A MILP Approach to DRAM Access Worst-Case Analysis

Matteo Andreozzi * Antonio Frangioni † Laura Galli † Giovanni Stea ‡
Raffaele Zippo ‡§

Draft 30th April 2021

Abstract

The Dynamic Random Access Memory (DRAM) is among the major points of contention in multi-core systems. We consider a challenging optimization problem arising in worst-case performance analysis of systems architectures: computing the *worst-case delay* (WCD) experienced when accessing the DRAM due to the interference of contending requests. The WCD is a crucial input for micro-architectural design of systems with reliable end-to-end performance guarantees, which is required in many applications, such as when strict real-time requirements must be imposed. The problem can be modeled as a mixed integer linear program (MILP), for which standard MILP software struggles to solve even small instances. Using a combination of upper and lower *scenario bounding*, we show how to solve realistic instances in a matter of few minutes. A novel ingredient of our approach, with respect to other WCD analysis techniques, is the possibility of computing the *exact* WCD rather than an upper bound, as well as providing the corresponding scenario, which represents a crucial information for future memory design improvements.

1 Introduction

With the advent of Industry 4.0, cyber-physical applications, such as self-driving cars, autonomous robots, etc., are expected to enjoy widespread diffusion. These applications have real-time constraints, and missing deadlines may result in harm to humans and/or damage to property. At the same time, systems architecture persist in the trend of resource sharing by multiple masters, for cost reasons. This makes it important to be able to bound the interference suffered because of resource sharing, so as to be able to provide worst-case performance bounds, chiefly a Worst-Case end-to-end Delay (WCD). Computing the WCD requires one to analyze single components in a system architecture (e.g., the shared interconnection network, the DRAM, etc.) and to provide *composable* worst-case guarantees for each of them. Network Calculus [1] is a framework for composable worst-case analysis, which is based on the concept of *service curve*, capturing the worst-case service obtained by a flow traversing a shared element. Composable worst-case guarantees allow one to perform end-to-end analysis, and end-to-end guarantees (e.g., between a sensor and an actuator) are the important parameters in a Service Level Agreement for critical services (see, e.g., [2], [3], [4]).

*ARM Ltd. Global Headquarters, 110 Fulbourn Road, CB1 9NJ, Cambridge, UK. E-mail: matteo.andreozzi@arm.com

†Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy. E-mail: {antonio.frangioni, laura.galli}@unipi.it

‡Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Largo Lucio Lazzarino 1, 56122 Pisa, Italy. E-mail: giovanni.stea@unipi.it, raffaele.zippo@phd.unipi.it

§Dipartimento di Ingegneria dell'Informazione, Università di Firenze, Via di S. Marta 3, 50139 Firenze, Italy. E-mail: raffaele.zippo@unifi.it

One of the main points of contention in a shared system architecture is the Dynamic Random Access Memory (DRAM). Modern DRAM is organized in a hierarchical structure, which is the result of a series of trade-offs between power consumption, access time, and throughput. The DRAM is managed by a DRAM controller, that serializes incoming requests (reads or writes) and issues commands to the memory chip(s) in order for these to be fulfilled. Common off-the-shelf (COTS) DRAM chips are often managed by a First-Ready-First-Come-First-Served (FR-FCFS) controller. The latter stores all reads and all writes, respectively, in two separate queues, and alternates between draining each queue, with an eye to minimizing the overhead of bus switching. While reads and writes are normally served FCFS, some of them may jump right at the front of the queue (so-called "hits") when doing so proves more efficient from a throughput perspective. In between serving requests, a controller has to periodically schedule *refresh operations* to preserve the memory content.

This paper analyzes the DRAM controller in order to find the worst-case delay that a read request undergoes. Read queues are in fact *on the critical path*, i.e., the master issuing them is blocked until they get a response (whereas a master does not need to block after a write request, which can therefore be delayed). We set out to compute that delay as a function of the position in the read queue of said request. The set of points (n, t_n), where t_n is the worst-case delay when the read queue consists of n requests, forms a service curve for the DRAM controller. We model the above worst-case problem as a mixed integer linear programming (MILP) problem.

We begin our analysis by modelling all the possible DRAM sequences of requests as "execution paths" in a *Finite State Machine* (FSM). The execution paths represent all possible schedules that may be executed starting from the DRAM initial state and reaching the state corresponding to the request under analysis being served. The FSM-digraph yields a "natural" MILP model for our problem.

However, standard MILP software is unable to solve even small instances of our problem in reasonable computing times. To circumvent this issue, we exploit our problem knowledge to compute lower and upper bounds on the worst-case delay, which we use to improve the running time by several orders of magnitude. This makes it possible to solve instances of realistic size to proven optimality in a matter of few minutes. Our approach allows DRAM vendors to compute provable worst-case guarantees for their products, and system engineers to use these guarantees in an end-to-end perspective, towards predictability of systems.

Other works in the past have dealt with the problem of worst-case analysis of DRAM access latencies. Some devise predictable-latency controllers or analyze existing, but simplified ones (see, e.g.,[5]), whereas others use knowledge of the task set run by the masters, to infer upper bounds on the worst-case delay (see, e.g., [6]). Our approach has some distinctive advantages over the latter. First, to the best of our knowledge, it is the first one to provide *exact* worst-case delays, rather than upper bounds. Second, it does so under looser hypotheses, i.e., without any assumption on the underlying task set. Third, it is the first one that reports guarantees in the composable form of a service curve, which allows our results to be used in conjunction with similar works for end-to-end analysis.

The paper is structured as follows. Section 2 discusses the related work. In Sections 3 and 4 we describe the system model and the corresponding FSM, respectively. In Section 5 we present a MILP model, while Section 6 is devoted to the scenario bounding technique. The results of some extensive computational experiments are given in Section 7. Some concluding remarks are made in Section 8.

2 Related Works

Several works in the literature have addressed memory command scheduling and the temporal analysis thereof, often in the context of real-time systems. Some of these works are concerned with designing new, predictable DRAM controllers. Predictability is enforced by imposing

constraints on the behaviour of the controller, e.g., about which master accesses which DRAM bank(s). A comparative evaluation of such works can be found in survey [5]. The authors of [7], for example, present a novel memory controller design, amenable to worst-case analysis, which capitalizes the open-row policy. This controller relies on statically partitioning the available main memory among possible requestors and disabling reordering of requests to avoid starvation.

Other studies, instead, analyse existing controllers, assuming that the tasks contending for memory access are known, and that they can be modeled according to standard frameworks used in real-time systems. For example, [6] describes the worst-case memory interference among a set of tasks scheduled according to fixed priorities, when the DRAM controller employs the FR-FCFS DRAM policy. Our analysis makes considerably fewer architectural assumptions on the tasks, their scheduling, their memory access request timing, the presence of caches, etc., thus being far more general. Moreover, the timing analysis in [6] does not consider memory refreshes, which are said to have the impact of a constant offset on the WCD. In our analysis, we show that refresh timers may instead interplay with the write arrival curve, resulting in non-constant increase to the WCD.

In [8, 9] the memory is modelled as a black-box system, where each memory request takes a constant service time; the scheduler is either Round-Robin (RR) or First-Come First-Serve (FCFS), which are different from those that are most often found in COTS system, i.e., FR-FCFS. In [8], CPU cores are allowed to issue multiple memory requests in parallel, giving a more realistic representation of modern COTS memory systems. The study evaluates the WCD suffered by a read request given that all the memory banks have their queues full, hence it considers both intra-bank and inter-bank interference delay. However, it does not allow one to derive the queueing delay of a request from the backlog.

The authors in [10] propose a method for characterizing the worst-case interference for a memory request in a FR-FCFS controller, which blends two different approaches, namely *job-driven* and *request-driven*. The first approach requires one to know the characteristics of the competing jobs and the number of outstanding requests, while the second one computes the worst-case for a single request, and then multiplies it by the number of outstanding requests. It is argued therein that either approach may yield a tighter bound, depending on the circumstances, hence the rationale in composing them into a unified framework.

Finally, [11] reports some initial thoughts about modelling the DRAM system as a FSM, as well as the idea of computing lower and upper bounds. However, no algorithm is explained, and no optimization method is described. The service curves obtained therein are computed using the upper bounds shown in Section 6, whereas the lower bounds are obtained via simulation, and are considerably worse than those described herein.

2.1 Our Contribution

To the best of our knowledge, this is the first study that computes *exact* (i.e., achievable) worst-case delays, rather than only upper bounds, using mathematical optimization techniques.

The main difference between our work and the ones discussed above is the level of *generality*. All the above works posit a precise system model and put several constraints on the set of tasks accessing the DRAM: at the very least a maximum number, or a per-task resource/period/deadline characterization. In some cases, a specific task scheduling algorithm is also envisaged. Our work does not require the above assumptions: the effect of competing traffic manifests itself via an overall maximum write bandwidth (i.e., the slope of the leaky-bucket characterization) and burst, and in the backlog that the request under study finds ahead in the read queue (as well as possible overtaking hits arriving later).

This also makes our work *composable*, whereas others are not. Indeed, the extra hypotheses required by the other studies only hold assuming that the stream of memory requests originated by the tasks verifies the stated constraints right at the arrival at the memory controller, i.e., if the network connecting the masters and the DRAM does not introduce any distortion due to

resource contention; in practice, it would need to have constant delay and infinite capacity. This is false in current systems, in particular within wormhole-switching networks where blocking effects introduce severe jitter. Conversely, our approach allows the memory controller to be modeled as a service curve, and therefore be composed with those obtained by modelling the underlying network elements (see, e.g. [1]), ultimately making it possible to perform end-to-end analysis. The need for end-to-end analysis of complex systems is made clear in [11]-[2].

We end up remarking that our work might seem to fail considering multiple banks, which would allow some parallelism; however, our assumption is easily justified by the worst-case setting, i.e., intra-bank interference being larger than, or equal to, inter-bank interference in all cases. Now, either one knows which task(s) access which bank(s), but this requires additional hypotheses on the task structure – which we steer clear of – or one does not, in which case the worst-case scenario is when all requests address the same bank, thus maximizing the interference of the one under study.

3 System Model and problem definition

In this Section we describe our assumptions to calculate the WCD and the main features of the DRAM system considered. As will become clear later, the order by which requests are served, as well as the numerical relations between timing parameters, can heavily impact on the WCD. Searching for the WCD scenario means evaluating all possible schedule combinations. This combinatorial aspect suggests to use mathematical optimization techniques to calculate DRAM WCD.

As already outlined in Section 1, there is a fundamental difference between reads and writes: read requests block the program that issues them until a response is received, hence are on the "critical path" and must be answered as soon as possible. On the other hand, writes are non-blocking, hence they can be deferred to maximize efficiency. This implies that our primary concern is the worst-case delay experienced by *read requests*. More precisely, we define the DRAM WCD problem as follows:

Definition 1 (DRAM WCD) *Given a queue of N read requests, we wish to calculate the WCD of the N^{th} read, a.k.a.* request under analysis*.*

In other words, assuming a backlog of $N-1$ read operations, and starting from the time t_0 of arrival of the N^{th} read request in the queue, our goal is to calculate the largest delay that the request under analysis can experience before being served by the DRAM system. Note that time is measured in system *clock cycles*.

The system under study, pictured in Figure 1, is composed of a DRAM controller and a DRAM device, consisting of one or more banks. Read and write requests arrive at the controller, which schedules them, together with periodic refreshes, according to its policy. The controller interfaces with the DRAM banks by issuing sequences of *commands* directed to them, each having specific timing constraints. We describe the controller first, and then the DRAM device.

3.1 DRAM controller

A FR-FCFS DRAM controller maintains one queue for reads and one for writes. Arriving reads (writes) usually queue FIFO in their respective queue, except if they target an *open row* of the DRAM bank, in which case serving them requires fewer commands and less time. Therefore, in the interest of efficiency, these requests (called read/write *hits*) are allowed to jump the queue – they are, in fact, "ready" to be served.

COTS FR-FCFS DRAM controllers employ a *watermark* policy to switch between reads and writes. As a default, the read queue is drained (since reads need faster response), and a controller only switches to serving writes when the backlog of the write queue is larger than

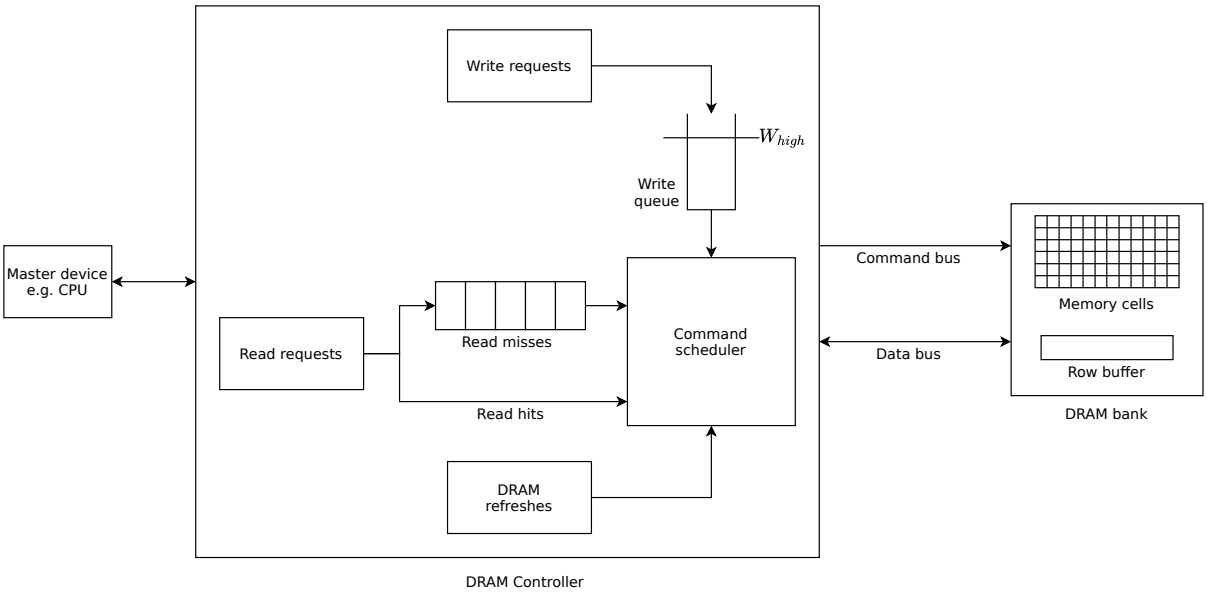

Figure 1: Overview of the System Model.

the watermark. This is because switching from one queue to another requires switching the direction on the data bus, which takes time and therefore reduces throughput.

Controller assumptions. We make the following assumptions on the DRAM controller:

1. *Open-page* **row policy**. When a request is completed, the corresponding row is left "open" in the row buffer. As we will see later, this means that subsequent requests targeting the same row (*row hits*) can be served faster. On the other hand, requests targeting a different row (*row misses*) will incur an additional latency, i.e., the time required to switch the row loaded in the row buffer. This choice makes FR-FCFS scheduling more efficient, but complicates the analysis.

2. *First-Ready First-Come First-Serve (FR-FCFS)* **arbitration policy**. Since row hits have strict priority over misses in the respective queue, we need a bound on the number of overtakes to avoid starvation and have a bounded WCD. We therefore assume that the controller allows a maximum of N_{cap} row hit *overtakes*.

3. *Watermark* **write-read switch policy**. When more than W_{high} write requests are backlogged, the controller switches to write requests and serves at least N_{wd} of them. Next, the controller switches back to read requests and keeps executing them until the watermark condition is triggered again. Note that this allows at least one read request to be served between two batches of writes, even if the arrival rate of write request is unbounded.

4. *Priority of* **refresh requests**. Refresh operations are supposed to occur periodically. However, read/write operations are atomic, hence cannot be interrupted. We consider two different policies that the controller can implement for scheduling refreshes, namely refresh *with* and *without* priority. The former forces the controller to schedule a refresh operation as soon as the refresh timer expires; the latter (aka *blind scheduling*), instead, schedules the refresh at any time after the timer expires.

5. *Single-bank* **(no parallelism)**. A single DRAM controller can control several DRAM banks, and commands to different banks can be issued in parallel. Schemes exist in the literature to assign banks to computing tasks, so as to minimize the interference of

different tasks (see Section 2). Clearly, multi-bank parallelism reduces the delays of queued requests. Since this work is concerned with finding the WCD, we assume that the DRAM controller manages a single bank, and do away with all the parallelism opportunities.

WCD assumptions. In order to simplify the problem, and without loss of generality, we make some further assumptions by considering conditions that would trivially reduce the WCD, and should therefore be excluded from our analysis. We assume that:

6. all N requests in the queue are targeted to the *same bank*, so that commands need to be *serialized*;
7. all the N read requests are *row misses*;
8. all write requests are misses, since write hits would cause less interference.

Write arrival rate assumptions. Because of the watermark-based queue switching policy, the WCD of a read request will be influenced by how many writes can be served before the N^{th} read. If the arrival rate of write requests is unbounded, we can easily envisage a scenario in which, periodically, a burst of N_{wd} writes follows a single read. Computing the WCD in this scenario is relatively easy (the alert reader can do it with pen and paper, without resorting to mathematical programming), but it may be pessimistic, for several reasons. First, masters do not send infinite batches of writes all the time; second, rate-limiters can be (and often are) employed at the entrance of a shared system (e.g., the interconnection network connecting the masters to the DRAM controller) to limit the amount of requests sent by a single (e.g., misbehaving) master; third, the interconnection network itself acts as a rate limiter, in that the write bandwidth of a master cannot exceed the network bandwidth along the path from that master to the controller. Therefore, the WCD computed assuming that the write queue is always above the watermark would be pessimistic. We can, however, capture the above three effects by assuming that the process feeding the write queue is *upper bounded*. A simple, yet effective way to model this is to state that the arrivals at the write queue are shaped by a token bucket shaper, shown in Figure 2, with arbitrary but known parameters burst b and rate r. The burst parameter b (the vertical offset) models the fact that a number of concurrent requests may arrive near-simultaneously (e.g., originating from different masters). The rate parameter r (the slope of the line) is the aggregate average rate of the masters that are using the DRAM. The fact that a process $Q(t)$ is upper bounded by a token bucket shaper with a shaping curve $\alpha(\tau) = b + r \cdot \tau,\ \tau > 0$, implies that $Q(t + \tau) \leqslant \alpha(\tau) + Q(t)\ \forall \tau$. In other words, the only legitimate processes are those that never intersect the shaping curve. Besides being a useful model for an aggregate traffic process, a token bucket shaper can be practically implemented in hardware (all it takes is a buffer and a timer). The token bucket constraint is called an *arrival curve* in Network Calculus. The above model allows us to represent physical limitations of the write arrival rate without requiring modelling of the tasks themselves. From now on, we denote with $\alpha_W(\tau)$ the token-bucket arrival curve that upper bounds the process that feeds the write queue.

Initial-state assumptions. Finally, the WCD is affected by the DRAM "status" at t_0, including the operation being served at t_0 and the size of the write backlog, for which we assume the following:

9. One cycle before the arrival of the request under analysis (i.e., at $t_0 - 1$) a read miss M_0 is issued. This means that said read miss is not counted in the backlog, but we still count in its maximum interference. We also assume that M_0 opens a row different from all other rows targeted by the reads in the backlog. As we show later on, this leads to a delay of tRC cycles before any command can be executed.

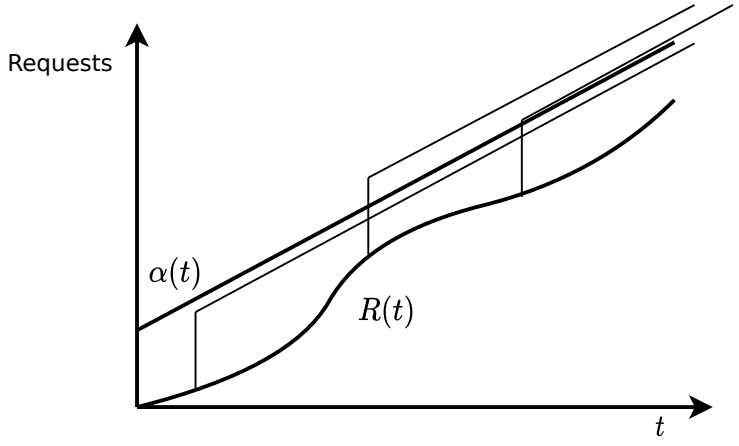

Figure 2: Example of token bucket shaper. The traffic process $R(t)$ is always below the arrival curve $\alpha(t)$ and its translations along $R(t)$.

10. We consider that the arrival process of write requests is bounded by an *arrival curve* $\alpha_W(t)$. Thus, the (initial) state of the write backlog affects the WCD. Therefore, we assume that when the N^{th} read arrives the write backlog has length $W_{high} - 1$, i.e., only one write request is required to trigger the first switch to serving writes. This is w.l.o.g.; indeed, if the WCD requires a batch to be served at the very beginning, then only one write request is needed, otherwise it is enough to have a zero arrival rate for a while.

3.2 DRAM commands and timing constraints

A DRAM device can be modeled as a 3-dimensional matrix of memory units organized in banks, rows and columns. As already stated, without loss of generality we focus on a single bank. A DRAM bank is a stateful entity, meaning that the operations (i.e., commands) allowed at any time depend on its state. At a first level of approximation, a DRAM bank can be modeled via the graph shown in Figure 3, where commands issued by the controller trigger transitions. This simplified view allows us to focus on high-level operations, without the added burden of timing parameters and constraints. These will be discussed later in this section.

The starting point is the *idle state*, where each cell of the bank is storing data and no operation is ongoing. When data needs to be read from a bank, an entire row needs to be *sensed* and *read* into an internal row buffer/sense amplifier. This process is called bank *activation*: the `ACT` command is issued to load the data and the row is *opened*. Once the row is open, `RD` (read) and `WR` (write) commands can be used on it. Before a different row can be read or written, the bitlines that enable sensing and reading data from the DRAM cells need to be *precharged*, i.e., the row must be *closed* with a `PRE` command. This command writes back the data to the bank cells, bringing the device back to its idle state. Memory access is subject to different latencies depending on the DRAM state:

- access to an *open row* (aka row hit) does not incur the precharge and activation latencies;
- access to a *closed row* (aka row miss with conflict) incurs both the precharge and the activation latencies, since that the controller uses the *open-row* policy, i.e., it leaves rows open as long as possible;
- access to a bank in the *idle* state incurs only the activation latency.

Therefore, a memory controller will favour schedules that maximize sequences of row hits. The *open row* policy of the controller means that the `PRE` command is delayed as much as possible after an operation, to leave the door open for future row hits. Periodically, the controller

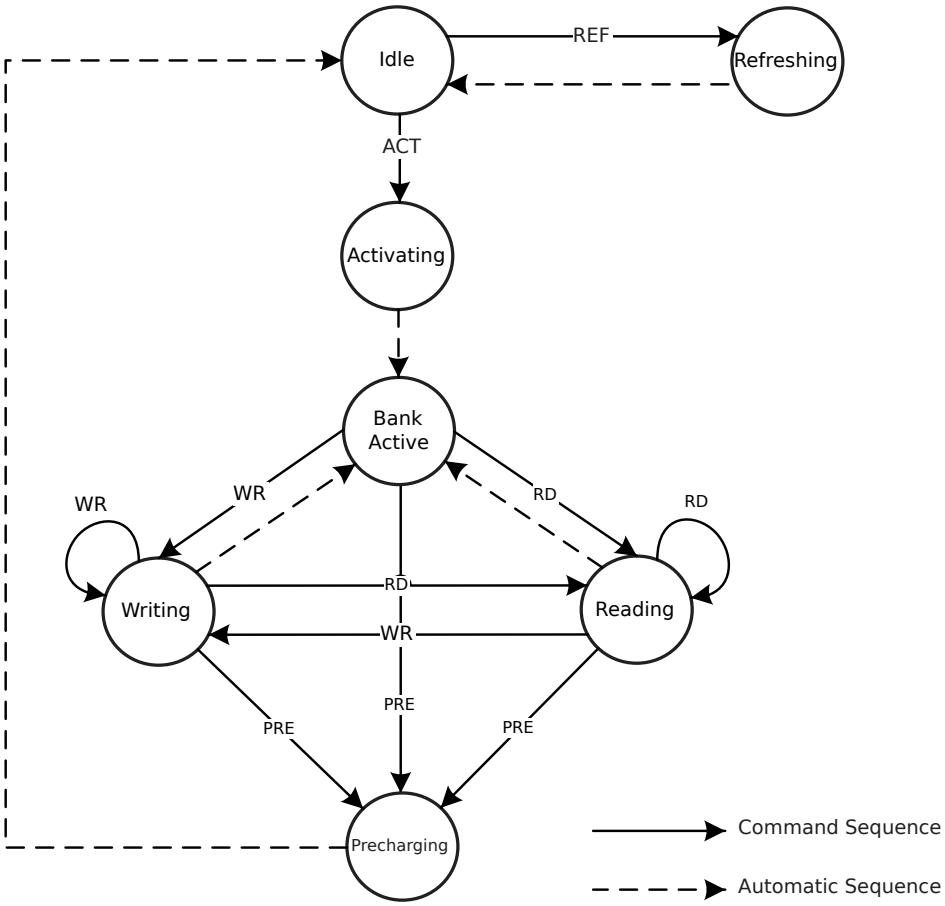

Figure 3: Graph model for a DRAM bank

schedules refresh (`REF`) commands. These can only be sent when the bank is in the idle state, i.e., no row is open.

The interface between the controller and the DRAM bank includes a *command* bus and a *data* bus. The two can transport information simultaneously. A DRAM bank has timing constraints on when each command can be received, that must be satisfied to ensure correct operation. These constraints are embodied in a set of timing parameter, which we need to describe hereafter. To do so, we follow the naming conventions of the Joint Electron Device Engineering Council (JEDEC) standards, and especially of those of the DDR4 DRAM. However, the model reported here also applies to other DRAM technologies (e.g., DDR3, LPDDR, etc.). The only difficulty for the interested reader is that, with a different technology, a timing parameter bearing the same name may have a slightly different definition, hence a tedious, yet straightforward re-mapping may be necessary. In the following, all times are reported as number of clock cycles, hence are integers.

We begin with the timing constraints related to single commands:

- `ACT` requires $tRCD$ cycles to complete (RCD stands from Row address to Column address Delay);
- `PRE` takes tRP cycles (RP stands for Row Precharge);
- both `RD` and `WR` require tCL (CL stands for Column access Latency);
- the `REF` takes $tRFC$ cycles (RFC stands for Refresh Cycle).

Note that row-level commands (`ACT` and `PRE`) are considerably more expensive than column-level commands (`RD` and `WR`), and the `REF` command is by far the most expensive. However, timing constraints can affect the *sequence* of commands. For instance:

- $tRAS$: Row Active Time, which is the minimum spacing between the `ACT` command and the successive `PRE` command: in today's architectures, $tRAS$ is usually between $tRCD + tCL$ and $tRCD + 2 \cdot tCL$;
- $tRTP$: Read to Precharge Delay, which is the minimum spacing between a `RD` and a `PRE` command.

Finally, we have timing constraints affecting the *data bus*, or start from the latter and affect the command bus:

- $tBurst$: The duration of the data transfer on the data bus for a single read/write command. As the `DDR` memories transfer data on both the positive and negative edges of the clock signal, in $tBurst$ clocks a DRAM transfers twice as many columns.
- $tWTR$: Write to Read command delay. This is the time it takes for the data bus to switch from the write direction to the read one, during which the bus is unusable.
- $tRTW$: Read to Write command delay. This is the time it takes for the data bus to switch from the read direction to the write one, during which the bus is unusable.
- tWR: Write Recovery Time. The minimum number of clocks between completion of a the write operation (i.e., end of the $tBurst$) and the next `PRE` command.

In this work, we assume that $tRAS > tRCD + \max\{tCL, tRTP\}$.
This assumption means that $tRAS$ is an active constraint in a command schedule, affecting the last read hit before a row is closed. This is exemplified in Figure 4. This assumption is verified by a large number of today's DRAM technologies, and allows us to simplify the analysis. The method described in the following can still be used even if this assumption is not met, via minor modifications which we leave to the interested reader.

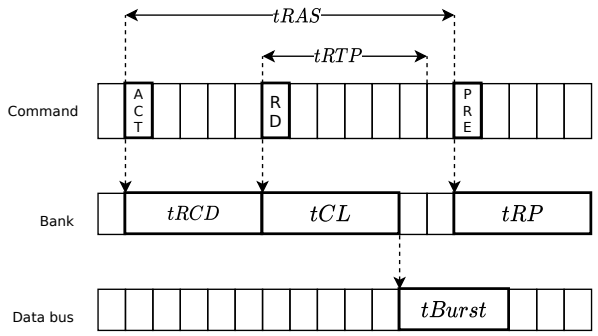

Figure 4: Example of the assumption on $tRAS$.

Timing constraints introduce additional delays to `ACT` and `PRE` commands duration. Let $tRC = tRP + tRAS$ be the *row cycle time*, i.e., the minimum time between two successive `ACT`s. Within that time interval, a "gap" (called *precharge bubble*) may appear between the completion of an operation and the first clock in which the `PRE` command can be issued again. During a precharge bubble, the bank can accommodate a read hit, but not read misses. Consider the example represented in Figure 5, where two different access patterns are depicted. Both schedules consider the timings of five read requests, which result in three row misses (M ⋄) and two row hits (H ⬥), but their *sequence* is different (MH-MH-M and MHH-M-M).

Pattern MHH-M-M (Figure 5 - *above*) This pattern has two row hits served back to back (⋄ ⬥ ⬥ ⋄). After the first miss, the two hits being back-to-back allows for maximum utilization of the data bus (if we ignore the small "gap" due to $tCL > tBurst$). As the two hits are served, when transitioning to the following miss, we only lose 1 clock cycle due to $tRTP$. Then, two consecutive misses are served; due to the $tRAS$ constraint, the `PRE` command of the second miss is delayed.

Pattern MH-MH-M (Figure 5 - *below*) Here the two row hits are split between the row misses (⬥ ⋄ ⬥ ⋄). This pattern executes *faster* than the previous one (as highlighted by the black arrow) due to the way each row hit "fits in" the previous row miss delay. Indeed, the two row hits execution does not add any additional delay because their tCL and $tRTP$ timings fall within the $tRAS$ time constraint of the previous row misses. Interestingly, their execution appears to be "for free", since serving pattern MH-MH-M takes exactly the same time as serving pattern MMM would.

The above example clearly shows that the same set of operations (read/write hits/misses) will have a different duration depending on their *sequence*. Indeed, the same operation can have a different impact depending on its position in the schedule (to facilitate the comparison, a black arrow marks the point at which the last miss request starts execution). This phenomenon affects the WCD.

Similar considerations apply to bubbles caused by write requests. In general, the parameters involved in write operations are analogous to those presented for reads, but their values can be different. This is the case of the parameter $tWTP$, which, analogously to $tRTP$, indicates the time between a `WR` command and the following `PRE`. Yet, the corresponding values are different. In particular, $tRCD + tWTP > tRAS$, which results in a longer delay to the next `PRE` for misses involving writes, instead of reads. For this reason, we also introduce $tRAS_W = \max\{tRAS, tRCD + tWTP\}$ and $tRC_W = tRP + tRAS_W$ to measure the *row cycle time* during a write batch, where $tRC_W \geqslant tRC$ typically holds. As already pointed out, we assume that all writes executed are misses and take an `ACT-PRE` cycle of length tRC_W each.

We denote by $tRPB = tRAS - (tRCD + tCL)$ the *read to precharge bubble* time, i.e., the delay after a read miss before another `PRE` command can be executed (during which a read hit

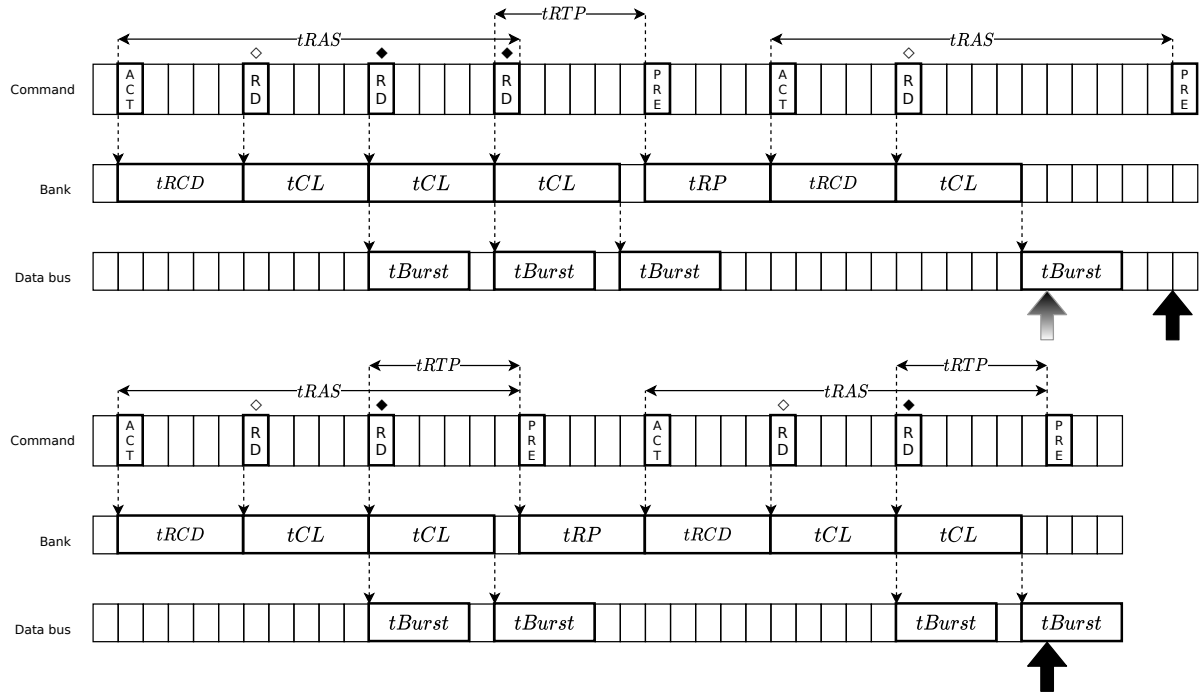

Figure 5: Timings of five reads with access pattern MHH-M-M (*above*) and MH-MH-M (*below*).

can be executed). Similarly, we denote by $tWPB = tRAS_W - (tRCD + tCL + tBurst + tWTR)$ the *write to precharge bubble* time, i.e., the delay after a write miss before another `PRE` command can be executed (during which a read hit can be executed).

The duration of a read/write precharge bubble influences the number of read hits that can fit into it. These hits, basically, come for free as far as delay computation is concerned. The number of read hits that fit *entirely* within a read precharge bubble is

$$N_{H^R}^{inner} = \left\lfloor \frac{tRPB - tRTP}{tCL} + 1 \right\rfloor \ .$$

After $N_{H^R}^{inner}$ read hits, some time may be still be left to the end of the read precharge bubble, but not enough to absorb another read hit. This means that another read hit scheduled right after the batch of $N_{H^R}^{inner}$ would come at a *reduced* cost with respect to tCL. We therefore define as N_{H^R} the number of read hits that can occupy the read precharge bubble, fully or partially. It is either $N_{H^R} = N_{H^R}^{inner}$, if $tRPB = N_{H^R}^{inner} \cdot tCL$, or $N_{H^R} = N_{H^R}^{inner} + 1$, otherwise.

Similarly, we define the same quantities for write precharge bubbles: the number of read hits fit *entirely* within a write precharge bubble, computed as

$$N_{H^W}^{inner} = \left\lfloor \frac{tWPB - tRTP}{tCL} + 1 \right\rfloor \ ,$$

and the number N_{H^W} of read hits that can occupy the write precharge bubble, fully or partially. Again, it is either $N_{H^W} = N_{H^W}^{inner}$, if $tWPB = N_{H^W}^{inner} \cdot tCL$, or $N_{H^W} = N_{H^W}^{inner} + 1$, otherwise.

Finally, we discuss the impact of refreshes and the scheduling of `REF` commands. Refresh requests queue up every $tREFI$, which is orders of magnitudes longer that the other timing parameters described so far. While this reduces the impact of refreshes on the *average* delay, it does not mean that refreshes can be neglected when computing the WCD. It is important to remark that, while refresh operations cannot be arbitrarily spaced (without incurring the risk of losing data), some jitter with respect to a perfectly periodic scheduling can be tolerated.

For this reason, we consider two refresh policies: *priority* and *blind* scheduling. With priority scheduling, the controller schedules a pending refresh request right after the ongoing (read or write) operation, to minimize the jitter. With blind scheduling, we assume that the controller does schedule all the pending refresh requests, but make no hypotheses as to when exactly. The rationale behind this is that, by removing a constraint, it leads to higher WCDs.

4 FSM Model

As already pointed out, the number of cycles required to execute a given DRAM schedule depends on both the required commands and their sequence. This combinatorial aspect implies that computing the WCD requires to (hopefully, implicitly) enumerate *all* feasible schedules to find the one maximizing the corresponding delay. In this section, we show how to model the DRAM schedules as *execution paths* in a FSM, see Figure 6. Recall that our WCD is the delay of the N^{th} read. Therefore, we need to describe all possible schedules that can be executed from the initial state S (i.e., marking the arrival of the N^{th} read in the queue) to a final state M_N (i.e., the N^{th} read is served).

A state is a description of the status of the DRAM system that is waiting to execute a transition. A transition is a set of actions to be executed when an event is received (i.e., a command from the memory controller). More precisely, the DRAM FSM is a weighted digraph $G^{\texttt{FSM}} = (V, A)$, in which a *state* (node) $X \in V$ corresponds to the completion of command X, and a *transition* (arc) $(X, Y) \in A$ models the execution of command Y after command X. The *cost* of the transition c_{XY} represents the number of cycles required to complete command Y after X, up to the point when the DRAM is ready to receive another command. In other words, each transition $X \rightarrow Y$ models the execution of command Y until the system reaches state Y corresponding to:

- expiration of time $tBurst$, if command Y is the N^{th} read miss (i.e., up to completion of the request under analysis);
- expiration of time tCL, if command Y is a read hit (i.e., up to the point when another read hit can be executed);
- expiration of any timing constraint that delays the execution of the following command (i.e., when a `PRE` can be executed), in all the other cases.

4.1 States

The FSM graph $G^{\texttt{FSM}}$ models all possible DRAM system evolutions as execution paths from the arrival of the N^{th} read in the queue (origin node S), to the N^{th} read being served (destination node M_N). The graph has a different node for each read hit, read miss, write miss and refresh operation.

S, M_0: these two *initial* states include a dummy source state S, and the read miss M_0 executing before the arrival of the request under analysis.

M_i $(i = 1 \dots N)$: states of type M represent the N *read misses* that have to be served. In particular, M_N is the destination state and we are not interested in what happens after M_N.

H_i^R $(i = 1 \dots N_{H^R})$: states of type H^R correspond to *read hits* within read precharge bubbles. Up to $N_{H^R}^{inner}$ can fit entirely into the precharge bubble at a null cost. If $N_{H^R} = N_{H^R}^{inner} + 1$, then another read hit can be included at a cost $< tCL$.

H_i^W $(i = 1 \dots N_{HW})$**:** states of type H^W correspond to *read hits* within write precharge bubbles. Up to $N_{H^W}^{inner}$ can fit entirely into the precharge bubble at a null cost. If $N_{H^W} = N_{H^W}^{inner} + 1$, then another read miss can be included at a cost $< tCL$.

H_i $(i = N_{hl} \dots N_{cap})$**:** states of type H correspond to ordinary (i.e., out-bubble) *read hits* with transition cost tCL. The number of H nodes depends on the N_{cap} parameter, which also includes in-bubble hits of type H^R and H^W. The lowest index that an out-bubble read hit can take in a feasible schedule is $N_{hl} = \min\{N_{H^R}, N_{H^W}\} + 1$.

W_i $(i = 1 \dots N_W)$**:** states of type W correspond to *write batches*. Each write batch consists N_{wd} write misses back to back. The scheduling of subsequent write batches is regulated by the length of the write queue, which is fed by an arrival process that can be bounded by the write arrival curve α_w. We assume knowledge of an upper bound N_W on the total number of W states that can be reached before M_N; in Section 6 we discuss how to compute N_W in an efficient way.

R_i $(i = 1 \dots N_R)$**:** states of type R correspond to *refresh cycles*. The execution of a refresh cycle costs $tRFC$, and the distance between two refresh cycles is regulated by the refresh period $tREFI$. We assume knowledge of an upper bound N_R on the total number of R states that can be reached before M_N; in Section 6 we will show how to compute N_R in an efficient way.

Note that, on one hand, we need to distinguish read hits between those that are served within a precharge bubble (i.e., H^R and H^W nodes) from ordinary hits (i.e., H nodes), since the corresponding transitions have different costs. On the other hand, from the point of view of the DRAM controller, in-bubble and out-bubble read hits are indistinguishable and the cap N_{cap} is enforced on all of them. As a consequence, the indexing of out-bubble hits depends on the number of in-bubble hits as well as on the cap: the lowest index N_{hl} represents the first hit that can be executed outside a precharge bubble, while the highest index N_{cap} is the last read hit allowed (including the ones that were previously executed inside a precharge bubble).

All the parameters that regulate the number of states for each type of operation (and their exact calculation) are described in the Appendix.

4.2 Transitions

We now describe in detail the possible FSM state transitions (i.e., arcs of the $G^{\texttt{FSM}}$ graph) and the corresponding costs. To this end, we recall the distinction between hit and non-hit states in our model, that is:

- a hit state represent the execution of a read hit, up to the expiration of timing tCL;
- a non-hit state represents the execution of a read miss, a write batch or a refresh, up to the expiration of any timing constraint that delays the execution of the following command (i.e., when a `PRE` can be executed).

We will use X to generically refer to a state of the latter category.

$S \to M_0$**:** the first (mandatory) FSM transition represents the initial delay $tRC - 1$ induced by the read miss M_0, whose execution is assumed to start one cycle before the arrival of the request under analysis.

$X \to M_i$ $(0 < i < N)$**:** this type of transition represents the execution of a *read miss* in the queue (excluding the request under analysis). The transition cost is $tRP + tRAS = tRC$ and requires:

- closing the currently open row with a `PRE` command;

- opening another row and executing a `RD` command;
- waiting until the row can be closed again.

$X \to M_N$**:** with this last FSM transition we consider the command M_N to be completed when the data transfer along the bus is finished. Therefore the cost is $tM_N = tRP + tRCD + tCL + tBurst$.

$X \to W_i$**:** executing a *write batch* W_i means executing N_{wd} write misses. Each write miss behaves like a read miss, except from the length tRC_W, which may be larger than tRC, according to the dominance between $tRAS$ and $tWTP$. Similarly to read misses, we consider the execution to finish when a `PRE` can be issued, thus also the last write takes tRC_W cycles. Therefore, the cost of such transition is $L_W = N_{wd}\, tRC_W$. Note that X cannot be a node of type W due to the write-read switch policy that forces write batches to alternate with read requests.

$X \to R_i$**:** the length of a *refresh cycle* is specified by the $tRFC$ parameter.

$X_1 \to \bar{H}_i \to X_2$**:** special attention must be paid when considering the cost of *read hits*, because it depends on the effect of precharge bubbles (i.e., $\bar{H}_i$ can be of type H, H^R or H^W):

- If the read hit is *inside* a precharge bubble ($\bar{H}_i$ is either of type H^R or H^W), the cost of transition $X_1 \to \bar{H}_i$ is already contained in the "wait-for-precharge" delay accounted for by state X_1. Thus, the first transition $X_1 \to \bar{H}_i$ has 0 cost. Also, if state X_2 needs precharge, the waiting cost for the `PRE` command to be issued has already been considered in X_1, so no adjustment is needed to the usual cost of X_2. For instance, the transition $\bar{H}_i \to M_j$, where $\bar{H}_i$ is a read hit inside a precharge bubble and M_j is a read miss, costs tRC. Note that X_2 can be another red hit inside the bubble (if N_{H^R} and $N_{H^W} > 1$), in which case the cost of the second transition is again 0.
- If the read hit is *outside* the precharge bubble ($\bar{H}_i$ is of type H), then its cost is larger than 0. For the first transition $X_1 \to H_i$, we consider the cost to be up to the point when another read hit (`RD` command) can be executed, so the cost is tCL. For the second transition $H_i \to X_2$, the cost depends on the state X_2:
  - if X_2 is another hit, then the `RD` command can be immediately executed and the cost is again tCL;
  - if X_2 is not a hit, then a `PRE` command is needed, therefore a delay of $tRTP - tCL$ must be added to the usual cost of X_2. For instance, $H_i \to M_j$, where H_i is a read hit outside the precharge bubble and M_j is a read miss, costs $tRTP - tCL + tRC$.
- If the read hit is *partially* contained in a precharge bubble, the cost of the first transition should account only for the part outside the bubble, namely:
  - if X_1 is a read miss, the cost is $tCL - (tRPB - N^{inner}_{H^R} \cdot tCL)$;
  - if X_1 is a write miss, the cost is $tCL - (tWPB - N^{inner}_{H^W} \cdot tCL)$.

  The second transition $\bar{H}_i \to X_2$ follows the rules already presented for the case of read hits outside the precharge bubble.

4.3 Execution paths

A *feasible* FSM execution path p has the following properties:

- p is a *simple* path in $G^{\texttt{FSM}}$. Since in a feasible schedule, bubble hits H^R and H^W can follow any state of type M and W, respectively, a path might need to visit the same in-bubble read hit nodes multiple times (i.e., once after each state of type M and W).

To ensure that all feasible paths are simple, we replicate H^R and H^W nodes for each read miss and write batch state, respectively. For instance, $M_1 \rightarrow H_1^R$ and $M_2 \rightarrow H_1^R$ becomes $M_1 \rightarrow H_{1,1}^R$ and $M_2 \rightarrow H_{1,2}^R$, where node H_1^R is removed and substituted by two nodes $H_{1,1}^R$ and $H_{1,2}^R$, one copy for each miss operation M_1 and M_2, respectively. Note, however, that the resulting graph $G^{\texttt{FSM}}$ is *not acyclic*, thus additional MILP constraints will be needed to avoid cycles, as shown in Section 5.

- The "cap" constraint requires p to visit *at most* N_{cap} read hits nodes.
- In a feasible path p, a node X_i can only be visited *if the previous* node of the same type X_{i-1} has already been visited. This guarantees that visits to states of the same class are "naturally ordered".
- In a feasible schedule, refresh operations must be performed to preserve DRAM memory bank information. We consider a refresh timer with fixed period $tREFI$ and two policies: *blind scheduling* and *priority scheduling*. The timing of the R nodes visited in a feasible path p must meet the scheduling policy considered.
- A feasible path p must guarantee the correct spacing between write batches, depending on the assumptions on the arrival curve α_W that bounds the write arrival process.

The resulting $G^{\texttt{FSM}}$ graph can be used to solve the DRAM WCD problem by looking for the *longest feasible* (simple) (S, M_N) path in it. To better understand the construction of the $G^{\texttt{FSM}}$ graph, we use the FSM DRAM example, represented in Figure 6.

Example 1 *Figure 6 represents a $G^{\texttt{FSM}}$ graph corresponding to a DRAM FSM with the following parameters:*

- $N = 2$ *(number of* read misses *in the queue);*
- $N_{cap} = 3$ *(*max *number of* read hits *that can be served);*
- $N_{H^R}^{inner} = 1$, $N_{H^W}^{inner} = 0$*;*
- $N_{H^R} = 2$, $N_{H^W} = 1$*;*
- $N_W = 5$ *(*max *number of* write batches*);*
- $N_R = 1$ *(*max *number of* refresh cycles*).*

M_2 is the final state (i.e., the request under analysis). Note that in this example the last in-bubble read hits are only partially *included in the corresponding bubbles: $N_{H^R} = N_{H^R}^{inner} + 1$ and $N_{H^W} = N_{H^W}^{inner} + 1$. Also, note that each in-bubble read hit node H_i^R, H_i^W is* replicated*, respectively, for each read miss M_j and write batch W_j operation: $H_{i,j}^R$, $H_{i,j}^R$ (as explained above).*

5 MILP Model

In this Section we derive a MILP model for the DRAM WCD problem starting from the FSM digraph $G^{\texttt{FSM}} = (V, A)$ described in the previous section.

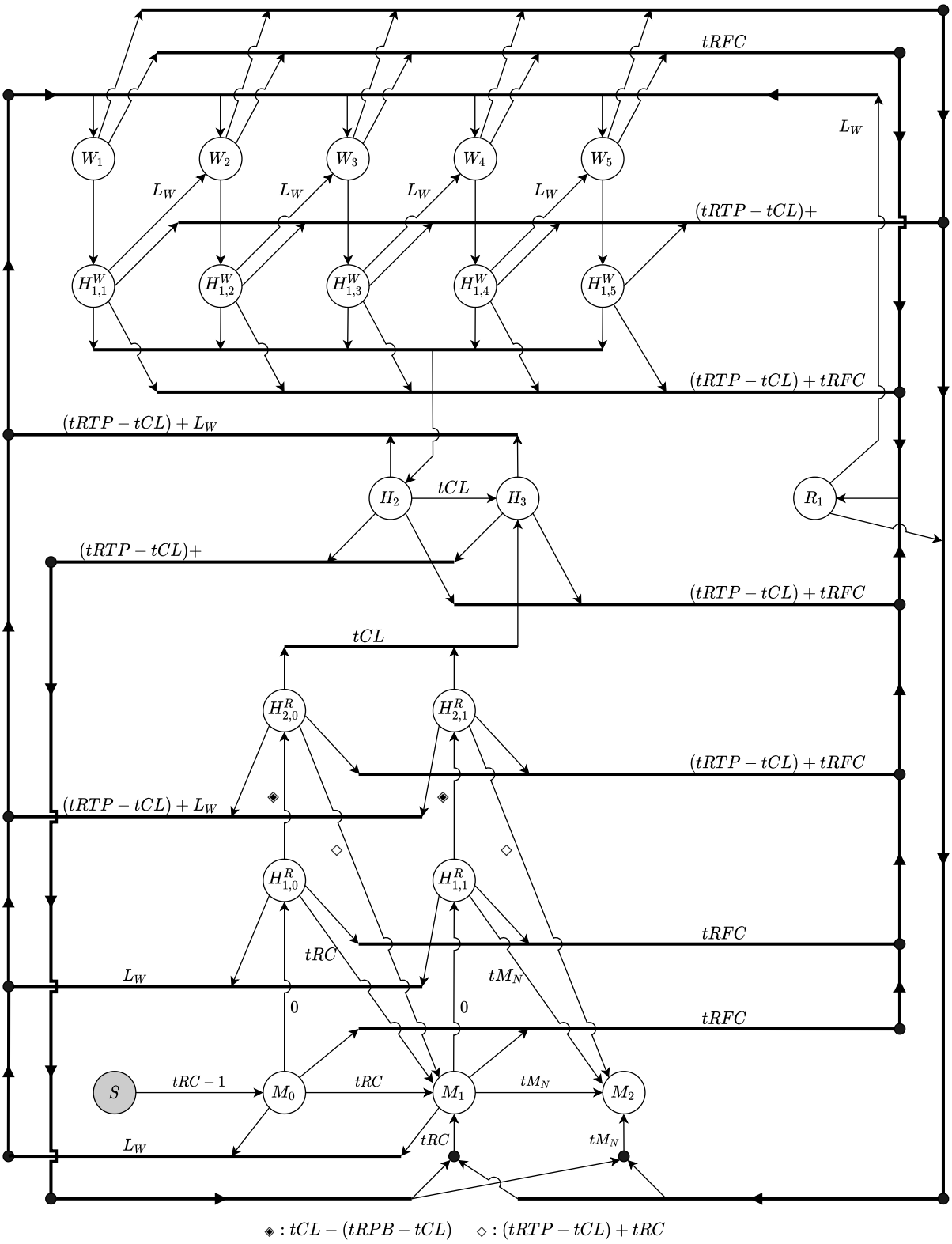

Figure 6: An example of FSM with $N = 2$, $N_{cap} = 3$, $N_{H^R} = 2$, $N_{H^W} = 1$.

5.1 Variables and objective

We start by introducing two types of variables:

- Binary variables x_{ij} for $(i,j) \in A$ to select the transition arcs in the $G^{\texttt{FSM}}$ graph, i.e., to define a (S, M_N) path.
- Continuous variables $t_i \geqslant 0$ for $i \in V$ to count the clock cycles required to execute a given schedule, i.e., to compute the cost of the selected (S, M_S) path. More precisely, variable t_i associated to (state) node $i \in V$ represents the (cycle) time at which command i is executed in the schedule corresponding to the path selected by the x_{ij} variables. Since time is measured in CPU cycles, t variables are bound to be integer.

Determining the WCD corresponds to calculating the *longest simple* (feasible) (S, M_N) *path* in $G^{\texttt{FSM}}$. Note that the length of the path is given by the value of the t_{M_N} variable, i.e., the (cycle) time at which the last FSM state M_N is executed. Hence, we look for a simple (feasible) (S, M_N) path maximizing t_{M_N}, the objective is just:

$$\max\ t_{M_N} \tag{1}$$

While these variables form the "backbone" of our model, some other auxiliary variables will have to be introduced later on in the context of modelling certain specific conditions.

5.2 Constraints

Flow conservation: To describe a path in $G^{\texttt{FSM}}$ from origin S to destination M_N we use the standard *flow conservation* constraints:

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji} = \begin{cases} +1 & \text{if } i = S \\ -1 & \text{if } i = M_N \\ 0 & \text{otherwise} \end{cases} \qquad i \in V \tag{2}$$

where $\delta^+(i)$ and $\delta^+(i)$ denote the set of outgoing and incoming arcs of node i.

Time propagation: Assuming $t_S = t_0 = 0$, to represent the correct execution cycle time for each node in the path requires enforceing *time propagation* constraints $x_{ij} = 1 \implies t_j = t_i + c_{ij}$:

$$t_i + c_{ij} - (1 - x_{ij})\,\Delta \leqslant t_j \leqslant t_i + c_{ij} + (1 - x_{ij})\,\Delta \qquad (i,j) \in A \tag{3}$$

where Δ represents a sufficiently large coefficient of a standard "big-M" formulation, whose calculation is discussed in Section 6. Note that, by prohibiting cycles, these constraints also guarantee the selected path to be *simple*.

Hit overtakes: We assume at most N_{cap} row hit overtakes to take place in a feasible schedule. Denoting by $\bar{H}$ the set of all hit states in the graph, this constraint can be expressed as *visiting at most* N_{cap} hit nodes in a path:

$$\textstyle\sum_{i\in\bar{H}}\ \sum_{a\in\delta^-(i)} x_a \leqslant N_{cap}. \tag{4}$$

State sequence: A transition to node X_i is allowed only if the previous node of the same type X_{i-1} has been visited in the sequence. In some cases, this is guaranteed by the graph structure, while in other cases additional constraints are needed:

- States of type H^R and H^W form "chains" of nodes either attached to a read miss or a write miss. In these chains, a transition to a node of index i can either start from a node of index $i-1$ of the same type or from the miss node that triggered the bubble hits. Thus, the ordering is guaranteed by the graph structure.
- States of type H, M, W and R, instead, need additional constraints:

$$t_{H_i} \geqslant t_{H_{i-1}} \qquad i = N_{hl}+1,\dots,N_{cap} \tag{5}$$
$$t_{M_i} \geqslant t_{M_{i-1}} \qquad i = 2,\dots,N \tag{6}$$
$$t_{W_i} \geqslant t_{W_{i-1}} \qquad i = 2,\dots,N_W \tag{7}$$
$$t_{R_i} \geqslant t_{R_{i-1}} \qquad i = 2,\dots,N_R \tag{8}$$
$$\textstyle\sum_{a\in\delta^-(H_i)} x_a \geqslant \sum_{a\in\delta^-(H_{i-1})} x_a \qquad i = N_{hl}+1,\dots,N_{cap} \tag{9}$$
$$\textstyle\sum_{a\in\delta^-(M_i)} x_a \geqslant \sum_{a\in\delta^-(M_{i-1})} x_a \qquad i = 2,\dots,N \tag{10}$$
$$\textstyle\sum_{a\in\delta^-(W_i)} x_a \geqslant \sum_{a\in\delta^-(W_{i-1})} x_a \qquad i = 2,\dots,N_W \tag{11}$$
$$\textstyle\sum_{a\in\delta^-(R_i)} x_a \geqslant \sum_{a\in\delta^-(R_{i-1})} x_a \qquad i = 2,\dots,N_R \tag{12}$$

  The combination of constraints (5)–(8) and (9)–(12) enforces a FSM path to visit each node type in sequence.

Refresh scheduling: We consider a refresh timer with fixed period $tREFI$ represented by an integer variable $t^{Timer} \in \{0,\dots,tREFI-1\}$, and we need to ensure that the timing of the R nodes in the path meets the refresh scheduling policy rules considered. Denoting by $t^Q_{R_i} = t^{Timer} + (i-1)\,tREFI$ the cycle time at which the i-th refresh request is enqueued:

- *Blind scheduling* allows to execute a `REF` command any time after the corresponding refresh request has arrived, which corresponds to adding the constraints

$$t_{R_i} \geqslant t^Q_{R_i} \qquad i = 1,\dots,N_R. \tag{13}$$

- *Priority Scheduling* forces a `REF` command after the ongoing operation terminates. For each transition (i.e., arc in the graph) and for each R node we need to ensure that the following logical condition is satisfied:

$$\left.\begin{array}{l} x_{X,Y} = 1 \\ t_X < t^Q_{R_i} \\ t_Y \geqslant t^Q_{R_i} \end{array}\right\} \implies x_{Y,R_i} = 1$$

  Introducing auxiliary indicator variables y'_{X,Y,R_i} and y''_{X,Y,R_i}, one for each arc $a = (X,Y) \in A$ and node R_i, $i \in \{1,\dots N_R\}$, the condition can be expressed by

$$\begin{array}{l} y'_{X,Y,R_i} \geqslant (t^Q_{R_i} - t_X - 1)/\Delta \\ y''_{X,Y,R_i} \geqslant (t_Y - t^Q_{R_i})/\Delta \\ x_{Y,R_i} \geqslant x_{X,Y} + y'_{X,Y,R_i} + y''_{X,Y,R_i} - 2 \end{array} \tag{14}$$

  where Δ, again, represents a sufficiently large coefficient of a standard "big-M" formulation, whose calculation is discussed in Section 6.

Write batch spacing: The controller is expected to schedule a write batch when the corresponding backlog size reaches the watermark level W_{high}. The execution of a write batch removes N_{wd} requests from the backlog. We consider both the unbounded case, and that of a (bounded) arrival curve $\alpha_W(t)$ for the write batch requests.

The *unbounded* case assumes that, whenever the controller checks for the next request to be scheduled, the write queue is always above the watermark level W_{high}. Since the bus

direction is switched after any write batch, a write batch operation cannot follow another write batch operation. On one hand, this is guaranteed by the fact that transition arcs $W_i \rightarrow W_{i+1}$ are not allowed in the $G^{\texttt{FSM}}$ graph. On the other hand, we need additional constraints to avoid the sequence $W_i \rightarrow R \rightarrow W_{i+1}$:

$$x_{W_i,R_j} + x_{R_j,W_{i+1}} \leqslant 1 \qquad\qquad i = 1, \ldots, N_W - 1,\ j = 1, \ldots, N_R \tag{15}$$

The *bounded* case assumes knowledge of an *arrival curve* $\alpha_W(t)$ and a backlog of size $W_{high} - 1$ at t_0. Let the variable $t^Q_{W_i}$ be the *enqueue* time of the i-th write batch W_i, i.e., the time the watermark is reached. Once enqueued, a write-batch may have to wait before being scheduled for different reasons:

- there is another operation already being executed;
- the current operation is a write, hence we must wait that a read is executed first;
- the current operation is a refresh, the previous was a write, and we are excluding $W \rightarrow R \rightarrow W$ sequences.

We therefore add, for all W_i, the constraint

$$t_{W_i} \geqslant t^Q_{W_i} + L_W \tag{16}$$

However, we still have to ensure that the enqueue times $t^Q_{W_i}$ are feasible w.r.t. the arrival curve α_W. To do so, we pair t_0 with each $t^Q_{W_i}$, and each $t^Q_{W_i}$ with each $t^Q_{W_j}$ with $j > i$. For each of these pairs we will add a constraint that the arrival curve property is maintained. This can be done considering the inverse of the arrival curve function

$$\alpha^{-1}_W(n) = \inf\{\ t \mid \alpha_W(t \cdot tCK) \geqslant n\ \}$$

which gives the minimum time (measured in clock cycles) for n write requests to arrive. We can then express the required constraints for the pairs $(t_0, t^Q_{W_i})$ as

$$t^Q_{W_i} \geqslant \alpha^{-1}_W(W_{high} + (i-1) \cdot N_{wd} - (W_{high} - 1)) \tag{17}$$

and for the pairs $(t^Q_{W_i}, t^Q_{W_j})$ as

$$t^Q_{W_j} \geqslant t^Q_{W_i} + \alpha^{-1}_W((j-i) \cdot N_{wd}) \tag{18}$$

6 Bounding Approaches

The performance of the MILP approach can be considerably improved by devising lower and upper bounds on the WCD.

The *lower bounds* are calculated by constructing *feasible schedules* that, under certain assumptions, would result in a WCD. The lower bound is therefore associated with a feasible solution that is provided to the MILP solver as a `mip starts`, also known as "warm starts", allowing the user to provide the solver with an advanced starting point for MIP optimization. The solver installs it as the incumbent solution, which allows it to more effectively eliminate portions of the search space, thus resulting in a smaller branch-and-bound tree and ultimately improving performance.

A valid *upper bound*, in general, can be obtained via *relaxation*. Instead of relaxing the MILP model, our upper bound on WCD is computed "directly" on the FSM, using an iteartive procedure that combines different worst-case conditions. The upper bound can then be used to estimate the maximum number of write batches N_W and refreshes N_R, as well as to tighten the value of the various "big-M" constants Δ, that the model entails (see constraints (3) and (14)), thereby improving the quality of the continuous relaxation and ultimately reducing the exploration of the enumeration tree.

6.1 Lower bounds

In the following, we describe how we obtained two of the most effective lower bounds, namely "patterns" that produce feasible schedules with large WCD. The two patterns are called *Max write rate* and *Single hit series*.

Max write rate: this pattern assumes that the write arrival process follows the α_W profile, i.e., it is the maximum allowed at any time. In other words, one has $Q(t) = \alpha_W(t)$. The schedule is constructed by repeating the following command sequence *until* the request under analysis is served:

- `IF` refresh timer expired → schedule a *refresh*.
- `IF` more than W_{high} writes in the queue `AND` last operation ≠ write batch → schedule a *write batch*.
- `IF` less than N_{cap} read hits performed → schedule a *read hit*.
- `ELSE` → schedule a *read miss*.

Intuitively, one can expect that maximizing the arrival process will generate a scenario with a large delay. However, a write batch may interrupt a read hit series, which could reduce the delay, as explained in Section 3.2.

Single hit series: this pattern attempts to improve on the previous one by delaying a write batch until the read hit series is completed. The schedule is constructed by repeating the following command sequence *until* the request under analysis is served:

- `IF` refresh timer expired → schedule a *refresh*.
- `IF` more than W_{high} writes in the queue `AND` last operation ≠ write batch `AND` no read hit series can be continued → schedule a *write batch*.
- `IF` less than N_{cap} read hits performed → schedule a *read hit*.
- `ELSE` → schedule a *read miss*.

If both patterns schedule the same number of write batches by the time the request under analysis is served, it can be proved that the "Single-hit-series" pattern yields a larger delay. However, the "Single-hit-series" pattern may actually schedule fewer write batches than the "Max-write-rate" one, in which case dominance is no longer guaranteed.

It should be remarked that the identification of such patterns was in part a by-product of the MILP model itself. Indeed, by examining the optimal solutions produced by early implementations of the MILP model, we were able to find useful sequences yielding large delays. In other words, a positive feedback loop, whereby an automated tool provided actionable insights, which in turn were used to further improve the tool itself.

6.2 Upper bounds

To compute the upper bound we devised an iterative algorithm that combines the delays due to the N *read misses* with three worst-case conditions, namely: maximum number of *write batches*, maximum number of *refreshes*, and a single (uninterrupted) *read-hit series*.

Read misses: We consider the request under analysis to be served when the data transfer along the bus is completed, i.e., $t_{M_N} = tRP + tRCD + tCL + tBurst$. For the preceding $N-1$ read misses, the delay cost is given by tRC cycles, i.e., the minimum spacing between two consecutive `PRE` commands. Thus, the total delay for the read misses is $(N-1)\,tRC + t_{M_N}$.

Write batches: Each write is a miss and requires tRC_W clocks. Writes are served in batches of N_{wd} operations, that are triggered when the backlog reaches the watermark level W_{high}. To calculate the corresponding delay we need to know the number of batches that are served before the request under analysis. If the write arrival process is unbounded, the write backlog will always be $> W_{high}$ and a write batch will be scheduled after each read. In this case, each read command will be followed by a write batch resulting in $N + N_{cap}$ write batches, hence, a delay of $(N + N_{cap})\, tRC_W$ cycles. If, instead, the arrival process is bounded by α_W, we need to find the maximum number of write requests that can arrive before the request under analysis is served. This number can be computed as follows. Denoting by δ_i the current upper bound estimate for the WCD, the corresponding number of write requests n^i_{WR} is $\alpha_W(\delta_i) + W_{high} - 1$, from which we can compute the number of write batches $n^i_W = \lfloor \frac{n^i_{WR}}{W_{high}} \rfloor$. Then, the upper bound estimate is updated $\delta_{i+1} = \delta_i + n^i_W\ L_W$, and can be used again to compute the corresponding number of write batches n^{i+1}_W. The process is iterated until the value becomes constant or $n^k_W = N + N_{cap}$ is reached.

Refreshes: The maximum number of refreshes can be computed in a similar way using an iterative process. Denoting again by δ_i the current upper bound estimate to the WCD, the corresponding number of refreshes n^i_R is given by $\lceil \frac{\delta}{tREFI} \rceil$. Then, the upper bound estimate is updated $\delta_{i+1} = \delta_i + n^i_R\ tRFC$ and can be used again to compute the corresponding number of refreshes n^{i+1}_R. The process is iterated until the value becomes constant.

Read hits: As a worst-case scenario we consider a single read-hit series that is scheduled after the shortest precharge bubble, yielding a delay of $N_{cap}\, tCL - \min\{tRPB, tWPB\}$ cycles.

To combine these delays, we use once angain an iterative approach. We start with an initial estimate δ_0 of the upper bound that only considers the read operations, with the exception of the request under analysis:

$$\delta_0 = tRC - 1 + (N-1)\, tRC + ((N_{cap} - 1)\, tCL + tRTP - \min\{tRPB, tWPB\})\,.$$

Then, δ_i (with $i = 0$ at the first iteration) is used to compute the corresponding number of write batches and refreshes

$$n^i_W = \left\lfloor \frac{\alpha_w(\delta_i) + W_{high} - 1}{W_{high}} \right\rfloor \quad , \quad n^i_R = \left\lceil \frac{\delta_i}{tREFI} \right\rceil \ .$$

Finally, the delay is updated accordingly

$$\delta_{i+1} = \delta_i + n^i_W\ L_W + n^i_R\ tRFC$$

and the process is iterated with the new value δ_{i+1}, until δ_k becomes constant yielding the final upper bound $\Delta = \delta_k + t_{M_N}$. As a byproduct of this iterative algorithm, the final values n^k_W and n^k_R are valid upper bounds on the number of write batches and refreshes that may be scheduled, hence we use them to set N_W and N_R in the final model.

It is important to remark that the upper bound algorithm described above can be run with or without assuming a specific arrival curve α_W for the write requests, yielding different Δ, N_W and N_R, hence different models:

"Tight" vs. "lax" FSM: We say the FSM is *tight* if α_W is used to to bound the number of nodes N_W and N_R when constructing $G^{\texttt{FSM}}$, otherwise the FSM is *lax*.

"Tight" vs. "lax" Δ: The upper bound Δ is used as a "big-M" coefficient in the MILP model, we say that the "big-M" are *tight* (*lax*) if α_W was (not) used to compute Δ.

Parameter	Value (ns)
tCL	14.16
tRCD	14.16
tBurst	3.332
tWTR	5
tWR	15
tRAS	32
tRP	14.16
tRTP	7.5
tRFC	350
tREFI	7800
tCK	0.833

Table 1: Parameters for a `DDR4 @2400 MHz`

"Tailored" Δ: In time-propagation constraints (3) one can try to compute a smaller Δ by considering the specific type of transition. For example, consider the time propagation constraint associated to transition $M_{N_s-1} \rightarrow M_{N_s}$:

$$t_{M_{N_s-1}} + c_{M_{N_s-1},M_{N_s}} - (1 - x_{M_{N_s-1},M_{N_s}})\,\Delta \leqslant t_{M_{N_s}} \leqslant t_{M_{N_s-1}} + c_{M_{N_s-1},M_{N_s}} + (1 - x_{M_{N_s-1},M_{N_s}})\,\Delta$$

The right value of Δ can be computed by using any upper bound on the maximum possible value of $t_{M_{N_s}}$ (since $t_{M_{N_s-1}} < t_{M_{N_s}}$ anyway). While the upper bound on the WCD is clearly such, when $N_s < N$ a tighter upper bound on $t_{M_{N_s}}$ can be computed by running the above upper-bound algorithm assuming $N_s\,(< N)$ read misses. We can apply this process to all time propagation constraints for transitions between M or H^R nodes. When we do this, we say that the "big-M" are *tailored*, otherwise they are *non-tailored*.

7 Computational Results

In this section we present some computational results for the proposed approach. As a test bench we took real data about current `DDR4 memory @2400 MHz`, whose parameters are reported in Table 1.
We consider 64 instaces obtained as follows:

- N between 1 and 16: the number of *read misses* (in the queue) to serve;
- write-rate (WR) between 4 and 7: the *rate* of incoming write requests, measured in Gbps (each request is 512 bits of data).

All parameters have been suggested and validated by experts of the field. The *service curves* provided by our model have been found to be useful in the analysis and design of `COTS FR-FCFS DRAM` controllers [11], but the details of such studies are outside the scope of our paper and of comparatively minor interest for the readership of this journal. Rather, we focus our analysis on the behaviour of our MILP model in terms of efficiency and effectiveness of the solution process, and in particular on the impact of the improvements discussed in Section 6. To do so we discuss the results of four configurations, which vary in how the bounds are used in the MILP approach, as described in Table 2.

We tested all 16 possible configurations, but for the sake of conciseness we only report results of four configurations, which we found to be the most significant ones. The computational results on the 64 instances of our test bench (16 Ns $\times$ 4 write rates) are shown in Table 3 and Table 4; in the former we group the results by N and in the latter by *write rate*, as doing so highlights

configuration	FSM	Δ	tailored Δ_s	mip starts
LL_nTM_nMP	lax	lax	no	no
TL_TM_MP	tight	lax	yes	yes
LT_nTM_nMP	lax	tight	no	no
TT_TM_MP	tight	tight	yes	yes

Table 2: Configurations tested

	Lax Δ						Tight Δ					
	LL_nTM_nMP			TL_TM_MP			LT_nTM_nMP			TT_TM_MP		
N	time	gap	#	time	gap	#	time	gap	#	time	gap	#
1	–	7.1024	0	151.29	7.1216	3	151.16	0.0020	3	150.83	0.0020	3
2	–	6.3401	0	451.97	6.2132	1	165.44	0.0009	3	150.58	0.0009	3
3	–	6.5851	0	–	6.5841	0	150.51	0.0003	3	150.46	0.0003	3
4	–	6.1360	0	–	6.1360	0	151.26	0.0003	3	164.86	0.0003	3
5	–	6.3526	0	–	6.3494	0	26.54	0.0000	4	15.73	0.0000	4
6	–	6.5565	0	–	6.5565	0	60.96	0.0000	4	41.61	0.0000	4
7	–	6.8822	0	–	6.8699	0	36.60	0.0000	4	37.62	0.0000	4
8	–	6.9164	0	–	6.9086	0	97.13	0.0000	4	98.75	0.0000	4
9	–	7.0970	0	–	7.0917	0	177.39	0.0003	3	119.52	0.0000	4
10	–	7.2783	0	–	7.2681	0	169.97	0.0000	4	232.94	0.0000	4
11	–	6.3739	0	–	6.3710	0	187.21	0.0005	3	204.01	0.0001	3
12	–	6.2858	0	–	6.2823	0	309.88	0.0268	2	302.31	0.0267	3
13	–	6.0193	0	–	6.0113	0	270.30	0.0003	3	300.15	0.0004	2
14	–	6.2430	0	–	6.2420	0	341.21	0.0007	2	83.42	0.0000	4
15	–	6.3974	0	–	6.3823	0	365.36	0.0005	2	204.31	0.0000	4
16	–	5.9456	0	–	5.8696	0	381.46	0.0242	2	327.14	0.0002	3

Table 3: Computational results grouped by N, time limit 600 seconds

some interesting trends about how the model scales with the fundamental parameters of the underlying system. Each instance was solved with the general-purpose, off-the-shelf, state-of-the-art MIP solver `Cplex` 12.10 run on a Virtual Machine with 48 Intel Xeon Gold 5120 virtual cores @ 2.20 GHz and 23 GiB of RAM. The solver was ran on *opportunistic* parallel mode, which by default used up to 32 threads, *feasibility* MIP emphasis, *polish-after time* set to 60 seconds and with a time limit of 600 seconds. In the tables, columns "time" report the average *running time* in seconds ("–" if all the instances terminated by time limit), columns "gap" report the average *mip gap* at termination, and columns "#" report the number of instances that were *solved* with default relative accuracy `1e-6` within the time limit.

The results clearly show how crucial the bound tightening techniques of Section 6 are for our MILP model, and in particular the computation of a "tight" bound Δ on the WCD. When Δ is "lax", no amount of improvement in the other parameters (the size of the FSM, the "tailored big-M" coefficients on the other constraints, and the "mip starts") is enough to significantly change the behaviour of the model (save for very small values of N), which within the time limit is not capable to solve the problem with any reasonable accuracy. However, tightening Δ suddenly and dramatically changes the outcome, with the model solving most instances to optimality, and anyway providing very accurate solutions within the (pretty reasonable) time limit. With a "tight" Δ, the other improvements have a minor but generally positive impact, especially as N grows. Somewhat surprisingly, table 3 shows a pretty limited impact of N—which significantly impacts the size of the FSM and therefore of the model—on the overall

	Lax Δ						Tight Δ					
	LL_nTM_nMP			TL_TM_MP			LT_nTM_nMP			TT_TM_MP		
WR	time	gap	#	time	gap	#	time	gap	#	time	gap	#
4	–	10.2418	0	525.56	10.2062	2	67.46	0.0001	15	0.84	0.0000	16
5	–	7.7226	0	562.60	7.7203	1	62.71	0.0000	16	79.35	0.0000	16
6	–	5.4814	0	562.84	5.4774	1	178.63	0.0005	13	134.42	0.0002	15
7	–	2.6821	0	–	2.6605	0	451.79	0.0136	5	431.44	0.0075	8

Table 4: Computational results grouped by *write-rate*, time limit 600 seconds

efficiency of the approach, which bodes well for the ability to apply it for a large number of misses to serve (with 16 considered already sufficient for most practical purposes). However, Table 4 shows that other aspects of the underlying system can have a significant impact. In particular, increasing the *write-rate* seems to negatively impact our "champion" models with "tight" Δ, although somewhat surprisingly having the opposite effect on the gap of the "lax" ones. Yet, the results show that for reasonable values of the input parameters our model is capable of finding very accurate estimates of the WCD in very reasonable time.

It should be remarked that our approach provides not only the WCD, that can be used to define service curves for existing DRAM controller and therefore be the basis for further analyses, but also feasible solutions (schedules) that realise them (possibly within a very minor gap). These have so far only been obtained by heuristic reasoning, which may well have missed some sources of delay in the context of a complex system like the one under study. In fact, our experience (see Section 6) is precisely that applying the model allowed us to gather precious insights on the behaviour of the system, that we had not been able to divine at the beginning of the study. While we used these to further improve the efficiency of the approach, the same information may provide system designers with important clues about which design decisions can lead to unexpectedly large WCD, and help in designing better next-generation DRAM controllers, more suited for application with tight worst-case requirements.

8 Conclusions

This work opens up an entirely new research direction, proving—to the best of our knowledge, for the first time—that mathematical optimization techniques and off-the-shelf MIP solvers can successfully compute tight estimates of the *service curve* of complex components like a DRAM controller. Indeed, a significant benefit of our approach is that of providing *composable* worst-case guarantees, in the form of a Network Calculus service curve, which allows our DRAM analysis to be reused as a building block in end-to-end analysis of multicore architectures. This is a crucial step for developing end-to-end performance guarantees for today's extremely complex systems, wherein a system-on-chip can require many interacting components whose complete characterization is highly nontrivial—yet necessary, all the more so as these systems become increasingly crucial in many advanced user-facing applications in telecommunications, automotive, robotics, and many other fields. As often happens, general-purpose tools cannot just be applied in a naïve way: deep knowledge of the underlying problem is necessary to build a successful approach, in our case under the form of upper and lower bounds on the WCD that can be used to strengthen the model. Tight collaboration between experts in optimization techniques and experts of the underlying application domain is therefore crucial for the success of such a project. When this is achieved, results that can be of immediate use to practitioners in the field, and be integrated in existing tools for, say, system-on-chip design, can be readily obtained. We hope that the precedent set by this work will encourage other researchers to pursue this line of inter-disciplinary approach, that may yield significant benefits both for the performance evaluation community and the mathematical optimization one.

References

[1] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[2] Falk Rehm, Jorg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high-performance platforms. In *25th Design, Automation and Test in Europe Conference DATE 2021, Grenoble, France, February 1-5, 2021*, 2021.

[3] Rakesh Kumar, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanya Piramanayagam, Sibin Mohan, and Rakesh B. Bobba. End-to-end network delay guarantees for real-time systems using sdn. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–242, 2017.

[4] Nassima Benammar, Frdric Ridouard, Henri Bauer, and Pascal Richard. Forward end-to-end delay for afdx networks. *IEEE Transactions on Industrial Informatics*, 14(3):858–865, 2018.

[5] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren D. Patel. A comparative study of predictable DRAM controllers. *ACM Trans. Embed. Comput. Syst.*, 17(2):53:1–53:23, 2018.

[6] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[7] Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. A composable worst case latency analysis for multi-rank DRAM devices under open row policy. *Real Time Syst.*, 52(6):761–807, 2016.

[8] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 299–308, 2012.

[9] R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 741–746, 2010.

[10] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[11] Matteo Andreozzi, Frances Conboy, Giovanni Stea, and Raffaele Zippo. Heterogeneous systems modelling with adaptive traffic profiles and its application to worst-case analysis of a DRAM controller. In *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*, pages 79–86. IEEE, 2020.

Appendix

Parameters of DRAM WCD problem belong to three classes

- Module: parameters of the underlaying DRAM module.
- Controller: parameters of the memory controller that schedules commands for the DRAM.
- Problem: other than the mentioned N, this includes other external environment factors such as the arrival profile $\alpha_W(t)$ for write requests.

DRAM Module parameters

The main parameters for the DRAM are the following.

- tCL: Column access (`CAS`) latency. For an open row, it counts the clock cycles between sending the column address to the DRAM and obtaining the result. It therefore measures the duration of a `RD` or `WR` command on an open row.
- $tRCD$: Row Address (`RAS`) to (`CAS`) Delay. It measures the duration of a `ACT` command on a closed row. After a row address is sent to the memory controller (via the `ACT` command), it counts the cycles before the column address can be sent. For a closed row, $tRCD + tCL$ cycles are required to get a result.
- tRP: Row Precharge Time. It is the duration of a `PRE` command. When a row is open, it counts the number of cycles required before selecting a different row. This means that it will take $tRP + tRCD + tCL$ cycles to access data in a different row.
- $tRAS$: Row Active Time. This is the minimum number of cycles that a row has to be open for, i.e. the minimum spacing between the `ACT` command and the successive `PRE` command. The value of $tRAS$ is usually between $tRCD + tCL$ and $tRCD + 2 \cdot tCL$.
- $tBurst$: The duration of the data burst transfer on the data bus for a single column command. As the `DDR` memories transfer data on both the rising and falling edges of the clock signal, if the burst length is BL, $tBurst = BL/2$ (i.e. $BL/2$ cycles are needed to transfer a number of adjacent columns equal to BL).
- $tRFEI$: Refresh Interval. Each cell in the array must be accessed and written back (restored) before the expiration of the refresh interval. Note that in order to refresh a row, the latter has to be selected: all banks must be precharged and idle for at least the Row Precharge Time (tRP) before the Refresh command can be applied. An address counter, internal to the device, supplies the bank address used during the course of the refresh cycle. When the refresh cycle has completed, all banks are left in the precharged (idle) state.
- $tRFC$: Refresh Cycle Time. The time it takes from the issuing of a refresh command until the bank can receive the next `ACT` command (i.e., can be used again for read/write access).
- $tWTR$: Write-to-Read command Delay. The number of clocks between the last valid write operation and the next read command to the same bank.
- $tRTP$: Read-to-Precharge Delay. Minimum number of clocks required between a `RD` command to a `PRE` command to the same rank. The computation of the time at which $tRTP$ begins to be considered depends on the DRAM type. For example, for a LPDDR2-S2 device, $tRTP$ begins $tBurst - 1$ clock cycles after the `RD` command. In this analysis, for simplicity, $tRTP$ is the minimum spacing between a `RD` and a `PRE` on the same bank.

- tWR: Write Recovery Time. The minimum number of clocks between the completion of a write operation (i.e. end of the $tBurst$) and the next `PRE` command.

From these, other useful parameters can be derived.

- tRC: Row Cycle Time. The minimum number of clock cycles between successive `ACT` commands to the same bank.
$$tRC = tRP + tRAS$$

- $tWTP$: Write-to-Precharge Delay. Minimum number of clocks that are inserted between a `WR` command to a `PRE` command to the same bank.
$$tWTP = tCL + tBurst + tWR$$

- $tRAS_W$: Write Row Active Time. The minimum number of cycles that a row has to be open for, when the operation executed is a write.
$$tRAS_W = \max(tRAS, tRCD + tWTP)$$

- tRC_W: Write Row Cycle Time. The minimum number of clock cycles between successive `ACT` commands to the same bank, when the operation executed is a write.
$$tRC_W = tRP + tRAS_W$$

- $tRPB$: Read-to-Precharge Bubble. The interval from a read miss to the time when the next `PRE` can be executed, during which a read hit could be executed.
$$tRPB = tRAS - (tRCD + tCL)$$

- $tWPB$: Write-to-Precharge Bubble. The interval from a write miss to the time when the next `PRE` can be executed, during which a read hit could be executed.
$$tWPB = tRAS_W - (tRCD + tCL + tBurst + tWTR)$$

- $N_{H^R}^{inner}$: Number of read hits that can fully occupy the bubble after a read miss.
$$N_{H^R}^{inner} = \lfloor \frac{tRPB - tRTP}{tCL} + 1 \rfloor$$

- N_{H^R}: Number of read hits that can occupy the bubble, fully or partially, after a read miss. It is equal to $N_{H^R}^{inner}$ if $tRPB = N_{H^R}^{inner} \cdot tCL$, otherwise it is equal to $N_{H^R}^{inner} + 1$.

- $N_{H^W}^{inner}$: Number of read hits that can fully occupy the bubble after a write miss.
$$N_{H^W}^{inner} = \lfloor \frac{tWPB - tRTP}{tCL} + 1 \rfloor$$

- N_{H^W}: Number of read hits that can occupy the bubble, fully or partially, after a read miss. It is equal to $N_{H^W}^{inner}$ if $tWPB = N_{H^W}^{inner} \cdot tCL$, otherwise it is equal to $N_{H^W}^{inner} + 1$.

DRAM Controller parameters

The type of DRAM memory controller assumed in this work is based on a FRFCFS policy, with a limit to the number of *overtakes* to avoid starvation. A *watermark* policy is used for switches between read and write requests. The main parameters for such a controller are the following.

- N_{cap}: Maximum number for read request overtakes that can happen.
- W_{high}: High watermark level for write requests backlog.
- W_{low}: Low watermark level for write requests backlog.
- N_{wd}: Minimum amount of writes per direction switch.

From these, we can derive the time L_W required to serve a write batch of N_{wd} writes. We assume all of them to be write misses, hence it is

$$L_W = N_{wd} \cdot tRC_W$$

Problem parameters

- N: the number of read requests enqueued at time t_0.
- α_w: the arrival curve for the write requests.