Real-Time Syst
DOI 10.1007/s11241-015-9235-y

Architecture and analysis of a dynamically-scheduled real-time memory controller

Yonghui Li[1] · **Benny Akesson**[2] · **Kees Goossens**[1]

Abstract Memory controller design is challenging as mixed time-criticality embedded systems feature an increasing diversity of real-time (RT) and non-real-time (NRT) applications with variable transaction sizes. To satisfy the requirements of the applications, tight bounds on the worst-case response time (WCRT) of memory transactions must be provided to RT applications, while the lowest possible average response time must be given to the remaining applications. Existing real-time memory controllers cannot efficiently achieve this goal as they either bound the WCRT by sacrificing the average response time, or cannot efficiently support variable transaction sizes. In this article, we propose to use dynamic command scheduling, which is capable of efficiently dealing with transactions with variable sizes. The three main contributions of this article are: (1) a memory controller architecture consisting of a front-end and a back-end, where the former uses a TDM arbiter with a new work-conserving policy and the latter has a dynamic command scheduling algorithm that is independent of the front-end, (2) a formalization of the timings of the memory transactions for the proposed algorithm and architecture, and (3) an analysis of WCRT for transactions to capture the behavior of both the front-end and the back-end. This WCRT analysis supports variable transaction sizes and different degrees of bank parallelism. The critical part of the WCRT is the worst-case execution time (WCET) of a transaction, which

✉ Yonghui Li
yonghui.li@tue.nl

Benny Akesson
kessoben@fel.cvut.cz

Kees Goossens
k.g.w.goossens@tue.nl

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 Czech Technical University in Prague, Prague, Czech Republic

Published online: 28 July 2015

is the time spent on command scheduling in the back-end. The WCET is bounded by two techniques applied to both fixed and variable transaction sizes, respectively. We experimentally evaluate the proposed memory controller and compare to an existing semi-static approach. The results demonstrate that dynamic command scheduling significantly outperforms the semi-static approach in the average case, while it performs equally well or better in the worst-case with only a few exceptions. The former reduces the average response time for NRT applications, and the latter pertains the WCRT for RT applications.

Keywords Real time · Memory controller · SDRAM · Formalization · Variable transaction sizes · Worst-case execution/response time

1 Introduction

The complexity of mixed time-criticality system design is growing as an increasingly diverse mix of real-time and non-real-time (NRT) 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 (Benini et al. 2012; van Berkel 2009; Kollig et al. 2009). 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 (Stevens 2010; Gomony et al. 2014). For example, the memory requestors in the NXP digital TV SoC (Kollig et al. 2009; Hansson and Goossens 2011) 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 response time (WCRT), while the lowest possible average response time are needed by NRT applications to be responsive. Since memory controllers typically consists of a front-end and a back-end (Akesson et al. 2007; Krishnapillai et al. 2014), the WCRT of a transaction is composed of the maximum interference delay caused by interfering transactions from other memory requestors in the front-end and its execution time consumed by scheduling commands in the back-end. Therefore, the WCRT captures the behavior of both front-end and back-end, and it is based on the WCET that captures the behavior of the back-end. 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 (Goossens et al. 2012).

Most existing memory controllers are not designed with real-time applications in mind and do not provide bounds on WCRT of transactions (Ipek et al. 2008; Kim et al. 2011; Hur and Lin 2007). Existing real-time memory controllers address real-time requirements by using either (semi-)static or dynamic command scheduling. The former provide bounds based on their static command schedules that are worst-case oriented, and cannot provide low average response time to NRT memory traffic (Bayliss and Constantinides 2009; Akesson and Goossens 2011; Reineke et al. 2011).

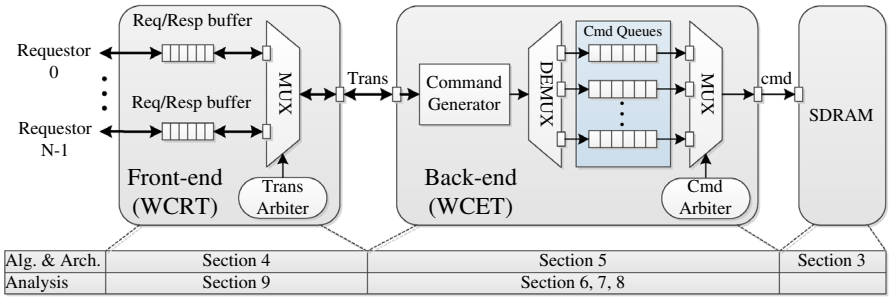


Fig. 1 A general real-time SDRAM controller supporting different requestors.

The latter can provide WCRT bounds and have lower average response times. However, they are *limited in architecture or analysis to a single transaction size and memory map configuration* (Paolieri et al. 2013; Shah et al. 2012; Choi et al. 2011; Wu et al. 2013; Krishnapillai et al. 2014; Kim et al. 2014), resulting in inefficiency for applications with variable transaction sizes.

This article addresses the memory problem of mixed time-criticality systems by providing tight bounds on the execution time and response time for RT memory transactions, while at the same time providing low average execution time and response time for both RT and NRT transactions in systems with variable transaction sizes and different memory map configurations. Fig. 1 shows an overview of our contributions for a real-time memory controller, and also the corresponding section for each of them in this article. We subdivide the three main contributions as mentioned in the abstract into the following five items in more detail:

(1) A memory controller architecture with a front-end and a back-end, as illustrated in Fig. 1. (i) The front-end receives transactions with variable sizes and schedules them by a TDM arbiter with a new work-conserving policy that exploits the back-end properties to reduce the WCRT. (ii) The back-end architecture is designed to support transactions with variable sizes and different memory-map configurations. It executes transactions sent by the front-end with a dynamic command scheduling algorithm that provides pipelining between successive transactions. This back-end can be used with any existing real-time memory controller front-end (transaction arbiter), such as Akesson et al. (2008) and Goossens et al. (2013), not just the one proposed here.
(2) A formalization of the timing behavior of the proposed dynamic command arbiter that captures all the SDRAM timing constraints within and between memory banks.
(3) A generic WCET analysis of an arbitrary transaction in the back-end parametric on the size of the previous transaction. Moreover, the WCET is derived for two specific cases: transactions with fixed size, and with variable sizes. This is achieved by two techniques. The first is analytical and is easy to use, but produces a slightly pessimistic WCET. The second technique employs our formalism to derive the worst-case initial states of banks and then uses an off-line implementation of the command scheduling algorithm to compute a tight WCET.

(4) The result that the WCET is monotonic in the transaction size. It is therefore safe to assume the maximum transaction size of a requestor with transactions of variable sizes when computing its WCRT.
(5) The WCRT analysis of transactions based on their WCET for fixed size and variable sizes, respectively, by adding a novel TDM arbitration mechanism in the front-end, as indicated by Trans Arbiter in Fig. 1.

We experimentally evaluate the proposed architecture, formalization and analysis with fixed and variable transaction sizes, respectively. The results indicate that the formalization is correct, and the WCET and WCRT of transactions are tightly bounded. Moreover, we show that our dynamic command scheduling significantly outperforms a state-of-the-art semi-static approach (Akesson and Goossens 2011) in average execution time and response times, and performs equally well or slightly better in the WCET and WCRT for different transaction sizes with only a few exceptions that are highly dependent on particular transaction size and the timings of the DDR3 SDRAM. For example, the average execution time of 32 Bytes transactions is reduced by 40.2 % for DDR3-800D, which is the main reason we gain 38.9 % improvement in average response time for the Mediabench application *jpegdecode* (Lee et al. 1997). Note that hard real-time systems have no interest in average results, but any mixed time-criticality systems could profit in terms of soft/non-real time tasks getting shorter response times while the WCRT of hard real-time tasks is still guaranteed. The results also show that smaller transactions benefit more from dynamic command scheduling because of a more efficient command scheduling pipeline between successive transactions in the back-end.

In the remainder of this article, Sect. 2 describes the related work. The background of SDRAM and real-time memory controllers are given in Sect. 3. Section 4 presents the memory controller front-end, while Sect. 5 introduces the back-end. The timing behavior of transactions under our dynamic command scheduling is formalized in Sect. 6. With the formalization, Sect. 7 defines the worst-case initial states for a transaction, which is used to derive the worst-case finishing time in Sect. 8. The WCET is then computed in Sect. 9, before the WCRT analysis is presented in Sect. 10. Experimental results are given in Sect. 11, before the article is concluded in Sect. 12.

2 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 single processor-centric view on systems. Most of this work focuses on commercial-off-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 (Dasari et al. 2011; Schliecker et al. 2010; Nowotsch et al. 2014). The work in this article 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) 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 (Bayliss and Constantinides 2009) or semi-static (Akesson and Goossens 2011; Reineke et al. 2011) controller designs are used to achieve a bounded execution time. In Bayliss and Constantinides (2009), an application-specific static command schedule is constructed using a local search method. However, it requires a known static sequence of transactions, which is not available in a system with multiple applications. A semi-static method is proposed in Akesson and Goossens (2011) that generates static memory patterns, which are shorter sub-schedules of SDRAM commands computed at design time, and schedules them dynamically based on incoming transactions at run time. The drawback of this solution is that it cannot efficiently handle variable transaction sizes as the patterns are statically computed for a particular size. When it is employed by transactions with variable sizes in a system, larger transactions use the pattern multiple times, but smaller transactions use the pattern and discard unneeded data. Reineke et al. (2011) presents a semi-static predictable DRAM controller that partitions sets of banks into virtual private resources with independent repeatable actual timing behavior. However, it requires constant duration of accessing the virtual resources. Thus, the actual or average case execution time is equal to the worst-case execution time.

Dynamic command scheduling is used because it more flexibly copes with variable transaction sizes and it does not require schedules or patterns to be stored in hardware. Several dynamically scheduled memory controllers have been proposed in the context of high-performance computing, e.g., Ipek et al. (2008), Kim et al. (2011), Hur and Lin (2007). 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. (2013) propose an analytical model to bound the execution time of transactions under dynamic command scheduling on a modified version of the DRAMSim memory simulator (Wang et al. 2005), 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 Shah (2012), where the WCET 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 Wu et al. (2013) and Krishnapillai et al. (2014), 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 requestors is not greater than the number of memory banks. The number of banks is at most 32 whereas complex heterogeneous systems, such as Kollig et al. (2009), have more memory requestors. This limitation also applies to Ecco et al. (2014). The memory controller in Kim et al. (2014) employs First-Ready First-come First-Serve (FR-FCFS) policy to dynamically schedule commands for transactions with different priorities. However, its analysis of the WCET is pessimistic because of the conservative interference delay among different memory commands. For example, the maximum switching delay between write and read is always taken as the maximum delay for a read or write command. The analysis is also limited to a single transaction size and memory map configuration.

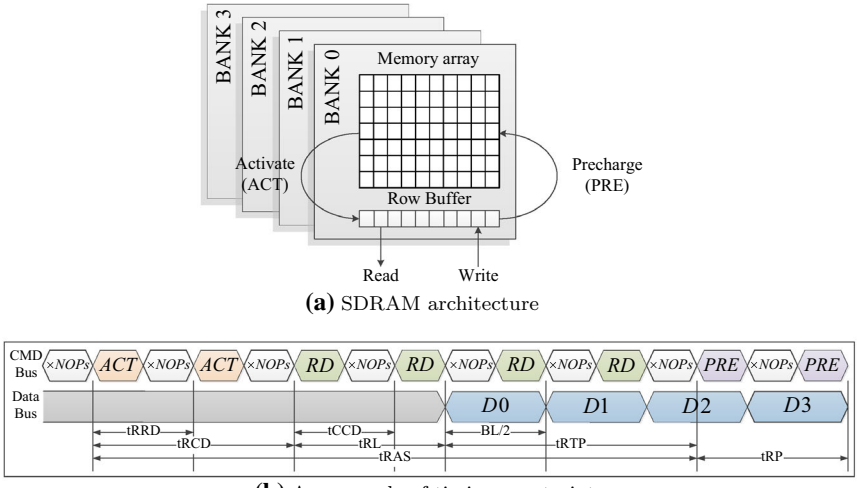
In short, current real-time memory controllers do not efficiently address the dynamic memory traffic in complex heterogeneous systems because of the limitations either in architecture or in analysis with respect to variable transaction sizes and memory map configurations, or both. To fill this gap, this article presents both a dynamically scheduled memory controller architecture and a corresponding analysis that supports different transaction sizes and memory map configurations. This requires a more elaborate analysis than previously published, 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, and by experimental validation.

3 Background

This section presents the required background information to understand the contents of this article. The architecture and basic operations of SDRAM memories are shown, and then a general real-time memory controller that executes memory transaction by scheduling commands to the SDRAM is introduced.

3.1 Introduction to SDRAM memories

An SDRAM chip comprises a set of banks, which contains memory elements arranged in rows and columns (Jacob et al. 2007), as shown in Fig. 2a. All DDR3 memories have 8 banks. Multiple such chips can be combined to form a DIMM with one or more ranks, although without loss of generality, this article focuses on a single chip configu-



(a) SDRAM architecture

(b) An example of timing constraints

Fig. 2 SDRAM architecture and an example of timing constraints.

ration, 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 (JEDEC Solid State Technology Association 2010). Note that although this article 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 first bit of the required data exists on the data bus after *tRL* cycles when issuing the *RD* command, while the delay between the *WR* command and the availability of the first bit of the input data on the data bus is *tWL* cycles. 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 no earlier than *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 NOP is issued if no other commands are valid, i.e., their timing constraints are not satisfied. Similarly, multiple banks are accessed by repeating this process for each of them. Finally, the SDRAM has to be periodically refreshed to retain the data. All these timing constraints are specified by JEDEC DDR3 standard (JEDEC Solid State Technology Association 2010), and are summarized in Table 1 that takes a 16-bit DDR3-1600G memory device with a capacity of 2 Gb as an example.

For example, read data from two banks are illustrated in Fig. 2b. An *ACT* command is scheduled to open the required row in one bank and two consecutive *RD* commands that have sequential addresses within the same row are scheduled to read data. The minimum time between two *RD* commands is *tCCD*. A *PRE* command is issued after the last access to the opened row. The second bank is accessed similarly. Specifically, its *ACT* command is issued earlier than the *RD* commands to the first bank to exploit bank parallelism.

3.2 Real-time memory controllers

In modern multi-core platforms, the off-chip SDRAM is accessed via a memory controller on behalf of memory requestors, such as cores, DMAs, and hardware accelerators. A general real-time memory controller is composed of a front-end and a back-end, as shown in Fig. 1. The front-end generally connects to these requestors that generate memory transactions via a communication infrastructure, e.g., a bus or an on-chip network (NoC). The interconnect must offer real-time performance guarantees, such as the dAElite NoC (Stefan et al. 2014) and be decoupled from the memory controller. The received transactions are buffered in separate queues per requestor. One

Table 1 Timing constraints for DDR3-1600G SDRAM (JEDEC Solid State Technology Association 2010).

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

of these transactions is then selected by the arbiter according to a scheduling policy, such as TDM (Goossens et al. 2013), Round Robin (Paolieri et al. 2013) or Credit-Controlled Static-Priority Arbitration (Akesson et al. 2008), and is finally sent to the back-end. Note that the general front-end shown in Fig. 1 is suitable for contemporary multi-core platforms. For many-core platforms with a very large number of requestors, techniques such as coupling NoC and memory controller (Dev Gomony 2014), distributed arbitration (Gomony et al. 2015) and multiple memory channels (Gomony et al. 2013) can be used. However, this is outside the scope of this article.

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, which determines the location of data in the SDRAM. The memory map also specifies how a transaction is split over the memory banks and thus the degree of bank parallelism used when serving it. This is captured by two parameters: the *bank interleaving* number (*BI*) and the *burst count* (*BC*) (Goossens et al. 2012). *BI* determines the number of banks that a transaction accesses while *BC* represents the number of *RD* or *WR* commands, i.e., bursts, per bank. Each burst has a fixed length that is specified by the SDRAM. The burst length (*BL*) is 8 words for DDR3 SDRAMs. For a given transaction size, a fixed *BI* and *BC* pair is used, and the product of *BI*, *BC* and *BL* is equal to the transaction size in memory words. The command generator translates the transaction into a sequence of commands that are stored in the command queues. Finally, the command arbiter issues

commands to the memory, subject to the timing constraints, and data transmission is triggered by a read or write command. Note that various memory mapping strategies can be supported by specifying different *BI* and *BC* combinations. For example, a small *BI* and a large *BC* support a block-oriented memory mapping that increases the row hit rate by mapping consecutive data bursts to the same row of a bank (Hameed et al. 2013). In contrast, stripe-oriented mapping with large *BI* and small *BC* allocates data bursts to different banks and exploits bank parallelism (Lin et al. 2001).

Real-time memory controllers (Paolieri et al. 2013; Shah et al. 2012; Akesson and Goossens 2011; Reineke et al. 2011) typically employ a close-page policy, where the open row is precharged 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. A close-page policy minimizes the WCET of a transaction with bank interleaving (Paolieri et al. 2013), which is the reason we use it in this article.

4 Memory controller front-end

In this section, we introduce the architecture of our memory controller front-end. We define the front-end hardware architecture that receives transactions from memory requestors and schedules them to the back-end according to a new work-conserving TDM arbiter, offering lower interference delay. The back-end that executes each transaction by dynamically scheduling commands to the SDRAM is later introduced in Sect. 5.

4.1 Front-end architecture

The front-end receives memory transactions on its ports from different requestors either directly or via a bus or a network-on-chip. Transactions are queued in the request buffers per requestor, as illustrated in Fig. 3. Typically, requestors generate memory transactions with fixed size, e.g., CPU cache misses (Stevens 2010). Therefore, each requestor is assumed to have a fixed transaction size, while it varies between different

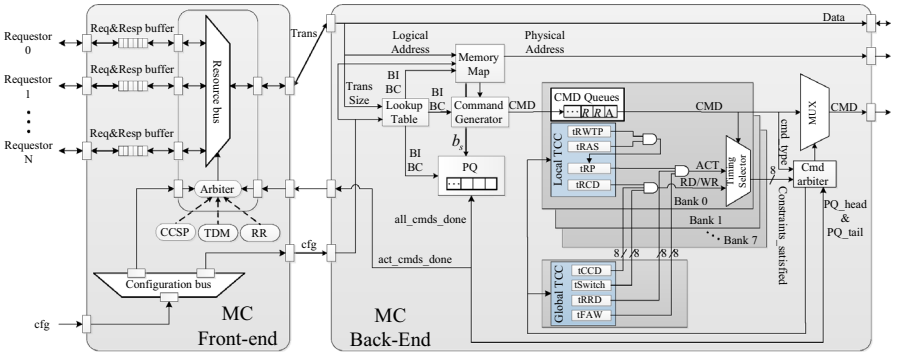


Fig. 3 A mixed time-criticality memory controller with dynamic command scheduling.

requestors. Note that this assumption can be relaxed later in Sect. 9.3, where a safe WCRT is guaranteed by using the largest transaction size of a requestor when it actually generates variable transaction sizes. The size of a request buffer is determined by the maximum number of outstanding transactions from the requestor. In a fully timing-compositional architecture, each of the requestors is assumed to have at most one outstanding memory transaction (a cache miss) (Axer et al. 2014). The delay caused by the shared SDRAM is additive to the application execution time. In this article, we assume that each requestor has a single outstanding transaction. After sending a read transaction, a requestor cannot schedule the next transaction before receiving the response, i.e., the data is returned. For a write transaction, an acknowledgment is sent back to the requestor after the data is written into the SDRAM memory, and then the requestor sends the next transaction. The response buffer per requestor receives the data that is read from the memory for a read transaction, before it is returned back to the corresponding requestor. This assumption ensures that requestors do not experience self-interference. The self-interference can be taken into account in the system-level model into which the memory controller is integrated.

The front-end in Fig. 3 supports any predictable arbiter chosen at design time. We have implemented CCSP (Akesson et al. 2008), TDM (Goossens et al. 2013), and Round Robin (PrimeCell AHB SDR and NAND memory controller 2006). The arbiter selects one request queue and sends the first transaction to the back-end. A novel TDM arbiter in the next section is taken as an example to show how the proposed memory controller with dynamic command scheduling works. It exploits static information of the TDM schedule of requestors with different transactions sizes, and this static information is also used by the proposed worst-case analysis. The arbiter makes a scheduling decision when triggered by the back-end via the arbitration signal *act_cmds_done* in Fig. 3.

4.2 Work-conserving TDM arbitration for variable-sized transactions

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

4.2.1 TDM arbitration issues for variable-sized transactions

The non-preemptive TDM arbiter, shown in the front-end in Fig. 3, serves requestors with different transaction sizes, which results in variable execution time for transactions and hence different time slot durations. The execution time is defined as the scheduling time of the last command of the transaction minus the starting time of the transaction, and it depends on both the size of the transaction and the initial bank states when it arrives at the back-end. In particular, the size of the previous transaction affects the bank states, and a smaller previous transaction results in a larger WCET of the current transaction. The reason is that larger successive transactions pipeline

more efficiently, as discussed later in Sect. 9. It hence follows that the order of serving requestors with different transaction sizes, i.e., their order in the TDM table influences the WCET of their transactions. From this discussion, we conclude that the duration of TDM slots varies and depends on several different factors. This is an issue that should be solved when using TDM arbitration for variable transaction sizes.

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

To solve these two issues, we firstly propose a new work-conserving policy for non-preemptive TDM arbiters used by requestors with variable transaction sizes. This policy has two innovations. (1) When a requestor r that is allocated the current TDM slot has no pending transactions, the current slot becomes idle and we specify that *this idle slot and the following slots belonging to r are skipped by the arbiter*. Instead, the next requestor with pending transaction(s) is served. As a result, idle slots are skipped, instead of being reallocated to another requestor (with larger transactions perhaps). Therefore, the maximum interference experienced by a requestor is always smaller than when its slots would have been reused by another requestor. Moreover, skipped slots reduce the waiting time for all other requestors. (2) We configure the TDM arbiter to *serve requestors in descending order of their transaction sizes*, such that their WCET and hence slot durations are smaller. This takes advantage of the fact that a transaction has a smaller WCET when preceded by a larger transaction. Note that the largest transaction is preceded by the smallest one because the TDM arbiter periodically serves requestors. However, the approach still results in the minimum total length of all slots. Sect. 11 experimentally validates these innovations.

4.2.2 Transaction scheduling algorithm

Algorithm 1 presents the proposed work-conserving TDM arbitration. The inputs of Algorithm 1 include the arbitration signal *act_cmds_done* in Fig. 3, which triggers the front-end to arbitrate when the back-end is ready to accept a new transaction. Another input is the information whether or not a request queue has a pending transaction, and it is collected by *RQueues*[ ]. The third input is the TDM slot allocation, which is configured in a table *TDM_Table*[ ]. It specifies the order of serving requestors and the number of slots per requestor. The TDM arbiter is configured to serve requestors in descending order of their transaction sizes, which is the second innovation presented previously. The transaction sizes of requestors hence decrease from requestor 0 to requestors N-1, and the requestors are served in this order. Finally, the output of Algo-

rithm 1 is the number of the request queue, denoted by *Q_ID*, whose head transaction is scheduled to the back-end.

Algorithm 1 Transaction scheduling with work-conserving TDM

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

To obtain *Q_ID*, Algorithm 1 uses two internal variables *r_index* and *s_index* that are the index of a requestor (associated with a request queue) and the index of its allocated slots, respectively. Algorithm 1 begins with initializing *Q_ID* to be invalid (line 5), and it ends when *Q_ID* becomes valid (line 16). However, the exploration of a valid *Q_ID* (between line 7 and 17) starts only if *act_cmds_done* is true and there exists at least one request queue with a pending transaction, as shown on line 6. If there are no queues with pending transactions, this algorithm restarts the following clock cycle until a new transaction arrives at the front-end. The algorithm firstly checks whether the request queue indexed by *r_index* has a transaction (line 8). If not, then its allocated slots become idle and are *skipped* by setting *r_index* to the next request queue, and *s_index* to 0 (line 14 to 15). This is different from traditional work-conserving TDM arbitration, where the idle slots are reallocated to another arbitrary requestor with pending transaction(s). This is the first innovation as discussed previously. If the request queue *r_index* has a pending transaction (line 8), the algorithm behaves the same as normal TDM arbitration.

Next, we position our proposed TDM arbiter with respect to other similar arbitration mechanisms. Round robin is a special case of the work-conserving TDM when each requestor is only allocated a single slot. In TDM, requestors can have more than one slot and the allocated slots can be placed in any order in the TDM table (Akesson et al. 2015). In our case, we allocate continuous slots to a requestor, and the slots of different requestors are placed in descending order of their transaction sizes. The FlexRay communication protocol (Pop et al. 2008) contains a dynamic segment in which requestors with variable messages sizes indicate whether or not they have a message to send in a given slot. If not, a mini-slot (i.e., a short time period) is wasted instead of the given slot. This mini-slot accumulates when a number of given slots are

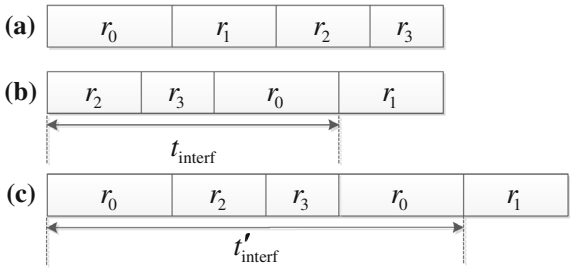


Fig. 4 The worst-case interference delay for requestor r_1: **a** TDM slot allocation; **b** the proposed work-conserving TDM arbiter; **c** traditional work-conserving TDM arbiter.

not used. Moreover, because the segment has fixed length, it may happen that the last message will not fit, resulting in that the remaining time in the segment is wasted. The proposed Algorithm 1 does not suffer these problems.

4.2.3 Example

Take four requestors r_0, r_1, r_2, and r_3 with different transaction sizes as an example to illustrate the benefits of the proposed work-conserving TDM arbitration. As previously stated, we assume the transaction sizes decrease from r_0 to r_3. Each of them is allocated one slot in the TDM table, as shown in Fig. 4a. The slot duration of each requestor is the WCET of its transactions experienced in the back-end. The larger transactions have larger WCET, as later shown in Sect. 9.3. As a result, the slot duration of r_0 is largest while it is the smallest for r_3 (see Fig. 4a). Moreover, the TDM arbiter is configured to serve requestors in the descending order of their transaction sizes. As a result, the requestors are served in the order from r_0 to r_3.

Take requestor r_1 as an example. In the worst case, a transaction from r_1 arrives just as it misses its slot. This idle slot is hence skipped according to the proposed work-conserving TDM arbitration, and the following requestors r_2, r_3 and r_0 use their allocated slots. This leads to the maximum interference delay t_{interf} for r_1, as shown in Fig. 4b. A traditional work-conserving TDM arbiter could reallocate this idle slot to another requestor, e.g., r_0 as a bonus, according to some slack-management policy. Then, the following requestors r_2, r_3, and r_0 consume their allocated slots (Fig. 4c), leading to interference delay $t'_{interf} > t_{interf}$. The difference between them is actually the duration of the idle slot that was given as a bonus to requestor r_0. Hence, the proposed work-conserving TDM arbiter is capable of providing smaller WCRT for transactions with variable sizes.

5 Memory controller back-end

The memory controller back-end receives scheduled transactions from the front-end, as shown in Fig. 3. However, it is a general component that could be used without the front-end, for example by connecting to a memory tree NoC (Gomony et al. 2015) that plays the arbitration role to schedule transactions from different requestors to the back-end. Each arriving transaction is translated into a number of memory commands that are scheduled to a number of consecutive banks subject to the timing constraints

of the memory. The basic idea underlying the back-end architecture (Sect. 5.1) and command arbiter (Sect. 5.2), is that each transaction generates an *ACT* command followed by *BC* times *RD* or *WR* commands, the last one with an auto-precharge. This commences with the starting bank b_s and is repeated *BI* times for all the required banks. Commands are generated one per cycle, but are usually scheduled more slowly, due to timing constraints. Commands are therefore buffered per bank (see Fig. 3, and discussed below). To limit the size of the command queues per bank while still enabling pipelining between transactions, a new transaction is sent by the front-end and hence new commands are admitted to the queues only when all the *ACT* commands of the current transaction have been issued to the memory. This is enforced by the command arbiter via the act_cmds_done signal in Fig. 3 that triggers a new scheduling decision in the front-end. To avoid read/write hazards or read-response reorder buffers, the *RD*/*WR* commands of a transaction are scheduled before those of the next transaction. This order (as a (BI, BC, b_s) tuple) of each transaction is stored in the parameter queue (PQ), and used by the command arbiter to guarantee in-order execution of transactions. This results in an efficient pipelined back-end.

5.1 Back-end architecture

We proceed by introducing the main components in the back-end to support variable transaction sizes, which include the **Lookup Table**, parameter queue (**PQ**), **Command Generator** and the **Cmd Arbiter**, as shown in Fig. 3. In addition, other common components used by existing memory controllers are also briefly introduced to show how these components constitute a dynamically scheduled back-end.

As shown in Fig. 3, (1) The **Lookup table** translates the transaction size to the bank interleaving number (*BI*) and burst count (*BC*), which are needed by the command generation. They are determined at design time when the memory map configuration is chosen and are programmed via a configuration interface (cfg) when the system is initialized. If there is no (*BI*, *BC*) corresponding to a transaction size in the Lookup Table, the (*BI*, *BC*) related to the next larger size is used with the additional data being masked out. An important (usually unstated) assumption on the translation from size to (*BI*, *BC*) is that it must be monotone, as given by Definition 1, where $S(T')$ and $S(T)$ are the sizes of transaction T' and T, respectively. A methodology to choose the memory map configuration based on the requirements of bandwidth, execution time and power consumption has been presented in Goossens et al. (2012). (2) With the *BI* and *BC*, the widely used **Memory Map** module in Fig. 3 translates the logical address of the transaction into the starting physical address that consists of the starting bank b_s, row, and column. (3) Then (BI, BC, b_s) of the transaction is inserted at the back of the parameter queue (PQ). This queue keeps track of the order of transactions in the back-end and is used by the command scheduling algorithm in Sect. 5.2.

Definition 1 (Monotone memory mapping) For $\forall T$ and T', $S(T') \leq S(T) \implies BI' \leq BI \wedge BC' \leq BC$.

Based on (BI, BC) and the physical address, (4) the **Command Generator** generates memory commands for each bank, i.e., an *ACT* command is generated, followed by

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

To account the timing constraints of the commands, (5) **timing counters** are commonly used by dynamically scheduled memory controllers. Each counter tracks one timing constraint specified by the JEDEC DDR3 standard (JEDEC Solid State Technology Association 2010). We classify the timing constraint counters (TCC) into local TCC and global TCC, which constrain the command scheduling for the same bank and different banks, respectively. Most timing constraints shown in Fig. 3 are directly provided by JEDEC, while *tRWTP* and *tSwitch* are derived from the JEDEC specification and are shown in Eqs. (1) and (2), respectively. *tRWTP* is the time between a *RD* or *WR* command and the precharging to the same bank, while *tSwitch* limits the time between two successive *RD* and/or *WR* commands. Due to the double data rate of DDR SDRAM, *BL*/2 is the time consumed transferring a burst of data associated with a *RD* or *WR* command.

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

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

A command that is at the head of the command queue can be issued only if its timing constraints are satisfied in the current cycle. It is then called a *valid command*. As shown in Fig. 3, the (6) **Timing Selector** of the bank shows whether the timing constraints for the head command are satisfied. Multiple command queues may have a valid command simultaneously. This implies *command scheduling collisions*, since only one command can be issued per cycle on the command bus. Therefore, an arbiter is required to select a valid command, which is the (7) **Cmd Arbiter** shown in Fig. 3. It has to guarantee in-order execution of transactions to avoid the architectural and analysis complexity of reordering. Moreover, it provides the valid arbitration signal act_cmds_done to the front-end when all the *ACT* commands of the current transaction have been scheduled, such that the front-end schedules a new transaction to enable pipelining of transactions. To achieve these goals, it uses the command scheduling algorithm presented in Sect. 5.2. Finally, the chosen command is removed from the 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. Lastly, a refresh command needs to be scheduled every *tREFI* cycles. Once triggered, it is scheduled after the data transmission of the currently executing transaction to prevent unnecessary interference, while still ensuring that no refresh command is delayed more than $9 \times tREFI$ clock cycles, as specified by the

DDR3 standard (JEDEC Solid State Technology Association 2010). Refresh is also implemented by timing counters, which are not depicted in Fig. 3 for simplicity.

5.2 Dynamic command scheduling algorithm

After memory commands are generated and stored in the command queues by the Command Generator in Fig. 3, the arbiter has to decide which command to schedule every clock cycle for transactions in the back-end. It has to solve three critical issues, namely:

(1) a single command must be chosen from the set of valid commands;
(2) transactions must be executed in first-come-first-serve (FCFS) order to avoid reorder buffers for the responses;
(3) to simplify logical-to-physical address translation (Goossens et al. 2012), successive banks of a single transaction have to be accessed in ascending order.

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

These priorities form the basis of Algorithm 2 that is used by the arbiter to select a command from the multiple valid commands in every cycle. Note that a NOP is scheduled when there is no valid command in a cycle. As shown in Fig. 3, the inputs of the arbiter include the outputs of the Timing Selectors, the type (*ACT*, *RD* or *WR*) of each command at the head of the command queues, and the head and tail elements of the parameter queue. These inputs are taken by Algorithm 2 and represented by constraint_satisfied, cmd_type, and PQ_head and PQ_tail, respectively. constraint_satisfied and cmd_type are arrays with sizes equal to the number of command queues. The outputs of Algorithm 2 are bank_id, all_cmds_done and act_cmds_done, where bank_id indicates the command queue whose head command can be scheduled to bank bank_id. all_cmds_done is true when all commands of the current transaction have been issued to the memory. The (BI, BC, b_s) triple at the head of the parameter queue is then removed. act_cmds_done indicates whether all *ACT* commands of the current transaction have been sent to the memory. When true, this triggers the front-end to arbitrate for a new transaction, even though *RD*/*WR* commands of current and past transactions are (likely to be) pending.

In Algorithm 2, line 6 checks whether there is a valid *RD*/*WR* command for the current bank (rw_bank) for reading/writing. Otherwise, line 14 checks whether there is a valid *ACT* command. This guarantees that a valid *RD* or *WR* command has higher priority than a valid *ACT* command. *act_bank* and *rw_bank* indicate the number of the bank to which an *ACT* or a *RD*/*WR* command can be scheduled, respectively. *act_bank* is increased by one after an *ACT* command has been selected (line 18),

Algorithm 2 Dynamic command scheduling

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

while *rw_bank* increases by one when *BC* number of *RD*/*WR* commands of the current transaction have been scheduled to bank *bank_id* (line 12). This update scheme ensures the banks are accessed in ascending order for each transaction. *act_bank* and *rw_bank* are initialized with the starting bank b_s of the transactions associated with the tail and head of the parameter queue, respectively (line 4, 5). A new transaction can only be sent to the back-end if all the *ACT* commands of the current transaction have been issued, as indicated by act_cmds_done (line 17). As a result, only one transaction has *ACT* commands in the command queues, namely the one at the tail of the parameter queue (PQ_tail). Hence, transactions are served in FCFS order, the banks of each transaction are served in ascending order, and command priorities ensure that only a single command is scheduled per cycle. Algorithm 2 thus addresses all three critical issues mentioned previously. Although command priorities are used, there is no livelock or starvation since transactions are executed in FCFS order.

Regarding the hardware cost, our memory controller is comparable to existing memory controllers, such as the one based on First-Ready First-Come First-Serve (FR-FCFS) policy (Kim et al. 2014), the ROC (Krishnapillai et al. 2014) and the cadence DDR controller (Cadence Design Systems Inc 2014). Our memory controller has the common components with these existing memory controllers, such as the request/response buffers in the front-end and the memory map, command queues, command generator, and the timing constraint counters in the back-end. The additional components of our memory controller are the lookup table and the parameter queue, which have a limited number of entries. Moreover, the arbiters in the front-end and back-end use Algorithms 1 and 2, respectively. They are similar to existing arbiters, such as the work-conserving TDM (Goossens et al. 2013) and the 3-level arbitrations of ROC (Krishnapillai et al. 2014). We therefore expect our memory controller to be similar in area and speed to existing designs.

6 Formalization of dynamic command scheduling

In this section, we introduce standard notation and definitions to formalize the back-end architecture and the dynamic command scheduling of transactions, as is specified by Algorithm 2 in Sect. 5.2. As introduced in the previous section, a transaction is translated into a series of *BI bank accesses*. Each bank access activates a bank, and then reads or writes *BC* times, the last time with auto-precharge. A command can only be scheduled and executed at its scheduling time when the timing constraints from previous commands are satisfied. Timing constraints therefore result in scheduling dependencies. A bank access is a natural self-contained group of commands, and each transaction is made up of one or more bank accesses. For this reason, in our analysis, we focus on sequences of individual bank accesses b_j, and care less (in the first instance) about the sequence of individual transactions T_i that generated these bank access. The notation used in this section is summarized in Table 2. Note that the formalization of dynamic command scheduling and the analysis in this article are the novelties and not the mathematical analysis techniques.

Table 2 Summary of notation.

Variables	*Descriptions*
i	The number of an arbitrary transaction arrived at the back-end. $i \geq 0$.
T_i	The ith transaction received by the back-end
$S(T_i)$	The size of transaction T_i
j	The first bank access number for the current transaction T_i. $j \geq 0$ is defined by Eq. (3).
BI_i, BC_i	The bank interleaving number (BI) and burst count (BC) of T_i
b_j	The bank number that is targeted by the jth bank access, which is also the starting bank of T_i. $b_j \in [0, 7]$ represents one of the 8 banks in DDR3 SDRAMs.
ACT_j	The ACT command of the j^{th} bank access
$t(ACT_j)$	The scheduling time of ACT_j
$C(j)$	The delay in scheduling ACT_j due to a collision; and it is either 1 or 0 cycle depending on whether the collision exists or not.
RW_j^k	The kth ($\forall k \in [0, BC_i - 1]$) RD or WR command of the j^{th} bank access
$t(RW_j^k)$	The scheduling time of RW_j^k
PRE_j	The PRE command for the j^{th} bank access
$t(PRE_j)$	The scheduling time of PRE_j
$t_s(T_i)$	The starting time of T_i
$\hat{t}_s(T_i)$	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
l	Used to index the banks of a transaction T_i and $\forall l \in [0, BI_i - 1]$
k	Used to index the bursts of a bank for T_i, and $\forall k \in [0, BC_i - 1]$

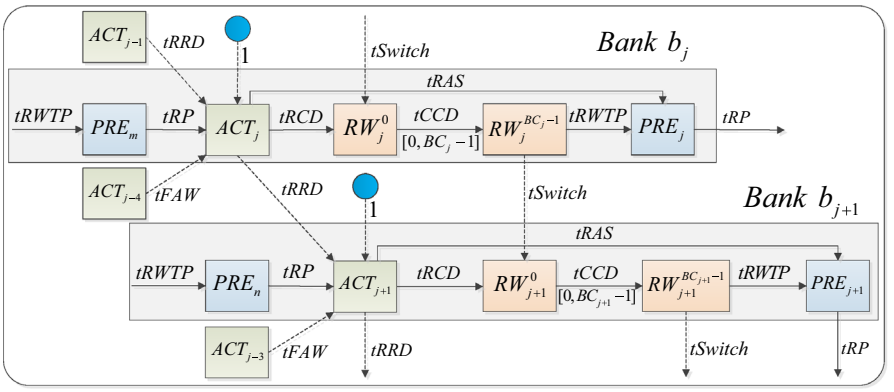


Fig. 5 Timing dependencies between successive bank accesses.

6.1 Timing dependencies

In dynamic command scheduling, the order of command execution is decided at run time on the basis of the timing constraints between commands. Timing constraints therefore impact the command schedule by introducing scheduling dependencies. This is shown in Fig. 5, where the dotted and solid arrows represent dependencies between banks (b_j and b_{j+1}) and within a single bank (b_j), respectively. Note that b_j and b_{j+1} may be from the same or different transactions. The scheduling of a command depends on the previous commands, which are specified by the input arrows. The labels near the arrows specify the timing constraint between the commands, i.e., the number of cycles that the following command has to wait before it could be scheduled. For example, the timing constraints to schedule an *ACT* command include *tRRD*, *tRP* and *tFAW*, previously described in Table 1. Therefore, the block of an *ACT* command (see Fig. 5) has three input arrows that represent the corresponding timing constraints. The scheduling of the first *RD* or *WR* command for a bank access has to satisfy the timing constraints *tRCD* and *tSwitch*. The following *RD* or *WR* commands of the bank access only need to take the timing constraint *tCCD* into account. Finally, an auto-precharge considers the timing constraints *tRAS* and *tRWTP*. The timing dependencies among the commands are illustrated in Fig. 5. Note that refresh commands are not depicted because their impact on WCET can be easily analyzed, as presented in Sect. 10. Moreover, the effect of *REF* is small (approximately 3 %) in memory interference delay, and it is not a main concern in this article.

According to Algorithm 2, an *ACT* command may be blocked by a higher-priority *RD* or *WR* command from previous bank accesses. This command scheduling conflict postpones the *ACT* command by one cycle. The collision is depicted by the filled circle in Fig. 5, which 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. The *PRE* in Fig. 5 does not use the command bus due to the auto-precharge policy, and hence 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.

6.2 Formalization

Having explained the dependencies between commands in a bank access according to the DDR3 standard and illustrated them in Fig. 5, we analyze the execution time of a transaction by computing the actual scheduling time of commands under our dynamic scheduling algorithm. We compute the worst-case execution time in Sect. 9.

Definition 2 (Arrival time of transaction T_i) $t_a(T_i)$ is defined as the time at which T_i arrives at the interface of the back-end.

When an arbitrary transaction T_i ($\forall i > 0$) arrives at the back-end with the arrival time defined by Definition 2, it is executed by scheduling commands to a number of banks. We assume T_i uses BI_i and BC_i, and the starting bank is b_j. j is the number of the first bank access of T_i and it is a function of i, as given by Eq. (3). It is the total number of bank accesses by previous transactions. Note that $j(i)$ is denoted by j for short throughout this article.

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

Expanding Fig. 5 to an entire transaction T_i, Fig. 6 illustrates the scheduling dependencies between all its commands. The command scheduling for T_i depends on zero or more previous transaction(s) $T_{i'}$. For $\forall l \in [0, BI_i - 1]$, the $(j+l)^{th}$ bank access comprises ACT_{j+l} and several RD or WR commands to bank b_{j+l}. The RD or WR commands are denoted by RW^k_{j+l}, where $\forall k \in [0, BC_i - 1]$. Moreover, an auto-precharge PRE_{j+l} is issued after the access of bank b_{j+l}, and it is specified by an auto-precharge flag issued together with $RW^{BC_i-1}_{j+l}$. For $BI_i > 4$, the scheduling of some ACT commands also depends on the previous ACT commands of the current transaction T_i because of the four-activate window ($tFAW$).

The finishing time of T_i (Definition 3) is the time when the last RD or WR command $RW^{BC_i-1}_{j+BI_i-1}$ is scheduled. The starting time of T_i (Definition 4) is defined as the earliest time at which the arbiter 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 (Definition 5) of the transaction.

Definition 3 (Finishing time of transaction T_i) $t_f(T_i) = t(RW^{BC_i-1}_{j+BI_i-1})$

Definition 4 (Starting time of transaction T_i) $t_s(T_i) = max\{t_a(T_i) + 2, t_f(T_{i-1}) + 1\}$

Definition 5 (Execution Time of transaction T_i) $t_{ET}(T_i) = t_f(T_i) - t_s(T_i) + 1$.

For T_i, Eq. (4) computes the scheduling time of ACT_{j+l} where $m = max_{k<j}\{k|b_k = b_{j+l}\}$, is the previous bank access to bank b_{j+l}. The max function in Eq. (4) 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 the look up table and command generation, as previously mentioned. In case of

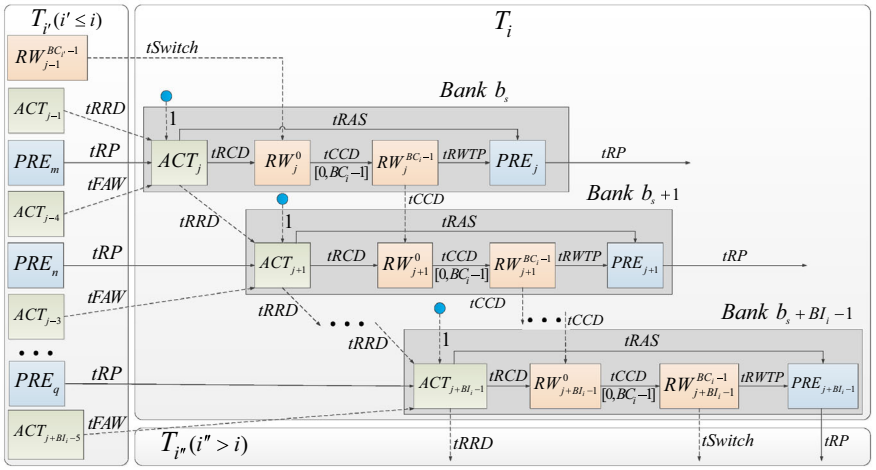


Fig. 6 The timing dependencies of command scheduling for transaction T_i.

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^k_{j+l} is given by Eqs. (5) and (6). Eq. (5) provides the scheduling time of the first *RD* or *WR* command of T_i to bank b_{j+l}. It depends on $t(RW^{BC_i-1}_{j+l-1})$, which is the scheduling time of the last *RD* or *WR* to b_{j+l-1}, and the scheduling time of ACT_{j+l}. Note that for $l = 0$, $t(RW^{BC_i-1}_{j-1})$ is defined as the finishing time of T_{i-1}. The scheduling time of the remaining *RD* or *WR* commands ($k \in [1, BC_i - 1]$) to bank b_{j+l} only depend on the previous *RD* or *WR* command, and is given by Eq. (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. We define the finish time of the first transaction as $t_f(T_0) = -\infty$, such that the *ACT* of the first transaction T_1 can be scheduled at time 0. These equations have been implemented in our open source tool (Li et al. 2014b) to provide the scheduling time of commands.

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

$$t(RW^0_{j+l}) = max\left\{t(RW^{BC_i-1}_{j+l-1}) + tSwitch, t(ACT_{j+l}) + tRCD\right\} \quad (5)$$

$$t(RW^k_{j+l}) = t(RW^0_{j+l}) + k \times tCCD \quad (6)$$

$$t(PRE_{j+l}) = max\left\{t(ACT_{j+l}) + tRAS, t(RW^{BC_i-1}_{j+l}) + tRWTP\right\} \quad (7)$$

Based on Eq. (4)–(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 first *RD* or *WR* commands and the *ACT* to each bank have

dependencies on previous transactions. The other *RD* or *WR* commands can be scheduled with the dependencies directly or indirectly originating from those commands. *Intuitively, the finishing time of T_i is determined only by the scheduling time of all its ACT commands, the finishing time of the previous transaction and JEDEC-defined timing constraints*. This intuition is formalized by Lemma 1 and the proof is included in the appendix.

Lemma 1 *For $\forall i > 0$ and $t_f(T_0) = -\infty$,*

$$t_f(T_i) = \underset{0 \le l \le BI_i - 1}{Max} \{ t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1) \times tCCD, \\ t(ACT_{j+l}) + tRCD + [(BI_i - l) \times BC_i - 1] \times tCCD \}$$

7 Worst-case initial bank states

The command scheduling for the current transaction T_i is highly dependent on the initial bank states resulting from when the commands of the previous transactions (e.g., T_{i-1} and T_{i-2}) were scheduled. Intuitively, given that the minimum starting time of T_i is fixed by the finishing time of T_{i-1}, the worst-case finishing time of T_i occurs when all the commands of T_{i-1} were scheduled as late as possible (*ALAP*), because this maximizes the timing dependencies. In this section, we formalize the *ALAP* scheduling of T_{i-1}, which defines the worst-case initial bank states for T_i. Later Sect. 8 computes the worst-case finishing time of T_i based on these worst-case initial bank states. The WCET of T_i is finally given in Sect. 9.

7.1 Worst-case starting time

From Definition 5, it follows that the execution time, $t_{ET}(T_i)$ is maximized if *the starting time is minimum while the finishing time is maximum*. According to Definition 4, the starting time $t_s(T_i)$ is determined by its arrival time $t_a(T_i)$ and the finishing time $t_f(T_{i-1})$ of the previous transaction T_{i-1}. In the worst-case situation, T_i has arrived before the finishing of T_{i-1}, such that the commands for T_i have to wait longer time for their timing constraints to be satisfied. Therefore, the worst-case starting time of T_i is only one cycle after the finishing time of T_{i-1} and is given by Eq. (8).

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

To derive the maximum finishing time of T_i, denoted by $\hat{t}_f(T_i)$, the scheduling time of its *ACT* commands should be maximized according to Lemma 1. Eq. (4) indicates that the scheduling of an *ACT* command depends on the previous *PRE* to the same bank, the previous *ACT* commands and the possible collision caused by a *RD* or *WR* command. Therefore, the worst-case finishing time of T_i is achieved by maximizing the scheduling time of the previous *PRE* and *ACT* commands as well as assuming there is always a command collision for every *ACT* command.

This preceding transaction T_{i-1} has many possibilities, since it is not statically known. For example, it may be a read or a write with variable sizes and requiring different sets of banks, and its commands were scheduled based on its initial bank states that were determined by even earlier transactions. Therefore, it is hard to statically know which T_{i-1} provides the worst-case initial bank states for T_i. However, the worst-case starting time given by Eq. (8) defines the finishing time $t(RW_{j-1}^{BC_{i-1}-1})$ of T_{i-1} and *we can conservatively assume all the commands of* T_{i-1} *were scheduled as late as possible* (*ALAP*) *with respect to the fixed finishing time of* T_{i-1}, *subject to the timing constraints of the memory.* This *ALAP* scheduling ensures the latest (i.e., maximum) possible scheduling time of the previous commands, which are the worst-case initial bank states for T_i.

7.2 *ALAP* scheduling

This section shows how to formalize the *ALAP* scheduling by computing the worst-case (latest possible) scheduling time of all the commands for the previous transaction. According to *ALAP* scheduling, the scheduling time of the previous *ACT*, *RD* or *WR* commands and *PRE* can be obtained by calculating backwards from the scheduling time of the last *RD* or *WR* command at $t(RW_{j-1}^{BC_{i-1}-1})$. Specifically, the time between any successive commands must be minimum while satisfying the timing constraints, thereby ensuring an *ALAP* schedule of the previous commands. Therefore, the *minimum time interval* between any two commands is significant to formalize the *ALAP* scheduling. Recall that T_{i-1} has BI_{i-1} and BC_{i-1}. First, as stated in Table 1, the minimum time between two *RD* or *WR* commands is *tCCD*. Since *RD* or *WR* commands targeting the same bank are scheduled sequentially, the minimum time between the first *RD* or *WR* commands to consecutive banks is $BC_{i-1} \times tCCD$. Second, an *ACT* command is followed by a *RD* or *WR* command to the same bank, and their minimum time interval is *tRCD* (see Table 1). This implies that an ACT command must be scheduled at least *tRCD* cycles before the first *RD* or *WR* command to the same bank. To calculate backwards, the time interval between two successive ACT commands to different banks has to be at least $BC_{i-1} \times tCCD$. In addition, Table 1 also states that the minimum time between two *ACT* commands to different banks is *tRRD*. Hence, the minimum time interval between two successive *ACT* commands to different banks without violating any timing constraints is $max\{tRRD, BC_{i-1} \times tCCD\}$.

Fig. 7 illustrates an example of *ALAP* scheduling for a DDR3-1600G SDRAM. This example assumes the current transaction T_i and the previous write transaction T_{i-1} have the same starting bank *Bank* 0. T_i has $BI_i = 4$ and $BC_i = 2$, while T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. With the fixed finishing time $t(RW_{j-1}^{1})$ of T_{i-1}, the scheduling time of all the previous commands is computed backwards with the minimum time interval between them. Fig. 7 shows the *ALAP* scheduling of the previous commands for T_{i-1} to *Banks* 0 and 1. In this way, some *ACT* commands have the same scheduling time as some *WR* commands, which indicates command scheduling collisions. However, we conservatively ignore these collisions so that larger scheduling time of the previous *ACT* and *WR* commands for T_{i-1} is achieved, which provide the initial bank states for the new transaction T_i.

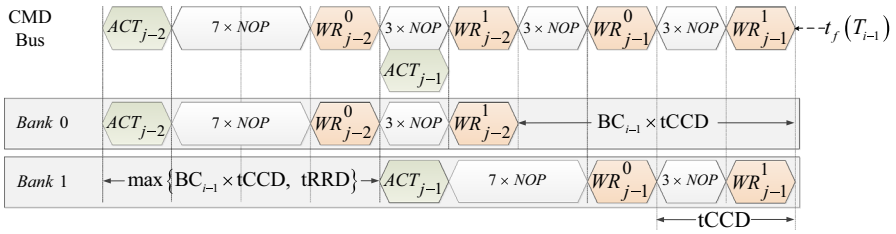


Fig. 7 An example of As-Late-As-Possible (*ALAP*) scheduling with DDR3-1600G SDRAM for T_i which has $BI_i = 4$ and $BC_i = 2$. The previous transaction T_{i-1} uses $BI_{i-1} = 2$ and $BC_{i-1} = 2$. The starting bank for both T_{i-1} and T_i is *Bank* 0.

Next, *ALAP* scheduling is formalized to provide the scheduling time of previous commands. First, the preceding transaction T_{i-1} must have banks in common with T_i, because the reactivation of a bank for T_i needs more time if it was accessed by T_{i-1}. To obtain larger finishing time, the starting bank b_j of T_i must have been accessed by T_{i-1}, and the last bank b_{j-1} of T_{i-1} is also required by T_i. As a result, the set of common banks is $[b_j, b_{j-1}]$. We introduce the short hand notation $b_{com} = b_{j-1} - b_j$ and the number of common banks is hence $b_{com} + 1$. For example, the set of common banks between T_{i-1} and T_i in Fig. 7 is [0, 1], and the number of common banks is 2. With the minimum time interval between commands, for $\forall l \in [0, b_{com}]$ and $\forall k \in [0, BC_{i-1}-1]$, the scheduling time of the *RD* or *WR* commands to a common bank $b_j + l$ is given by Eq. (9). Note that $\hat{t}_s(T_i) - 1$ is the finishing time of T_{i-1} according to Eq. (8). Eq. (9) can be used to conservatively determine the *ALAP* scheduling time of all *RD* or *WR* commands of T_{i-1}.

$$\begin{aligned}\hat{t}(RW^k_{j-1-(b_{com}-l)}) = \hat{t}_s(T_i) - 1 - (BC_{i-1} - 1 - k) \times tCCD \\ - (b_{com} - l) \times BC_{i-1} \times tCCD\end{aligned} \tag{9}$$

Given a finishing time of T_{i-1}, the scheduling time of its last *ACT* command is obtained since the minimum time interval between an *ACT* command and the first *RD* or *WR* command to the same bank is *tRCD* (see Table 1). Thus, with the minimum time interval between *ACT* commands, the scheduling time of the previous *ACT* commands is calculated by Eq. (10). 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 Eqs. (9) and (10), respectively. It is given by Eq. (11) based on two observations of the timing constraints in JEDEC DDR3 standard (JEDEC Solid State Technology Association 2010), namely: (1) *tRWTP* is larger for a write transaction than for a read transaction, and (2) there is $tRWTP > tRAS - tRCD$ for a write transaction. *Therefore, the worst-case initial states for T_i is that T_{i-1} is write rather than read*, and Eq. (11) is further simplified.

$$\begin{aligned}\hat{t}(ACT_{j-1-(b_{com}-l)}) = \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ - (b_{com} - l) \times max\{tRRD, BC_{i-1} \times tCCD\}\end{aligned} \tag{10}$$

$$\begin{aligned}
&\hat{t}(PRE_{j-1-(b_{com}-l)}) \\
&= max\left\{\hat{t}(ACT_{j-1-(b_{com}-l)}) + tRAS, \hat{t}(RW^{BC_{i-1}-1}_{j-1-(b_{com}-l)}) + tRWTP\right\} \\
&= \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD
\end{aligned} \tag{11}$$

Note that Eq. (9), (10) and (11) formalize *ALAP* command scheduling for T_{i-1}, leading to the worst-case initial bank states for T_i. This formalization is parameterized to BI_{i-1} and BC_{i-1} used by T_{i-1}. If more is known about T_{i-1}, e.g., its accessed banks, the *ALAP* can be specialized to obtain better analysis results. For example, bank privatization is employed by the PRET (Reineke et al. 2011) and ROC (Krishnapillai et al. 2014) memory controllers for different requestors. We leave the exploitation of this static knowledge to obtain a tighter WCET as future work.

8 Worst-case finishing time

Based on the worst-case initial bank states given by the *ALAP* scheduling in Sect. 7, we can compute the maximum scheduling time of commands for T_i, resulting in the worst-case finishing time $\hat{t}_f(T_i)$. Before deriving $\hat{t}_f(T_i)$, this section firstly proves that the off-line *ALAP* scheduling of the preceding transaction T_{i-1} indeed guarantees a conservative $\hat{t}_f(T_i)$. This is achieved by introducing Lemmas 2 and 3 that demonstrate the maximum scheduling time of the *ACT* commands for T_i only rely on the *ALAP* command scheduling of T_{i-1}. Finally, in Lemma 4, $\hat{t}_f(T_i)$ is computed based on Lemma 1.

8.1 Conservative $\hat{t}_f(T_i)$ based on *ALAP* scheduling

Intuitively, *ALAP* scheduling of commands for the previous write transaction T_{i-1} provides the worst-case initial bank states for T_i. However, the actual command scheduling for T_i may not only depend on T_{i-1} but also earlier transactions. Fig. 8 shows an example where T_i uses 4 banks from *Bank* 0 to *Bank* 3. T_{i-1} has the common banks *Bank* 0 and *Bank* 1 with T_i. T_{i-2} and T_{i-3} accessed *Bank* 2 and *Bank* 3, respectively. We can see that the *ACT* commands for T_i to the common banks *Bank* 0 and *Bank* 1 have to follow the constraints from the previous *WR* commands of T_{i-1}. For the non-common

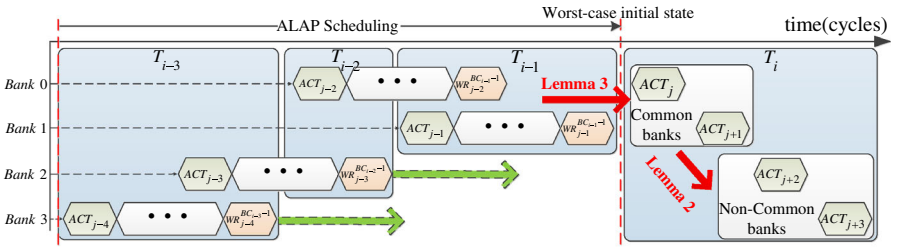


Fig. 8 An illustration of the *ALAP* scheduling that provides worst-case initial bank states for the current transaction T_i.

banks *Bank* 2 and *Bank* 3, the *ACT* commands of T_i may be scheduled according to the *WR* commands of earlier transactions T_{i-2} and T_{i-3} to the same bank. Moreover, *tFAW* must be satisfied between ACT_j of T_i and ACT_{j-4} of T_{i-3}.

We proceed by formally proving that the *ALAP* command scheduling of T_{i-1} guarantees a conservative $\hat{t_f}(T_i)$, even though earlier transactions (e.g., T_{i-2}, T_{i-3}) may actually have constraints that dominate in the command scheduling for T_i. This goal is achieved by three steps. As shown in Fig. 8, the first step is given by Lemma 2 stating that the scheduling of *ACT* commands of T_i to *non-common banks* with T_{i-1} is only determined by the *ACT* commands to the common banks. This indicates the constraints from earlier transactions T_{i-2} and T_{i-3}, as depicted by the green arrows in Fig. 8, cannot dominate in the scheduling of these *ACT* commands. The second step given by Lemma 3 guarantees that the *ACT* commands of T_i to *common banks* with T_{i-1} can be scheduled only dependent on the *ALAP* scheduling of commands of T_{i-1}, as shown in Fig. 8. As a result of Lemmas 2 and 3, the scheduling of *ACT* commands of T_i only depends on T_{i-1}. Finally, the third step computes the $\hat{t_f}(T_i)$ based on the *ALAP* command scheduling of T_{i-1} in Lemma 4. All the proofs are included in the appendix.

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

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

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

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

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

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

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

8.2 Worst-case finishing time

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

Lemma 3 illustrates that $t(ACT_{j+l})$ is determined by either the scheduling time $t(ACT_{j+l-1})$ of the previous *ACT* command or the last precharge time $t(PRE_{j-1-(b_{com}-l)})$ to the same bank, where $l \in [0, b_{com}]$. $\hat{t}(PRE_{j-1-(b_{com}-l)})$ is given by Eq. (11) according to the *ALAP* command scheduling for the previous write transaction T_{i-1}. Moreover, Lemma 2 shows that the scheduling time $t(ACT_{j+l})(l \in (b_{com}, BI_i - 1])$ of *ACT* commands to non-common banks is determined by $t(ACT_{j+b_{com}})$, which is the scheduling time of the *ACT* to the last common bank and can be computed with Lemma 3. As a result, $\hat{t}(ACT_{j+l})$ can be obtained by iteratively using Eq. (11), Lemmas 2 and 3. We proceed by introducing Lemma 4, which gives the worst-case finishing time of T_i. The proof is presented in the appendix. Intuitively, the worst-case finishing time of T_i is the worst-case starting time of T_i plus the maximum of all relevant timing dependencies (assuming an *ALAP* schedule).

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

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

9 Worst-case execution time

After deriving the worst-case finishing time in Sect. 8, this section proceeds by computing the WCET, the time between the worst-case starting time and the worst-case

finishing time. A generic parameterized WCET is first derived, followed by two interesting special cases, fixed transaction size and variable transaction sizes, respectively.

9.1 Generic worst-case execution time

According to Definition 5, the WCET is the difference between the worst-case starting time and the worst-case finishing time, which are both included in Lemma 4. Therefore, the WCET is obtained by rewriting Lemma 4. We observe that the expressions in the max{ } of Lemma 4 either linearly increase or decrease with l and l'. As a result, these expressions can be simplified to give the worst-case finishing time $\hat{t}_f(T_i)$ and hence the WCET $\hat{t}_{ET}(T_i)$. We proceed by introducing Theorem 1 that shows $\hat{t}_{ET}(T_i)$ is only determined by the JEDEC DDR3 timing constraints (JEDEC Solid State Technology Association 2010), and the sizes of T_i and T_{i-1} via (BI_{i-1}, BC_{i-1}) and (BI_i, BC_i) according to the chosen memory map configurations. Therefore, Theorem 1 provides a WCET parameterized by the sizes of T_i and T_{i-1}. The proof of Theorem 1 is presented in the appendix.

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

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

In general systems with variable transaction sizes, the specific size of the previous transaction that leads to WCET is unknown. We have found that the smallest previous transaction size must be assumed to derive a conservative WCET. This is later captured by Corollary 1. However, in the special case of the TDM arbitration presented in Sect. 4.2, the previous transaction size is known in the worst-case due to the static mapping of requestors to TDM slots. Therefore, less pessimistic WCET is obtained based on the known size of the previous transaction. A special case is that a system has a single fixed transaction size, such as × 64 Byte cache lines. As a result, the previous transaction size is statically known. The WCET for this special case is given by Corollary 2 in the next section. Note that the analysis of these two special cases only needs to instantiate Theorem 1 that is generic to any preceding and current transaction sizes.

Theorem 1 defines that the WCET $\hat{t}_{ET}(T_i)$ is parameterized by the sizes of T_i and T_{i-1}. We can observe that $\hat{t}_{ET}(T_i)$ increases when BI_{i-1} and BC_{i-1} decrease. By taking

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

Corollary 1 (Analytical WCET for variable transaction sizes) *For* $\forall i > 0$,

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

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

Corollary 2 (Analytical WCET for fixed transaction size) *For* $\forall i > 0$, $BI_{i-1} = BI_i = BI$ *and* $BC_{i-1} = BC_i = BC$,

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

9.2 Scheduled worst-case execution time

The analytical WCET given by Corollaries 1 and 2 have the benefit of being simple equations that bound the WCET by just inserting the timings of the particular memory device and the chosen memory map configuration for a transaction. However, they are somewhat pessimistic, since they conservatively assume that there is a command collision for every *ACT* command. Here, we present a second approach that builds on the presented formalism and ALAP schedule to overcome this limitation and derive a tighter bound.

The idea is to derive the worst-case initial bank state for a transaction based on the ALAP schedule as presented in Sect. 7.2, followed by actually scheduling the commands of the transaction off-line. *This has the advantage of only accounting for the actual number of command collisions and knowing exactly how many cycles the WCET increases due to the collisions*. The drawback of the approach is that it is no longer a

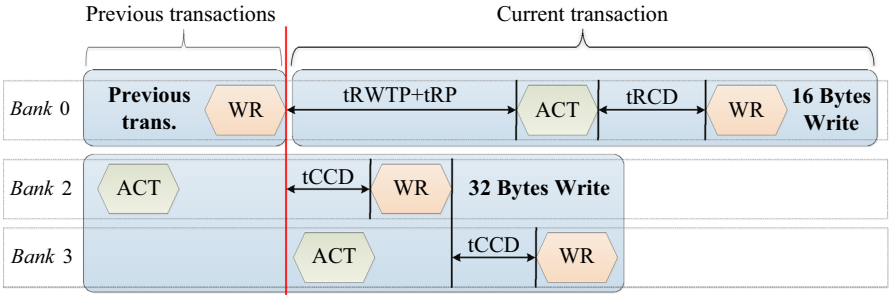


Fig. 9 An example illustrating that the actual execution time of a larger transaction (32 Bytes write) can be less than that of a smaller transaction (16 Bytes write).

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 Sect. 6, has been implemented as an open-source off-line scheduling tool (Li et al. 2014b). For the remainder of this article, we will refer to this approach as the *scheduled WCET* and the bounds obtained from Corollaries 1 and 2 as the *analytical WCET*. Both of them can be obtained from our tool (Li et al. 2014b).

9.3 Monotonicity of worst-case execution time

Intuitively, a transaction with a smaller size should have lower execution time than a larger one. However, it is not always true in the actual execution of transactions. The reason is that the execution time is highly dependent on the initial bank states for the current transaction, i.e., the bank accesses by previous transactions. Fig. 9 shows a counter example. The 32-Byte write transaction uses bank 2 and bank 3 and the corresponding ACT commands can be scheduled in a pipelined manner with the previous write transaction that uses bank 0. As a result, the scheduling of the *WR* commands is dominated by the *tCCD* constraint. In contrast, the 16-Byte write transaction accesses bank 0. It has to wait longer (*tRWTP+tRP*) to precharge bank 0 and then activate it. This shows that a smaller transaction may have a longer actual execution time.

However, the *WCET* of a smaller transaction cannot be larger than that of a larger transaction. This is guaranteed by Theorem 1 that shows the WCET of an arbitrary transaction T_i monotonically increases with BI_i and BC_i. Moreover, Definition 1 states that the transaction size is monotone with its *BI* and *BC*. Theorem 2 hence states that the WCET monotonically increases with transaction size. The proof is included in the appendix.

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

Theorem 2 allows us to use the WCET of the largest transaction that a requestor can issue as an upper bound for all its transactions. This is especially useful to relax

the requirement of fixed transaction size per requestor in the front-end (see Sect. 4) by conservatively using the largest transaction size from the requestor.

10 Worst-case response time in the front-end

The worst-case response time (WCRT) of a transaction represents the maximum time consumed to access the shared memory including both front-end and back-end. It is based on the WCET computed in Sect. 9. This section introduces the analysis of the WCRT based on the proposed front-end that uses a work-conserving TDM arbiter for requestors with variable transaction sizes, previously presented in Sect. 4.

As shown in Fig. 3, a transaction arrives at the head of the request queue at time t_0. In particular, the arrival of a write transaction is recognized as the time its last word arrives. The service of a read transaction is finished at the time $t_{lastword}$ when the last data word returns to the response queue. Similarly, a write transaction is finished when its acknowledgment arrives, which is sent by the back-end when the last write command is scheduled at time $t_{lastwrite}$, since the data is written into the memory afterwards. Therefore, the response time t_{RESP} of a read or a write transaction is defined by Definition 6.

Definition 6 (Response time of a transaction)

$$t_{RESP} = \begin{cases} t_{lastword} - t_0, & \text{Read transaction} \\ t_{lastwrite} - t_0, & \text{Write transaction} \end{cases} \tag{12}$$

A TDM arbiter is used in the front-end of the memory controller to serve transactions from different requestors. We assume the number of requestors is N. For an arbitrary requestor $r \in [0, N-1]$, the TDM arbiter allocates N_r consecutive TDM slots to it. Moreover, the TDM arbiter is configured to serve requestors in descending order of their transaction sizes to achieve smaller WCET, as discussed in Section 4.2. We assume the TDM arbiter serves requestors in the order from Requestor 0 to Requestor $N-1$, where Requestor 0 has the largest and Requestor $N-1$ has the smallest transactions.

The response time of a transaction from requestor r consists of the *interference delay* that is caused by other requestors, its own *execution time* in the back-end, and the time to return read data. As a result, the transaction experiences the worst-case response time (WCRT) only if its interference delay is maximum, after which it suffers its WCET in the back-end. With the proposed work-conserving TDM arbitration in Sect. 4.2, the maximum interference delay for a transaction occurs only if it misses any of its slots, causing all its following consecutive slots to be skipped by the arbiter, while the following requestors use all their allocated slots. Moreover, we have to conservatively assume that each transaction from requestor r is executed with the worst-case execution time $\hat{t}^r_{ET}$ in the back-end. Since the previous transaction size is known when using TDM arbitration, we use Theorem 1 to compute $\hat{t}^r_{ET}$. As a result, the WCET $\hat{t}^r_{ET}$ is less pessimistic than using Corollary 1, leading to a shorter TDM slot length. The TDM *frame size*, which is the sum of all slot lengths in the TDM

table (given by Definition 7), is hence smaller. We experimentally show the benefits of this approach in Sect. 11.4.3.

Definition 7 (Frame size of the TDM table) The frame size $FS = \sum_{r=0}^{N-1} N_r \times \hat{t}^r_{ET}$.

The worst-case response time $\hat{t}^r_{RESP}$ of a transaction from requestor r comprises three parts, as shown in Eq. (13). $\hat{t}^r_{interf}$ is the maximum interference delay for requestor r, which is given by Eq. (14). It is the sum of the WCET of transactions from all other requestors that are executed within their slots. The WCET results of these transactions are given by Theorem 1 with known previous transaction sizes. For the first interfering transaction, its WCET is computed assuming its preceding transaction has the minimum size in the TDM table. This results in conservative WCET of the first interfering transaction, since its previous transaction may be from any requestor and is hence unknown. The second part of Eq. (13) is the worst-case execution time of the transaction. Since the execution time of a transaction finishes when the last *RD* or *WR* command is scheduled, the Δt (the third part of Eq. (13)) represents the extra time spent on returning the data of the last RD command to the response buffer and is given by Eq. (15) that only comprises JEDEC specified timings.

$$\hat{t}^r_{RESP} = \hat{t}^r_{interf} + \hat{t}^r_{ET} + \Delta t \tag{13}$$

$$\hat{t}^r_{interf} = \sum_{\forall r' \in [0,N-1], r' \neq r} \hat{t}^{r'}_{ET} \times N_{r'} \tag{14}$$

$$\Delta t = \begin{cases} tRL + BL/2, & \text{Read transaction} \\ 0, & \text{Write transaction} \end{cases} \tag{15}$$

Finally, the transaction may be delayed by a refresh. The worst-case refresh delay $\hat{t}_{REF}$ is given by Eq. (16), which consists of the time between the last *WR* command of the previous transaction and the associated *PRE* and the precharge period as well as the refresh period. However, a refresh is regularly needed every *tREFI* cycles, i.e. a relatively long period of $7.8\mu s$. Therefore, the penalty caused by refreshing leads to only about 3 % increase in the total delay of accessing memory for an application. As a result, it is not added to the WCRT of each transaction to avoid pessimism, but added as an average cost in the system-level analysis of the application.

$$\hat{t}_{REF} = tRWTP + tRP + tRFC \tag{16}$$

11 Experimental results

This section experimentally evaluates our memory controller and the corresponding analysis. First, the experimental setup is presented, followed by three experiments. The first experiment shows that the formalization accurately describes the timing behavior of the back-end. The last two experiments evaluate our analysis for fixed transaction size and variable transaction sizes, respectively. The results in terms of WCET, WCRT, and the average execution time are analyzed and compared to a state-of-the-art semi-static approach (Akesson and Goossens 2011).

Table 3 An overview of the experiments.

Experiment	*Trans Size*	*Content*	*Section*
1	Any	Formalization validation	11.2
2	Fixed	Execution time	11.3.1
3	Fixed	Response time	11.3.2
4	Variable	Execution time	11.4.1
5	Variable	Effect of preceding transaction size	11.4.2
6	Variable	Effect of TDM service orders	11.4.3
7	Variable	Response time	11.4.4
8	Variable	WCET monotonicity	11.5

11.1 Experimental setup

Our memory controller is implemented as a cycle-accurate SystemC model. The experiments use a combination of independent real application traces and synthetic traffic. Each of them generates one transaction stream and they are reflected by a mixed stream in the memory controller back-end. The WCRT of a transaction derived in Sect. 10 does not cover the aspects, such as synchronization within a single application. However, these issues should be addressed by using the WCRT of transactions in a higher-level formalism (e.g., dataflow model and network calculus) for WCET estimation of applications. We use application traces generated by running applications from the MediaBench benchmark suite (Lee et al. 1997) on the SimpleScalar 3.0 processor simulator (Austin et al. 2002), which uses separate data and instruction caches, each with a size of 16 KB. The L2 caches are private unified 128 KB caches where the cache-line size varies depending on the experiments. Synthetic traffic is generated using a normal distribution with very low variance, resulting in near-periodic traffic inspired by e.g., some hardware accelerators and display controllers in the multimedia domain. For each transaction size in the experiments, we have chosen the memory map configuration that provides the lowest execution time for transactions by interleaving more banks to exploit bank parallelism. The configured (*BI*, *BC*) for transaction sizes of 16 Bytes, 32 Bytes, 64 Bytes and 128 Bytes are hence (1, 1), (2, 1), (4, 1) and (4, 2), respectively (Goossens et al. 2012). (4, 2) is used by 128 byte transactions instead of (8, 1) because of *tFAW* that causes a larger execution time with (8, 1). 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 (JEDEC Solid State Technology Association 2010). Table 3 shows an overview of all these experiments.

11.2 Experimental validation of the formalization

The purpose of our first experiment is to validate the formalization of the timing behavior of the dynamic command scheduling Algorithm 2 by verifying that the scheduling

Table 4 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

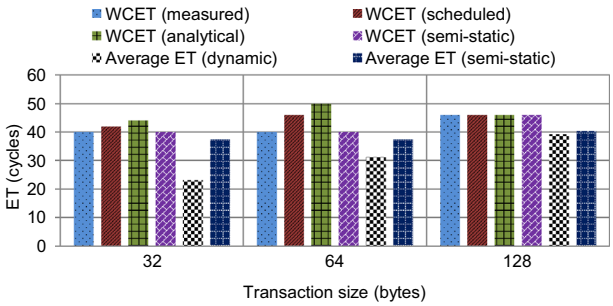
time of each command is the same as given by the SystemC implementation. To this end, the open-source off-line scheduling tool (Li et al. 2014b) that implements the formalism has been provided with the same inputs as the SystemC implementation for all experiments in this article, covering a wide range of read and write transactions with different sizes and inter-arrival time under different memory map configurations. The results of this experiment are that all commands of all transactions are scheduled *identically*, indicating that the formalization accurately captures the implementation. This is important since the formalization forms the base for both the analytical and the scheduled WCET bounds. Moreover, it suggests the SystemC implementation is correct. *This proven relation between the formal model and the implementation is an important result of our work and a distinguishing feature compared to the related work.*

11.3 Fixed transaction size

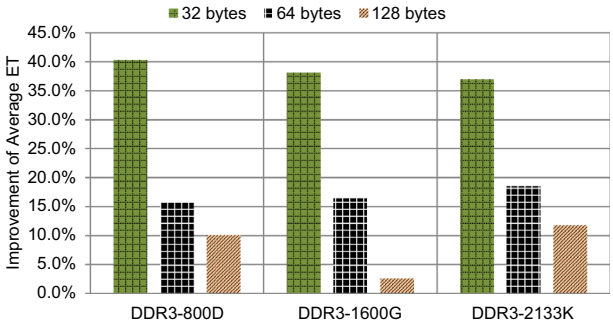
This experiment evaluates our approach for transactions with fixed transaction size, and compares both the worst-case and average results to a semi-static approach (Akesson and Goossens 2011), the only other approach that supports different memory map configurations. Four memory requestors are used, corresponding to four processors executing different Mediabench applications (*gsmdecode*, *epic*, *unepic* and *jpegencode*). The TDM arbiter in the front-end allocates one slot per requestor. For each application, the total number of transactions (*TransN*) and the ratio (or percentage) of read transactions (*RRatio*) are shown in Table 4. The processors execute through their L2 cache and have the same cache-line size, enabling Corollary 2 to be used to bound the WCET. The experiment is executed for three different cache-line sizes of 32 Bytes, 64 Bytes and 128 Bytes with different memory map configurations, respectively.

11.3.1 Execution time

The execution time of a transaction is the time required by the back-end to schedule commands to the memory. This experiment hence only evaluates the dynamically scheduled back-end and that the front-end will be included later when evaluating response times. The WCET and average execution time of transactions with fixed size accessing a DDR3-1600G memory are presented in Fig. 10a. The results for other memories are similar and not shown. We can observe that:



(a) DDR3-1600G



(b) Comparison in average ET

Fig. 10 WCET and average ET for different DDR3 SDRAMs with fixed transaction size. Results are compared to a semi-static approach (Akesson and Goossens 2011).

(1) The maximum measured WCET from the experiments is equal to or slightly smaller than the scheduled WCET. This indicates that the proposed analysis provides a tight WCET bound. The scheduled WCET is a little too conservative for some transaction sizes, e.g., 32 bytes and 64 bytes for DDR3-1600G that use $BC = 1$. This is caused by the worst-case initial states determined by the *ALAP* scheduling in Sect. 7.2, which is conservative since *tCCD* is used as the time interval between two *RD* or *WR* commands. However, for $BC = 1$, the actual interval is larger than *tCCD* because *ACT* command dominates in the scheduling of a *RD* or *WR* command. This conservative is eliminated for 128 byte transactions that have $BI = 4$ and $BC = 2$, where the *ALAP* scheduling accurately determines the worst-case initial states.

(2) The analytical WCET derived from Corollary 2 is equal to or slightly larger than the scheduled WCET. The difference is because Theorem 1 conservatively assumes a collision per *ACT* command, which may not actually be the case and the collisions do not lead to an increased execution time, since the *ACT* command does not always dominate in the computation of the finishing time (see Lemma 1). The maximum difference is *BI* cycles.

(3) The WCET given by the semi-static approach is identical to the measured WCET of our approach. Note that there is only one exception for 32 byte transactions with

DDR3-2133K, where the measured WCET given by our approach is 61 cycles, while it is 59 cycles for the semi-static approach. Since this exception is highly dependent on the timing constraints, it does not occur for most DDR3 SDRAMs. For example, there is no such exception for DDR3-800D and the results have been presented in Li et al. (2014a).

(4) Our memory controller has significantly better average ET than the semi-static approach for all DDR3 SDRAMs, as shown in Fig. 10a, where DDR3-1600G is taken as an example. This is because dynamic command scheduling monitors the actual state of the required banks and issues commands earlier for a transaction that requires a different set of banks from that of the previous transaction. In contrast, the semi-static scheduling (Akesson and Goossens 2011) uses precomputed schedules that always assume worst-case initial bank state for every transaction. Fig. 10b shows the improvement of average ET, which is defined as $100\,\% \times (1 - \bar{t}^d_{ET}/\bar{t}^s_{ET})$. $\bar{t}^d_{ET}$ and $\bar{t}^s_{ET}$ denote the average ET of our approach and the semi-static approach, respectively.

(5) We see that smaller transactions benefit more from dynamic command scheduling. For example with DDR3-1600G, 32 byte transactions gain 38.1 % while 128 byte transactions gain 2.6 %. The reason is that smaller transactions require a fewer banks, leading to higher chance for the next transaction to access the different banks and can thus be scheduled earlier.

11.3.2 Response time

The WCRT of a transaction is given by Eq. (13): it is essentially determined by accumulating the WCET of transactions from each requestor. This experiment shows both the worst-case and average-case response time of a transaction. Fig. 11 presents the WCRT for DDR3-1600G with fixed transaction sizes. The results are derived on the basis of the WCET shown in Fig. 10a, and new observations from Fig. 11 include:

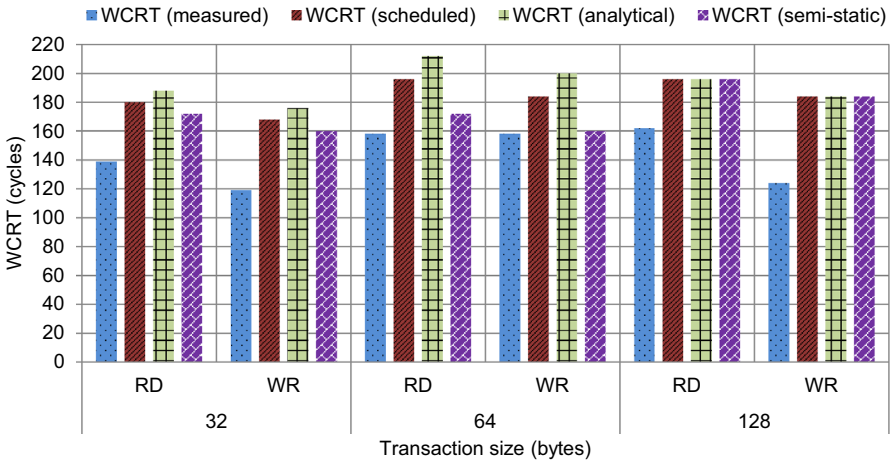


Fig. 11 WCRT for DDR3-1600G with fixed transaction size.

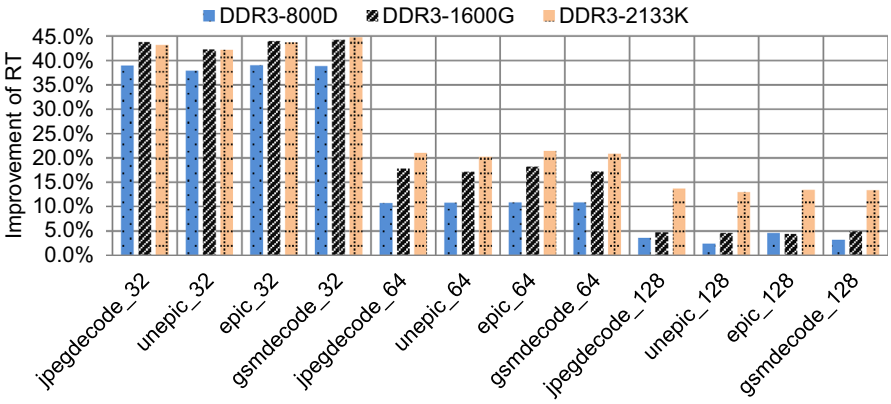


Fig. 12 Comparison to a semi-static approach 2011 in average response time of Mediabench application traces for different DDR3 SDRAMs with fixed transaction size.

(1) the response time of transactions are bounded. The measured WCRT is smaller than the bound in terms of scheduled and analytical WCRT. The difference between them is because the worst-case situation is unlikely to occur in both the front-end and back-end simultaneously, which requires transactions from all requestors competing in the front-end, while each transaction in the back-end experiences worst-case initial bank state. (2) The analytical WCRT is more pessimistic than the scheduled WCRT, because the analytical WCRT is derived by accumulating the analytical WCET of transactions from each requestor. This exaggerates the conservative assumption of a collision per ACT command for computing the analytical WCET. These observations also hold for other DDR3 SDRAMs, although their WCRT results are not presented for brevity.

Our dynamically scheduled memory controller has significantly smaller average response time (RT) compared to the semi-static approach for all DDR3 memories. Fig. 12 shows the gained improvement percentage of average RT by using our approach compared to the semi-static approach for various Mediabench application traces. The improvement is defined similarly to that of average ET. Fig. 12 supports the observations (4) and (5) from Fig. 10(b) as given in Sect. 11.3.1 that significantly better average RT is achieved while smaller transaction size benefits more from our dynamically scheduled approach.

11.4 Variable transaction size

The last experiment evaluates our approach with variable transaction size. First, the WCET of a transaction is evaluated without any priori information, e.g., the size of the previous transaction, where the worst-case is assumed. Second, we experiment the case when the size of the previous transaction is known, which is guaranteed by the TDM arbiter. Then the impact of the service order of requestors with their transaction sizes on the WCRT is evaluated, based on which the WCRT and average response time are evaluated. The setup is loosely inspired by a High-Definition video

and graphics processing system featuring a number of CPU, GPU, hardware accelerators and peripherals with variable transaction sizes. This system has 4 requestors with the transaction sizes of 16 bytes, 32 bytes, 64 bytes and 128 bytes, respectively. The first requestor Req_1 represents a GPU with 128 byte cache line size, executing a Mediabench application *jpegdecode*. A video engine corresponding to requestor, Req_2, is used for *mpeg2decode* and it generates memory transactions of 64 bytes. The Mediabench application *epic* is executed by a processor with a cache-line size of 32 bytes, which is denoted Req_3. A synthetic memory trace is used by a CPU which has a 16 byte cache-line size, resulting in read and write transactions with 16 bytes. This is requestor Req_4. The TDM arbiter in the front-end allocates one slot per requestor and it serves these requestors from Req_1 to Req_4 in descending order of their transaction sizes.

11.4.1 Execution time

Corollary 1 is used to compute the WCET of transactions with variable size, and the results for DDR3-1600G are shown in Fig. 13. It also shows the WCET results given by the semi-static approach for particular sizes, including 16 bytes, 32 bytes, 64 bytes and 128 bytes, respectively. It is worth noting that the static command schedules (also named patterns) used by the semi-static approach are computed at design time for a particular fixed size, and are configured before the system is running. We get similar conclusions as previously presented in Sect. 11.3.1. New interesting observations are: (1) the scheduled WCET bound is perfectly tight, since the worst-case situation for a transaction is accurately captured by Corollary 1 for variable sizes, and actually occurs during simulation. The situation is that the previous transaction is a write and its finishing bank is the starting bank of the new transaction. (2) when the semi-static approach is used for variable transaction sizes, it has to choose a particular pattern size such that the total WCET of all requestors is minimum, leading to smaller WCRT. For a particular pattern size, transactions with larger sizes have to be split into several pieces that are served in consecutive TDM slots. If the transaction size is smaller than the pattern size, it will fetch the data and throw the unnecessary part away. This has two consequences. First, the WCET of transactions with variable sizes highly depend on the chosen pattern size. For example, the 16 bytes pattern provides very high WCET

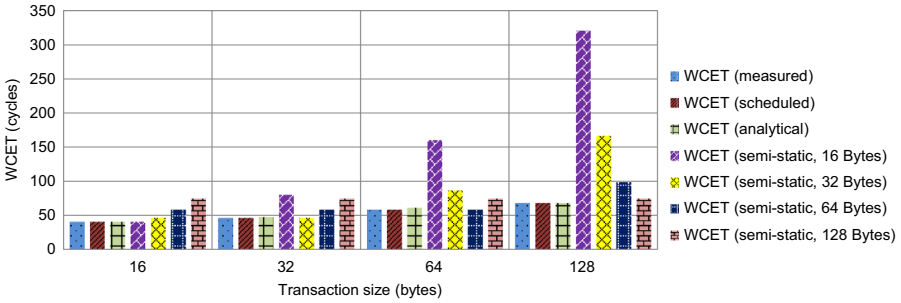


Fig. 13 WCET for DDR3-1600G with variable transaction sizes.

for larger transaction sizes, as shown in Fig. 13. Second, since data is discarded, it wastes power and reduces the bandwidth provided by the SDRAM, which is a scarce resource. The bandwidth analysis is out of the scope of this article. The best pattern size depends on the mix of the transaction sizes and the timing constraints of the memory. For example, the best pattern size used in our experiments for DDR3-1600G is 128 bytes, while it is 64 bytes and 128 bytes for DDR3-800D and DDR3-2133K, respectively. (3) the WCET for each transaction obtained from our approach is less than or equal to than that of the semi-static approach. This demonstrates our dynamically scheduled memory controller outperforms the semi-static approach in the worst case with variable sizes. (4) moreover, the average execution time of transactions with variable sizes are much lower just like in the case of fixed sizes. The average execution time results are not shown.

11.4.2 WCET with known/unknown previous transaction size

The WCET of a transaction is given by Corollary 1 for unknown previous transaction size, referred to as pre-size, while it is provided by Theorem 1 for known pre-size. As discussed in Sect. 9, if there is no static information about the size of the previous transaction, it has to assume the worst-case situation for a transaction that its starting bank was the finishing bank of the previous write transaction. This results in pessimism for the WCET given by Corollary 1. The TDM arbiter in the front-end provides static information about the slot allocation per requestor. Therefore, the size of the previous transaction is statically known in the worst case. In this experiment, four requestors have transaction sizes of 128 bytes, 64 bytes, 32 bytes and 16 bytes, respectively. The TDM arbiter allocates one slot per requestor and serves them in descending order of sizes, e.g., from 128 bytes to 16 bytes. Fig. 14 shows the WCET of a transaction with known and unknown size of the previous transaction for DDR3 SDRAMs, respectively. We can see that the WCET with unknown previous transaction size is greater than or equal to the case with known size. For example, a 128 byte transaction is preceded by a 16 byte transaction consisting of one burst, leading to no difference for its WCET if the previous size is known or unknown. In contrast, a 64 byte transaction is preceded by a 128 byte transaction. Its starting bank cannot be the finishing bank of

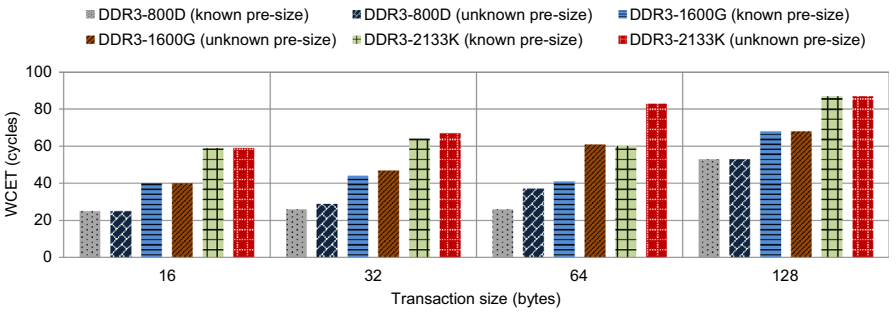
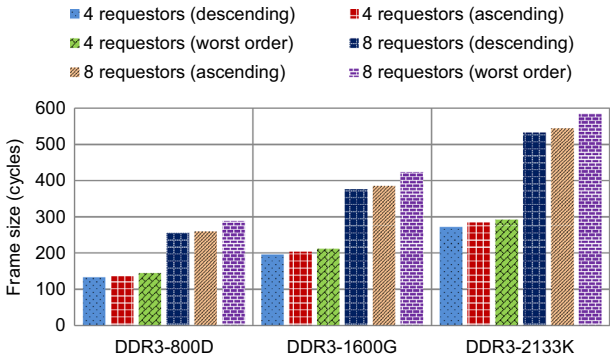


Fig. 14 WCET with known/unknown previous transaction size. Requestors are allocated to TDM slots in descending order of their transaction sizes.

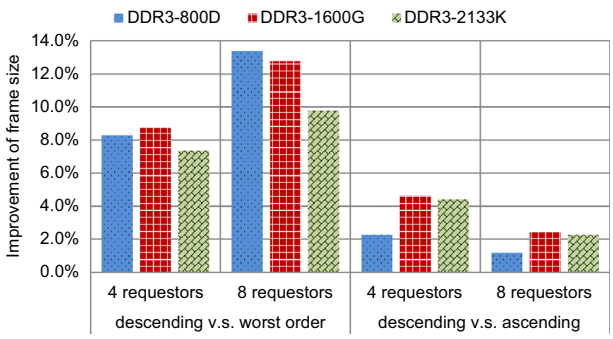
the 128 byte transaction for aligned transactions, resulting in much better WCET with known previous transaction size (see Fig. 14). Therefore, shorter worst-case frame size is obtained if the size of previous transaction is known. This leads to smaller WCRT as presented in the following section.

11.4.3 TDM service order of requestors

Besides known size of the previous transaction, lower WCET is obtained if transactions are executed in descending order of their sizes because of improved pipelining between successive transactions, as previously discussed in Sect. 4.2. This results in a shorter frame size. An experiment is carried out to explore all the possible orders of serving 4 and 8 requestors with transaction sizes of 16 byte, 32 byte, 64 byte and 128 byte, respectively. For the case of 8 requestors, there are two requestors with each transaction size. Each requestor has one slot in the TDM table. All the possible orders of serving these requestors have been evaluated, although only frame sizes for descending, ascending and the worst possible order are shown in Fig. 15a. The best way to serve requestors is descending order of their transaction sizes. The worst order



(a) The worst-case frame size



(b) The improvement of frame size

Fig. 15 The worst-case frame size of a TDM table for different number of requestors, and the improvement by serving requestors in descending order of their transaction sizes.

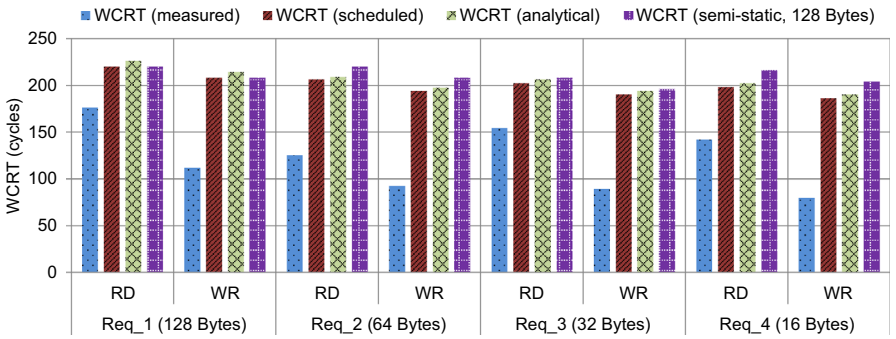


Fig. 16 WCRT for DDR3-1600G with variable transaction sizes.

is the one that results in the maximum frame size. The experiment shows that the minimum frame size is always obtained using the descending order. Compared to the worst order, the improved percentage of frame size by using descending order is given by Fig. 15b. It indicates that a system with a larger number of requrestors benefits more from the descending order, e.g., 13.4 % is gained for 8 requestors with DDR3-800D. Note that this is a free improvement by using our analysis in Sect. 9.1, which provides the WCET by exploiting more detailed information about the bank state when the size of the previous transaction is known. This has not been considered by existing work.

11.4.4 Response time

The WCRT for the four requestors is derived from Eq. (13) and the results for DDR3-1600G are shown in Fig. 16. They are obtained on the basis of the WCET by using Theorem 1. In addition, to fairly compare with the semi-static approach, we choose the best pattern size, e.g., 128 byte for DDR3-1600G. As can be seen from Fig. 16, it also supports the conclusion given by Fig. 13 that our dynamically scheduled memory controller outperforms the semi-static approach in worst case, since the WCRT given by the semi-static approach is greater than or equal to that of our approach for transactions from different requestors. As the observation also holds for the other DDR3 SDRAMs, their results are not shown. Moreover, by collecting the average response time of transactions from different requestors, we can conclude similarly to Sect. 11.3.2 that our approach significantly reduces the time for each application to access the memory. For example, compared to the semi-static approach, 53.3 % reduction of the average response time for accessing DDR3-1600G is achieved by the Mediabench application *epic* that has 32 byte memory transactions.

11.5 Monotonicity of WCET

Theorem 2 states that the analytical WCET monotonically increases with the transaction size, and it is based on the WCET given by Theorem 1. However, we cannot prove this for the scheduled WCET, as presented in Sect. 9.2. We proceed by provid-

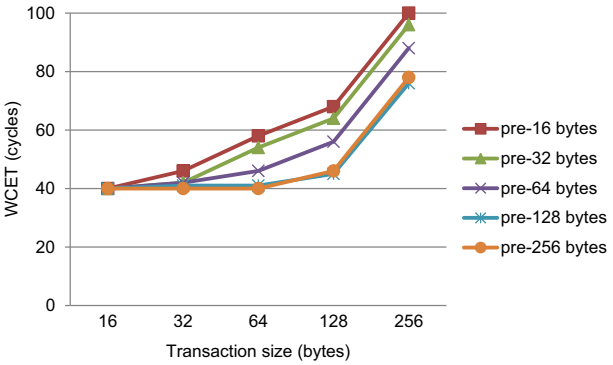


Fig. 17 The monotonicity of scheduled WCET with transaction size for a requestor. DDR3-1600G is taken as an example.

ing experimental evidence to show that the monotonicity property also holds for the scheduled approach.

Experiments have been done with DDR3-800D, DDR3-1600G and DDR3-2133K to collect the scheduled WCET of transactions. All pair-wise combinations of 16, 32, 64, 128, and 256 bytes transactions have been tested. Fig. 17 shows the scheduled WCET results of transactions with different sizes under different preceding transaction sizes for DDR3-1600G. The results are similar for the other memories and are not shown for brevity. Using Theorem 1 we conclude that *the scheduled WCET monotonically increases with the transaction size for DDR3-800D/1600G/2133K memories*.

12 Conclusions

This article provides tight WCRT bounds for memory transactions of real-time applications, while offering competitive low average response time to transactions of NRT applications. To this end, we defined a memory command scheduling algorithm and architecture supporting both fixed and variable transaction sizes with different memory map configurations by dynamically scheduling commands. In addition, a formalization of the dynamic command scheduling is proposed to capture the timings of commands, based on which the WCET of transactions is defined. Based on the analysis, two techniques are presented to bound the WCET. The first technique is an equation that computes the WCET for a given transaction size and memory map configuration, while the second technique tries to provide a tighter bound by using an off-line implementation of the dynamic command scheduling to compute actual command collisions. We formally prove that the analytical WCET monotonically increases with the transaction size, and we provide experimental evidence of DDR3-800D/1600G/2133K SDRAMs that this also holds for the scheduled approach. With the WCET of transactions, the WCRT is derived based on a new work-conserving TDM arbiter that schedules transactions from different requestors in a way that provides low average response time without negative impacting the worst case. Comparison with a state-of-the-art semi-static scheduling approach shows that our approach significantly reduces the average

response times, implying shorter time for each application to access the memory, while it performs equally well or better in the worst-case with only a few exceptions.

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.

Appendix

Proof of Lemma 1

Proof To prove Lemma 1 that states the finishing time of a transaction is only determined by either the finishing time of the previous transaction, or the scheduling time of its *ACT* commands, we only need to iteratively compute the scheduling time of its *RD* or *WR* commands and finally obtain the scheduling time of the last *RD* or *WR* command, which is defined as the finishing time of the transaction by Definition 3.

For an arbitrary transaction T_i ($i > 0$) that has BI_i and BC_i, its finishing time $t_f(T_i)$ is shown in Eq. (17), which is the scheduling time of its last *RD* or *WR* (named *RW*) command.

$$t_f(T_i) = t(RW^{BC_i-1}_{j+BI_i-1}) \tag{17}$$

According to Eq. (6) that gives the scheduling of a *RW* command, the scheduling time of the last *RW* command in Eq. (17) is given by Eq. (18). We see that this is determined by the scheduling time of the first *RW* to the same bank.

$$t(RW^{BC_i-1}_{j+BI_i-1}) = t(RW^{0}_{j+BI_i-1}) + (BC_i - 1) \times tCCD \tag{18}$$

Equation (5) provides the scheduling time of the first *RW* command to a bank, which is determined by either the scheduling time of the *ACT* command to the same bank due to *tRCD*, or that of the previously scheduled *RW* command for the same transaction because of *tCCD*, which is the last command to the previous bank. As a result, $t(RW^{0}_{j+BI_i-1})$ in Eq. (18) is derived based on Eq. (5), and is shown in Eq. (19).

$$t(RW^{0}_{j+BI_i-1}) = max\{t(ACT_{j+BI_i-1}) + tRCD, t(RW^{BC_i-1}_{j+BI_i-2}) + tCCD\} \tag{19}$$

We proceed by combining Eq. (17), (18) and (19) to obtain a new expression of the finishing time, as given by Eq. (20).

$$\begin{aligned} t_f(T_i) = max\{&t(ACT_{j+BI_i-1}) + tRCD + (BC_i - 1) \times tCCD, \\ &t(RW^{BC_i-1}_{j+BI_i-2}) + BC_i \times tCCD\} \end{aligned} \tag{20}$$

In the same way, we can compute $t(RW_{j+BI_i-2}^{BC_i-1})$ in Eq. (20), and it can be further expressed by $t(ACT_{j+BI_i-2})$ and $t(RW_{j+BI_i-3}^{BC_i-1})$. We iteratively substitute the scheduling time of the last *RW* command to each bank of transaction T_i, and Eq. (21) is derived, which consists of BI_i number of terms.

$$t_f(T_i) = \underset{1\le l\le BI_i-1}{Max}\{t(RW_j^{BC_i-1}) + (BI_i - 1)\times BC_i \times tCCD, \\ t(ACT_{j+l}) + tRCD + [(BI_i - l)\times BC_i - 1]\times tCCD\} \tag{21}$$

We proceed by substituting $t(RW_j^{BC_i-1})$ in Eq. (21) with the scheduling time of the first *RW* to the same bank, which is given by Eq. (22), and is derived according to Eq. (6).

$$t(RW_j^{BC_i-1}) = t(RW_j^0) + (BC_i - 1)\times tCCD \tag{22}$$

Furthermore, $t(RW_j^0)$ in Eq. (22) can be computed based on Eq. (5). As a result, Eq. (23) is obtained. Note that $t_f(T_{i-1})$ is the finishing time of the previous transaction T_{i-1} that is also the scheduling time of the last *RW* command of the previous bank. It was scheduled just before RW_j^0 and the timing constraint between them is *tSwitch* (given by Eq. (2)).

$$t(RW_j^0) = \max\{t(ACT_j) + tRCD, t_f(T_{i-1}) + tSwitch\} \tag{23}$$

By combining Eq. (21), (22) and (23), Eq. (24) is derived.

$$t_f(T_i) = \underset{0\le l\le BI_i-1}{Max}\{t_f(T_{i-1}) + tSwitch + (BI_i \times BC_i - 1)\times tCCD, \\ t(ACT_{j+l}) + tRCD + [(BI_i - l)\times BC_i - 1]\times tCCD\} \tag{24}$$

Hence, for $\forall l \in [0, BI_i - 1]$, $t_f(T_i)$ is expressed by Eq. (24). It indicates that $t_f(T_i)$ only depends on the scheduling times of its *ACT* commands, the finishing time of T_{i-1}, the memory map configuration in terms of BI_i and BC_i and the JEDEC-specified timing constraints, which are constant values. □

Proof of Lemma 2

Proof For $\forall l \in (b_{com}, BI_i - 1]$, the scheduling time of the command ACT_{j+l} to bank b_j+l can be obtained from Eq. (4). It indicates $t(ACT_{j+l})$ is determined by $t(ACT_{j+l-1})$, $t(ACT_{j+l-4})$ or $t(PRE_m)$, where m was the latest bank access number to bank $b_j + l$ before T_i. This lemma can be proved by simplifying Eq. (4) to derive the scheduling time of ACT_{j+l}, which is finally given by Eq. (34). First, a simplified Eq. (25) is obtained because of the dominance of $t(ACT_{j+l-1})$ in this case. We proceed by explaining its derivation.

$$\begin{aligned} t(ACT_{j+l}) = & max\{t(ACT_{j+l-1}) + tRRD, t(PRE_m) + tRP, \\ & t(ACT_{j+l-4}) + tFAW\} + C(j+l) \\ = & t(ACT_{j+l-1}) + tRRD + C(j+l) \end{aligned} \tag{25}$$

$t(ACT_{j+l-1})$ dominates in the max{} of Eq. (25). We demonstrate this by showing two relations between the terms in the expression: i) $t(ACT_{j+l-1}) > t(PRE_m) + tRP$. For $\forall l > b_{com}$, Eq. (4) is employed to derive Eq. (26), which shows a later bank access (larger l) has a larger scheduling time of the *ACT* command. This is intuitive since the scheduling algorithm (Algorithm 2) schedules *ACT* commands in order.

$$BI_i - 1 \geq \forall l > b_{com} \Longrightarrow t(ACT_{j+l-1}) \geq t(ACT_{j+b_{com}}) \tag{26}$$

The command $ACT_{j+b_{com}}$ is scheduled to bank $b_j + b_{com} = b_{j-1}$ that is the finishing bank of T_{i-1}. As a result, Eq. (27) is derived on the basis of Eq. (4). It simply states that a bank cannot be activated until it has been precharged.

$$b_j + b_{com} = b_{j-1} \Longrightarrow t(ACT_{j+b_{com}}) \geq t(PRE_{j-1}) + tRP \tag{27}$$

Moreover, the precharge of a bank is triggered by the auto-precharge flag appended to a *RD* or *WR* command, which is issued sequentially. Therefore, banks are precharged in the order of bank accesses, resulting in Eq. (28), where the latest access number m for bank $b_j + l$ is smaller than the latest bank access number $j-1$. Finally, by substituting Eq. (26), (27) and (28), we can prove the relation that $t(ACT_{j+l-1}) > t(PRE_m) + tRP$.

$$\forall m < j - 1 \Longrightarrow t(PRE_{j-1}) > t(PRE_m) \tag{28}$$

ii) $t(ACT_{j+l-1}) > t(ACT_{j+l-4}) + tFAW$. According to Eq. (7), we can obtain Eq. (29), which shows that the precharging time of a bank is after issuing the last *RD* or *WR* command of a transaction to the same bank.

$$t(PRE_{j-1}) \geq t(RW_{j-1}^{BC_{i-1}-1}) + tRWTP \tag{29}$$

With Eq. (5) and (6) that capture the timing dependencies for a *RD* or *WR* command, Eq. (30) is derived and it indicates that the last *RD* or *WR* command of a transaction to a bank is scheduled later than the *ACT* command to the same bank.

$$t(RW_{j-1}^{BC_{i-1}-1}) \geq t(ACT_{j-1}) + tRCD + (BC_{i-1} - 1) \times tCCD \tag{30}$$

Since *ACT* commands are scheduled in order by Algorithm 2, the previously scheduled command ACT_{j+l-4} ($l < 4$) was not scheduled later than that of ACT_{j-1}. We can get Eq. (31).

$$\forall l < 4 \Longrightarrow t(ACT_{j-1}) \geq t(ACT_{j+l-4}) \tag{31}$$

By combining Eq. (29), (30), and (31), Eq. (32) is derived.

$$t(PRE_{j-1}) \geq t(ACT_{j+l-4}) + tRCD + (BC_{i-1} - 1) \times tCCD + tRWTP \tag{32}$$

We now proceed by obtaining Eq. (33) based on the combination of Eq. (26), (27), and (32). Moreover, we can observe $tFAW \leq tRC = tRAS + tRP \leq tRCD + tRWTP + tRP$ for all DDR3 devices from the JEDEC DDR3 timing constraints 2010. Therefore, $t(ACT_{j+l-1}) > t(ACT_{j+l-4}) + tFAW$ according to Eq. (33), proving the second relation.

$$t(ACT_{j+l-1}) > t(ACT_{j+l-4}) + tRCD + (BC_{i-1} - 1) \times tCCD + tRWTP + tRP \quad (33)$$

With the above two reasons, the simplified equation is given by Eq. (25). It indicates the scheduling time of ACT_{j+l} is only determined by that of the previous ACT_{j+l-1}. Based on Eq. (25) and $\forall l \in (b_{com}, BI_i - 1]$, we can get Eq. (34), which shows the scheduling time of ACT_{j+l} depends on that of $ACT_{j+b_{com}}$. Note that $ACT_{j+b_{com}}$ was scheduled to the last bank b_{j-1} of T_{i-1}.

$$t(ACT_{j+l}) = t(ACT_{j+b_{com}}) + [l - b_{com}] \times tRRD + \sum_{l'=b_{com}+1}^{l} C(j + l') \quad (34)$$

□

Proof of Lemma 3

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

We proceed by proving that the scheduling of *ACT* commands for T_i to common banks is only dependent on T_{i-1}. For a common bank $b_j + l$ between T_{i-1} and T_i where $\forall l \in [0, b_{com}]$, the scheduling time of its *ACT* command ACT_{j+l} is obtained from Eq. (4) and is shown in Eq. (35), which indicates that $t(ACT_{j+l})$ depends on the scheduling time $t(ACT_{j+l-1})$ of the previous *ACT*, the scheduling time $t(PRE_{j-1-(b_{com}-l)})$ of the latest *PRE* to bank $b_j + l$ and the scheduling time $t(ACT_{j+l-4})$ of the fourth previous *ACT* command (due to *tFAW*). Note that $j - 1 - (b_{com} - l)$ is the latest access number to bank $b_j + l$ according to the *ALAP* command scheduling of T_{i-1}. For example, if $l = b_{com}$, bank $b_j + l = b_j + b_{com}$ is the last common bank between T_{i-1} and T_i, where its latest access number is $j - 1$. Since ACT_{j+l-1} is a command for T_i or T_{i-1} ($l = 0$) while $PRE_{j-1-(b_{com}-l)}$ was for T_{i-1}, only ACT_{j+l-4} is possible to be a command for

earlier transactions, e.g., T_{i-2} or T_{i-3}, if $BI_{i-1} < 4$. To prove the lemma, we only need to prove $t(ACT_{j+l-4})$ does not dominate in the max{ } of Eq. (35), which is then further simplified as shown in Eq. (35).

$$\begin{aligned} t(ACT_{j+l}) &= max\{t(ACT_{j+l-1}) + tRRD, t(PRE_{j-1-(b_{com}-l)}) + tRP, \\ &\qquad t(ACT_{j+l-4}) + tFAW\} + C(j+l) \\ &= max\{t(ACT_{j+l-1}) + tRRD, t(PRE_{j-1-(b_{com}-l)}) + tRP\} \\ &\quad + C(j+l) \end{aligned} \tag{35}$$

For $\forall l < 4 - BI_{i-1}$, ACT_{j+l-4} is a command for earlier transactions, e.g., T_{i-2} or T_{i-3} when $BI_{i-1} < 4$. We proceed by computing the scheduling time of ACT_{j+l-4} based on the *ALAP* scheduling time of *ACT* commands for T_{i-1}, which are given by Eq. (10). In particular, the possible maximum scheduling time of the first *ACT* command of T_{i-1} is obtained by using Eq. (10), which is shown in Eq. (36).

$$\begin{aligned} \hat{t}(ACT_{j-1-(BI_{i-1}-1)}) &= \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ &\quad - (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\} \end{aligned} \tag{36}$$

By conservatively using the minimum time interval *tRRD* between two successive *ACT* commands, Eq. (37) is derived, which provides the possible maximum scheduling time of ACT_{j+l-4}. Moreover, by substituting Eq. (36) into Eq. (37), an explicit expression of $\hat{t}(ACT_{j+l-4})$ is obtained.

$$\begin{aligned} \hat{t}(ACT_{j+l-4}) &= \hat{t}(ACT_{j-1-(BI_{i-1}-1)}) - (4 - l - BI_{i-1}) \times tRRD \\ &= \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ &\quad - (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\} \\ &\quad - (4 - l - BI_{i-1}) \times tRRD \end{aligned} \tag{37}$$

In order to prove that $\hat{t}(ACT_{j+l-4}) + tFAW$ cannot dominate in the max{ } of Eq. (35), we only need to prove that $\hat{t}(ACT_{j+l-4}) + tFAW \leq \hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP$. Since $\hat{t}(PRE_{j-1-(b_{com}-l)})$ is given by Eq. (11) with assumption that T_{i-1} is write while $\hat{t}(ACT_{j+l-4})$ is provided by Eq. (37), Eq. (38) is derived.

$$\begin{aligned} &\hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP - [\hat{t}(ACT_{j+l-4}) + tFAW] \\ &= \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD + tRP \\ &\quad - [\hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD - (BI_{i-1} - 1) \\ &\quad \times max\{tRRD, BC_{i-1} \times tCCD\} - (4 - l - BI_{i-1}) \times tRRD + tFAW] \\ &= tRWTP - (b_{com} - l) \times BC_{i-1} \times tCCD + tRP + tRCD \\ &\quad + (BC_{i-1} - 1) \times tCCD + (BI_{i-1} - 1) \times max\{tRRD, BC_{i-1} \times tCCD\} \\ &\quad + (4 - l - BI_{i-1}) \times tRRD - tFAW \end{aligned} \tag{38}$$

The result of this equation is non-negative, as the positive terms in Eq. (38) cancel out all the negative ones for the following four reasons: *1)* $max\{tRRD$,

$BC_{i-1} \times tCCD\} \geq BC_{i-1} \times tCCD$. *2)* $b_{com} - l \leq BI_{i-1} - 1$ since $\forall l \in [0, b_{com}]$ and $b_{com} = \min\{BI_{i-1}, BI_i\} - 1$. *3)* the observation from JEDEC DDR3 timing constraints that $tFAW \leq tRWTP + tRP + tRCD$ for all DDR3 memories with write transaction. *4)* $l < 4 - BI_{i-1}$ from the above discussion. Therefore, $\hat{t}(PRE_{j-1-(b_{com}-l)}) + tRP \geq \hat{t}(ACT_{j+l-4}) + tFAW$, which indicates $t(ACT_{j+l-4})$ cannot dominate in the max{} of Eq. (35). These earlier *ACT* commands (ACT_{j+l-4}) hence cannot dominate in the scheduling of the *ACT* commands for T_i because of *tFAW* in the worst case. Thus, Eq. (35) guarantees that the scheduling of *ACT* commands for T_i only depends on the maximum possible scheduling time of the previous *PRE* for T_{i-1} in the worst case or ACT_{j+l-1} that belongs to T_{i-1} for l=0. We can conclude that the *ALAP* command scheduling of the previous write transaction T_{i-1} is sufficient to give worst-case initial bank state to T_i. □

Proof of Lemma 4

Proof According to Lemma 1, the finishing time of a transaction T_i is determined by the finishing time of the previous transaction T_{i-1} and the scheduling time of all its *ACT* commands. Therefore, the worst-case finishing time of T_i is obtained by using the worst-case scheduling time (maximum) of its *ACT* commands, and the maximum finishing time of T_{i-1} that is $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$ based on Eq. (8), where we fix the worst-case starting time $\hat{t}_s(T_i)$ of T_i.

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

For $\forall l \in [0, b_{com}]$, the ACT_{j+l} command is scheduled to bank $b_j + l$ that is a common bank between T_i and T_{i-1}. Lemma 3 guarantees that the *ALAP* scheduling of commands for the write transaction T_{i-1} is sufficient to provide the worst-case initial bank state for T_i. As a result, the worst-case scheduling time $\hat{t}(ACT_{j+l})$ can be obtained based on this worst-case initial states. Eq. (35) is hence used to compute $\hat{t}(ACT_{j+l})$, which indicates that the scheduling time of ACT_{j+l} is either determined by its previous ACT_{j+l-1} or the latest precharge, $PRE_{j-1-(b_{com}-l)}$, to the same bank. By iteratively using Eq. (35) to obtain the scheduling time of each of the ACT commands to the common banks, we can derive a new expression of the scheduling time of ACT_{j+l} that is given by Eq. (39). Note that $\forall l' \in [0, l]$ indexes a bank ($b_j + l'$) that is not accessed later than bank $b_j + l$, since $b_j + l' \leq b_j + l$.

$$t(ACT_{j+l}) = \underset{0 \leq \forall l' \leq l}{Max} \{t(ACT_{j-1}) + (l+1) \times tRRD + \sum_{h=0}^{l} C(j+h),$$
$$t(PRE_{j-1-(b_{com}-l')}) + tRP + (l-l') \times tRRD + \sum_{h=l'}^{l} C(j+h)\} \quad (39)$$

$t(ACT_{j-1})$ in Eq. (39) is the scheduling time of the last ACT command for T_{i-1}. According to $ALAP$ scheduling, the worst-case scheduling time $\hat{t}(ACT_{j-1})$ can be derived based on Eq. (10), and it is given by Eq. (40).

$$\hat{t}(ACT_{j-1}) = \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \tag{40}$$

Moreover, the worst-case scheduling time of $PRE_{j-1-(b_{com}-l')}$ to the common bank $b_j + l'$ based on $ALAP$ scheduling is given by Eq. (11). Therefore, by substituting $t(ACT_{j-1})$ and $t(PRE_{j-1-(b_{com}-l')})$ in Eq. (39) with their worst-case scheduling time given by Eq. (40) and Eq. (11), we can obtain the worst-case scheduling time of ACT_{j+l}, as shown in Eq. (41).

$$\begin{aligned}\hat{t}(ACT_{j+l}) = \underset{0\le l'\le l\le b_{com}}{Max} \Big\{ & \hat{t}_s(T_i) - 1 - tRCD - (BC_{i-1} - 1) \times tCCD \\ & + (l+1) \times tRRD + \sum_{h=0}^{l} C(j+h), \\ & \hat{t}_s(T_i) - 1 + tRWTP - (b_{com} - l') \times BC_{i-1} \times tCCD \\ & + tRP + (l - l') \times tRRD + \sum_{h=l'}^{l} C(j+h) \Big\}\end{aligned} \tag{41}$$

For $\forall l \in (b_{com}, BI_i - 1]$, the ACT_{j+l} command is scheduled to the non-common bank $b_j + l$. Since Lemma 2 ensures that the scheduling time of an ACT command to a non-common bank is only determined by that of the ACT command to the last common bank, Eq. (34) is used to compute $t(ACT_{j+l})$, and it is only dependent on $t(ACT_{j+b_{com}})$. Eq. (41) is used to compute the worst-case scheduling time $t(ACT_{j+b_{com}})$, which is further substituted into Eq. (34). Hence, $\hat{t}(ACT_{j+l})$ is also derived when $\forall l \in (b_{com}, BI_i - 1]$.

Finally, we can use $\hat{t}_f(T_{i-1}) = \hat{t}_s(T_i) - 1$ and $\hat{t}(ACT_{j+l})$ to derive the worst-case finishing time $\hat{t}_f(T_i)$ of T_i based on Lemma 1 (described by Eq. (24)). It is described by Eq. (42), where $\forall l'[0, b_{com}]$ and $\forall l \in [l', BI_i - 1]$. Intuitively, Eq. (42) illustrates that the worst-case finishing time of a transaction is dependent on the precharging time of the common banks with the previous write transaction.

$$\begin{aligned}\hat{t}_f(T_i) = \underset{0\le l'\le b_{com}, l'\le l\le BI_i-1}{Max} \Big\{ & \hat{t}_s(T_i) - 1 - (BC_{i-1} - 1) \times tCCD + (l+1) \times tRRD \\ & + [(BI_i - l) \times BC_i - 1] \times tCCD + \sum_{h=0}^{l} C(j+h), \hat{t}_s(T_i) - 1 \\ & + tRWTP - (b_{com} - l') \times BC_{i-1} \times tCCD + tRP + tRCD + (l - l') \times tRRD \\ & + [(BI_i - l) \times BC_i - 1] \times tCCD + \sum_{h=l'}^{l} C(j+h), \hat{t}_s(T_i) - 1 \\ & + tSwitch + (BI_i \times BC_i - 1) \times tCCD \Big\}\end{aligned} \tag{42}$$

□

Proof of Theorem 1

Proof Since Lemma 4 provides the worst-case finishing time of a transaction T_i, we can hence compute the worst-case execution time (WCET) according to Definition 5. Then we only need to simplify the expressions in the equation and obtain the WCET.

Lemma 4 indicates that the worst-case finishing time $\hat{t}_f(T_i)$ depends on its worst-case starting time $\hat{t}_s(T_i)$, the BI and BC used by T_{i-1} and T_i, and the JEDEC DDR3 timing constraints. According to Definition 5, the WCET is the time between $\hat{t}_s(T_i)$ and $\hat{t}_f(T_i)$, and is given by Eq. (43).

$$\hat{t}_{ET}(T_i) = \hat{t}_f(T_i) - \hat{t}_s(T_i) + 1 \tag{43}$$

Based on Eq. (42) that gives the worst-case finishing time, we further obtain Eq. (44) according to Eq. (43) by moving $\hat{t}_s(T_i) - 1$ from the right side to the left of Eq. (42).

$$\begin{aligned}\hat{t}_{ET}(T_i) = \underset{0\le l'\le b_{com}, l'\le l\le BI_i-1}{Max} \Big\{ & -(BC_{i-1}-1)\times tCCD \\ & +(l+1)\times tRRD + [(BI_i - l)\times BC_i - 1]\times tCCD + \sum_{h=0}^{l} C(j+h), \\ & tRWTP - (b_{com}-l')\times BC_{i-1}\times tCCD + tRP + tRCD \\ & +(l-l')\times tRRD + [(BI_i - l)\times BC_i - 1]\times tCCD + \sum_{h=l'}^{l} C(j+h), \\ & tSwitch + (BI_i \times BC_i - 1)\times tCCD \Big\}\end{aligned} \tag{44}$$

Since we conservatively assume there is always scheduling collisions for ACT commands, i.e., $C(j+h) = 1$, Eq. (44) can be simplified based on $\sum_{h=0}^{l} C(j+h) = l+1$ and $\sum_{h=l'}^{l} C(j+h) = l - l' + 1$, as shown in Eq. (45).

$$\begin{aligned}\hat{t}_{ET}(T_i) = \underset{0\le l'\le b_{com}, l'\le l\le BI_i-1}{Max} \Big\{ & (BI_i \times BC_i - BC_{i-1})\times tCCD \\ & + l\times(tRRD + 1 - BC_i \times tCCD) + tRRD + 1, \\ & tRWTP + tRP + tRCD + [BI_i \times BC_i - 1 - b_{com}\times BC_{i-1}]\times tCCD + 1 \\ & + l'\times(BC_{i-1}\times tCCD - tRRD - 1) + l\times(tRRD + 1 - BC_i \times tCCD), \\ & tSwitch + (BI_i \times BC_i - 1)\times tCCD \Big\}\end{aligned} \tag{45}$$

We can observe from Eq. (45) that the expressions in the max{} function either linearly increase or decrease with l and l'. Therefore, Eq. (45) can be further simplified to obtain $\hat{t}_{ET}(T_i)$ by using both the maximum and minimum values of l and l' in the max{} of Eq. (45). Since $\forall l'[0, b_{com}]$ and $\forall l \in [l', BI_i - 1]$, we substitute (l', l) with $(0, 0)$, $(0, BI_i - 1)$, (b_{com}, b_{com}), and $(b_{com}, BI_i - 1)$ in all the terms of the max{} in Eq. (45). $\hat{t}_{ET}(T_i)$ is further given by Eq. (46). Note that some of the terms are removed, since they cannot dominate in the max{} when deriving Eq. (46).

$$\begin{aligned}
\hat{t}_{ET}(T_i) = \max \Big\{ & (BI_i \times BC_i - BC_{i-1}) \times tCCD \\
& + (BI_i - 1) \times (tRRD + 1 - BC_i \times tCCD) + tRRD + 1, \\
& tRWTP + tRP + tRCD + [BI_i \times BC_i - 1 - b_{com} \times BC_{i-1}] \times tCCD + 1, \\
& tRWTP + tRP + tRCD + [(BI_i - b_{com}) \times BC_i - 1] \times tCCD + 1, \\
& tRWTP + tRP + tRCD + [BC_i - 1 - b_{com} \times BC_{i-1}] \times tCCD \\
& + (BI_i - 1) \times (tRRD + 1) + 1, \\
& tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD \\
& + [BI_i - 1 - b_{com}] \times (tRRD + 1) + 1, \\
& tSwitch + (BI_i \times BC_i - 1) \times tCCD \Big\}
\end{aligned} \tag{46}$$

Note that $b_{com} = b_{j-1} - b_j$ and is determined by the size of T_{i-1} and T_i, whichever is smaller, i.e., $b_{com} = \min\{BI_{i-1}, BI_i\} - 1$. Moreover, some of the expressions in the max{} of Eq. (46) are further simplified according to the observation from JEDEC DDR3 timing constraints that $tSwitch > tRRD + 1$ when T_{i-1} is a write. The simplified $\hat{t}_{ET}(T_i)$ is finally shown in Eq. (47).

$$\begin{aligned}
\hat{t}_{ET}(T_i) = \max \Big\{ & (BC_i - BC_{i-1}) \times tCCD + BI_i \times (tRRD + 1), \\
& tRWTP + tRP + tRCD + 1 \\
& + [BI_i \times BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}] \times tCCD, \\
& tRWTP + tRP + tRCD + 1 \\
& + [(BI_i - (\min\{BI_{i-1}, BI_i\} - 1)) \times BC_i - 1] \times tCCD, \\
& tRWTP + tRP + tRCD + (BI_i - 1) \times (tRRD + 1) + 1 \\
& + [BC_i - 1 - (\min\{BI_{i-1}, BI_i\} - 1) \times BC_{i-1}] \times tCCD, \\
& tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD \\
& + [BI_i - \min\{BI_{i-1}, BI_i\}] \times (tRRD + 1) + 1, \\
& tSwitch + (BI_i \times BC_i - 1) \times tCCD \Big\}
\end{aligned} \tag{47}$$

□

Proof of Theorem 2

Proof To prove the WCET of a transaction provided by Theorem 1 monotonically increases with its size, it is only necessary to prove that the WCET monotonically increases with its *BI* and *BC*. The WCET of T_i is given by Theorem 1 (i.e., Eq. (47)). We can see that the WCET, $\hat{t}_{ET}(T_i)$, is determined by one of the 6 expressions in the max{} function, which are all also functions of BI_i and BC_i. These 6 expressions are denoted by *expr1* to *expr6*, respectively, corresponding to the expressions from top to bottom in Eq. (47). We proceed by proving the monotonicity for each of them.

Expression 1:

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

Expression 2:

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

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

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

We can observe that $expr5 > tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD + 1$ in this case for any BI_i and BC_i. As a result, *expr2* can dominate the max{} function of Eq. (47) only if the given BI_i and BC_i cannot make it smaller than $tRWTP + tRP + tRCD + (BC_i - 1) \times tCCD + 1$. Therefore, $BC_i \geq BC_{i-1}$ is a necessary condition for *expr2*. On this condition, we can derive $expr2(BI_i, BC_i) \geq expr2(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

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

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

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

With these two cases, when *expr2* dominates the max{} function of Eq. (47), there is $expr2(BI_i, BC_i) \geq expr2(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

Expression 3:

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + [(BI_i - (\min\{BI_{i-1}, BI_i\} - 1)) \times BC_i - 1] \times tCCD$. For this expression, there are again two cases, where the theorem follows straight-forwardly for both of them.

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

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + (BC_i - 1) \times tCCD$. As a result, $BI'_i \leq BI_i \wedge BC'_i \leq BC_i \implies expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$.

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

$expr3(BI_i, BC_i) = tRWTP + tRP + tRCD + 1 + [(BI_i - BI_{i-1} + 1) \times BC_i - 1] \times tCCD$. So, $BI'_i \leq BI_i \wedge BC'_i \leq BC_i \implies expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$.

According to these two cases, there is $expr3(BI_i, BC_i) \geq expr3(BI'_i, BC'_i)$, where $BI'_i \leq BI_i$ and $BC'_i \leq BC_i$.

Expression 4, 5, and 6:

With a similar discussion as for **Expression 2**, we can conclude that *expr4* monotonically increases with BI_i and BC_i. This conclusion also holds for *expr5* if it is analyzed in the same way as **Expression 3**, while *expr6* can be discussed similarly to **Expression 1**. The detailed derivation is not shown here for brevity. □

References

Akesson, B., Goossens K, Ringhofer M (2007) Predator: A predictable SDRAM memory controller. In: Int'l conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 251–256

Akesson B, Goossens K (2011) Architectures and modeling of predictable memory controllers for improved system integration. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1–6

Akesson B, Minaeva A, Sucha P, Nelson A, Hanzalek Z (2015) An efficient configuration methodology for time-division multiplexed single resources. In: Real-time and embedded technology and applications symposium (RTAS), pp 161–171

Akesson B, Steffens L, Strooisma E, Goossens K (2008) Real-time scheduling using credit-controlled static-priority arbitration. In: Int'l conference on embedded and real-time computing systems and applications (RTCSA), pp 3–14

Austin T, Larson E, Ernst D (2002) Simple scalar: an infrastructure for computer system modeling. Computer 35(2):59–67

Axer P, Ernst R, Falk H, Girault A, Grund D, Guan N, Jonsson B, Marwedel P, Reineke J, Rochange C, Sebastian M, Hanxleden RV, Wilhelm R, Yi W (2014) Building timing predictable embedded systems. ACM Trans Embed Comput Syst 13(4):82:1–82:37

Bayliss S, Constantinides GA (2009) Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search. In: International conference on field-programmable technology (FPT), pp 304–307

Benini L, Flamand E, Fuin D, Melpignano D (2012) 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 (DATE), pp 983–987

Cadence Design Systems Inc.: Multi-Protocol LPDDR4/3/DDR4/3 Controller and PHY Subsystem IP (2014)

Choi H, Lee J, Sung W (2011) Memory access pattern-aware DRAM performance model for multi-core systems. In: Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS), pp 66–75

Dasari D, Andersson B, Nelis V, Petters SM, Easwaran A, Lee J (2011) Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In: Proceedings of the 2011 IEEE 10th international conference on trust, security and privacy in computing and communications (TRUSTCOM), pp 1068–1075

Dev Gomony M, Akesson B, Goossens K (2014) Coupling TDM NoC and DRAM controller for cost and performance optimization of real-time systems. In: Design, automation and test in europe conference and exhibition (DATE), pp 1–6

Ecco L, Tobuschat S, Saidi S, Ernst R (2014) A mixed critical memory controller using bank privatization and fixed priority scheduling. In: Proceedings of the 20th IEEE international conference on real-time computing systems and applications (RTCSA), pp 1–10

Gomony MD, Akesson B, Goossens K (2013) Architecture and optimal configuration of a real-time multi-channel memory controller. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1307–1312

Gomony MD, Garside J, Akesson B, Audsley N, Goossens K (2015) A generic, scalable and globally arbitrated memory tree for shared DRAM access in real-time systems. In: Design, automation test in Europe conference exhibition (DATE), pp 193–198

Gomony MD, Akesson B, Goossens K (2014) A real-time multi-channel memory controller and optimal mapping of memory clients to memory channels. ACM Trans Embed Comput Syst 14(2):25

Goossens S, Akesson B, Goossens K (2013) Conservative open-page policy for mixed time-criticality memory controllers. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 525–530

Goossens S, Kouters T, Akesson B, Goossens K (2012) Memory-map selection for firm real-time memory controllers. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 828–831

Goossens S, Kuijsten J, Akesson B, Goossens K (2013) A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 1–10

Hameed F, Bauer L, Henkel J (2013) Simultaneously optimizing DRAM cache hit latency and miss rate via novel set mapping policies. In: Proceedings of the 2013 international conference on compilers, architectures and synthesis for embedded systems (CASES'13), pp 11:1–11:10

Hansson A, Goossens K (2011) A quantitative evaluation of a network on chip design flow for multi-core consumer multimedia applications. Des Autom Embed Syst 15(2):159–190

Hur I, Lin C (2007) Memory scheduling for modern microprocessors. ACM Trans Comput Syst 25(4):10

Ipek E, Mutlu O, Martinez JF, Caruana R (2008) Self-optimizing memory controllers: a reinforcement learning approach. In: Proceedings of the 35th annual international symposium on computer architecture (ISCA), pp 39–50

Jacob B, Ng S, Wang D (2007) Memory systems: cache, DRAM. Disk. Morgan Kaufmann Publishers Inc., San Francisco

JEDEC Solid State Technology Association: DDR3 SDRAM Specification (2010)
Kim Y, Papamichael M, Mutlu O, Harchol-Balter M (2011) Thread cluster memory scheduling. IEEE Micro 31(1):78–89
Kim H, de Niz D, Andersson B, Klein M, Mutlu O, Rajkumar R (2014) Bounding memory interference delay in COTS-based multi-core systems. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 145–154
Kollig P, Osborne C, Henriksson T (2009) Heterogeneous multi-core platform for consumer multimedia applications. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1254–1259
Krishnapillai Y, Wu ZP, Pellizzoni, R (2014) A rank-switching, open-row DRAM controller for time-predictable systems. In: Proceedings Euromicro conference on real-time systems (ECRTS), pp 27–38
Lee C, Potkonjak M, Mangione-Smith WH (1997) MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture, MICRO, pp 330–335
Li Y, Akesson B, Goossens K (2014a) Dynamic command scheduling for real-time memory controllers. In: Proceedings euromicro conference on real-time systems (ECRTS), pp 3–14
Li Y, Akesson B, Goossens K (2014b) RTMemController: open-source WCET and ACET analysis tool for real-time memory controllers. http://www.es.ele.tue.nl/rtmemcontroller/. Accessed 23 July 2015
Lin WF, Reinhardt S, Burger D (2001) Reducing DRAM latencies with an integrated memory hierarchy design. In: The seventh international symposium on high-performance computer architecture (HPCA), pp 301–312
Nowotsch J, Paulitsch M, Buhler D, Theiling H, Wegener S, Schmidt M (2014) Multicore interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In: Proceedings Euromicro conference on real-time systems (ECRTS), pp 109–118
Paolieri M, Qui nones E, Cazorla FJ (2013) Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. ACM Trans Embed Comput Syst 12(1):64:1–64:26
Pop T, Pop P, Eles P, Peng Z, Andrei A (2008) Timing analysis of the FlexRay communication protocol. R Time Syst 39(1–3):205–235
PrimeCell AHB SDR and NAND memory controller (PL242) (2006). http://www.arm.com. Accessed 23 July 2015
Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) 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 (CODES+ISSS), pp 99–108
Schliecker S, Negrean M, Ernst R (2010) Bounding the shared resource load for the performance analysis of multiprocessor systems. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 759–764
Shah H, Raabe A, Knoll A (2012) Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 665–670
Stefan R, Molnos A, Goossens K (2014) dAElite: A tdm noc supporting qos, multicast, and fast connection set-up. IEEE Trans Comput 63(3):583–594
Stevens A (2010) QoS for high-performance and power-efficient HD multimedia. ARM White paper
van Berkel CHK (2009) Multi-core for mobile phones. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1260–1265
Wang D, Ganesh B, Tuaycharoen N, Baynes K, Jaleel A, Jacob B (2005) DRAMsim: a memory system simulator. SIGARCH Comput Archit News 33:100–107
Wu ZP, Krish Y, Pellizzoni R (2013) Worst case analysis of DRAM latency in multi-requestor systems. In: Proceedings of the 34th IEEE real-time systems Symposium (RTSS), pp 372–383

Yonghui Li received his bachelor and master degree in Telecommunication Engineering both from Xidian University, China in 2009 and 2012, respectively. Since May 2012, he has been working toward a PhD at Eindhoven University of Technology, the Netherlands. His research interests include Networks on Chip (NoC), memory controllers, and WCET analysis of real-time systems.

Benny Akesson received his M.Sc. degree at Lund Institute of Technology, Sweden in 2005 and a Ph.D. from Eindhoven University of Technology, the Netherlands in 2010. Since then, he has been employed as a Postdoctoral Researcher at Eindhoven University of Technology, CISTER-ISEP Research Unit, and Czech Technical University in Prague. His research interests include design and analysis of multi-core real-time systems with shared resources.

Kees Goossens received his PhD in Computer Science from the University of Edinburgh in 1993 on hardware verification using embeddings of formal semantics of hardware description languages in proof systems. He worked for Philips/NXP Research from 1995 to 2010 on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time full professor at Delft University from 2007 to 2010. Since then he has been full-time full professor at Eindhoven University of Technology, where his research focusses on composable (virtualised), predictable (real-time), low-power embedded systems, supporting multiple models of computation. He published 3 books, 100+ papers, and 24 patents.