Mode-Controlled Data-Flow Modeling of Real-Time Memory Controllers

Yonghui Li[1], Hrishikesh Salunkhe[1], Joao Bastos[1], Orlando Moreira[2], Benny Akesson[3] and Kees Goossens[1]
[1]Eindhoven University of Technology, [2]Intel Corporation, [3]CISTER/INESC TEC, ISEP

***Abstract*—SDRAM is a shared resource in modern multi-core platforms executing multiple real-time (RT) streaming applications. It is crucial to analyze the minimum guaranteed SDRAM bandwidth to ensure that the requirements of the RT streaming applications are always satisfied. However, deriving the worst-case bandwidth (WCBW) is challenging because of the diverse memory traffic with variable transaction sizes. In fact, existing RT memory controllers either do not efficiently support variable transaction sizes or do not provide an analysis to tightly bound WCBW in their presence.**

We propose a new mode-controlled data-flow (MCDF) model to capture the command scheduling dependencies of memory transactions with variable sizes. The WCBW can be obtained by employing an existing tool to automatically analyze our MCDF model rather than using existing static analysis techniques, which in contrast to our model are hard to extend to cover different RT memory controllers. Moreover, the MCDF analysis can exploit static information about known transaction sequences provided by the applications or by the memory arbiter. Experimental results show that 77% improvement of WCBW can be achieved compared to the case without known transaction sequences. In addition, the results demonstrate that the proposed MCDF model outperforms state-of-the-art analysis approaches and improves the WCBW by 22% without known transaction sequences.

I. Introduction

The memory subsystem is becoming critical to the overall performance of multi-core systems, since the off-chip SDRAM is shared by an increasing number of memory clients (e.g., cores, DMAs, and hardware accelerators) that execute more memory-intensive real-time (RT) streaming applications [1]. The analysis of worst-case bandwidth becomes a key issue to meet the throughput requirements of these RT applications. However, this analysis is difficult because of the diverse memory traffic generated by the different clients [2], [3]. It is a mix of read and write transactions with variable sizes, which access different sets of SDRAM banks. They are executed in a pipelined manner by scheduling memory commands, a process which is highly dependent on when the previous commands (possibly corresponding to previous transactions) were scheduled. These complicated dependencies make the worst-case bandwidth analysis challenging.

Most existing real-time memory controllers are limited to a single transaction size, either in architecture or analysis, and cannot efficiently deal with diverse memory traffic. The memory controller in [4] addresses the diversity by dynamically scheduling commands for each memory transaction. However, its worst-case execution time (WCET) of transactions is derived by a complex static analysis approach, where exploiting static information, e.g., the static sequence of transactions specified by the application or by the arbitration of memory clients, to obtain better worst-case results is difficult. Moreover, it does not provide bounds on worst-case bandwidth. Another issue with state-of-the-art RT memory controller analyses is that all the manual time-consuming analysis has to be done all over again if there are any changes to the mechanisms of the memory controller because of the complex command scheduling dependencies. As a result, the existing analyses are difficult to extend.

978-1-4673-8164-2/15/$31.00

In this paper, we propose to use mode-controlled data-flow (MCDF) models [5] to capture the memory command scheduling, where the command scheduling dependencies are described by an MCDF graph. Moreover, the worst-case bandwidth (bytes/second) is used as a notion of the critical cycle path of the MCDF graph rather than the typical maximum cycle mean (MCM) for worst-case throughput (tokens/second) of data-flow graphs. The advantages of the MCDF model include: 1) It leverages standard data-flow analysis techniques and tools to analyze the worst-case bandwidth of memory command scheduling without the need to manually develop new complex static analyses. 2) It can easily exploit static information, such as the transaction sequence given by the application or static arbitration of memory clients (e.g., time-division multiplexing). 3) The analysis of the MCDF model returns the critical cycle that shows the sequence of commands (corresponding to transactions) that limit the worst-case bandwidth, which is beyond the capability of existing analyses. This information is useful when designing scheduling algorithms, such that the critical sequence of transactions is avoided and hence a better worst-case bandwidth is obtained. 4) The validation of the MCDF model is easier than existing analyses because the formal model is also executable. 5) The MCDF model can be easily adapted to cover other memory controllers with different scheduling policies. 6) Finally, the worst-case bandwidth results are better than state-of-the-art analyses [4], [6] with a maximum improvement of 22%. We also experimentally show that exploiting static sequences of transactions achieves up to 77% higher worst-case bandwidth.

In the remainder of this paper, Section II describes the related work, followed by the background of RT memory controllers and MCDF modeling in Section III. The MCDF modeling of memory command scheduling is given in Section IV, while the WCBW analysis is presented in Section V. Experimental results are shown in Section VI, before this paper is concluded in Section VII.

II. Related Work

The worst-case memory bandwidth is challenging to analyze because of the command scheduling dependencies based on the complex internal states of SDRAM [7] and the diverse memory traffic. Most existing approaches to compute memory bandwidth abstract away the complexity of SDRAM internal states. A memory access control approach has been proposed in [8] to allocate enough bandwidth to a critical core that runs a real-time application. However, it uses constant memory access time to compute the bandwidth, which is pessimistic for variable transaction sizes with different execution time. This drawback also applies to the bandwidth sharing scheme in [9]

that treats every memory access as a constant delay. A mixed-criticality memory controller in [10] provides guaranteed bandwidth based on a fixed TDM schedule of command scheduling. This is similar to the Predator controller [6] that computes bandwidth based on semi-static command schedules. Both of them cannot efficiently deal with variable transaction sizes because the hardware restricts the number of static schedules. These problems also apply to [11], resulting in pessimisms in its WCBW. The PRET DRAM controller [12] computes bandwidth based on the conservative periodic cycles of issuing commands, leading to pessimisms in the WCBW as well. The dynamically-scheduled memory controller in [4] avoids this hardware restriction by dynamically scheduling commands for each transaction. However, its analysis is complicated and only the WCET is provided. In this paper, we tackle this complexity by modeling command scheduling with a data-flow model [5], where existing analysis techniques and tools can be used.

Data-flow models have been widely used to model shared resources in modern multi-core systems and provide guaranteed performance. For example, the behavior of a network-on-chip (NoC) is captured by a data-flow model [13] that is used to compute the required buffer size of a network interface, such that the performance of an application is guaranteed. Another example is the data-flow modeling of TDM arbitration [14], that enables an optimized TDM slot allocation to meet the requirements of concurrent applications. The data-flow models of these two examples actually describe the dependencies of resource sharing, and existing data-flow analysis techniques are employed to provide the worst-case results. *This paper proposes to capture the complex command scheduling dependencies by a data-flow model, where we extend an analysis tool to address these complexities and provide the WCBW.*

III. Background

This section introduces background information about SDRAM and a dynamically scheduled real-time memory controller [4], followed by the basic concepts and analysis of data-flow models in general and mode-controlled data-flow models [5] in particular.

A. Real-Time Memory Controller

A memory controller receives transactions from the memory clients, such as processor cores, last-level caches, DMAs or hardware accelerators. These heterogeneous components generate diverse memory traffic in terms of mixed read or write transactions with variable sizes. The memory controller translates each transaction into a sequence of memory commands that are scheduled to the SDRAM, as shown in Fig. 1. For RT streaming applications, the memory controller must offer guaranteed bandwidth to ensure correct operation [15].

A DDR3 SDRAM chip is composed of 8 banks, which consists of memory cells arranged in rows and columns, as presented in Fig. 1. Each cell contains a number of data bits. This paper focuses on a single chip, which is common in the embedded domain, but can be easily extended to multiple chips forming more than one rank. The SDRAM is controlled by different commands via the command bus, while the physical address of each command in terms of bank, row, and column is sent on the address bus. As shown in Fig. 1, an activate (*ACT*) command copies the contents of the required row in an array into the row buffer, such that a burst of data can be read or written triggered by a read (*RD*) or write (*WR*) command. As a consequence, the data burst is transferred via the data bus, i.e. read from or written into the SDRAM. The burst length (BL) is 8 words for all DDR3 SDRAMs. A *RD* or *WR* command is followed by a precharge (*PRE*) command to write back the contents of the row buffer to the storage cells. A close-page policy is often used by RT memory controllers [6], [16], [4] to achieve better worst-case results, and it implies that a *PRE* is always scheduled after finishing read or write, regardless of which bank or row it targets. Note that the scheduling of a command has to satisfy timing constraints that are specified by the JEDEC DDR3 standard [7]. The inter/intra-bank timing constraints are the minimum time between commands to SDRAM banks. Finally, the SDRAM device has to be refreshed periodically to maintain the data stored in the cells. The relevant timing constraints of DDR3 SDRAM are shown in Table I, where a 16-bit DDR3-1600G device with a capacity of 2 Gb is taken as an example.

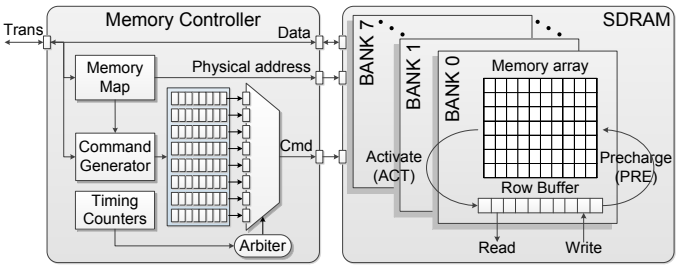


Fig. 1. The general structure of a dynamically scheduled memory controller [4] and the SDRAM.

TABLE I. Timing constraints for DDR3-1600G SDRAM [7].

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 time	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

The memory controller in [4] executes a transaction by interleaving it over a number of banks in parallel, and each bank may receive several data bursts. The number of interleaved banks (BI) and the data burst count (BC) per bank are used by the *Command Generator* in Fig. 1 to determine the total number of commands that must be generated for a transaction. To access a bank using close-page policy, an *ACT* is required to open the target row followed by BC *RD* or *WR* commands that transmit the data. Finally, the data in the row buffer are copied back to the array by issuing either an explicit *PRE* or using an auto-precharge flag attached to the last *RD*

or *WR* command of a bank. These commands are sequentially buffered in the queue per bank. This repeats BI times for all the required consecutive banks, and the data access granularity is $BI \times BC \times BL$ words [17]. By varying BI and BC, different transaction sizes are supported [4].

The commands in the queues are scheduled by an arbiter subject to the timing constraints that are tracked by timing counters, as shown in Fig. 1. *RD* and *WR* commands are scheduled in a first-come first-serve (FCFS) manner of transactions to ensure coherent memory. While *ACT* commands to different banks are issued in a pipelined manner with the prioritized *RD* or *WR* commands to improve performance.

Fig. 2 shows the scheduling dependencies between commands to any two sequentially accessed banks, which are caused by the JEDEC timing constraints [7] and the order specified by the command scheduling algorithm in [4]. 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. b_j denotes the bank identifier (id) and $b_j \in [0, 7]$ for DDR3 SDRAMs. The solid and dotted arrows in Fig. 2 represent dependencies within a bank and between two successive banks, respectively. The labels near the arrows specify the timing constraints given in Table I or derived from [4]. For example, the timing constraints to schedule an *ACT* include *tRRD*, *tRP* and *tFAW*. Therefore, an *ACT* has three input arrows that denote the corresponding timing constraints.

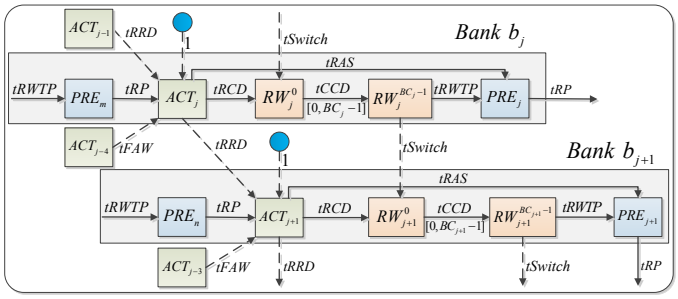


Fig. 2. The command scheduling dependencies of successively accessing any two banks [4].

As can be seen from Fig. 2, the scheduling of an *ACT* can be pipelined with *RD* or *WR* commands for previous banks. Since the command bus only sends one command per cycle, an *ACT* may be blocked by a prioritized *RD* or *WR* [4]. This command scheduling conflict postpones the *ACT* by one cycle. The collision is depicted by the filled circle in Fig. 2, which represents the blocking *RD* or *WR* command.

B. Data-Flow Modeling

Data-flow models are popular to describe concurrent processes with unidirectional graphs, where a process is represented by a node (i.e., *actor*) while an edge between two nodes is a FIFO communication *channel* between the corresponding processes. In the simplest version of data-flow, e.g., single rate data-flow (SRDF), each actor has a fixed execution time and communicates with other actors using tokens (i.e., a chunk of data) through its channels. An actor *fires*, i.e. the process executes, by consuming one token from each input channel and producing one token on each output channel. Initial tokens are specified on some edges of the SRDF graph, such that the graph starts firing with particular actor(s). An SRDF model expresses the dependencies between concurrent processes while having good analytical properties that guarantee the latency and throughput of the graph [18].

Mode-controlled data-flow (MCDF) [5] is a restricted variant of Boolean data-flow [19] that supports dynamism by selecting different sub-graphs of the MCDF graph to fire for each graph iteration. These sub-graphs, called modes, are actually smaller data-flow graphs. MCDF features single rate dataflow (SRDF) actors and two types of special actors, named select and switch, which conditionally consume/produce tokens from/on specific edges depending on the mode selected for that firing, which is defined by the value of the token consumed from its mode control input. In addition, a special single-rate actor is marked as mode controller (MC) and produces all tokens consumed through the control ports of all switches and selects in the MCDF graph. For each firing of MC, one token with the same mode value is produced by MC on all control inputs of all switches and selects, which are enabled to fire exactly once in an iteration. Note that an iteration of an MCDF graph is defined as a set of actor firings, such that all the initial tokens return to their initial edges, i.e., the MCDF graph returns to the initial state. The single-rate actors can be either amodal or modal depending on topology. In each iteration, all amodal actors and all modal actors of a single selected mode fire exactly once, while all modal actors of non-selected modes do not fire.

The *construction rules* of an MCDF model are that 1) it uses a single MC actor and an arbitrary number of switch, select and tunnel actors. These actors always fire for any chosen mode. 2) MC selects a mode by sending a control token. The switch and select activate the actors of the selected mode to fire. The actors of unselected modes cannot fire. 3) An actor is not allowed to belong to more than one mode. With these rules, the MCDF model has strong expressiveness to capture the dynamism of a system by dynamically choosing modes. *A pre-defined static mode sequence (SMS) specifies a static order of modes to fire, while multiple SMSs can be used dynamically in any random order.* In addition, MC can repeat an SMS, resulting in *recurring SMS*. Note that a MCDF model is able to guarantee analytical properties [5].

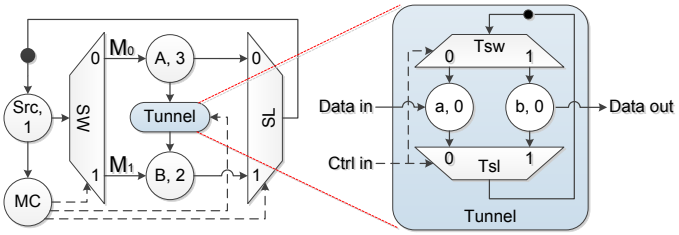
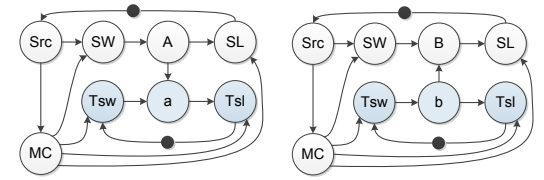


Fig. 3. An MCDF graph and a basic tunnel.

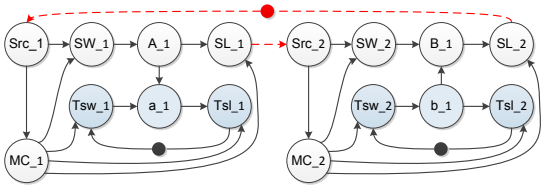
Fig. 3 shows a simple MCDF graph that consists of two modes, M_0 and M_1, where the former consists of actor A and latter actor B. All other actors do not belong to any mode, as they fire once per iteration, independently from the values of the mode control tokens. The MC produces control tokens that are sent to the control input port of the switch (SW), select (SL), and tunnel. Tunnel actors encapsulate an MCDF construct enabling well-defined communication between different modes, as explained below. Besides the control input port, a SW has a data input port and a number of output ports that connect to actors belonging to different modes. The SW consumes both the data token sent by the source (Src) actor and the control token given by MC, and produces the same data token on the output port that connects to the mode specified by the control token. Conversely, a SL consumes the control token and the input data token associated with the mode indicated by its received control token, and produces the same data token on the output port. Fig. 3 also shows a tunnel constructed by

a switch (Tsw) and a select (Tsl). It has an internal (initial) token that is always replaced by its input data token. As a result, it always passes the latest token from the input mode to the output mode via the data out port.

By choosing modes according to pre-defined static mode sequences (SMS), a *worst-case throughput analysis* of the MCDF model can be based on the SMSs. Each SMS specifies a static firing order of modes. *The firing dependencies of a recurring SMS are hence equivalently described by a static data-flow graph, which is obtained by eliminating the actors and edges (i.e., dependencies) of the modes that are not chosen by the SMS.* We simply assume that SMS_1 only contains mode M_0 and SMS_2 has mode M_1 (see Fig. 3), i.e., $SMS_1=[M_0]$ and $SMS_2=[M_1]$. When SMS_1 or SMS_2 is repeatedly used by MC, recurring mode sequences are brought and are represented by $[M_0]^*$ and $[M_1]^*$ for SMS_1 and SMS_2, respectively. The equivalent SRDF graphs are shown in Fig. 4(a) and Fig. 4(b) for $SMS_1=[M_0]^*$ and $SMS_2=[M_1]^*$, respectively. For a recurring $SMS_3=[M_0, M_1]^*$, its equivalent SRDF is shown in Fig. 4(c) that is obtained by unrolling the MCDF model in Fig. 3, where M_0 is always followed by M_1 and the transitions are denoted by the red dashed edges. Therefore, to analyze the worst-case throughput of a given SMS, we only need to analyze its equivalent static data-flow graph with existing data-flow analysis techniques [5].



(a) SRDF for $SMS_1=[M_0]^*$ (b) SRDF for $SMS_2=[M_1]^*$



(c) SRDF for $SMS_3=[M_0, M_1]^*$ by unrolling the MCDF graph

Fig. 4. The equivalent SRDF of recurring SMS for the MCDF in Fig. 3.

The transitions across multiple SMSs are usually not known apriori, since they are dynamically executed. All the possible transitions can be described by a finite-state machine (FSM), where each SMS is represented by a state that is able to transit to any states including itself. By assuming the total number of SMSs to be *NS* ($NS > 0$), this case is described by $[SMS_1 \mid SMS_2 \mid ... \mid SMS_{NS}]^*$, where the transitions are given by the FSM. The worst-case analysis approach in [20] actually does not need to explore all the transitions of the FSM to obtain the worst-case results of $[SMS_1 \mid SMS_2 \mid ... \mid SMS_{NS}]^*$ for an MCDF model. Instead, it merges all the equivalent static data-flow graphs of each individual recurring SMS (e.g., SMS_i, $\forall i \in [1, NS]$), resulting in a larger equivalent static data-flow graph that captures all the dependencies of dynamically executing an arbitrary SMS_i. This merging is achieved by adding all the dependencies (i.e., edges with initial token(s)) between the actors chosen by different SMSs. For example, Fig. 5 shows the merging of the SRDF graphs of $SMS_1=[M_0]^*$ and $SMS_2=[M_1]^*$, which are shown in Fig. 4(a) and Fig. 4(b). The added dependencies are denoted by those red dashed edges with an initial token. The merged graph is the equivalent SRDF of executing $[SMS_1 \mid SMS_2]^*$. Therefore, it only requires to equivalently analyze the merged static data-flow graph to derive the worst-case results.

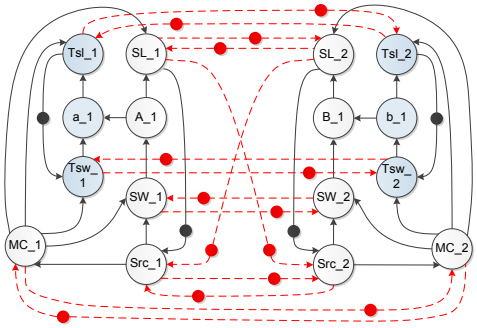


Fig. 5. Merging the equivalent SRDF graphs of SMS_1 and SMS_2. This results in the equivalent SRDF graph of $[SMS_1 \mid SMS_2]^*$.

Fig. 6 illustrates the execution of each actor in Fig. 5 for two iterations. The execution trace in Fig. 6 shows that the actor firings of each SMS in a new iteration depends on the slowest SMS executed in the previous iteration. *Therefore, the worst-case situation is guaranteed for any SMS that is dynamically executed.* As highlighted by the red dashed arrows, the firing of both SMS_1 and SMS_2 in the second iteration starts after the finishing of SMS_1 in the first iteration. The reason is that actor A has the longest execution time (i.e., 3) in the MCDF graph in Fig. 3, resulting in the critical path (shown by the red dashed arrows) of executing SMS_1 in the first iteration.

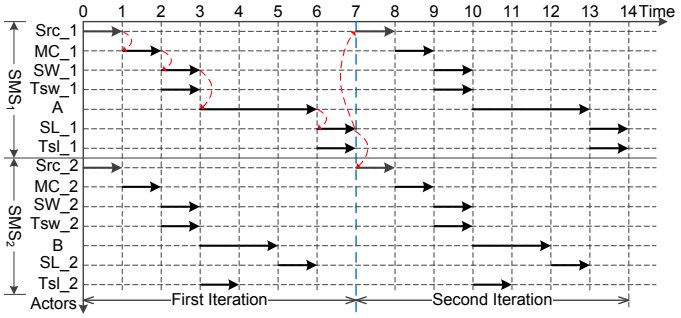


Fig. 6. The execution of the merged equivalent SRDF graph shown in Fig. 5.

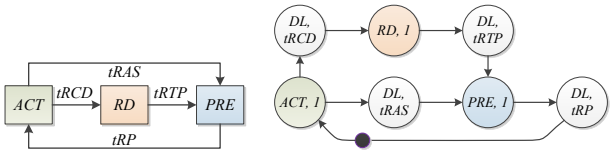
IV. MCDF Modeling of Memory Command Scheduling

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

A. Data-Flow Modeling of Command Scheduling

The timing dependencies of command scheduling are essentially the same as the data dependencies described by data-flow graphs. A command can be scheduled only if all its timing constraints are satisfied, as shown in Fig. 2, while a data-flow actor fires when all its input tokens are available. Therefore, a data-flow graph can describe the command scheduling dependencies by means of *1) modeling each command as an actor and its execution time is set as the time spent on the command bus; 2) tracking the timing constraints by using delay (DL) actors, whose execution time are equal to the constants of the DDR3 JEDEC-specified timing constraints [7];*

3) capturing the command scheduling dependencies by adding edges between the actors of commands and timing constraints. For example, Fig. 7 illustrates an example of modeling a simple periodic schedule of an ACT, RD, and PRE to a bank (Fig. 7(a)) with a static data-flow graph (Fig. 7(b)) according to the above scheme.



(a) Cmd scheduling of a bank (b) The data-flow graph

Fig. 7. An example of data-flow modeling of commands to a bank.

A memory transaction is executed by interleaving it over BI banks, each of which requires an ACT, followed by BC times of RD or WR and finally a PRE according to a close-page policy. The command scheduling of a particular transaction in terms of specific type (read or write), BI, BC, and physical address (starting bank) can be modeled by a specific static data-flow graph, such as the simple example shown in Fig. 7. Therefore, different static data-flow graphs are needed to capture the command scheduling of transactions depending on type, size, and physical address. *MCDF captures the command scheduling dependencies of various transactions by specifying static mode sequences (SMSs).*

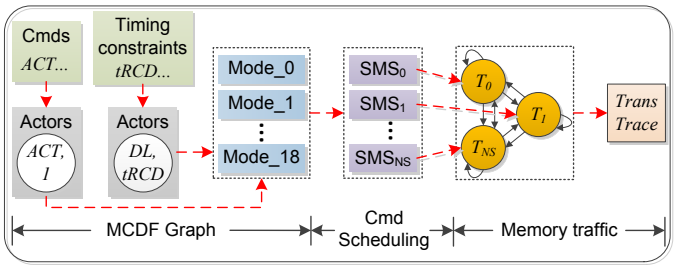


Fig. 8. An overview of the MCDF modeling of memory controllers.

Fig. 8 shows a high-level overview of the MCDF modeling of memory controllers. As aforementioned, memory commands and the SDRAM timing constraints are modeled by actors, which further constitute each individual mode (shown in Fig. 9). The scheduling of commands per transaction is captured by using a proper pre-computed static mode sequence (SMS) (presented in Section IV-C). Note that SMSs are computed for different kinds of transactions (T) in terms of type, sizes, and physical addresses. These SMSs are dynamically employed to model the memory traffic. A fully connected FSM can describe the transitions among all kinds of transactions associated with the SMSs (NS denotes the total number) if there is no apriori information about the traffic (discussed in Section V). Overall, the MCDF graph naturally captures commands and timing constraints of SDRAM and can be generally used by various memory controllers, which only need to compute their proper SMSs. When static knowledge about the traffic is provided, e.g., the known transaction sequence given by the application or by memory arbiter (e.g., TDM), it only requires to restrict the FSM to keep the known sequence. Therefore, the proposed MCDF model can be easily extended to different RT memory controllers.

B. MCDF Modeling of Command Scheduling

We proceed by generally modeling command scheduling of transactions with an MCDF graph. There are four different commands that are used to execute each transaction, *ACT*, *RD*, *WR*, and *PRE*. Note that the refresh (*REF*) command is not modeled explicitly because it is only needed for a large interval of tREFI and reduces the bandwidth by approximately 3% [16]. Its effect will be taken into account later when computing the worst-case bandwidth in Section V. The commands can be dynamically scheduled to the required banks according to various scheduling algorithms subject to the inter/intra-bank timing constraints. To generally support any command scheduling algorithm for transactions, *each per bank command can be modeled by creating a mode that has actors representing the command and the inter/intra-bank timing constraints.* In particular, the actors of inter-bank timing constraints are used to support the transitions across modes. Finally, *the execution of a transaction is modeled by using a mode sequence that specifies the order of modes corresponding to the required commands.* In this way, various command scheduling algorithms for transactions can be supported by specifying their mode sequences.

Fig. 9 shows the MCDF model of command scheduling. It consists of 18 modes, representing the scheduling of different memory commands to any of the 8 banks. Each mode consists of a command (*ACT*, *RD*, *AWR*, or *PRE*) actor and several delay (DL) actors that track the relevant timing constraints. The edges between actors in Fig. 9 show the dependencies. Since the command bus transfers one command per cycle, the execution time of all the command actors is 1 cycle except for *ACT* actors where it is 2 cycles. This is because an *ACT* commonly has lower priority than a *RD* or a *WR* as stated in Section III-A. We hence conservatively assume an *ACT* always collides with a *RD* or WR, resulting in one cycle additional delay. The execution time of DL actors in Fig. 9 are configured to be the values of the JEDEC timing constraints in Table I.

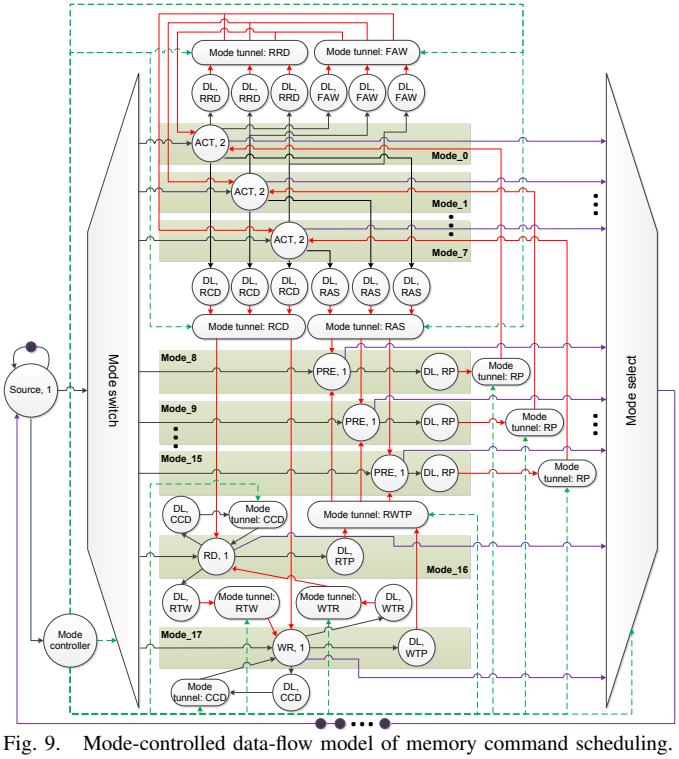


Fig. 9. Mode-controlled data-flow model of memory command scheduling.

The *ACT* and *PRE* commands to different banks have to be modeled by different modes because of the intra-bank timing constraints. For example, the scheduling of an *ACT* has to satisfy the *tRP* constraint from the previous *PRE* to the same

bank, as shown in Fig. 2. Mode_0 to Mode_7 in Fig. 9 model the *ACT* to a bank from Bank 0 to Bank 7, respectively. While Mode_8 to Mode_15 capture the *PRE* to a bank from Bank 0 to Bank 7, respectively. For $\forall i \in [0, 7]$, the transition between Mode_*i* and Mode_(*i*+8) captures the timing constraints (e.g., tRAS and tRP) between the *ACT* and *PRE* to the same bank. Note that the transition between modes is supported by tunnels. A basic tunnel is shown in Fig. 3 that only supports transition from one mode to another. The proposed MCDF model in Fig. 9 requires a general $M \times N$ tunnel that supports the transition from M modes to N modes, where $M > 0$ and $N > 0$. In the same way, the timing constraints between ACT commands are also captured, such as the *tRRD* and *tFAW*. Finally, *RD* or *WR* commands are sequentially scheduled, and it is not necessary to distinguish different banks. Therefore, *RD* and *WR* are modeled by Mode_16 and Mode_17, respectively, where all the relevant timing constraints are captured by tunnels. For example, the RCD Tunnel keeps the *tRCD* constraint from an ACT to a RD or WR, as shown in Fig. 9. The general $M \times N$ tunnel will be detailed later in Section IV-D.

The Source actor in Fig. 9 triggers the command scheduling each clock cycle by producing a token on the input port of the Mode switch, and its execution time is 1. This models the worst-case behavior, where pending transactions are backlogged, i.e., enough commands make the scheduler always busy. The Mode controller (MC) determines which mode to choose, i.e., which command to schedule, by specifying the mode sequence corresponding to a transaction. Therefore, when the Source actor produces a token to trigger a mode, it also gives a token to MC that produces a control token based on the mode sequence for all the switch, select, and tunnel actors to choose the mode. The translation from transactions to mode sequences will be detailed later in Section IV-C.

The memory controller schedules commands with limited speed due to the timing constraints. This is captured by a feedback edge from the Mode select to the Source actor (see Fig. 9), which makes the proposed MCDF model strongly connected. The token on this edge is produced by the Mode select that is triggered after the firing of each command actor per mode, and the token is then consumed by the Source to produce a new token to trigger the next command actor, i.e., schedule a new command. Note that the initial tokens on this edge must guarantee that commands are scheduled as soon as all timing constraints are satisfied. The proper number of the initial tokens will be obtained from experiments.

C. Mode Sequences

The MCDF model in Fig. 9 is able to capture the dependencies of different command scheduling mechanisms, e.g., close/open-page policy, bank privatization/interleaving, priorities. For a particular mechanism used by a memory controller to execute transactions, we only need to create the appropriate mode sequences that specify the firing order of modes, and hence the order of commands. To create a mode sequence of executing a transaction, we firstly create a mode sequence per bank, and then combine these per-bank mode sequences for the transaction according to its required banks.

Take the scheduling algorithm of dynamic command scheduling in [4] as an example. We show next how mode sequences are created for it. The dynamic command scheduling is discussed in Section III-A, where a transaction interleaves over BI banks and there are BC data bursts per bank. This requires commands to be scheduled to all BI banks, where each of them receives an *ACT*, followed by BC number of *RD* or *WR* commands and finally a *PRE* (see Fig. 2). Therefore, *the mode sequence for each bank must be an ACT mode, BC times of RD or WR mode and a PRE mode*. This is given by Definition 1 that defines the mode sequence *ms*(*k*, BC) for an arbitrary bank k ($\forall k \in [0, 7]$). As shown in Fig. 9, Mode_*k* captures the *ACT* command to bank k and the mode for the *PRE* command to the same bank is Mode_(*k*+8). The mode number for the BC number of *RD* or *WR* commands is Mode_16 or Mode_17, as given by Eq. (1).

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

$$RW_l = \begin{cases} \text{Mode\_16}, & RD\ command \\ \text{Mode\_17}, & WR\ command \end{cases} \quad (1)$$

For an arbitrary transaction T_i ($\forall i \geq 0$) that uses BI_i and BC_i, its corresponding mode sequence $MS(T_i)$ is given by Definition 2, which is a sequential combination of the BI_i number of mode sequences per bank.

Definition 2 (Mode sequence per transaction): For $\forall i \geq 0$ and $\forall j \in [0, BI_i - 1]$, $MS(T_i)$ = [*ms*(*bs*, BC_i), ..., *ms*(*bs*+*j*, BC_i), ..., *ms*(*bs*+$BI_i - 1$, BC_i)], where *bs* is the starting bank of T_i.

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

D. Tunnels

The tunnels of the MCDF model shown in Fig. 9 are used to support the transition between modes, and they need multiple data inputs and data outputs. As a result, the basic tunnel shown in Fig. 3 has to be extended, since it only has single data input and output. We generalize these tunnels to an $M \times N$ tunnel that has M data inputs and N data outputs, as depicted by Fig. 10. In addition, it consists of a single internal token. This generic tunnel is instantiated to support all the tunnels in Fig. 9 except the FAW tunnel that captures the tFAW constraint to restrict the scheduling of at most four *ACT* commands within the time window. A single internal token cannot support tFAW. The FAW tunnel is designed with a cascade structure of four internal tokens, as shown in Fig. 11.

1) Generic Tunnel: The generic tunnel presented in Fig. 10 consists of a switch and a select, and the edge between them has an initial token (i.e., internal state). The switch has 18 inputs while the select contains 18 outputs corresponding to all the 18 modes in the MCDF model. For an arbitrary input/output of the select/switch m ($\forall m \in [0, 17]$), the corresponding mode is Mode_*m* in Fig. 9. The tunnel is instantiated to support M data inputs and N data outputs, where $\forall M, N \in [1, 18]$. It also has one control input that delivers control tokens sent by the MC to the switch and select. The M inputs correspond to the modes from Mode_*i* to Mode_(*i*+*M*-1), while the N outputs are associated with Mode_*j* to Mode_(*j*+*N*-1), where $\forall i \in [0, 18 - M]$ and $\forall j \in [0, 18 - N]$.

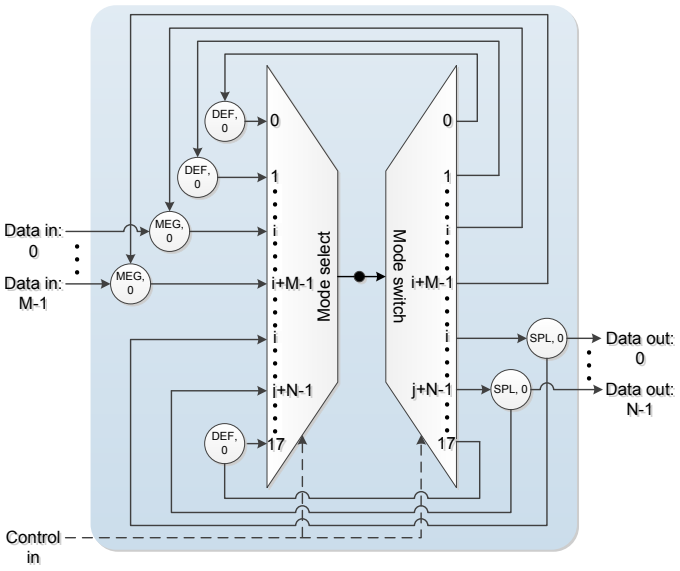


Fig. 10. A generic mode tunnel with M inputs and N outputs.

Each data input of the generic tunnel firstly connects to an actor with 0 execution time, called *merger* (MEG), and it consumes both the input data token and the internal token and produces the same data token. Note that the internal token is forwarded by the switch (see Fig. 10). The token produced by the MEG is consumed by the select when the control token indicates the mode corresponding to this data input, and the select produces the same token that becomes the new internal token. In this way, the internal state is updated. For a data output of the generic tunnel in Fig. 10, the output token is provided by an actor, namely *splitter* (SPL) that is connected by the output of the switch. The execution time of a SPL is 0. The output of the switch forwards the internal token to the SPL that produces the data output token and also returns it to the internal state via the select.

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

We proceed by introducing the connections of the M data inputs and the N data outputs. For an arbitrary data input k ($\forall k \in [0, M-1]$) with the corresponding Mode_(i+k), it connects to a MEG that further connects to the $(i+k)^{th}$ input of the select in Fig. 10. An output of the select connects to a SPL that connects to an arbitrary data output h ($\forall h \in [0, N-1]$) corresponding to Mode_(j+h). If $\exists h$ such that j+h = i+k (i.e., the same mode), an output of the SPL connects to the input of the MEG. Otherwise, the $(i+k)^{th}$ output of the switch connects to the input of the MEG. In addition, one of the outputs of the SPL goes back to the $(j+h)^{th}$ input of the select.

2) Cascade FAW Tunnel: The FAW tunnel in Fig. 9 captures the tFAW constraint (in Table I) that allows maximally 4 *ACT* commands to be scheduled within the rolling time window. It supports any transitions amongst Mode_0 to Mode_7. As a result, the FAW tunnel consists of 8 data inputs and 8 data outputs, which connect Mode_0 to Mode_7. Note that we cannot simply add four internal tokens to the generic tunnel in Fig. 10 to support tFAW. It is because all the four internal tokens may be consumed when the modes indicated by control tokens are not between Mode_0 to Mode_7. Therefore, when a control token for a mode between Mode_0 to Mode_7 arrives, it cannot be consumed by the select since the four internal tokens have already been consumed. To overcome this problem, a cascade tunnel with four pairs of select and switch is designed, as shown in Fig. 11, where the internal state of each pair contains an initial token. These four initial tokens allow at most 4 different *ACT* modes execute within the tFAW time window. When one of them is triggered, an initial token of the FAW tunnel is consumed by its *ACT* command actor and a new token will be produced by its DL actor with the execution time of tFAW. This new token goes to one of the data inputs of the FAW tunnel to update the internal state.

The execution of an *ACT* mode requires one internal token of the FAW tunnel. After tFAW cycles, it produces a token to update the internal state, such that new *ACT* mode can be triggered. The four initial tokens of the FAW tunnel are able to trigger four *ACT* modes, while the fifth one has to wait for an internal token that is updated by the first *ACT* mode after tFAW cycles. In this way, the rolling tFAW constraint is captured. When the FAW tunnel receives control tokens for modes from Mode 8 to Mode 17, they are consumed by each pair of switch and select through the default connection, i.e., the edge with a DEF actor, as shown in Fig. 11. Note that an internal token can be transferred to the next pair of select and switch or the data output of the FAW tunnel only if a control token for Mode 0 to Mode 7 is received. So, the firing of an *ACT* mode either gets an internal token or waits for the update of the internal state when the tFAW constraint is met.

V. Worst-Case Bandwidth

This section gives the definition of bandwidth provided by scheduling commands for transactions. The worst-case bandwidth is then analyzed by employing the MCDF analysis technique [20], as briefly introduced in Section III-B. Note that the MCM (tokens/second) is replaced by worst-case bandwidth (bytes/second) to define the critical cycle path of the MCDF graph.

A. Definition

The memory bandwidth determines how fast data can be transferred into/from SDRAM. It is determined by the memory controller that executes transactions by scheduling commands to the memory. The execution time caused by scheduling commands is highly dependent on the transaction type, size, and physical address. As a result, the bandwidth provided by a memory controller varies depending on the application(s). Definition 3 defines the bandwidth, where S is the transaction size and t_{ET} denotes the execution time of the transaction. In addition, e^{ref} is the refresh efficiency given by Eq. (2), which accounts for the cycles that are lost due to refreshing the memory array. t_{ref} is the time overhead caused by refreshing, and it includes the time for precharging all the banks after executing a transaction and completing the refresh itself. *tREFI* is the JEDEC-specified refresh interval as given in Table I. f_{mem} denotes the SDRAM frequency.

Definition 3 (Bandwidth): $bw = \frac{S}{t_{ET}} \times f_{mem} \times e^{ref}$

$$e^{ref} = 1 - \frac{t_{ref}}{tREFI} \tag{2}$$

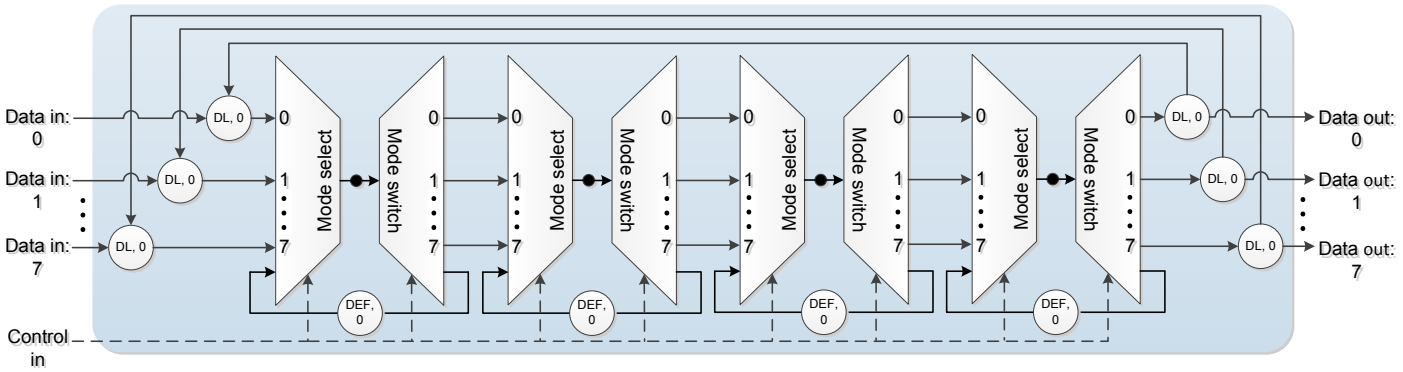


Fig. 11. A cascade tunnel structure to support multiple initial tokens for a specific set of modes.

The bandwidth given by Definition 3 varies according to the transaction sizes *S* and its execution time t_{ET}, while all other variables are determined by the particular SDRAM device. The worst-case bandwidth (WCBW) is provided when transactions suffer the longest execution time for a given size.

B. The Analysis of WCBW

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

Definition 2 previously showed the method to derive the mode sequence for a transaction, which requires information about the transaction type, size, and physical address. The type determines whether *RD* or *WR* commands are needed, and hence the corresponding Mode_16 or Mode_17 in Fig. 9. The size is mapped to *BI* and *BC*, while the physical address gives the starting bank (b_s). For example, when a system only generates 64 byte read and write transactions, e.g., the L2 cache line size is 64 bytes for all cores, the most efficient configuration of BI=4 and BC=1 is used to access a DDR3 SDRAM with a 16-bit data bus [17]. Since DDR3 SDRAMs have 8 banks, the transactions may either interleave consecutively over Bank 0 to Bank 3 or Bank 4 to Bank 7 for alignment reasons [17]. Therefore, four SMSs ([SMS_1 | SMS_2 | SMS_3 | SMS_4]*) are needed. Take a read transaction interleaving over Bank 0 to Bank 3 as an example. The SMS_1 is [Mode_0, Mode_16, Mode_8, Mode_1, Mode_16, Mode_9, Mode_2, Mode_16, Mode_10, Mode_3, Mode_16, Mode_11]. Similarly, the rest of the SMSs can be obtained. When a static transaction sequence is known, a larger SMS can be obtained by sequentially concatenating the SMS of each transaction with the sequence. The WCBW is hence analyzed based on the combined SMS that guarantees the static transaction sequence.

As introduced in Section III-B, the analysis of the MCDF model is performed by merging the equivalent static data-flow graphs of each SMS, resulting in a larger static data-flow graph that captures the dependencies of executing [SMS_1 | SMS_2 | ... | SMS_{NS}]*. The critical cycle defined by the MCM is obtained when executing the merged static data-flow graph, and it consists of a number of actors belonging to a single mode or different modes, which are specified in one or more SMS(s). As a result, these SMSs lead to the worst-case situation, i.e., the corresponding transactions experience a maximum average time (i.e., MCM) to execute each of them. Note that MCM is defined as the maximum of the total execution time of the critical cycle divided by the total number of initial tokens on the critical cycle. Hence, the minimum throughput (transactions per second) of the memory controller is 1/MCM. However, the minimum throughput is not always equivalent to the WCBW (bytes/second). For example, the critical cycle can be obtained based on large transactions that consume more time than small ones, but carry more data.

According to Definition 3, the bandwidth of the memory controller depends on the size of the transaction and its execution time. The critical cycle of the merged static data-flow graph must be defined as the cycle that provides WCBW rather than MCM. Similarly to the definition of MCM, the WCBW is defined by Eq. (3). For every cycle *C* of the MCDF graph *G*, the total execution time of the actors on *C* is denoted by $|C|$, while the total number of initial tokens (or delays) on the edges of *C* is $\omega(C)$. In addition, the total number of SMSs associated with *C* is $NS(C)$. Each SMS is used by a transaction and its size is S_i ($\forall i \in [1, NS(C)]$). However, it is not guaranteed that existing data-flow analysis algorithms can handle the WCBW defined by Eq. (3), since both $\omega(C)$ and S_i vary with *C*. As a result, we simply assume $\omega(C) = 1$, such that conservative WCBW can be provided by existing analysis algorithms.

$$WCBW = \min_{\forall C \in G} \frac{\omega(C) \times \sum_{i=1}^{NS(C)} S_i}{|C|} \times f_{mem} \times e^{ref} \qquad (3)$$

The WCBW given by Eq. (3) is a new notion for defining the critical cycle of the merged static data-flow graph to provide the WCBW. This critical cycle gives the order of executing the associated mode sequences that correspond to transactions. We can extract the worst-case order of transactions from the critical cycle, which can be used to design better scheduling algorithms to eliminate this bottleneck.

VI. Experimental Results

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

A. Experimental Setup

The proposed MCDF model has been verified and analyzed with Heracles [5], a temporal analysis tool developed at Ericsson. The transaction sizes used by the experiments include 16 bytes, 32 bytes, 64 bytes, 128 bytes, and 256 bytes. We have chosen the memory map configuration (i.e., *BI* and *BC*) for

each size that provides the lowest execution time (i.e., higher memory bandwidth) by interleaving over more banks to exploit bank parallelism. The configured (*BI*, *BC*) for these transaction sizes are hence (1, 1), (2, 1), (4, 1), (4, 2), and (4, 4) [17]. Note that (4, 2) and (4, 4) are used by 128 Byte and 256 Byte transactions instead of (8, 1) and (8, 2), because of *tFAW* that causes a larger execution time with *BI*=8. Experiments have been done with JEDEC-compliant, DDR3-800D, DDR3-1600G, and DDR3-2133K, all with interface widths of 16 bit and a capacity of 2 Gb [7].

B. Validation of Command Scheduling

This experiment validates that the proposed MCDF model conservatively captures the command scheduling behavior of a dynamically scheduled memory controller. This is achieved by comparing the scheduling time of each command obtained by executing the MCDF model to that given by an open-source scheduling tool [21] that implements the timing behavior of the memory controller in [4]. First, we have to find the proper number of initial tokens on the feedback edge of the MCDF model in Fig. 9. This is achieved by experimentally increasing the initial tokens until the feedback edge cannot dominate in any command scheduling. The results show that 20 initial tokens are enough for DDR3 SDRAMs and they are used by the following experiments. This experimental method is a quick, safe, and easy way to derive the proper number of initial tokens, such that the proposed MCDF model accurately captures the command scheduling of the memory controller.

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

C. Worst-Case Bandwidth

This section presents the WCBW given by the analysis of the MCDF model. The results are also compared to those given by the scheduled and analytical approaches of dynamic command scheduling in [4] and the semi-static approach in [6]. Those approaches compute the WCBW based on their WCET of transactions. Note that the collisions for *ACT* commands are actually detected by the scheduled approach, while collisions are always assumed by the analytical approach. The semi-static approach uses pre-computed command schedules with fixed lengths in cycles, and the WCBW is obtained based on them.

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

Fig. 12 gives the WCBW results obtained from the MCDF model. They are compared to that given by the scheduled, analytical, and semi-static approaches, respectively. We can observe that 1) the MCDF model always outperforms the analytical approach, where the maximum improvement is 22.0% for 64 byte transactions. This improvement is achieved because the MCDF analysis technique provides WCBW results without assumptions that are needed by the analytical approach. 2) It is also better than the scheduled approach with a single exception for 16 bytes, where it is 2.4% less. The reason is that the MCDF model conservatively assumes a collision per *ACT* command. 3) This exception also applies when comparing to the semi-static approach that statically resolves command collisions at design time. However, for large transaction sizes (e.g., 256 bytes), the MCDF model provides higher WCBW. This is because the semi-static approach uses pre-computed static command schedules, which pipelines *ACT* commands less efficiently. These observations also hold for other DDR3 SDRAMs, although the results are not shown for brevity.

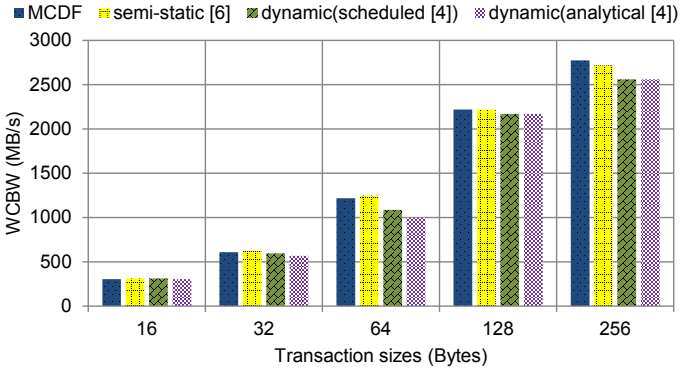


Fig. 12. The WCBW given by different analysis approaches for DDR3-1600G SDRAM with fixed transaction size.

2) Variable Transaction Sizes: The memory controller receives transactions with variable sizes, which are generated by different clients in a heterogeneous system, such as a High-Definition video and graphics processing system featuring a CPU, hardware accelerators and peripherals with variable transaction sizes [22]. If there is no static information about the transactions, e.g., the execution order of different transaction sizes, we have to conservatively analyze the WCBW results by assuming any possible transaction order. However, when clients with different transaction sizes are served by an arbiter using static schedules, such as time-division multiplexing (TDM) [23], the static order of transactions with variable sizes is known. Note that we assume a fixed transaction size per client. This static order of transaction sizes can be exploited to give less pessimistic (but still conservative) WCBW results.

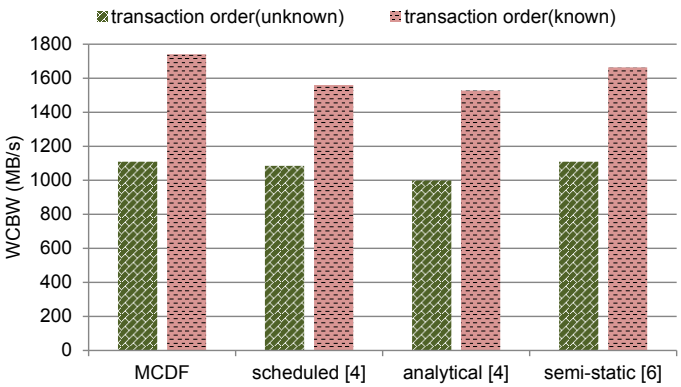


Fig. 13. The WCBW given by different analysis approaches for DDR3-1600G SDRAM with known/unknown static order of variable transaction sizes.

This experiment considers mixed transactions with sizes of 64 bytes and 128 bytes, which arrive at the memory controller with statically known or unknown order, respectively. The static order used in this experiment is that a 128 byte transaction is always followed by a 64 byte transaction and they are alternately executed by the memory controller. For instance, this can be enforced by a TDM arbiter. For unknown transaction order, transactions with these two sizes may be executed in any possible order. Fig. 13 shows the WCBW results for DDR3-1600G SDRAM given by different analysis approaches. We can see that the WCBW given by the MCDF model is always better than other approaches for both known and unknown transaction order. This indicates the MCDF model outperforms these existing approaches because the scheduled and analytical approaches use conservative assumptions while the semi-static approach cannot efficiently deal with variable transaction sizes. The maximum improvement is 14% compared to the analytical approach with known transaction order.

Another experiment compares the WCBW results with known and unknown transaction order for different DDR3 SDRAMs. The results are shown in Fig. 14, which demonstrate that much better WCBW is obtained by exploiting the static order of transactions. It achieves maximally 77.2% improvement of WCBW for DDR3-800D SDRAM by exploiting the static order of transactions.

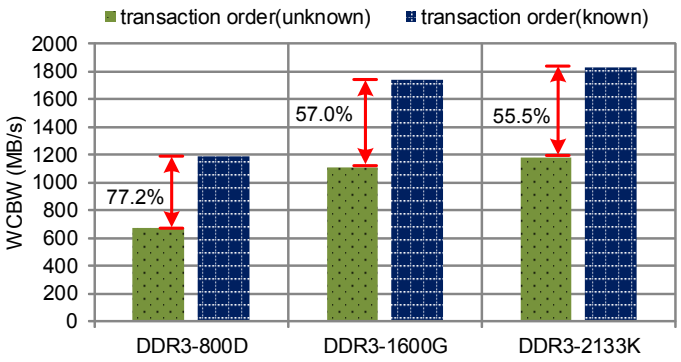


Fig. 14. The WCBW achieved by MCDF model for DDR3 SDRAMs with known/unknown static order of variable transaction sizes.

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

VII. Conclusion

The worst-case memory bandwidth is critical to satisfy the requirements of memory-intensive real-time streaming applications in modern multi-core systems. This paper proposes a new MCDF model to capture the scheduling dependencies of command scheduling, which is able to support different mechanisms of memory controllers. The worst-case bandwidth (WCBW) is analyzed using existing analysis technique, where a new notion of the critical cycle of the MCDF graph is introduced. The proposed MCDF model can easily exploit the static order of different transaction sizes, which provides (much) better WCBW results compared to the case with unknown order. It also provides information about how transactions are executed/scheduled in the worst case. Experiments have been carried out to validate the MCDF model and the results demonstrate that the MCDF model outperforms existing analysis approaches and achieves better WCBW.

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 EU FP7 288008 T-CREST, CA505 BENEFIC, CA703 OpenES, and 621353 DEWI.

References

[1] Y. Heechul *et al.*, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. RTAS*, 2013.

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

[3] A. Stevens, *"QoS for High-Performance and Power-Efficient HD Multimedia"*, ARM White paper, 2010.

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

[5] O. Moreira *et al.*, *Scheduling Real-Time Streaming Applications Onto an Embedded Multiprocessor*. Springer, 2013.

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

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

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

[9] N. Rafique *et al.*, "Effective management of DRAM bandwidth in multicore processors," in *Proc. PACT*, 2007.

[10] L. Ecco *et al.*, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *Proc. RTCSA*, 2014.

[11] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *Proc. RTAS*, 2015.

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

[13] A. Hansson *et al.*, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *Computers Digital Techniques, IET*, vol. 3, 2009.

[14] A. Lele *et al.*, "A new data flow analysis model for TDM," in *Proc. EMSOFT*, 2012.

[15] P. van der Wolf *et al.*, "SoC infrastructures for predictable system integration," in *Proc. DATE*, 2011.

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

[17] S. Goossens *et al.*, "Power/performance trade-offs in real-time SDRAM command scheduling," *Computers, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[18] M. Geilen *et al.*, "Synchronous dataflow scenarios," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, 2011.

[19] J. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Proc. SSC*, 1994.

[20] A. Lele *et al.*, "FP-scheduling for mode-controlled dataflow: A case study," in *Proc. DATE*, 2015.

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

[22] M. D. Gomony *et al.*, "A real-time multi-channel memory controller and optimal mapping of memory clients to memory channels," *ACM TECS*, vol. 14, no. 2, 2015.

[23] S. Goossens *et al.*, "A reconfigurable real-time SDRAM controller for mixed time-criticality systems," in *Proc. CODES+ISSS*, 2013.