T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 4.4 Dynamic Memory Controller Design and Implementation

Version 1.0
13 September 2013
Final

Public Distribution

Eindhoven University of Technology

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First draft	1 September 2013
0.2	Draft sent to T-CREST partners	9 September 2013
1.0	Final version	13 September 2013

Executive Summary

This document describes the deliverable *D 4.4 Dynamic Memory Controller Design and Implementation* of work package 4 of the T-CREST project, due 24 months after project start as stated in the Description of Work. First, the problem of increasingly dynamic memory request streams in firm real-time systems is explained. We then survey related work on real-time memory controllers and conclude that existing controllers are either too static, both in terms of implementation and performance analysis, to efficiently cope with dynamic memory streams (e.g. requests with variable sizes and alignments), or too dynamic and fail to provide firm bounds on bandwidth and response times.

To address this problem, we propose to use predictable dynamic command scheduling which is capable of dealing with transactions in various sizes. The three main contributions of this document are: 1) a back-end architecture for a real-time memory controller with a dynamic scheduling algorithm. The architecture is delivered as a fully functional SystemC implementation. The scheduling algorithm is furthermore integrated into the DRAMPower tool, which is an open-source tool for memory power estimation. 2) a formalization of the timing behavior of the proposed architecture and algorithm, which to the best of our knowledge is the most complete and flexible formalization of memory controller timing behavior to date, and 3) the analytical worst-case execution time for transactions with both fixed and variable sizes under different memory map configurations; Finally, we experimentally evaluate the proposed memory controller with various memory map configurations and compare the worst-case execution time of transactions to the semi-static command scheduling approach used as a baseline in the T-CREST project. The results demonstrate that dynamic command scheduling outperforms the baseline slightly in the worst-case situation, thus delivering on the goal of the design as it was stated in the concept document (D 4.3). In addition, it is more flexible when handling variable transaction sizes, improving the average-case performance.

1 Introduction

Designing embedded systems is getting increasingly complex as more and more applications, some of which have real-time requirements, are integrated. To provide the necessary computational power at reasonable power consumption, there is a trend towards multi-core systems where important functions are accelerated in hardware [5, 17, 24]. The diversity of applications and processing elements in such systems is reflected in the memory traffic going to shared DRAM, which features an irregular mix of transactions of different sizes that is difficult to characterize at design time. This makes it difficult to provide tight bounds on the response times of memory transactions, which is required to determine the worst-case execution times of applications mapped to the platform. Bounding response times of memory transactions is further complicated, since it varies depending on the *memory map configuration*, which provides different trade-offs between bandwidth, response times, and power consumption by varying the number of memory banks that are used in parallel to serve a transaction [11].

Most current DRAM controllers are not designed with real-time applications in mind and do not provide bounds on worst-case response times on transactions. On the other hand, existing real-time memory controllers with static [4] or semi-static [1, 21] command scheduling bound response times by generating and analyzing DRAM command schedules or sub-schedules at design time. The main drawbacks of these approaches are that they only support a single request size and memory map configuration and that they rely on design-time characterizations of the memory traffic, which may be difficult to obtain in the considered systems.

Dynamic command scheduling is a promising way to increase the flexibility of real-time memory controllers, as it is not necessary to know the exact traffic characterization at design time, and transactions are executed dynamically at run time. Existing work [8, 19, 23] on real-time dynamic command scheduling provide bounded response times. However, despite the flexibility of the memory controllers themselves, the provided analyses are limited to a single fixed request size and memory controller configuration.

This document addresses this issue by considering dynamic command scheduling of DDR3 memories in real-time systems with variable transaction sizes and different memory map configurations. The three main contributions of this document are: 1) A back-end architecture of a real-time memory controller with a dynamic as-soon-as-possible (ASAP) command scheduling algorithm. It accepts transactions with variable sizes and supports different memory map configurations. This back-end can be used with existing real-time memory controller front-ends (transaction schedulers), as well as the reconfigurable memory controller front-end (D 4.2) or the memory NoC, both developed in the T-CREST project. 2) A formalization of the timing behavior of the proposed dynamic command scheduler that captures the scheduling dependencies within and between banks. 3) The analytical worst-case execution time for transactions with variable sizes under different memory map configurations is derived based on the proposed formalism. We experimentally evaluate the proposed architecture and the analysis with different memory map configurations and the results indicate that the worst-case execution time is tightly bounded and our dynamic command scheduling outperforms the existing semi-static approach that is used as a baseline in the T-CREST project [1].

The rest of this document is organized as follows. Section 2 describes the related work. The background of SDRAM memories and real-time memory controllers is given in Section 3. Section 4

presents the back-end architecture and the dynamic command scheduling algorithm. In Section 5, the dynamic command scheduling is formalized while Section 6 provides the worst-case execution time bound. Experimental results are presented in Section 7, before we discuss the relation between this work and the requirements on the memory subsystem in the T-CREST project (Section 8). Lastly, we draw conclusions in Section 9.

2 Related Work

Analyzing the impact of having a single shared memory on worst-case execution time of applications is receiving increasing attention in the real-time community, as multi-core systems are challenging the traditional processor-centric view on systems. Most of this work, focuses commercial-of-the-shelf systems and consider the system bus and the memory controller as a poorly documented black box, whose access time is typically represented by a constant value obtained by assumptions or using non-conservative measurements [9, 22, 26]. The work in this document is complimentary to this effort, as it focuses on the architecture and scheduling algorithm of an important part of that black box (the back-end) and provides results that are required to derive that constant value for different transaction sizes and memory map configurations.

Several types of memory controller designs have been proposed during the past decade. Static [4] or semi-static [1, 21] controller designs are used to achieve bounded execution time, as their command schedules can be obtained and analyzed at design time. In [4], an application-specific static command schedule is constructed using a local search method. Its main weakness is that it requires exact knowledge of the sequence of transactions both in terms of sizes and if they are reads or writes, which is not available in complex dynamic systems that concurrently execute multiple applications. A semi-static method is proposed in [1]. It generates static memory patterns, which are shorter sub-schedules of SDRAM commands, at design time and schedules them dynamically based on incoming transactions at run time. However, this solution cannot efficiently manage variable transaction sizes as the patterns are generated to read or write a fixed amount of data. Larger transactions are dealt with by issuing multiple patterns and smaller transactions by padding or masking, consuming unnecessary time and power. Another problem of both mentioned static and semi-static approaches is that the static (sub-)schedules have to be stored in hardware, which results in significant overhead if more (sub-)schedules are introduced to increase flexibility. Reineke et al. [21] present a semi-static predictable DRAM controller for the PRET architecture, which partitions banks or sets of banks into virtual private resources with independent repeatable actual timing behavior. This concept inherently cannot support the idea of using different command schedules for transactions with different sizes, since this would make response times of requests dependent on previous transactions to other virtual resources. No results are furthermore presented for different memory map configurations.

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 [12,13,16,18]. However, these controllers aim at maximizing average performance and do not provide any real-time guarantees, making them unsuitable for firm real-time systems. Paolieri, et al. [19, 20] proposes an analytical model based on timing constraints to bound the execution time of transactions under

dynamic command scheduling on a modified version of the DRAMSim memory simulator [25], although the modifications to the original scheduling algorithm are not specified. Furthermore, the analytical model is limited to a fixed transaction size and a single memory map configuration. In addition, they rely on simulation to detect command scheduling collisions, which makes it difficult to support dynamic traffic since all possible combinations of transaction types and sizes would have to be simulated to guarantee conservative results. This problem also applies to Shah et al. [23], where the worst-case execution time of transactions with fixed size is analyzed on an FPGA instance of a dynamically scheduled Altera SDRAM controller using an on-chip logic analyzer. In [8], the execution time is obtained on the basis of memory access patterns, which are defined according to command pairs. These command pairs are composed of any two commands including *Read*, *Write*, *Activate* and *Precharge*. Each pattern results in a busy time of a bank, resulting in bounded execution time of a transaction since it can be broken down into a set of bank accesses.

In short, current real-time memory controllers fail to efficiently address the dynamic memory traffic in complex heterogeneous systems because of the limitations in architecture, or in the provided analysis with respect to variable transaction sizes and memory map configurations, or in both. In contrast, this paper presents both an architecture of a dynamically scheduled back-end and a corresponding analysis that supports different transaction sizes and memory map configurations. This requires a more elaborate analysis, since different timing constraints become bottlenecks for different transaction sizes and configurations, requiring more of them to be included in the model. Compared to earlier work, our analysis is supported by an elaborate formal framework in which the correctness of the results are proven. To the best of our knowledge, this makes it the most complete and flexible formalization of memory controller timing behavior to date.

3 Background

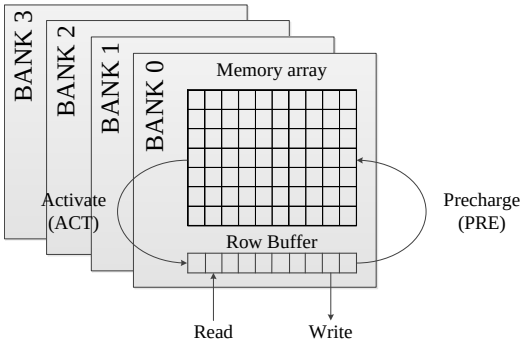
This section presents the required background information to understand the contents of this document. First, the architecture and basic operations of SDRAM memories are presented, followed by an explanation of a general real-time memory controller.

3.1 Introduction to SDRAM Memories

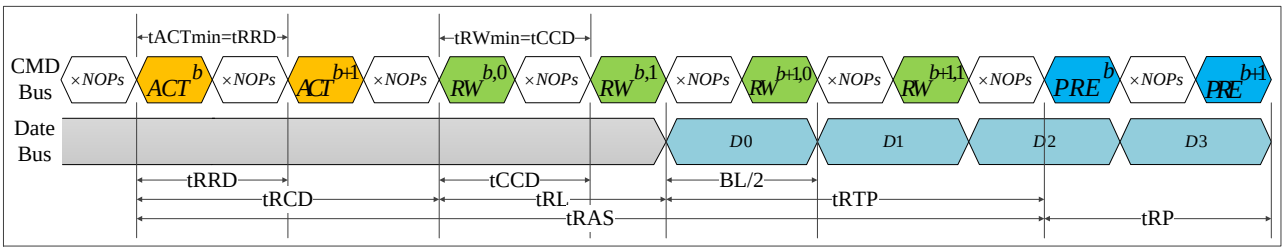
SDRAM memories are composed of a set of banks that include memory elements arranged in rows and columns [14], as shown in Fig. 1(a). Typical numbers for contemporary DDR3 devices are 8 banks, 8192 rows and 1024 columns. The SDRAM interface consists of command, address, and data buses. A single command can be transferred per clock cycle, while two data words can be transferred per cycle by a contemporary DDR3 memory. Before a command can be scheduled, several timing constraints have to be satisfied as specified by the JEDEC DDR3 standard [15]. Note that although this document focuses on DDR3 SDRAMs, it requires only minor adaptations to work with other types of SDRAMs, such DDRx, LPDDRx and Wide I/O.

When accessing an SDRAM, the contents of a row are copied to the row buffer in each bank by issuing an *Activate* (*ACT*) command. It takes *tRCD* cycles to fetch the data from the storage cells and copy it to the row buffer. Then a set of *Read* (*RD*) or *Write* (*WR*) commands can be issued to the

open row to transfer bursts of a programmed burst length (*BL*) (typically 8 words). The required data become available on the data bus within *tRL* or *tWL* cycles after issuing the *RD* or *WR* command, respectively. Before another row in the same bank is activated, the current row must be closed by writing back the contents to the storage cell using a *Precharge (PRE)* command. The *PRE* command can be issued at least *tRAS* cycles after the *ACT* command and *tRTP* cycles after the *RD* command to the same bank. The *PRE* command can be issued either as a separate command that is scheduled on the command bus or by adding an auto-precharge flag to a *RD* or *WR* command, where precharging is automatically triggered as soon as all timing constraints are satisfied. A *PRE* command following a *WR* cannot be issued until *tWR* cycles after the last data has been written to the bank. The minimum time between successive *RD* or *WR* commands is *tCCD* cycles. In addition, the timing constraint *tFAW* specifies a time window in which there are at most 4 scheduled *ACT* commands to meet the power limits. Finally, the memory elements must be refreshed regularly with the period of *tREFI* and the duration of this operation is *tRFC* cycles. These timing constraints are summarized in Table 1, where a 16-bit DDR3-1600 memory device with a capacity of 64 Mb is taken as an example.



(a) SDRAM architecture



(b) Timing constraints

Figure 1: SDRAM architecture and timing constraints.

The banks are accessed by scheduling a number of commands. An individual bank is accessed by scheduling an *ACT* command and several *RD* or *WR* commands. For example, the accesses of bank b and $b+1$ are illustrated in Figure 1(b). An ACT^b command is scheduled to open the required row in bank b and two consecutive *RD* or *WR* commands ($RW^{b,0}$ and $RW^{b,1}$) that have sequential addresses within the same row are scheduled to fetch or write data. The minimum time interval between ACT^b and $RW^{b,0}$ is *tRCD* while $RW^{b,1}$ is scheduled *tCCD* cycles later than $RW^{b,0}$. A PRE^b command is issued after the last access to the opened row in bank b. Bank $b+1$ is accessed in a similar way. All the corresponding timing constraints are illustrated in the figure.

Table 1: Relevant timing constraints for DDR3 SDRAM.

TC	*Description*	DDR3-1600 [cycles]
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	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 period time	8
tWTR	Internal *WR* command to *RD* command delay	6
tWR	Write recovery time. Minimum time after the last data has been written to a bank until a precharge may be issued	12
tRFC	Refresh period time	72
tREFI	Refresh interval	6240

3.2 Real-Time Memory Controllers

A general real-time memory controller is composed of a front-end and a back-end, as shown in Fig. 2. The front-end receives transactions from memory clients, such as processors or hardware accelerators, and buffers them in separate queues per client. One of these transactions is then selected by the arbiter according to some policy, such as Round Robin [19] or Credit-Controlled Static-Priority Arbitration [3], and sent to the back-end.

In the back-end, the logical address of a transaction is translated into physical address (bank, row, and column) according to the memory map. The configuration of this memory map determines how a transaction is split over the memory banks and thus the degree of bank parallelism used when serving it. This is captured by two critical parameters being the bank interleaved number (*BI*) and the burst count *BC* [11]. *BI* determines the number of banks that are accessed on behalf of a transaction while *BC* represents the number of *RD* or *WR* commands per bank. The product of BI and BC is hence always constant for a given transaction size, since it corresponds to a fixed number of read or write bursts. After the physical address is determined by the memory map, the back-end executes the transaction by generating and scheduling the required commands to the memory banks to access the memory. The command generator is responsible for generating the appropriate commands which are then issued to the memory by the command scheduler subject to the timing constraints of the memory.

Real-time memory controllers [1, 10, 19, 21, 23] typically employ a close-page policy. Under a close-page policy, the SDRAM controller precharges the open row as soon as possible after each bank access. The advantage is that the time from the precharge-to-activate can be (partially) hidden by bank parallelism. Therefore, the execution time of a transaction can be minimized if it requires

access to a different row than the currently opened one, reducing the worst-case execution time of a transaction [11].

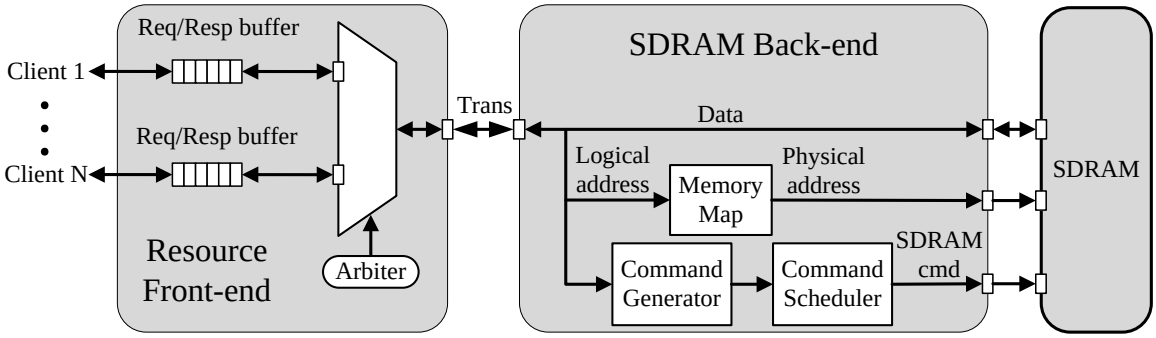


Figure 2: A general SDRAM controller supporting N clients.

4 Dynamically scheduled back-end

This section presents our dynamically scheduled memory controller back-end. First, the architecture of the back-end and the responsibilities of its constituent parts are described, followed by a specification of the dynamic scheduling algorithm. Here, we focus on the functional behavior of the architecture and later return to formalize the timing behavior of the back-end in Section 5.

4.1 Back-End Architecture

The back-end of a memory controller receives transactions scheduled by the front-end, which are read or write requests with variable sizes. The received transaction is executed by generating and scheduling commands to one or more consecutive banks of the memory. The first step towards this is to determine the *BI* and *BC* of the scheduled transaction to make sure the appropriate commands are generated. This is implemented by means of a Lookup Table, which is indexed by the transaction size, as shown in Fig. 3. The values of this Lookup Table are determined at design time when the memory map configuration is chosen and are programmed via a configuration interface when the system is initializing. A methodology for choosing the memory map configuration based on the requirements on bandwidth, response time, and power consumption has been presented in [11]. With the *BI* and *BC* of the transaction obtained from the Lookup Table, the Memory Map module in Fig. 3 translates the logical address of the transaction into the physical address. In addition, the Memory Map module also provides the ID of the starting bank.

Based on *BI*, *BC*, the starting bank ID and the physical address, the Command Generator generates the memory commands for accessing all required banks. An *ACT* command is necessary for opening a row in a bank and then a number of successive *RD* or *WR* commands are used to read/write data. Finally, a *PRE* command is required to close the opened row, implementing a close-page policy. *BI* determines the number of required *ACT* and *PRE* commands, one of each type per accessed bank, while *BC* determines the number of *RD* or *WR* commands per bank access. However, no *PRE* commands are actually generated. Instead, the auto-precharge flag is used together with the last *RD*

or *WR* command to each bank, thereby reducing the number of potential command conflicts while guaranteeing that the precharge happens as early as possible. Hence, each bank access comprises an *ACT* command and *BC* number of *RD* or *WR* commands and an auto-precharge. These commands are generated and stored in separate command queues per bank. The command queues operate in a First-In-First-Out (FIFO) manner and operate in parallel for each individual bank. Commands for each bank are generated sequentially starting with the *ACT* command and followed by the *RD* or *WR* commands.

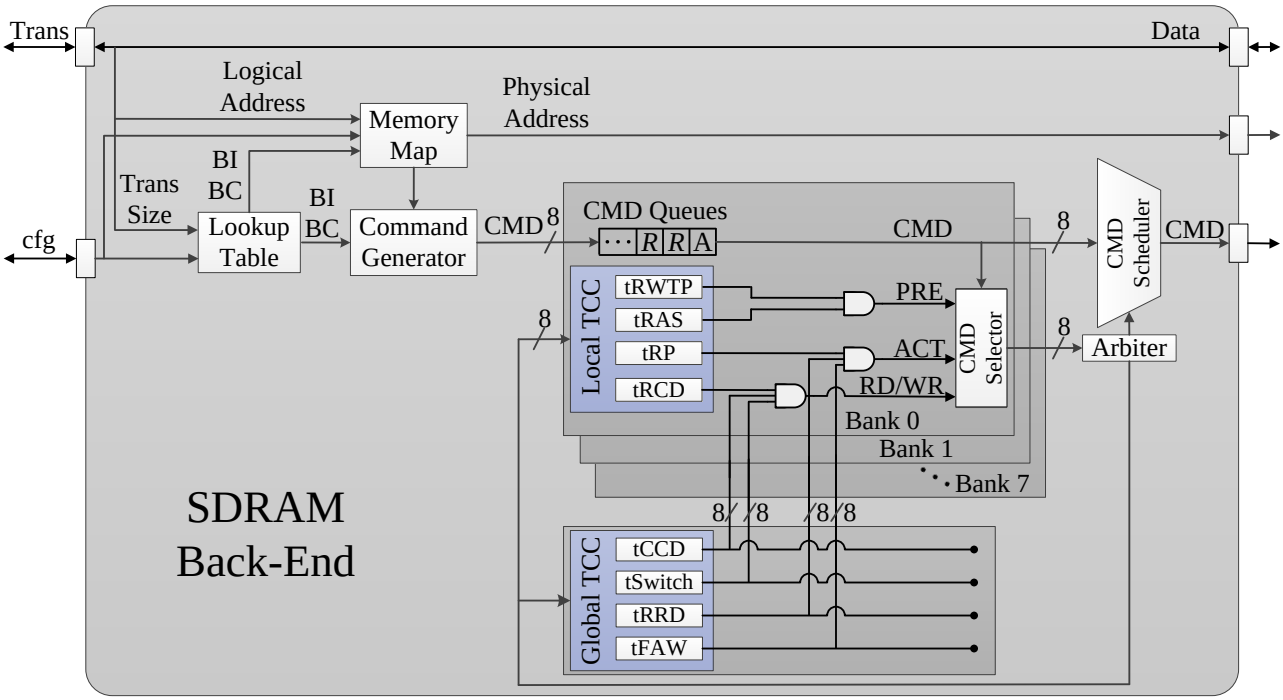


Figure 3: Back-end architecture of the dynamically scheduled SDRAM controller.

To satisfy the timing constraints of the commands, timing counters are used. They are initialized with the timing specifications given by the JEDEC DDR3 standard [15], and implement the timing constraints by counting down towards zero on every clock signal. As shown in Fig. 3, the timing constraint counters (TCC) are classified into two groups: local TCC and global TCC. The local TCC consider the *tRWTP*, *tRAS*, *tRP* and *tRCD* timing constraints that determine the command scheduling for a single bank. The global TCC consider the *tCCD*, *tRRD*, *tFAW* and *tSwitch* timing constraints and are shared by all the banks. These timing constraints are all specified by JEDEC [15], except *tRWTP* and *tSwitch* which follow directly from the specification and are shown in Eq. (1) and (2), respectively. *tRWTP* is the time interval between a *RD* or *WR* command and the *PRE* to the same bank, while *tSwitch* gives the time interval between two successive *RD* or/and *WR* commands. Due to the double data rate of DDR SDRAM, *BL*/2 is the time consumed by transferring the data associated with a *RD* or *WR* command.

$$tRWTP = \begin{cases} tRTP, & \textit{if } \text{PRE } \textit{follows } \text{RD} \\ tWL + BL/2 + tWR, & \textit{if } \text{PRE } \textit{follows } \text{WR} \end{cases} \quad (1)$$

$$tSwitch = \begin{cases} tRTW, & \textit{if} \text{ WR } \textit{follows} \text{ RD} \\ tWL + BL/2 + tWTR, & \textit{if} \text{ RD } \textit{follows} \text{ WR} \\ tCCD, & \textit{otherwise} \end{cases} \quad (2)$$

When a command of the transaction moves to the head of its command queue, the command scheduler starts trying to schedule this command to the memory. However, it has to wait until all the commands belonging to the previous transaction have been issued. There is hence no pipelining of commands belonging to different transactions, which makes the scheduler simple to analyze at the expense of slightly longer execution times. The scheduler has to wait at least one cycle after the last command of the previous transaction is issued before trying to schedule the first *ACT* command for the following one. Moreover, it has to wait at least two cycles after the arrival of the transaction at the interface of the back-end, corresponding to the two pipeline stages of the Lookup Table and Command Generator.

As shown in Fig. 3, each CMD selector has three inputs that represent whether the timing constraints for *ACT*, *PRE* and *RD* or *WR* are satisfied in the current clock cycle. The valid input is selected according the command at the head of the queue. Moreover, multiple CMD Selectors may have valid outputs at the same time. This indicates that the timing constraints for several commands at the heads of the different queues are satisfied simultaneously. This implies a command scheduling collision, since only one command can be issued per cycle. Therefore, it requires an arbiter to choose only one valid output of the CMD Selectors. Finally, the chosen command is removed from the head of its queue and scheduled to the memory. Meanwhile, both the local and global TCCs associated with the scheduled command are updated. This is shown by the feedback wires from the output of the arbiter to the TCC in Fig. 3.

4.2 Scheduling Algorithm

We proceed by turning our focus to the scheduling algorithm used by the back-end. In order to guarantee the dynamic command scheduling is analyzable in worst-case situation, transactions are executed in the back-end without inter-transaction pipelining. In addition, the banks are required to be accessed in ascending order according to the bank ID, thereby preventing data to return out of order. Thirdly, the commands have to be prioritized because of the scheduling collisions. This section specifies the dynamic command scheduling algorithm that implement these rules and is used by the Arbiter block, as shown in Fig. 3. The proposed dynamic scheduling algorithm is a simple priority-based mechanism to make analysis of the worst-case execution time easy with different memory map configurations.

The priorities between commands that are eligible for scheduling at the same time depend on whether or not they belong to the same transaction. For commands belonging to the same transaction, *RD* or *WR* commands have higher priority than *ACT* commands to immediately get data on the bus, and because delaying an *ACT* command to another bank does not necessarily delay the corresponding *RD* or *WR* commands, as we will show in Section 5. In addition, the algorithm accesses banks in ascending order to make sure data is not reordered. Among commands with the same type, higher priority is hence given to the one that require access to a bank with a smaller ID. To keep things simple, there is no pipelining of commands belonging to different transactions and the first *ACT*

command of a transaction is hence not allowed to be scheduled until the last *RD* or *WR* command of the previous transaction has been issued. Note that the algorithm does not specify any priorities between *RD* or *WR* and *ACT* commands targeting the same bank, since they are queued sequentially in the Command Queues and cannot be eligible at the same time.

Algorithm 1 formally specifies the scheduling priorities of commands of a transaction T, which has BI and BC, and its starting bank ID is b_s. The previous transaction is T'. It uses BI' and BC', and its starting bank ID is b'_s. The ACT and RD or WR commands are denoted by ACT^b and $RW^{b,k}$, respectively, where b represents the ID of the bank to which the commands are issued and k is the number of a RD or WR command to the same bank. The priority of a command is denoted by $P(cmd)$.

Algorithm 1 Priorities for dynamic command scheduling

Within Transaction:
  For T, $\forall b \in [b_s, b_s + BI - 1]$ and $\forall m, \forall n \in [0, BC - 1]$,
    $P(ACT^b) > P(ACT^{b+1})$
    $P(RW^{b,m}) > P(RW^{b+1,n})$
    $P(RW^{b,m}) > P(ACT^{b+1})$
Between Transactions:
  For T', $\forall b' \in [b'_s, b'_s + BI' - 1]$ and $\forall k \in [0, BC' - 1]$,
    $P(ACT^{b'}) > P(ACT^b)$
    $P(RW^{b',k}) > P(ACT^b)$

5 Formalization of Dynamic Command Scheduling

In this section, the formalization of dynamic command scheduling is carried out considering the timing dependencies for successive bank accesses. Based on the dependencies, several basic equations are derived for calculating the time at which a command is issued to a bank (referred to as the scheduling time). For convenience, the notation in this section is summarized in Table 2.

5.1 Timing dependencies

In dynamic command scheduling, commands are scheduled sequentially according to their associated timing constraints, resulting in scheduling dependencies. This is shown in Fig. 4, where the dotted and solid arrows represent dependencies between banks and within a single bank, respectively. The scheduling of a command depends on the previous commands, which are specified by the input arrows. The parameters near the arrows provide the number of cycles that the following command has to wait until the timing constraints are satisfied. For example, the timing constraints for scheduling an *ACT* command include *tRRD*, *tRP* and *tFAW*, previously described in Table 1. Therefore, the block of an *ACT* command (see Fig. 4) has three input arrows that represent the corresponding timing constraints. Regarding the scheduling of a *RD* or *WR* command, it has to satisfy the timing constraints *tRCD* and *tSwitch* for the first *RD* or *WR* command of the bank access. However, the scheduling of the following *RD* or *WR* commands of the bank access only takes the timing constraint *tCCD* into

Table 2: Summary of notation.

Symbol	*Description*
j	The current bank access number in the back-end
T_i	The i^{th} transaction
BI_i, BC_i	The bank interleaved number (BI) and burst count (BC) used by T_i
$ACT_j^{b_j}$	The ACT command for the j^{th} access targeting bank b_j
$t(ACT_j^{b_j})$	The scheduling time of $ACT_j^{b_j}$
$RW_j^{b_j,k}$	The k^{th} ($\forall k \in [0, BC_i]$) RD or WR command of the j^{th} access targeting bank b_j
$t(RW_j^{b_j,k})$	The scheduling time of $RW_j^{b_j,k}$
$PRE_j^{b_j}$	The PRE command for the j^{th} access targeting bank b_j
$t(PRE_j^{b_j})$	The actual precharge time specified by $PRE_j^{b_j}$
$t_s(T_i)$	The starting time of T_i
$\hat{t}_s(T_i)$	The worst-case starting time of T_i
$t_f(T_i)$	The finishing time of T_i
$\hat{t}_f(T_i)$	The worst-case finishing time of T_i
$t_{ET}(T_i)$	The execution time of T_i

account. Finally, an auto-precharge must consider the timing constraints *tRAS* and *tRWTP*. The timing dependencies among the commands are illustrated in Fig. 4.

According to Algorithm 1, an *ACT* command may be blocked by a *RD* or *WR* command from previous bank accesses since they have higher priorities. Therefore, a command scheduling conflict may be caused and this collision postpones the *ACT* command to be scheduled in the next cycle. The collision is depicted by the blue circle. This represents a *RD* or *WR* command that blocks the *ACT* command. The arrow corresponding to the maximum time dominates the scheduling of a dependent command, since all relevant timing constraints must be satisfied. Moreover, the *PRE* in Fig. 4 does not use the command bus due to the auto-precharge policy. However, the time at which the auto-precharge actually happens is necessary to determine when the bank can be reactivated.

5.2 Formalization of bank accesses

Having explained the dependencies between commands in a bank access according to the DDR3 standard and illustrated them in Fig. 4, we proceed by computing their scheduling times under our dynamic scheduling algorithm.

For $\forall j > 0$, the j^{th} bank access is implemented by scheduling an $ACT_j^{b_j}$ and several *RD* or *WR* commands to bank b_j. The *RD* or *WR* commands are denoted by $RW_j^{b_j,k}$, where $\forall k \in [0, BC_j - 1]$. BC_j is the number of *RD* or *WR* commands for the j^{th} bank access, obtained from the Lookup Table in Section 4. Moreover, an auto-precharge, $PRE_j^{b_j}$, is implemented after the access of bank b_j and it is specified by an auto-precharge flag issued together with $RW_j^{b_j,BC_j-1}$.

Eq. (3) computes the scheduling time of $ACT_j^{b_j}$ where m ($m < j$) is the previous access to bank b_j. The $max\{\}$ function guarantees that all the timing constraints for scheduling $ACT_j^{b_j}$ are satisfied. In case

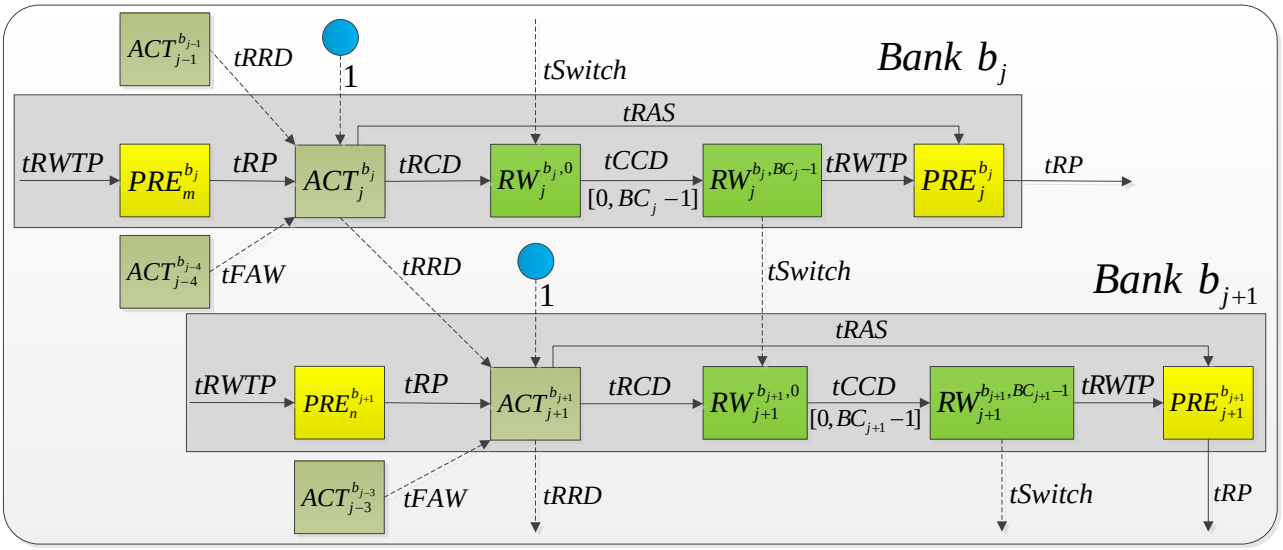


Figure 4: Timing dependencies between successive bank accesses.

of a command scheduling collision, $C(j)$ is equal to 1 if $ACT_j^{b_j}$ is blocked by a *RD* or *WR* command, and 0 otherwise. Similarly, the scheduling time of $RW_j^{b_j,k}$, which follows $ACT_j^{b_j}$, is given by Eq. (4) and (5). Eq. (4) provides the scheduling time of the first *RD* or *WR* command to bank b_j. It depends on $t(RW_{j-1}^{b_{j-1},BC_{j-1}-1})$, which is the scheduling time of the last *RD* or *WR* during the previous bank access ($(j-1)^{th}$), and the scheduling time of the *ACT* command to the same bank b_j. However, the successive *RD* or *WR* commands to bank b_j can be issued only based on the previous *RD* or *WR* command. The scheduling time of these commands are given by Eq. (5). Finally, the actual precharging time of the auto-precharge for bank b_j is given by Eq. (6). 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.

$$\begin{aligned} t(ACT_j^{b_j}) = max\{&t(ACT_{j-1}^{b_{j-1}}) + tRRD, \\ &t(PRE_m^{b_j}) + tRP, \\ &t(ACT_{j-4}^{b_{j-4}}) + tFAW\} + C(j) \end{aligned} \tag{3}$$

$$\begin{aligned} t(RW_j^{b_j,0}) = max\{&t(RW_{j-1}^{b_{j-1},BC_{j-1}-1}) + tSwitch, \\ &t(ACT_j^{b_j}) + tRCD\} \end{aligned} \tag{4}$$

$$t(RW_j^{b_j,k}) = t(RW_j^{b_j,0}) + k \times tCCD \tag{5}$$

$$\begin{aligned} t(PRE_j^{b_j}) = max\{&t(ACT_j^{b_j}) + tRAS, \\ &t(RW_j^{b_j,BC_j-1}) + tRWTP\} \end{aligned} \tag{6}$$

5.3 Formalization of transactions

We now proceed by formalizing important events in the life of an arbitrary transaction T_i ($\forall i > 0$) with BI_i and BC_i, and the starting bank is b_s. Definition 1 defines the arrival time of T_i as the time at which it arrives at the interface of the SDRAM back-end. On the other hand, Definition 2 defines the execution of T_i as finished when the last *RD* or *WR* command $RW_{j+BI_i-1}^{b_s+BI_i-1,BC_i-1}$ is scheduled. The starting time of T_i is defined as the earliest time at which its commands can be scheduled. This is either one cycle after the finishing time of the previous transaction T_{i-1} (no pipelining between transactions) or two cycles after the arrival time (pipeline stages for the Lookup Table and Command Generation), whichever is larger. Lastly, the difference between the finishing time and the starting time is referred to as the execution time of the transaction, defined in Definition 4. These definitions are illustrated in Fig. 5 for the case of a transaction T_i with $BI_i = 2$ and $BC_i = 2$.

Definition 1 (Arrival time). *$t_a(T_i)$ is defined as the time at which T_i arrives at the interface of the back-end.*

Definition 2 (Finishing time). $t_f(T_i) = t(RW_{j+BI_i-1}^{b_s+BI_i-1,BC_i-1})$

Definition 3 (Starting time). $t_s(T_i) = max\{t_a(T_i) + 2, t_f(T_{i-1}) + 1\}$

Definition 4 (Execution Time). *The execution time of T_i is defined as $t_{ET}(T_i) = t_f(T_i) - t_s(T_i) + 1$.*

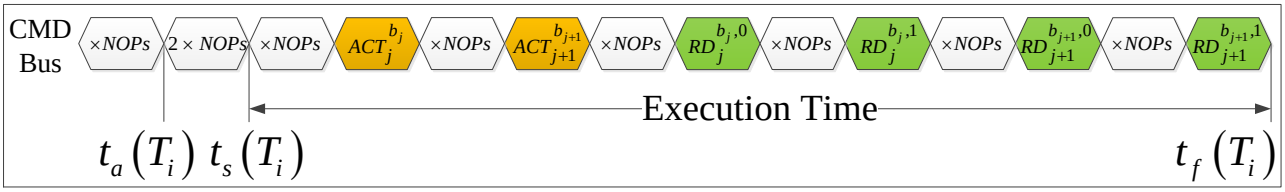


Figure 5: The time definitions for dynamic command scheduling.

Fig. 4 only shows the basic bank access dependencies caused by the associated timing constraints and the possible scheduling collisions. As transactions are executed without pipeline, the scheduler cannot schedule the first *ACT* command for T_i until the previous transaction T_{i-1} finishes, i.e., the last *RD* or *WR* command of T_{i-1} has been finished. We assume the current bank access number is j in the back-end. The timing dependencies of the bank accesses for T_i are given by Fig. 6. The scheduling of $ACT_j^{b_s}$ not only depends on the former $PRE_m^{b_s}$ and $ACT_{j-4}^{b_{j-4}}$, but also on the scheduling time $t(RW_{j-1}^{b_{j-1},BC_{i'}-1})$ ($i' = i-1$), which is the finishing time of T_{i-1}. It follows by definition that $ACT_j^{b_s}$ cannot be scheduled earlier than the starting time, previously expressed in Definition 3. Therefore, the scheduling time of $ACT_j^{b_s}$ is given by Eq. (7), where $t'(ACT_j^{b_s})$ is obtained from Eq. (3). In addition, the definitions of starting time and finishing time prevent a command scheduling collision for $ACT_j^{b_s}$, since the last *RD* or *WR* of T_{i-1} must have been scheduled before T_i starts. As a result, $C(j) = 0$. For $BI_i > 4$, the scheduling of some *ACT* commands depends on the previous *ACT* commands belonging to T_i as well because of the four-activate window (*tFAW*). As a result, $i' = i$ only if $BI_i > 4$.

$$t(ACT_j^{b_s}) = max\{t'(ACT_j^{b_s}), t_s(T_i)\} \tag{7}$$

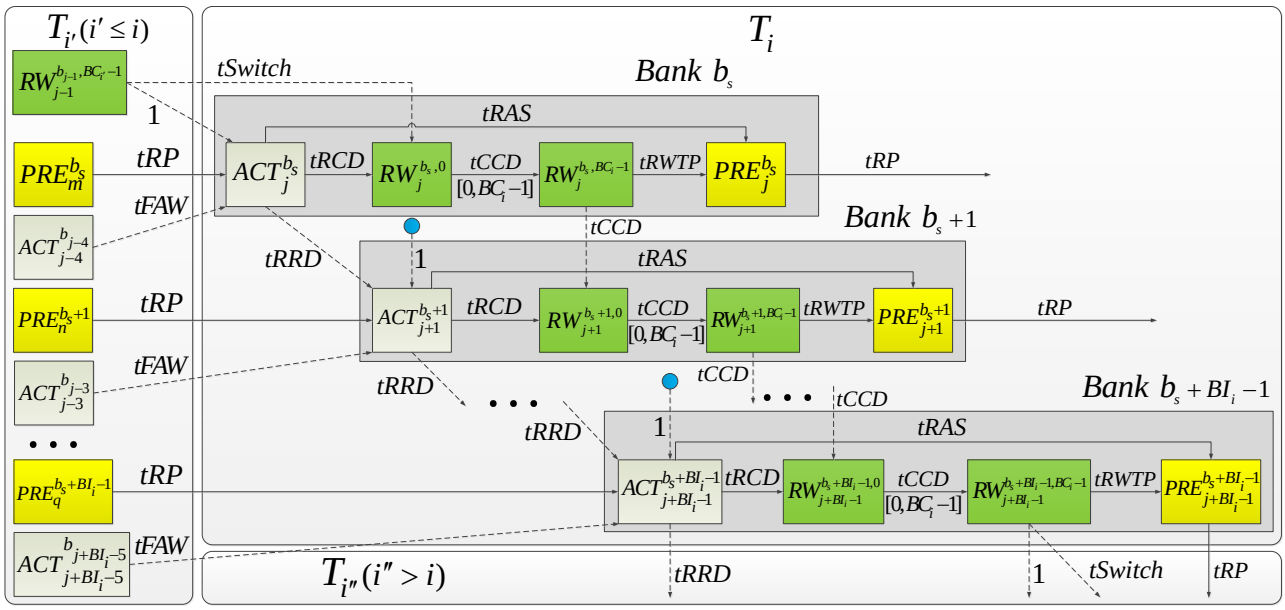


Figure 6: The timing dependencies of command scheduling for transaction T_i.

Based on Eq. (3) to (7), 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 Fig. 6, only the *ACT* and the first *RD* or *WR* commands of T_i to each bank have dependencies on the previous transactions. Then the rest *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 by the scheduling time of all its *ACT* commands and the finishing time of the previous transaction. This intuition is formalized by Lemma 1 and the proof is included in the appendix.

Lemma 1. *For* $\forall l \in [0, BI_i - 1]$,

$$
\begin{aligned}
&t_f(T_i) = \\
&max\{t(ACT^{b_s+l}_{j+l}) + tRCD + ((BI_i - l) \times BC_i - 1) \times tCCD, \\
&\quad t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned}
$$

6 Worst-case Execution Time

In this section, the worst-case execution time of our proposed dynamic command scheduling algorithm is analyzed for an arbitrary transaction T_i. Firstly, the worst-case situation is discussed, which defines the worst-case scheduling time of the previous commands targeting the same set of banks as T_i. Furthermore, the worst-case finishing time of T_i is computed based on the worst-case situation, its size and the memory map configuration.

6.1 Worst-Case Situation

An arbitrary transaction T_i is executed by scheduling its commands to the memory, where the execution time depends on the state of the required banks at the beginning of the execution. However, this initial state is determined by the previously executed transactions $T_{i'}$ ($i' < i$) that have accessed these banks. Due to the diversity of $T_{i'}$ in terms of type (read/write), transaction size, and different required banks, etc., it is difficult to determine the worst-case initial state for T_i. This issue is discussed on the basis of the dependencies in Eq. (3) to (7) in the following paragraphs.

Definition 4 states that the execution time, $t_{ET}(T_i)$, is maximized only if the starting time is minimum while the finishing time is maximum. According to Definition 3, 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 to wait until T_{i-1} is finished, which results in the minimum starting time of T_i. Therefore, the worst-case starting time of T_i is only one cycle later than the finishing time of T_{i-1} and is given by Eq. (8), where the current bank access number is j, and T_{i-1} has BC_{i-1}.

$$\hat{t}_s(T_i) = t_f(T_{i-1}) + 1 = t(RW_{j-1}^{b_{j-1},BC_{j-1}-1}) + 1 \tag{8}$$

In order to get the worst-case finishing time of T_i, the scheduling time of all the *ACTs* should be maximized according to Lemma 1. According to Eq. (3), the scheduling of an *ACT* command for T_i depends on the previous *PRE* command to the same bank and the previous *ACT* commands. Therefore, the maximum scheduling times of the previous *PRE* commands and *ACT* commands result in the worst-case finishing time of T_i. Due to the worst-case starting time given by Eq. (8), the finishing time $t(RW_{j-1}^{b_{j-1},BC_{j-1}-1})$ of T_{i-1} is obtained. Since we do not know exactly how T_{i-1} was scheduled, we conservatively assume they were scheduled *As-Late-As-Possible* (*ALAP*) subject to the timing constraints. The reason is that *ALAP* ensures the maximum scheduling time of the previous *PRE* and *ACT* commands.

According to *ALAP* scheduling, the scheduling time of the previous *ACT*, *RD* or *WR* and *PRE* commands can be obtained by calculating backwards from $t(RW_{j-1}^{b_{j-1},BC_{i-1}-1})$, which is fixed by Eq. (8) if we assume the execution of T_i starts at the worst-case starting time $\hat{t}_s(T_i)$. Specifically, the time interval between any successive commands must be minimum while satisfying the timing constraints, thereby ensuring an *ALAP* schedule of the previous commands. We assume T_{i-1} has BI_{i-1} and BC_{i-1}. As stated in Table 1, the minimum time interval between two *RD* or *WR* commands is *tCCD*. Since *RD* or *WR* commands targeting the same bank are scheduled sequentially, the time interval between the first *RD* or *WR* commands to consecutive banks is $BC_{i-1} \times tCCD$. An *ACT* command is followed by a *RD* or *WR* command to the same bank, and their minimum time interval is *tRCD* (see Table 1). Therefore, the scheduling time of each *ACT* command is obtained through the scheduling time of the *RD* or *WR* commands issued to the same bank. As a result, the time interval between two successive *ACT* commands is at least $BC_{i-1} \times tCCD$. In addition, Table 1 also states that the minimum time interval between two *ACT* commands to different banks is *tRRD*. Hence, for *ALAP* scheduling, the minimum time interval between two successive *ACT* commands to different banks is $max\{tRRD, BC_{i-1} \times tCCD\}$.

Fig. 7 illustrates an example of *ALAP* scheduling for a DDR3-1600 SDRAM, and the associated timing constraints are presented in Table 1. This example assumes the current transaction T_i has

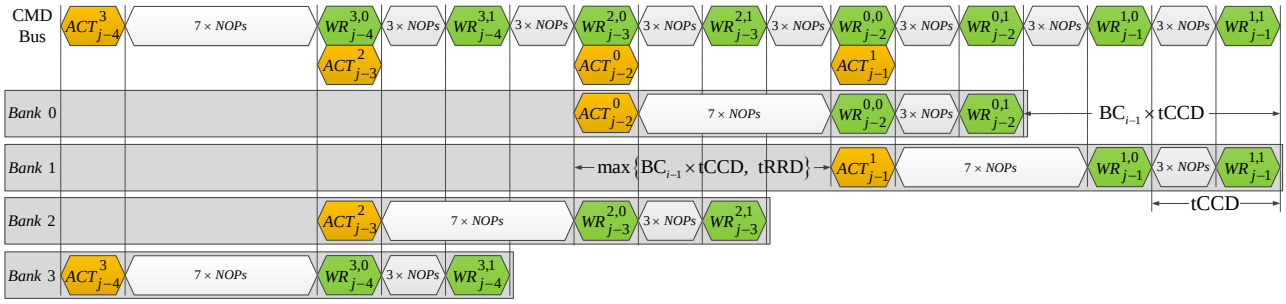


Figure 7: An example of As-Late-As-Possible (ALAP) scheduling with DDR3-1600 SDRAM for T_i which has $BI_i = 4$ and $BC_i = 2$. The previous transaction T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. The starting bank for both T_{i-1} and T_i is *Bank* 0.

$BI_i = 4$ and $BC_i = 2$ while the previous write transaction T_{i-1} is half the size and uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$, both transactions having *Bank* 0 as their starting bank. We assume the current bank access number is j. With the fixed finishing time ($t(RW^{1,1}_{j-1})$) of T_{i-1}, the scheduling time of all the previous commands is computed backwards with the minimum time interval between them. In this way, some *ACT* commands have the same scheduling time as some *WR* commands, which indicate command scheduling collisions. However, we conservatively ignore these collisions so that larger scheduling time of the previous *ACT* and *WR* commands is achieved. Fig. 7 shows the scheduling times of the previous commands that are scheduled to the banks (*Bank* 0 and 1) required by T_{i-1} and the banks (see *Bank* 2 and 3) required by even earlier transactions, e.g., T_{i-2}. Since it will access *Bank* 2 first and then *Bank* 3 for T_i, the scheduling times of the previous commands to *Bank* 2 are computed backwards first, resulting in larger times than that to *Bank* 3. In this way, the computed scheduling times of *WR* commands for *Bank* 2 and 3 are larger than the actual value because the minimum time interval between *RD* or *WR* commands from different transactions is larger than *tCCD*. Hence, the computed scheduling times of the previous commands guarantees a conservative execution time bound for later transactions.

ALAP scheduling can be formalized to provide the scheduling times of previous commands. We assume T_i has BI_i and BC_i, and its starting bank is b_s, while T_{i-1} has BI_{i-1} and BC_{i-1}. Firstly, we assume the starting bank of T_{i-1} is b_s as well, because the scheduling time of the previous commands targeting the banks that are not required by T_i can be ignored. Secondly, in the worst-case situation, there must be $b_s + BI_{i-1} - 1 \in [b_s, b_s + BI_i - 1]$, which represents the finishing bank of T_{i-1}. It indicates T_{i-1} finished at a bank which is required by T_i. With the minimum time interval between commands, for $\forall l \in [0, BI_i - 1]$ and $\forall k \in [0, BC_{i-1} - 1]$, the scheduling time of the previous *RD* or *WR* commands for bank $b_s + l$ is given by Eq. (9). In case $BI_{i-1} < BI_i$, Eq. (10) is used to compute of scheduling time for *RD* or *WR* commands targeting a bank $b_s + l$ ($l \in [BI_{i-1}, BI_i - 1]$) that is not required by T_{i-1}.

$$\hat{t}(RW^{b_{j-1-\Delta l},k}_{j-1-\Delta l}) = \hat{t}_s(T_i) - 1 - (BC_{i-1} - 1 - k) \times tCCD - \Delta l \times BC_{i-1} \times tCCD \quad (9)$$

$$\Delta l = \begin{cases} BI_{i-1} - 1 - l, & l \leq BI_{i-1} - 1 \\ l, & other \end{cases} \tag{10}$$

Due to the timing constraint *tFAW*, we just need the scheduling time of the four *ACT* commands that were scheduled previously. Based on the fixed 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 1). Thus, with the minimum time interval between *ACT* commands, the scheduling time of the previous four *ACT* commands is calculated by Eq. (11). Based on Eq. (6), the scheduling time of the previous *PRE* commands is obtained by using the worst-case scheduling times for *RD* or *WR* and *ACT* commands from Eq. (9) and (11), respectively. It is given by Eq. (12) based on the observations of the timing constraints in JEDEC DDR3 SDRAM standard [15] that: i) *tRWTP* is larger for a write transaction than for a read transaction, and hence: ii) there is $tRWTP > tRAS - tRCD$ for a write transaction.

$$\begin{aligned} \hat{t}(ACT_{j-1-\Delta l}^{b_{j-1-\Delta l}}) = \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ - \Delta l \times max\{tRRD, BC_{i-1} \times tCCD\} \end{aligned} \tag{11}$$

$$\begin{aligned} \hat{t}(PRE_{j-1-\Delta l}^{b_{j-1-\Delta l}}) &= max\{\hat{t}(ACT_{j-1-\Delta l}^{b_{j-1-\Delta l}}) + tRAS, \\ &\qquad \hat{t}(RW_{j-1-\Delta l}^{b_{j-1-\Delta l, BC_{i-1}}}) + tRWTP\} \\ &= \hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times BC_{i-1} \times tCCD \end{aligned} \tag{12}$$

Hence, the worst-case situation for T_i is that the scheduling time of the previous *WR*/*ACT*/*PRE* command is given by Eq. (9), (11) and (12), respectively, which only depend on the worst-case starting time, transaction size (through BI_{i-1} and BC_{i-1} given by the memory map) and JEDEC specified timing constraints.

6.2 Worst-Case Finishing Time

Lemma 1 states that the finishing time of T_i is determined by the finishing time $t_f(T_{i-1})$ of the previous transaction T_{i-1} and the scheduling times $t(ACT_{j+l}^{b_s+l})$ ($\forall l \in [0, BI_i - 1]$) of the *ACT* commands for T_i. Therefore, the worst-case finishing time $\hat{t}_f(T_i)$ is obtained by using $\hat{t}_f(T_{i-1})$ and $\hat{t}(ACT_{j+l}^{b_s+l})$. By fixing the worst-case starting time of T_i, $\hat{t}_f(T_{i-1})$ is obtained by Eq. (8), where there is $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$. Regarding $\hat{t}(ACT_{j+l}^{b_s+l})$, it can be expressed by the worst-case scheduling time of the previous *ACT* commands and the *PRE* commands, given by Eq. (11) and (12), respectively. Eq. (3) indicates that $t(ACT_{j+l}^{b_s+l})$ is determined by $t(ACT_{j+l-1}^{b_s+l-1})$, $t(PRE_{j-(BI_i-l)}^{b_s+l})$ and $t(ACT_{j+l-4}^{b_{j+l-4}})$. As a result, $\hat{t}(ACT_{j+l}^{b_s+l})$ can be obtained by using $\hat{t}(PRE_{j-(BI_i-l)}^{b_s+l})$ and $\hat{t}(ACT_{j+l-4}^{b_{j+l-4}})$. In addition, $\hat{t}(ACT_{j+l}^{b_s+l})$ can be iteratively expressed by $\hat{t}(PRE_{j-1-(BI_i-l)}^{b_s+l-1})$ and $\hat{t}(ACT_{j+l-5}^{b_{j+l-5}})$ because they determine $\hat{t}(ACT_{j+l-1}^{b_s+l-1})$ according to Eq. (3). Eq. (12) provides $\hat{t}(PRE_{j-(BI_i-l)}^{b_s+l})$, while Eq. (11) provides $\hat{t}(ACT_{j+l-4}^{b_{j+l-4}})$ if $l < 4$. In order to simplify the expression, Lemma 2 gives $\hat{t}_f(T_i)$ with $BI_i \leq 4$, which ensures $\forall l < 4$. The proof is presented in the appendix. However, it is not difficult to extend Lemma 2 to support $BI_i > 4$.

Lemma 2. *For* $\forall l \in [0, BI_i - 1]$ *and* $\forall l' \in [l, BI_i - 1]$,

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

Lemma 2 indicates that the worst-case finishing time $\hat{t}_f(T_i)$ is determined by the variables l, l', BI_{i-1}, BC_{i-1}, BI_i and BC_i, which may be changed from transaction to transaction. However, the expressions in the $max\{\}$ of $\hat{t}_f(T_i)$, shown in Lemma 2, can be simplified since they are linearly increasing or decreasing with those variables. The maximum expression is obtained with $BI_{i-1} = BC_{i-1} = 1$, $l = 0$ and $l' = 0$ or $l' = BI_i - 1$. Hence, Theorem 1 is concluded to show the worst-case finishing time of T_i, which is determined by its starting time, its size and the memory map configuration (through BI_i and BC_i), and JEDEC defined timing constraints. Its proof is given in the appendix. Intuitively, it indicates that the worst-case situation for T_i is that its starting bank ($l = 0$) is accessed by the previous short write transaction T_{i-1}, which requires only one bank with one burst.

Theorem 1 (Variable transaction size)**.**

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

Theorem 1 provides a pessimistic worst-case execution time bound for systems where all transactions have fixed size and hence all have the same BI and BC. As a result, $BI_{i-1} = BI_i = BI$ and $BC_{i-1} = BC_i = BC$. Similarly, $\hat{t}_f(T_i)$ is obtained with $l' = BI - 1$ and $l = 0$, or $l' = l = BI - 1$ for transactions with fixed size. This is shown in Theorem 2, where the proof is shown in the appendix.

Theorem 2 (Fixed transaction size)**.**

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

Finally, the worst-case execution time of T_i can be obtained according to Definition 4 where the worst-case finishing time is presented by Theorem 1 and 2 for transactions with variable and fixed size, respectively. When the back-end is combined with a predictable memory controller front-end, such as [2], these results can be used to obtain the total response time of transactions under a particular transaction scheduling policy. This also requires the refresh time of the memory controller to be known. For our proposed memory controller architecture, a refresh operation is triggered every *tREFI* clock cycles and requires $tRWTP + tRP$ cycles to complete, where *tRWTP* is given in Eq. (1). This time includes the cycles required to precharge all banks before the refresh command is issued.

7 Experimental Results

This section experimentally evaluates our approach. First, we present the experimental setup, followed by three experiments. The first experiment shows the correspondence between the implementation of the proposed dynamically scheduled back-end and the formalization of its timing behavior. The second experiment evaluates the tightness of the bounds with fixed transaction size for several memories and memory map configurations, and compare the execution time to the semi-static pattern-based back-end that serves as the baseline for comparison in the T-CREST project. The last experiment shows that the derived bounds are conservative for transactions with variable size.

7.1 Experimental Setup

For our experiments, we select a wide range of DDR3 SDRAMs with frequencies of 800, 1066, 1333, 1600, 1866, and 2133 MHz. For each frequency, we chose devices with the minimum core timings, corresponding to the fastest memories within each speed bin. All memories have an interface width of 16 bits and a capacity of 64 Mb. All timing constraints are taken from the JEDEC DDR3 standard [15].

The proposed back-end architecture has been delivered as a fully functional SystemC implementation. In addition, the scheduling algorithm has been integrated into DRAMPower [6, 7], an open-source tool for memory power estimation. The experiments in this section are based on the scheduling algorithm in the SystemC implementation, which accurately captures the behavior of all timing constraint counters and the dynamic scheduling algorithm. This implementation is exercised using traces with different transaction sizes and memory map configurations. Each trace has 5000 transactions with sizes of 16, 32, 64, 128, 256, and 512 bytes.

7.2 Verification of the Formalization

The purpose of our first experiment is to check the correspondence between the implementation of our dynamically scheduled back-end from Section 4 and the formalization of its timing behavior from Section 5. Various traces are generated for this experiment, including random traces and fixed traces. The random traces randomly choose read or write transactions with random sizes while the fixed traces are generated by selecting transactions with the same size. Different memory map configurations are used by these traces. This experiment is done by executing these traces and compare the scheduling time of each command in the implementation to the formalization (Eq. (1) to (7)). The results of this experiment showed that the scheduling times of all commands of all transactions are identical, suggesting that the formalization accurately models the implementation.

7.3 Fixed Transaction Size

The second experiment investigates the tightness of the bound on execution time for fixed transaction sizes (Theorem 2) by comparing it to the maximum measured execution time from executing traces on our implementation of the back-end. We also compare the measured execution times to those of

the semi-static approach in [1], which is the only other approach supporting multiple memory map configurations. Traces with fixed transaction sizes are used in this experiment. However, for each size, we use all possible combinations of *BI* and *BC* to cover all memory map configurations. To increase coverage of the state space during the execution of the trace and increasing the chance of finding the worst-case situation, transactions vary between reads and writes according to an alternate parameter varying from 1 to 9. First, we have a look at the results for the DDR3-1600 memory for different combinations of *BI* and *BC*, shown in Fig. 8.

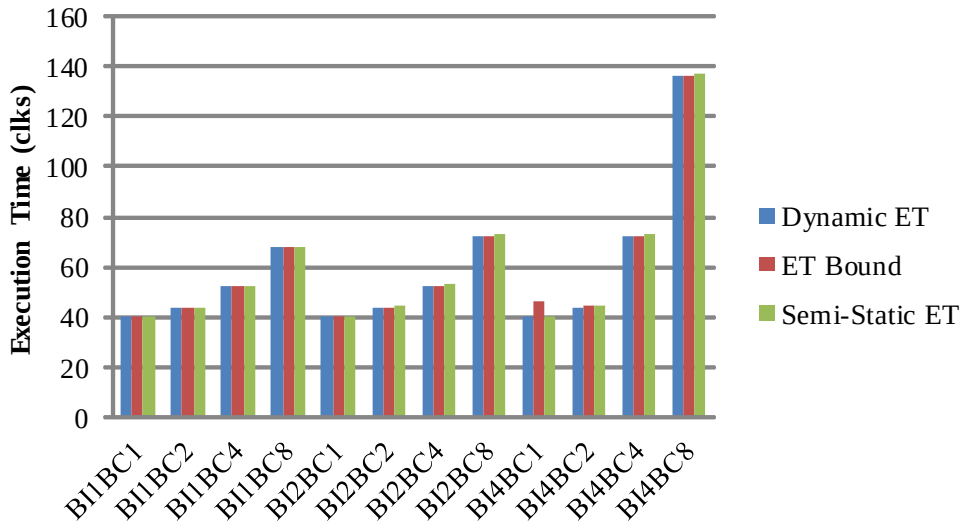


Figure 8: Execution time (ET) comparison between dynamic command scheduling and semi-static memory patterns.

The results in Fig. 8 gives an idea of execution times in absolute numbers for different memory map configurations. Note that transaction sizes corresponding to each configuration equals $BI \times BC \times 16$ bytes for these memories with a programmed burst length of 8 words. The figure shows that there is not a linear relation between transaction size and execution time (actual or bound) due to the timing constraints, and that increasing the number of banks used to serve a transaction reduces the execution time (although at the cost of increasing power consumption [11]. This is particularly clear when looking transactions of 128 B by comparing the different memory map configurations where $BI \times BC = 8$. Another important observation is that the maximum measured execution time is always less than the bound, suggesting that the bound is conservative for all memory map configurations. We also see that the bounds for our dynamic approach are very similar to those of the semi-static approach in [1], despite that both the proposed architecture and analysis are more general.

Fig. 9 shows the average tightness of the bound for all memory map configurations for all considered memories, as well as the reduction in actual execution time compared to the semi-static approach in [1]. This figure allows us to draw two important conclusions from this experiment: 1) The bound for our dynamic scheduling approach is conservative and is maximally 1.7% higher than the maximum measured execution time (for DDR3-1066), suggesting that the bound is tight, and 2) The execution time of transactions on our proposed dynamic scheduling algorithm is lower on average than the semi-static approach in [1] for fixed transactions sizes, although the maximum reduction is only 0.9% (again for DDR3-1066).

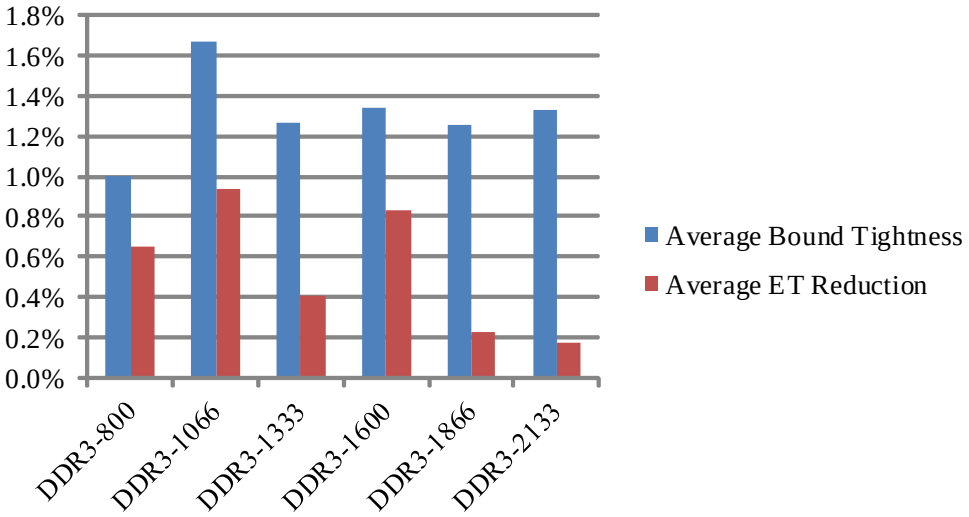


Figure 9: The average bound loss and the execution time (ET) reduction of dynamic command scheduling comparing with static memory patterns.

7.4 Variable Transaction Size

The last experiment evaluates our approach with variable transaction sizes. In this case, we cannot compare to other approaches, since there is no previous architecture and analysis that supports variable request sizes and memory map configurations. We hence only aim to show that the bound provided by Theorem 1 is conservative.

Fig. 10 shows the actual execution time of transactions and the bound for each individual transaction with a random trace for a DDR3-1600 memory. The results show that the bound is conservative for requests all requests of all sizes. It is furthermore clear that the dynamically scheduled back-end significantly outperforms the semi-static back-end used as a baseline, since this is unable to handle requests smaller than the access granularity of a pattern. This implies that both the actual execution time and the bound would be identical for smaller and larger requests and that unnecessary power would be consumed reading and writing unnecessary data that is later discarded.

8 Requirements

This section lists all requirements in aspect CORE and scope NEAR from Deliverable D 1.1 that are relevant for the memory work package. NON-CORE and FAR requirements are not listed here. The requirements are followed by a comment regarding the extent to which it is fulfilled by the implementation prototype in this document.

M-2-010 No Re-Order:

> The processor may have several read or write requests outstanding. The memory controller shall not reorder these read or write requests from the processor.
>
> *The proposed dynamically scheduled back-end preserves FIFO ordering between requests from its memory clients.*

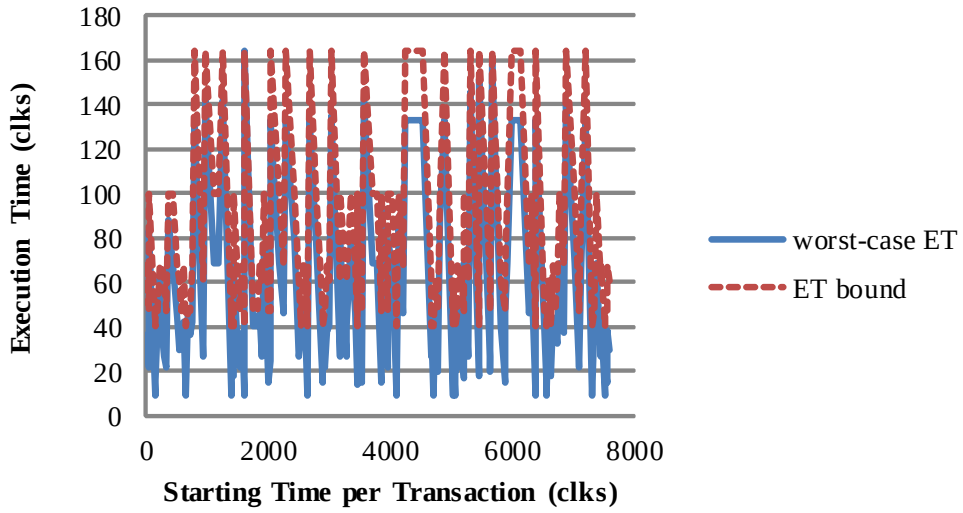


Figure 10: The worst-case execution time (ET) and the ET bound for each transaction in DDR3-1600 SDRAM with random trace.

M-2-012 Compare-and-Swap:

The memory controller (and network interface) shall support a CAS (compare-and-swap) operation.

An SDRAM controller works efficiently with large blocks of data and small accesses, such as CAS, should be avoided to enable efficient bounds on bandwidth and response times. CAS is hence more suitable for implementation in the controller of a scratchpad memory (that may or more not be close to the SDRAM controller) and is hence not discussed in this document.

M-4-020 DRAM Configuration:

DRAM configuration is memory mapped within the memory controller.

The delivered implementation of the memory controller is configured using XML parameters.

M-0-062 Memory Performance Counters:

The Memory shall have a cycle counter, which can be read out for performance analysis.

The delivered implementation is able to log time stamps for performance evaluation.

M-5-064 Memory Access Instruction Latency:

The latency of instructions to access main memory shall be latency-bounded.

The memory controller is time-predictable and is designed to provide efficient bounds on response times of requests for different sizes and different memory map configurations.

M-6-041 Bounded Memory Access Time:

Any access to a processor external resource (i.e. memory, NoC) shall execute in bounded time (depending on resource and access time).

See previous requirement.

9 Conclusions

This document proposes a back-end architecture of a real-time memory controller for supporting dynamic command scheduling. We focus on scheduling commands dynamically to the banks of SDRAM device, and the architecture is flexible to be used with any transaction arbitration algorithm in the front-end, such as the reconfigurable TDM arbiter presented in D 4.2 of the T-CREST project. Transactions are executed dynamically by scheduling commands to a number of consecutive banks using a simple predictable scheduling algorithm based on fixed command priorities. The dynamic command scheduling is formalized and the worst-case execution time is analyzed both for fixed and variable request sizes and for different memory map configurations. The proposed memory controller has been implemented in SystemC. Additionally, the proposed scheduling algorithm has been integrated with DRAMPower, an open-source tool for memory power estimation. Experimental results with the proposed scheduling algorithm shows that the execution time is tightly bounded, and the dynamic command scheduling has a slightly reduced worst-case execution time compared to the semi-static memory patterns. It hence provides increased flexibility without losing worst-case performance, thereby delivering on the goal of the design.

A Proofs

This section presents the proofs of the lemmas and theorems in the document.

A.1 Proof of Lemma 1

Proof. With the definition of finishing time (Definition 2), $t_f(T_i)$ can be obtained from Eq. (5), which provides the scheduling time of the last *RD* or *WR* command. Eq. (5) also indicates that the scheduling of a *RD* or *WR* command only depends on the previous *RD* or *WR* command targeting the same bank, except the first one that is determined by the *ACT* command to the same bank and the previous *RD* or *WR* command scheduled to a different bank.

Eq. (4) gives the scheduling time of the first *RD* or *WR*. Hence, through substitutions, the finishing time of T_i can be expressed by the scheduling times of the *ACT* commands and the finishing time $t_f(T_{i-1})$, which is the scheduling time of the last *RD* or *WR* command for T_{i-1}. In addition, *tSwitch* is equal to *tCCD* for the switching from one bank to another that is accessed by the same transaction. As a result, for computing the scheduling time for *RD* or *WR* commands belonging to the same transaction, *tCCD* is used by Eq. (4) instead of *tSwitch*.

For $\forall l \in [0, BI_i - 1]$, $t_f(T_i)$ is expressed according to Eq. (13) by iteratively using Eq. (4) and (5). It indicates that $t_f(T_i)$ only depends on the scheduling time of its *ACT* commands, the finishing time of T_{i-1} and JEDEC-specified timing constraints, which are constant.

$$\begin{aligned}
&t_f(T_i) = t(RW_{j+BI_i-1}^{b_s+BI_i-1,BC_i-1}) \\
&= t(RW_{j+BI_i-1}^{b_s+BI_i-1,0}) + (BC_i - 1) \times tCCD \\
&= max\{t(ACT_{j+BI_i-1}^{b_s+BI_i-1}) + tRCD, \\
&\quad t(RW_{j+BI_i-2}^{b_s+BI_i-2,BC_i-1}) + tCCD\} + (BC_i - 1) \times tCCD \\
&= ... \\
&= max\{ \\
&\quad t(ACT_{j+l}^{b_s+l}) + tRCD + ((BI_i - l) \times BC_i - 1) \times tCCD, \\
&\quad t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{13}$$

□

A.2 Proof of Lemma 2

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. We assume T_i has BI_i and BC_i while T_{i-1} has BI_{i-1} and BC_{i-1}. The current bank access number is j and the starting bank of T_i is b_s. Firstly, the finishing time of T_{i-1} can be obtained from Eq. (8) and there is $t_f(T_{i-1}) = \hat{t}_s(T_i) - 1$. Secondly, the scheduling time of $ACT_{j+l}^{b_s+l}$ ($\forall l \in [0, BI_i - 1]$) is given by Eq. (3), which depends on $t(ACT_{j+l-1}^{b_s+l-1})$, $t(PRE_m^{b_s+l})$ and $t(ACT_{j+l-4}^{b_{j+l-4}})$. Let m be the latest access number for bank $b_s + l$. Therefore, we can get Eq. (14), where $\forall l' \in [l, BI_i - 1]$ by iteratively employing Eq. (3) and (13). According to Eq. (3), the scheduling of $ACT_{j+l}^{b_s+l}$ directly depends on $t(PRE_{j-1-\Delta l}^{b_s+l})$, $t(ACT_{j+l-4}^{b_{j+l-4}})$ and $ACT_{j+l-1}^{b_s+l-1}$. Therefore, other *ACT* commands that follow $ACT_{j+l}^{b_s+l}$ have a direct or indirect dependency on $ACT_{j+l}^{b_s+l}$. Therefore, all of them have an indirect dependency on $t(PRE_m^{b_s+l})$ and $t(ACT_{j+l-4}^{b_{j+l-4}})$. Eq. (14) illustrates this intuition.

$$\begin{aligned}
&t_f(T_i) = max\{ \\
&t(PRE_m^{b_s+l}) + tRP + (l' - l) \times tRRD + tRCD \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&t(ACT_{j+l-4}^{b_{j+l-4}}) + tFAW + (l' - l) \times tRRD + tRCD \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&t_s^w(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{14}$$

As Eq. (11) and (12) provide the worst-case scheduling time of the previous *ACT* and *PRE* commands, respectively, the worst-case finishing time $\hat{t}_f(T_i)$ can be obtained by using $\hat{t}(PRE_m^{b_s+l})$ and $\hat{t}(ACT_{j+l-4}^{b_{j+l-4}})$ according to Eq. (14). $\hat{t}(PRE_m^{b_s+l})$ is obtained directly from Eq. (12). For $l < 4$ ($BI_i \leq 4$), $\hat{t}(ACT_{j+l-4}^{b_{j+l-4}})$ is obtained directly from Eq. (11). If $BI_i > 4$, which induces $l \geq 4$, $\hat{t}(ACT_{j+l-4}^{b_{j+l-4}})$ can be achieved by Eq. (3), which is composed of $\hat{t}(PRE_n^{b_{j+l-4}})$ and $\hat{t}(ACT_{j+l-8}^{b_{j+l-8}})$. Let n be the latest access number to bank b_{j+l-4} and $\hat{t}(PRE_n^{b_{j+l-4}})$ is given by Eq. (12). Due to $l - 8 < 0$, $\hat{t}(ACT_{j+l-8}^{b_{j+l-8}})$ can be obtained from Eq. (11). Therefore, it is not difficult to get $\hat{t}_f(T_i)$ if T_i uses $BI_i > 4$. However, in order to simplify the expression of $\hat{t}_f(T_i)$ if T_i, it is described by Eq. (15), which only supports $BI_i \leq 4$.

$$\begin{aligned}
&\hat{t}_f(T_i) = max\{\\
&\hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times BC_{i-1} \times tCCD\\
&\quad + tRP + (l' - l) \times tRRD + tRCD\\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h),\\
&\hat{t}_s(T_i) - 1 - (BC_{i-1} - 1) \times tCCD + (l' - l) \times tRRD\\
&\quad + tFAW - \Delta l \times max\{tRRD, BC_{i-1} \times tCCD\}\\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h),\\
&\hat{t}_s(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{15}$$

Eq. (15) can be further simplified to Eq. (16). The reason is that we can observe that $tRWTP + tRP + tRCD > tFAW$ for DDR3 SDRAMs according to the JEDEC specification [15]. Eq. (16) shows the result of Lemma 2.

$$\begin{aligned}
&\hat{t}_f(T_i) = max\{\\
&\hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times BC_{i-1} \times tCCD\\
&\quad + tRP + (l' - l) \times tRRD + tRCD\\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h),\\
&\hat{t}_s(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{16}$$

□

A.3 Proof of Theorem 1

Proof. As shown in Lemma 2 (see Eq. (16)), the expressions in the $max\{\}$ of $\hat{t}_f(T_i)$ are: 1) linearly decreasing with BI_{i-1} (hidden in Δl) and BC_{i-1}, respectively, and 2) linearly increasing or decreasing with l and l'. Firstly, the maximum expression is obtained if $BI_{i-1} = BC_{i-1} = 1$. As a result, we can rewrite Eq. (16) and the simplified worst-case finishing time is given by Eq. (17). It indicates

the expressions in the $max\{\}$ of $\hat{t}_f(T_i)$ in Eq. (17) are linearly decreasing with l and increasing or decreasing with l' (determined by BC_i). Therefore, the maximum expression can be obtained only if $l = 0$ and $l' = 0$ or $l' = BI_i - 1$. Moreover, we assume there is always a command scheduling collision for the *ACT* commands except the first one. According to the timing constraints in JEDEC [15], there is $tSwitch < tRWTP + tRP + tRCD$ for all DDR3 SDRAM memories. Hence, Eq. (18) is obtained and shows the result of Theorem 1.

$$\begin{aligned}\hat{t}_f(T_i) =& \hat{t}_s(T_i) - 1 + max\{ \\ & tRWTP - l \times (tCCD + tRRD) + tRP + tRCD \\ & + l' \times (tRRD - BC_i \times tCCD) \\ & + (BI_i \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\ & tSwitch + (BI_i \times BC_i - 1) \times tCCD\}\end{aligned} \tag{17}$$

$$\begin{aligned}\hat{t}_f(T_i) =& max\{(BI_i \times BC_i - 1) \times tCCD, \\ & (BI_i - 1) \times (tRRD + 1) + (BC_i - 1) \times tCCD\} \\ & + \hat{t}_s(T_i) - 1 + tRWTP + tRP + tRCD\end{aligned} \tag{18}$$

□

A.4 Proof of Theorem 2

Proof. For transactions with fixed size, all of them have the same BI and BC, i.e. $BI_{i-1} = BI_i = BI$ and $BC_{i-1} = BC_i = BC$. Therefore, Lemma 2 (see Eq. (16)) is simplified and the worst-case finishing time of T_i is described by Eq. (19). Therefore, the maximum expression in the $max\{\}$ of $\hat{t}_f(T_i)$ can be obtained only if $l' - l = BI - 1$ or $l' - l = 0$. They require $l' = BI - 1$ and $l = 0$, or $l' = l = BI - 1$. In addition, we assume there is always a command scheduling collision for the *ACT* commands except the first one. Hence, Eq. (19) is further simplified to Eq. (20), which shows the result of Theorem 2.

$$\begin{aligned}\hat{t}_f(T_i) =& max\{tRWTP + tRP + tRCD \\ & + (l' - l) \times (tRRD - BC \times tCCD) \\ & + (BC - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\ & tSwitch + (BI \times BC - 1) \times tCCD\} + \hat{t}_s(T_i) - 1\end{aligned} \tag{19}$$

$$\begin{aligned}\hat{t}_f(T_i) = & max\{tRWTP + tRP + (BC - 1) \times tCCD \\ & + (BI - 1) \times (tRRD + 1 - BC \times tCCD) + tRCD, \\ & tRWTP + tRP + (BC - 1) \times tCCD + tRCD + 1, \\ & tSwitch + (BI \times BC - 1) \times tCCD\} + \hat{t}_s(T_i) - 1\end{aligned} \tag{20}$$

□

References

[1] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2011.

[2] Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 547–555. IEEE, 2009.

[3] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 3–14. IEEE, 2008.

[4] Samuel Bayliss and George A Constantinides. Methodology for designing statically scheduled application-specific sdram controllers using constrained local search. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 304–307. IEEE, 2009.

[5] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012.

[6] Karthik Chandrasekar, Benny Akesson, and Kees Goossens. Improved power modeling of ddr sdrams. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 99–108. IEEE, 2011.

[7] Karthik Chandrasekar et al. Drampower: Open-source dram power & energy estimation tool, 2012. http://www.drampower.info/.

[8] Hyojin Choi, Jongbok Lee, and Wonyong Sung. Memory access pattern-aware dram performance model for multi-core systems. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 66–75. IEEE, 2011.

[9] Dakshina Dasari, B Anderssom, Vincent Nelis, Stefan M Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1068–1075. IEEE, 2011.

[10] Yiqiang Ding, Lan Wu, and Wei Zhang. Bounding worst-case dram performance on multicore processors. *JCSE*, 7(1):53–66, 2013.

[11] 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.

[12] Ibrahim Hur and Calvin Lin. Memory scheduling for modern microprocessors. *ACM Transactions on Computer Systems (TOCS)*, 25(4):10, 2007.

[13] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 39–50. IEEE, 2008.

[14] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[15] JEDEC Solid State Technology Association. *DDR3 SDRAM Specification*, jesd79-3e edition, 2010.

[16] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling. *Micro, IEEE*, 31(1):78–89, 2011.

[17] Peter Kollig, Colin Osborne, and Tomas Henriksson. Heterogeneous multi-core platform for consumer multimedia applications. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 1254–1259. IEEE, 2009.

[18] Kun-Bin Lee, Tzu-Chieh Lin, and Chein-Wei Jen. An efficient quality-aware memory controller for multimedia platform soc. *Circuits and Systems for Video Technology, IEEE Transactions on*, 15(5):620–633, 2005.

[19] Marco Paolieri, Eduardo Quiñones, and Francisco J. Cazorla. Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–64:26, March 2013.

[20] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time cmps. *Embedded Systems Letters, IEEE*, 1(4):86–90, 2009.

[21] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–108. ACM, 2011.

[22] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the conference on design, automation and test in Europe*, pages 759–764. European Design and Automation Association, 2010.

[23] Hardik Shah, Andreas Raabe, and Alois 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. IEEE, 2012.

[24] CH Van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265. European Design and Automation Association, 2009.

[25] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005.

[26] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.