Ramulator: A Fast and Extensible DRAM Simulator

Yoongu Kim
Carnegie Mellon University

Weikun Yang
Peking University

Onur Mutlu
Carnegie Mellon University, onur@cmu.edu

Follow this and additional works at: http://repository.cmu.edu/ece

Part of the Electrical and Computer Engineering Commons

Published In

This Article is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Electrical and Computer Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.
1 INTRODUCTION
In recent years, we have witnessed a flurry of new proposals for DRAM interfaces and organizations. As listed in Table 1, some were evolutionary upgrades to existing standards (e.g., DDR4, LPDDR4), while some were pioneering implementations of die- stacking (e.g., WIO, HMC, HBM), and still others were academic research projects in experimental stages (e.g., Udipi et al. [38], Kim et al. [24]).

<table>
<thead>
<tr>
<th>Segment</th>
<th>DRAM Standards &amp; Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commodity</td>
<td>DDR3 (2007) [14]; DDR4 (2012) [18]</td>
</tr>
<tr>
<td>Performance</td>
<td>eDRAM [28], [32]; RLDRAM3 (2011) [29]</td>
</tr>
</tbody>
</table>

Table 1. Landscape of DRAM-based memory

At the forefront of such innovations should be DRAM simulators, the software tool with which to evaluate the strengths and weaknesses of each new proposal. However, DRAM simulators have been lagging behind the rapid-fire changes to DRAM. For example, two of the most popular simulators (DRAMSim2 [36] and USIMM [7]) provide support for only one or two DRAM standards (DDR2 and/or DDR3), as listed in Table 2. Although these simulators are well suited for their intended standard(s), they were not explicitly designed to support a wide variety of standards with different organization and behavior. Instead, the simulators are implemented in a way that the specific details of each standard are integrated tightly into their codebase. As a result, researchers — especially those who are not intimately familiar with the details of an existing simulator — may find it cumbersome to implement and evaluate new standards on such simulators.

The lack of an easy-to-extend DRAM simulator is an impediment to both industrial evaluation and academic research. Ultimately, it hinders the speed at which different points in the DRAM design space can be explored and studied. As a solution, we propose Ramulator, a fast and versatile DRAM simulator that treats extensibility as a first-class citizen. Ramulator is based on the important observation that DRAM can be abstracted as a hierarchy of state-machines, where the behavior of each state-machine — as well as the aforementioned hierarchy itself — is dictated by the DRAM standard in question. From any given DRAM standard, Ramulator extracts the full specification for the hierarchy and behavior, which is then entirely consolidated into just a single class (e.g., DDR3.h/cpp). On the other hand, Ramulator also provides a standard-agnostic state-machine (i.e., DRAM.h), which is capable of being paired with any standard (e.g., DDR3.h/cpp or DDR4.h/cpp) to take on its particular hierarchy and behavior. In essence, Ramulator enables the flexibility to reconfigure DRAM for different standards at compile-time, instead of laboriously hardcoded different configurations of DRAM for different standards.

The distinguishing feature of Ramulator lies in its modular design. More specifically, Ramulator decouples the logic for querying/updating the state-machines from the implementation specifics of any particular DRAM standard. As far as we know, such decoupling has not been achieved in previous DRAM simulators. Internally, Ramulator is structured around a collection of lookup-tables (Section 2.3), which are computationally inexpensive to query and update. This allows Ramulator to have the shortest runtime, outperforming other standalone simulators, shown in Table 2, by 2.5× (Section 4.2). Below, we summarize the key features of Ramulator, as well as its major contributions.

- Ramulator is an extensible DRAM simulator providing cycle-accurate performance models for a wide variety of standards: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, SALP, AL-DRAM, TL-DRAM, RowClone, and SARP. Ramulator’s modular design naturally lends itself to being augmented with additional standards. For some of the standards, Ramulator is capable of reporting power consumption by relying on DRAMPower [5] as the backend.
- Ramulator is portable and easy to use. It is equipped with a simple memory controller which exposes an external API for sending and receiving memory requests. Ramulator is available in two different formats: one for standalone usage and the other for integrated usage with gem5 [4]. Ramulator is written in C++11 and is released under the permissive BSD-license [1].

2 RAMULATOR: HIGH-LEVEL DESIGN
Without loss of generality, we describe the high-level design of Ramulator through a case-study of modeling the widespread DDR3 standard. Throughout this section, we assume a working knowledge of DDR3, otherwise referring the reader to literature [18]. In Section 2.1, we explain how Ramulator employs a reconfigurable tree for modeling the hierarchy of DDR3. In Section 2.2, we describe the tree’s nodes, which are reconfigurable state-machines for modeling the behavior of DDR3. Finally, Section 2.3 provides a closer look at the state-machines, revealing some of their implementation details.
2.1 Hierarchy of State-Machines

In Code 1 (left), we present the DRAM class, which is Ramulator’s generalized template for building a hierarchy (i.e., tree) of state-machines (i.e., nodes). An instance of the DRAM class is a node in a tree of many other nodes, as is evident from its pointers to its parent node and children nodes in Code 1 (left, lines 4–6). Importantly, for the sake of modeling DDR3, we specialize the DRAM class for the DDR3 class, which is shown in Code 1 (right). An instance of the resulting specialized class (DRAM<DDR3>) is then able to assume one of the five levels that are defined by the DDR3 class.

```
1 // DRAM.h
2 template <typename T>
3 class DRAM {
4 DRAM<T>* parent;
5 vector<DRAM<T>*> children;
6 T::Level level;
7 int index;
8 // more code...
9 };
10
```

Code 1. Ramulator’s generalized template and its specialization

In Figure 1, we visualize a fully instantiated tree, consisting of nodes at the channel, rank, and bank levels.1 Instead of having a separate class for each level (DDR3_Channel, DDR3_Rank, DDR3_Bank), Ramulator simply treats a level as just another property of a node — a property that can be easily reassigned to accommodate different hierarchies with different levels. Ramulator also provides a memory controller (not shown in the figure) that interacts with the tree through only the root node (i.e., channel). Whenever the memory controller initiates a query or an operation, it results in a traversal down the tree, touching only the relevant nodes during the process. This, and more, will be explained next.

```
1 // DDR3.h/cpp
2 class DDR3 {
3 enum class Level { Channel, Rank, Bank, Row, Column, MAX };
4 DRAM<Channel> Instance
5 - level = DDR3::Level::Channel
6 - index = 0
7 ;
```

```
1 // DDR3.h/cpp
2 class DDR3 {
3 enum class Status { Open, Closed, ..., MAX };
4 enum class Command { ACT, PRE, RD, WR, ..., MAX };
5 }
```

Code 2. Specifying the DDR3 state-machines: states and functions

2.2 Behavior of State-Machines

**States.** Generally speaking, a state-machine maintains a set of states, whose transitions are triggered by an external input. In Ramulator, each state-machine (i.e., node) maintains two types of states as shown in Code 2 (top, lines 5–6): `status` and `horizon`. First, `status` is the node’s state proper, which can assume one of the statuses defined by the DDR3 class in Code 2 (bottom). The node may transition into another status when it receives one of the `commands` defined by the DDR3 class. Second, `horizon` is a lookup-table for the earliest time when each command can be received by the node. Its purpose is to prevent a node from making premature transitions between statuses, thereby honoring DDR3 `timing parameters` (to be explained later). We purposely neglected to mention a third state called `leaf_status`, because it is merely an optimization artifact — `leaf_status` is a sparsely populated hash-table used by a bank to track the status of its rows (i.e., leaf nodes) instead of instantiating them.

**Functions.** Code 2 (top, lines 9–11) also shows three functions that are exposed at each node: `decode`, `check`, and `update`. These functions are recursively defined, meaning that an invocation at the root node (by the memory controller) causes these functions to walk down the tree. In the following, we explain how the memory controller relies on these three functions to serve a memory request — in this particular example, a read request.

1 Due to their sheer number (tens of thousands), nodes at or below the row level are not instantiated. Instead, their bookkeeping is relegated to their parent — in DDR3’s particular case, the bank.

2 An address is an array of node indices specifying a path down the tree.

11 void update(T::Command cmd , int addr[], long now);
10 bool check(T::Command cmd , int addr[], long now);
9 T::Status status;
8 long horizon[T::Command::MAX];
7 map<int, T::Status> leaf_status; // for bank only
6 // functions (recursively traverses down tree)
5 T::Command decode(T::Command cmd, int addr[]);
4 void check(T::Command cmd, int addr[], long now);
3 update(T::Command cmd, int addr[], long now);
2 class DDR3 {
1 enum class Status { Open, Closed, ..., MAX};
0 enum class Command { ACT, PRE, RD, WR, ..., MAX };
information to do so, forcing it to invoke the decode function at its child (i.e., rank). When a command cannot be decoded at a level, the lambda returns a sentinel value (i.e., MAX), indicating that the recursion should continue on down the tree, until the command is eventually decoded by a different lambda at a lower level (or until the recursion stops at the lowest-level).

```c
DRAM.h

template <typename T>
class DRAM{
private:
    //...}
public:
    //...}

12   // consult lookup-table to decode command
13   if (prereq[level][cmd]) {
14       // invoke decode() at the target child...
15       return cmd; // decoded successfully
16   }
17   if (children.size() == 0) // lowest-level
18       return cmd; // decoded successfully
19   // use addr[] to identify target child...
20   // invoke decode() at the target child...
21   }

22   // DDR3.h/cpp
23   class DDR3{
24       // declare 2D lookup-table of lambdas
25       function<Command(DRAM<DDR3>*)>
26           prereq[Level::MAX][Command::MAX];
27       // populate an entry in the table
28       prereq[Level::Rank][Command::REF] =
29           [] (DRAM<DDR3>* node) -> Command {
30               for (auto bank : node->children)
31                   if (bank->status == Status::Open)
32                       return Command::PREAD;
33                       return Command::REF;
34       }
35       // populate other entries...
36       }
37
38   Code 3. The lookup-table for decode(): prereq

Check & Update. In addition to prerequisite, the DDR3 class also provides two other lookup-tables: transition and timing. As is apparent from their names, they encode the status transitions and the timing parameters, respectively. Similar to prerequisite, these two are also indexed using some combination of levels, commands, and/or statuses. When a command is issued, the update function consults both lookup-tables to modify both the status (via lookups into transition) and the horizon (via lookups into timing) for all of the affected nodes in the tree. In contrast, the check function does not consult any of the lookup-tables in the DDR3 class. Instead, it consults only the horizon, the localized lookup-table that is embedded inside the DRAM class itself. More specifically, the check function simply verifies whether the following condition holds true for every node affected by a command: horizon[cmd] <= now. This ensures that the time, as of right now, is already past the earliest time at which the command can be issued. The check function relies on the update function for keeping the horizon lookup-table up-to-date. As a result, the check function is able to remain computationally inexpensive — it simply looks up a horizon value and compares it against the current time. For performance reasons, we deliberately optimized the check function to be lightweight, because it could be invoked many times each cycle — the memory controller typically has more than one memory request whose scheduling eligibility must be determined. In contrast, the update function is invoked at most once-per-cycle and can afford to be more expensive. The implementation details of the update function, as well as that of other components, can be found in the source code.

3 Extensibility of Ramulator

Ramulator’s extensibility is a natural result of its fully-decoupled design: Ramulator provides a generalized skeleton of DRAM (i.e., DRAM.h) that is capable of being infused with the specifics of an arbitrary DRAM standard (e.g., DDR3.h/cpp). To demonstrate the extensibility of Ramulator, we describe how easy it was to add support for DDR4: (i) copy DDR3.h/cpp to DDR4.h/cpp, (ii) add BankGroup as an item in DDR4::Level, and (iii) add or edit 20 entries in the lookup-tables — 1 in prerequisite, 2 in transition, and 17 in timing. Although there were some other changes that were also required (e.g., speed-bins), only tens of lines of code were modified in total — giving a general idea about the ease at which Ramulator is extended. As far as Ramulator is concerned, the difference between any two DRAM standards is simply a matter of the difference in their lookup-tables, whose entries are populated in a disciplined and localized manner. This is in contrast to existing simulators, which require the programmer to chase down each of the hardcoded for-loops and if-conditions that are likely scattered across the codebase.

In addition, Ramulator also provides a single, unified memory controller that is compatible with all of the standards that are supported by Ramulator (Table 2). Internally, the memory controller maintains three queues of memory requests: read, write, and maintenance. Whereas the read/write queues are populated by demand memory requests (read, write) generated by an external source of memory traffic, the maintenance queue is populated by other types of memory requests (refresh, power-down, self-refresh) generated internally by the memory controller as they are needed. To serve a memory request in any of the queues, the memory controller interacts with the tree of DRAM state-machines using the three functions described in Section 2.2 (i.e., decode, check, and update). The memory controller also supports several different scheduling policies that determine the priority between requests from different queues, as well as those from the same queue.

4 Validation & Evaluation

As a simulator for the memory controller and the DRAM system, Ramulator must be supplied with a stream of memory requests from an external source of memory traffic. For this purpose, Ramulator exposes a simple software interface that consists of two functions: one for receiving a request into the controller, and the other for returning a request after it has been served. To be precise, the second function is a callback that is bundled inside the request. Using this interface, Ramulator provides two different modes of operation: (i) standalone mode where it is fed a memory trace or an instruction trace, and (ii) integrated mode where it is fed memory requests from an execution-driven engine (e.g., gem5 [4]). In this section, we present the results from operating Ramulator in standalone-mode, where we validate its correctness (Section 4.1), compare its performance with other DRAM simulators (Section 4.2), and conduct a cross-sectional study of contemporary DRAM standards (Section 4.3). Directions for conducting the experiments are included in the source code release [1].

4.1 Validating the Correctness of Ramulator

Ramulator must simulate any given stream of memory requests using a legal sequence of DRAM commands, honoring the status transitions and the timing parameters of a standard (e.g. DDR3). To validate this behavior, we created a synthetic memory trace that would stress-test Ramulator under a wide variety of command interleavings. More specifically, the trace contains 10M memory requests, the majority of which are reads and writes (9:1 ratio) to a mixture of random and sequential addresses (10:1 ratio), and the minority of which are refreshes, power-downs, and self-refreshes. While this trace was fed into Ramulator as fast as possible (without overflowing the controller’s request buffer), we collected a timestamped log of every command that was issued by Ramulator. We then used this trace as part of an RTL simulation by feeding it into Micron’s DDR3 Verilog model [30] — a reference implementation of DDR3. Throughout the entire duration of the RTL simulation (~10 hours), no violations were ever reported, indicating that Ramulator’s DDR3 command sequence is indeed legal. Due to the lack of corresponding Verilog models, however, we could not employ the same methodology to validate other standards. Nevertheless, we are reasonably confident in their correctness, because we implemented them by making careful modifications to Ramulator’s DDR3 model, modifications that were expressed succinctly in just a few lines of code — minimizing the

We exclude maintenance-related requests which are not supported by Ramulator or other simulators: e.g. ZQ calibration and mode-register set.

This verifies that Ramulator does not issue commands too early. However, the Verilog model does not allow us to verify whether Ramulator issues commands too late.
risk of human error, as well as making it easy to double-check. In fact, the ease of validation is another advantage of Ramulator, arising from its clean and modular design.

### 4.2 Measuring the Performance of Ramulator

In Table 3, we quantitatively compare Ramulator with four other standalone simulators using the same experimental setup. All five were configured to simulate DDR3-1600⁶ for two different memory traces, Random and Stream, comprising 100M memory requests (read/write=9:1) to random and sequential addresses, respectively. For each simulator, Table 3 presents four metrics: (i) simulated clock cycles, (ii) simulation runtime, (iii) simulated request throughput, and (iv) maximum memory consumption. From the table, we make three observations. First, all five simulators yield roughly the same number of simulated clock cycles, where the slight discrepancies are caused by the differences in how their memory controllers make scheduling decisions (e.g., when to issue reads vs. writes). Second, Ramulator has the shortest simulation runtime (i.e., the highest simulated request throughput), taking only 752/249 seconds to simulate the two traces — a 2.5×/3.0× speedup compared to the next fastest simulator. Third, Ramulator consumes only a small amount of memory while it executes (2.1MB). We conclude that Ramulator provides superior performance and efficiency, as well as the greatest extensibility.

### 4.3 Cross-Sectional Study of DRAM Standards

With its integrated support for many different DRAM standards — some of which (e.g., LPDDR4, WIO2) have never been modeled before in academia — Ramulator unlocks the ability to perform a comparative study across them. In particular, we examine nine different standards (Table 4), whose configurations (e.g., timing) were set to reasonable values. Instead of memory traces, we collected instruction traces from 22 SPEC2006 benchmarks,⁸ which were fed into a simplistic “CPU” model that comes with Ramulator.⁷

<table>
<thead>
<tr>
<th>Simulator (clang -O3)</th>
<th>Cycles (10⁹)</th>
<th>Runtime (sec.)</th>
<th>Regesc (10⁹)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramulator</td>
<td>652 411</td>
<td>752 249</td>
<td>133 402</td>
<td>2.1</td>
</tr>
<tr>
<td>DRAMSim2</td>
<td>645 413</td>
<td>2,030 876</td>
<td>49 114</td>
<td>1.2</td>
</tr>
<tr>
<td>USM4</td>
<td>665 409</td>
<td>1,880 750</td>
<td>53 133</td>
<td>4.5</td>
</tr>
<tr>
<td>DrSim</td>
<td>647 406</td>
<td>8,110 12,984</td>
<td>6 8</td>
<td>1.6</td>
</tr>
<tr>
<td>NVMain</td>
<td>666 413</td>
<td>6,881 5,023</td>
<td>15 20</td>
<td>4,230.0</td>
</tr>
</tbody>
</table>

Table 3. Comparison of five simulators using two traces

Figure 2 contains the violin plots and geometric means of the normalized IPC compared to the DDR3 baseline. We measure several broad observations. First, newly upgraded standards (e.g., DDR4) perform better than their older counterparts (e.g., DDR3). Second, standards for embedded systems (i.e., LPDDR, WIO) have lower performance because they are optimized to consume less power. Third, standards for graphics systems (i.e., GDDR5, HBMO) provide a large amount of bandwidth, leading to higher average performance than DDR3 even for our non-graphics benchmarks. Fourth, a recent academic proposal, SALP, provides significant performance improvement (e.g., higher than that of WIO2) by reducing the serialization effects of bank conflicts without increasing peak bandwidth. These observations are only a small sampling of the analyses that are enabled by Ramulator.

### 5 Conclusion

In this paper, we introduced Ramulator, a fast and cycle-accurate simulation tool for current and future DRAM systems. We demonstrated Ramulator’s advantages in efficiency and extensibility, as well as its comprehensive support for DRAM standards. We hope that Ramulator would facilitate DRAM research in an era when main memory is undergoing rapid changes [23], [31].

**REFERENCES**


