3-2015

Ramulator: A Fast and Extensible DRAM Simulator

Yoongu Kim
Carnegie Mellon University

Weikun Yang
Peking University

Onur Mutlu
Carnegie Mellon University, onur@cmu.edu

Published In
Computer Architecture Letters, forthcoming.

Ramulator: A Fast and Extensible DRAM Simulator

Yoongu Kim[1] Weikun Yang[1,2] Onur Mutlu[1]
[1]Carnegie Mellon University [2]Peking University

Abstract—Recently, both industry and academia have proposed many different roadmaps for the future of DRAM. Consequently, there is a growing need for an *extensible* DRAM simulator, which can be easily modified to judge the merits of today's DRAM standards as well as those of tomorrow. In this paper, we present *Ramulator*, a fast and cycle-accurate DRAM simulator that is built from the ground up for extensibility. Unlike existing simulators, Ramulator is based on a generalized template for modeling a DRAM system, which is only later infused with the specific details of a DRAM standard. Thanks to such a decoupled and modular design, Ramulator is able to provide out-of-the-box support for a wide array of DRAM standards: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, as well as some academic proposals (SALP, AL-DRAM, TL-DRAM, RowClone, and SARP). Importantly, Ramulator does not sacrifice simulation speed to gain extensibility: according to our evaluations, Ramulator is 2.5× faster than the next fastest simulator. Ramulator is released under the permissive BSD license.

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]).

Segment	*DRAM Standards & Architectures*
Commodity	DDR3 (2007) [14]; DDR4 (2012) [18]
Low-Power	LPDDR3 (2012) [17]; LPDDR4 (2014) [20]
Graphics	GDDR5 (2009) [15]
Performance	eDRAM [28], [32]; RLDRAM3 (2011) [29]
3D-Stacked	WIO (2011) [16]; WIO2 (2014) [21]; MCDRAM (2015) [13]; HBM (2013) [19]; HMC1.0 (2013) [10]; HMC1.1 (2014) [11]
Academic	SBA/SSA (2010) [38]; Staged Reads (2012) [8]; RAIDR (2012) [27]; SALP (2012) [24]; TL-DRAM (2013) [26]; RowClone (2013) [37]; Half-DRAM (2014) [39]; Row-Buffer Decoupling (2014) [33]; SARP (2014) [6]; AL-DRAM (2015) [25]

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 one 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.

Type	*Simulator*	*DRAM Standards*
Standalone	DRAMSim2 (2011) [36]	DDR2, DDR3
	USIMM (2012) [7]	DDR3
	DrSim (2012) [22]	DDR2, DDR3, LPDDR2
	NVMain (2012) [34]	DDR3, LPDDR3, LPDDR4
Integrated	GPGPU-Sim (2009) [3]	GDDR3, GDDR5
	McSimA+ (2013) [2]	DDR3
	gem5 (2014) [9]	DDR3*, LPDDR3*, WIO*

*Not cycle-accurate [9].

Table 2. Survey of popular DRAM simulators

Manuscript submitted: 26-Nov-2014. Manuscript accepted: 22-Feb-2015.
Final manuscript received: 15-Mar-2015.

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 hardcoding 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 [14]. 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>*>
6      children;
7    T::Level level;
8    int index;
9    // more code...
10 };
```

```
1  // DDR3.h/cpp
2  class DDR3 {
3    enum class Level {
4      Channel, Rank, Bank,
5      Row, Column, MAX
6    };
7
8
9    // more code...
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.

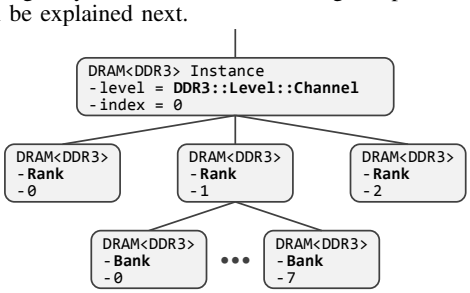

Figure 1. Tree of DDR3 state-machines

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.

1. `decode()`: The ultimate goal of a read request is to read from DRAM, which is accomplished by a read command. Depending on the status of the tree, however, it may not be possible to issue the read command: e.g., the rank is powered-down or the bank is closed. For a given command to a given address,[2] the decode function returns a "prerequisite" command that must be issued before it, if any exists: e.g., power-up or activate command.
2. `check()`: Even if there are no prerequisites, it doesn't mean that the read command can be issued right away: e.g., the bank may not be ready if it was activated just recently. For a given command to a given address, the check function returns whether or not the command can be issued right *now* (i.e., current cycle).
3. `update()`: If the check is passed, there is nothing preventing the memory controller from issuing the read command. For a given command to a given address, the update function triggers the necessary modifications to the status/horizon (of the affected nodes) to signify the command's issuance at the current cycle. In Ramulator, invoking the update function *is* issuing a command.

```
1  // DRAM.h
2  template <typename T>
3  class DRAM {
4    // states (queried/updated by functions below)
5    T::Status status;
6    long horizon[T::Command::MAX];
7    map<int, T::Status> leaf_status; // for bank only
8    // functions (recursively traverses down tree)
9    T::Command decode(T::Command cmd, int addr[]);
10   bool check(T::Command cmd, int addr[], long now);
11   void update(T::Command cmd, int addr[], long now);
12 };
```

```
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

2.3 A Closer Look at a State-Machine

So far, we have described the role of the three functions without describing how they exactly perform their role. To preserve the standard-agnostic nature of the DRAM class, the three functions defer most of their work to the DDR3 class, which supplies them with all of the standard-dependent information in the form of three lookup-tables: *(i) prerequisite*, *(ii) timing*, and *(iii) transition*. Within these tables are encoded the DDR3 standard, providing answers to the following three questions: *(i)* which commands must be preceded by which other commands at which levels/statuses? *(ii)* which timing parameters at which levels apply between which commands? *(iii)* which commands to which levels trigger which status transitions?

Decode. Due to space limitations, we cannot go into detail about all three lookup-tables. However, Code 3 (bottom) does provide a glimpse of only the first lookup-table, called *prerequisite*, which is consulted inside the decode function as shown in Code 3 (top). In brief, prerequisite is a two-dimensional array of *lambdas* (a C++11 construct), which is indexed using the *(i)* level in the hierarchy at which the *(ii)* command is being decoded. As a concrete example, Code 3 (bottom, lines 7–13) shows how one of its entries is defined, which happens to be for *(i)* the rank-level and *(ii)* the refresh command. The entry is a lambda, whose sole argument is a pointer to the rank-level node that is trying to decode the refresh command. If any of the node's children (i.e., banks) are open, the lambda returns the precharge-all command (i.e., PREA, line 11), which would close all the banks and pave the way for a subsequent refresh command. Otherwise, the lambda returns the refresh command itself (i.e., REF, line 12), signaling that no other command need be issued before it. Either way, the command has been successfully decoded at that particular level, and there is no need to recurse further down the tree. However, that may not always be the case. For example, the only reason why the rank-level node was asked to decode the refresh command was because its parent (i.e., channel) did not have enough

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

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).

```
// DRAM.h
template <typename T>
class DRAM {
  T::Command decode(T::Command cmd, int addr[]) {
    if (prereq[level][cmd]) {
      // consult lookup-table to decode command
      T::Command p = prereq[level][cmd](this);
      if (p != T::Command::MAX)
        return p; // decoded successfully
    }
    if (children.size() == 0) // lowest-level
      return cmd; // decoded successfully
    // use addr[] to identify target child...
    // invoke decode() at the target child...
  }
};
```

```
// DDR3.h/cpp
class DDR3 {
  // declare 2D lookup-table of lambdas
  function<Command(DRAM<DDR3>*)>
    prereq[Level::MAX][Command::MAX];
  // populate an entry in the table
  prereq[Level::Rank][Command::REF] =
    [] (DRAM<DDR3>* node) -> Command {
      for (auto bank : node->children)
        if (bank->status == Status::Open)
            return Command::PREA;
      return Command::REF;
    };
  // populate other entries...
};
```

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]` $\leq$ `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`, `powerdown`, `selfrefresh`) 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 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.[3] 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 ($\sim$10 hours), no violations were ever reported, indicating that Ramulator's DDR3 command sequence is indeed legal.[4] 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

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

[4]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 DDR3-1600[5] 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.

Simulator (clang -O3)	*Cycles (10^6)*		*Runtime (sec.)*		*Req/sec (10^3)*		*Memory (MB)*
	Random	*Stream*	*Random*	*Stream*	*Random*	*Stream*	
Ramulator	652	411	752	249	133	402	2.1
DRAMSim2	645	413	2,030	876	49	114	1.2
USIMM	661	409	1,880	750	53	133	4.5
DrSim	647	406	18,109	12,984	6	8	1.6
NVMain	666	413	6,881	5,023	15	20	4,230.0

Table 3. Comparison of five simulators using two traces

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,[6] which were fed into a simplistic "CPU" model that comes with Ramulator.[7]

Standard	*Rate (MT/s)*	*Timing (CL-RCD-RP)*	*Data-Bus (Width×Chan.)*	*Rank-per-Chan*	*BW (GB/s)*
DDR3	1,600	11-11-11	64-bit × 1	1	11.9
DDR4	2,400	16-16-16	64-bit × 1	1	17.9
SALP†	1,600	11-11-11	64-bit × 1	1	11.9
LPDDR3	1,600	12-15-15	64-bit × 1	1	11.9
LPDDR4	2,400	22-22-22	32-bit × 2*	1	17.9
GDDR5 [12]	6,000	18-18-18	64-bit × 1	1	44.7
HBM	1,000	7-7-7	128-bit × 8*	1	119.2
WIO	266	7-7-7	128-bit × 4*	1	15.9
WIO2	1,066	9-10-10	128-bit × 8*	1	127.2

†*MASA [24] on top of DDR3 with 8 subarrays-per-bank.*
**More than one channel is built into these particular standards.*

Table 4. Configuration of nine DRAM standards used in study

Figure 2 contains the violin plots and geometric means of the normalized IPC compared to the DDR3 baseline. We make 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., LPDDRx, WIOx) have lower performance because they are optimized to consume less power. Third, standards for graphics systems (i.e., GDDR5, HBM) 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.

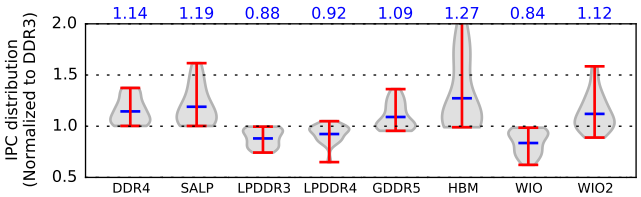

Figure 2. Performance comparison of DRAM standards

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 advantage 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].

[5]Single rank, 800Mhz, 11-11-11, row-interleaved, FR-FCFS [35], open-row policy.

[6]perlbench, bwaves, gamess, povray, calculix, tonto were unavailable for trace collection.

[7]3.2GHz, 4-wide issue, 128-entry ROB, no instruction-dependency, one cycle for non-DRAM instructions, instruction trace is pre-filtered through a 512KB cache, memory controller has 32/32 entries in its read/write request buffers.

References

[1] Ramulator source code. https://github.com/CMU-SAFARI/ramulator.
[2] J. H. Ahn et al. McSimA+: A Manycore Simulator with Application-Level+ Simulation and Detailed Microarchitecture Modeling. In *ISPASS*, 2013.
[3] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[4] N. Binkert et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, May 2011.
[5] K. Chandrasekar et al. DRAMPower: Open-Source DRAM Power & Energy Estimation Tool. http://www.drampower.info, 2012.
[6] K. Chang et al. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.
[7] N. Chatterjee et al. USIMM: the Utah SImulated Memory Module. *UUCS-12-002, University of Utah*, Feb. 2012.
[8] N. Chatterjee et al. Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads. In *HPCA*, 2012.
[9] A. Hansson et al. Simulating DRAM Controllers for Future System Architecture Exploration. In *ISPASS*, 2014.
[10] Hybrid Memory Cube Consortium. *HMC Specification 1.0*, Jan. 2013.
[11] Hybrid Memory Cube Consortium. *HMC Specification 1.1*, Feb. 2014.
[12] Hynix. *GDDR5 SGRAM H5GQ1H24AFR*, Nov. 2009.
[13] James Reinders. Knights Corner: Your Path to Knights Landing, Sept. 17, 2014.
[14] JEDEC. *JESD79-3 DDR3 SDRAM Standard*, June 2007.
[15] JEDEC. *JESD212 GDDR5 SGRAM*, Dec. 2009.
[16] JEDEC. *JESD229 Wide I/O Single Data Rate (Wide/IO SDR)*, Dec. 2011.
[17] JEDEC. *JESD209-3 Low Power Double Data Rate 3 (LPDDR3)*, May 2012.
[18] JEDEC. *JESD79-4 DDR4 SDRAM*, Sept. 2012.
[19] JEDEC. *JESD235 High Bandwidth Memory (HBM) DRAM*, Oct. 2013.
[20] JEDEC. *JESD209-4 Low Power Double Data Rate 3 (LPDDR4)*, Aug. 2014.
[21] JEDEC. *JESD229-2 Wide I/O 2 (WideIO2)*, Aug. 2014.
[22] M. K. Jeong et al. DrSim: A Platform for Flexible DRAM System Research. http://lph.ece.utexas.edu/public/DrSim, 2012.
[23] U. Kang et al. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum (Co-located with ISCA)*, 2014.
[24] Y. Kim et al. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.
[25] D. Lee et al. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*, 2015.
[26] D. Lee et al. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.
[27] J. Liu et al. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.
[28] M. Meterelliyoz et al. 2nd Generation Embedded DRAM with 4X Lower Self Refresh Power in 22nm Tri-Gate CMOS Technology. In *VLSI Symposium*, 2014.
[29] Micron. Micron Announces Sample Availability for Its Third-Generation RLDRAM(R) Memory. http://investors.micron.com/releasedetail.cfm?ReleaseID=581168, May 26, 2011.
[30] Micron. DDR3 SDRAM Verilog model, 2012.
[31] O. Mutlu. Memory Scaling: A Systems Architecture Perspective. MemCon, 2013.
[32] S. Narasimha et al. 22nm High-Performance SOI Technology Featuring Dual-Embedded Stressors, Epi-Plate High-K Deep-Trench Embedded DRAM and Self-Aligned Via 15LM BEOL. In *IEDM*, 2012.
[33] S. O et al. Row-Buffer Decoupling: A Case for Low-latency DRAM Microarchitecture. In *ISCA*, 2014.
[34] M. Poremba and Y. Xie. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *ISVLSI*, 2012.
[35] S. Rixner et al. Memory Access Scheduling. In *ISCA*, 2000.
[36] P. Rosenfeld et al. DRAMSim2: A Cycle Accurate Memory System Simulator. *CAL*, 2011.
[37] V. Seshadri et al. RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data. In *MICRO*, 2013.
[38] A. N. Udipi et al. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *ISCA*, 2010.
[39] T. Zhang et al. Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation. In *ISCA*, 2014.