Dynamic Command Scheduling for Real-Time Memory Controllers

Yonghui Li[1], Benny Akesson[2] and Kees Goossens[1]
[1]Eindhoven University of Technology, [2]Czech Technical University in Prague

***Abstract*—Memory controller design is challenging as real-time embedded systems feature an increasing diversity of real-time and non-real-time applications with variable transaction sizes. To satisfy the requirements of the applications, tight bounds on the worst-case execution time (WCET) of memory transactions must be provided to real-time applications, while the lowest possible average execution time must be given to the rest. Existing real-time memory controllers cannot efficiently achieve this goal as they either bound the WCET by sacrificing the average execution time, or are not scalable to directly support variable transaction sizes, or both.**

In this paper, we propose to use dynamic command scheduling, which is capable of efficiently dealing with transactions with variable sizes. The three main contributions of this paper are: 1) a back-end architecture for a real-time memory controller with a dynamic command scheduling algorithm, 2) a formalization of the timings of the memory transactions for the proposed architecture and algorithm, and 3) two techniques to bound the WCET of transactions with both fixed and variable sizes, respectively. We experimentally evaluate the proposed memory controller and compare both the worst-case and average-case execution times of transactions to a state-of-the-art semi-static approach. The results demonstrate that dynamic command scheduling outperforms the semi-static approach by 33.4% in the average case and performs at least equally well in the worst case. We also show the WCET is tight for transactions with fixed and variable sizes, respectively.

I. Introduction

The complexity of real-time system design is growing as an increasingly diverse mix of real-time and non-real-time applications are integrated on the same platform. To provide the necessary computational power at reasonable power consumption, there is a trend towards heterogeneous multi-core systems where important functions are accelerated in hardware [1]–[3]. The diversity of applications and processing elements in such systems is reflected in the memory traffic going to the shared SDRAM, which features an irregular mix of transactions with variable sizes and heterogeneous requirements. For example, the memory clients in the NXP DTV SoC [4] have small and large transaction sizes, as well as different bandwidth and response time requirements. Memory transactions from real-time applications require tightly bounded worst-case execution times (WCET), while the transactions from non-real-time applications need the lowest possible average execution time to make the applications responsive. A particular challenge when bounding the WCET of memory transactions is that the bound depends on the *memory map configuration*, which is used to provide different trade-offs between bandwidth, execution time, and power consumption, by varying the number of banks that are used in parallel to serve a transaction [5].

Most existing memory controllers do not fully satisfy these requirements since they are not designed with real-time applications in mind and do not provide bounds on WCET of transactions [6]–[8]. Existing real-time memory controllers address real-time requirements by using either (semi-)static command scheduling, but do not provide low average execution time to memory traffic [9]–[11], or dynamic scheduling, as they are limited in architecture or analysis to a single transaction size and memory map configuration [12]–[15].

This paper addresses the problem of providing tight bounds on the execution time of real-time memory transactions, while providing low average execution time to non-real-time transactions in systems with variable transaction sizes and different memory map configurations. The three main contributions of this paper are: 1) A back-end architecture of a real-time memory controller with a dynamic 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), such as [16]. 2) A formalization of the timing behavior of the proposed dynamic command scheduler that captures all the SDRAM timing dependencies within and between banks. 3) The WCET for transactions with both fixed and variable sizes under different memory map configurations is derived in two ways based on the proposed formalism. The first one is analytical and is easy to use, but produces a slightly pessimistic WCET. The second way uses the formalism to derive the worst-case initial state of banks for a given transaction and then uses an off-line implementation of the scheduling algorithm to compute a tight WCET.

We experimentally evaluate the proposed architecture and analysis with fixed and variable transaction sizes, respectively. The results indicate that the analysis is valid, and the WCET of transactions is tightly bounded. Moreover, we show that our dynamic command scheduling outperforms a state-of-the-art semi-static approach [10] in the average case and performs equally well in the worst case. The results also show that small transactions benefit more from dynamic command scheduling than large ones.

In the remainder of this paper, Section II describes the related work. The background of SDRAM memories and real-time memory controllers is given in Section III. Section IV presents the back-end architecture and the dynamic command scheduling algorithm. In Section V, the timing behavior of transactions under our dynamic command scheduling is formalized. Section VI provides the WCET bound on the basis of the proposed formalism. Experimental results are presented in Section VII, before the paper is concluded in Section VIII.

II. Related Work

Analyzing the impact of using a shared memory on worst-case execution time of applications receives increasing attention in the real-time community, as multi-core systems challenge the traditional processor-centric view on systems. Most of this work focus on 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 obtained by assumptions or using measurements [17]–[19]. The work in this paper is complementary to this effort, as it focuses on the architecture and scheduling algorithm of an important part of that black box

(the memory controller back-end) and provides results that are required to derive that constant for different transaction sizes and memory map configurations.

Several types of real-time memory controller designs have been proposed in the past decade. Static [9] or semi-static [10], [11] controller designs are used to achieve bounded execution time. In [9], an application-specific static command schedule is constructed using a local search method. It requires a known static sequence of transactions, which is not available with multiple independent requestors. A semi-static method is proposed in [10] that generates static memory patterns [20], 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 handle variable transaction sizes as the patterns are generated to read or write a fixed amount of data. [11] presents a semi-static predictable DRAM controller that partitions banks or sets of banks into virtual private resources with independent repeatable actual timing behavior. However, since accesses to virtual resources must have constant duration, this controller is unable to provide competitive average execution times for transactions with variable sizes.

Dynamic command scheduling is used because it more flexibly copes with varied 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., [6]–[8]. 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. [12] propose an analytical model to bound the execution time of transactions under dynamic command scheduling on a modified version of the DRAMSim memory simulator [21], although the modifications to the original scheduling algorithm are not specified. In addition, the analytical model is limited to a fixed transaction size and a single memory map configuration. This also applies to [13], 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 [15], 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 time. However, the analysis is limited to a single transaction size and memory map configuration, and the assumption that the number of memory clients is not greater than the number of memory banks.

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 the provided analysis with respect to varied transaction sizes and memory map configurations, or both. To fill this gap, 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 memory map configurations, requiring more of them to be included in the model. Our analysis is supported by a formal framework in which the correctness of the results are proven.

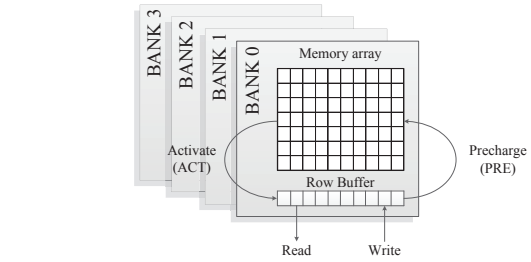
III. Background

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

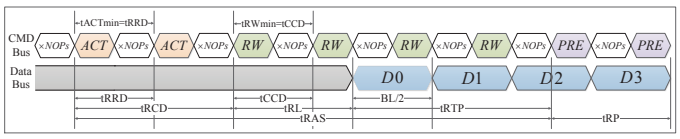
A. Introduction to SDRAM Memories

An SDRAM chip compromises a set of banks, which contains memory elements arranged in rows and columns [22], as shown in Fig. 1(a). Multiple such chips can be combined to form a DIMM with one or more ranks, although without loss of generality, this paper focuses on a single chip configuration, which is common in the embedded domain. The SDRAM interface consists of command, address, and data buses. A single command is transferred per clock cycle, while two data words are transferred per cycle by a contemporary DDR3 memory. To issue a command, several timing constraints have to be satisfied as specified by the JEDEC DDR3 standard [23]. Note that although this paper focuses on DDR3 SDRAMs, it requires only minor adaptations to work with other types of SDRAMs, such as DDRx, LPDDRx and Wide I/O.

To access a bank, the contents of the required row must be copied into the row buffer by issuing an *Activate* (*ACT*) command, which takes *tRCD* cycles. As a result, the required row is open. Then, a set of *Read* (*RD*) or *Write* (*WR*) commands are issued to the open row to transfer bursts of a programmed burst length (*BL*) (typically 8 words). The required data becomes available on the data bus after *tRL* or *tWL* cycles when issuing the *RD* or *WR* command, respectively. Before activating another row in the same bank, the current row must be closed by issuing a *Precharge (PRE)* command to write back the contents to the storage cells. 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. It is either issued via the command bus or by adding an auto-precharge flag to a *RD* or *WR* command, where precharging is automatically triggered when 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.



(a) SDRAM architecture



(b) An example of timing constraints

Fig. 1. SDRAM architecture and an example of timing constraints.

Similarly, multiple banks are accessed by scheduling a number of commands. For example, the accesses of two banks are illustrated in Figure 1(b). An *ACT* command is scheduled to open the required row in one bank and two consecutive *RD* or *WR* commands (*RW*) that have sequential addresses within the same row are scheduled to read or write data. The minimum time between two *RW* commands is *tCCD*. Finally, a *PRE* command is issued after the last access to the opened row of the bank. Another bank is accessed similarly. Particularly, its *ACT* command is issued earlier than the *RW* command to the first bank to exploit bank parallelism. Next, the four activate

window (FAW) and refresh constraints are independent of a particular bank. These timing constraints are summarized in Table I, where a 16-bit DDR3-800D memory device with a capacity of 2 Gb is taken as an example.

TABLE I. TIMING CONSTRAINTS FOR DDR3-800D SDRAM [23].

TC	*Description*	*Cycles*
tCK	Clock period	1
tRCD	Minimum time between *ACT* and *RD* or *WR* commands to the same bank	5
tRRD	Minimum time between *ACT* commands to different banks	4
tRAS	Minimum time between *ACT* and *PRE* commands to the same bank	15
tFAW	Window in which at most four banks may be activated	20
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	5
tRL	Read latency. Time after a *RD* command until first data is available on the bus	5
tRTP	Minimum time between a *RD* and a *PRE* command to the same bank	4
tRP	Precharge period time	5
tWTR	Internal *WR* command to *RD* command delay	4
tWR	Write recovery time. Minimum time after the last data has been written to a bank until a precharge may be issued	6
tRFC	Refresh period time	64
tREFI	Refresh interval	3120

B. 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 policies, such as TDM [24], Round Robin [12] or Credit-Controlled Static-Priority Arbitration [25], and sent to the back-end.

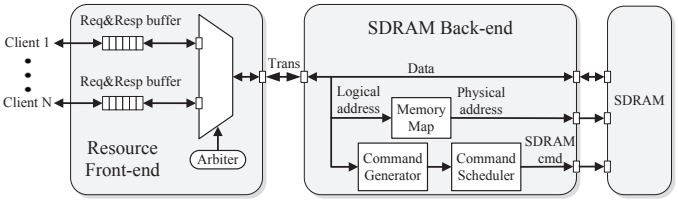


Fig. 2. A general real-time SDRAM controller supporting N clients.

In the back-end, the logical address of a transaction is translated into a 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: the *bank interleaving* number (*BI*) and the *burst count* (*BC*) [5]. *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 always constant for a given transaction size, since it corresponds to a fixed number of read or write bursts. The command generator produces the appropriate commands. Finally, the memory is accessed by the command scheduler issuing these commands subject to the timing constraints of the memory.

Real-time memory controllers [10]–[13] typically employ a close-page policy, under which 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. The execution time of a transaction is minimized if it requires access to a different row than the currently opened one. A close-page policy minimizes the worst-case execution time.

IV. DYNAMICALLY SCHEDULED BACK-END

This section presents our proposed dynamically scheduled memory controller back-end. The architecture of the back-end is described, followed by a specification of the dynamic scheduling algorithm. Here, we focus on the functional behavior of the architecture and later formalize the timing behavior in Section V.

A. Back-End Architecture

The back-end of a memory controller receives transactions scheduled by the front-end, which are read or write requests with different sizes. Each 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 transaction, which are needed by the command generation. This is implemented by means of a Lookup Table, as shown in Fig. 3, mapping each transaction size to a (*BI*, *BC*) pair. These are determined at design time when the memory map configuration is chosen and they 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 a larger size (nearest) is used with the additional data being masked out. A methodology to choose the memory map configuration based on the requirements of bandwidth, execution time and power consumption has been presented in [5]. Once the *BI* and *BC* are obtained, the Memory Map module in Fig. 3 translates the logical address of the transaction into the initial physical address, which consists of the starting bank b_s, row and column. The physical address for every subsequent command of the transaction can then be calculated based on the initial physical address. Then, (*BI*, *BC*, b_s) of the transaction is inserted into the parameter queue (ParaQueue). This queue keeps track of the order of transactions in the back-end and is used by the scheduling algorithm in Section IV-B.

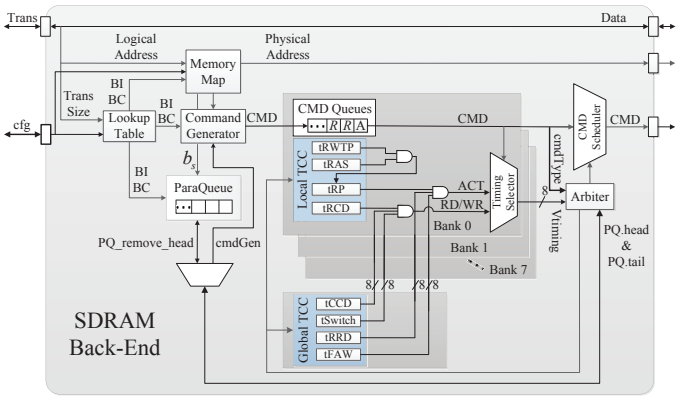


Fig. 3. Architecture of the dynamically scheduled SDRAM controller.

Based on (*BI*, *BC*) and the physical address, the Command Generator generates the memory commands to access all the required banks. *BI* determines the number of required *ACT* commands while *BC* determines the number of *RD* or *WR* commands per bank access. First, an *ACT* command is generated to open a row in a bank, followed by *BC* number of generated *RD* or *WR* commands to read or write data. In addition, an auto-precharge flag is attached to the last *RD* or *WR* command to trigger the closing of the opened row. These commands are sequentially inserted into the command queue corresponding to the bank. This is repeated for each bank accessed by the transaction. Note that the command generation

for a transaction cannot start until all the *ACT* commands of the previous transaction are no longer in the command queue, i.e., issued to the memory. On the same condition, the front-end sends a new transaction to the back-end, such that unnecessary transactions waiting in the front-end instead. This limits the size of the command queues, while provides pipelining between transactions.

To account the timing constraints of the commands, timing counters are used. They are initialized with the timing specifications given by the JEDEC DDR3 standard [23], and count down towards zero on every clock signal. As shown in Fig. 3, the timing constraint counters (TCC) include local TCC and global TCC. The local TCC considers *tRWTP*, *tRAS*, *tRP* and *tRCD* that constrain the command scheduling for a single bank. *tRP* counts the time for precharging and is reset according to *tRWTP* or *tRAS*, whichever is larger. The global TCC considers *tCCD*, *tRRD*, *tFAW* and *tSwitch*, which affect all banks. These timing constraints are all specified by JEDEC, except *tRWTP* and *tSwitch*, which are derived from the specification and are shown in Eq. (1) and (2), respectively. *tRWTP* is the time between a *RD* or *WR* command and the precharging to the same bank, while *tSwitch* gives 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 data associated with a *RD* or *WR* command.

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

$$tSwitch = \begin{cases} tRL + tCCD + 2tCK - tWL, & \text{WR } follows \text{ RD} \\ tWL + BL/2 + tWTR, & \text{RD } follows \text{ WR} \\ tCCD, & other \end{cases} \quad (2)$$

When a command of the transaction moves to the head of its command queue, the command (CMD) scheduler starts trying to schedule it while satisfying the timing constraints. As shown in Fig. 3, each Timing Selector has two inputs that represent whether the timing constraints for *ACT* and *RD* or *WR* are satisfied in the current clock cycle. An input is valid only if the timing constraints for scheduling a command are satisfied. The valid input is selected according to the command at the head of the queue. Multiple Timing Selectors corresponding to different banks may have a valid output at the same time when the timing constraints for head commands in their command queues are satisfied simultaneously, where these commands are called valid commands. This implies command scheduling collisions, since only one command can be issued per cycle. Therefore, it requires an arbiter to choose only one valid output of the Timing Selectors. In addition, the arbiter has to guarantee an in-order execution of transactions, which avoids the architectural and analysis cost of re-ordering. For this, it uses a scheduling algorithm presented in Section IV-B. Finally, the chosen command is removed from the head of its command queue and is passed to the memory. Meanwhile, 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 Fig. 3. Additionally, a refresh command needs to be scheduled every *tREFI* cycles. Once triggered, it is scheduled after the data transmission of the executing transaction to prevent unnecessary interference, while still ensuring that no refresh command is delayed more than $9 \times tREFI$ [23]. Refresh is also implemented by timing counters, which are not depicted in Fig. 3 for simplicity.

B. Scheduling Algorithm

The scheduling algorithm in the back-end decides how a command is chosen according to the inputs of the arbiter. It has to solve three critical issues, namely: 1) a single command must be chosen from a set of multiple valid commands; 2) transactions must be executed in first-come-first-served (FCFS) order to avoid reorder buffers; 3) to simplify logical-to-physical address translation [5], 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 scheduler. To guarantee the FCFS, the valid commands of a transaction have higher priority than those 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. Within a transaction, the head of a command queue corresponding to a bank with a lower id has higher priority, forcing banks to be accessed in ascending order.

The priorities are used to select a command from the multiple valid commands in every cycle. As shown in Fig. 3, the inputs of the arbiter include the outputs of the Timing Selectors, the types (*ACT*, *RD* or *WR*) of each command at the head of the command queues, and the head and tail elements of the ParaQueue. These inputs are taken by Algorithm 1 and represented by Vtiming, cmdType and PQ.head and PQ.tail, respectively. The dimensions of Vtiming and cmdType are equal to the number of banks (and hence command queues) in the memory. For an arbitrary command at the head of command queue i, cmdType[i] contains its type and Vtiming[i] (valid or invalid) determines whether or not the timing constraints of the command are satisfied. The outputs of Algorithm 1 are BankID, PQ_remove_head and cmdGen, where BankID indicates the command queue whose head command can be scheduled to bank BankID, and PQ_remove_head (true or false) and cmdGen (true or false) decide whether to remove the head of the ParaQueue and trigger command generation for the next transaction.

Algorithm 1 Dynamic command scheduling

```
1:  Inputs: PQ, Vtiming, cmdType
2:  Internal state: bankRW, bankAct
3:  Initialization: BankID ← null; cmdGen ← true;
4:                  PQ_remove_head ← false;
5:  if bankAct = null then bankAct ← PQ.tail.b_s; cmdGen ← false;
6:  if bankRW = null then bankRW ← PQ.head.b_s;
7:  if cmdType[bankRW] = RD/WR and
8:       Vtiming[bankRW] = valid then
9:     BankID ← bankRW;
10:    if last RD/WR of PQ.head transaction then
11:       bankRW ← null;
12:       PQ_remove_head ← true;
13:    else if last RD/WR of PQ.head transaction to bank BankID
14:       then bankRW ← bankRW+1;
15: else if bankAct != null
16:    if cmdType[bankAct] = ACT and
17:       Vtiming[bankAct] = valid then
18:       BankID ← bankAct;
19:       if last ACT of PQ.tail transaction then
20:          bankAct ← null; cmdGen ← true;
21:       else bankAct ← bankAct+1;
22: Output: BankID, PQ_remove_head, cmdGen
```

In Algorithm 1, a *RD*/*WR* command is checked whether it is valid to be scheduled (line 8, 9). Otherwise, an *ACT* command is checked (line 17, 18). In this way, it guarantees a valid *RD* or *WR* command has higher priority than a valid *ACT*

command. Two extra variables *bankAct* and *bankRW* assist in achieving the priorities. They provide the identifier of the banks that can accept an *ACT* or a *RD*/*WR* command, respectively. *bankAct* is increased by one after an *ACT* command is selected (line 22), while *bankRW* increases by one only if *BC* number of *RD*/*WR* commands of the current transaction are scheduled to bank *bankRW* (line 12). This update scheme ensures the banks are accessed in ascending order for each transaction. *bankAct* and *bankRW* are initialized with the b_s of the tail and head of PQ, respectively (line 5, 6). Note that command generation starts for a new transaction only if all the *ACT* commands of previous transactions have been issued (line 21). As a result, only the transaction associated with PQ.tail has *ACT* commands in the command queues, and is used for initializing *bankAct*. Transactions are hence served in FCFS order and their corresponding banks are accessed in ascending order, while priorities ensure that only a single command is scheduled per cycle. Algorithm 1 thus addresses all three critical issues mentioned above. Though command priorities are used, there is no livelock or starvation since transactions are executed in order.

V. Formalization of 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 to calculate 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 III as shown in the appendix.

A. Timing dependencies

In dynamic command scheduling, commands are scheduled sequentially subject 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 bank access is tracked by an increasing variable j ($j \geq 0$) that is the bank access number. The scheduling of a command depends on the previous commands, which are specified by the input arrows. The parameters near the arrows specify the number of cycles that the following command has to wait until the timing constraints are satisfied. For example, the timing constraints to schedule an *ACT* command include *tRRD*, *tRP* and *tFAW*, previously described in Table I. Therefore, the block of an *ACT* command (see Fig. 4) has three input arrows that represent the corresponding timing constraints. The scheduling of a *RD* or *WR* command has to satisfy the timing constraints *tRCD* and *tSwitch* for the first *RD* or *WR* command of the bank access. However, the following *RD* or *WR* commands of the bank access only takes the timing constraint *tCCD* into account. Finally, an auto-precharge must consider the timing constraints *tRAS* and *tRWTP*. The timing dependencies among the commands are illustrated in Fig. 4. Note that refresh commands are not depicted because their impact on WCET can be easily analyzed, as presented in Section VI. Moreover, the effect of *REF* is negligible in memory interference delay [26], and it is not a main concern in this paper.

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 by one cycle. The collision is depicted by the filled circle in Fig. 4. The arrow corresponding to the maximum time dominates the scheduling of a dependent command, since all relevant timing constraints must be satisfied. The *PRE* in Fig. 4 does not use the command bus due to the auto-precharge policy, which cannot cause a command collision. However, the time at which the auto-precharge actually happens is necessary to determine when the bank can be reactivated.

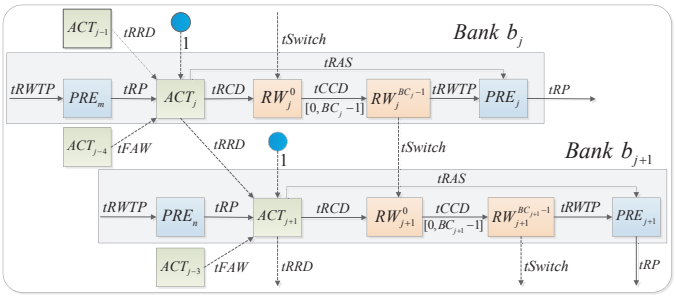


Fig. 4. Timing dependencies between successive bank accesses.

B. Formalization

Having explained the dependencies between commands in a bank access according to the DDR3 standard and illustrated them in Fig. 4, we analyze the execution time of a transaction by computing the actual scheduling times of commands under our dynamic scheduling algorithm. The worst-case execution time is later computed in Section VI.

An arbitrary transaction T_i ($\forall i > 0$) arrives at the interface of the back end at time $t_a(T_i)$ which is defined by Definition 1. We assume T_i uses BI_i and BC_i, and the starting bank is b_s. A transaction is executed by scheduling commands to a number of banks. The bank access number j of accessing b_s is given by Eq. (3). It is the total number of bank accesses by previous transactions.

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.*

$$j = \sum_{k=0}^{i-1} BI_k \tag{3}$$

Based on Fig. 4, the timing dependencies of command scheduling for T_i is illustrated in Fig. 5. It indicates the command scheduling for T_i depends on that for the previous transaction (s) $T_{i'} (i' \leq i)$. For $\forall l \in [0, BI_i - 1]$, the $(j+l)^{th}$ bank access is implemented by scheduling an ACT_{j+l} and several *RD* or *WR* commands to bank $b_{j+l} = b_s + l$. 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 implemented 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 T_i (not T_{i-1}) because of the four-activate window (*tFAW*). As shown in Fig. 5, $i' = i$ if and only if $BI_i > 4$. Definition 2 defines the finishing time of T_i as the time when the last *RD* or *WR* command $RW^{BC_i-1}_{j+BI_i-1}$ is scheduled to $b_s + BI_i - 1$. The starting time of T_i is defined as the earliest time at which the scheduler tries to schedule its commands. This is either one cycle after the finishing time of the previous transaction T_{i-1} or two cycles after the arrival time (pipeline stages for the Lookup Table and Command Generation in Fig. 3), 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 by Definition 4.

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

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 $t_{ET}(T_i) = t_f(T_i) - t_s(T_i) + 1$.*

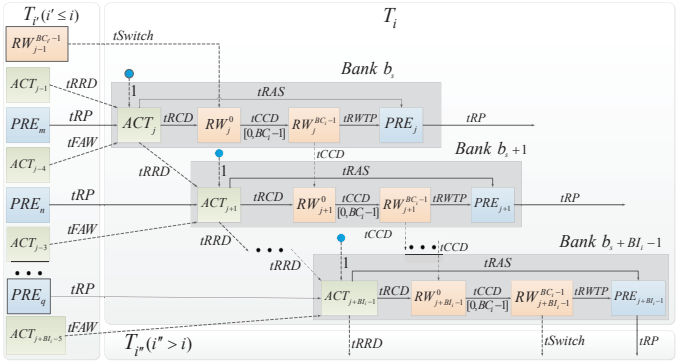


Fig. 5. The timing dependencies of command scheduling for transaction T_i.

For T_i, Eq. (4) computes the scheduling time of ACT_{j+l} where m ($m < j$) is the latest bank access number to bank b_{j+l}, i.e., $b_m = b_{j+l}$. The *max* function guarantees that all the timing constraints for scheduling ACT_{j+l} are satisfied. In addition, the scheduling time of ACT_{j+l} is at least 2 cycles after T_i arrives, which are consumed by looking up table and command generation. In case of a command scheduling collision when 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_{j+l}^k is given by Eq. (5) and (6). Eq. (5) provides the scheduling time of the first *RD* or *WR* command to bank b_{j+l}. It depends on $t(RW_{j+l-1}^{BC_i-1})$, 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 $l = 0$, $t(RW_{j-1}^{BC_i-1})$ represents the finishing time of T_{i-1} though it does not use BC_i. The scheduling time of the the remaining *RD* or *WR* commands to bank b_{j+l} only depends on the previous *RD* or *WR* command, and is given by Eq. (6). Finally, the precharging time of the auto-precharge for bank b_{j+l} is given by Eq. (7). 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. For initialization, we assume the transaction T_0 finished long time ago, e.g., $t_f(T_0) = -\infty$, such that the *ACT* of the first transaction T_1 is scheduled at 0.

$$\begin{aligned} t(ACT_{j+l}) = max\{&t(ACT_{j+l-1}) + tRRD,\\ &t(PRE_m) + tRP, t_a(T_i) + 2\\ &t(ACT_{j+l-4}) + tFAW\} + C(j+l) \end{aligned} \tag{4}$$

$$\begin{aligned} t(RW_{j+l}^0) = max\{&t(RW_{j+l-1}^{BC_i-1}) + tSwitch,\\ &t(ACT_{j+l}) + tRCD\} \end{aligned} \tag{5}$$

$$t(RW_{j+l}^k) = t(RW_{j+l}^0) + k \times tCCD \tag{6}$$

$$\begin{aligned} t(PRE_{j+l}) = max\{&t(ACT_{j+l}) + tRAS,\\ &t(RW_{j+l}^{BC_i-1}) + tRWTP\} \end{aligned} \tag{7}$$

Based on Eq. (4) 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. 5, 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 timings. This intuition is formalized by Lemma 1 and the proof is included in the appendix.

Lemma 1. *For $\forall i > 0$ and j as given by Eq.* (3)*,*

$$\begin{aligned} t_f(T_i) = \max_{0 \le l \le BI_i-1} \{ &\\ &t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD,\\ &t(ACT_{j+l}) + tRCD + ((BI_i - l) \times BC_i - 1) \times tCCD\} \end{aligned}$$

VI. Worst-case Execution Time

This section analyzes the worst-case execution time of our proposed dynamic command scheduling algorithm for an arbitrary transaction T_i. First, 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. Next, the worst-case finishing time of T_i is computed based on the worst-case situation, the transaction size and the memory map configuration.

A. 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. (4) to (7) in the following paragraphs.

Definition 4 states 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 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 arrived before the finishing of T_{i-1}. 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. (8), where the current bank access number is j, and T_{i-1} has BC_{i-1}.

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

In order to get the worst-case finishing time of T_i, the scheduling time of its *ACT* commands should be maximized according to Lemma 1. According to Eq. (4), the scheduling of an *ACT* command for T_i depends on the previous *PRE* to the same bank, the previous *ACT* commands and the possible collisions between an *ACT* command and a *RD* or *WR* command. 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. The worst-case starting time given by Eq. (8) defines the finishing time $t(RW_{j-1}^{BC_{j-1}-1})$ of T_{i-1}. Since we do not know exactly how T_{i-1} was scheduled, we conservatively assume it was scheduled *as late as possible* (*ALAP*) subject to the timing constraints, ensuring the maximum scheduling time of the previous commands.

According to *ALAP* scheduling, the scheduling time of the previous *ACT*, *RD* or *WR* commands and *PRE* can be obtained by calculating backwards from $t(RW_{j-1}^{BC_{i-1}-1})$. It 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 between any successive commands must be minimum while satisfying the timing constraints, thereby ensuring an *ALAP* schedule of the previous commands. T_{i-1} has BI_{i-1} and BC_{i-1}. As stated in Table I, the minimum time between two *RD* or *WR* commands is *tCCD*. Since *RD* or *WR* commands targeting the same bank are scheduled sequentially, the time 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 I). 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 between two successive *ACT* commands is at least $BC_{i-1} \times tCCD$ cycles. In addition, Table I also states that the minimum time between two *ACT* commands to different bank is *tRRD*. Hence, for *ALAP* scheduling, the minimum time interval between two successive *ACT* commands to different banks is $max\{tRRD, BC_{i-1} \times tCCD\}$.

The minimum time interval between the first *RD* or *WR* commands to consecutive banks can be analyzed accurately considering the cases of fixed or varied transaction sizes separately. As shown in Fig. 4, the scheduling of the first *RD* or *WR* command to a bank depends on the *ACT* command to the same bank and the final *RD* or *WR* command from the previous bank access. For a fixed transaction size, the *ACT* command can dominate in determining the scheduling of the first *RD* or *WR* command to the same bank. For example, if *BC* is 1 for all the transactions, the minimum time interval between the first *RD* or *WR* command to consecutive banks is *tRRD* instead of $BC_{i-1} \times tCCD$. The reason is that *tRRD* is equal to or larger than *tCCD* for DDR3 memories according to JEDEC standard [23]. However, it is difficult to decide the dominance for *RD* or *WR* command scheduling with varied transaction sizes because different *BI* and *BC* are used. Hence, Eq. (9) gives the minimum time interval *RWInterval* between the first *RD* or *WR* commands to consecutive banks for transactions with fixed size and varied sizes, respectively.

$$RWInterval = \begin{cases} max(tRRD, BC_{i-1} \times tCCD), \text{fixed size} \\ BC_{i-1} \times tCCD, \text{varied sizes} \end{cases} \tag{9}$$

Fig. 6 illustrates an example of *ALAP* scheduling for a DDR3-800D SDRAM. This example assumes the current transaction T_i has $BI_i = 4$ 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. j is the current bank access number. 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. 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 later scheduling times of the previous *ACT* and *WR* commands are achieved. Fig. 6 shows the scheduling time of the previous commands that are scheduled to the banks 0 and 1 required by T_{i-1} and the banks 2 and 3 required by even earlier transactions, e.g., T_{i-2} and T_{i-3}. Since *Bank* 2 is first accessed and then *Bank* 3 for T_i, the scheduling time of the previous commands to *Bank* 2 is computed backwards first. It can be recognized as two small transactions T_{i-2} and T_{i-3}, which require only *Bank* 2 and *Bank* 3, respectively. Hence, the computed scheduling time of the previous commands guarantees a conservative execution time bound for later transactions.

ALAP scheduling is formalized to provide the scheduling time 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}. First, 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. Second, in worst-case, 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 that 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. (10). In case $BI_{i-1} < BI_i$, Eq. (11) is used to compute the scheduling time of *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}, e.g., *Bank* 2 and 3 in Fig. 6.

$$\begin{aligned} \hat{t}(RW^k_{j-1-\Delta l}) =& \hat{t}_s(T_i) - 1 - (BC_{i-1} - 1 - k) \times tCCD \\ & - \Delta l \times RWInterval \end{aligned} \tag{10}$$

$$\Delta l = \begin{cases} BI_{i-1} - 1 - l, & l \le BI_{i-1} - 1 \\ l, & otherwise \end{cases} \tag{11}$$

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*. Thus, with the minimum time interval between *ACT* commands, the scheduling time of the previous four *ACT* commands is calculated by Eq. (12). Based on Eq. (7), the time of the previous *PRE* is obtained by using the worst-case scheduling time for *RD* or *WR* and *ACT* commands from Eq. (10) and (12), respectively. It is given by Eq. (13) based on the observations of the timing constraints in JEDEC DDR3 standard [23] 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.

$$\begin{aligned} \hat{t}(ACT_{j-1-\Delta l}) =& \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ & - \Delta l \times max\{tRRD, BC_{i-1} \times tCCD\} \end{aligned} \tag{12}$$

$$\begin{aligned} \hat{t}(PRE_{j-1-\Delta l}) =& max\{\hat{t}(ACT_{j-1-\Delta l}) + tRAS, \\ & \hat{t}(RW^{BC_{i-1}-1}_{j-1-\Delta l}) + tRWTP\} \\ =& \hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times RWInterval \end{aligned} \tag{13}$$

Hence, the worst-case situation for T_i is that T_{i-1} is a write transaction and the scheduling time of its *WR*/*ACT*/*PRE* command is given by Eq. (10), (12) and (13), 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.

B. Analytical Worst-Case Finishing Time

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 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 $\hat{t}(ACT_{j+l})$. By fixing the worst-case starting time of T_i, $\hat{t}_f(T_{i-1})$ is obtained by Eq. (8), as is $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$. Regarding $\hat{t}(ACT_{j+l})$, it can be expressed by the worst-case scheduling time of the previous *ACT* commands and the *PRE* commands, given by Eq. (12)

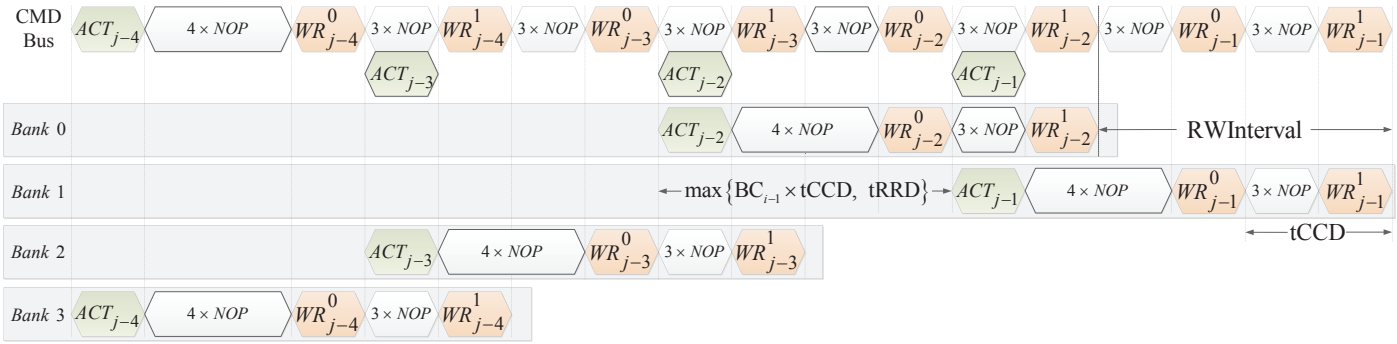


Fig. 6. An example of As-Late-As-Possible (ALAP) scheduling with DDR3-800D SDRAM for T_i that has $BI_i = 4$ and $BC_i = 2$. The previous transaction T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. The starting bank for both T_{i-1} and T_i is *Bank* 0. T_{i-2} and T_{i-3} have $BI_{i-2} = BI_{i-3} = 1$ and $BC_{i-2} = BC_{i-3} = 2$, while their starting bank is *Bank* 2 and *Bank* 3, respectively.

and (13), respectively. Eq. (4) indicates that $t(ACT_{j+l})$ is determined by $t(ACT_{j+l-1})$, $t(PRE_{j-(BI_i-l)})$ and $t(ACT_{j+l-4})$. As a result, $\hat{t}(ACT_{j+l})$ can be obtained by using $\hat{t}(PRE_{j-(BI_i-l)})$ and $\hat{t}(ACT_{j+l-4})$. In addition, $\hat{t}(ACT_{j+l})$ can be iteratively expressed by $\hat{t}(PRE_{j-1-(BI_i-l)})$ and $\hat{t}(ACT_{j+l-5})$ because they determine $\hat{t}(ACT_{j+l-1})$ according to Eq. (4). Eq. (13) provides $\hat{t}(PRE_{j-(BI_i-l)})$, while Eq. (12) provides $\hat{t}(ACT_{j+l-4})$ if $l < 4$. In order to simplify the expression, Lemma 2 gives $\hat{t}_f(T_i)$ with $BI_i \leq 4$ (ensuring $l < 4$) and the assumption that T_{i-1} is a write transaction. The proof is presented in the appendix. However, it is not difficult to extend Lemma 2 to support $BI_i > 4$.

Lemma 2. *For $\forall i > 0$ and Δl as given by Eq.* (11)*,*

$$\begin{aligned}\hat{t}_f(T_i) = \max_{0\leq l\leq l'\leq BI_{i-1}} \{ & \\ & \hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times RWInterval \\ & + tRP + (l' - l) \times tRRD + tRCD \\ & + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\ & \hat{t}_s(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}\end{aligned}$$

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 $l' = BI_i - 1$. Consequently, we can use Theorem 1 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. Intuitively, it indicates the worst-case situation for T_i is that its starting bank ($l = 0$) is the finishing bank of the previous write transaction T_{i-1}. As a result, there is no collision for the first *ACT* of T_i. The proof is given in the appendix.

Theorem 1 (Variable transaction size)**.** *T_{i-1} is write rather than read. $BI_i \leq 4$.*

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

Theorem 1 provides too pessimistic worst-case execution time bound for systems when 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, of which the proof is shown in the appendix.

Theorem 2 (Fixed transaction size)**.** *T_{i-1} is write rather than read. $BI_i \leq 4$.*

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

Note that Theorem 1 and 2 are based on the previous write transaction T_{i-1}. However, if T_{i-1} is read, they give conservative $\hat{t}_f(T_i)$. Moreover, they are also easy to be extended for $BI_i > 4$ as discussed previously. 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.

C. Scheduled Worst-Case Finishing Time

The bounds derived in Section VI-B 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 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 scheduling as presented in Section VI-A, followed by actually scheduling the commands of the transaction. 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 the collisions in case the scheduling of the *ACT* commands is a bottleneck. 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 V, has been implemented as an open-source off-line scheduling tool [27]. For the remainder of this paper, we will refer to this approach as the *scheduled WCET* and the bounds obtained in Section VI-B as the *analytical WCET*. Both of them can be obtained from our tool.

We have now presented two techniques to compute the worst-case execution time of transactions with the proposed dynamically scheduled back-end. Note that refresh command cannot be scheduled until the current transaction has finished and is hence not included in the execution time. However, it requires at most $tRWTP + tRP + tRFC$ cycles to complete. When the back-end is combined with a predictable memory controller front-end, such as [16], this result can be used to obtain the total response time of transactions under a particular transaction scheduling policy.

VII. Experimental Results

This section experimentally evaluates our dynamically scheduled back-end and its corresponding analysis. A front-end based on round-robin is used. However, our back-end is flexible for front-end with any scheduling policies. The experimental setup is presented, followed by three experiments. The first one shows that the formalization accurately describes the timing behavior of the back-end. The last two experiments show the WCET and average execution time results of transactions with fixed size and variable sizes, respectively, and compare our approach to a state-of-the-art approach.

A. Experimental Setup

The dynamically scheduled back-end is implemented as a cycle-accurate SystemC model. To provide predictable memory access to multiple clients, the memory controller front-end in [16] is used, fitted with a round robin arbiter as the transaction scheduler. The experiments use a combination of real application traces and synthetic traffic. The application traces are generated by running applications from the MediaBench benchmark suite [28] on the SimpleScalar 3.0 processor simulator [29] using separate data and instruction caches, each with a size of 16 KB. The L2 cache is a unified 128 KB cache where the cache-line size varies depending on the experiment. 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. The configured (*BI*, *BC*) for transaction sizes of 16 B, 32 B, 64 B and 128 B are hence (1, 1), (2, 1), (4, 1) and (4, 2), respectively [5]. 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 [23]. Due to space limitations, this section focuses on results for DDR3-800D, making the presented results conservative, as benefits of dynamic command scheduling are larger for faster memories. However, the WCET results of all these DDR3 memories with fixed and varied transaction sizes are summarized in Table IV and V, respectively, available in the appendix.

B. Experimental Validation of the Formalization

The purpose of our first experiment is to validate the formalization of the timing behavior of the dynamically scheduled controller by verifying that the scheduling time of each command is equivalent to the SystemC implementation. To this end, the open-source off-line scheduling tool [27] that implements the formalism has been provided with the same inputs as the SystemC implementation for all experiments in this paper, covering a wide range of read and write transactions with different transaction sizes and inter-arrival times under different memory map configurations. The results of this experiment is 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.

C. Fixed Transaction Size

This experiment evaluates our approach for transactions with fixed transactions size. Four memory clients are used, corresponding to four processors executing different Mediabench applications (*gsmdecode*, *epic*, *unepic* and *jpegencode*). For each application, the total number of transactions (*TransN*) and the ratio of read transactions (*RRatio*) are illustrated in Table II. The processors execute through a partitioned L2 cache and thus have the same cache-line size, enabling Theorem 2 to be used to bound the WCET. The experiment is executed for three different cache-line sizes of 32 B, 64 B and 128 B with different memory map configurations, respectively. The results are compared to the semi-static approach in [10], the only other approach that supports different memory map configurations.

TABLE II. Characterization of memory traffic.

Size	*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%

Fig. 7 illustrates the WCET for the DDR3-800D SDRAM using different fixed cache-line sizes. We can observe that: *1)* the maximum measured WCET from the experiments is equal to the scheduled WCET bound. This indicates that the proposed formalization provides an exact WCET bound; *2)* the scheduled WCET is never larger than the bound provided by the semi-static approach. This suggests our dynamic command scheduling performs at least as well as the semi-static scheduling in the worst case; *3)* the analytical WCET given by Theorem 2 is equal to or slightly larger than the scheduled WCET. This difference is because Theorem 2 conservatively assumes that every *ACT* command results in a command collision, which may not actually be the case and some (all) of the collisions may not result in an increased execution time, because the *ACT* command is not always dominating in the computation of the finishing time (see Lemma 1). However, the maximum difference is *BI* cycles.

Fig. 7. WCET and average ET of dynamic command scheduling and semi-static scheduling for fixed transaction size.

The average ET of dynamic command scheduling and semi-static scheduling are also shown in Fig. 7. For each transaction size, dynamic command scheduling achieves lower average ET. This is because dynamic command scheduling monitors the actual state of the required banks and can issue commands earlier for a transaction that requires a different set of banks from that of the previous transaction. Semi-static scheduling [10] uses pre-computed schedules that assume

worst-case initial bank state for every transaction. Fig. 7 also shows that smaller transactions benefit more from dynamic command scheduling. For example, 32 B transactions gain 33.4% while 128 B transactions only gain 2.3%. The reason is that smaller transactions require fewer banks, which increases the probability that the following transaction accesses an independent set of banks and can thus be scheduled earlier.

D. Varied Transaction Size

The last experiment evaluates our approach with variable transaction sizes. The setup is loosely inspired by a High-Definition video and graphics processing system featuring a CPU, hardware accelerators and peripherals with variable transaction sizes. The CPU executes the *jpegdecode* application from MediaBench through a cache with a 64 B cache-line size, while the other components are represented by synthetic traffic generators with transaction sizes of 16 B, 32 B and 128 B, respectively. The total number of transactions is 153963 and the read ratio is 50% for the synthetic traffic and 62.9% for the *jpegdecode* trace. Theorem 1 is used to provide the analytical WCET. Note that we cannot fairly compare our approach to other approaches in this experiment, as no other memory controller provides an analysis that supports variable transaction sizes and different memory map configurations.

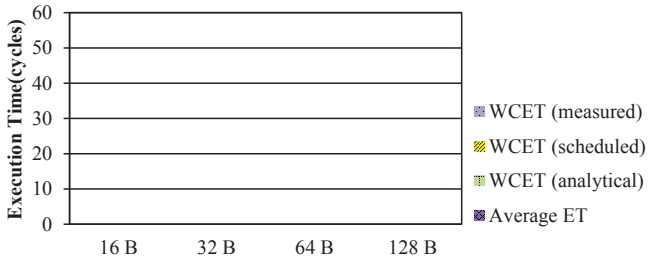


Fig. 8. WCET and average ET for variable transaction sizes.

Fig. 8 illustrates the WCET of dynamic command scheduling with variable transactions sizes. As shown, the measured WCET is equal to the scheduled WCET, implying that the bound is exact. The analytical WCET is again slightly pessimistic as it assumes a command collision for every *ACT* command. However, the maximum difference is *BI* cycles. The average execution time for each transaction size corresponding to each memory client is also shown in Fig. 8.

VIII. Conclusions

This paper addresses the problem of providing tight WCET bounds for memory transactions to real-time memory clients, while offering competitive average execution time to the rest. To this end, an architecture and analysis for a dynamically scheduled memory controller back-end that supports transactions with both fixed and variable sizes, as well as different memory map configurations is proposed. Based on the analysis, two techniques are presented to bound the WCET. The first technique is a simple equation that computes the WCET given the transaction size and memory map configuration, while the second technique tries to provide a tighter bound by using an offline implementation of the scheduler to resolve command collisions. Experimental results show that the first and simpler method over-estimates the WCET by a few clock cycles, while the bounds computed using the second method are perfectly tight. Comparison with a state-of-the-art semi-static scheduling approach shows that our approach performs equally well in the worst case, but outperforms it dramatically in the average case.

Acknowledgments

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, CA505 BENEFIC, CA703 OpenES, ARTEMIS-2013-1 621429 EMC2 and 621353 DEWI, and the European social fund CZ.1.07/2.3.00/30.0034.

References

[1] P. Kollig *et al.*, "Heterogeneous multi-core platform for consumer multimedia applications," in *Proc. DATE*, 2009.

[2] C. van Berkel, "Multi-core for mobile phones," in *Proc. DATE*, 2009.

[3] L. Benini *et al.*, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. DATE*, 2012.

[4] P. Kollig *et al.*, "Heterogeneous multi-core platform for consumer multimedia applications," in *Proc. DATE*, 2009.

[5] S. Goossens *et al.*, "Memory-map selection for firm real-time memory controllers," in *Proc. DATE*, 2012.

[6] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proc. ISCA*, 2008.

[7] K. Yoongu *et al.*, "Thread cluster memory scheduling," *Micro, IEEE*, vol. 31, no. 1, 2011.

[8] I. Hur *et al.*, "Memory scheduling for modern microprocessors," *ACM Trans. Comput. Syst.*, vol. 25, no. 4, 2007.

[9] S. Bayliss *et al.*, "Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search," in *Proc. FPT*, 2009.

[10] B. Akesson *et al.*, "Architectures and modeling of predictable memory controllers for improved system integration," in *Proc. DATE*, 2011.

[11] J. Reineke *et al.*, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proc. CODES+ISSS*, 2011.

[12] M. Paolieri *et al.*, "Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1, 2013.

[13] H. Shah *et al.*, "Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs," in *Proc. DATE*, 2012.

[14] H. Choi *et al.*, "Memory access pattern-aware DRAM performance model for multi-core systems," in *Proc. ISPASS*, 2011.

[15] Z. P. Wu *et al.*, "Worst case analysis of DRAM latency in multi-requestor systems," in *Proc. RTSS*, 2013.

[16] B. Akesson *et al.*, "Composable resource sharing based on latency-rate servers," in *Proc. DSD*, 2009.

[17] H. Yun *et al.*, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Proc. ECRTS*, 2012.

[18] D. Dasari *et al.*, "Response time analysis of COTS-based multicores considering the contention on the shared memory bus," in *Proc. TrustCom*, 2011.

[19] S. Schliecker *et al.*, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *Proc. DATE*, 2010.

[20] B. Akesson *et al.*, "Classification and analysis of predictable memory patterns," in *Proc. RTCSA*, 2010.

[21] D. Wang *et al.*, "DRAMsim: a memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, 2005.

[22] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[23] *DDR3 SDRAM Specification*, Jesd79-3e ed., JEDEC Solid State Technology Association, 2010.

[24] S. Goossens *et al.*, "Conservative open-page policy for mixed time-criticality memory controllers," in *Proc. DATE*, 2013.

[25] B. Akesson *et al.*, "Real-time scheduling using credit-controlled static-priority arbitration," in *Proc. RTCSA*, 2008.

[26] H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *Proc. RTAS*, 2014.

[27] Y. Li *et al.*, "RTMemController: Open-source WCET and ACET analysis tool for real-time memory controllers." http://www.es.ele.tue.nl/rtmemcontroller/, 2014.

[28] C. Lee *et al.*, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proc. MICRO*, 1997.

[29] T. Austin *et al.*, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, 2002.

APPENDIX

TABLE III. SUMMARY OF NOTATION.

Variables	*Descriptions*
T_i	The i^{th} transaction received by the back-end
j	The number of the current bank access in the back-end
BI_i, BC_i	The bank interleaving number (*BI*) and burst count (*BC*) used by T_i
b_j	The bank number that is targeted by the j^{th} bank access
ACT_j	The *ACT* command for the j^{th} bank access targeting bank b_j
$t(ACT_j)$	The scheduling time of ACT_j
$C(j)$	The extra cycle for scheduling ACT_j because of a collision
RW_j^k	The k^{th} ($\forall k \in [0, BC_i]$) *RD* or *WR* command of the j^{th} bank access targeting bank b_j
$t(RW_j^k)$	The scheduling time of RW_j^k
PRE_j	The *PRE* command for the j^{th} bank access targeting bank b_j
$t(PRE_j)$	The precharge time of PRE_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

A. Proof of Lemma 1

Proof: With the definition of finishing time (Definition 2), $t_f(T_i)$ can be obtained from Eq. (6), which provides the scheduling time of the last *RD* or *WR* command. Eq. (6) 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 final *RD* or *WR* command of the previous transaction.

Eq. (5) 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. (5) instead of *tSwitch*.

For $\forall l \in [0, BI_i - 1]$, $t_f(T_i)$ is expressed by Eq. (14), which is obtained by iteratively using Eq. (5) and (6). 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.

$$\begin{aligned}
&t_f(T_i) = t(RW_{j+BI_i-1}^{BC_i-1}) \\
&= t(RW_{j+BI_i-1}^{0}) + (BC_i - 1) \times tCCD \\
&= max\{t(ACT_{j+BI_i-1}) + tRCD, t(RW_{j+BI_i-2}^{BC_i-1}) + tCCD\} \\
&\quad + (BC_i - 1) \times tCCD \\
&= ... \\
&= \max_{0 \le l \le BI_i - 1} \{t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD, \\
&\quad t(ACT_{j+l}) + tRCD + ((BI_i - l) \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{14}$$

■

B. 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. The scheduling time of ACT_{j+l} ($\forall l \in [0, BI_i - 1]$) is given by Eq. (4), which depends on $t(ACT_{j+l-1})$, $t(PRE_m)$ and $t(ACT_{j+l-4})$. m represents the latest access number for bank $b_s + l$ where $b_m = b_s + l$. Therefore, $\forall l' \in [l, BI_i - 1]$, we can get Eq. (15) by iteratively employing Eq. (4) and (14). Intuitively, the scheduling of the *ACT* commands which follow ACT_{j+l} has direct or indirect dependencies on ACT_{j+l}. As a result, the scheduling of these *ACT* commands has indirect dependencies on $t(PRE_m)$ and $t(ACT_{j+l-4})$. This intuition is illustrated by Eq. (15). Due to the JEDEC defined timings are constant, Eq. (15) shows that the finishing time of T_i is determined by the latest *PRE* to the same bank, the previous *ACT* commands because of the four activate window constraint in JEDEC standard [23] and the finishing time of the previous transaction T_{i-1}. In a word, Eq. (15) captures the finishing time of a transaction replies on the initial state of banks at the beginning of its command scheduling.

$$\begin{aligned}
&t_f(T_i) = \max_{0 \le l \le l' \le BI_i - 1} \{ \\
&t(PRE_m) + tRP + (l' - l) \times tRRD + tRCD \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&t(ACT_{j+l-4}) + tFAW + (l' - l) \times tRRD + tRCD \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{15}$$

According to Eq. (15), the worst-case finishing time $\hat{t}_f(T_i)$ can be obtained by using $\hat{t}(PRE_m)$, $\hat{t}(ACT_{j+l-4})$ and $t_f(T_{i-1})$. Firstly, the finishing time of T_{i-1} can be obtained from Eq. (8) and there is $t_f(T_{i-1}) = \hat{t}_s(T_i) - 1$. Moreover, Eq. (12) and (13) provide the worst-case scheduling time of the previous *ACT* and *PRE* commands, respectively. $\hat{t}(PRE_m)$ is obtained directly from Eq. (13) while Eq. (12) provides $\hat{t}(ACT_{j+l-4})$ with $BI_i \le 4$. Hence, by replacing $t_f(T_{i-1})$, $t(PRE_m)$ and $t(ACT_{j+l-4})$ in Eq. (15), $\hat{t}_f(T_i)$ is described by Eq. (16) which only supports $BI_i \le 4$ for simplicity.

However, for $BI_i > 4$, which induces $l \ge 4$, $\hat{t}(ACT_{j+l-4})$ can be achieved by Eq. (4), which consists of $\hat{t}(PRE_n)$ and $\hat{t}(ACT_{j+l-8})$. Let n be the latest access number to bank b_{j+l-4} and $\hat{t}(PRE_n)$ is given by Eq. (13). Due to $l-8 < 0$, $\hat{t}(ACT_{j+l-8})$ can be obtained from Eq. (12). Therefore, Eq. (16) can be easily extended to support $BI_i > 4$. However, for simple expression, it only supports $BI_i \le 4$.

$$\begin{aligned}
&\hat{t}_f(T_i) = \max_{0 \le l \le l' \le BI_i - 1} \{ \\
&\hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times RWInterval \\
&\quad + tRP + (l' - l) \times tRRD + tRCD \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&\hat{t}_s(T_i) - 1 - (BC_{i-1} - 1) \times tCCD + (l' - l) \times tRRD \\
&\quad + tFAW - \Delta l \times max\{tRRD, BC_{i-1} \times tCCD\} \\
&\quad + ((BI_i - l') \times BC_i - 1) \times tCCD + \sum_{h=l}^{l'} C(j+h), \\
&\hat{t}_s(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1) \times tCCD\}
\end{aligned} \tag{16}$$

A write transaction T_{i-1} provides a worst-case situation for whatever transaction T_i rather than a read transaction. The

reason is that the prehcarging (*PRE*) of a bank which accepted a *WR* command has to wait longer time than receiving a *RD* command because of the timing constraints in JEDEC DDR3 standard [23]. In addition, for DDR3 SDRAM, we can observe that $tRWTP + tRP + tRCD > tFAW$ for a write transaction. As a result, Eq. (16) is further simplified to Eq. (17) which shows the result of Lemma 2.

$$\begin{aligned}\hat{t}_f(T_i) = &\max_{0\le l\le l'\le BI_i-1}\{\\ &\hat{t}_s(T_i) - 1 + tRWTP - \Delta l \times RWInterval\\ &+ tRP + (l'-l)\times tRRD + tRCD\\ &+ ((BI_i - l')\times BC_i - 1)\times tCCD + \sum_{h=l}^{l'} C(j+h),\\ &\hat{t}_s(T_i) - 1 + tSwitch + (BI_i \times BC_i - 1)\times tCCD\}\end{aligned} \tag{17}$$

■

C. Proof of Theorem 1

Proof: For variable transaction sizes, *RWInterval* is $BC_{i-1} \times tCCD$ according to Eq. (9). As shown in Lemma 2 (see Eq. (17)), 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} (in *RWInterval*), respectively, and 2) linearly increasing or decreasing with l and l' depending on the constant timings and BI_i and BC_i used by T_i. Therefore, the maximum expression in the $max\{\}$ is achieved at least with $BI_{i-1} = BC_{i-1} = 1$. As a result, we can rewrite Eq. (17) and the simplified worst-case finishing time is given by Eq. (18). It indicates the expressions in the $max\{\}$ of $\hat{t}_f(T_i)$ in Eq. (18) 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 $l' = BI_i - 1$. Intuitively, $l = 0$ indicates T_i starts with bank 0 which is the finishing bank of T_{i-1} that requires only one bank because $BI_{i-1} = BC_{i-1} = 1$. Therefore, we assume there is always a command scheduling collision for the *ACT* commands except the first one. According to the timing constraints in JEDEC [23], for the write transaction T_{i-1}, there is $tSwitch < tRWTP + tRP + tRCD$ with all DDR3 SDRAM memories. Hence, Eq. (19) is obtained and shows the result of Theorem 1.

$$\begin{aligned}\hat{t}_f(T_i) = &\hat{t}_s(T_i) - 1 + max\{\\ &tRWTP - l\times(tCCD + tRRD) + tRP + tRCD\\ &+ l'\times(tRRD - BC_i \times tCCD)\\ &+ (BI_i \times BC_i - 1)\times tCCD + \sum_{h=l}^{l'} C(j+h),\\ &tSwitch + (BI_i \times BC_i - 1)\times tCCD\}\end{aligned} \tag{18}$$

$$\begin{aligned}\hat{t}_f(T_i) = &max\{(BI_i \times BC_i - 1)\times tCCD,\\ &(BI_i - 1)\times(tRRD + 1) + (BC_i - 1)\times tCCD\}\\ &+ \hat{t}_s(T_i) - 1 + tRWTP + tRP + tRCD\end{aligned} \tag{19}$$

■

D. 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 $BC_{i-1} = BC_i = BC$. Moreover, *RWInterval* is $max(tRRD, BC \times tCCD)$ on the basis of Eq. (9). Therefore, Lemma 2 (see Eq. (17)) is rewritten and the worst-case finishing time of T_i is described by Eq. (20). Hence, the maximum expression in the $max\{\}$ of $\hat{t}_f(T_i)$ can be obtained only if l' and l select either 0 or $= BI-1$, respectively. In addition, we assume there is always a command scheduling collision for the *ACT* commands. Finally, Eq. (20) is further simplified to Eq. (21), which shows the result of Theorem 2.

$$\begin{aligned}\hat{t}_f(T_i) = &max\{tRWTP + tRP + tRCD + (l'-l)\times tRRD\\ &- (BI - 1 - l)\times max(tRRD, BC \times tCCD)\\ &+ ((BI - l')\times BC - 1)\times tCCD + \sum_{h=l}^{l'} C(j+h),\\ &tSwitch + (BI \times BC - 1)\times tCCD\} + \hat{t}_s(T_i) - 1\end{aligned} \tag{20}$$

$$\begin{aligned}\hat{t}_f(T_i) = &max\{tRWTP + tRP + (BI \times BC - 1)\times tCCD\\ &- (BI - 1)\times max\{tRRD, BC \times tCCD\} + tRCD\\ &+ max\{1, (BI - 1)\times(tRRD - BC \times tCCD) + BI\},\\ &tSwitch + (BI \times BC - 1)\times tCCD\} + \hat{t}_s(T_i) - 1\end{aligned} \tag{21}$$

■

E. WCET Results for Different DDR3 SDRAMs

This section presents additional WCET results for a range of DDR3 SDRAM memories. Results are provided for DDR3-800D, DDR3-1600G and DDR3-2133K SDRAMs with an interface width of 16 bits and a capacity of 2 Gb for several common transaction sizes, 16 B, 32 B, 64 B, 128 B and 256 B. The memory map configurations in terms of *BI* and *BC* for each transaction size are chosen to provide the lowest possible execution time. The scheduled and analytical WCET results of fixed transaction size are presented in Table IV, while Table V contains results for systems with variable transaction sizes. These results can be used as a reference for research on WCET analysis of applications. Note that many commercial systems use DIMMs with a 64-bit interface and that the request size in the first column of the tables has to be multiplied by 4 to get the correct WCET for these memories.

TABLE IV. WCET (CYCLES) OF DIFFERENT DDR3 SDRAMS WITH FIXED TRANSACTION SIZE.

Size	*BI*	*BC*	DDR3-800D		DDR3-1600G		DDR3-2133K	
			Scheduled	*Analytical*	*Scheduled*	*Analytical*	*Scheduled*	*Analytical*
16	1	1	25	26	40	41	52	53
32	2	1	25	27	40	42	52	54
64	4	1	25	29	40	44	52	56
128	4	2	41	41	46	46	56	57
256	4	4	73	73	78	78	82	82

TABLE V. WCET (CYCLES) OF DIFFERENT DDR3 SDRAMS WITH VARIED TRANSACTION SIZES.

Size	*BI*	*BC*	DDR3-800D		DDR3-1600G		DDR3-2133K	
			Scheduled	*Analytical*	*Scheduled*	*Analytical*	*Scheduled*	*Analytical*
16	1	1	25	25	40	40	52	52
32	2	1	29	30	46	47	59	60
64	4	1	37	40	58	61	73	76
128	4	2	53	53	68	68	80	80
256	4	4	85	85	100	100	112	112