CISTER
Research Center in
Real-Time & Embedded
Computing Systems

# Conference Paper

## Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers

**Yonghui Li**
**Benny Åkesson***
**Kees Goossens**

*CISTER Research Center

CISTER-TR-160202

2016/04/11

# Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers

Yonghui Li, Benny Åkesson*, Kees Goossens

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: kbake@isep.ipp.pt

http://www.cister.isep.ipp.pt

## Abstract

In modern multi-core systems with multiple real-time (RT) applications, memory traffic accessing the shared SDRAM is increasingly diverse, e.g., transactions have variable sizes. RT memory controllers with dynamic command scheduling can efficiently address the diversity by issuing appropriate commands subject to the SDRAM timing constraints. However, the scheduling dependencies between commands make it challenging to derive tight bounds for the worst-case response time (WCRT) and the worst-case bandwidth (WCBW) of a memory controller. Existing modeling and analysis techniques either do not provide tight WCRT and WCBW bounds for diverse memory traffic with variable transaction sizes or are difficult to adapt to different RT memory controllers.

This paper models a memory controller using Timed Automata (TA), where model checking is applied for analysis. Our TA model is modular and accurately captures the behavior of a RT memory controller with dynamic command scheduling. We obtain WCRT and WCBW bounds, which are validated by simulating the worst-case transaction traces obtained by model checking with a cycle-accurate model of the memory controller. Our method outperforms three state-of-the-art analysis techniques. We reduce WCRT bound by up to 20%, while the average improvement is 7.7%, and increase the WCBW bound by up to 25% with an average improvement of 13.6%. In addition, our modeling is generic enough to extend to memory controllers with different mechanisms.

# Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers

Yonghui Li[1], Benny Akesson[2], Kai Lampka[3], and Kees Goossens[1]
[1]Eindhoven University of Technology, [2]CISTER/INESC TEC, ISEP, [3]Uppsala University

***Abstract*—In modern multi-core systems with multiple real-time (RT) applications, memory traffic accessing the shared SDRAM is increasingly diverse, e.g., transactions have variable sizes. RT memory controllers with dynamic command scheduling can efficiently address the diversity by issuing appropriate commands subject to the SDRAM timing constraints. However, the scheduling dependencies between commands make it challenging to derive tight bounds for the worst-case response time (WCRT) and the worst-case bandwidth (WCBW) of a memory controller. Existing modeling and analysis techniques either do not provide tight WCRT and WCBW bounds for diverse memory traffic with variable transaction sizes or are difficult to adapt to different RT memory controllers.**

**This paper models a memory controller using Timed Automata (TA), where model checking is applied for analysis. Our TA model is modular and accurately captures the behavior of a RT memory controller with dynamic command scheduling. We obtain WCRT and WCBW bounds, which are validated by simulating the worst-case transaction traces obtained by model checking with a cycle-accurate model of the memory controller. Our method outperforms three state-of-the-art analysis techniques. We reduce WCRT bound by up to 20%, while the average improvement is 7.7%, and increase the WCBW bound by up to 25% with an average improvement of 13.6%. In addition, our modeling is generic enough to extend to memory controllers with different mechanisms.**

## I. Introduction

Heterogeneous multi-core platforms executing real-time (RT) applications use different hardware resources, such as processor cores, hardware accelerators, DMA controllers, and the off-chip SDRAM memory [4], [17]. To meet the RT requirements of applications, each resource has to guarantee its performance [2], [26]. It is challenging to provide guaranteed performance for the SDRAM, because 1) memory clients generate diverse traffic with arbitrarily mixes of read/write transactions with variable sizes, 2) SDRAM locations are randomly accessed by different transactions, and 3) the execution time of a transaction depends on the SDRAM state, which relies on earlier transactions.

To ease the complexities and to provide guaranteed performance, semi-static RT memory controllers [1], [8], [11], [28] use pre-computed static command schedules for a fixed transaction size. This results in low data efficiency for traffic with variable transaction sizes. Dynamic RT memory controllers [12], [13], [18], [23] efficiently address the diversity by scheduling appropriate commands as soon as timing constraints are satisfied. However, the scheduling together with timing dependencies between commands is hard to analyze. Therefore, existing manual timing analyses only provide pessimistic bounds on worst-case response time (WCRT) and worst-case bandwidth (WCBW), because they have to assume worst-case initial SDRAM state for a new transaction and also assume conflicts on the command bus. In addition, any change to the command scheduling or timing constraints (e.g. for a different memory device) may require these manual timing analyses to be adapted.

This paper proposes an accurate Timed Automata (TA) model of a memory controller with dynamic command scheduling [23], and the worst-case bounds are derived by model checking. Our TA model does not employ any simplifying abstractions, resulting in tighter worst-case bounds than the state-of-the-art analyses. The four main contributions are:

1) *A modular TA model consisting of the DDR3 SDRAM device and the memory controller*. The former captures all SDRAM timing constraints, while the latter models the timing behavior of the memory controller architecture. The TA model can be easily extended or reused for different memory controllers or different DDR3 SDRAM devices. Our TA model is publicly available [21].
2) We *validate our TA model with the open-source RT-MemController tool* [20], which has been shown to be equivalent to a cycle-accurate SystemC simulator of the dynamic command scheduling algorithm under consideration [22]. We execute random transaction traces with around 1200 commands using both the TA model and RTMemController, resulting in *identical* scheduling times of each command. This gives evidence that our TA model accurately captures the command timings of the memory controller with dynamic command scheduling.
3) *Worst-case response time and worst-case bandwidth* are derived by verifying properties of the TA model with the Uppaal model checker [3]. Uppaal provides diagnostic traces that lead to the worst-case bounds. Executing these traces with the cycle-accurate simulator provides *identical* WCRT and WCBW results as Uppaal, which speaks for the accuracy of our model.
4) Finally, the proposed TA analysis reduce the WCRT bound by up to 20% and improve the WCBW bound by up to 25% compared to the state-of-the-art [23], [24]. The average improvements of the bounds on WCRT and WCBW are 7.7% and 13.6%, respectively.

In the remainder of this paper, the background of SDRAM, RT memory controllers, and TA is given in Section II. Section III presents our TA model of the memory controller with dynamic command scheduling. Section IV shows how the WCRT and WCBW can be derived by verifying properties of the TA model. We then review related work in Section V before the experiments in Section VI. Finally, we conclude in Section VII.

## II. BACKGROUND

This section introduces SDRAM, the architecture and command scheduling of real-time memory controllers, and Timed Automata (TA). In the remainder of this paper, we use the Uppaal toolbox [3] to implement and analyze our TA.

### A. SDRAM and Real-Time Memory Controllers

*1) SDRAM and Timing Constraints:* A DDR3 SDRAM chip is typically composed of 8 banks, each of which contains a memory array arranged in rows and columns, as shown in Fig. 1. The SDRAM interface consists of command, address, and data buses. To read/write data from/into the SDRAM, memory commands must be scheduled on the command bus every clock cycle while data is transferred on a separate data bus. An activate (*ACT*) command opens a row and copies the contents into the row buffer. Next, a read (*RD*) or write (*WR*) command is issued to read/write a burst of data from/into the row buffer. The burst length (*BL*) is 8 words for DDR3 SDRAMs [14]. Before activating a new row, the contents of the row buffer must be closed and copied back into the row in the memory array. This is achieved by explicitly issuing a precharge (*PRE*) command or attaching an auto-precharge flag to a *RD* or *WR* command to trigger the internal precharging of the SDRAM. We employ auto-precharge in this paper, since it reduces the contention on the command bus. Finally, the SDRAM must be periodically refreshed every *tREFI* cycles to retain the data.

The scheduling of the memory commands has to satisfy the JEDEC-specified timing constraints [14], which are the minimum time between commands. The relevant timing constraints are summarized in Table I, where a 16-bit DDR3-1600G SDRAM with the capacity of 2Gb is taken as an example. Intra-bank timing constraints restrict commands executed by the same bank, such as *tRCD* that specifies a minimum time between *ACT* and *RD* or *WR* commands to the same bank. Other examples are *tRAS* and *tRP*. Inter-bank timing constraints restrict commands executed by different banks. For example, *tRRD* is the minimum time between two successive *ACT* commands to two different banks. Other examples are *tFAW* and the read/write switching constraints *tWTR* and *tRTW*.

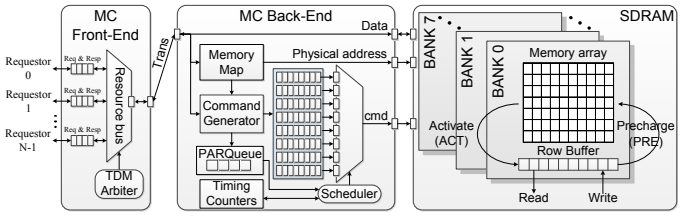



Fig. 1. Memory controller and DDR3 SDRAM architectures.

*2) Memory Controller Architecture:* We use the memory controller of [22], consisting of a front-end and back-end, as shown in Fig. 1. The front-end receives memory transactions from different memory requestors, e.g., processing cores, and places them in per-requestor request buffers. The response buffers, also for each requestor, receive the data that is read from the memory. A typical requestor generates transactions with fixed size, such as CPU cache misses [30], but transaction sizes vary between requestors. Transactions from different requestors are forwarded to the back-end using a time-division multiplexing (TDM) arbiter. When the back-end is ready, the selected transaction is sent.

TABLE I. TIMING CONSTRAINTS FOR DDR3-1600G SDRAM [14].

| *TC* | *Description* | *Cycles* |
|---|---|---|
| tRCD | Minimum time between *ACT* and *RD* or *WR* command 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 from a *WR* command until first data is available on the bus | 8 |
| tRL | Read latency. Time from 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 | 8 |
| tWTR | Minimum time between *WR* and *RD* commands | 6 |
| tWR | Write recovery time. Minimum time from 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 |

In the back-end, a transaction's logical address is translated into the physical address (bank, row, and column) according to the memory-map configuration. This configuration determines how a transaction interleaves over the banks and hence how much bank parallelism is exploited. It specifies the *number of interleaved banks* (*BI*) over which the transaction is interleaved and the *burst count* (*BC*) per bank. *BI* determines the number of consecutive banks to be accessed. *BC* specifies the number of data bursts per bank, which corresponds to the number of *RD* or *WR* commands. *BI* and *BC* determine the transaction size, i.e., $size = BI \times BC \times BL$, where *BL* is the burst length (e.g., 8 words for DDR3 SDRAMs). *BI* and *BC* are limited to powers of two for efficient address decoding. Thus, the *BI* consecutive banks of a transaction must start on a bank aligned with *BI* [8], and a transaction has $8/BI$ possible different starting banks. This impacts the scalability of model checking, as we will see in Section VI.

With the chosen *BI* and *BC*, the Command Generator in Fig. 1 generates commands for the transaction. For each bank, an *ACT* command is generated followed by *BC RD* or *WR* commands, where the last one has an auto-precharge flag to implement a close-page policy. These commands are buffered in the FIFO queue of the bank. Moreover, these parameters (BI, BC, bank id) of the transaction is inserted into the PARQueue as an element, and they are useful for command scheduling.

*3) Dynamic Command Scheduling:* The commands in the command queues are dynamically scheduled to the SDRAM according to the algorithm in [23]. The basic idea is that the command scheduler in Fig. 1 only makes arbitration

among the head queue commands, whose timing constraints are satisfied. The arbitration rules are that 1) *RD* and *WR* commands are scheduled in a first-come first-serve (FCFS) manner of transactions to ensure coherent memory. 2) *RD* and *WR* have higher priority than *ACT* in case the timing constraints are satisfied for both. *ACT* commands to different banks are issued in a pipelined manner with the prioritized *RD* or *WR* commands to enhance the scheduling efficiency. When an *ACT* and a *RD*/*WR* command contend on the command bus, i.e., implying a collision, the *ACT* command is postponed to the next clock cycle, since it has lower priority specified by the command scheduling algorithm [23]. Note that all constraints are tracked by timing counters shown in Fig. 1. When a command is scheduled, the relevant counters are reset. The command scheduling algorithm is reproduced in full details in Appendix A for convenience.

The timing constraints and the scheduling algorithm create dependencies between commands. Fig. 2 illustrates the dependencies between commands to any two sequentially accessed banks $b_j$ and $b_{j+1}$. For $\forall j \geq 0$, the $j^{th}$ bank access has an $ACT_j$ followed by $BC_j$ *RD* or *WR* ($RW_j^k, k \in [0, BC_j - 1]$), where $BC_j$ is the burst count for bank $b_j$. The dependencies caused by inter- and intra-bank timing constraints are represented by the dashed and solid arrows, respectively. An *ACT* may be blocked by a higher-priority *RD* or *WR* command because of a collision on the command bus, which is depicted by the solid circle in Fig. 2. When the command is scheduled, all the relevant timing counters are updated. The back-end is ready to accept a new transaction from the front-end when all the *ACT* commands of the current transaction are scheduled. This enables pipelining between successive transactions [23]. Note that the timing constraints shown in Fig. 2 are included in Table I except the *tRWTP*, which is calculated based on other timing constraints and is included in [23].

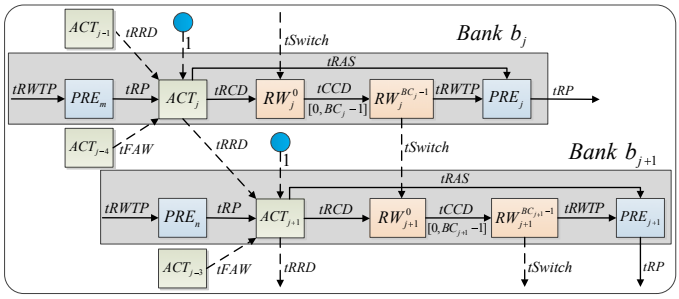


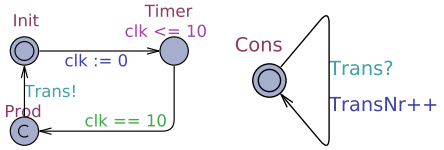

Fig. 2. The command dependencies for two successive bank accesses [23].

## B. Timed Automata (TA)

The Uppaal toolbox [3] implements TA as finite state machines extended with clocks and variables. All the clocks progress synchronously. Uppaal models a system as a network of several TA in parallel. A state of a TA is a set of active locations and a value for each clock and variable. Fig. 3(a) shows a TA that periodically produces memory transactions. It consists of an initial location (Init), and two other locations labeled Timer and Prod that are used to guard a minimal time between any two successive memory transactions of 10 cycles.

An edge connecting two locations can be traversed only if its guard evaluates to true. For example, the edge from location Timer to location Prod in Fig. 3(a) is traversed when the guard $clk == 10$ becomes true. Similarly, locations have invariants that have to be true while the location is marked active. Otherwise, the TA cannot reside in this location. As shown in Fig. 3(a), the invariant $clk \leq 10$ of location Timer guarantees that the clock variable clk does not exceed 10 when location Timer is marked active. All clocks in the TA are real-valued and increase their value at the same rate. Clocks and variables can only be inspected and reset upon the traversal of an edge. In our model clocks are only reset to 0 or 1.

In this paper, we heavily exploit the concepts of *urgent* and *committed* locations offered by Uppaal. Urgent locations are marked with `U` and committed locations are marked with `C`. For example, location Prod in Fig. 3(a) is a committed location. Urgent and committed locations need to be left without time progress, i.e, clocks do not progress when urgent or committed locations are marked active. Contrary to urgent locations, any of the active committed location has to be left immediately on the next transition. This gives their outgoing edges a higher priority and thereby reduces the non-determinism in the model. As a result, using committed locations greatly reduces the state space for model checking.




(a) Producer (b) Consumer

Fig. 3. A Timed Automata model of producing and consuming transactions.

In networks of TA, the component TA interact via shared variables and synchronization labels. The communication through synchronization labels is realized as a synchronized edge traversal of a sending edge (label with !) and receiving edge (label with ?). This atomic step includes the manipulation of associated variables and reset of associated clocks. Updates on sending edges are performed before the receiving edges. Since pairs of sending and receiving edges that synchronize are selected non-deterministically, all synchronization pairs emanating from active locations need to be generated when analyzing a TA. Besides binary synchronization, Uppaal also features 1:n synchronization for modeling broadcasts. Please refer to the Uppaal manual [3] for more details.

The originally infinite transition system generated from a network of TA can be reduced to a finite quotient system. Instead of tracking individual values of clocks, the domain of each clock is partitioned into finitely many intervals, denoted as clock regions or their conjunction denoted as clock zones. Therefore, timed reachability queries formulated in a temporal logic, e.g., Timed CTL can be verified with a TA in a finite number of steps as only finitely many combinations of clock regions need to be traversed. However, this number can be huge in practice.

# III. Modular Modeling of Dynamic Command Scheduling with TA

This section presents the TA model of the real-time memory controller with dynamic command scheduling [23].

A high-level overview of the model is shown, followed by discussing each component TA in detail. The TA model is available as open-source software on-line [21].

### A. Overview of the TA Model

A memory controller arbitrates between requestors. The selected transaction from a requestor is executed by dynamically scheduling its commands to consecutive banks of the SDRAM. To capture the behavior of the memory controller, we model the components shown in Fig. 1, including the source of memory traffic, the TDM arbiter in the front-end, and the back-end including the memory mapping, command scheduler, and timing-constraint counters.

Fig. 4 presents the components in our TA model of the RT memory controller together with their communication dependencies. Each of the components in Fig. 4 is implemented by its own template TA. Communication between them is realized by synchronization labels. Our TA model: 1) Accurately describes the functionality of the memory controller, *without any simplifying over-approximations*, cf. Section V. This is a key to derive tight WCRT and WCBW bounds. 2) *Scalably* models transactions with different sizes and starting banks, in the sense that the size of the model is independent of the number of sizes and starting banks. 3) Is *modular*, i.e., each memory-controller component is modeled by a corresponding TA. Since memory controllers have common components, e.g., the timing constraint counters and command bus, the corresponding TA can be reused when modeling other memory controllers. 4) Is *easily adapted* to different memory generations (e.g., DDR3 and LPDDR3) by replacing the timing constraint values for the specific memory device [8].

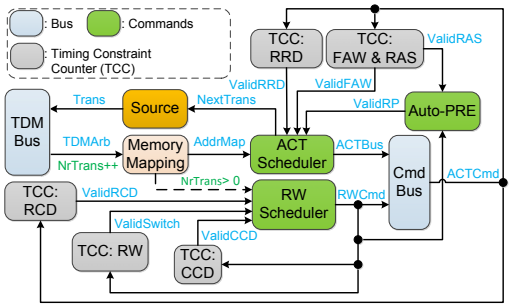



Fig. 4. Abstracted overview of TA model for the RT memory controller.

The Source in Fig. 4 generates read and write transactions for the requestors sharing a bus with a TDM arbiter (i.e., TDM Bus). TDM Bus also specifies the transaction size corresponding to the requestor and sends it to the back-end via the TDMArb synchronization label. As stated in Section II-A3, to capture the pipelining between successive transactions, the back-end accepts the next transaction when all *ACT* commands of the current transaction are scheduled [22]. This is accurately captured by the Source that generates a new transaction when it is notified by the ACT Scheduler in the back-end via the NextTrans synchronization label. Note that this Source component makes the memory controller busy, i.e., there is always a transaction ready when the back-end can accept a new transaction. This ensures that each requestor has pending transactions to be executed within its allocated slots [23]. This results in maximum interferences between requestors, leading to the worst-case scenario. Note that the work-conserving TDM arbitration [23] skips idle slots rather than reallocates them. As a result, a transaction experiences the worst-case response time when all other requestors have pending transactions.

The Memory Mapping in Fig. 4 specifies the *BI*, *BC* and the starting bank address (*BS*) for a transaction sent by the TDM Bus through the TDMArb synchronization label. These parameters are required by the ACT Scheduler and RW Scheduler, which accurately capture the dynamic command scheduling algorithm [22] for *ACT* and *RD*/*WR* commands, respectively. In particular, the ACT Scheduler issues an *ACT* command for the *BI* consecutive banks on the command bus (Cmd Bus), subject to the timing constraints of the memory. Timing counters (TCC) are used to track these constraints. Since the scheduling of an *ACT* command has to satisfy the *tRRD*, *tFAW*, and *tRP* constraints shown in Fig. 2, the ACT Scheduler is notified via the ValidRRD, ValidFAW, and ValidRP synchronization labels when the timing constraints are satisfied, allowing for a new *ACT* command to be scheduled. Then, the ACT Scheduler synchronizes with the Cmd Bus using the ACTBus label. In the same way, *RD* and *WR* commands are scheduled by RW Scheduler according to the relevant timing constraints, e.g., *tCCD*, *tRCD*, and the read and write switching constraint captured by RW Counter. It also synchronizes with the Cmd Bus using the RWCmd label when all these timing constraints are satisfied. Note that the RW Scheduler starts when NrTrans $> 0$, indicating there is at least one transaction in the back-end. The ACT Scheduler is triggered by Memory Mapping through synchronization via the AddrMap label. Since the RW Scheduler and the ACT Scheduler work in parallel, the TA model captures the pipelining between scheduling *ACT* commands for the next transaction and scheduling *RD* or *WR* commands for the current transaction.

The Cmd Bus accurately models collisions between *ACT* and *RD* or *WR* commands by prioritizing the latter. As a result, the *ACT* command is delayed by 1 cycle when a collision occurs. After scheduling an *ACT* command, the relevant timing constraint counters are reset through broadcast synchronization using the ACTCmd label. Similarly, the timing constraint counters related to *RD* and *WR* commands are reset via broadcast synchronization labeled as RWCmd. Finally, the Auto-PRE describes the behavior of auto-precharging, which is triggered by RWCmd. These timing constraints are explicitly shown in Table I except *tRTW* and *tWTP*, which can be indirectly derived from [22]. When the precharging of a bank is finished, the ACT Scheduler is notified by synchronizing with the ValidRP label. Note that all the synchronizations of our TA model are *urgent*, since commands are scheduled as soon as timing constraints are met.

Our TA model does not include the scheduling of refresh, which is required periodically with a relatively large time interval *tREFI*. The reason is that the WCRT bound of each transaction is too pessimistic if the refresh period is included. Alternatively, the refresh penalty can be taken into the analysis of the application rather than individual transactions [29]. Moreover, the elimination of refresh also simplifies the TA model and reduces the state space for model checking. However, it is not difficult to model the refresh mechanism. Our

model only needs to use an extra TA template to trigger the refresh every *tREFI* cycles.

### B. TA Modeling of Command Scheduling

We proceed by introducing the TA model for the dynamic command scheduling shown in Fig. 5.

*1) Automata Templates of the Requestors and Front-End:* Requestors are modeled by the Source TA, shown in Fig. 5(a). It non-deterministically generates an infinite sequence of read and write transactions. The type of each transaction is defined by the global variable TransType. The Source TA synchronizes with the TDM Bus TA using the Trans label. The TDM Bus TA, shown in Fig. 5(e), models a TDM bus instance with, as an example, five requestors, each of which has one TDM slot. Note that this TDM behaves the same as the round-robin (RR) scheme, since they have the same worst-case behavior when each requestor has only one slot. The guard on each edge specifies the slot and the requestor, and the corresponding transaction size TransSize is specified upon the edge traversal. The TDM Bus TA then synchronizes with the Memory Mapping TA using the TDMArb label.

*2) Automata Template of the Memory Mapping:* The Memory Mapping TA in Fig. 5(j) determines the *BI* and *BC* based on the TransSize, while the starting bank (*BS*) is given by non-deterministically selecting one outgoing edge. For each transaction size, *BI* and *BC* are configured to achieve the lowest execution time [23]. *BS* is aligned with *BI* to simplify the physical address decoding [8]. When the memory mapping is finished, the number (i.e., NrTrans) of transactions in the back-end is increased and command scheduling is triggered. PARQueue contains the information (i.e., TransType, BI, BC, BS) of the active transactions, cf. Fig. 1 and Appendix A.

*3) Automata Template of the ACT Scheduler:* The ACT Scheduler TA, shown in Fig. 5(f), starts scheduling *ACT* commands of a transaction after synchronizing with the Memory Mapping TA through the AddrMap label. An *ACT* command is scheduled to each of the *BI* consecutive banks, starting at bank *BS*. This is achieved by repeatedly scheduling each *ACT* command subject to the timing constraints *tRRD*, *tFAW*, and *tRP*, as given in Table I. These constraints are tracked by TA, as shown in Fig. 5(d) for the *tRRD* constraint. The ACT Scheduler TA waits for each timing constraint to be met. It advances through locations RRD, FAW, and RP when the relevant TA indicates that timing constraint is met through the ValidRRD, ValidFAW, and ValidRP labels, respectively. The ACT Scheduler TA and Cmd Bus TA synchronize using the ACTBus and ACTCmd label. The former notifies the Cmd Bus TA to schedule the *ACT* command, while the latter triggers the ACT Scheduler TA to schedule the next *ACT* command once the previous one has been scheduled by the Cmd Bus TA.

*4) Automata Template of the RW Scheduler:* The RW Scheduler TA, shown in Fig. 5(h), works similarly to the ACT Scheduler. It repeatedly schedules *BC RD* or *WR* commands in a sequence to each of the *BI* consecutive banks subject to the timing constraints. The relevant timing constraints are *tRCD* and *tCCD*. Note that the first *RD*/*WR* command of a transaction additionally has to satisfy the switching timing constraint when the previous transaction is write/read. The *tCCD*, *tRCD*, *tRTW* and *tWTR* timing constraints are captured by TA shown in Fig. 5(b), 5(c), and 5(l), respectively.

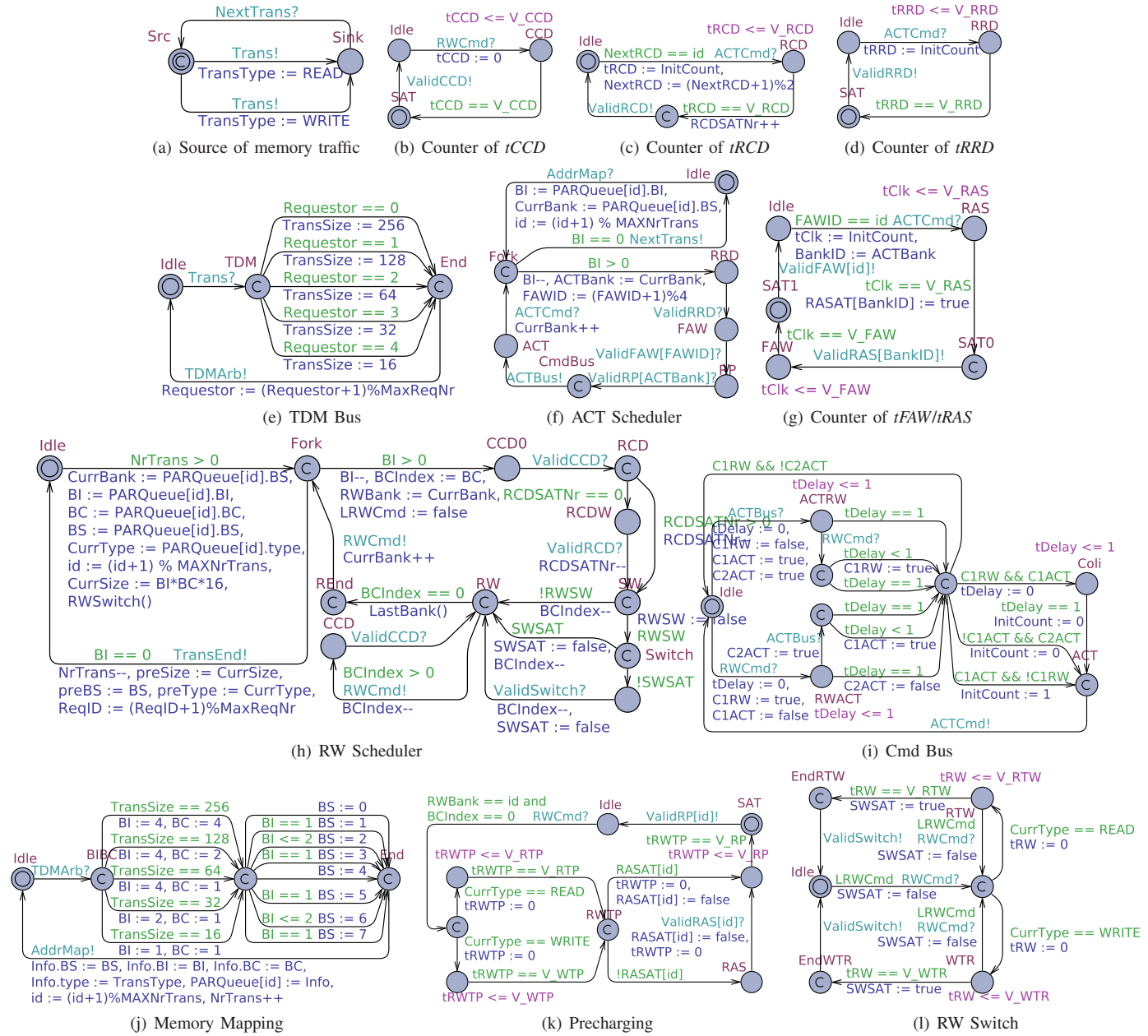
Recall that a *RD* or *WR* command has a higher priority than an *ACT* command that may have its timing constraints satisfied at the same time. Therefore, when a *RD*/*WR* command is scheduled, the RW Scheduler TA synchronizes with the Cmd Bus TA using the RWCmd label. As explained below, the Cmd Bus TA then ensures that the *ACT* command is scheduled one cycle later, thus solving the command collision. Finally, the broadcast synchronization using the RWCmd label tells the relevant timing constraint TA to reset their counters.

*5) Automata Templates of the Command Bus:* The command bus is modeled by the TA shown in Fig. 5(i), which detects and solves command bus collisions. The Cmd Bus TA synchronizes with the ACT Scheduler and RW Scheduler TA using the ACTBus and RWCmd labels, respectively. Although these two TA run in parallel, the synchronizations labeled RWCmd and ACTBus are received sequentially. The ACTRW and RWACT locations in Fig. 5(i) ensure that these actions can be received in either order. After synchronizing through either a RWCmd label or an ACTBus label, the Cmd Bus TA waits until the end of the cycle to see if the other synchronization arrives within this time. If so, a command collision has to be resolved by postponing the *ACT* to the next cycle.

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

*6) Automata Templates of Timing Constraint Counters:* Timing constraints are tracked by counters, each counting to the JEDEC-specified value [14]. For example, the timing constraint *tRAS* (see Table I) is modeled by the TA shown in Fig. 5(g). It uses a clock variable tClk, which is initialized to InitCount (0 or 1, see the previous paragraph) after an *ACT* command was scheduled, as indicated by the ACTCmd label. This counter counts to the constant V_RAS of *tRAS* provided by JEDEC DDR3 standard [14]. When the timing constraint is satisfied, i.e., tClk == V_RAS, it immediately synchronizes using the ValidRAS label. The timing constraints, such as *tCCD*, *tRCD*, *tRRD* are modeled in the same way, and their TA are shown in Figs. 5(b), Fig. 5(c), and Fig. 5(d), respectively. Note that different memory devices can be supported by using their timing constraint values in these counters [8].

As mentioned in Section II-A1, the SDRAM timing con-




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

(e) TDM Bus (f) ACT Scheduler (g) Counter of *tFAW*/*tRAS*

(h) RW Scheduler (i) Cmd Bus

(j) Memory Mapping (k) Precharging (l) RW Switch

Fig. 5. The TA templates of dynamic command scheduling within the Uppaal toolbox.

straints are classified into inter- and intra-bank. Intuitively, an inter-bank constraint can be tracked by a single counter, e.g., *tRRD*, while an intra-bank constraint, such as *tRAS*, should be tracked by a counter per bank. This encoding uses 8 counters per intra-bank constraint. Each counter increases the state space, resulting in longer time and higher memory usage when verifying properties of the TA model. To reduce the number of states, clocks, variables, and edges, we introduce two optimizations. Whenever possible, *(i) we model multiple constraints with a single TA instead of modeling them with separate TA. (ii) We reuse counters for different timing constraints.*

The above optimizations are used by our TA model. i) The TA shown in Fig. 5(g) tracks both the *tRAS* and *tFAW* constraints. The JEDEC [14] standard guarantees that *tFAW* $\geq$ *tRAS*, and thus the tClk in Fig. 5(g) counts first until *tRAS*, and then until *tFAW*. The inter-bank four activate window constraint *tFAW* can be tracked by using four counters for four *ACT* commands. The intra-bank *tRAS* constraint can be implemented with 8 counters, one for each bank of DDR3 SDRAM. However, using the aforementioned inequality, four counters suffice to track *tRAS* for all 8 banks. The fifth *ACT* has to wait until *tFAW* before it can be scheduled. By then, it is guaranteed that at least one counter has passed *tRAS* because *tFAW* $\geq$ *tRAS*. Therefore, the counter for the fifth *ACT* command can be eliminated. It is also the same case for the next three *ACT* commands.

ii) The TA shown in Fig. 5(k) captures both the constraints from a *RD*/*WR* command to an auto-precharge, and the precharge period *tRP*.

iii) Fig. 5(l) tracks the two timing constraints of the switching between read and write transactions.

iv) Finally, we can observe from JEDEC-specified DDR3 timing constraints that $tRCD \leq 2 \times tRRD$. *tRRD* is the minimum time between two successive *ACT* commands. Within $2 \times tRRD$ cycles, at most three *ACT* commands can be scheduled. When the third *ACT* command is scheduled, the counter triggered by the first *ACT* command is guaranteed to be larger than *tRCD* and hence can be reused. Therefore, only two counters are needed to track the intra-bank timing constraint *tRCD*, shown in Fig. 5(c). These optimizations rely on particular relations between timing constraints, e.g., they hold for all DDR3 and LPDDR3.

*7) Reflection:* Although the TA model of the RT memory controller is involved, its structure mirrors that of the memory controller hardware architecture and algorithm (Appendix A). We accurately model all timing constraints, without having to resort to conservative, but pessimistic assumptions such as a worst-case initial state. Moreover, the collision on the command bus is resolved only when needed, rather than conservatively assumed to happen for each *ACT* command. So, the TA model is able to provide better worst-case results than existing analyses, which employ these pessimistic assumptions. Finally, to speed up verification of the model, timing counters were eliminated, although at the cost of model simplicity.

### C. TA Model Extension

The proposed TA model captures the timing behavior of a dynamically-scheduled memory controller [23] for DDR3 SDRAMs. This model can be easily extended to support either memory controllers using different mechanisms and/or new generations of DDR memories, such as DDR4 [15]. For example, the open-page policy used by existing memory controllers [12], [13], [18], [33] can be modeled by employing a TA template to detect the page-hit or page-miss, which determines the scheduling of different commands. To support new DDR memories, we only need to model their timing constraints in the same way as our TA model. For example, DDR4 employs bank groups, where different timing constraints apply to the banks within a group or across groups, respectively. These timing constraints are captured by counters, which can be modeled similarly as, for example, Fig. 5(d) for *tRRD*. *It is worth to emphasize our claim that the extension of the TA model for a different memory controller and/or DDR SDRAM is much easier than carrying out manual worst-case analysis of the complex command scheduling dependencies, since the analysis of a TA model is automatically supported by model checking with existing tools, e.g., Uppaal [3].*

## IV. Verification with Model Checking

In this section, we define worst-case response time (WCRT) and worst-case bandwidth (WCBW), and show how to compute them by checking properties of our TA model. Note that this paper focuses on deriving the WCRT and WCBW for the SDRAM rather than analyzing the worst-case execution time of a task or an application running on processor(s). However, our WCRT and WCBW can be used by existing tools [32] to provide better WCET of tasks/applications, since our worst-case results are tight. Moreover, the proposed TA model of the memory controller can be integrated with other TA models of processor and caches [5], leading to better worst-case bounds. However, this is beyond the scope of this paper.

### A. Definitions

A transaction $T$ arrives at the memory controller of a requestor and is stored in the request buffer in the front-end at time $t_a(T)$, see Fig. 1. The arrival of a write transaction is defined as the time when its last data word is queued in the buffer. A write transaction finishes when the last *WR* command is scheduled at time $t_{lastWR}(T)$. A read transaction finishes when the last required data word is returned from the SDRAM to the response buffer at time $t_{lastRD}(T)$. The response time $t_{RT}(T)$ of a transaction is defined by Eq. (1) as the time between the arrival time and the finishing time of the transaction. It is determined by the interference from other requestors and the transaction's own execution time spent on scheduling its commands. The WCRT is the largest possible $t_{RT}(T)$ for any $T$, in any execution. A smaller WCRT is better.

$$t_{RT}(T) = \begin{cases} t_{lastRD}(T) - t_a(T), & \text{Read transaction} \\ t_{lastWR}(T) - t_a(T), & \text{Write transaction} \end{cases} \quad (1)$$

The memory bandwidth is the long-term rate of data transmission from/into the SDRAM. It is mainly determined by the execution of transactions, i.e., the scheduling of memory commands. In addition, the SDRAM requires refreshing, during which no transactions can be executed. A refresh is required every *tREFI* cycles (7.8 $\mu s$ for DDR3 SDRAMs [14]), and then takes $t_{ref}$ cycles to precharge all banks and finish the refresh itself [23]. Refresh efficiency $e^{ref}$ (Eq. (2)) defines the reduction in bandwidth due to refreshes.

$$e^{ref} = 1 - \frac{t_{ref}}{tREFI} \quad (2)$$

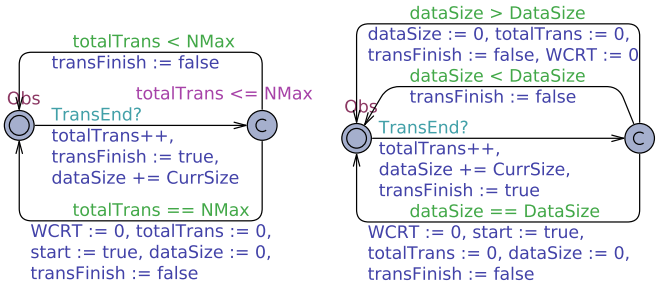
The bandwidth for a given trace $\overline{T}$ of transactions is defined in Eq. (3). $S(T)$ denotes the size of transaction $T$, and $f_{mem}$ is the SDRAM clock frequency. $t_{ET}(T)$ is the execution time of $T$, which is spent on scheduling its command by the back-end of the memory controller. Due to the pipelining between transactions, $t_{ET}(T)$ is the time from either the arrival of $T$ at the back-end or the last *WR*/RD command of the previous transaction, whichever is larger, to its last *WR*/*RD* command. The precise definition is given in [23].

$$bw(\overline{T}) = \frac{\sum_{\forall T \in \overline{T}} S(T)}{\sum_{\forall T \in \overline{T}} t_{ET}(T)} \times f_{mem} \times e^{ref} \quad (3)$$

### B. Worst-Case Analysis with Model Checking

When requestors are served by a TDM arbiter in the front-end of the memory controller (see Fig. 1), a transaction experiences the worst-case response time (WCRT) when the maximum number of interfering transactions must be executed before the execution of this new transaction. Moreover, the execution of all these transactions needs the maximum time to schedule their commands. We assume each requestor has at most one outstanding transaction to avoid arbitrarily high self-interference in the WCRT [2]. Therefore, the total number of interfering transactions and the transaction from the requestor is denoted by *NMax*, which can be computed based on the TDM slot configuration.

An observer TA is designed to track the response time of each transaction from a particular requestor, see Fig. 6(a).




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

Fig. 6. The TA to verify the WCRT and WCBW bounds.

This observer counts the number of executed transactions (totalTrans), and *the end of a transaction is signaled by the TransEnd synchronization label when its last RD or WR command is scheduled.* Meanwhile, the clock variable WCRT in Fig. 6(a) tracks the total time of executing transactions. By specifying the maximum number (i.e., *NMax*) of transactions to be observed, we use standard reachability queries to verify the clock WCRT with the verifier module of the Uppaal tool suite. By manually executing a binary search on the bound of the clock WCRT, it directly translates into the maximum time of executing these *NMax* transactions. The observer records this maximum time when the last *RD* or *WR* command of a transaction is scheduled. For read transactions we need to add the constant data offset, which is $tRL + BL/dataRate$ as given by the JEDEC timing constraints [14]. It is worth noting that the Uppaal model checker explores the full state space for all the possible *NMax* successive transactions and derives the WCRT bound. As a consequence, our observer only needs to cycle through every *NMax* successive transactions rather than uses a sliding window for any *NMax* transactions.

The worst-case bandwidth (WCBW) is defined in Eq. (4) as the minimum bandwidth of all infinitely-long transaction traces, i.e., $\forall \overline{T}, |\overline{T}| = \infty$. However, practically, we can only compute the time to transfer a finite number of bytes (Eq. (5)). We therefore use an observer TA (see Fig. 6(b)) to verify the maximum time to transfer *DataSize* data for any possible traces in the system, which can generate *DataSize* bytes. Intuitively, the minimum rate observed in a long time period of repeatedly transferring a fixed amount of data cannot be larger than the average rate of transferring the total amount of data in the whole time period. Lemma 1 captures this intuition, and states that any $WCBW(DataSize)$ is a conservative lower bound for the *WCBW*. The former is the minimum rate of transferring *DataSize* data, while the latter is the long-term (minimum) average rate of transferring data from/into the SDRAM for infinitely-long traces. The proof is given in the Appendix.

$$WCBW = \min_{\forall \overline{T}, |\overline{T}|=\infty} bw(\overline{T}) \tag{4}$$

$$WCBW(DataSize) = \frac{DataSize}{\mathrm{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)} \times f_{mem} \times e^{ref},$$
$$where\ DataSize = \sum_{\forall T \in \overline{T}} S(T) \tag{5}$$

*Lemma 1:*

$$\forall DataSize > 0, WCBW(DataSize) \leq WCBW$$

Fig. 6(b) shows the observer TA that tracks the total time for transferring *DataSize* data. Time is tracked by reusing the clock variable WCRT, while dataSize accumulates the transferred data when a transaction is finished, as notified by the TransEnd synchronization label. We manually execute a binary search on the bound of the clock WCRT with Uppaal. This bound is the maximum time of transferring *DataSize* data. Multiplying the result by $f_{mem} \times e^{ref}$ returns a conservative lower bound for the WCBW, as described above.

## V. Related Work

Existing analyses of semi-static RT memory controllers [1], [8], [11], [28] provide WCRT and/or WCBW by dynamically using a set of pre-computed static command schedules for transactions. The command schedules cannot exploit dynamic information about the SDRAM state caused by timing constraints and the exact SDRAM banks required by individual transactions. Moreover, these command schedules transfer a fixed amount of data. When the transaction size varies, unwanted data is discarded, resulting in low data efficiency.

To overcome the inefficiency of semi-static command schedules, dynamic command scheduling can be used, where commands are scheduled by some dynamic algorithms when the SDRAM timing constraints are satisfied. However, analysis of dynamic scheduling constrained by timing dependencies is difficult. The analytical approach of [23] abstracts the state of previous transactions to a worst-case initial state, by pessimistically assuming that their commands were scheduled as late as possible (ALAP). This results in conservative command scheduling times for the current transaction. In addition, it assumes that every *ACT* command collides on the command bus. The dataflow model of [24] provides the WCBW of dynamic command scheduling. It also assumes that the *ACT* commands always have command-bus collisions, but does not require the ALAP assumption. Conversely, the scheduled approach [23] accurately models command-bus conflicts, but assumes ALAP schedules. The analyses in [16], [18], [12] assume that the *RW* to *RD* switching timing constraint and the four-activate window constraint are always incurred, even though these constraints do not always dominate in the command schedule. These analysis approaches may not be easy to manually adapt to memory controllers with different mechanisms or memories. Similarly to [24], we shift the manual effort to derive performance bounds from analysis to modeling. In other words, rather than providing a specialized WCRT/WCBW analysis of a memory controller, a specialized model of a memory controller is defined, which is analyzed automatically using state-of-the-art tools. [24] uses dataflow, while this paper uses TA, which is more expressive and results in better bounds.

TA have been used extensively to address the complexity of sharing resources. Yi et al. [27] were the first to use TA to represent a system resource (CPU or communication element) as a scheduler model together with a notion of discrete events that trigger the execution of RT tasks on this scheduler. In [25], the basic idea has been extended to analyze multi-core architectures and different memory access policies. A similar

approach is also presented by Gustavsson et al. [9]. Lampka et al. [19] present an approach that abstracts from individual core-local workloads by modeling access requests to a shared resource with an aggregated access request curve fed into a network of TA. These works intend to bound the worst-case execution time of applications rather than individual memory accesses/transactions. We focus on the effect of sharing on the timing of individual memory transactions in order to find a tight bound. Our WCRT and WCBW bounds can be used by the cited works for more accurate modeling and analysis.

## VI. Experimental Results

This section experimentally validates the proposed TA model of dynamic command scheduling, and then analyzes the WCRT and WCBW results for fixed and variable transaction sizes, respectively. First, the results obtained with Uppaal are validated with the open-source RTMemController tool [20], which has been proven to be equivalent to a cycle-accurate SystemC simulator [22] for capturing the timing behavior of the memory controller. Next, the results are compared to the analysis results of the same memory controller design using (a) the analytical and scheduled approaches presented in [23], and (b) the mode-controlled dataflow (MCDF) model introduced in [24].These techniques were discussed in Section V.

### A. Experimental Setup

The proposed TA model is simulated and verified with Uppaal v4.1.19 on a 64-bit CentOS 6.6 system with 24 Intel Xeon(R) CPUs running at 2.10 GHz and with 125 GB RAM. Experiments have been done with a JEDEC-compliant DDR3-1600G SDRAM memory with interface width of 16 bits and a capacity of 2 Gb [14]. The memory controller front-end uses a TDM arbiter with one slot per memory requestor, which is assumed to have one outstanding transaction to avoid self-interference. The transaction sizes used by the experiments are 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes. The memory map configuration (i.e., BI and BC) of each transaction size is chosen to achieve the lowest execution time and the highest memory bandwidth, where more banks are interleaved when possible to exploit bank parallelism. The configured (BI, BC) for these sizes are hence (1, 1), (2, 1), (4, 1), (4, 2), and (4, 4) [8], respectively. Note that transactions with 128 bytes and 256 bytes use (4, 2) and (4, 4) instead of (8, 1) and (8, 2) because of the tFAW constraint that leads to larger execution time with BI = 8.

### B. Validation of TA Model and WCRT/WCBW

The first experiment shows that the proposed TA model can accurately capture the timing behavior of the memory controller with dynamic command scheduling. We compare the scheduling time of each command in every trace obtained with Uppaal to that of the RTMemController tool [20], which is equivalent to a cycle-accurate SystemC simulator of the memory controller under consideration [22]. We simulate the TA model for 1200 commands corresponding to random read and write transactions with different physical addresses (i.e., the starting bank is different). The transactions are generated by the Source and TDM Bus TA shown in Fig. 5(a). We execute RTMemController with the same transactions to obtain the scheduling time of each command. The same experiment is repeated for the 5 transaction sizes and for a random mix of them. From the experimental results, we observe that *the scheduling time is always identical for each command*. This suggests that our TA model correctly and accurately captures the timing behavior of the dynamic command scheduling. For the given traces, *the TA model is equivalent to the cycle-accurate implementation of the dynamic command scheduling.*

For all experiments in the following sections, Uppaal generates a witness that leads to the WCRT/WCBW, i.e., the diagnostic trace of transactions. Again, we have fed all diagnostic traces to the RTMemController tool, and it always shows exactly the same results as Uppaal. *This validates the correctness of the TA model, and gives strong reason to believe that the analysis results derived from our TA model are tight.*

### C. Fixed Transaction Size

This experiment uses Uppaal to test the TA model and to obtain the WCRT and WCBW for fixed transaction sizes. Four memory requestors are employed, corresponding to, e.g., four cores that have the same cache-line size. This experiment tests an arbitrary read/write mix, for the five different transaction sizes. The model checker explores the full state space for each size, except for 16 bytes that uses BI=1. For the purpose of worst-case analysis, it is not necessary to explore all 8 banks. Transactions with 16 bytes only access a single bank. It hence only matters if transactions access the same bank or a different bank. As a result, we arbitrarily select 2 banks (e.g., Bank 0, Bank 1) to be tested by Uppaal. The trace is an arbitrary read/write mix, but we show the WCRT for read (RD) and write (WR) transactions separately in Fig. 7. They are also compared to those given by the existing analytical and scheduled approaches of [23]. Note that we do not compare with the MCDF model of [24] as it does not analyze the WCRT. Moreover, each result is validated by executing the diagnostic trace from Uppaal with RTMemController.

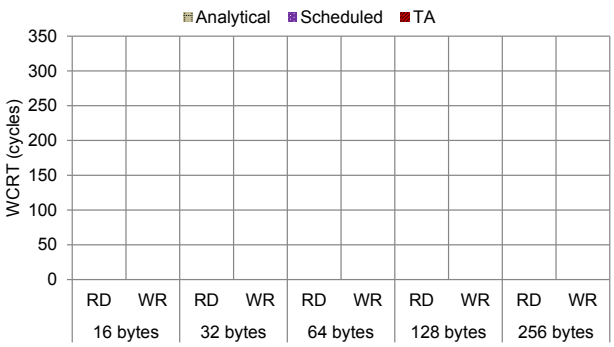



Fig. 7. The WCRT for 4 requestors with fixed transaction sizes.

We observe that *the WCRT results from the TA model are better than or equal to those given by either the scheduled or analytical approaches.* The maximum improvement is 20% for write transactions with 64 bytes, while the average improvement over all these experiments is 7.7%. The improvement is achieved for the following two reasons. 1) The TA model accurately models scheduling collisions on the command bus, just like the scheduled approach. Conversely, the analytical approach always conservatively assumes a collision for each *ACT* command. 2) The TA model accurately captures the

worst-case initial values of the timing counters for an arbitrary transaction. In contrast, both the analytical and scheduled approaches conservatively assume as-late-as-possible (ALAP) scheduling of the previous commands to provide the worst-case initial values of timing counters, which is pessimistic.

The WCBW for different fixed transaction sizes is shown in Fig. 8. With a trace $\overline{T}$ of 8 transactions, the WCBW obtained with the TA model is already better than that of the analytical and scheduled approaches, as well as of the MCDF model [24]. Compared to the analytical approach, the improvement reaches up to 25% for 64-byte transactions. The average improvement on the WCBW bounds is 13.6% for all these experiments. The reasons for obtaining better WCBW than analytical and scheduled approaches are the same as those for WCRT. Our TA model is better than or performs equally well as the MCDF model [24] because the latter conservatively assumes a collision per ACT command. Larger improvements are obtained for small transactions, while the same WCBW is obtained for large transaction sizes, e.g., 256 bytes. This is because larger transactions have more *RD* or *WR* commands, which dominate the command scheduling. As a result, the collisions with ACT commands have no influence on the WCBW, and the MCDF can perform equally well as our TA model for large transaction sizes.

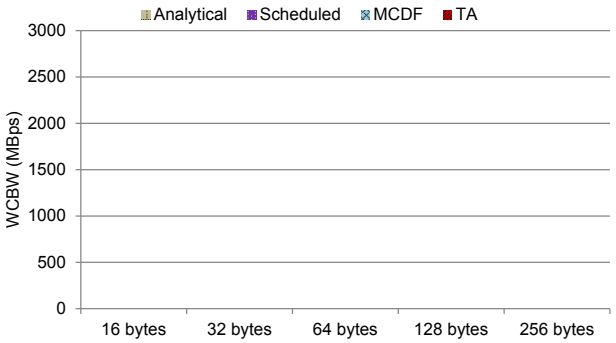



Fig. 8. The WCBW for fixed transaction sizes.

We evaluate the run time and memory usage of Uppaal in our experiments. Uppaal takes at most 1221 seconds and consumes up to 7 GB to successfully verify properties for fixed transaction sizes. This occurs when verifying a property to derive the WCRT of 16-bytes write transactions. Note that this experiment explores two different starting banks for 16-byte transactions. If more starting banks (e.g., 4 banks) are explored, it takes around 30 hours before running out of the 125 GB RAM memory. Due to the limited RAM memory, we alternatively carry out this experiment on a server with 1 TB RAM and an Intel(R) Xeon(R) CPU E7-4850 running at 2.0 GHz. Note that this server is remotely provided by SURF-sara [31], a Dutch Cooperative providing high-performance computing and data infrastructure for science and industry. The model checking finally takes around 241.1 hours and consumes 557.9 GB RAM memory to provide the same results as when only exploring 2 banks. We have observed that larger transaction sizes need less time and RAM to verify a property. The reasons include: 1) Larger transactions use larger BI that have a fewer possible starting banks, resulting in a smaller state space. 2) The scheduling algorithm schedules more commands sequentially (i.e., deterministically) for larger transactions. In contrast, smaller transactions have fewer commands and transactions arrive randomly. As a result, the TA model performs more deterministic state transitions. Recall that the scheduling of commands is modeled by state transitions.

### D. Variable Transaction Sizes

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

To illustrate this effect, we perform a case study of the HD video and graphics processing system of [6]. It consists of 5 requestors representing CPU, GPU, input processor (IP), video engine (VE), and HDLCD DMA controller. [6] focuses on a multi-channel memory controller with 256-byte transactions that are interleaved over multiple channels. Since our memory controller has a single channel, we use smaller transaction sizes, which are also current practice in today's systems [7], [10]. CPU and IP have cache-lines of 64 bytes, while those of the GPU are 128 bytes. The VE and the HDLCD DMA use transactions of 128 bytes. All produce an arbitrary read/write mix of transactions. The five requestors have one TDM slot each, and are serving in descending order of their transaction sizes. This ordering increases the bank parallelism between successive transactions [23], improving performance. Fig. 9 separately shows the WCRT of read/write transactions, for each requestor.

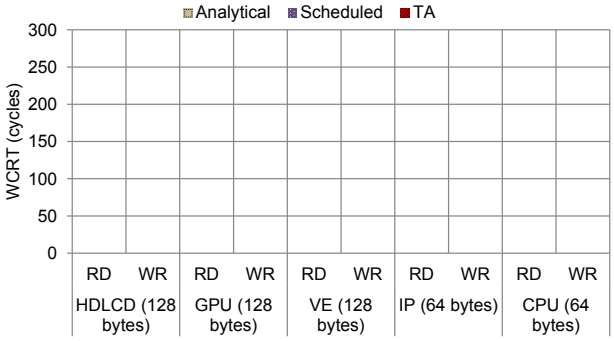



Fig. 9. The WCRT for the requestors in a HD video and graphics processing system [6] with variable transaction sizes.

Our TA model outperforms the analytical and scheduled approaches for WCRT, for the same reasons as discussed in Section VI-C. For example, our TA model improves the WCRT of 128-byte transactions of analytical and scheduled

approaches by 10.4% and 8.5%, respectively. As before, all bounds have been validated to be identical to the cycle-accurate timings of the RTMemController tool.

The WCBW results obtained from different approaches are shown in Fig. 10. Using the WCBW bound obtained with 8 successive transactions (cf. Section VI-C), our TA model outperforms the analytical and scheduled approaches by 9.3% and 7.1%, respectively. We cannot compare to the MCDF model [24], as its analysis tool does not support our use-case. Instead, we compare all approaches for a system with five requestors, with arbitrary read/write transactions of 64 or 128 bytes. The arbitration is unknown (not specified). The WCBW results are shown in Fig. 10. The TA model outperforms the analytical, scheduled, and MCDF approaches by 24.6%, 14.6%, and 12.1%, respectively.

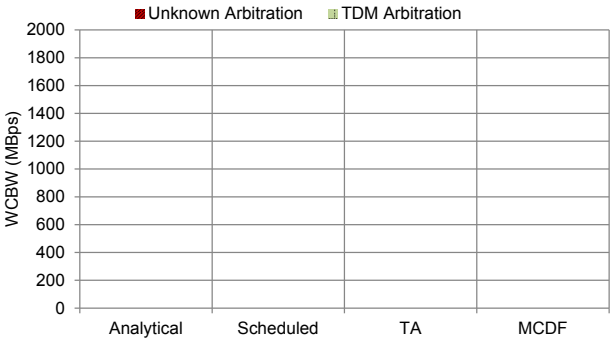



Fig. 10. The WCBW for variable transaction sizes.

## VII. Conclusion

This paper proposes a modular Timed Automata (TA) model of an SDRAM memory controller with dynamic command scheduling. It accurately captures both the architecture and command scheduling algorithm. When modeling other memory controllers with different mechanisms, the TA of the common components can be reused, while it only requires to extend other TA to capture their timing behavior, as opposed to repeat a time-consuming manual analytical effort. The worst-case response time (WCRT) and worst-case bandwidth (WCBW) of the memory controller can be automatically derived using Uppaal. The command schedules of diagnostic traces provided by Uppaal for the WCRT and WCBW are identical to those given by cycle-accurate simulation of those traces, providing strong validation of our results. Moreover, the experimental results demonstrate that the proposed TA model outperforms three state-of-the-art analysis approaches of dynamic command scheduling for real-time memory controllers, by up to 25%. The reason is that the TA model accurately captures both the scheduling collisions on the command bus and the initial timing states of SDRAM for each transaction.

## Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0001/2013-JU grant nr. 621429 (EMC2); also partially funded by projects CATRENE CA505 BENEFIC, CA703 OpenES, CT217 RESIST, and 621353 DEWI. This work was partially carried out on the Dutch national e-infrastructure with the support of SURF Cooperative under number e-infra150168.

## References

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

[2] P. Axer *et al.* Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, 2014.

[3] G. Behrmann *et al.* Uppaal 4.0. In *Proc. QEST*, 2006.

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

[5] A. E. Dalsgaard *et al.* Metamoc: Modular execution time analysis using model checking. In *WCET*, 2010.

[6] M. D. Gomony *et al.* A real-time multi-channel memory controller and optimal mapping of memory clients to memory channels. *TECS*, 2015.

[7] K. Goossens *et al.* Interconnect and memory organization in SOCs for advanced set-top boxes and TV — evolution, analysis, and trends. *Interconnect-Centric Design for Advanced SoC and NoC*, 2004.

[8] S. Goossens *et al.* Power/performance trade-offs in real-time SDRAM command scheduling. *IEEE Trans. on Computers*, PP(99):1–1, 2015.

[9] A. Gustavsson *et al.* Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Workshop on WCET Analysis*, 2010.

[10] A. Hansson *et al.* A quantitative evaluation of a network on chip design flow for multi-core consumer multimedia applications. *Design Automation for Embedded Systems*, 15(2), 2011.

[11] M. Hassan *et al.* A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proc. RTAS*, 2015.

[12] K. Hokeun *et al.* A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Proc. RTAS*, 2015.

[13] J. Jalle *et al.* A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *Proc. RTSS*, 2014.

[14] JEDEC Solid State Technology Association. *DDR3 SDRAM Specification*, 2010.

[15] JEDEC Solid State Technology Association. *DDR4 SDRAM Specification*, 2012.

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

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

[18] Y. Krishnapillai *et al.* A rank-switching, open-row DRAM controller for time-predictable systems. In *Proc. ECRTS*, 2014.

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

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

[21] Y. Li *et al.* Timed automata model of a dynamically-scheduled real-time memory controller. http://www.es.ele.tue.nl/rtmemcontroller/TA.zip.

[22] Y. Li *et al.* Dynamic command scheduling for real-time memory controllers. In *Proc. ECRTS*, 2014.

[23] Y. Li *et al.* Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, pages 1–55, 2015.

[24] Y. Li *et al.* Mode-controlled data-flow modeling of real-time memory controllers. In *Proc. ESTIMedia*, 2015.

[25] M. Lv *et al.* Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proc. RTSS*, 2010.

[26] A. Nelson *et al.* Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. *Journal of Systems Architecture*, 2015.

[27] C. Norström *et al.* Timed Automata as Task Models for Event-Driven Systems. In *Proc. RTCSA*, 1999.

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

[29] H. Shah *et al.* Bounding SDRAM interference: Detailed analysis vs. latency-rate analysis. In *Proc. DATE*, 2013.

[30] A. Stevens. Qos for high-performance and power-efficient HD multimedia. 2010.

[31] SURFsara. https://www.surf.nl/en/about-surf/subsidiaries/surfsara.

[32] R. Wilhelm *et al.* The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

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

## APPENDIX

### A. Command scheduling algorithm reproduced from [23]

The command scheduling algorithm in [23] uses three rules: 1) transactions are executed in first-come-first-serve (FCFS) order to avoid reorder buffers for the responses, 2) to simplify logical-to-physical address translation [8], successive banks of a single transaction have to be accessed in ascending order, and 3) *RD* or *WR* commands have higher priority than *ACT* commands to avoid collisions on the command bus when timing constraints are satisfied for both of them.

We reproduce the algorithm in [23] in Algorithm 1 to adapt to the notation of this paper. It has three inputs. The FIFO queue PARQueue contains the information (type, BI, BC, and BS) of each transaction arrived in the back-end of the memory controller. Recall that a new transaction arrives only if all the *ACT* commands have been scheduled for the current transaction. As a result, only the tail element of the PARQueue corresponds to the latest transaction that has *ACT* commands. For the rest, there are only *RD* or *WR* commands to be scheduled. Secondly, the constraint_satisfied indicates whether the timing constraints are satisfied for commands at the head of each command queue. If timing constraints are satisfied, the head command becomes valid. Finally, the cmd_type shows whether a head command is a *RD*, a *WR* or an *ACT*. The output of Algorithm 1 is bank_id indicating the command queue whose head command can be scheduled to bank bank_id.

Algorithm 1 uses two internal states ACTBank and RWBank, which indicate the number of the bank to which an ACT or a RD/WR command can be scheduled, respectively. When scheduling commands for a transaction, ACTBank and RWBank are initialized with the starting bank $b_s$ of the transactions associated with the tail and head of the PARQueue, respectively (line 4, 5). This guarantees the FCFS order of transactions. In Algorithm 1, line 6 checks whether there is a valid (i.e., timing constraints are satisfied) *RD*/*WR* command for the current bank (RWBank). Otherwise, line 13 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. ACTBank is increased by one after an *ACT* command has been selected (line 17), while RWBank increases by one when *BC* number of *RD*/*WR* commands of the current transaction have been scheduled to bank *bank_id* (line 11). This update scheme ensures that the banks are accessed in ascending order for each transaction. 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 1** Dynamic command scheduling

```
1: Inputs: PARQueue, constraint_satisfied, cmd_type
2: Internal state: ACTBank, RWBank
3: Initialization: bank_id ← null; ACTBank ← null; RWBank ← null;
4: if ACTBank = null then ACTBank ← PARQueue_tail.b_s;
5: if RWBank = null then RWBank ← PARQueue_head.b_s;
6: if cmd_type[RWBank] = RD/WR and constraint_satisfied[RWBank] = true then
7:     bank_id ← RWBank;
8:     if last RD/WR of PARQueue_head transaction then
9:         RWBank ← null;
10:    else if last RD/WR of PARQueue_head transaction to bank bank_id
11:        then RWBank ← RWBank+1;
12: else if ACTBank != null and then
13:        if cmd_type[ACTBank] = ACT and constraint_satisfied[ACTBank] = true then
14:        bank_id ← ACTBank;
15:        if last ACT of PARQueue_tail transaction then
16:            ACTBank ← null;
17:        else ACTBank ← ACTBank+1;
18: Outputs: bank_id
```

### B. Proof of Lemma 1

*Proof:* For a given *DataSize*, there is $\mathit{DataSize} = \sum_{\forall T \in \overline{T}} S(T)$ and its maximum execution time is $\mathrm{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)$, which can be obtained by verifying the bound of the clock WCRT with the observer TA shown in Fig. 6(b). For $\forall N > 1$, the larger data size $\mathit{DataSize}' = N \times \mathit{DataSize} = \sum_{\forall T \in \overline{T'}} S(T)$ corresponding to trace $\overline{T'}$, and its maximum execution time is $\mathrm{Max}_{\forall \overline{T'}} \sum_{\forall T \in \overline{T'}} t_{ET}(T)$. Conservatively, we obtain Eq. (6), since the transaction trace $\overline{T'}$ generates $N$ times more data than the trace $\overline{T}$. Therefore, Eq. (7) is derived, which shows that better WCBW can be obtained when verifying with larger data size. Intuitively, larger amount of data is generated by more transactions, where more pipelining between transactions can be exploited to achieve better WCBW. When $N$ approaches $+\infty$, we get the long-term WCBW, which is given by Eq. (8). According to Eq. (7), we can conclude that the $\mathit{WCBW}(\mathit{DataSize})$ of any given *DataSize* is a conservative lower bound for the long-term *WCBW*.

$$\max_{\forall \overline{T'}} \sum_{\forall T \in \overline{T'}} t_{ET}(T) \leq N \times \max_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T) \tag{6}$$

$$\begin{aligned} \mathit{WCBW}(N \times \mathit{DataSize}) &= \frac{N \times \mathit{DataSize}}{\mathrm{Max}_{\forall \overline{T'}} \sum_{\forall T \in \overline{T'}} t_{ET}(T)} \\ &\geq \frac{N \times \mathit{DataSize}}{N \times \mathrm{Max}_{\forall \overline{T}} \sum_{\forall T \in \overline{T}} t_{ET}(T)} \\ &\geq \mathit{WCBW}(\mathit{DataSize}) \end{aligned} \tag{7}$$

$$\begin{aligned} \mathit{WCBW} &= \lim_{N \to +\infty} \mathit{WCBW}(N \times \mathit{DataSize}) \\ &\geq \mathit{WCBW}(\mathit{DataSize}) \end{aligned} \tag{8}$$

■