Heterogeneous systems modelling with Adaptive Traffic Profiles and its application to worst-case analysis of a DRAM controller

Invited paper

Matteo Andreozzi[(a)], Frances Conboy[(a)], Giovanni Stea[(b)], Raffaele Zippo[(c,b)]

(a) Arm Ltd.
Cambridge, UK
{name.surname}@arm.com

(b) Dip. di Ingegneria dell'Informazione
University of Pisa, Italy
{name.surname@unipi.it}

(c) DINFO
University of Florence, Italy
raffaele.zippo@unifi.it

***Abstract*— Computing Systems are evolving towards more complex, heterogeneous systems where multiple computing cores and accelerators on the same system concur to improve computing resources utilization, resources re-use and the efficiency of data sharing across workloads. Such complex systems require equally complex tools and models to design and engineer them so that their use-case requirements can be satisfied. Adaptive Traffic Profiles (ATP) introduce a fast prototyping technology, which allows one to model the dynamic memory behavior of computer system devices when executing their workloads. ATP defines a standard file format and comes with an open source transaction generator engine written in C++. Both ATP files and the engine are portable and pluggable to different host platforms, to allow workloads to be assessed with various models at different levels of abstraction. We present here the ATP technology developed at Arm and published in [5]. We present a case-study involving the usage of ATP, namely the analysis of the worst-case latency at a DRAM controller, which is assessed via two separate toolchains, both using traffic modelling encoded in ATP.**

Keywords—ATP, modeling, worst-case analysis, embedded systems

I. INTRODUCTION

Embedded systems are acquiring more and more importance in the context of critical applications, such as Industry 4.0, automotive industry, robotics and avionics. In the above context, providing firm upper bounds on the execution time of applications becomes of paramount importance. On the other hand, the paradigm of hardware design is moving more and more towards resource sharing, be it the transit network on chip or shared memories. By tying the performance of an application to the concurrent presence and behavior of others, this makes it more difficult to perform design-space exploration, verification and performance evaluation. Recently, hardware designers (such as Arm, with MPAM [3], and Intel, with CAT [4]) have addressed some of these concerns by envisaging hardware systems where *labeling* of transactions allows for Quality of Service provisioning techniques, such as static/dynamic resource partitioning, admission control, path routing, and packet scheduling. This has been shown to make systems more predictable, abating the so-called tail latency [2].

Such complex systems, defined as heterogeneous due to the variety of devices composing them, as opposed to simpler CPU/Memory SoCs, can be especially difficult to design and dimension for their target use-cases. Tools which ease design-space exploration of heterogeneous system are therefore in need, to allow designers to test and achieve the desired performance level on such systems when running their target use-cases.

In this paper, we introduce the AMBA Adaptive Traffic Profiles (ATP) framework, which is a portable way to generate input for heterogeneous system verification and/or design space exploration. ATP are flexible rule-based profiles, which can be configured to emulate the traffic injection behavior of masters in a heterogeneous system, e.g., a GPU accessing a range of RAM addresses. We introduce the profile syntax, and then describe the ATP engine, that allows one to run both *standalone* applications, where ATP masters communicate with ATP slaves, or *mixed-mode* applications, where the same two entities communicate through a *host platform* (e.g., the gem5 simulator [6]), which takes care of traffic forwarding. Moreover, as a case-study involving ATP, we discuss the worst-case analysis of a First-Ready, First-Come-First-Served (FR-FCFS) DRAM controller: we compute an upper bound on the maximum delay that a *read* request may undergo as a function of its position in the read queue, when the *write* request traffic is generated according to profiles that match ATP specifications. This allows us to characterize the DRAM controller through its *service curve*, hence to use it in worst-case analysis of complex scenarios. We compound our theoretical analysis with a simulative analysis, carried out in similar conditions, using gem5 fed with ATP profiles.

The rest of this paper is organized as follows. Section II introduces the ATP framework. In Section III we present the modeling of the DRAM controller, whereas Section IV discusses the tools used for its evaluation and presents performance results. We conclude the paper in Section V.

II. THE AMBA ATP FRAMEWORK

This section describes the AMBA ATP modelling framework and the event-based engine that allows it to be run.

A. Adaptive Traffic Profiles

Adaptive Traffic Profiles (ATP) are a synthetic traffic modelling framework. ATP enables users to model the dynamic behavior of computer systems devices when executing a specific workload. ATP was released by Arm in the form of non-confidential specifications and subsequently in the form of an open-

source reference implementation [5]-[6]. The ATP specifications lay down a series of fundamental principles upon which the Arm implementation and all other ones build.

ATPs represent a device executing a workload. There are four classes of profiles: *Master*, *Slave*, *Delay* and *Monitor*. A Master ATP models what a master (e.g., a GPU) would do, i.e. send memory requests and receive responses, according to some configurable address space/time pattern. A Slave ATP models what a slave (e.g., a memory) would do, i.e. respond to requests according to a fixed latency and bandwidth. A Delay ATP just does nothing for a configurable amount of time, thus acting as a null source. A Monitor ATP, finally, logs events related to other profiles (typically Master ones, possibly more than one simultaneously). Master and Slave ATPs can be *active*, *terminated* or *locked*. The last state occurs when the Master (Slave) ATP cannot send (receive) any more requests (e.g., because it has reached its limit of *outstanding transactions*). The ATP reverts to the *active* state when the above condition is removed. After completing all the transactions specified by its configuration, the ATP switches to *terminated*.

An ATP consists of a collection of *ATP FIFOs* specifying its behavior. These are linked in time and/or space via an event-based mechanism. An ATP FIFO is the basic block of the ATP Technology: it can be utilized to compose complex waveforms in the same way base harmonics can compose complex signals in signal theory. The ATP FIFO is a dynamic structure with its own *size* and *rate*. Two types of FIFOs are defined in ATP: a producer ("*write*") FIFO and a consumer ("*read"*) FIFO. A write FIFO fills a buffer at constant rate, up to its configured size (see Fig. 1).

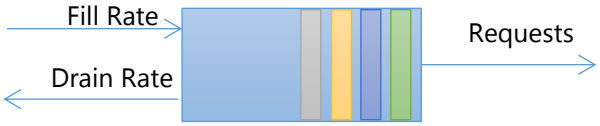

Fig. 1. A FIFO

After every fill, it is inspected and a number of memory write-requests are synthesized, based on their configured size and amount of data in the FIFO. Such data is marked as in-flight. Upon reception of acknowledgements (memory responses), in-flight data are marked as committed and removed from the FIFO. Should the FIFO be unable to accommodate the amount of data produced by its fill-rate, an *overrun* event would be logged instead.

A read FIFO is the dual of the write one and behaves very similarly: its rate is a *depletion* rate, and every time the FIFO updates its state, it consumes an amount of data according to such rate. Then, the FIFO produces as many memory read requests as can fit its *EMPTY* space and marks such requests as in-flight. Upon reception of acknowledgments (memory responses and data), the data is stored in the in-flight marked empty locations, and it is therefore made available to be consumed by the next FIFO rate activation. Should the read FIFO be unable to deplete enough data to satisfy its depletion rate, an *underrun* type even would be logged when that occurs.

An additional parameter, *TxnLimit*, can be set to limit the number of transactions "in-flight" that a FIFO can have at any given time, thus potentially reducing the amount generated after its fill/deplete phase. ATP FIFOs are complemented by a *pattern* object which describes how the addresses and data size fields of its generated transactions should be filled. Finally, an ATP FIFO Profile element groups together a FIFO and a pattern object into a self-contained descriptor, and assigns it to a system device master, as shown in the example of Fig. 2.

```
profile {
  type: READ
  master_id: "EXAMPLE"
  fifo {
       Full: 2048
       TxnLimit: 32
       Start: EMPTY
       Rate: "12GB/s"
       FrameSize: "2MB"
  }
  pattern {
    address {
      base:     0x0000
      increment:64
    }
  }
  name: "EXAMPLE_READS"
}
```

Fig. 2. Simple FIFO example

In this example, the FIFO will generate transactions at a sustained rate of 12GB/s, with a latency tolerance of about 165ns (12GB/s * 165ns = 1980 bytes, i.e. the FIFO size – 1 transaction, the one that needs to be drained to avoid underrun). All transactions will be filled with linearly incrementing memory addresses starting from the memory location 0x0000 with increments of 64 bytes.

```
profile {
  type: WRITE
  master_id: "EXAMPLE"
  fifo {
  Full: 2048
  TxnLimit: 32
  Start: EMPTY
       Rate: "12GB/s"
  FrameSize: "2MB"
  }
  pattern {
    address {
      base:     0x0000
      increment:64
    }
  }
  name: "EXAMPLE_WRITES"
}
```

```
profile {
  type: READ
  master_id: "EXAMPLE"
  fifo {
    Full: 2048
    TxnLimit : 32
    Start: FULL
    Rate: "12GB/s"
    FrameSize: "2MB"
  }
  pattern {
    address {
      base:     0xFFFF
      increment:  64
    }
  }
  name: "EXAMPLE_READS"
  wait_for: "EXAMPLE_WRITES PROFILE_LOCKED"
}
```

Fig. 3. Linked FIFOs for a *memcopy* operation.

As a slightly more complex example, we discuss how to emulate a *memcopy operation*. This boils down to simply adding an additional write-type FIFO to the one of Fig. 2, and linking them via a *"Linked FIFO"* type of event, in this case "PROFILE_LOCKED", which causes a FIFO to lock (i.e., it cannot issue transactions, as if its *TxnLimit* was reached, or it did not have any more data to use for issuing write requests – or empty space to issue read requests) when its linked one unlocks and vice-versa. This type of event creates correlation between the two linked FIFOs, in this case between a READ and a WRITE one, therefore replicating the typical read/write alternate behavior of a *memcopy* operation. The example is reported in Fig. 3.

B. The AMBA ATP Engine

The AMBA ATP Engine is a platform-independent module that generates synthetic traffic according to the ATP specifications. It can be plugged into event- or time-driven software modelling, simulation and testing platforms, such as gem5 [7], via a simple API. The Engine can work in *standalone* or *mixed* mode. In the first mode, traffic is exchanged among TPs directly. In the mixed mode, an external software (*host platform*) conveys traffic to/from TPs.

The core component of the Engine is the Traffic Profile Manager (TPM), which manages the configured ATPs, schedules their events, and connects to external platforms via the API. Events are triggered by ATPs, and are scheduled in an efficient event list, implemented as a calendar queue. Examples of events are "FIFO_EMPTY/FULL", which occur when an ATP's FIFO becomes empty or full, "PROFILE_LOCKED" when a Master ATP is unable to send data, etc.

The Engine includes facilities for logging events to a stream, with different levels of verbosity, and for gathering statistics, aggregated per master, such as the number of packets sent/received, the number of overruns/underruns in the buffer, the send/receive rate, latency, jitter, and average FIFO level.

As anticipated, the Engine can be connected to a host platform. For instance, this allows a Master ATP to send memory requests to the host platform and receive responses from it. This is realized in practice by adding an *adaptor layer* to the host platform, acting as a bridge between the Engine's API and the host platform's API. ATP comes pre-packaged with a gem5 adaptor layer, so that its integration with gem5 does not require any effort on the part of the user. Any adaptor layer should always interact with the TPM engine, and exchange information with the latter in the form of either C++ objects or serializable Google Protocol Buffer [8] objects. If required, the adaptor layer should take care of format conversion between ATP and host packets. The host platform is required to provide time ticks for the Engine.

The ATP Engine provides a handful of APIs available to users wishing to develop their own adaptor. *Control* API can be utilized to configure the TPM (e.g. to set its options, its time domain scaling, logging level etc.) and to load ATP files into the Engine. At run time, the adaptor layer is only required to call the *send* API to obtain packets from the Engine and to provide packets to the Engine via the *receive* API. The gem5 Adaptor *ProfileGen* - is implemented as a *MemObject* derived class, which wraps around the Traffic Profile Manager and allows gem5 to send and receive memory request and response packets from the ATP Engine. *ProfileGen* connects to other gem5 objects by instantiating a configurable number of master ports dedicated to the individual ATP masters, through which it sends and receive packets belonging to such masters.

III. Worst-case analysis of a DRAM controller

As an example of an application, we propose a study on the worst-case access delay at a DRAM controller. In a heterogeneous system the DRAM is a shared resource used by multiple devices. As such it is a point of contention that may affect both the average performance and the compliance to real-time constraints. This study focuses on the second aspect, i.e. to obtain the maximum delay that a *read* request may suffer under a given contention scenario. We compare the results of two distinct techniques: a simulative approach with gem5 and an analytical one using Network Calculus (NC, [1]). For both techniques, we leverage ATP modeling of traffic.

The system under study is a DRAM module accessed by multiple devices via a controller that schedules commands (reads, writes, refreshes) according a combination of policies. This system is represented in Fig. 4.

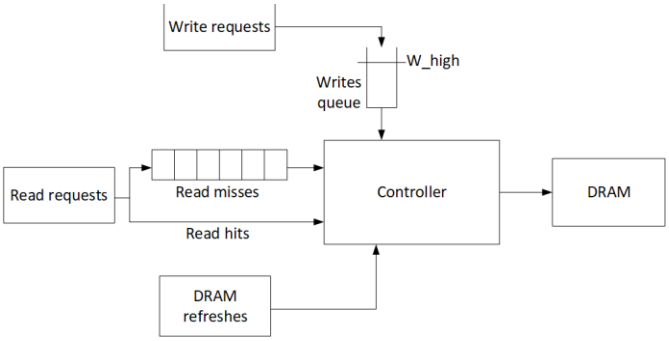

Fig. 4. System model of a DRAM controller

In a general case, contention among requests is avoided with various mechanisms. A DRAM stick is composed of multiple chips which may serve requests independently. The controller may leverage this to run requests concurrently. Another example is when a read request addresses data for which a write request is enqueued. The controller may then short-circuit the read request with that data, without waiting for the write request to be committed to DRAM first. Since we focus on the worst case, we need to assume a scenario where these mechanisms are ineffective: all requests target the same bank, so that they cannot run concurrently, and no read-write pair addresses the same data, so that no short-circuit occurs. Under these assumptions, the controller must arbitrate and issue the requests one at a time.

Due to the DRAM hardware characteristics, any request needs to "open the row" *before* accessing cells. This leads to different servicing times between requests that target a row that had already been opened by a previous request ("row hit") and those that must instead close the current row and open another one ("row miss"). The controller takes this in consideration, servicing read requests according to a "First Ready, First Come First Served" policy (FR-FCFS): row miss requests are scheduled in order of arrival, but a request that would results in a row hit can *overtake* the others and be scheduled with higher priority. The controller limits the amount of times a given read can be *overtaken* to a maximum of N_{cap}. In fact, since row hits have strict priority over row misses, this limit is necessary to guarantee that row misses are served within a finite time in a worst case. Note that, depending on the DRAM parameters (but this is the most frequent case in our experience), serving *all* the N_{cap} read hits back-to-back often results in the highest delay.

To serve a write request, the bus direction must be reversed first, which incurs a time overhead. Thus, controller policies normally aim at avoiding unnecessary switches between the two directions. In our system, a *watermark* approach is used to switch between reads and writes. With reference to Fig. 5, the relevant

parameters are the *high* and *low watermark thresholds*, W_{high}, W_{low}, and the *write batch length* N_{wd}.

When in *read* mode, the controller switches to serving writes when either of the following conditions holds:

1. The read queue is empty, and there are at least W_{low} write requests in queue;
2. There at least W_{high} write requests in queue.

When in *write* mode, the controller switches to serving reads when either of the following conditions holds:

1. The read queue is empty, and write queue is below max $(W_{low} - N_{wd}, 0)$;
2. The read queue is not empty, and N_{wd} writes have been served.

As we are envisaging a worst-case scenario where the read queue is never empty, we can neglect conditions 1 of each case without any loss of generality. Thus, only parameters W_{high} and N_{wd} are used in our study.

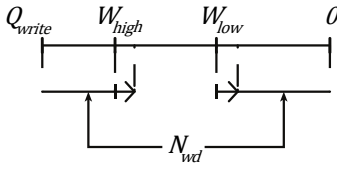

Fig. 5. Watermark policy for read/write switching. Depending on the presence of enqueued read requests, a different watermark is used to decide when to switch to serving writes and vice versa.

Lastly, DRAM memories require periodic refreshes to avoid loss of data, so a refresh timer is used to schedule them. The controller schedules refreshes whenever the timer fires, after the current read or write operation is completed.

We focus on bounding from above the maximum delay experienced by a *row miss* read request which enqueues at the N^{th} position of the read queue, call it t_N. In fact, the curve that joins points (t_N, N) is a *service curve* for the system under study. A service curve is an NC construct that represents the worst-case impulse response of a system to a batch of requests. We refer the interested reader to [1] for a tutorial on NC, and limit ourselves to mention that the main property of service curves is *composability*: given a tandem of devices characterized by service curves S_1 and S_2, we can compute a service curve for the tandem $S_{1,2} = S_1 \otimes S_2$, where $\otimes$ denotes the *min-plus convolution* algebraic operator. Modeling a FR-FCFS controller via a service curve enables the study of the worst-case performance of composite systems involving DRAM memory access, as well as (say) network-on-chip traversal, in a heterogeneous setting with multiple masters and several resource contention points.

In order to obtain an upper bound on the maximum delay of a row miss, we need to envisage a worst-case scenario, possibly including pejorative assumptions. However, overestimating the maximum delay inevitably leads to overprovisioning a system, hence it pays to keep these pejorative assumptions to a minimum in order to obtain *tight* upper bounds. In this respect, characterizing the arrival process of *write* requests makes a considerable difference. Lacking any such characterization, in fact, the only assumption compatible with a worst-case scenario is that writes are *unbounded*, i.e., the write queue is always above the watermark, so that each *read* request is always followed by a *write* batch. In this case, envisaging the scenario leading to the worst-case delay of the N^{th} read is relatively straightforward (we leave it to the alert reader), but the delay thus obtained is unrealistic for at least three independent reasons: first, masters do not send infinite batches of write all the time; second, rate-limiters can be (and often are) employed at the entrance of a shared NoC to limit the amount of requests sent by a single (e.g., misbehaving) master; third, the NoC itself acts as a rate limiter, hence the write bandwidth cannot exceed the NoC bandwidth along the path from the master to the controller. For all the above reasons, we assume that the process feeding the write queue at the controller is *upper bounded*. In fact, ATP masters emit requests at a constant rate. However, since the DRAM is a shared resource, multiple masters may emit requests at the same time. Compounded with the NoC traversal, where contentions can lead to increase of burstiness, we can then assume that multiple requests, possibly from different masters, may arrive at the controller at a higher *peak* rate than those at which they were emitted. For these reasons, we model the arrival process with a *token bucket shaper* (Fig. 6), with arbitrary but known parameters *burst* and *rate*. The *burst* parameter b (the vertical offset) models the fact that a number of concurrent requests may arrive near-simultaneously. 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 $R(t)$ is upper bounded by a token bucket shaper with a shaping curve $\alpha(\tau) = b + r \cdot \tau, \tau > 0$, implies that $\forall \tau, R(t + \tau) \leq \alpha(\tau) + R(t)$. 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).

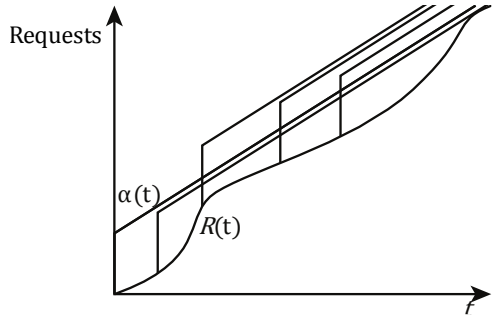

Fig. 6. Example of token bucket shaper. The traffic process $R(t)$ is always below the shape function $\alpha(t)$ and its translations along $R(t)$.

IV. Performance Evaluation

This section reports performance evaluation results. We describe the tools that were used for the assessment, the configuration of the experiments, and the results.

A. Tools used

We present the two approaches used to study the above system. The first one is an analytical approach, where we developed a mathematical model for the worst-case timing of concurrent events that would delay the read of interest. The second one is a simulative approach, where ATP has been used to create request streams close to the worst-case assumptions of the analytical

model, observing the behavior of the controller under such scenario.

In the analytical approach, the assumptions are used to develop a mathematical model of the worst-case timing of concurrent requests and refresh timer. This is done by focusing on the time windows in which a concurrent event (arrival of hit reads, switching to writes, scheduling of a refresh) may add the highest delay to the servicing of miss reads. We modelled the possible sequences of such events as a Finite State Machine (FSM), where each operation is modelled as a state and the transition cost between states A and B is the "miss read delay" introduced by scheduling B after A. An example of such FSM is in Fig. 7. The parameters to construct such model are the timing parameters of the DRAM chip, controller parameters such as N_{cap} and the write watermark parameters, the target queue position of the request under study N.

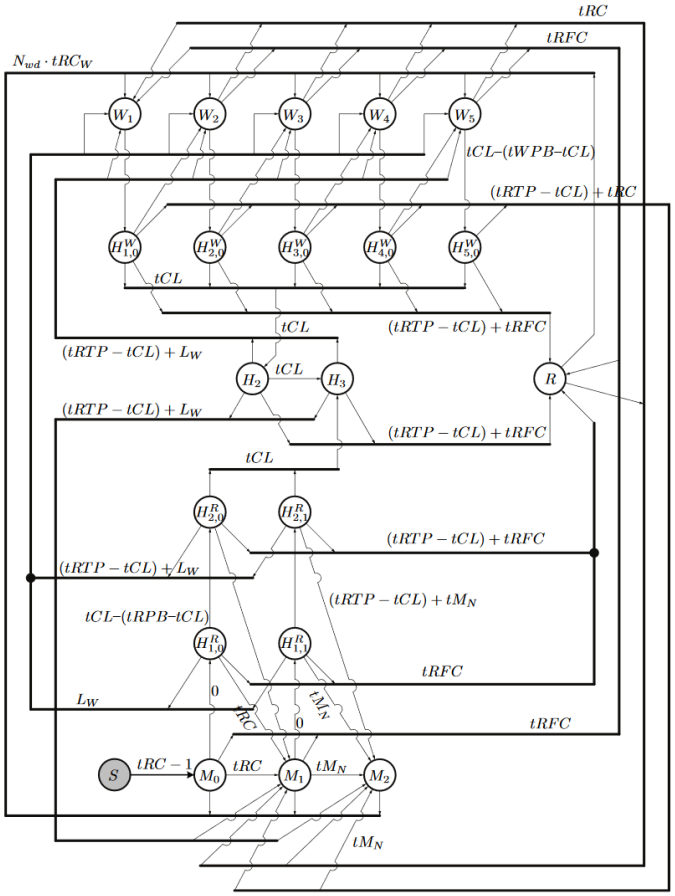

Fig. 7. Example of Finite State Machine for the possible events occurring before serving N read misses. The number of instances of each type of node depends on the problem parameters.

We then developed a C++ tool that efficiently computes an upper bound to the path of maximum cost that can be followed on such FSMs to go from the starting point to the execution of the N^{th} read, whose worst-case delay we set to obtain. The upper bound is constructed as follows:

- we assume that, at time 0, the controller has just started serving a read miss; N more reads are enqueued, and all of them will result in a miss;
- we assume that, at time 0, the write queue contains $W_{high} - 1$ requests;
- from time 0 onward, the arrival process of writes is the maximum allowed by the token bucket constraint, i.e., $W(t) - W(0) = b + r \cdot t$, where $W(t)$ are the writes arrived by time t;
- just after time 0, N_{cap} new read requests will arrive, which will result in hits (hence will overtake read N, whose worst-case delay we are assessing); these will be served all in a batch, to maximize the impact on the delay of read N;
- by the time the N^{th} read is served (call it t_N), $\lceil t_N/T \rceil$ refresh cycles will occur, T being the refresh period.

Under the above hypotheses, we navigate the FSM according to the FR-FCFS controller policy described above. This allows us to compute an upper bound on the delay, counting in all the overhead induced by every operation and change of state, quantified based on the timing parameters of the DRAM model being analyzed and on the FR-FCFS configuration (e.g., values of N_{cap}, W_{high}, N_{wd}). We stress that what we obtain is *an upper bound* on the worst-case delay. In fact, some of the conditions in the above bullet list may not be actually possible. We show this via a simple example.

Assume that the write arrival process is shaped by a token bucket whose rate is r. Assume also that N is very large, so that the controller will have to serve many write batches before scheduling the N^{th} read miss. A quick back-of-the-envelope computation is sufficient to compute the largest interval of time during which the controller will be serving reads between any such two time batches, which is $\Delta = N_{wd}/r$. Among our hypotheses, we find that the controller should schedule N_{cap} read hits back-to-back, since this maximizes the delay of the N^{th} read miss. However, one can always envisage an N_{cap} so large that the longest read interval Δ is not long enough to serve all the hits back-to-back. This means that, in order to compute the maximum *feasible* delay for the N^{th} read miss, we should try to partition the set of N_{cap} read hits among several read intervals. Depending on the timing parameters of the controller, finding such optimal partitioning may not be straightforward.

In order to obtain the *exact* worst-case delay, an alternative algorithm is required, which would exhaustively explore the FSM to obtain the highest-cost *feasible* path to the N^{th} read miss. Such an algorithm, which can clearly be expected to have a much higher computational costs, is being researched at the time of writing. The advantage of the one presented in this paper is that it is computationally cheap (few milliseconds, even for large values of N, under very broad settings of DRAM and controller parameters). This, together the fact that it does not require complex operations, means that it could also be used to make *online* decisions in a system (e.g., to decide whether or not a new master should be admitted, based on expected worst-case delays for reads), possibly implemented in hardware.

In order to benchmark the above analytical approach, we simulate the system under study using gem5. In the simulative approach, ATP FIFOs are used as input to a modified gem5

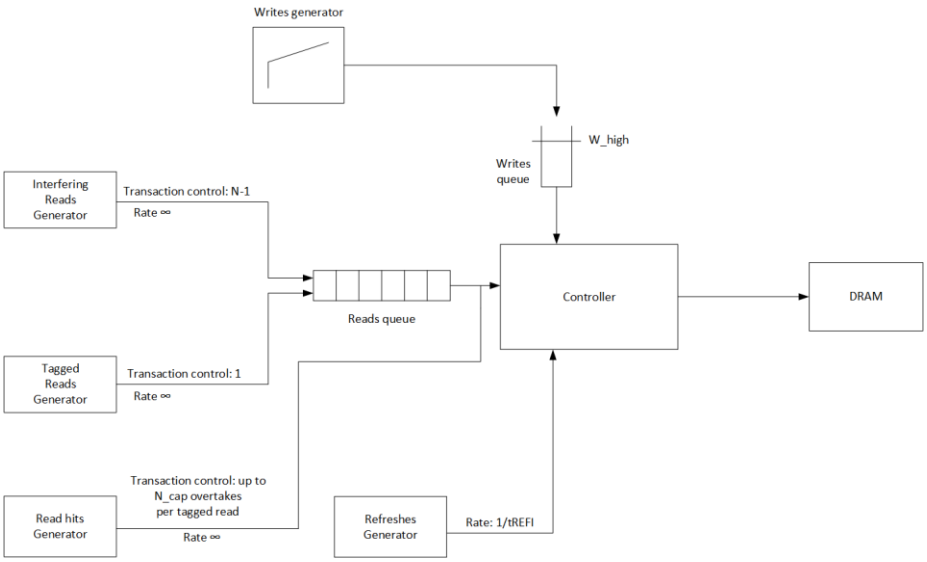

Fig. 8. Simulation schema, using ATP FIFOs to recreate a quasi-worst-case scenario.

model of a DRAM memory device and controller to recreate a scenario close to the worst case, as summarized in Fig. 8. The modifications to the gem5 model include a "locking" mechanism to prevent requests from being served until there are the required number of packets enqueued, and a manually triggered refresh to ensure that the time taken by refreshing the memory at least once is taken into account in the worst-case measurement.

The ATP FIFOs used are set out in Fig. 9, and can be split into "setup" and "run" profiles. The setup profiles produce the packets that fill the input queues of the controller to meet the initial conditions as described above – N read-misses in the read queue and $W_{high} - 1$ write-misses in the write queue. The interfering reads generator produces $N - 1$ packets, as the tagged request is enqueued in the N^{th} position. The run profiles include one to produce the tagged packet (upon receiving which the controller will start to serve requests), a profile producing read-hits that overtake the tagged request, and one producing writes at a variable rate. The controller serves at least one read before switching to writes even if the write queue is above the watermark, so the read-hits address row 101, which is the row of the first interfering read to be served after the initial set of N_{wd} writes.

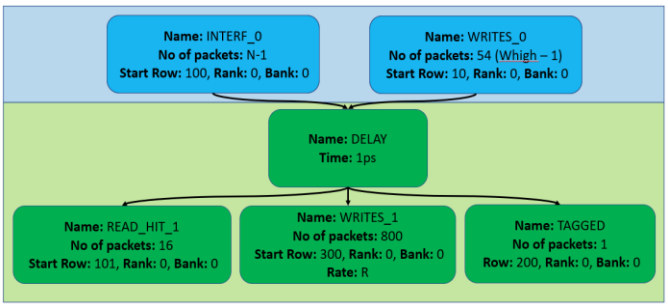

Fig. 9. ATP FIFO configuration, setup in blue, run in green. Arrows indicate waited for profiles

B. *Experiment description and configuration table*

The controller parameters for the write queue are shown in Table I. Three memory configurations were tested and compared: a DDR3-1600 with timings based on a DDR3-1600 4 Gbit datasheet, a DDR4-2400 based on an 8 Gbit datasheet, and a LPDDR-3200 based on a 4Gbit per channel datasheet. An open-adaptive page management policy was used, which means that the controller keeps a row buffer open once accessed, but closes it if there are no row hits and there are bank conflicts in the queue. Address mapping can be varied according to the number of ranks and the page policy being used. The mapping used was RoRaBaCoCh, which can be expanded to Row-Rank-Bank-Column-Channel and determines how the address is decoded. The memory parameters are reported in Table II. The timing parameters are given in Table III, and are derived from the Joint Electron Device Engineering Council (JEDEC) standards for DDR3, DDR4, and LPDDR4. Experiments were run using values of N between 2 and 55 (higher values of N led to the read-hits being re-sent and were not added to the input queue in time to be served as hits). The bitrate of the incoming write request packets was varied from 1 Gbps to 8 Gbps.

TABLE I. WRITE QUEUE PARAMETERS

Maximum entries	64
W_{high}	55
W_{low}	32
N_{wd}	16

TABLE II. MEMORY PARAMETERS

	DDR3_1600	DDR4_2400	LPDDR4_3200
Device size	512 MB	1 GB	512 MB
Bus width	8 b	4 b	16 b
Burst length	8	8	16
Device row buffer size	1 kB	512 B	2 kB
Banks per rank	8	16	8
Ranks per channel	2	2	1
Page policy	Open-adaptive	Open-adaptive	Open-adaptive
Address mapping	RoRaBaCoCh	RoRaBaCoCh	RoRaBaCoCh

TABLE III. DRAM TIMING PARAMETERS (NS)

	DDR3_1600	DDR4_2400	LPDDR4_3200
tCK	1.25	0.833	0.625
tBURST	5	3.332	5
tRCD	13.75	14.16	18
tCL	13.75	14.16	18
tRP	13.75	14.16	18
tRAS	35	32	42
tRRD	6	3.332	10
tXAW	30	13.328	40
tRFC	260	350	180
tWR	15	15	18
tWTR	7.5	5	10
tRTP	7.5	7.5	7.5
tRTW	2.5	1.666	2.5
tCS	2.5	1.666	1.25
tREFI	7800	7800	3900
tXP	6	6	7.5
tXS	270	340	188

C. Results

We studied the DRAM models described in the previous paragraph under different write contention scenarios, comparing the results from the analytical and simulative approaches.

Fig. 10 reports the DDR3 service curve as a function of the write bitrate. As can be seen, the worst-case rate (i.e., the slope of the service curve) is reduced as the "write intervals" become more frequent, which implies that the "read intervals" get shorter. Comparing the two approaches, we can verify that the analytical results are upper bounds to the scenarios produced via simulation. The distance between the two increases with the contention, which is likely due to a) pessimism in the upper bound approach becoming more impactful, and b) simulation scenarios becoming less capable to capture the worst-case sequence of events. Fig. 11 and Fig. 12 show similar results, respectively, for DDR4 and LPDDR4, for which the same reasoning applies.

Comparing the results for the DDR3 and DDR4 memories in Fig. 13, we can see that DDR4 has *inferior* worst-case performance with respect to the DDR3. This counterintuitive result is due to design trade-offs aimed to improve memory density and bandwidth, for which the DDR4 model has higher timing parameters than DDR3, namely the refresh length tRFC (350 ns vs. 260 ns). This difference is stressed out in a worst-case scenario, since targeting a single bank prevents one from leveraging the other improvements of the DDR4. This shows that improving the *average* performance or the *worst-case* performance are different objectives, and one may be achieved at the expenses of the other.

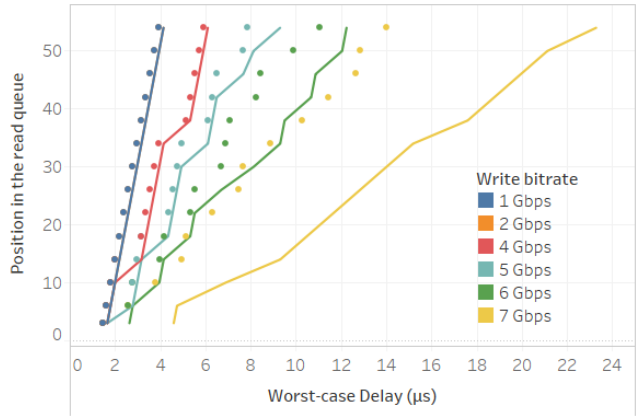

Fig. 10. Service curve of the DDR3 DRAM under varying write contention. Continuous lines show the Network Calculus upper bounds, dots are the gem5 simulative lower bounds.

Fig. 11. Service curve of the DDR4 DRAM under varying write contention. Continuous lines show the Network Calculus upper bounds, dots are the gem5 simulative lower bounds.

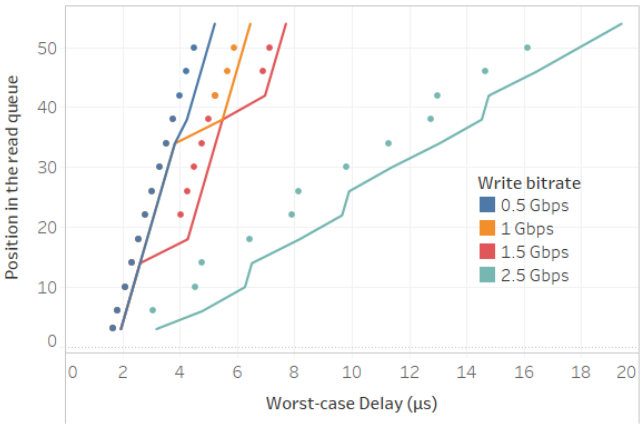

Fig. 12. Service curve of the LPDDR4 DRAM under varying write contention. Continuous lines show the Network Calculus upper bounds, dots are the gem5 simulative lower bounds.

Fig. 14 compares the Low Power DRAM to the others, and clearly shows the effects on performance of the power-efficiency tradeoff. Note that the studied LPDDR4 memory has a different packetization to the other two, since each write request consists of 256 bits of data, against 512 bits in DDR3 and DDR4. Thus, for the same write rate, the worst-case contention is further amplified.

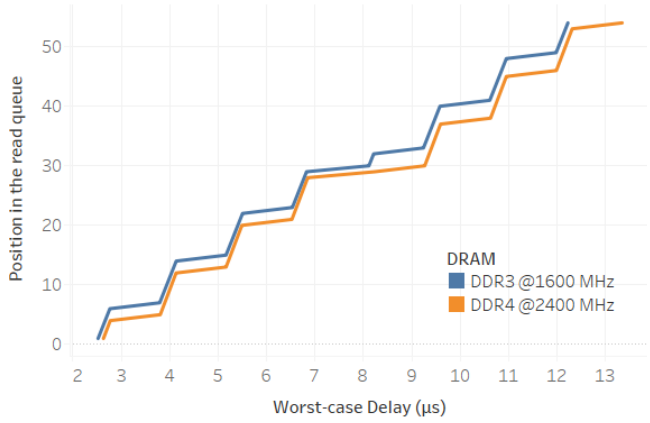

Fig. 13. Service curve of DDR3 and DDR4 memories when the write rate is equal to 6 Gbps.

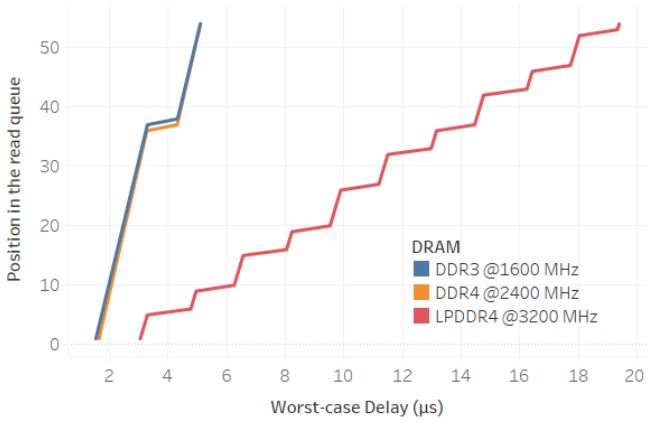

Fig. 14. Service curve of DDR3, DDR4 and LPDDR4 memories when the write rate is 2.5 Gbps.

V. Conclusions

This paper presented an overview of the AMBA ATP profiles, a fast prototyping technology, which allows users to model the dynamic memory behavior of computer system devices when executing their workloads. ATP profiles can be used either in a standalone mode, or combined with a host platform (e.g., the gem5 simulator), to which they can inject packets (e.g., requests) and from which they obtain the associated responses. We used ATP modeling to derive the worst-case behavior of a memory controller through Network Calculus, in the form of a service curve. ATP modeling allows one to capture the inherent limitations of interference due to the environment (e.g., the fact that the amount of write requests per unit of time is limited), in order to derive tighter characterizations, which can in turn lead to a higher system utilization. Our analysis allows one to obtain non-trivial results: for instance, the fact that DDR4 actually exhibits *worse* performance than a DDR3, as far as worst-case delay is concerned.

Acknowledgments

Work partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence). The authors would like to thank Mike Campbell of Arm, for his insight on the DRAM technology, and Lorenzo Biagini, former MSc student at the University of Pisa, for fruitful discussion on the worst-case analysis.

References

[1] J.-Y. Le Boudec and P. Thiran. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet, volume LNCS 2050. Springer-Verlag, revised version 4, 2019.

[2] YG Bao, S Wang, “Labeled von Neumann architecture for software-defined cloud”, Journal of Computer Science and Technology 32 (2), 219-223

[3] Arm® Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A, available online at https://developer.arm.com/docs/ddi0598/latest

[4] Intel Cache Allocation Technology, https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[5] AMBA Adaptive Traffic Profiles, https://pages.arm.com/amba_adaptive_traffic_profiles_specifications.html

[6] ATP Engine, https://github.com/ARM-software/ATP-Engine

[7] Gem5, http://www.gem5.org

[8] Google Protocol Buffers, https://developers.google.com/protocol-buffers