisions on the command bus, just like the scheduled approach. Conversely, the analytical approach always conservatively assumes a collision for each *ACT* command. 2) The TA model accurately captures the worst-case initial values of the timing counters for an arbitrary transaction according to the exact JEDEC timing constraint values. In contrast, both the analytical and scheduled approaches conservatively assume as-late-as-possible (ALAP) scheduling of the previous commands to provide the worst-case initial values of timing counters, which is pessimistic. 3) Run-DMC performs closely to the semi-static approach for these small transaction sizes. When executing large transaction sizes, e.g., 128 bytes or 256 bytes, Run-DMC can dynamically exploit the pipelining between successive transactions. In contrast, the static command schedules used by the semi-static approach cannot efficiently pipeline commands across successive transactions.

An experiment is carried out to test the intuitive TA model, which provides the same WCRT bounds as the optimized TA model. However, verifying a property of the intuitive TA model takes longer time and consumes more RAM of the host server. For example, for 128-byte transactions from four requestors, the verification speedup reaches 2.4x when deriving the WCRT bound. Moreover, the RAM usage is decreased by 31%. These results demonstrate that it is easier to derive a bound with the optimized TA model.

The bound on the worst-case bandwidth (WCBW) depends on the *DataSize* used by the Observer shown in Figure 6.5(b). As discussed in Section 6.3.2, better WCBW bound can be obtained when increasing *DataSize*, since more pipelining between successive transactions can be exploited. However, the improvement on the WCBW bound is not always significant with larger *DataSize*. This is demonstrated in Figure 6.7(a), which takes 32-byte transactions as an example. As a result, *we derive a sufficient WCBW bound when it cannot increase dramatically with DataSize*. For example, we provide the best practical WCBW bounds when they cannot increase with more than 1%, and the results will be shown in Figure 6.8. Moreover, the WCBW bound does not increase for certain transaction sizes with DDR3-1600G, such as 16 bytes and 64 bytes. Figure 6.7(b) shows the verified WCBW bound for 16-byte transactions when increasing *DataSize*. Note that 16-byte transactions only use one bank ($BI = 1$). This bound is obtained when the transactions are write and they use the same bank. In this case, there is no pipelin-



(a) 32 Bytes



(b) 16 Bytes

Figure 6.7: The WCBW using different DataSize for fixed transaction sizes with DDR3-1600G.

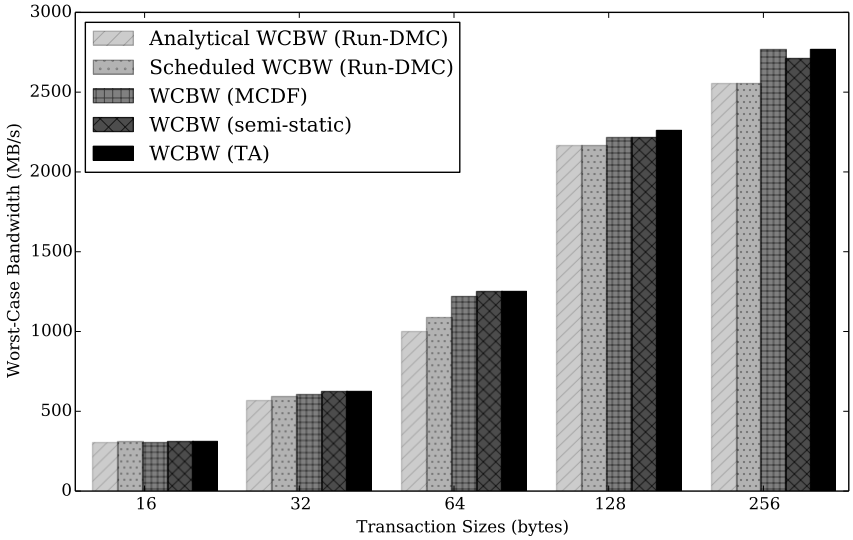


Figure 6.8: The WCBW for fixed transaction sizes.

ing between successive transactions. As a result, the bound can be verified with the minimum *DataSize* corresponding to a single transaction. In addition, it is not only the sufficient bound, but also the best bound that can be given by verifying properties of the TA model. The reason is that the bound is constant for different *DataSize*. Figure 6.7 only shows the results for 32-byte and 16-byte transactions, respectively, while the results for other transaction sizes are not given for brevity, since we can draw the same conclusions and derive sufficient WCBW bounds.

Figure 6.8 shows either the sufficient or the best WCBW bounds for fixed transaction sizes with DDR3-1600G, and they are compared to existing approaches. Compared to the analytical approach, the improvement reaches up to 25% for 64-byte transactions. The average improvement on the WCBW bounds when comparing to these approaches is 13.6% for all these experiments. The reasons for obtaining better WCBW than analytical and scheduled approaches are the same as those for WCRT. Our TA model is better than or performs equally well as the MCDF model previously presented in Chapter 5 because the latter conservatively assumes a collision per ACT command. Larger improvements are obtained for small transactions, while the same WCBW is obtained for large transaction sizes, e.g., 256 bytes. This is because larger transactions have more *RD* or *WR* commands, which dominate the command scheduling. As a result, the collisions with ACT commands have no influence on the WCBW, and the MCDF can perform equally well as the TA model for large transaction sizes. Finally, we compare the WCBW bounds of Run-DMC to the semi-static memory controller [3]. Figure 6.8 illustrates that Run-DMC provides larger WCBW bounds for all the transaction sizes. The only exception is

caused by 32-byte transactions, where the bound given by the semi-static memory controller is 0.2% higher than Run-DMC, because its static command schedules are slightly more efficient in the worst-case for 32-byte transactions. In contrast, Run-DMC schedules command dynamically and it may lead to initial bank states that make the execution of a 32-byte transaction experiences longer time than the semi-static approach. Recall from Chapter 3 that Run-DMC has much better average performance, which can benefit non-real-time applications.

We evaluate the run time and memory usage of Uppaal in our experiments. Uppaal takes at most 1221 seconds and consumes up to 7 GB to successfully verify properties for fixed transaction sizes. This occurs when verifying a property to derive the WCRT of 16-byte write transactions. Note that this experiment explores two different starting banks for 16-byte transactions. If more starting banks (e.g., 4 banks) are explored, it takes around 30 hours before running out of the 125 GB usable RAM. Due to the limited RAM memory, we alternatively carry out this experiment on a server with 1 TB RAM and an Intel(R) Xeon(R) CPU E7-4850 running at 2.0 GHz. Note that this server is remotely provided by SURFsara [100], a Dutch Cooperative providing high-performance computing and data infrastructure for science and industry. The model checking finally takes around 241.1 hours and consumes 557.9 GB RAM memory to provide the same results as when only exploring 2 banks. We have observed that larger transaction sizes need less time and RAM to verify a property. The reasons include: 1) Larger transactions use larger BI that have a fewer possible starting banks, resulting in a smaller state space. 2) The scheduling algorithm schedules more commands sequentially (i.e., deterministically) for larger transactions. In contrast, smaller transactions have fewer commands and transactions arrive randomly. As a result, the TA model performs more deterministic state transitions. Recall that the scheduling of commands is modeled by state transitions, e.g., the RW Scheduler TA in Figure 6.4(l).

6.5.4 *Variable Transaction Sizes*

Ideally, the TA model is used to analyze the WCRT and WCBW of any mix of transactions, e.g., resulting from different requestors, with different transaction sizes and starting banks. Due to the state space explosion, it is not possible to obtain general WCRT and WCBW for all combinations of transactions by model checking, because Uppaal fails to verify a property after consuming all the RAM (e.g., 125GB) of the host server. However, *system designers* are usually less interested in general WCRT and WCBW bounds than in response-time and bandwidth *bounds for a particular system under design*. By taking into account system-specific information, "design-specific bounds" can be expected to be closer to the true worst case that can occur in the design than "general bounds" that necessarily include traces that cannot occur in the particular system. System-specific information includes transaction sequences and sizes per

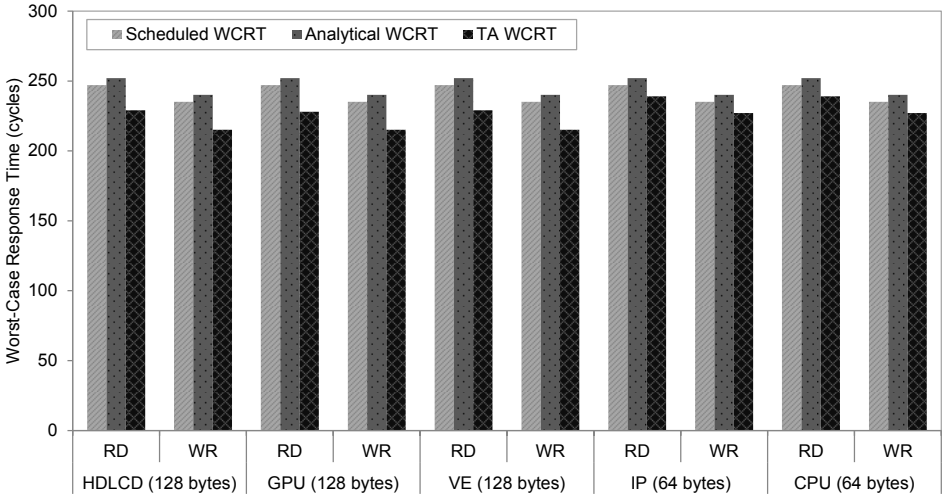


Figure 6.9: The WCRT for the requestors in a HD video and graphics processing system [31] with variable transaction sizes.

requestor, TDM allocations, etc. With this information, the analysis is less pessimistic, allowing for tighter bounds or lower system cost.

To illustrate this effect, we perform a case study of the HD video and graphics processing system of [31]. It consists of 5 requestors representing CPU, GPU, input processor (IP), video engine (VE), and HDLCD DMA controller. [31] focuses on a multi-channel memory controller with 256-byte transactions that are interleaved over multiple channels. Since our memory controller has a single channel, we use smaller transaction sizes, which are also current practice in today's systems [34, 44]. CPU and IP have cachelines of 64 bytes, while those of the GPU are 128 bytes. The VE and the HDLCD DMA use transactions of 128 bytes. All produce an arbitrary read/write mix of transactions. The five requestors have one TDM slot each, and are served in descending order of their transaction sizes. This ordering increases the bank parallelism between successive transactions [72], improving performance. Figure 6.9 separately shows the WCRT of read/write transactions, for each requestor. The results show that our TA model outperforms the analytical and scheduled approaches for WCRT, for the same reasons as discussed in Section 6.5.3. For example, our TA model improves the WCRT of 128-byte transactions of analytical and scheduled approaches by 10.4% and 8.5%, respectively. As before, all bounds have been validated to be identical to the cycle-accurate timings of the *RTMemController* tool.

The WCBW bounds obtained from different approaches are shown in Figure 6.10. Note that these results are the sufficient WCBW bounds, which are derived in the same way as those for the fixed transaction sizes in Section 6.5.3. For example, Run-DMC uses the TDM arbiter presented in Section 3.2.2 to serve the five requestors in the HD

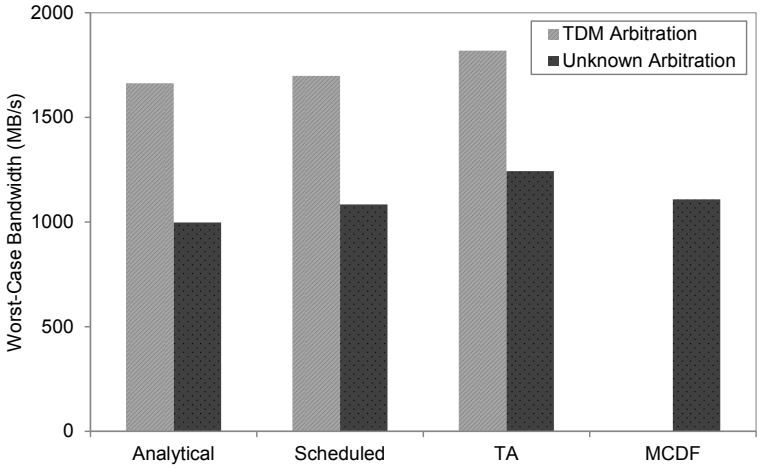


Figure 6.10: The WCBW for variable transaction sizes.

video and graphics processing system. The sufficient WCBW bound derived from our TA model is shown in Figure 6.10. It is obtained when the Observer (see Figure 6.5(b)) uses $DataSize = 2048\ bytes$, which corresponds to 20 successive transactions executed according to the TDM schedule. Moreover, the Uppaal verifier takes around 6.8 hours and consumes 30.3 GB RAM to provide this sufficient WCBW bound. The results in Figure 6.10 show that our TA model outperforms the analytical and scheduled approaches by 11.2% and 8.9%, respectively. We cannot compare to the MCDF model [71], as its analysis tool does not support our use-case. Instead, we compare all approaches for a system with five requestors with arbitrary read/write transactions of 64 or 128 bytes. The arbitration is unknown (not specified). The WCBW results are shown in Figure 6.10. The TA model outperforms the analytical, scheduled, and MCDF approaches by 24.6%, 14.6%, and 12.1%, respectively. The average improvement from all our experiments reaches to 14%.

6.6 SUMMARY

This chapter proposes a modular Timed Automata (TA) model of the Run-DMC memory controller, which has been previously introduced in Chapter 3. Bounds on worst-case response time (WCRT) and bandwidth (WCBW) are automatically derived by verifying properties of the TA model using model checking. The TA model is based on instantiating multiple TA templates, each of which describes either the timing behavior of a component used by Run-DMC or a JEDEC-specified DDR3 timing constraint. An intuitive TA model is given, followed by an optimized version. The former describes each memory controller component and timing constraint with a timed automaton, leading to a

straightforward model of Run-DMC. However, the verification of the intuitive TA model has to explore a large state-space, resulting in long verification time and high memory usage of the host server. In contrast, the latter optimized TA model captures the timing behavior of several components with a single automaton. In the same way, multiple timing constraints are tracked with a single automaton as well. Therefore, the optimized TA model provides the WCRT/WCBW bounds through model checking faster (e.g., 2.4x).

The proposed TA model is beneficial for analyzing real-time memory controllers, because: 1) it can easily capture the behavior of new memory controllers, since the automata templates used by our TA model can either be reused for common components or be easily extended. 2) The WCRT/WCBW bounds are provided automatically via model checking, as opposed to repeating a time-consuming manual analytical effort. 3) The TA model accurately captures the timing behavior of a memory controller without any abstractions, such as conservatively assuming a collision for each *ACT* command or the static maximum timing interval between commands. This is a prerequisite for tight WCRT and WCBW bounds. 4) The TA model not only provides the WCRT/WCBW bounds, but is also an executable model for simulating the memory controller. As a result, the TA model is easily validated by comparing the timings of each command from the simulation to those given by existing cycle-accurate simulators of the memory controller. 5) The verification of the TA model with Uppaal provides a witness of the worst-case results, i.e., the diagnostic traces of commands/transactions. By feeding the traces to the open-source *RTMemController* tool that is equivalent to a cycle-accurate SystemC simulator, identical timings of commands are obtained. This demonstrates that the TA is indeed without any abstraction and the bound is tight for the encountered traces. 6) Finally, the experimental results demonstrate that the proposed TA model outperforms three state-of-the-art analysis approaches of dynamic command scheduling for real-time memory controllers, by up to 25%. The reason is that the TA model accurately captures both the scheduling collisions on the command bus and the initial timing states of SDRAM for each transaction.

7

CONCLUSIONS AND FUTURE WORK

In this chapter, we conclude this thesis by briefly discussing the problems of design and analysis of real-time memory controllers, and summarizing our contributions presented in the previous chapters. Then, we provide possible extensions of the work in this thesis.

7.1 CONCLUSIONS

The off-chip SDRAM is shared by other on-chip resources to read or write data via a memory controller integrated in the SoC, where both real-time and non-real-time applications are concurrently executed. These resources generate diverse traffic for the memory controller, which receives arbitrary read/write transactions with variable sizes and different physical addresses. The memory controller must provide guaranteed performance for real-time applications to meet their requirements, while giving the best possible average performance to make the non-real-time applications feel responsive. The difficulty with achieving this goal is the complex interferences, which are caused by 1) different requestors that contend for the SDRAM, 2) the pipelining between transactions with variable sizes, since commands are scheduled to multiple banks in parallel subject to the JEDEC-specified timing constraints.

Next, Section 7.1.1 presents our contribution of designing a memory controller that efficiently deals with the diverse memory traffic. The proposed analysis approaches are summarized in Section 7.1.2, along with a discussion about their respective strengths and weaknesses.

7.1.1 *Design of Real-Time Memory controllers*

To efficiently deal with the diverse memory traffic, Chapter 3 proposes a memory controller named Run-DMC, which generates appropriate commands for transactions with variable sizes. The commands are dynamically scheduled according to the run-time SDRAM state and the timing constraints. It meets the design and analysis requirements posed on a real-time memory controller that must provide guaranteed performance to meet the requirements of real-time applications and good average performance to the non-real-time applications to feel responsive. Run-DMC outperforms a state-of-the-art

semi-static memory controller [3], because it uses the following mechanisms: 1) Run-DMC employs a novel TDM arbiter in the front-end to enable the worst-case response time (WCRT) and the worst-case bandwidth (WCBW) for requestors to be bounded. Requestors can be allocated a different number of time slots with variable lengths according to their requirements. Moreover, it skips idle slots to reduce the interference between requestors and configures the service order of requestors in descending of their transaction sizes. The latter results in a smaller execution time of a transaction, and has been experimentally demonstrated to achieve the minimum WCRT. 2) In the back-end, each transaction is executed with the configured bank interleaving number (BI) and burst count (BC) per bank. These two parameters provide the flexibility of exploiting different levels of bank parallelism. 3) To achieve better worst-case performance, a close-page policy is used by Run-DMC. This also eliminates the complexity of distinguishing page-hits and page-misses, resulting in a simpler analysis. 4) Transactions are executed by the back-end in a FCFS manner, such that the hardware overhead and analysis difficulty of transaction re-ordering are eliminated. 5) Finally, commands are dynamically scheduled as soon as the timing constraints are satisfied. This leads to efficient pipelining between successive transactions. In addition, the run-time state of the SDRAM is exploited to achieve good average performance.

Run-DMC has been implemented as a cycle-accurate SystemC model to measure the worst-case observed performance and the average behavior. Experimental results show that Run-DMC significantly outperforms an existing semi-static memory controller [3] in the average case, while they are comparable in the worst-case. For fixed transaction sizes with different DDR3 SDRAMs, the overall improvement reaches 17.3%. For variable transaction sizes, it achieves 44.9% smaller average RT according to the experiments. We have also observed that smaller transaction sizes benefit more from our dynamically-scheduled memory controller. For example, with variable transaction sizes, 79% improvement is achieved by 16-byte transactions to access a DDR3-1600G device, while it is 18.8% for 128-byte transactions. The reason is that smaller transactions require fewer banks, resulting in that the next transaction has higher chance to access different banks and thus be executed earlier. Finally, Run-DMC provides 16.7% more average bandwidth than the semi-static approach. In addition, higher bandwidth is obtained with larger transaction sizes. The reason is that more consecutive data bursts are transferred for each bank activation, resulting in higher efficiency of transferring data.

7.1.2 *Analysis Techniques for Real-Time Memory Controllers*

This thesis has proposed three approaches to analyze real-time memory controllers, based on a formal analysis, a mode-controlled dataflow (MCDF) model, and a timed automata (TA) model, respectively. These approaches have been used to analyze our memory controller with dynamic command scheduling, and provide the WCRT and/or WCBW. To the best of our knowledge, this is the first work to analyze a real-time mem-

ory controller with different techniques. Therefore, we discuss their strengths and weaknesses in the following sections.

7.1.2.1 *Analysis vs. Modeling*

The formal analysis approach accurately formalizes the command scheduling dependencies. This formalization has been implemented as an open-source tool called *RTMem-Controller* [70]. Then, conservative worst-case results are obtained by assuming that 1) the commands of the preceding transaction are scheduled as-late-as-possible (ALAP), resulting in the worst-case initial SDRAM state, and 2) each *ACT* command always collides with a *RD* or *WR* command. However, the obtained results are guaranteed to be conservative based on manual proofs, which are complex and very time-consuming to develop. In contrast, this problem has been resolved by two other approaches, which model the timing behavior of Run-DMC and employ existing tools to analyze these models, resulting in the WCRT and/or WCBW. The second approach is based on an MCDF model that analyzes the WCBW using the Heracles tool [79], which does not support analyzing the WCRT. The third approach captures the timing behavior of Run-DMC with a TA model, and the Uppaal tool suite [13] is used to provide both the WCRT and WCBW via model checking. However, the formal analysis approach has formally proved that the execution time of a transaction monotonically increases with its size, as shown in Theorem 2. Note that the formal analysis approach assumes that each requestor has a fixed transaction size (i.e., the maximum size from the requestor), while requestors have different sizes. Theorem 2 guarantees that the analysis results based on the maximum transaction size of a requestor are safe to use even if it generates transactions with variable sizes. For the MCDF model, Heracles cannot provide the worst-case response/execution time, and hence cannot derive Theorem 2. In contrast, the TA model would have to enumerate all the transaction sizes and obtain the bound for each of them individually. As a result, Theorem 2 can be experimentally obtained based on all these bounds. However, this takes a long time or even is not possible for some transaction sizes, such as 16-byte.

7.1.2.2 *Accuracy of Worst-Case Bound*

The bounds on the WCRT and WCBW vary between the approaches based on the formal analysis, the MCDF model, and the TA model. *The accuracy of the bound depends on the number of simplifying assumptions used by these approaches.* As explained in Chapter 4, our formal analysis results in an analytical approach and a scheduled approach with different assumptions in terms of *1) ALAP command scheduling* and *2) conservative collision per ACT command.* More specifically, the scheduled approach only assumes ALAP command scheduling and eliminates the collision assumption by actually detecting the collisions with a tool. The MCDF model does not assume ALAP scheduling. However, it conservatively assumes a collision for each *ACT* command to avoid the unpredictable

collisions in the model. A collision occurs when all the timing constraints are satisfied for scheduling an *ACT* and a *RD/WR* command at the same time. Since dataflow models only capture the data dependent behavior, they cannot accurately model the collisions that depend on time. In contrast, the TA model does not apply any assumptions, since it can dynamically detect the collisions, based on which state transitions are enabled. Therefore, the TA model provides the tightest bounds. The MCDF model is slightly worse than the TA model, although it provides tighter WCBW bounds than both the scheduled and analytical approaches. Finally, the scheduled approach gives tighter bounds than the analytical approach. Another reason for the varied bounds achieved by these approaches is the exploitation of different degrees of pipelining between transactions. The formal analysis (i.e., analytical and scheduled) approach only captures pipelining with the previous transaction. However, the MCDF model and the TA model can exploit pipelining within a sequence of transactions.

These approaches have been experimentally compared in Section 6.5. For fixed transaction sizes, TA provides a WCRT that is maximally 20% smaller than the analytical approach for 64-byte write transactions. The average improvement is 7.7% according to the experiments. The improvement of the WCBW is up to 25% for 64-byte transactions, compared to the analytical approach. The total average improvement on WCBW reaches 10%, 6.5%, and 1.9% compared to the analytical, scheduled, and the MCDF approaches, respectively. For variable transaction sizes, the TA model gives a smaller WCRT than the analytical and scheduled approaches, and the total average improvement is 7.1%. The WCBW results show that the TA model outperforms the analytical, scheduled, and MCDF approaches by 24.6%, 14.6%, and 12.1%, respectively.

7.1.2.3 *Portability for New Memory Controllers*

When extending the proposed analysis approaches to new memory controllers using different mechanisms (e.g., open-page policy, different command-level priorities) and/or different memory devices, varied efforts are needed. Toward this issue, the TA model is superior to others. The reason is that the TA model is more expressive and can accurately capture the dynamic behavior of a memory controller. It is a network of timed automata, where each automaton describes the behavior of a component in the memory controller, resulting in a modular model. In contrast, to model a memory controller with MCDF, proper modes must be created, followed by determining the transitions between modes. Then, static mode sequences are needed to capture some dynamic behaviors. All these cannot be easily done on a per-component basis. The MCDF model follows the structure of the memory controller less, instead follows the resulting behavior more. Therefore, the MCDF model requires more effort than the TA model for a new memory controller or SDRAM device. The most difficult approach is the formal analysis, which is based on a formalization of the command scheduling dependencies. Although the proposed formalization is parameterized based on the transaction size and the timing constraints of DDR3 SDRAMs, it is tailored for a close-page policy and the FCFS sche-

duling algorithm. When an open-page policy and/or reordering scheme are used, the command scheduling dependencies are changed. Since the dependencies are complex, it needs significant effort to extend the formalization. For the same reason, the formal analysis approach takes much effort to adapt to new memory controllers.

7.1.2.4 *Scalability*

The scalability of the proposed analysis approaches varies when the memory controller is configured to support different number of requestors with fixed or variable transaction sizes. The formal analysis approach is parameterized to the transaction size and number of requestors. As a result, it is scalable to deal with different configurations of the memory controller, though it sacrifices the accuracy of the worst-case bounds. The approaches based on the MCDF model and the TA model cannot easily scale with different configurations. The verification of the TA model takes a long time and consumes a lot of RAM on the host server to derive the results, when exploring the state space via model checking with Uppaal. The MCDF model is slightly better in this regard. The reason is that each mode of the MCDF model is a smaller single-rate dataflow (SRDF) graph that has a deterministic execution behavior, and the transitions between modes are statically predefined. As a result, the state space explored by the MCDF model is smaller than the TA model. The Heracles [79] tool is used to analyze the WCBW bounds by converting the MCDF graph into its equivalent SRDF, where the critical cycle is easily obtained. However, Heracles cannot analyze the WCRT. Moreover, it is an Ericsson internal tool that is not publicly available to the research community. However, it is worth noting that MCDF model is very similar to the scenario-aware dataflow (SADF) [95, 101], which can be analyzed by the open-source tool SDF^3 [98].

The verification of the TA model may experience the state-space explosion problem, and the dataflow graph may be too large to be analyzed. This limits the scalability of these models. With different configurations of our dynamically-scheduled memory controller, Table C.2 presented in Appendix C collects the time and RAM consumed by analyzing the MCDF model with Heracles and the verification of the TA model using Uppaal, respectively. The results demonstrate that on average, the analysis of the MCDF model is 1150.3 times faster and consumes 97.1% less RAM than the verification of the TA model. However, Heracles cannot analyze a large number of static mode sequences (SMSs) capturing the behavior of executing transactions interleaved over different number of banks. As a result, it cannot analyze the case when requestors are served in a static order, e.g., a TDM manner, which is captured by more specific SMSs. In contrast, the verification of the TA model can easily address the static service order, since this reduces the size of the state-space. When the arbitration between requestors is unknown, the state-space is larger and the verification can easily fail, i.e., terminate when all the RAM of the host server is consumed. In this case, the MCDF model needs fewer SMSs, and it can analyze the WCBW. The MCDF and TA models thus behave conversely. The MCDF model is large (e.g., consisting of around 500 actors and 990 edges)

but it reduces the state space, resulting in fast run-time; whereas the TA model is compact (e.g., 137 nodes and 186 edges) because of its good expressiveness. However, the state space is large, leading to a large run-time.

7.1.2.5 *Exploitation of Static/System Information*

The analysis of the memory controller can give better worst-case bounds when statically exploiting more information about the memory requestors and the transaction schedule. For example, the TDM arbiter determines a static order of executing transactions. The formal analysis approach can benefit from the case where the size of the preceding transaction is known, and obtains 9.3% tighter bound on WCBW according to the results shown in Figure 4.11. In this way, when the TDM arbiter provides a static order of transactions, it gains 10% improvement compared to the case when the order is unknown, i.e., the preceding transaction is unknown as well. However, the MCDF model and the TA model are able to analyze a complete static sequence of transactions (i.e., with particular sizes, types, and physical addresses). Since more static information is exploited, they provide much better results. For example, the MCDF model achieves 63.2% improvement on the WCBW when the static sequence of transactions from two requestors is given by the TDM arbiter. The TA model provides 32.8% more WCBW when the TDM arbiter is used to serve 5 requestors in a HD video and graphic processing system. Note that it is hard to use the same experimental setup for the MCDF model and the TA model, due to the limitations of their analysis tools. Hence, these numbers are not comparable.

To exploit even more information of the system, the model of a memory controller can be integrated into a higher level model to analyze the behavior of a system and provide better worst-case results. For example, the TA model of a memory controller could be integrated into another TA model that captures the behavior of the rest of the system. As a result, a larger TA model can be used to analyze the worst-case execution time or the minimum throughput of an application, where its tasks or jobs are running on cores while generating memory transactions. When integrating the MCDF model of the memory controller into a dataflow model describing the system, new modes must be created to capture the complex behaviors. It may be too difficult to obtain the modes or the final MCDF model is too large to be analyzed by Heracles. On contrary, the only concern for the verification of the TA model is the state-space explosion issue. The reasons are that 1) the SDRAM banks are non-deterministically used, and 2) the command collisions are unpredictable. However, in a realistic system, the allocation of the banks to cores are known, e.g., using a static memory mapping. In addition, the large TA model can also apply the conservative assumption of collision per *ACT* command to simplify the model, although this slightly degrades the accuracy of the worst-case results. However, the large TA model may gain enough benefits to offset this drawback, since more static information of the system can be exploited.

7.1.2.6 *Simulation, Validation, and Verification*

In addition to the worst-case bounds, portability for new memory controllers, and the exploitation of static/system information, the formal analysis approach, the MCDF model, and the TA model also provide other features. Firstly, the MCDF and TA models are executable and support simulation of the memory controller. As a result, these models can be validated by the cycle-accurate SystemC model of Run-DMC. Moreover, the analysis of the MCDF model returns the critical cycle corresponding to a sequence of transactions that cause the WCBW bound. By feeding these transactions to the SystemC simulator, an identical bandwidth is obtained. Similarly, the verification of the TA model gives a diagnostic transaction trace, which is a witness of the worst-case bound. This bound is validated to be tight in the same way. In contrast, the bounds computed by the formal analysis approach cannot be validated like this. The confidence in the bounds can be improved by simulation, where the measured experimental results do not break the bounds.

7.2 FUTURE WORK

This section introduces two feasible extensions of the work in this thesis. We firstly discuss the extension of the proposed formal analysis approach to obtain the optimal memory mapping for requestors. Secondly, we show how to improve the average performance and the memory utilization using Run-DMC.

7.2.1 *Bank-Aware Memory Mapping*

A memory requestor typically uses a range of memory addresses, which are mapped to specific locations inside the SDRAM, e.g., a number of banks between which bank interleaving is employed. However, *an issue of bank interleaving is how to statically allocate the consecutive banks to a requestor, such that its WCRT is minimized while the overall WCBW is maximized.* This requires an optimal bank-aware memory mapping. To achieve this goal, an analysis framework is needed, such that the WCRT/WCBW is analyzed by capturing the transaction size and the allocated banks per requestor, as well as the service order of requestors specified by a TDM or a RR arbiter. This analysis framework will be an extension of the analytical approach introduced in Chapter 4 from only capturing the preceding transaction size to covering sizes, allocated banks to requestors, and the service order of requestors. Based on this analysis framework, a holistic approach can be used to explore the optimal bank allocation and TDM service order of requestors with variable transaction sizes, such that the lowest WCRT and highest WCBW can be obtained.

7.2.2 *Enhancement of SDRAM Utilization with Hybrid Page Policies*

SDRAM is a scarce resource and it must be efficiently used to not only provide good performance, but also reduce cost. However, state-of-the-art real-time memory controllers have been focusing on providing guaranteed performance, while the memory utilization has not been given sufficient attention. The basic argument is that guaranteed performance can be achieved by sacrificing memory utilization. An example is bank privatization, which spatially isolates the memory. It prevents the interference of sharing banks, resulting in predictable performance for the requestor running real-time tasks. However, many real-time applications are not memory intensive. For example, control of a mechanical motor typically consists of the procedures of sensing, processing, and actuation. It does not generate heavy memory traffic. Moreover, bank privatization does not support data sharing in SDRAM. For example, Wu et al. [108] add separate queues in the memory controller to support data sharing when bank privatization is used.

To enhance the SDRAM utilization and provide an efficient memory sharing mechanism, our dynamically-scheduled memory controller can be extended to use the following mechanisms. 1) Banks are allocated to requestors according to their data requirements. 2) Since one requestor may not consume the whole capacity of the allocated bank(s), the banks can also be used by other requestors. 3) To provide good worst-case performance for real-time applications, a close-page policy will be used. 4) To achieve good average performance by non-real-time applications, an open-page policy can be used to exploit page-hits. With this hybrid page policy, guaranteed performance will be given to real-time applications, while providing best possible average performance for the rest. In particular, memory banks are shared and the utilization can be increased. Finally, the timed automata model can be extended to capture this hybrid page policy and the allocation of SDRAM banks to requestors, where the worst-case results can be automatically obtained by model checking.

BIBLIOGRAPHY

[1] PrimeCell AHB SDR and NAND Memory Controller (PL242). http://www.arm.com, 2006. (Cited on page 34.)

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, 1998. (Cited on pages 1 and 6.)

[3] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2011. (Cited on pages xi, xii, 6, 32, 45, 46, 47, 48, 51, 52, 77, 78, 79, 80, 81, 92, 93, 111, 115, 134, 137, 140, and 146.)

[4] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, 2007. (Cited on pages xi, xii, 31, 53, 54, 55, 56, 58, and 59.)

[5] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA, pages 3–14, 2008. (Cited on pages 7, 19, and 34.)

[6] B. Akesson, L. Steffens, and K. Goossens. Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 59–68, 2009. (Cited on page 7.)

[7] B. Akesson, W. Hayes, and K. Goossens. Classification and analysis of predictable memory patterns. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 367–376, 2010. (Cited on page 18.)

[8] B. Akesson, A. Minaeva, P. Sucha, A. Nelson, and Z. Hanzalek. An efficient configuration methodology for time-division multiplexed single resources. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 161–171, 2015. (Cited on pages 3, 37, and 45.)

[9] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999. (Cited on page 1.)

[10] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. (Cited on pages 45 and 76.)

[11] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and Y. Wang. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, 2014. (Cited on pages 1, 5, 34, 57, and 130.)

[12] S. Bayliss and G. A. Constantinides. Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search. In *International Conference on Field-Programmable Technology*, FPT, pages 304–307, 2009. (Cited on pages 32 and 58.)

[13] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on Quantitative Evaluation of Systems (QEST)*, pages 125–126, 2006. (Cited on pages 12, 118, 119, and 147.)

[14] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core SoC with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, 2008. (Cited on page 3.)

[15] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004. (Cited on pages 12, 31, and 117.)

[16] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 396–407, 2003. (Cited on pages 1 and 6.)

[17] C. V. Briciu, I. Filip, and F. Heininger. A new trend in automotive software: Autosar concept. In *2013 IEEE 8th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 251–256, 2013. (Cited on page 3.)

[18] J. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 508–513 vol.1, Oct 1994. (Cited on page 95.)

[19] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 24. Springer, 2011. (Cited on page 1.)

[20] *Multi-Protocol LPDDR4/3/DDR4/3 Controller and PHY Subsystem IP.* Cadence Design Systems Inc., 2014. (Cited on page 43.)

[21] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075, 2011. (Cited on page 5.)

[22] R. de Groote, J. Kuper, H. Broersma, and G. J. M. Smit. Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 29–38, Sept 2012. (Cited on page 99.)

[23] M. Dev Gomony, B. Akesson, and K. Goossens. Coupling TDM NoC and DRAM controller for cost and performance optimization of real-time systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2014. (Cited on page 19.)

[24] L. Ecco and R. Ernst. Improved DRAM timing bounds for real-time DRAM controllers with read/write bundling. In *IEEE Real-Time Systems Symposium*, pages 53–64, 2015. (Cited on pages 33, 66, and 71.)

[25] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2014. (Cited on pages 6, 32, 33, 59, and 93.)

[26] L. Ecco, S. Saidi, A. Kostrzewa, and R. Ernst. Real-time DRAM throughput guarantees for latency sensitive mixed QoS MPSoCs. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015. (Cited on pages 32, 66, 71, and 93.)

[27] M. Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, 2011. (Cited on page 94.)

[28] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design, ACSD*, pages 25–36, 2006. (Cited on page 99.)

[29] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT, pages 63–72. ACM, 2012. (Cited on page 5.)

[30] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1307–1312, 2013. (Cited on page 19.)

[31] M. D. Gomony, B. Akesson, and K. Goossens. A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. *ACM Trans. Embed. Comput. Syst.*, 14(2):25:1–25:27, 2015. ISSN 1539-9087. (Cited on pages xiii, 51, 112, and 142.)

[32] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared DRAM access in real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 193–198, 2015. (Cited on pages 19 and 38.)

[33] K. Goossens and A. Hansson. The Aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 306–311. ACM, 2010. (Cited on page 4.)

[34] K. Goossens, O. P. Gangwal, J. Röover, and A. Niranjan. *Interconnect-Centric Design for Advanced SoC and NoC*, chapter Interconnect and Memory Organization in SOCs for Advanced Set-Top Boxes and TV, pages 399–423. Springer US, Boston, MA, 2005. ISBN 978-1-4020-7836-1. (Cited on page 142.)

[35] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *SIGBED Rev.*, 10(3):23–34, 2013. (Cited on page 117.)

[36] S. Goossens, T. Kouters, B. Akesson, and K. Goossens. Memory-map selection for firm real-time SDRAM controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 828–831, 2012. (Cited on pages 18, 19, 39, and 45.)

[37] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 525–530, 2013. (Cited on pages 6, 19, 32, 34, and 59.)

[38] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. (Cited on page 43.)

[39] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. Power/performance trade-offs in real-time SDRAM command scheduling. *IEEE Transactions on Computers*, PP(99):1–1, 2015. (Cited on pages 6, 19, 20, 32, 41, 99, 108, 109, 120, 124, 127, 134, and 136.)

[40] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. *Memory Controllers for Mixed-Time-Criticality Systems: Architectures, Methodologies and Trade-offs*. Embedded Systems Series. Springer, first edition edition, 2016. ISBN 978-3-319-32093-9. (Cited on page 19.)

[41] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. (Cited on page 1.)

[42] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using uppaal. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010. (Cited on page 135.)

[43] F. Hameed, L. Bauer, and J. Henkel. Simultaneously optimizing DRAM cache hit latency and miss rate via novel set mapping policies. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 11:1–11:10, 2013. (Cited on page 20.)

[44] A. Hansson and K. Goossens. A quantitative evaluation of a network on chip design flow for multi-core consumer multimedia applications. *Design Automation for Embedded Systems*, 15(2):159–190, 2011. (Cited on page 142.)

[45] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *Computers Digital Techniques, IET*, 3(5):398–412, 2009. ISSN 1751-8601. (Cited on page 93.)

[46] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 307–316, 2015. (Cited on pages 3, 59, 93, and 134.)

[47] B. Heidergott, G. J. Olsder, and J. W. v. d. Woude. *Max Plus at work : modeling and analysis of synchronized systems : a course on Max-Plus algebra and its applications*. Princeton series in applied mathematics. Princeton University Press, Princeton (N.J.), 2006. ISBN 0-691-11763-2. (Cited on page 99.)

[48] I. Hur and C. Lin. Memory scheduling for modern microprocessors. *ACM Trans. Comput. Syst.*, 25(4), 2007. (Cited on page 32.)

[49] A. Intrater, M. Doron, G. Intrater, L. Epstein, M. Valentaten, and I. Greiss. Integrated digital signal processor/general purpose CPU with shared internal memory, May 13 1997. US Patent 5,630,153. (Cited on page 3.)

[50] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, pages 39–50, 2008. (Cited on page 32.)

[51] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007. (Cited on page 13.)

[52] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. Cazorla. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 207–217, 2014. (Cited on pages 33, 59, 66, and 71.)

[53] JEDEC. DDR3 SDRAM specification JESD79-3E, 2010. (Cited on pages xiv, 14, 15, 24, 29, 40, 41, 46, 66, 70, 78, 92, 99, 110, 126, 127, 131, 136, and 166.)

[54] L. Karam, I. Alkamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans. Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, 26(6):38–49, 2009. (Cited on page 3.)

[55] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154, 2014. (Cited on pages 6, 33, 43, 59, 101, and 135.)

[56] H. Kim, D. Broman, E. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 317–326, 2015. (Cited on pages 6, 66, 71, and 135.)

[57] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003. (Cited on page 3.)

[58] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling. *Micro, IEEE*, 31(1):78–89, 2011. (Cited on page 32.)

[59] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA'12, pages 368–379, 2012. (Cited on page 5.)

[60] P. Kollig, C. Osborne, and T. Henriksson. Heterogeneous multi-core platform for consumer multimedia applications. In *Design, Automation Test in Europe Conference Exhibition*, DATE, pages 1254–1259, 2009. (Cited on pages 1, 3, and 33.)

[61] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997. (Cited on page 2.)

[62] M. Krichen and S. Tripakis. *11th International SPIN Workshop on Model Checking Software*, chapter Black-Box Conformance Testing for Real-Time Systems, pages 109–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Cited on page 118.)

[63] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A rank-switching, open-row DRAM controller for time-predictable systems. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 27–38, 2014. (Cited on pages 32, 43, 59, 66, 71, and 135.)

[64] K. Lampka *et al.* A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50 (5-6):736–773, 2014. (Cited on page 135.)

[65] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO, pages 330–335, 1997. (Cited on pages 45 and 76.)

[66] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. (Cited on page 94.)

[67] A. Lele, O. Moreira, and P. J. Cuijpers. A new data flow analysis model for TDM. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 237–246, 2012. (Cited on page 93.)

[68] A. Lele, O. Moreira, and K. van Berkel. FP-scheduling for mode-controlled dataflow: A case study. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1257–1260, 2015. (Cited on pages 92, 98, 107, and 115.)

[69] Y. Li, B. Akesson, and K. Goossens. Dynamic command scheduling for real-time memory controllers. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–14, 2014. (Cited on pages 59, 79, 93, 121, and 136.)

[70] Y. Li, B. Akesson, and K. Goossens. RTMemController: Open-source WCET and ACET analysis tool for real-time memory controllers. http://www.es.ele.tue.nl/rtmemcontroller/, 2014. (Cited on pages xii, 8, 10, 44, 58, 63, 72, 75, 76, 77, 78, 118, 135, 136, and 147.)

[71] Y. Li, H. Salunkhe, J. Bastos, O. Moreira, B. Akesson, and K. Goossens. Mode-controlled data-flow modeling of real-time memory controllers. In *13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10, 2015. (Cited on pages 134 and 143.)

[72] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, 52(5):675–729, 2016. (Cited on pages 29, 34, 59, 79, 93, 121, 124, 134, 135, and 142.)

[73] Y. Li, B. Akesson, K. Lampka, and K. Goossens. Timed automata model of a dynamically-scheduled real-time memory controller. http://www.es.ele.tue.nl/rtmemcontroller/TA.zip, 2016. (Cited on pages 8 and 120.)

[74] W.-F. Lin, S. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, pages 301–312, 2001. (Cited on page 20.)

[75] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 339–349, 2010. (Cited on page 135.)

[76] M. Mehendale, S. Das, M. Sharma, M. Mody, R. Reddy, J. Meehan, H. Tamama, B. Carlson, and M. Polley. A true multistandard, programmable, low-power, full HD video-codec engine for smartphone SoC. In *2012 IEEE International Solid-State Circuits Conference*, pages 226–228, 2012. (Cited on page 3.)

[77] A. Minaeva, P. Šůcha, B. Akesson, and Z. Hanzálek. Scalable and efficient configuration of time-division multiplexed resources. *Journal of Systems and Software*, 113:44 – 58, 2016. (Cited on pages 3, 6, and 45.)

[78] S. K. Mitra and Y. Kuo. *Digital signal processing: a computer-based approach*, volume 2. McGraw-Hill New York, 2006. (Cited on page 3.)

[79] O. Moreira and H. Corporaal. *Scheduling Real-Time Streaming Applications Onto an Embedded Multiprocessor*. Springer, 2014. (Cited on pages 11, 31, 91, 92, 93, 95, 97, 109, 147, and 149.)

[80] A. Nelson, K. Goossens, and B. Akesson. Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor {SOC}. *Journal of Systems Architecture*, 61(9):435 – 448, 2015. (Cited on pages 5, 93, and 117.)

[81] C. Norstrom, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 182–189, 1999. (Cited on pages 118 and 135.)

[82] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference (EDCC)*, pages 132–143, 2012. (Cited on page 3.)

[83] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The rosace case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318, April 2014. (Cited on page 2.)

[84] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4), 2009. (Cited on page 59.)

[85] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1):64:1–64:26, 2013. (Cited on pages 6, 19, 32, and 59.)

[86] Qualcomm. *Snapdragon S4 processors: System on chip solutions for a new mobile age*. Qualcomm White Paper, 2011. (Cited on page 3.)

[87] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 245–258, 2007. (Cited on page 93.)

[88] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, 2011. (Cited on pages 6, 32, 59, 66, 93, and 134.)

[89] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Appl. Ontol.*, 1(1):77–106, Jan. 2005. (Cited on page 2.)

[90] H. Salunkhe, O. Moreira, and K. van Berkel. Mode-controlled dataflow based modeling & analysis of a 4G-LTE receiver. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–4, 2014. (Cited on pages 92 and 115.)

[91] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 759–764, 2010. (Cited on page 5.)

[92] H. Shah, A. Raabe, and A. Knoll. Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 665–670, 2012. (Cited on page 32.)

[93] H. Shah, A. Knoll, and B. Akesson. Bounding sdram interference: Detailed analysis vs. latency-rate analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 308–313, 2013. (Cited on page 122.)

[94] J. Shen, A. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload partitioning for accelerating applications on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2015. (Cited on page 4.)

[95] F. Siyoum, M. Geilen, O. Moreira, and H. Corporaal. Worst-case throughput analysis of real-time dynamic streaming applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 463–472, 2012. (Cited on page 149.)

[96] R. Stefan, A. Molnos, and K. Goossens. dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up. *IEEE Transactions on Computers*, 63(3):583–594, 2014. (Cited on page 19.)

[97] A. Stevens. *"QoS for High-Performance and Power-Efficient HD Multimedia"*. ARM White paper, 2010. (Cited on page 33.)

[98] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *6th International Conference on Application of Concurrency to System Design*, ACSD, pages 276–278. http://www.es.ele.tue.nl/sdf3, 2006. (Cited on pages 91 and 149.)

[99] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems (SAMOS)*, pages 404–411, 2011. (Cited on page 4.)

[100] SURFsara. https://www.surf.nl/en/about-surf/subsidiaries/surfsara. (Cited on page 141.)

[101] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE*, pages 185–194, 2006. (Cited on page 149.)

[102] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu. Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim.*, 12(4):65:1–65:28, 2016. (Cited on page 3.)

[103] C. H. K. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, pages 1260–1265, 2009. (Cited on page 3.)

[104] K. W., C. H., C. H.-D., , and K. Y. *Enjoy the ultimate WQXGA solution with Exynos 5 Dual.* Samsung Electronics White Paper, 2012. (Cited on page 3.)

[105] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory system simulator. *SIGARCH Comput. Archit. News*, 33:100–107, 2005. (Cited on page 32.)

[106] W. Wang, P. Mishra, and A. Gordon-Ross. Dynamic cache reconfiguration for soft real-time systems. *ACM Trans. Embed. Comput. Syst.*, 11(2):28:1–28:31, 2012. (Cited on page 2.)

[107] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 372–383, 2013. (Cited on pages 32, 59, 66, and 71.)

[108] Z. P. Wu, R. Pellizzoni, and D. Guo. A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems*, pages 1–47, 2016. (Cited on pages 17 and 152.)

[109] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308, 2012. (Cited on pages 5 and 93.)

[110] H. Yun, R. Pellizzoni, and P. Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 184–195, 2015. (Cited on page 33.)

A

PROOF OF LEMMAS

A.1 PROOF OF LEMMA 1

Proof. To prove Lemma 1 that states the finishing time of a transaction is only determined by either the finishing time of the previous transaction, or the scheduling time of its *ACT* commands, we only need to iteratively compute the scheduling time of its *RD* or *WR* commands and finally obtain the scheduling time of the last *RD* or *WR* command, which is defined as the finishing time of the transaction by Definition 5.

For an arbitrary transaction T_i ($i \geq 0$) that has BI_i and BC_i, its finishing time $t_f(T_i)$ is shown in Eq. (A.1), which is the scheduling time of its last *RD* or *WR* (named *RW*) command.

$$t_f(T_i) = t(RW^{BC_i-1}_{j+BI_i-1}) \tag{A.1}$$

According to Eq. (4.3) that gives the scheduling of a *RW* command, the scheduling time of the last *RW* command in Eq. (A.1) is given by Eq. (A.2). We see that this is determined by the scheduling time of the first *RW* to the same bank.

$$t(RW^{BC_i-1}_{j+BI_i-1}) = t(RW^{0}_{j+BI_i-1}) + (BC_i - 1) \times tCCD \tag{A.2}$$

Eq. (4.2) provides the scheduling time of the first *RW* command to a bank, which is determined by either the scheduling time of the *ACT* command to the same bank due to *tRCD*, or that of the previously scheduled *RW* command for the same transaction because of *tCCD*, which is the last command to the previous bank. As a result, $t(RW^{0}_{j+BI_i-1})$ in Eq. (A.2) is derived based on Eq. (4.2), and is shown in Eq. (A.3).

$$t(RW^{0}_{j+BI_i-1}) = max\{t(ACT_{j+BI_i-1}) + tRCD, t(RW^{BC_i-1}_{j+BI_i-2}) + tCCD\} \tag{A.3}$$

We proceed by combining Eq. (A.1), (A.2) and (A.3) to obtain a new expression of the finishing time, as given by Eq. (A.4).

$$\begin{aligned} t_f(T_i) = max\{&t(ACT_{j+BI_i-1}) + tRCD + (BC_i - 1) \times tCCD, \\ &t(RW^{BC_i-1}_{j+BI_i-2}) + BC_i \times tCCD\} \end{aligned} \tag{A.4}$$

In the same way, we can compute $t(RW^{BC_i-1}_{j+BI_i-2})$ in Eq. (A.4), and it can be further expressed by $t(ACT_{j+BI_i-2})$ and $t(RW^{BC_i-1}_{j+BI_i-3})$. We iteratively substitute the scheduling time

of the last RW command to each bank of transaction T_i, and Eq. (A.5) is derived, which consists of BI_i number of terms.

$$\begin{aligned} t_f(T_i) = \underset{1\le l\le BI_i-1}{\text{Max}} \{ & t(RW_j^{BC_i-1}) + (BI_i-1)\times BC_i \times tCCD, \\ & t(ACT_{j+l}) + tRCD + \left[(BI_i-l)\times BC_i - 1\right]\times tCCD\} \end{aligned} \tag{A.5}$$

We proceed by substituting $t(RW_j^{BC_i-1})$ in Eq. (A.5) with the scheduling time of the first RW to the same bank, which is given by Eq. (A.6), and is derived according to Eq. (4.3).

$$t(RW_j^{BC_i-1}) = t(RW_j^0) + (BC_i-1)\times tCCD \tag{A.6}$$

Furthermore, $t(RW_j^0)$ in Eq. (A.6) can be computed based on Eq. (4.2). As a result, Eq. (A.7) is obtained. Note that $t_f(T_{i-1})$ is the finishing time of the previous transaction T_{i-1} that is also the scheduling time of the last RW command of the previous bank. It was scheduled just before RW_j^0 and the timing constraint between them is $tSwitch$ (given by Eq. (3.2)).

$$t(RW_j^0) = \max\{t(ACT_j) + tRCD, t_f(T_{i-1}) + tSwitch\} \tag{A.7}$$

By combining Eq. (A.5), (A.6) and (A.7), Eq. (A.8) is derived.

$$\begin{aligned} t_f(T_i) = \underset{0\le l\le BI_i-1}{\text{Max}} \{ & t_f(T_{i-1}) + tSwitch + (BI_i\times BC_i - 1)\times tCCD, \\ & t(ACT_{j+l}) + tRCD + \left[(BI_i-l)\times BC_i - 1\right]\times tCCD\} \end{aligned} \tag{A.8}$$

Hence, for $\forall l \in [0, BI_i-1]$, $t_f(T_i)$ is expressed by Eq. (A.8). It indicates that $t_f(T_i)$ only depends on the scheduling times of its ACT commands, the finishing time of T_{i-1}, the memory map configuration in terms of BI_i and BC_i and the JEDEC-specified timing constraints, which are constant values. □

A.2 PROOF OF LEMMA 2

Proof. For $\forall l \in (b_{com}, BI_i-1]$, the scheduling time of the command ACT_{j+l} to bank $b_j + l$ can be obtained from Eq. (4.1). It indicates $t(ACT_{j+l})$ is determined by $t(ACT_{j+l-1})$, $t(ACT_{j+l-4})$ or $t(PRE_m)$, where m was the latest bank access number to bank $b_j + l$ before T_i. This lemma can be proved by simplifying Eq. (4.1) to derive the scheduling time of ACT_{j+l}, which is finally given by Eq. (A.18). First, a simplified Eq. (A.9) is obtained because of the dominance of $\hat{t}(ACT_{j+l-1})$ in this case. We proceed by explaining its derivation.

$$\begin{aligned} \hat{t}(ACT_{j+l}) = & max\{\hat{t}(ACT_{j+l-1}) + tRRD, \hat{t}(PRE_m) + tRP, \\ & \hat{t}(ACT_{j+l-4}) + tFAW\} + C(j+l) \\ = & \hat{t}(ACT_{j+l-1}) + tRRD + C(j+l) \end{aligned} \tag{A.9}$$

$\hat{t}(ACT_{j+l-1})$ dominates in the max{} of Eq. (A.9). We demonstrate this by showing two relations between the terms in the expression: i) $\hat{t}(ACT_{j+l-1}) > \hat{t}(PRE_m) + tRP$. For $\forall l > b_{com}$, Eq. (4.1) is employed to derive Eq. (A.10), which shows a later bank access (larger l) has a larger scheduling time of the *ACT* command. This is intuitive since the scheduling algorithm (Algorithm 2) schedules *ACT* commands in order.

$$BI_i - 1 \geq \forall l > b_{com} \Longrightarrow \hat{t}(ACT_{j+l-1}) \geq \hat{t}(ACT_{j+b_{com}}) \tag{A.10}$$

The command $ACT_{j+b_{com}}$ is scheduled to bank $b_j + b_{com} = b_{j-1}$ that is the finishing bank of T_{i-1}. As a result, Eq. (A.11) is derived on the basis of Eq. (4.1). It simply states that a bank cannot be activated until it has been precharged.

$$b_j + b_{com} = b_{j-1} \Longrightarrow \hat{t}(ACT_{j+b_{com}}) \geq \hat{t}(PRE_{j-1}) + tRP \tag{A.11}$$

Moreover, the precharge of a bank is triggered by the auto-precharge flag appended to a *RD* or *WR* command, which is issued sequentially. Therefore, banks are precharged in the order of bank accesses, resulting in Eq. (A.12), where the latest access number m for bank $b_j + l$ is smaller than the latest bank access number $j - 1$. Finally, by substituting Eq. (A.10), (A.11) and (A.12), we can prove the relation that $\hat{t}(ACT_{j+l-1}) > \hat{t}(PRE_m) + tRP$.

$$\forall m < j - 1 \Longrightarrow \hat{t}(PRE_{j-1}) > \hat{t}(PRE_m) \tag{A.12}$$

ii) $\hat{t}(ACT_{j+l-1}) > \hat{t}(ACT_{j+l-4}) + tFAW$. According to Eq. (4.4), we can obtain Eq. (A.13), which shows that the precharging time of a bank is after issuing the last *RD* or *WR* command of a transaction to the same bank.

$$\hat{t}(PRE_{j-1}) \geq \hat{t}(RW_{j-1}^{BC_{i-1}-1}) + tRWTP \tag{A.13}$$

With Eq. (4.2) and (4.3) that capture the timing dependencies for a *RD* or *WR* command, Eq. (A.14) is derived and it indicates that the last *RD* or *WR* command of a transaction to a bank is scheduled later than the *ACT* command to the same bank.

$$\hat{t}(RW_{j-1}^{BC_{i-1}-1}) \geq \hat{t}(ACT_{j-1}) + tRCD + (BC_{i-1} - 1) \times tCCD \tag{A.14}$$

Since *ACT* commands are scheduled in order by Algorithm 2, the previously scheduled command ACT_{j+l-4} ($l < 4$) was not scheduled later than that of ACT_{j-1}. We can get Eq. (A.15).

$$\forall l < 4 \Longrightarrow \hat{t}(ACT_{j-1}) \geq \hat{t}(ACT_{j+l-4}) \tag{A.15}$$

By combining Eq. (A.13), (A.14), and (A.15), Eq. (A.16) is derived.

$$\hat{t}(PRE_{j-1}) \geq \hat{t}(ACT_{j+l-4}) + tRCD + (BC_{i-1} - 1) \times tCCD + tRWTP \tag{A.16}$$

We now proceed by obtaining Eq. (A.17) based on the combination of Eq. (A.10), (A.11), and (A.16). Moreover, we can observe $tFAW \leq tRC = tRAS + tRP \leq tRCD + tRWTP + tRP$ for all DDR3 devices from the JEDEC DDR3 timing constraints [53]. Therefore, $\hat{t}(ACT_{j+l-1}) > \hat{t}(ACT_{j+l-4}) + tFAW$ according to Eq. (A.17), proving the second relation.

$$\hat{t}(ACT_{j+l-1}) > \hat{t}(ACT_{j+l-4}) + tRCD + (BC_{i-1} - 1) \times tCCD + tRWTP + tRP \tag{A.17}$$

With the above two reasons, the simplified equation is given by Eq. (A.9). It indicates the scheduling time of ACT_{j+l} is only determined by that of the previous ACT_{j+l-1}. Based on Eq. (A.9) and $\forall l \in (b_{com}, BI_i - 1]$, we can get Eq. (A.18), which shows the scheduling time of ACT_{j+l} depends on that of $ACT_{j+b_{com}}$. Note that $ACT_{j+b_{com}}$ was scheduled to the last bank b_{j-1} of T_{i-1}.

$$\hat{t}(ACT_{j+l}) = \hat{t}(ACT_{j+b_{com}}) + [l - b_{com}] \times tRRD + \sum_{l'=b_{com}+1}^{l} C(j + l') \tag{A.18}$$

□

A.3 PROOF OF LEMMA 3

Proof. To prove the lemma, we have to separate the problem into two pieces by analyzing the scheduling of commands for T_i to common banks with T_{i-1} and to the non-common banks, respectively. Since Lemma 2 implies the scheduling of *ACT* commands to non-common banks is only determined by the scheduling of the *ACT* command for T_i to the last common bank, we only need to prove the first piece that the scheduling of *ACT* commands to common banks is only dependent on T_{i-1} in the worst case. The common banks have been accessed by T_{i-1}, resulting in worst-case initial bank state for T_i because of the timing dependencies. Moreover, when $BI_{i-1} < 4$, the scheduling of an *ACT* command for T_i may be determined by the *ACT* commands of earlier transactions, e.g., T_{i-2} or T_{i-3}, through the *tFAW* timing constraint. We hence only need to prove that these earlier *ACT* commands cannot dominate in the initial bank state given by the *ALAP* command scheduling of T_{i-1}. Note that $BI_{i-1} \geq 4$ ensures that there were at least four *ACT* commands for T_{i-1}. As a result, the command scheduling of T_i is only dependent on that of T_{i-1} when $BI_{i-1} \geq 4$. So, the following only considers $BI_{i-1} < 4$.

We proceed by proving that the scheduling of *ACT* commands for T_i to common banks is only dependent on T_{i-1}. For a common bank $b_j + l$ between T_{i-1} and T_i where $\forall l \in [0, b_{com}]$, the scheduling time of its *ACT* command ACT_{j+l} is obtained from Eq. (4.1) and is shown in Eq. (A.19), which indicates that $\hat{t}(ACT_{j+l})$ depends on the scheduling time $\hat{t}(ACT_{j+l-1})$ of the previous *ACT*, the scheduling time $\hat{t}(PRE_{j-1-(b_{com}-l)})$ of the latest *PRE* to bank $b_j + l$ and the scheduling time $\hat{t}(ACT_{j+l-4})$ of the fourth previous *ACT* command (due to *tFAW*). Note that $j - 1 - (b_{com} - l)$ is the latest access number to bank $b_j + l$ according to the *ALAP* command scheduling of T_{i-1}. For example, if $l = b_{com}$, bank

$b_j + l = b_j + b_{com}$ is the last common bank between T_{i-1} and T_i, where its latest access number is $j-1$. Since ACT_{j+l-1} is a command for T_i or T_{i-1} ($l = 0$) while $PRE_{j-1-(b_{com}-l)}$ was for T_{i-1}, only ACT_{j+l-4} is possible to be a command for earlier transactions, e.g., T_{i-2} or T_{i-3}, if $BI_{i-1} < 4$. To prove the lemma, we only need to prove $\hat{t}(ACT_{j+l-4})$ does not dominate in the max{} of Eq. (A.19), which is then further simplified as shown in Eq. (A.19).

$$\begin{aligned}\hat{t}(ACT_{j+l}) =& max\{\hat{t}(ACT_{j+l-1}) + tRRD, \hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP,\\ & \hat{t}(ACT_{j+l-4}) + tFAW\} + C(j+l)\\ =& max\{\hat{t}(ACT_{j+l-1}) + tRRD, \hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP\}\\ & + C(j+l)\end{aligned} \tag{A.19}$$

For $\forall l < 4 - BI_{i-1}$, ACT_{j+l-4} is a command for earlier transactions, e.g., T_{i-2} or T_{i-3} when $BI_{i-1} < 4$. We proceed by computing the scheduling time of ACT_{j+l-4} based on the *ALAP* scheduling time of *ACT* commands for T_{i-1}, which are given by Eq. (4.7). In particular, the possible maximum scheduling time of the first *ACT* command of T_{i-1} is obtained by using Eq. (4.7), which is shown in Eq. (A.20).

$$\begin{aligned}\hat{t}(ACT_{j-1-(BI_{i-1}-1)}) =& \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD\\ & - (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\}\end{aligned} \tag{A.20}$$

By conservatively using the minimum time interval *tRRD* between two successive *ACT* commands, Eq. (A.21) is derived, which provides the possible maximum scheduling time of ACT_{j+l-4}. Moreover, by substituting Eq. (A.20) into Eq. (A.21), an explicit expression of $\hat{t}(ACT_{j+l-4})$ is obtained.

$$\begin{aligned}\hat{t}(ACT_{j+l-4}) =& \hat{t}(ACT_{j-1-(BI_{i-1}-1)}) - (4 - l - BI_{i-1}) \times tRRD\\ =& \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD\\ & - (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\}\\ & - (4 - l - BI_{i-1}) \times tRRD\end{aligned} \tag{A.21}$$

In order to prove that $\hat{t}(ACT_{j+l-4}) + tFAW$ cannot dominate in the max{} of Eq. (A.19), we only need to prove that $\hat{t}(ACT_{j+l-4}) + tFAW \leq \hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP$. Since $\hat{t}(PRE_{j-1-(b_{com}-l)})$ is given by Eq. (4.8) with assumption that T_{i-1} is write while $\hat{t}(ACT_{j+l-4})$ is provided by Eq. (A.21), Eq. (A.22) is derived.

$$\begin{aligned}&\hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP - \left[\hat{t}(ACT_{j+l-4}) + tFAW\right]\\ &= \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD + tRP\\ &\quad - \left[\hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD - (BI_{i-1} - 1)\right.\\ &\quad \left.\times max\{tRRD, BC_{i-1} \times tCCD\} - (4 - l - BI_{i-1}) \times tRRD + tFAW\right]\\ &= tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD + tRP + tRCD\\ &\quad + (BC_{i-1} - 1) \times tCCD + (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\}\\ &\quad + (4 - l - BI_{i-1}) \times tRRD - tFAW\end{aligned} \tag{A.22}$$

The result of this equation is non-negative, as the positive terms in Eq. (A.22) cancel out all the negative ones for the following four reasons: *1)* $max\{tRRD, BC_{i-1} \times tCCD\} \geq BC_{i-1} \times tCCD$. *2)* $b_{com} - l \leq BI_{i-1} - 1$ since $\forall l \in [0, b_{com}]$ and $b_{com} = \min\{BI_{i-1}, BI_i\} - 1$. *3)* the observation from JEDEC DDR3 timing constraints that $tFAW \leq tRWTP + tRP + tRCD$ for all DDR3 memories with write transaction. *4)* $l < 4 - BI_{i-1}$ from the above discussion. Therefore, $\hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP \geq \hat{t}(ACT_{j+l-4}) + tFAW$, which indicates $t(ACT_{j+l-4})$ cannot dominate in the max{} of Eq. (A.19). These earlier *ACT* commands (ACT_{j+l-4}) hence cannot dominate in the scheduling of the *ACT* commands for T_i because of *tFAW* in the worst case. Thus, Eq. (A.19) guarantees that the scheduling of *ACT* commands for T_i only depends on the maximum possible scheduling time of the previous *PRE* for T_{i-1} in the worst case or ACT_{j+l-1} that belongs to T_{i-1} for l=0. We can conclude that the *ALAP* command scheduling of the previous write transaction T_{i-1} is sufficient to give worst-case initial bank state to T_i. □

A.4 PROOF OF LEMMA 4

Proof. According to Lemma 1, the finishing time of a transaction T_i is determined by the finishing time of the previous transaction T_{i-1} and the scheduling time of all its *ACT* commands. Therefore, the worst-case finishing time of T_i is obtained by using the worst-case scheduling time (maximum) of its *ACT* commands, and the maximum finishing time of T_{i-1} that is $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$ based on Eq. (4.5), where we fix the worst-case starting time $\hat{t}_s(T_i)$ of T_i.

We proceed by obtaining the worst-case scheduling time of the *ACT* commands for T_i. Without loss of generality, T_i has BI_i and BC_i while T_{i-1} uses BI_{i-1} and BC_{i-1}. The current bank access number is j, and the starting bank of T_i is b_j, while the finishing bank of T_{i-1} is b_{j-1}. This results in $b_{com} = b_{j-1} - b_j$. For $\forall l \in [0, BI_i - 1]$, the worst-case scheduling time of the *ACT* command to bank $b_j + l$ is denoted by $\hat{t}(ACT_{j+l})$. It can be computed with two cases that $l \in [0, b_{com}]$ and $l \in (b_{com}, BI_i - 1]$, respectively.

For $\forall l \in [0, b_{com}]$, the ACT_{j+l} command is scheduled to bank $b_j + l$ that is a common bank between T_i and T_{i-1}. Lemma 3 guarantees that the *ALAP* scheduling of commands for the write transaction T_{i-1} is sufficient to provide the worst-case initial bank state for T_i. As a result, the worst-case scheduling time $\hat{t}(ACT_{j+l})$ can be obtained based on this worst-case initial states. Eq. (A.19) is hence used to compute $\hat{t}(ACT_{j+l})$, which indicates that the scheduling time of ACT_{j+l} is either determined by its previous ACT_{j+l-1} or the latest precharge, $PRE_{j-1-(b_{com}-l)}$, to the same bank. By iteratively using Eq. (A.19) to obtain the scheduling time of each ACT command to the common banks, we can derive a new expression of the scheduling time of ACT_{j+l} that is given by Eq. (A.23). Note

that $\forall l' \in [0, l]$ indexes a bank $(b_j + l')$ that is not accessed later than bank $b_j + l$, since $b_j + l' \leq b_j + l$.

$$\begin{aligned} t(ACT_{j+l}) = \underset{0 \leq \forall l' \leq l}{\text{Max}} \{ & t(ACT_{j-1}) + (l+1) \times tRRD + \sum_{h=0}^{l} C(j+h), \\ & t(PRE_{j-1-(b_{com}-l')}) + tRP + (l-l') \times tRRD + \sum_{h=l'}^{l} C(j+h) \} \end{aligned} \tag{A.23}$$

$t(ACT_{j-1})$ in Eq. (A.23) is the scheduling time of the last ACT command for T_{i-1}. According to $ALAP$ scheduling, the worst-case scheduling time $\hat{t}(ACT_{j-1})$ can be derived based on Eq. (4.7), and it is given by Eq. (A.24).

$$\hat{t}(ACT_{j-1}) = \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \tag{A.24}$$

Moreover, the worst-case scheduling time of $PRE_{j-1-(b_{com}-l')}$ to the common bank $b_j + l'$ based on $ALAP$ scheduling is given by Eq. (4.8). Therefore, by substituting $t(ACT_{j-1})$ and $t(PRE_{j-1-(b_{com}-l')})$ in Eq. (A.23) with their worst-case scheduling time given by Eq. (A.24) and Eq. (4.8), we can obtain the worst-case scheduling time of ACT_{j+l}, as shown in Eq. (A.25).

$$\begin{aligned} \hat{t}(ACT_{j+l}) = \underset{0 \leq l' \leq l \leq b_{com}}{\text{Max}} \{ & \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ & + (l+1) \times tRRD + \sum_{h=0}^{l} C(j+h), \\ & \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l') \times BC_{i-1} \times tCCD \\ & + tRP + (l-l') \times tRRD + \sum_{h=l'}^{l} C(j+h) \} \end{aligned} \tag{A.25}$$

For $\forall l \in (b_{com}, BI_i - 1]$, the ACT_{j+l} command is scheduled to the non-common bank $b_j + l$. Since Lemma 2 ensures that the scheduling time of an ACT command to a non-common bank is only determined by that of the ACT command to the last common bank, Eq. (A.18) is used to compute $t(ACT_{j+l})$, and it is only dependent on $t(ACT_{j+b_{com}})$. Eq. (A.25) is used to compute the worst-case scheduling time $t(ACT_{j+b_{com}})$, which is further substituted into Eq. (A.18). Hence, $\hat{t}(ACT_{j+l})$ is also derived when $\forall l \in (b_{com}, BI_i - 1]$.

Finally, we can use $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$ and $\hat{t}(ACT_{j+l})$ to derive the worst-case finishing time $\hat{t}_f(T_i)$ of T_i based on Lemma 1 (described by Eq. (A.8)). It is described by Eq. (A.26), where $\forall l' [0, b_{com}]$ and $\forall l \in [l', BI_i - 1]$. Intuitively, Eq. (A.26) illustrates that

the worst-case finishing time of a transaction is dependent on the precharging time of the common banks with the previous write transaction.

$$\begin{aligned}\hat{t}_f(T_i) = \max_{0\le l'\le b_{com}, l'\le l\le BI_i-1}\{&\hat{t}_s(T_i) - 1 - (BC_{i-1}-1)\times tCCD\\ &+ (l+1)\times tRRD + \left[(BI_i - l)\times BC_i - 1\right]\times tCCD + \sum_{h=0}^{l} C(j+h),\\ &\hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l')\times BC_{i-1}\times tCCD + tRP + tRCD\\ &+ (l-l')\times tRRD + \left[(BI_i - l)\times BC_i - 1\right]\times tCCD + \sum_{h=l'}^{l} C(j+h),\\ &\hat{t}_s(T_i) - 1 + tSwitch + (BI_i\times BC_i - 1)\times tCCD\}\end{aligned} \tag{A.26}$$

□

A.5 PROOF OF THEOREM 1

Proof. Since Lemma 4 provides the worst-case finishing time of a transaction T_i, we can hence compute the worst-case execution time (WCET) according to Definition 7. Then we only need to simplify the expressions in the equation and obtain the WCET.

Lemma 4 indicates that the worst-case finishing time $\hat{t}_f(T_i)$ depends on its worst-case starting time $\hat{t}_s(T_i)$, the *BI* and *BC* used by T_{i-1} and T_i, and the JEDEC DDR3 timing constraints. According to Definition 7, the WCET is the time between $\hat{t}_s(T_i)$ and $\hat{t}_f(T_i)$, and is given by Eq. (A.27).

$$\hat{t}_{ET}(T_i) = \hat{t}_f(T_i) - \hat{t}_s(T_i) + 1 \tag{A.27}$$

Based on Eq. (A.26) that gives the worst-case finishing time, we further obtain Eq. (A.28) according to Eq. (A.27) by moving $\hat{t}_s(T_i) - 1$ from the right side to the left of Eq. (A.26).

$$\begin{aligned}\hat{t}_{ET}(T_i) = \max_{0\le l'\le b_{com}, l'\le l\le BI_i-1}\{&-(BC_{i-1}-1)\times tCCD\\ &+ (l+1)\times tRRD + \left[(BI_i - l)\times BC_i - 1\right]\times tCCD + \sum_{h=0}^{l} C(j+h),\\ &tRWTP - (b_{com} - l')\times BC_{i-1}\times tCCD + tRP + tRCD\\ &+ (l-l')\times tRRD + \left[(BI_i - l)\times BC_i - 1\right]\times tCCD + \sum_{h=l'}^{l} C(j+h),\\ &tSwitch + (BI_i\times BC_i - 1)\times tCCD\}\end{aligned} \tag{A.28}$$

Since we conservatively assume there is always scheduling collisions for *ACT* commands, i.e., $C(j+h) = 1$, Eq. (A.28) can be simplified based on $\sum_{h=0}^{l} C(j+h) = l+1$ and $\sum_{h=l'}^{l} C(j+h) = l - l' + 1$, as shown in Eq. (A.29).

$$\begin{aligned}
\hat{t}_{ET}(T_i) = \max_{0 \le l' \le b_{com}, l' \le l \le BI_i - 1} \{ & (BI_i \times BC_i - BC_{i-1}) \times tCCD \\
& + l \times (tRRD + 1 - BC_i \times tCCD) + tRRD + 1, \\
& tRWTP + tRP + tRCD + \left[BI_i \times BC_i - 1 - b_{com} \times BC_{i-1}\right] \times tCCD + 1 \\
& + l' \times (BC_{i-1} \times tCCD - tRRD - 1) + l \times (tRRD + 1 - BC_i \times tCCD), \\
& tSwitch + (BI_i \times BC_i - 1) \times tCCD \}
\end{aligned} \tag{A.29}$$

We can observe from Eq. (A.29) that the expressions in the max{} function either linearly increase or decrease with l and l'. Therefore, Eq. (A.29) can be further simplified to obtain $\hat{t}_{ET}(T_i)$ by using both the maximum and minimum values of l and l' in the max{} of Eq. (A.29). Since $\forall l' [0, b_{com}]$ and $\forall l \in [l', BI_i - 1]$, we substitute (l', l) with $(0, 0)$, $(0, BI_i - 1)$, (b_{com}, b_{com}), and $(b_{com}, BI_i - 1)$ in all the terms of the max{} in Eq. (A.29). $\hat{t}_{ET}(T_i)$ is further given by Eq. (A.30). Note that some of the terms are removed, since they cannot dominate in the max{} when deriving Eq. (A.30).

$$\begin{aligned}
\hat{t}_{ET}(T_i) = \max\{ & (BI_i \times BC_i - BC_{i-1}) \times tCCD \\
& + (BI_i - 1) \times (tRRD + 1 - BC_i \times tCCD) + tRRD + 1, \\
& tRWTP + tRP + tRCD + \left[BI_i \times BC_i - 1 - b_{com} \times BC_{i-1}\right] \times tCCD + 1, \\
& tRWTP + tRP + tRCD + \left[(BI_i - b_{com}) \times BC_i - 1\right] \times tCCD + 1, \\
& tRWTP + tRP + tRCD + \left[BC_i - 1 - b_{com} \times BC_{i-1}\right] \times tCCD \\
& + (BI_i - 1) \times (tRRD + 1) + 1, \\
& tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD \\
& + \left[BI_i - 1 - b_{com}\right] \times (tRRD + 1) + 1, \\
& tSwitch + (BI_i \times BC_i - 1) \times tCCD \}
\end{aligned} \tag{A.30}$$

Note that $b_{com} = b_{j-1} - b_j$ and is determined by the size of T_{i-1} and T_i, whichever is smaller, i.e., $b_{com} = \min\{BI_{i-1}, BI_i\} - 1$. Moreover, some of the expressions in the max{} of Eq. (A.30) are further simplified according to the observation from JEDEC DDR3

timing constraints that $tSwitch > tRRD + 1$ when T_{i-1} is a write. The simplified $\hat{t}_{ET}(T_i)$ is finally shown in Eq. (A.31).

$$\begin{aligned}\hat{t}_{ET}(T_i) = \max\{&(BC_i - BC_{i-1}) \times tCCD + BI_i \times (tRRD + 1),\\ &tRWTP + tRP + tRCD + 1\\ &+ \left[BI_i \times BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}\right] \times tCCD,\\ &tRWTP + tRP + tRCD + 1\\ &+ \left[(BI_i - (\min\{BI_{i-1}, BI_i\} - 1)) \times BC_i - 1\right] \times tCCD,\\ &tRWTP + tRP + tRCD + (BI_i - 1) \times (tRRD + 1) + 1\\ &+ \left[BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}\right] \times tCCD,\\ &tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD\\ &+ \left[BI_i - \min\{BI_{i-1}, BI_i\}\right] \times (tRRD + 1) + 1,\\ &tSwitch + (BI_i \times BC_i - 1) \times tCCD\}\end{aligned} \tag{A.31}$$

□

A.6 PROOF OF THEOREM 2

Proof. To prove the WCET of a transaction provided by Theorem 1 monotonically increases with its size, it is only necessary to prove that the WCET monotonically increases with its *BI* and *BC*. The WCET of T_i is given by Theorem 1 (i.e., Eq. (A.31)). We can see that the WCET, $\hat{t}_{ET}(T_i)$, is determined by one of the 6 expressions in the max{} function, which are all also functions of BI_i and BC_i. These 6 expressions are denoted by *expr1* to *expr6*, respectively, corresponding to the expressions from top to bottom in Eq. (A.31). We proceed by proving the monotonicity for each of them.

Expression 1:

Since $expr1(BI_i, BC_i) = (BC_i - BC_{i-1}) \times tCCD + BI_i \times (tRRD + 1)$, $BI'_i \leq BI_i \wedge BC'_i \leq BC_i \implies expr1(BI_i, BC_i) \geq expr1(BI'_i, BC'_i)$.

Expression 2:

$expr2(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + \left[BI_i \times BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}\right] \times tCCD$.

Case 1: $BI_i \leq BI_{i-1}$,

$BI_i \leq BI_{i-1} \implies expr2(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + \left[BI_i \times (BC_i - BC_{i-1}) + BC_{i-1} - 1\right] \times tCCD$.

We can observe that $expr5 > tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD + 1$ in this case for any BI_i and BC_i. As a result, *expr2* can dominate the max{} function of Eq. (A.31) only if the given BI_i and BC_i cannot make it smaller than $tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD + 1$. Therefore, $BC_i \geq BC_{i-1}$ is a necessary condition for *expr2*. On this condition, we can derive $expr2(BI_i, BC_i) \geq expr2(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

Case 2: $BI_i > BI_{i-1}$,

$BI_i > BI_{i-1} \implies \min\{BI_{i-1}, BI_i\} = BI_{i-1} \implies expr2(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + \left[BI_i \times BC_i - 1 - (BI_{i-1} - 1) \times BC_{i-1}\right] \times tCCD$.

For this expression, it follows that $expr2(BI_i, BC_i) \geq expr2(BI'_i, BC'_i)$ if $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$ in this case.

With these two cases, when *expr2* dominates the max{} function of Eq. (A.31), there is $expr2(BI_i, BC_i) \geq expr2(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

Expression 3:

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + \left[(BI_i - (\min\{BI_{i-1}, BI_i\} - 1)) \times BC_i - 1\right] \times tCCD$. For this expression, there are again two cases, where the theorem follows straightforwardly for both of them.

Case 1: $BI_i \leq BI_{i-1}$,

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + (BC_i - 1) \times tCCD$. As a result, $BI'_i \leq BI_i \wedge BC'_i \leq BC_i \implies expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$.

Case 2: $BI_i > BI_{i-1}$,

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + \left[(BI_i - BI_{i-1} + 1) \times BC_i - 1\right] \times tCCD$. So, $BI'_i \leq BI_i \wedge BC'_i \leq BC_i \implies expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$.

According to these two cases, there is $expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

Expression 4, 5, and 6:

With a similar discussion as for **Expression 2**, we can conclude that *expr4* monotonically increases with BI_i and BC_i. This conclusion also holds for *expr5* if it is analyzed in the same way as **Expression 3**, while *expr6* can be discussed similarly to **Expression 1**. The detailed derivation is not shown here for brevity. □

A.7 PROOF OF LEMMA 5

Proof. For a given *DataSize*, there is $DataSize = \sum_{\forall T \in \overline{T}} S(T)$ and its maximum execution time is $\text{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)$, which can be obtained by verifying the bound of the clock WCRT with the observer TA shown in Figure 6.5(b). For $\forall N > 1$, the larger data size $DataSize' = N \times DataSize = \sum_{\forall T \in \overline{T'}} S(T)$ corresponding to trace $\overline{T'}$, and its maximum execution time is $\text{Max}_{\forall \overline{T'}} \sum_{\forall T \in \overline{T'}} t_{ET}(T)$. Conservatively, we obtain Eq. (A.32), since the transaction trace $\overline{T'}$ generates N times more data than the trace $\overline{T}$. Therefore, Eq. (A.33) is derived, which shows that better WCBW can be obtained when verifying with larger data size. Intuitively, larger amount of data is generated by more transactions, where more pipelining between transactions can be exploited to achieve better WCBW. When N approaches $+\infty$, we get the long-term WCBW, which is given by Eq. (A.34). Accord-

ing to Eq. (A.33), we can conclude that the $\hat{bw}(DataSize)$ of any given *DataSize* is a conservative lower bound for the long-term *WCBW*.

$$\underset{\forall \overline{T'}}{\text{Max}} \sum_{\forall T \in \overline{T'}} t_{ET}(T) \leq N \times \underset{\forall \overline{T}}{\text{Max}} \sum_{\forall T \in \overline{T}} t_{ET}(T) \tag{A.32}$$

$$\begin{aligned} \hat{bw}(N \times DataSize) &= \frac{N \times DataSize}{\text{Max}_{\forall \overline{T'}} \sum_{\forall T \in \overline{T'}} t_{ET}(T)} \\ &\geq \frac{N \times DataSize}{N \times \text{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)} \\ &\geq \hat{bw}(DataSize) \end{aligned} \tag{A.33}$$

$$\begin{aligned} \hat{bw} &= \lim_{N \to +\infty} \hat{bw}(N \times DataSize) \\ &\geq \hat{bw}(DataSize) \end{aligned} \tag{A.34}$$

□

B

SYSTEM DECLARATIONS FOR TIMED AUTOMATA MODEL

B.1 INTUITIVE TIMED AUTOMATA MODEL

// Place template instantiations here.
Src = *Source()*;

ACTScheduler := *ActScheduler()*;

RAS0 = *RAS(0)*;
RAS1 = *RAS(1)*;
RAS2 = *RAS(2)*;
RAS3 = *RAS(3)*;
RAS4 = *RAS(4)*;
RAS5 = *RAS(5)*;
RAS6 = *RAS(6)*;
RAS7 = *RAS(7)*;

FAW0 = *FAW(0)*;
FAW1 = *FAW(1)*;
FAW2 = *FAW(2)*;
FAW3 = *FAW(3)*;

RRD0 = *RRD()*;

RCD0 := *RCD(0)*;
RCD1 := *RCD(1)*;
RCD2 := *RCD(2)*;
RCD3 := *RCD(3)*;
RCD4 := *RCD(4)*;
RCD5 := *RCD(5)*;
RCD6 := *RCD(6)*;
RCD7 := *RCD(7)*;

RWScheduler := *RwScheduler()*;

PRE0 := *PRE(0)*;
PRE1 := *PRE(1)*;
PRE2 := *PRE(2)*;
PRE3 := *PRE(3)*;
PRE4 := *PRE(4)*;
PRE5 := *PRE(5)*;
PRE6 := *PRE(6)*;
PRE7 := *PRE(7)*;

RWTP0 := *RWTP(0)*;
RWTP1 := *RWTP(1)*;
RWTP2 := *RWTP(2)*;
RWTP3 := *RWTP(3)*;
RWTP4 := *RWTP(4)*;
RWTP5 := *RWTP(5)*;
RWTP6 := *RWTP(6)*;
RWTP7 := *RWTP(7)*;

CCD0 := *CCD()*;
RWCounter := *SWCounter()*;

CMDBUS := *CmdBus()*;
TDMBUS := *TDM()*;
MeMAP = *MemMap()*;

//List one or more processes to be composed into a system.
system
Src, TDMBUS, MeMAP, ACTScheduler, RWScheduler, CMDBUS, RWCounter,
RCD0, RCD1, RCD2, RCD3, RCD4, RCD5, RCD6, RCD7,
RAS0, RAS1, RAS2, RAS3, RAS4, RAS5, RAS6, RAS7,
PRE0, PRE1, PRE2, PRE3, PRE4, PRE5, PRE6, PRE7,
RWTP0, RWTP1, RWTP2, RWTP3, RWTP4, RWTP5, RWTP6, RWTP7,
FAW0, FAW1, FAW2, FAW3, RRD0, CCD0;

B.2 SIMPLIFIED TIMED AUTOMATA MODEL

// Place template instantiations here.
Src = *Source()*;

ACTScheduler := *ActScheduler()*;

FAW0 = *FAW(0)*;
FAW1 = *FAW(1)*;
FAW2 = *FAW(2)*;
FAW3 = *FAW(3)*;

RRD0 = *RRD()*;

RCD0 := *RCD(0)*;
RCD1 := *RCD(1)*;

RWScheduler := *RwScheduler()*;

PRE0 := *PREScheduler(0)*;
PRE1 := *PREScheduler(1)*;
PRE2 := *PREScheduler(2)*;
PRE3 := *PREScheduler(3)*;
PRE4 := *PREScheduler(4)*;
PRE5 := *PREScheduler(5)*;
PRE6 := *PREScheduler(6)*;
PRE7 := *PREScheduler(7)*;

CCD0 := *CCD()*;
RWCounter := *SWCounter()*;

CMDBUS := *CmdBus()*;
TDMBUS := *TDM()*;
MeMAP := *MemMap()*;

//List one or more processes to be composed into a system.
system
Src, TDMBUS, MeMAP, ACTScheduler, RWScheduler,
CMDBUS, RWCounter, RCD0, RCD1,
PRE0, PRE1, PRE2, PRE3, PRE4, PRE5, PRE6, PRE7,
FAW0, FAW1, FAW2, FAW3, RRD0, CCD0;

C

SCALABILITY OF MODE-CONTROLLED DATAFLOW AND TIMED AUTOMATA

A memory controller can be configured to support a different number of requestors with a fixed transaction size or variable sizes, respectively. In our experiments, the fixed transaction size includes 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes. When mixing some of these sizes, the cases of variable sizes are obtained. Table C.1 provides 8 cases when the memory controller serves requestors with variable sizes. In particular, the requestors can be served by a TDM arbiter that specifies a static order, as described by the arrows in Table C.1. When the arbiter is unknown, requestors can be served in any order. For example, when two requestors Req_0 and Req_1 are served by an unknown arbiter, the order is denoted as "Req_0 | Req_1". In this case, we can only analyze the WCBW, while the WCRT of a transaction cannot be analyzed because the number of the preceding transactions is unpredictable, as indicated with "n/a" in Table C.2.

Experiments are carried out to analyze the worst-case bandwidth (WCBW) and/or worst-case response time (WCRT) using the mode-controlled dataflow (MCDF) model and the timed automata (TA) model, respectively. To fairly compare the time and RAM used by the MCDF model and the TA model, the same server is used. The server consists of 125 GB usable RAM and 24 Intel Xeon(R) CPUs running at 2.1 GHz, and it uses a CentOS 6.8 system. Table C.2 presents the time and RAM consumed by analyzing the MCDF model with Heracles and verifying properties of the TA model using Uppaal, respectively. Moreover, it also provides the worst-case results if the analysis or verification is successful. Otherwise, it indicates "failed" and also shows the time and RAM usage when the analysis or verification fails.

Table C.1: Different configurations for Run-DMC with variable sizes.

Configuration	*Requestor: transaction size (bytes)*	*Arbitration & Service Order*
Case 1	Req_0: 128, Req_1: 128, Req_2: 128, Req_3: 64, Req_4: 64	Unknown; Req_0 | Req_1 | Req_2 | Req_3 | Req_4
Case 2	Req_0: 128, Req_1: 128, Req_2: 128, Req_3: 64, Req_4: 64	TDM; Req_0 → Req_1 → Req_2 → Req_3 → Req_4
Case 3	Req_0: 256, Req_1: 128, Req_2: 64, Req_3: 32, Req_4: 16	Unknown; Req_0 | Req_1 | Req_2 | Req_3 | Req_4
Case 4	Req_0: 256, Req_1: 128, Req_2: 64, Req_3: 32, Req_4: 16	TDM; Req_0 → Req_1 → Req_2 → Req_3 → Req_4
Case 5	Req_0: 256, Req_1: 128, Req_2: 64, Req_3: 32	Unknown; Req_0 | Req_1 | Req_2 | Req_3
Case 6	Req_0: 256, Req_1: 128, Req_2: 64, Req_3: 32	TDM; Req_0 → Req_1 → Req_2 → Req_3
Case 7	Req_1: 128, Req_2: 64	Unknown; Req_0 | Req_1
Case 8	Req_1: 128, Req_2: 64	TDM; Req_0 → Req_1

Table C.2: WCBW (MB/s) and WCRT (cycles) of different DDR3 SDRAMs with fixed transaction size.

		MCDF with Heracles				TA with Uppaal						
	Size (bytes)	Starting Bank	WCBW (MB/s)	Time (s)	RAM (MB)	Starting Bank	WCBW (MB/s)	Time (s)	RAM (MB)	WCRT (cycles)	Time (s)	RAM (MB)
Fixed Size	16	0, 1, 2, 3, 4, 5, 6, 7	312	5376	29	0, 1, 2	320	12470	30499	160	4419	13631
						0, 1, 2, 3	failed	72563	121911	failed	63398	118358
	32	0, 2, 4, 6	624	2	26	0, 2, 4, 6	638	8571	14981	161	1983	5447
	64	0, 4	1249	2	26	0, 4	1280	215	722.0	160	76	1668
	128	0, 4	2276	3	25	0, 4	2295	2128	6007	179	616	2505
	256	0, 4	2844	6	33	0, 4	2844	130	658	288	36	289
Variable Size	Case 1	0, 4	1138	80	104	0, 4	failed	70870	125132	n/a		
	Case 2	0, 4	failed	65	6192	0, 4	1276	3850	6692	227	4024	6693
	Case 3	0, 1, 2, 3, 4, 5, 6, 7	178	129	156	0, 1, 2, 3, 4, 5, 6, 7	failed	26647	117537	n/a		
	Case 4	0, 1, 2, 3, 4, 5, 6, 7	failed	346	37350	0, 1, 2, 3, 4, 5, 6, 7	1503	17794	42438	264	17771	42440
	Case 5	0, 2, 4, 6	356	65	104	0, 2, 4, 6	failed	32656	117421	n/a		
	Case 6	0, 2, 4, 6	failed	24	3404	0, 2, 4, 6	1746	834	2969	220	851	2969
	Case 7	0, 4	1138	9	38	0, 4	1191	31769	71590	n/a		
	Case 8	0, 4	1786	160	153	0, 4	1804	756	1403	96	429	1118

D

LIST OF ACRONYMS

SDRAM	Synchronous Dynamic Random-Access Memory . . . 1
TDM	time-division multiplexing . . . viii
IoT	Internet of Things . . . 1
DMA	direct memory access . . . 1
QoS	Quality of Service . . . 2
SRAM	static random-access memory . . . 3
NoC	network-on-chip . . . 4
SoC	System-on-Chip . . . 3
DSP	digital signal processor . . . 3
RR	round-robin . . . 6
Run-DMC	a memory controller with dynamic command scheduling at run-time . . . 8
FCFS	first-come first-serve . . . 9
ALAP	as-late-as-possible . . . 10
MCDF	mode-controlled dataflow . . . 11
TA	timed automata . . . 12
DDR	double data rate . . . 13
ACT	Activate . . . 14
RD	read . . . 14
WR	write . . . 14
RW	a *RD* or *RD* command . . . 22
PRE	precharge . . . 14
REF	refresh . . . 14
NOP	no operation . . . 14
BL	burst length . . . 14
LPDDR	low-power double data rate . . . 14
WCET	worst-case execution time . . . 26
WCRT	worst-case response time . . . 26

WCBW	worst-case bandwidth . . . 26
CMD	command . . . 27
RAM	random-access memory . . . 45
DRAM	dynamic random-access memory . . . 32
FPGA	field-programmable gate array . . . 32
DIMM	dual in-line memory module . . . 33
GT	guaranteed throughput . . . 32
BE	best-effort . . . 33
FR-FCFS	First-Ready First-come First-Serve . . . 33
BC	Burst count . . . 18
BI	Bank interleaving . . . 18
BS	starting bank number . . . 21
RT	response time . . . 24
CCSP	credit-controlled static-priority arbitration . . . 7
FBSP	frame-based static priority . . . 7
TCC	timing constraint counters . . . 40
XML	extensible markup language . . . 43
Gb	gigabit . . . 15
GB	gigabyte . . . 45
SRDF	single rate dataflow . . . 94
MCM	maximum cycle mean . . . 95
SL	select . . . 96
SW	switch . . . 96
MC	model controller . . . 95
SMS	static mode sequence . . . 97

E

LIST OF SYMBOLS

GENERAL

i	the arrived number of a transaction	21
T_i	the i^{th} arrived transaction, where $\forall i \geq 0$	21
$S(T_i)$	size of transaction T_i	21
$Type(T_i)$	type of transaction T_i and is either read or write	21
BI_i	the BI used by T_i	21
BC_i	the BC used by T_i	21
$j(i)$	the current bank access number for T_i	22
j	the shorthand for $j(i)$	21
b_j	bank number of the j^{th} bank access. It is also the starting bank of T_i	21
ACT_j	ACT command for bank b_j	23
$t(ACT_j)$	the scheduling time of ACT_j	61
RW_j^k	k^{th} RD or WR command for bank b_j	23
$t(RW_j^k)$	the scheduling time of RW_j^k	61
PRE_j	PRE command for bank b_j	23
$t(PRE_j)$	the scheduling time of PRE_j	61
$RW_{j+BI_i-1}^{BC_i-1}$	the last RD or WR command of T_i	26
$t(RW_{j+BI_i-1}^{BC_i-1})$	the scheduling time of $RW_{j+BI_i-1}^{BC_i-1}$	26
$t_a^{fe}(T_i)$	arrival time of T_i in the front-end	26
$t_a(T_i)$	arrival time of T_i in the back-end	26
$t_s(T_i)$	starting time of transaction T_i in the back-end	26
$t_f(T_i)$	finishing time of transaction T_i in the back-end	26
$t_{ET}(T_i)$	execution time of an arbitrary transaction T_i in the back-end	26
$\hat{t}_{ET}(T_i)$	worst-case execution time of an arbitrary transaction T_i	26
$t_f^{fe}(T_i)$	finishing time of transaction T_i in the front-end	27
$t_{RT}(T_i)$	response time of transaction T_i in the front-end	27

F

ABOUT THE AUTHOR

Yonghui Li was born on July 21, 1986 in Shaanxi, China. He received a B.Sc. in Space Information and Digital Technology from Xidian University, Xi'an, China in 2009, and a M.Sc. in Communication and Information System from the same university in 2012. He carried out research on design and analysis of Network-on-Chip for manycore System-on-Chip with the State Key Laboratory of Integrated Services Networks in Xidian University from 2008 to 2012. In May 2012, he became a PhD candidate in the Electronic Systems Group at Eindhoven University of Technology, the Netherlands. His research interests include real-time systems, dataflow modeling, timed automata modeling, model checking, Network-on-Chip, and memory controller. He won a Best Paper Award from the 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2015.

LIST OF PUBLICATIONS

Articles Included in the Thesis:

[1] **Y. Li**, B. Akesson, and K. Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–14, 2014.

[2] **Y. Li**, B. Akesson, and K. Goossens. Architecture and Analysis of a Dynamically-Scheduled Real-Time Memory Controller. In *Real-Time Systems*, Volume 52, Issue 5, pages 675–729, 2016.

[3] **Y. Li**, H. Salunkhe, J. Bastos, O. Moreira, B. Akesson, and K. Goossens. Mode-Controlled Data-Flow Modeling of Real-Time Memory Controllers. In *13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10, 2015 (**Best Paper Award**).

[4] **Y. Li**, B. Akesson, K. Lampka, and K. Goossens. Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers. In *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

Open-Source Tools:

[5] **Y. Li**, B. Akesson, and K. Goossens. *RTMemController:* An Open-Source WCET and ACET Analysis Tool for Real-Time Memory Controllers. http://www.es.ele.tue.nl/rtmemcontroller/, 2014.

[6] **Y. Li**, B. Akesson, K. Lampka, and K. Goossens. Timed Automata Model of Real-Time Memory Controller with Dynamic Command Scheduling. http://www.es.ele.tue.nl/rtmemcontroller/TA.zip, 2016.

[7] K. Chandrasekar, C. Weis, **Y. Li**, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. http://www.es.ele.tue.nl/drampower/, 2012.

Posters:

[8] **Y. Li**, B. Akesson, and K. Goossens. Dynamic Command Scheduling for Real-Time Memory Controller. In *the ICT.OPEN 2012 Conference: The Interface for Dutch ICT-Research*, 2012 (Abstract & Poster).

[9] **Y. Li**, O. Moreira, B. Akesson, and K. Goossens. Dataflow Modeling of Real-Time Memory Controllers. In *the ICT.OPEN 2015 Conference: The Interface for Dutch ICT-Research*, 2015 (Abstract & Poster & Oral Presentation).

[10] **Y. Li**, B. Akesson, and K. Goossens. Design and Formal Analysis of Run-DMC, a Dynamically-Scheduled Real-Time Memory Controller. In *DATE PhD Forum*, 2016 (Abstract & Poster).

Other Co-authored Articles:

[11] K. Goossens, A. Azevedo, K. Chandrasekar, M.D. Gomony, S. Goossens, M. Koedam, **Y. Li**, D. Mirzoyan, A. Molnos, A. Beyranvand Nejad, A. Nelson, and S. Sinha. Virtual Execution Platforms for Mixed-Time-Criticality systems: The CompSOC Architecture and Design Flow. *SIGBED Rev.*, 10(3):23–34, 2013.

[12] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, **Y. Li**, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-Predictable Multi-Core Architecture for Embedded Systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.