Modeling DRAM Timing in Parallel Simulators with Immediate-Response Memory Model

Stijn Eyerman, Wim Heirman, Ibrahim Hur

Abstract—Accurately modeling memory timing in a processor simulator is crucial for obtaining accurate and useful performance predictions. DRAM has a complex timing and reordering scheme, which results in highly varying access latencies depending on the type of operations, address stream patterns and bandwidth load. Therefore, DRAM simulators model the DRAM timings as a clocked state machine. However, some processor simulators, in particular loosely synchronized parallel simulators, assume an immediate-response memory model, requesting an immediate estimation of the memory latency. In this paper, we discuss the modeling issues in transforming a state machine DRAM simulator into an immediate-response simulator, which can be directly plugged into a processor simulator supporting an immediate-response memory model and/or a relaxed parallel simulation model. We show that the adapted model is accurate within 2% compared to the state machine simulator.

Index Terms—Memory timing simulation, parallel simulation, high-level simulation

1 Introduction

As the gap between memory and compute speed continues to increase, more and more applications become memory-bound. Therefore, performance estimation models need to incorporate accurate memory timing models to provide accurate and useful projections. Modern DRAM modules (e.g., DDR4) have a complex timing behavior, and efficient memory controllers reorder memory operations to optimize memory performance. As a result, the latency of a memory operation can vary a lot, and memory operations can be reordered to increase throughput.

To accurately model this behavior, most memory simulation models are clocked state machines: each individual memory command (precharge, activate, read, write, refresh, etc.) is modeled as a state transition, and the next command starts when the previous is finished, following the timing specifications. If, in the meantime, other requests arrive in the memory controller, they are queued, and when the command bus is free, the next request is selected from the queue following the scheduling policy. When a request finishes (its last command is done), it is removed from the queue and its latency is known.

Processor simulators model the timings of instructions within the cores and caches of a (multicore) processor. Some processor simulators focus on core modeling and have a very simple memory model, often a fixed latency. A fixed latency has the advantage that the latency is immediately known, and the latency can be added to the instruction latency. We refer to this model as an *immediate-response* model: the latency is obtained in a single function call. A state-machine memory simulator can not be plugged in easily into a processor simulator with an immediate-response memory model, because the latency is only known after the request completes, during which the processor also makes progress, i.e., the processor model cannot be stalled until the request finishes. On the other hand, an immediate-response model is more straightforward to implement (and maintain) and is usually faster, because the latency is determined in one step as opposed to multiple state changes. However, an immediate-response model cannot model reorderings and command interleaving, as the latency of a request must be known before seeing the next requests.

In this paper, we target an immediate-response memory model that can be directly plugged into a dito processor simulator, *and* that approximates the timings of a state-machine simulator as closely as possible. This paper makes the following contributions:

- We raise the issue that processor simulators with an immediate-response memory model interface and/or that use relaxed synchronization cannot use accurate state machine DRAM models.
- We discuss the fundamental modeling issues in matching an immediate-response memory model with a more accurate state machine model.
- We modify a state machine DRAM simulator into an immediate-response simulator, resulting in an average 2% error or lower versus the state machine model, while being 1.7× faster.
- We integrate this model into a multicore performance simulator, and show that more accurate DRAM models have a considerable impact on the performance simulations compared to a fixed-latency model.

2 Processor and Memory Simulators

Processor timing simulators need to incorporate a memory timing model, because the performance of memory-bound applications is determined by memory performance. Separate memory timing simulators, such as DRAMSim2 [6] and Ramulator [5], use a clocked state machine to accurately model the individual DRAM commands and their timings. Therefore, most of the commonly used processor and GPU simulators, such as gem5 [2] and GPGPUSim [1], moved to an asynchronous memory model interface: they send a request to the memory model, along with a callback function that should be called when the request is finished. In the meanwhile, the processor simulator continues its simulation, potentially sending more requests to the memory model. When the callback function is executed, the processor timing model continues the simulation of the instruction. Integrating a state machine memory model with a processor simulator therefore requires that the simulated processor clock is always synchronized with the simulated memory clock.

However, high-level simulators (such as interval simulation [4]) or simulators that focus on specific aspects of a processor (such as the simulators used for branch predictor/cache replacement/value prediction championships) use a simple immediate-response model: the memory model needs to immediately respond with the latency, such that the processor model can continue modeling the timing

- *All authors are with Intel Corporation, {stijn.eyerman, wim.heirman, ibrahim.hur}@intel.com*

of the memory operation before proceeding to the next instruction. Immediate-response models often assume a constant memory access time, possibly extended with a queue model to estimate queuing time. Memory timings are, however, highly variable depending on whether an access is a page hit or miss, or whether or not it is reordered to increase the page hit rate. Not only the access latency is variable, the available bandwidth can also differ depending on the read/write ratio.

Additionally, fast parallel multicore simulators are loosely synchronized to limit synchronization overhead. A memory controller is accessed by multiple cores, each with their own clock, where some clocks might be further in the future than others. This means that the memory model receives requests out of order: a core that is lagging behind can send a request that should have appeared before already submitted requests from the other cores. State machine simulators cannot handle this timing anomaly, they assume a monotonically increasing clock. Therefore, loosely synchronized parallel simulators have to resort to an immediate-response memory model.

Because of this synchronization issue, the most commonly used processor simulators have a single-threaded simulation model, even if multiple cores are simulated. To the best of our knowledge, there are no parallel processor simulators with a state machine memory model. An immediate-response model with the accuracy of a state machine model could increase the parallelization opportunities for accurate simulators. For parallel simulators, such as Sniper [3], and high-level simulators, such as interval simulation, it can improve accuracy by moving away from an inaccurate constant memory latency model.

3 Modeling Events in an Immediate-Response Model

During our attempt to transform a state machine (SM) memory simulator into an immediate-response (IR) model, we encountered three fundamental issues that required a novel modeling approach to enable accurate memory latency projections. Before discussing these issues and our solutions, we briefly explain the memory timing model that is used in SM simulators and how we mimic that in our IR model.

3.1 Memory Timing Model

In the memory controller (MC), each memory request (load or store) is translated into one or more memory commands, e.g., a precharge, activate and read command for a load to a bank that has a different page currently open. Each of these commands has several timing restrictions wrt to other commands, which can differ across the memory structure levels (channel, bank, rank). For example, on a channel (which is directly connected to the MC), the MC needs to make sure that there are at least 4 cycles between two read or write operations, to leave enough room to send the cache line data. On the bank level, timings between precharge, activate and read need to be enforced to make sure that the command has physically finished.

In a state machine simulator, these timings are modeled as a 'ready' clock per command per level. For example, there is a ready clock for read commands for each bank, indicating when the next read command can start. On arrival, requests are pushed in the read or write queue, and on every tick of the memory model clock, the queues are checked to see if a command is ready, i.e., all ready clocks at each level for that command are at or before the current memory clock. If a command is scheduled, all of the timing constraints it induces are added to the respective ready clocks, delaying future commands. Once a request has finished all its commands, it is removed from the queue and a callback function is called, reporting that the request has finished.

In our immediate-response model, we should be able to schedule requests out of order although we need to process them and estimate their finish time in arrival order. Therefore, we do not keep a single ready clock, but a recent history of timings, containing the timings of the last few commands and their timing restrictions. When a request is made, it contains its simulated arrival time, which could be earlier than already processed requests. The request is split up into individual commands, and all commands are scheduled one after another, but in a single transaction, because we need to answer immediately with the expected latency. For each command, the history is checked, and the command is scheduled at the earliest simulated time after its arrival time, which could be before already scheduled commands if it does not violate the timing restrictions.

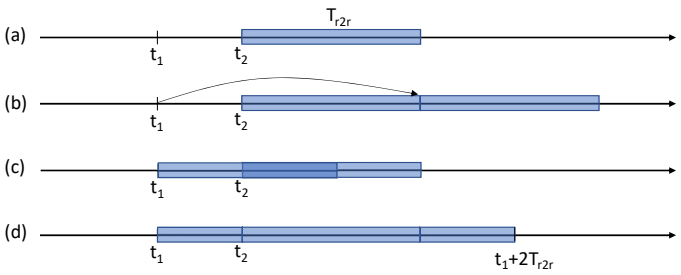

Fig. 1. Out-of-order arrival model. (a) A read access is scheduled at t_2, after which an earlier access arrives at $t_1 > t_2 - T_{r2r}$. (b) Scheduling the access at t_1 after the access at t_2. (c) Overlapping the timing constraints to allow the access to start at t_1. (d) Our solution: splitting the timing constraint of t_1 such that there is no overlap and all gaps are filled.

Our IR model also models DRAM refresh cycles. Refresh happens at fixed points in time, namely multiples of the refresh period. If there is such a point in between the simulated time of the previous request and that of the current request, we schedule a refresh command. Note that if a request is already scheduled at that particular point in time, the refresh is scheduled later, obeying all timing restrictions between commands.

While this transformation from an SM to an IR model seems straightforward, there are a few issues that need to be investigated and that require non-straightforward solutions, which we discuss in the next sections.

3.2 Out-of-Order Arrival and Scheduling

In parallel simulators, requests may occur out-of-order, which means that future requests are already scheduled, while an earlier request should have been scheduled before. Scheduling the earlier request before the future requests might violate the timings, but we cannot move later the already scheduled requests, because their latency is already reported to the core simulation model.

Figure 1 sketches the issue: a read command is scheduled at t_2 and adds the restriction that no other read can be scheduled between t_2 and $t_2 + T_{r2r}$, with T_{r2r} the minimum read-to-read command time[1]. Somewhat later, a read command is scheduled at t_1, which is earlier in simulated time. However, $t_2 - t_1 < T_{r2r}$, so a there is a timing violation. In a globally synchronized SM simulator, we would first see the command at t_1 and then that at t_2, which we need to delay until $t_1 + T_{r2r} > t_2$.

In our IR model, we could argue that since we violate the timing constraints, we need to schedule the request at t_1 after the request at t_2, so at $t_2 + T_{r2r}$ (Figure 1b). More generally, we look for the next gap that fits a full block of length T_{r2r} to schedule the request. This will lead to many small gaps that cannot be filled, and therefore an underutilization of the capacity. This does not occur in the SM simulator, which can potentially schedule each command back to back, resulting in a performance underestimation by the IR model.

Inversely, we can decide to schedule the t_1 command at time t_1 (like in the SM simulator) and ignore the timing constraint violation,

1. We use this notation for readability, the actual model uses the conventional DRAM timings, such as tRAS, tRCD, etc.

i.e., we let the timing constraints overlap (Figure 1c). Clearly, this will lead to an overutilization of the capacity if there are many overlaps, and thus a performance overestimation.

With the insight of achieving average overall accuracy in mind, the solution is to schedule the command at t_1 but to add the timing constraint that would overlap after that of t_2, such that the next command can start at $t_1 + 2T_{r2r}$ at the earliest, see Figure 1d. Note that the processor simulator still sees the request at t_2, because that was reported before we encountered the request at t_1. Compared to the SM model, the timing of the request at t_1 is correct, but that at t_2 is too early. Any later request will need to start at $t_1 + 2T_{r2r}$ at the earliest, which is in line with the SM model. The earliness of t_2 is compensated by a longer delay for the next request, which comes $t_1 + 2T_{r2r} - t_2 > T_{r2r}$ after t_2.

Next to the command-to-command delays (where both commands are the same), there are also inter-command delays. For example, at the rank level, there is a delay of starting a new read operation after a write operation (write to read delay T_{w2r}). These delays can and should overlap. For example, a write at $t = 0$ forces the next read to be at $t = T_{w2r}$ at the earliest. A subsequent write (without an intervening read) at $t = T_{w2w} < T_{w2r}$ does not delay reads until $t = 2T_{w2r}$, but until $t = T_{w2w} + T_{w2r}$, i.e., the T_{w2r} delays overlap. This should be taken into account in the IR model: inter-command delays can overlap with each other, but cannot overlap with intra-command delays.

3.3 Delayed Writes

Writes to memory, mainly writebacks from caches, are treated differently from reads. They are not critical for performance, because the cores are not waiting for their completion, in contrast to reads. Therefore, they are buffered and processed on idle memory cycles (with no reads pending) or when the write buffer gets full. Additionally, because there is an extra penalty of switching between reads and writes, writes are done in bursts.

Delaying writes is not straightforward in a IR memory model. Fortunately, the processor simulator does not request a latency number on writes, because writes are not stalling a core, except when a burst of writes delays a subsequent read. Therefore, we can keep them in a queue and we do not need to model their timing immediately. On every read request, we check whether since the previous read, there were cycles with no pending reads. If so, we first model draining the write queue up to a certain occupancy, before modeling the timing of the current read operation, which could be delayed because of the write burst. We also start this process whenever the write queue occupancy exceeds a certain threshold.

3.4 First-Ready Policy

Memory controllers use a first-ready first-come-first-serve (FR-FCFS) scheduling policy: of all pending requests, the one that can be scheduled first is selected; if multiple requests can be scheduled, the oldest is selected. For example, of all requests going to the same bank, the ones going to the currently open page are scheduled first, because they only need a read command to finish. Thereafter, a new page is opened and the requests to that page are scheduled. Similarly, if there are many requests to the same bank followed by some requests to other banks, the later requests can be scheduled in between the first requests to exploit bank parallelism. The latter case is already naturally modeled by our IR mechanism: a burst of requests to the same bank will induce a large delay on that bank's access history, but later requests to different banks will see idle time on their bank's history, which means that they can be scheduled at an earlier simulated time than the last seen access to the contended bank.

The open page based reordering is not modeled: accesses to the same bank will be scheduled in their arrival order, because we cannot see future accesses. To model this reordering, we do not only look at the page buffer content at the moment the read request can be scheduled, but we scan the full open page history between its arrival time and its schedule time, see Figure 2. If during that period, the requested page was opened by another request, we assume that the current request was scheduled while that page was open and will see a page hit. However, we cannot schedule this request at that particular time (because of the timing constraints), so we schedule it later, but assuming a page hit, and thus requiring a single read command. In reality, this request would have been scheduled earlier, so in this case, we do not update the open page history with the timing of the current request, because its timing is wrong (too late). This avoids unrealistic chains of open pages, i.e., requests that arrive after the page has been closed (in simulated time), will not see page hits because of this delayed page hit.

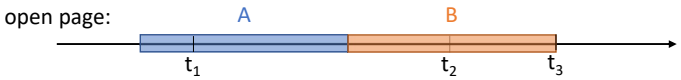

Fig. 2. Open page history. The next request is to page A and can be scheduled at t_3 at the earliest. At t_3, B is the currently open page. If the next request arrives at t_1, page A is open during its queuing time, so we model a page hit, but we do not reopen page A at t_3. If it arrives at t_2, only page B was open, so we model a page miss, and open page A at t_3. In both cases, the request access time is at t_3.

4 Experimental Setup

We use Ramulator [5] as our baseline state machine memory simulator. Ramulator has been validated against other DRAM simulators and hardware, and is, according to the authors, one of the fastest DRAM simulators. It also has an interesting implementation for our study: the timing engine (the SM scheduler) is completely separate from the specific memory timings model (DDR3-4-5, GDDR, LPDDR, HBM, etc.). Each memory model can introduce new commands and timing restrictions, without requiring changes to the timing engine. By transforming the timing engine to a immediate-response model, we can reuse the specific memory timing models unchanged and we are ready to use new Ramulator models as they appear. We refer to the baseline Ramulator model as *SMRam*, and use *IRRam* for our transformed IR model. For our initial results, we compare the *SMRam* and *IRRam* using the simple core simulator in Ramulator.

Next, we integrate *IRRam* into the Sniper multicore simulator [3], which models a fixed memory latency by default. Because of the IR approach, it can be plugged in without changing anything to the Sniper timing model. Note that we cannot easily integrate *SMRam* into Sniper, because that would require drastic changes to the Sniper core model *and* cycle-by-cycle synchronization, which significantly reduces simulation speed. We show that modeling memory more accurately has a considerable impact on performance compared to using the same fixed latency and bandwidth for all applications.

We perform all experiments on 1-billion instruction traces of the SPEC CPU 2017 benchmarks. These traces are collected using the SimPoint methodology, with a maximum of 10 representative traces per benchmark, for all reference inputs, resulting in 272 different traces. To show the accuracy of our technique on diverse workloads, we do not average results per benchmark; instead we consider each trace as an individual workload. We evaluate three memory technologies: DDR3, DDR4 and GDDR5. For the Sniper simulations, we configure a single core based on the Intel Skylake architecture.

5 Evaluation

As a first soundness test, we evaluate *SMRam* and *IRRam* on 2 synthetic traces: one with sequential addresses and one with random addresses. *IRRam* is 0.3% and 1.6% off compared to *SMRam* for

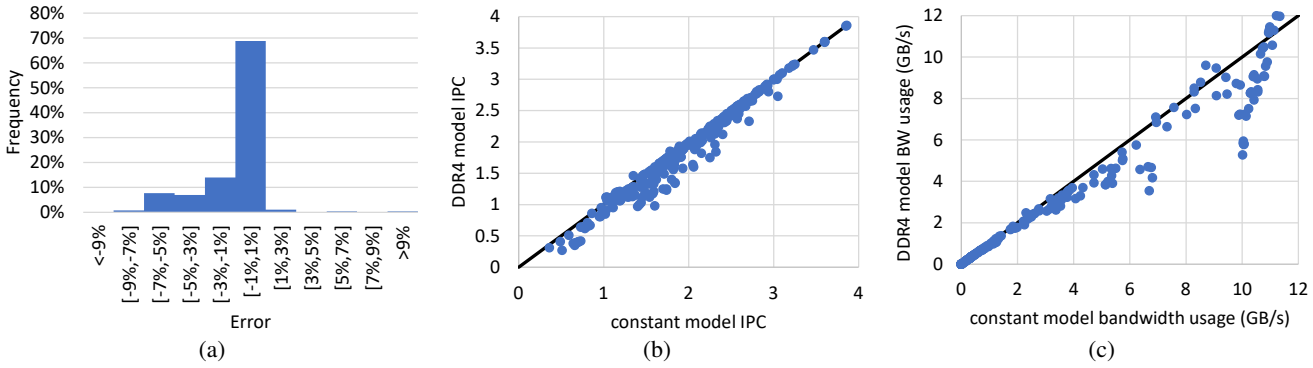

Fig. 3. DDR4 model evaluation: (a) Simulated time error histogram of *IRRam* versus *SMRam* on the 272 SPEC CPU 2017 traces, both using Ramulator's simple core model. (b,c) IPC and bandwidth comparison between a constant memory model and a DDR4 timing model. Points under the bisection have a lower value for the DDR4 model than for the constant model.

sequential and random, respectively, and reports the same page hit rate (99% and 0%).

Next, Figure 3(a) shows the error histogram of running the 272 SPEC CPU2017 traces on *IRRam* for DDR4 versus baseline *SMRam*. The simple Ramulator core model is an out-of-order superscalar core (width 4) where all non-memory operations and cache hits take one cycle (ignoring dependences), only the timing of memory accesses is simulated using the memory timing model.

The average absolute error is 2.1%, 1.3% and 0.9% for DDR3, DDR4 and GDDR5, respectively. The error is less than 5% for 87% (DDR3), 91% (DDR4) and 98% (GDDR5) of the traces. In comparison, a constant latency model, similar to the default Sniper model, has an average error of 11%, with outliers of more than 50%. The average error on the page hit rate, defined as *IRRam* page hit rate minus *SMRam* page hit rate, is 3.1%, 1.8% and 1.1% for the three memory technologies, with again the vast majority of errors smaller than 5%. The page hit rates of all traces vary between 0.1% and 97.8%. *IRRam* is on average 1.7× faster than the original *SMRam*, and up to 4.5× for traces with low memory traffic.

Next, we integrate *IRRam* model into Sniper, and compare against the default constant bandwidth constant latency memory model. The constant model includes a queuing delay estimation: if memory traffic gets close to the configured constant bandwidth, queuing delay is added based on a simple bus occupancy model. To compare to the best possible constant memory model, we first run Sniper with *IRRam* for DDR4. We then determine the largest achieved bandwidth (12 GB/s), and use that as the constant model's bandwidth. The maximum obtained bandwidth is lower than the theoretical peak bandwidth of 19.2 GB/s, because of timing constraints other than the channel bandwidth. We also calculate the average access latency across all simulations, and use that as the constant latency. Note that this is a unrealistically optimistic model for constant latency: in absence of an accurate memory model, we cannot determine these values as accurately. Compared to the simple constant latency memory model, simulation time increases by 14% on average when using *IRRam*.

Figure 3(b,c) shows the constant model versus *IRRam* in terms of IPC and used memory bandwidth for each of the simulated points. The memory access latency (total latency minus queueing latency) is constant at 27 ns for the fixed model and ranges between 17 ns and 60 ns for *IRRam*. There is a large variation in the IPC (9% on average and up to 89%) and bandwidth (9% on average and up to 88%), especially for the memory-intensive traces (low IPC, high bandwidth).

The *IRRam* IPC and bandwidth usage are mostly lower than that of the constant model, despite the fact that they have exactly the same average latency and peak bandwidth. For non-memory intensive benchmarks, the memory latency is usually lower than average, however, this does not lead to higher IPC because they are not sensitive to memory latency. On the other hand, memory intensive benchmarks have a higher memory latency in *IRRam*, and they are sensitive to this higher latency, explaining why they have a lower IPC in the DDR model than in the constant latency model.

Bandwidth usage is lower, because the constant model does not take into account the lower bandwidth efficiency when mixing read and write operations. At higher bandwidths, the usage of *IRRam* is sometimes higher, because the available peak bandwidth (19.2 GB/s) is actually higher than the configured constant bandwidth (12 GB/s). However, setting the constant bandwidth higher will result in even higher overestimations of used bandwidth.

Lastly, we also ran a straightforward IR DDR model, which keeps the open page state and busy times of all banks, without modeling overlaps, reorderings and bursts. The average IPC is 12% lower than that of *IRRam* and page hit rate is underestimated by 15% (and up to 30%), because of not modeling the first-ready reorderings and overlaps.

6 Conclusions

Accurately modeling memory timing is crucial for accurate processor performance predictions. The complexity of current memory technology requires a clocked state machine memory simulation model. However, because of simplicity, speed or relaxed synchronization, some simulators do not support a state machine model, but instead require an immediate estimate of the memory latency. We transformed a state machine memory model to an immediate-response model to target the best of both worlds: the simplicity and speed of an immediate response interface and the accuracy of a state machine model.

References

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*. IEEE, 2009, pp. 163–174.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *SC*, Nov. 2011.

[4] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA-16*. IEEE, 2010, pp. 1–12.

[5] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE CAL*, vol. 15, no. 1, pp. 45–49, 2015.

[6] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE CAL*, vol. 10, no. 1, pp. 16–19, 2011.