Journal of Computer Science

Original Research Paper

DDRSHARP: A Fast and Extensible DRAM Simulator

[1]Tesfamichael Gebregziabher Gebrehiwot, [1]Fitsum Assamnew Andargie and [2]Mohammed Ismail

[1]School of Electrical and Computer Engineering, Addis Ababa Institute of Technology, Addis Ababa, Ethiopia
[2]Department of Electronics and Communication Engineering, Sasi Institute of Technology and Engineering, Andhra Pradesh, India

Article history
Received: 21-02-2023
Revised: 18-04-2023
Accepted: 01-06-2023

Corresponding Author:
Tesfamichael Gebregziabher Gebrehiwot
School of Electrical and Computer Engineering, Addis Ababa Institute of Technology, Addis Ababa, Ethiopia
Email: tesfamichael.gegziabher@aait.edu.et

Abstract: Dynamic Random-Access Memory (DRAM) is a crucial component of modern computing systems and there exist a variety of DRAM standards. The variance in DRAM architecture calls for an extensible simulator to accommodate current and future DRAM standards. Moreover, as every researcher may not be an expert in programming, choosing an easier programming language to construct a simulator would reduce the efforts of a researcher who seeks to reuse/modify existing code to meet the demands of his/her work. Performance bottleneck is another challenge of cycle-accurate simulators; some researchers even suggest statistical modeling to achieve higher speed by sacrificing accuracy. We present DDRSHARP, a cycle-accurate DRAM simulator written entirely in C#. It provides simplified configuration and evaluation of different DRAM standards. It includes both the performance and power/energy models of DRAM. In order to improve the performance of DDRSHARP, we skipped infeasible iterations on queued requests, minimized the number of branch instructions, saved repetitive calculations for later use, and minimized code execution paths. Since our simulator is constructed using C# and the garbage collector consumes a big amount of CPU time, we worked on minimizing heap allocation of immutable objects such as strings. Our approach has enabled more than 1.8 times speedup in performance compared to contemporary simulators.

Keywords: DRAM, CPU, Simulation, Modeling

Introduction

Dynamic Random Access Memory (DRAM) has become the primary choice of memory in modern computer systems. Improvement of DRAM standards and technology, and decreasing prices are the major reasons for its wide adoption. As computations are becoming data-intensive, there is also a drive for more and more fast and efficient memory. In order to satisfy these requirements, researchers use DRAM simulators to study the performance and energy consumption of new DRAM designs. There exist Several DRAM simulators (Alakarhu and Niittylahti, 2002; Binkert *et al*., 2011; Chatterjee *et al*., 2012; Healy and Hong, 2017; Kim *et al*., 2015; Rosenfeld *et al*., 2011; Mirosanlou *et al*., 2020; Steiner *et al*., 2020) and most of these simulators are amalgamated with CPU simulators and majority of them are designed to be cycle accurate to ensure timing accuracy.

Simulation performance has been the driving factor behind many simulators. Thus, most developers worked hard to outperform previously constructed simulators. Some of them even advocate the importance of sacrificing accuracy for speed. Li *et al*. (2019) criticize the importance of cycle-accurate simulation and proposed statistical DRAM modeling which sacrifices a 2% error for a 400 times speedup gain (Li and Jacob, 2019). However, this approach may not be applicable to situations where 100% accuracy is required. Hansson *et al*. (2014) proposed an event-based memory model and achieved 7 times the performance speed. However, they implemented a few DRAM timing parameters that they regard as most important and ignore to enforce the remaining timing parameters. They claim to have maintained the accuracy of the simulation. However, Li *et al*. (2019) compared the accuracy of this event-based memory model (Hansson *et al*., 2014) with a cycle-accurate simulator, DRAMsim3 (Li *et al*., 2020). The study reveals, without enforcing all timing parameters, it is difficult to claim that such an approach would always provide correct simulation for all workloads.

Besides the performance and accuracy of simulation, we should also consider simplicity, extensibility, scalability, and code productivity when designing a new simulator. A simulator model should be extensible enough to include both existing DRAM and future standards. For example, DRAMSim2 (Rosenfeld *et al*., 2011) supports a few DRAM standards (DDR2 and DDR3). Similarly, USIMM (Chatterjee *et al*., 2012) is limited to DDR3.

Ramulator (Kim *et al*., 2015), DRAMsys (Steiner *et al*., 2020), and DRAMsim3 (Li *et al*., 2020) support DDR2/DDR3/DDR4 standards. Ramulator can also be extended to support many academic memory models.

If an extensible simulator could not handle all future DRAM standards, a future researcher is forced to customize an existing simulator or design a new simulator from scratch. The time and effort needed to customize an existing software depends on the architecture of the software. Software components should be loosely coupled for a simplified customization process.

We propose DDRSHARP, a cycle-accurate DRAM simulator that models both the performance and power/energy parameters of DRAM. We followed an object-oriented design methodology to decompose basic components of the memory system into minimally coupled objects. It enables run-time configuration of different DRAM specifications; avoids the hard-coding of the parameters of different DRAM standards to simplify the evaluation of various memory systems. Our contribution includes 1) a high-performance DRAM simulator 2). Extensible simulator that can be configured to support many DRAM standards.

We chose C# to construct the simulator to exploit the features of automatic memory management and minimize the chance of memory leaks. As compared to C/C++, C# helps you reduce programming errors by ensuring code safety, performing reference checking, performing automatic garbage collection, and ensuring code security by preventing buffer overflows. These features, however, come at a considerable performance penalty and could significantly degrade the speed of the simulator.

Having studied the coding techniques of contemporary simulators, we applied some code optimization techniques in order to improve the performance of the simulation. We employed a smart iteration technique that skips unnecessary iterations on queued requests; that is all requests queued to a busy bank are skipped. We also reduced the number of branch instructions in the simulator to avoid the penalty of CPU's branch mispredictions. We pre-computed and saved some computations which are implemented within frequently called functions. We also worked on minimizing the path of code executions as much as possible.

Since our simulator is constructed using C# and the Garbage Collector (GC) consumes a big amount of CPU time when activated, we worked on minimizing heap allocation of immutable objects such as strings. In order to prevent multiple allocations and avoid performance degradation, strings are allocated on the stack, and manipulation is done by directly accessing slices of the data; Sub-portion (s) of the data are not allocated on the heap. A combination of all these optimizations has enabled DDRShap to achieve at least 1.8 times speedup over existing simulators.

DDRSHARP models both the CPU and the memory subsystem. The CPU models consist of the CPU core, ROB (reorder buffer), and the MSHR (miss status holding register). The memory subsystem models all DRAM components. Some of which are the channel, rank, bank group, and bank objects. DDRSHARP accepts both CPU and memory traces that are sent to the memory controller object by the CPU core or the memory system. DDRSHARP uses a memory controller which does not contain complex logic. It simply delegates basic functionalities to other dependencies; dependencies such as refresher, scheduler, queue manager and read/write mode selector.

The queue manager keeps track of every read and every write request. If DDRSHARP is in read mode, the read queue is served otherwise the write queue is served. The transition from read mode to write mode or vice versa is performed by the Read/Write mode selector and is decided at every memory cycle as per chosen policy. According to the selected mode, the scheduler selects the best command to be issued from either the read queue or the write queue. The scheduler uses the scheduling policy, the timing constraints, and the DRAM state to pick a command to issue.

DRAM Background

DRAM is hierarchically organized into channels, ranks, bank groups, and banks. A channel is connected to one or more ranks; a rank is a 64-bit wide module that contains a set of DRAM chips. A DRAM is configured as ×4, ×8, or ×16. In τηε ×8 configurations, 8 physical chips each with a bit-width of 8 (×8) are connected together as shown in Fig. 1. Other configurations include, ×4 configurations (16 chips) or ×16 (4 chips). All ranks work independently. However, full parallelism is limited as all ranks connected to the same channel share the same data lines.

Each chip contains several banks which are grouped into one or more bank groups. A typical DDR5 rank contains 32 banks. Fig. 2 shows the functional block diagram of DDR4 SDRAM; in this configuration, a total of 16 banks are grouped into 4 bank groups.

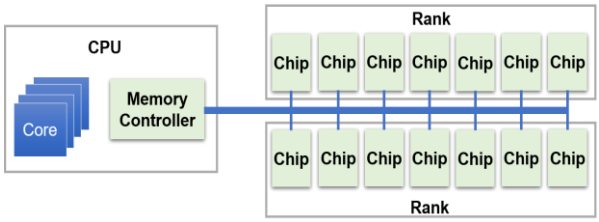

Fig. 1: Memory hierarchy: ×8 configuration

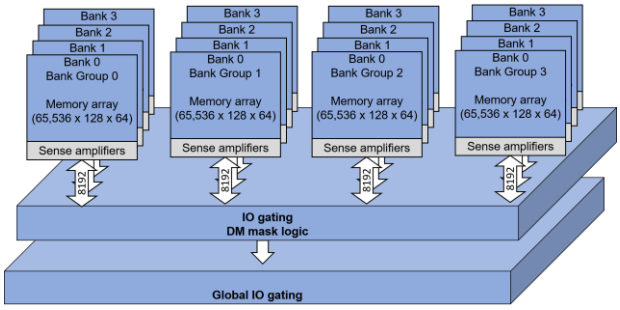

Fig. 2: DDR4 SDRAM functional block diagram

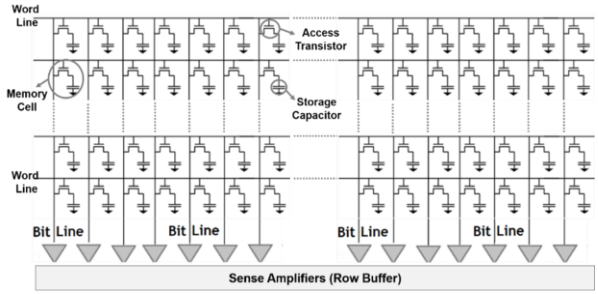

Fig. 3: DRAM Bank structure

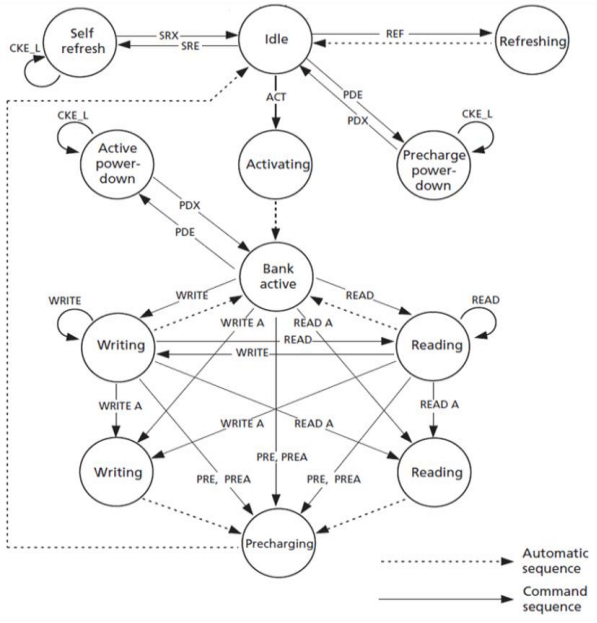

Fig. 4: Simplified State Diagram of SDRAM (reproduced from JEDEC (2022))

In a typical DRAM configuration, a bank contains 32 subarrays where each subarray is composed of 32 MATs (multiple cell matrices). Each MAT contains 262,144 memory cells (512 rows and 512 columns. A memory cell is composed of an access transistor and a capacitor as shown in Fig. 3. The capacitor of the memory cell stores a single bit. The transistor serves as a switch that connects/disconnects the capacitor to/from the bit line. Each bit line is connected to a set of sense amplifiers (row buffer). Activating a word line enables access to all memory cells of a row; this copies the contents of a row to the row buffer. The row buffer keeps recently accessed rows.

Subsequent read requests for data located on the same row are served from the row buffer. Otherwise, before any new read/write operation, an open bank must be closed before activating a new row; a bank is called open if it contains a row that is open in the row buffer. Closing a bank involves rewriting back the contents of the row buffer back to the target row. This restores the lost charges of the capacitors during the sensing process to the original level.

Modeling DRAM

Vendors provide data sheets (JEDEC, 2022) that contain the specifications of a given DRAM model. The design of the performance/power model of DRAM requires an understanding of these architectural, timing, and power parameters. The timing parameters are constraints that need to be satisfied before a command is allowed to be issued. Similarly, the power parameters determine the current needed to execute a DRAM command. In this section, we also discuss DRAM parameters with respect to performance/power model design.

DRAM Commands and Memory States

The memory states and the commands that dictate state transitions will be discussed in this section. The simplified state diagram shown in Fig. 4 shows is based on the default read/write burst length of 16 (BL16). However, DDR5 (JEDEC, 2022) provides a 32-bit burst read/write operation (BL32 mode) for ×4 devices only. The DRAM could be, at any given time, in one of the valid memory states; it could be in a power down state, standby state, pre-charging, activating, refreshing, reading, or writing. DRAM enters power down mode when the Clock Enables signal (CKE), which enables output drivers and other components such as input buffers and internal clocks signals, is set to low. The PDE (power down entry) command sends the device to PDN (power down mode). With CKE set to low, a bank transitions from the active state to active power down; it could also transition to a precharge power down state from an idle.

The memory leaves the power down state when a PDX command is issued. The bank may return to an idle state or active state depending on whether it contains an active row. As shown in Fig. 4, a bank can in an idle state directly transition to a refreshing state or activating state via ACT and REF commands respectively. When the CKE is set to low, it can also transition to a self-refresh state via SRE command; during this period the DRAM can perform self-refreshes without the help of the memory controller and the processor. The memory exits a self-refresh state using the SRX command.

When the memory is in activating state, it can only transition to an active state after a period of time specified by RCD is elapsed. Once it is in an active state, however, it can transition to a reading state using the RD command; it can also transition to a writing state using the WR command. If RDA/WRA (RD/WR then auto precharge) command is issued, the bank transitions automatically to a precharging state after a successful read/write; it then returns to an idle state at the end of the PRE (precharge) command.

Address bits that identify the bank and row to be accessed are registered along with the ACT command JEDEC (2021). The starting column location for the burst operation is also determined by the address bits registered along with the RD or WR command. Moreover, these bits determine whether the auto precharge command is to be issued after a successful RD/WR command.

When decoding a read/write request, a single memory access can trigger the issuance of one or more additional DRAM commands depending on the device's current state. For example, a read request can be translated to RD command if the row to be accessed is already active. It could also be translated into a sequence of ACT and RD commands if the bank is in an idle state. However, if a different row is open, a sequence of PRE, ACT, and RD commands are executed.

Performance Model

Understanding the timing specification is crucial in designing the performance model of DRAM as it defines the latency of every DRAM command that is associated with any read/write memory request. A brief description of key timing parameters is shown in Table 1.

Before any row activation (issuing ACT command), the rank checks that no more than four activations can be performed within the time frame specified by FAW; the request for activation during a FAW window is denied if four activations are already performed. The rank also checks if the time gap between any two consecutive activations obeys the RRD timing constraint. On the other hand, the activation gap between two activations within a bank is limited by the RC time constraint. Usually, RRD is shorter than RC. This parameter is checked at the rank to keep track of the activations of all banks within the rank.

The read process requires moving column bits into the bus. First, a row that contains the data is activated using ACT followed by data sense during which the contents of the row are copied to the row buffer. The time between the issuance of the ACT command and the availability of data on the row buffer is specified by the timing parameter RCD.

The time needed to make the first bit available on the bus after the data is copied to the row buffer is specified by the CAS/CL parameter. The CWD parameter defines the same constraint for a write command. The CCD parameter defines the time gap between any two consecutive column accesses for both read and write operations. The maximum number of column accesses is limited by the burst length of the device. The time gap between the end of a write burst and the start of a read operation is specified by the WTR parameter.

A READ command, as shown in Fig. 5 uses the column address and bank/group address to copy the data to the DQ (data) pins. This operation requires a time of CL (column latency) cycles. For the 16-bit read burst operation (BL16) shown in Fig. 5, D_n (0, D_1, D_2,...D_{13}, D_{14}, D_{15}) represents the data-out from column n. the figure shows the DES (Device DESelect, CS# false) for the purpose of simplicity.

As charges stored in capacitors may leak or decrease due to other reasons, the DRAM needs to be periodically refreshed at specific time intervals. The task of refreshing a memory involves reading and rewriting the contents of a memory cell. The REFI parameter specifies the interval between two refresh commands; a time specified by the RFC parameter specifies the time needed to complete the task of refreshing memory cells. Before a refresh operation starts, open banks are closed using the PRE command. This operation needs a time equivalent to RP (row precharge).

Table 1. Description of key timing parameters (reproduced from JEDEC (2022))

Parameter	Explanation
FAW	Four activation window time limit between the first and fourth activation of any four consecutive activation
RCD	Row to column delay time needed to move data from DRAM cells (row) to the row buffer
RAS	Row access strobe minimum time gap between activate command and precharge command
RP	Row precharge time needed to close/precharge a bank for next-row accesses
RC	Row cycle time gap between two-row activation (same bank, different rows)
CCD	Column-to-column delay time needed for column read; bursts are used for multiple column reads
CAS	Column access strobe time between issuance of column access command and availability of the first bit
RFC	Refresh cycle time gap between the refresh command and the next command
WTR	Write to read time between the end of write and the start of a read command
CWD	Column write delay time between issuance of the write command and placement of data on the bus
REFI	Refresh interval time interval between refresh commands
RRD	Row-to-row delay time interval between activities that are not in the same bank

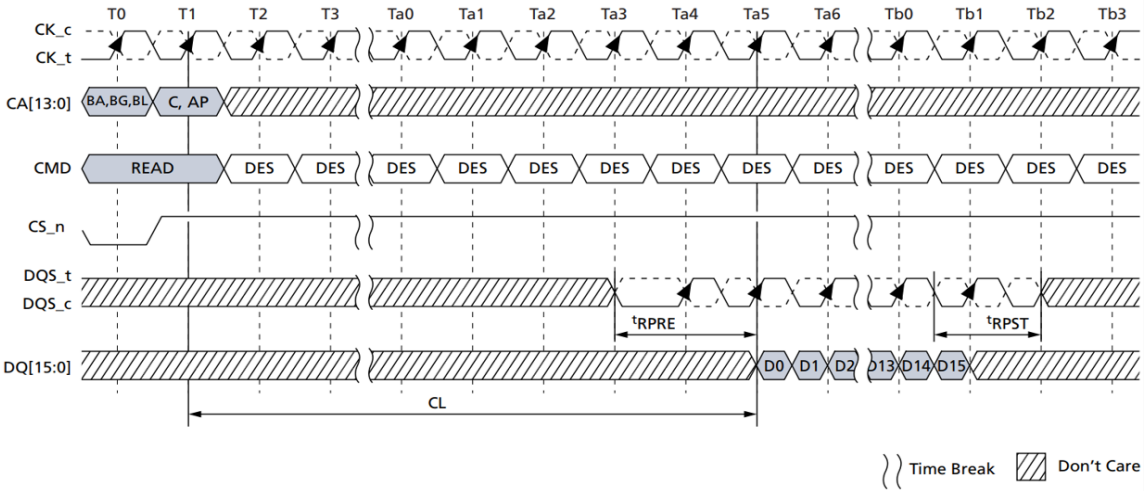

Fig. 5: READ burst operation (BL16)

During a refresh or any read/write operation, the minimum time gap between ACT and PRE commands should not be violated. This gap is specified by the RAS (row access strobe) timing parameter. In DRAM devices that contain more than one bank group, some timing parameters have both a longer and shorter version. For example, CCD is defined as CCDL (long) and CCDS (short); RRDL/RRDS and WTRL/WTRS are also timing parameters with 2 versions. The longer version specifies the duration between consecutive commands which are executed on the same bank group; otherwise, the timing is determined by the shortest version.

Power Model

We use both the power parameters and timing parameters of a given memory to design the power model of DRAM. We employ Micron (2017b) power calculator to determine the maximum power consumption for basic DRAM operations and the background power when DRAM is in standby or power-down mode. Table 2 describes basic DRAM power parameters. If the memory is in power-down mode, the active current and precharge current are IDD_{3P} and IDD_{2P} respectively. Otherwise, IDD_{2N} and IDD_{3N} are background currents.

Table 2: Description of key current parameters (reproduced from Micron (2020))

Current	Explanation
IDD_0	Average activate-to-precharge current
IDD_{2N}	Precharge standby current
IDD_{2P}	Precharge power-down current
IDD_{3N}	Active standby current
IDD_{3P}	Active power-down current
IDD_{4R}	Burst read current
IDD_{4W}	Burst writes current
IDD_{5R}	Distributed refresh current (1X REF)

For example, the current due to activate command (over a period of t_{RC}) is calculated by subtracting background currents; IDD_{3N} (over a period of t_{RAS}) and IDD_{2N} (over a period of t_{RC}-t_{RAS}) as shown in Eq. 1. Similarly, for RD, WR, and REF commands, IDD_{3N} is deducted from IDD_{4R}, IDD_{4W}, and IDD_5, respectively. List of all equations used in our power model:

$$I_{ACT} = I_{DD0} - \frac{IDD_{3N}t_{RAS} + IDD_{2N}\left(t_{RC} - t_{RAS}\right)}{tRC} \quad (1)$$

DDRSHARP

DDRSHARP, which is implemented in C#, models both the performance and power of DRAM devices. It is composed of two projects; the simulation library and the User Interface (UI). The UI component enables the user to select and set up the parameter of a DRAM, CPU, or Memory Controller. Configuration files can be added to a folder so as to avoid the laborious hard coding of different configurations of many standards. At the core of the simulation library is the simulator object which is responsible for the construction of the Memory System and the CPU core. As an input, DDRSHARP accepts DRAM trace as well as cache-filtered CPU trace. The implementation of the trace reader and address translator (address mapping) is injected into the Simulator object. DRAM traces are managed by the Memory System while the Core object of the CPU takes the same responsibility of fetching and dispatching of cache filtered traces to the memory controller. Once a trace record is decoded into a valid memory request, a delegate function that will be executed upon successful completion of the request is attached to it. Fig. 6 shows basic DDRSHARP components.

For simplicity and maintainability, the memory controller does not contain complex logic. Key

functionalities such as scheduling, refreshing, queuing system, and read/write mode switching are not implemented by the controller. They are implemented by dependencies that are injected into the memory controller. In case the user fails to select a scheduling policy or any other option, DDRSHARP implements default settings. At every cycle, the memory controller delegates the task of scheduling, refreshing, read mode, or write mode switching to the injected dependencies. If a refresher determines a REF command should be issued, the REF command is given priority over other commands. Otherwise, the read/write mode selector decides if the DRAM should be in read mode or write mode. If the simulator is in read mode, the scheduler selects the best request from the read queue; else, the best request is selected from the write queue. The criteria for selecting the best request is determined by the implemented scheduling policy. If the selected request fails to meet the timing constraints or if the device, in its current state, is not ready for this request, the next best is selected. Once the best request is selected and the state of the target bank is active, the active row is compared with the row address of the request. If they are equal (row hit), the RD or WR command is issued based on the request type. Otherwise, if a row conflict happens, a PRE command is issued to close the bank and the request remains in the queue. However, if the target bank is idle (already closed bank), the ACT command is issued to activate the row address of the request and the request remains in the queue. A request remains in a queue until an RD/RDA or WR/WRA command associated with that request is issued.

Enforcing Timing Constraints

DDRSHARP updates the state of the target bank and computes the minimum time the system should wait before transitioning to the next state. When a command is successfully issued, the state of the bank is updated to one of the valid states shown in Fig. 4. A bank object maintains four variables; the NextRD, NextWR, NextACT, and NextPRE variables store the minimum amount of time that the bank should wait before issuing an RD, WR, ACT or a PRE command. Similarly, a countdown variable holds the number of cycles required before the state of the bank can transition to a state specified by the NextSate variable. In other words, it holds the time needed for the issued command to complete. At every cycle, the value of the countdown variable is decremented. When the value of the countdown reaches zero, the state of the bank is changed to the next valid state.

Statistical data for every issued command is collected by the bank, rank, and channel objects. Every command is implemented differently on the Channel, Rank, and BankGroup classes. Every memory component follows its own implementation as per DRAM specification.

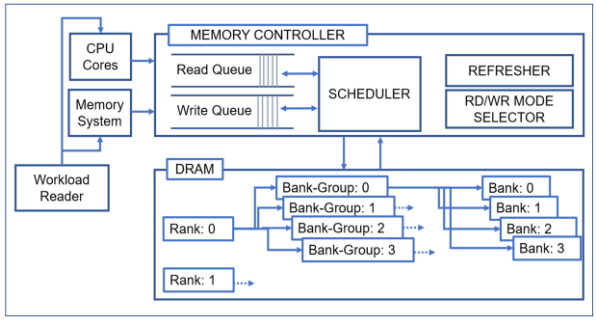

Fig. 6: DDRSHARP components

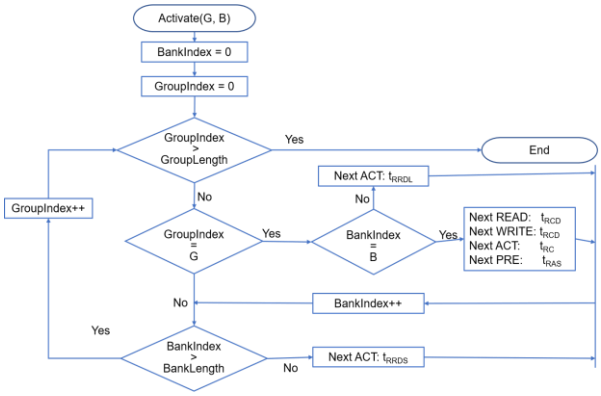

Fig. 7: Conceptual flowchart showing how timing constraints are updated. This does not reflect the actual implementation

For example, before the executing ACT command, the activate function has to pass several layers of validation checks. First, the bank must be idle; secondly, the rank should confirm the time constraint specified by the FAW parameter is obeyed; the rank also checks if the RRDS constraint is fulfilled. The bank group that contains the bank also validates if the RRDL constraint is not violated. Finally, the bank itself should check if the Row Cycle (RC) time limit is not broken. Figure 7 provides a conceptual overview.

As DDRSHARP follows an objected oriented design methodology, the channel, the rank, the bank groups, and the bank are designed to be separate objects. Each object manages its own state and timing constraints independently. Hence, once a scheduler selects a command, the command needs to be propagated from the channel of the memory controller all the way to the target bank. The propagation of command execution through the channel, rank, bank group, and bank are described below:

1. The scheduler selects a request to be issued and calls a method on the memory controller which in turn calls an appropriate method on the channel. The called function in the channel iterates through all of

its ranks and calls a proper function on the target rank (the rank that contains the address of the request) and its siblings (all other ranks that do not match the requested address)

2. The target rank and sibling ranks update their time constraints differently as per DRAM specifications. Moreover, the target rank decodes the address of the request and calls the appropriate method on the target bank-group object. The target bank group object, in turn, calls a function on the target bank
3. Upon completion of the read/write task, the delegate function attached to the request is executed to notify the CPU (for CPU traces) or the memory system (for DRAM traces)

Computing Power Consumption

The power and energy consumption are calculated using collected data and power parameters of the DRAM. Some of the power equations that we used, based on the Micron power calculator (Micron, 2017b) to estimate the power consumption of read, write, activate, or refresh operations are shown below:

$$P_{RD} = \left(I_{DD4R} - I_{DD3N}\right) \cdot V_{DD} RDCP \tag{2}$$

where:

- *RDCP* is the percent of read cycles

$$P_{WR} = \left(I_{IDD4W} - I_{IDD3N}\right) \cdot V_{DD} \cdot WRCP \tag{3}$$

where:

- *WRCP* is the percent of write cycles

$$P_{ACT} = \left(IDD_0 - \frac{IDD_{3N} t_{RAS}}{t_{RC}} + \frac{IDD_{2N}\left(t_{RC} - t_{RAS}\right)}{t_{RC}} \right) . V_{DD} \tag{4}$$

$$P_{REF} = \left(I_{IDD5} - I_{IDD3N}\right) \cdot V_{DD} \cdot \frac{t_{RFC}}{t_{REFI}} \tag{5}$$

At any moment in time, the memory can enter a power-down mode by setting the CKE to low. During this period of time, the current that causes power dissipation is IDD_{2P} if all banks are closed. If there exists a single bank that is open, however, the power consumption is due to IDD_{3P}, the active standby current. Equations 6-7 show the power-down background power:

$$PDN_{PRE} = I_{DD2P} \cdot V_{DD} \cdot PPC \tag{6}$$

where:

- PDN_{PRE} is the precharge power in down mode
- *PPC* is the percent of precharge cycles in down mode

$$PDN_{ACT} = I_{DD3P} \cdot V_{DD} \cdot PAC \tag{7}$$

where:

- PDN_{ACT} is the activation power in down mode
- *PAC* is the percent of active cycles in down mode

When the CKE is high, the memory to standby mode and the background power consumption is by I_{DD2N} if all banks are closed and I_{DD3N} otherwise:

$$PSTBY_{PRE} = I_{IDD2N} \cdot V_{DD} \cdot SPC \tag{8}$$

where:

- $PSTBY_{PRE}$ is the precharge power in standby mode
- *SPC* is the percent of the precharge cycle in standby mode

$$PSTBY_{ACT} = I_{DD3N} \cdot V_{DD} \cdot SAC \tag{9}$$

where:

- $PSTBY_{ACT}$ is the active power in standby mode
- *SAC* is the percent of active cycle in standby mode

The total power is the sum of all the aforementioned consumptions plus I/O power. The I/O power, which depends on the density and form factor of the system, includes the output driver or ODT power, the power needed to drive the bus, and the power consumed terminating a write operation. It also includes the power used to terminate a write/read operation to/from another rank.

Code Optimization

DDRSHARP is constructed using managed code namely, C#. We chose C# to delegate the responsibility of garbage collection, code safety, reference checking, and code security. The GC, garbage collector, however, consumes a big portion of CPU time. To compensate for the performance penalty that is brought by GC and other features, we followed a profile-guided optimization technique to pinpoint key code blocks that contribute to a very high-performance bottleneck. The top 4 sources of latency and the optimization technique that we implemented are listed below.

Zero copying: Sometimes, operation on some portion of an object triggers the creation of a new temporary object (s). A good example is a string object. In C# strings are immutable; code blocks that allocate and manipulate strings will end up having extra string object (s) allocated on the heap which puts an extra burden on the garbage collector. We used a stack-allocated reference window to manipulate immutable objects; it allows us to work on slices of the data without copying. Moreover, Span¡char¿ is allocated on the stack, and values are deleted by default when the function goes out of scope; this relieves the garbage collector.

Pre-computation: Whenever a command is issued, timing constraints are updated; some simulators repeatedly perform unnecessary computations. For example, after issuing an RD command, USIM (Chatterjee *et al*., 2012) updates the time constraint for the next write as Cycle + T_{CAS} + T_{BL} + T_{RTRS} -T_{CWL}. However, except for the Cycle variable which is incremented at every cycle, the sum of constant values (T_{CAS} + T_{BL} + T_{RTRS} - T_{CWD}) should have been calculated during initialization and saved for later use. DDRSHARP avoids repetitive computation by pre-computing the values of these calculations during the initialization process.

Minimum Iterations: Iterating on queued requests at every cycle takes a large chunk of the simulation time. At every memory cycle, most simulators (Li *et al*., 2020; Kim *et al*., 2015) iterate through every request in the queue in order to find the best request that satisfies the scheduling policy; all requests are visited even if the target bank is busy. For example, USIM (Chatterjee *et al*., 2012) performs a double iteration on the request queue; one to mark the request obeys timing constraints and another to select the command that is going to be issued. DDRSHARP skips iterations if the target bank is busy; that is, a request is visited and decoded if and only if the target bank is not busy doing some other activity. We employed a bank-based queueing system that allows us to skip all requests queued to a busy bank at once.

Minimum Branches: If a branch instruction is mispredicted by the CPU, the performance penalty will increase and the cumulative penalty of every cycle would magnify the simulation latency. For example, Listing 1 is a code fragment from the Controller class of DRAMsim3 (Li *et al*., 2020), reportedly the fastest cycle-accurate simulator. We can see that the Isvalid method is called twice inside of an if-block. Moreover, the implementation of the GetCommandToIssue function in line 10 also calls the IsValid method inside an if-block. It also calls another function named GetFirstReadyInQueue which calls the IsValid method inside an if-block by iterating on each command in the queue. Even though the implementation of DDRSHARP differs from most contemporary simulators, we worked on minimizing branches as much as possible; we removed many and allowed only those which we couldn't.

Evaluation

In this section, we validate the correctness of the simulator by testing if the synchronous communications between the memory controller and DRAM obey the rules of DRAM standards. We also compare the performance of DDRSHARP with other DRAM simulators.

```
1  void Controller:ClockTick()
2  {
3      // update refresh counter
4      refresh_.ClockTick();
5      bool cmd_issued = false;
6      Command cmd;
7      if (channel_state_.IsRefreshWaiting()) {
8          cmd = cmd_queue_.FinishRefresh();
9      }
10     if (!cmd.IsValid()) {
11         cmd = cmd_queue_.GetCommandToIssue();
12     }
13     if (cmd.IsValid()) {
14         IssueCommand(cmd);
15         cmd_issued = true;
16 ...
17
18 }
```

Listing 1: A code snippet from DRAMsim3 simulator

Validation and Verification

Every memory command that is issued by DDRSHARP must obey the timing constraints and state transition rules of DRAM standards. To evaluate its accuracy, we compared the simulation results of DDRSHARP with that of Ramulator (Kim *et al*., 2015). In validating our memory model, we used the same DRAM device (DDR3 4Gb ×8) for both simulators. We used a single-channel organization with one rank. We also employed the same address mapping and applied the same policy for paging and command scheduling. For each simulator, we used a sequence of 403.gcc workload of the CPU2006 benchmark as an input and we run both simulators for 20 million CPU cycles. We recorded the time-stamp of every command that is issued to respective banks and then compared the timestamped records of our simulator with that of the Ramulator.

The results were found to be identical; the sequence of commands, the execution cycle of every command, and the channel, rank, and bank where the commands were executed are found to be the same. We have observed that DDRSHARP obeys the timing constraints and state transition rules of standard DRAM. Its memory controller, channel, rank, and bank state machines acted according to DRAM specifications.

Performance Comparison with Existing DRAM Simulators

We have selected DRAMsim3 (Li *et al*., 2020) and Ramulator (Kim *et al*., 2015), among those shown in Table 3, to evaluate the performance of DDRSHARP; we compared the performance of each simulator using stream trace and random trace. Each trace contains 25 million read/write requests. The number of generated read requests is double the number of write requests. Each trace is formatted to match the trace reader of each simulator. Stream requests have higher row-buffer hit probability as they access continuous memory blocks compared to random requests which access random memory locations.

Table 3: Description of current DRAM simulators

The author details	Approach	Limitation
Li and Jacob (2019)	Statistical DRAM modeling • uses synthetic traces to train the model • Sacrifices accuracy for speed which makes	Lacks accuracy • Unusable in situations where100% accuracy is required
Feldmann *et al*. (2020)	Performance-optimized DRAM model • Uses lookup tables for latency estimation • Uses GPU to accelerate offline trace analysis using neural network allows 5% error in accuracy	lacks accuracy • Unusable in situations where 100% accuracy is required Sacrifices speed for code simplicity
Mirosanlou *et al*. (2020)	An extensible memory controller • Provides a simple interface that minimizes LOC (lines-of-code) • On average, it reduces the LOC by 11%	• The generalization feature of effects on the speed of simulation
Kim *et al*. (2015)	Ramulator: Cycle-accurate open-source simulator • Supports many DRAM models • Provides support for extensibility • Last updated on May 11, 2021	Speed
Li *et al*. (2020)	DRAMsim3: Cycle-accurate open-source simulator • Supports many DRAM models • Last updated on Apr 5, 2021	Speed

Materials and Methods

For the experiment, we have used the same configuration; DDR3 4Gb ×8 1600 K memory model, same gage policy (open gape policy), same refresh policy (rank refresh) and same scheduling policy that is First Ready, First Come First Served (FRFCS).

Results and Discussion

Fig. 8 shows the experimental results for the input of random traces. DDRSHARP is 1.88 times faster than DRAMsim3 and 2.84 times faster than Ramulator. As illustrated in Fig. 9, experimental results for a trace of stream requests show that DDRSHARP is more than twice (2.88) faster than Ramulator and 2.1 times faster than DRAMsim3.

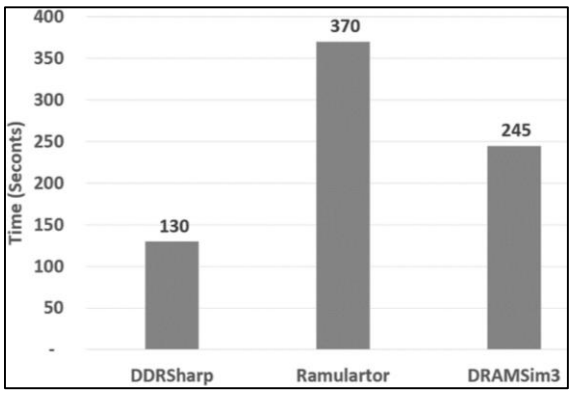

Fig. 8: Comparing the simulation time of 25 million random requests

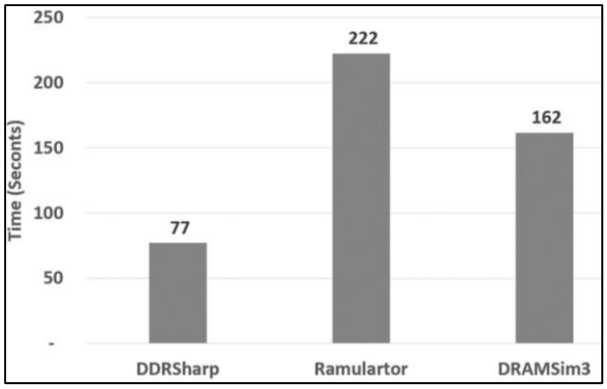

Fig. 9: Comparing simulation time of 25 million streams of requests

Conclusion

In this study, we have introduced DDRSHARP, a flexible and extensible memory simulator that supports a wide range of commercial DRAM standards. Compared to contemporary open-source simulators, DDRSHARP is at least 1.88 times faster for a trace of random requests and 2.1 times faster for a trace of stream requests.

DDRSHARP is developed using C#. Although C# code runs slower than native codes such as C/C++, we implemented code optimization techniques to speed it up. We minimized the burden of the garbage collector by allocating some data on the stack rather than the heap, minimized the number of branches (if blocks) in source code, precomputed repetitive calculations and

saved them for later use, skipped unnecessary iterations on requests queued to a busy bank. These optimization techniques have enabled better performance making DDRSHARP faster than contemporary cycle-accurate simulators which are written in C/C++.

The extensibility and performance of DRSHARP would help DRAM researchers achieve their goals. It reduces the efforts of a researcher who seeks to reuse/modify existing code to meet the demands of his/her work; especially for those who are not comfortable with pointers of C/C++ and would like to reduce programming errors and ensure code safety.

Acknowledgment

We thank the publisher for allowing us to publish our research article. We would also like appreciate Mr. Jeffery Daniels, head of technology of Science Publications, for his relentless support during the publication process.

Funding Information

The authors have not received any financial support or funding to report.

Author's Contributions

Tesfamichael Gebregziabher Gebrehiwot: Designed and constructed the simulator, he evaluated its performance and wrote the paper.

Fitsum Assamnew Andargie: Provided critical feedback and helped shape the article.

Mohammed Ismail: Supervised the work.

Ethics

This article is original and is not published elsewhere. The corresponding author confirms that all of the other authors have read and approved the manuscript.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this study.

References

Alakarhu, J., & Niittylahti, J. (2002). DRAM simulator for design and analysis of digital systems. *Microprocessors and Microsystems*, *26*(4), 189-198. https://doi.org/10.1016/S0141-9331(02)00013-3

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... & Wood, D. A. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1-7. https://doi.org/10.1145/2024716.2024718

Chatterjee, N., Balasubramonian, R., Shevgoor, M., Pugsley, S., Udipi, A., Shafiee, A., ... & Chishti, Z. (2012). Usimm: The utah simulated memory module. *University of Utah, Tech. Rep*, 1-24. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=543a8c4fc86b539deb7b350465fe2bf48083b155

Feldmann, J., Kraft, K., Steiner, L., Wehn, N., & Jung, M. (2020, March). Fast and accurate DRAM simulation: Can we further accelerate it?. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 364-369). IEEE. https://doi.org/10.23919/DATE48585.2020.9116275

Hansson, A., Agarwal, N., Kolli, A., Wenisch, T., & Udipi, A. N. (2014, March). Simulating DRAM controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 201-210). IEEE. https://doi.org/10.1109/ISPASS.2014.6844484

Healy, M. B., & Hong, S. (2017, October). Cramsim: Controller and memory simulator. In *Proceedings of the International Symposium on Memory Systems* (pp. 83-85). https://doi.org/10.1145/3132402.3132408

JEDEC. (2021). Jesd79-4d, DDR5 SDRAM. https://www.jedec.org/sites/default/files/docs/JESD79-4D.pdf

JEDEC. (2022). Jesd79-5b, DDR5 SDRAM. https://www.jedec.org/standards-documents/docs/jesd79-5b

Kim, Y., Yang, W., & Mutlu, O. (2015). Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters*, *15*(1), 45-49. https://doi.org/10.1109/LCA.2015.2414456

Li, S., & Jacob, B. (2019, September). Statistical DRAM modeling. In *Proceedings of the International Symposium on Memory Systems* (pp. 521-530). https://doi.org/10.1145/3357526.3357576

Li, S., Verdejo, R. S., Radojković, P., & Jacob, B. (2019, September). Rethinking cycle accurate DRAM simulation. In *Proceedings of the International Symposium on Memory Systems* (pp. 184-191). https://doi.org/10.1145/3357526.3357539

Li, S., Yang, Z., Reddy, D., Srivastava, A., & Jacob, B. (2020). DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters*, *19*(2), 106-109. https://doi.org/10.1109/LCA.2020.2973991

Micron. (2017a). System power calculators. https://www.micron.com/support/tools-and-utilities/power-calc

Micron (2017b). Tn-40-07: Calculating memory power for DDR4 SDRAM. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf

Micron (2020). DDR5 SDRAM features. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf

Mirosanlou, R., Guo, D., Hassan, M., & Pellizzoni, R. (2020). Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters*, *19*(2), 105-109. https://doi.org/10.1109/LCA.2020.3008288

Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, *10*(1), 16-19. https://doi.org/10.1109/L-CA.2011.4

Steiner, L., Jung, M., Prado, F. S., Bykov, K., & Wehn, N. (2020). DRAMSys4. 0: A fast and cycle-accurate systemC/TLM-based DRAM simulator. In *Embedded Computer Systems: Architectures, Modeling and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings 20* (pp. 110-126). Springer International Publishing. https://doi.org/10.1007/978-3-030-60939-9_8