MCsim: An Extensible DRAM Memory Controller Simulator

Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni

Abstract—Numerous proposals for memory controller (MC) designs have been exposed to the research community. Interest has since been growing in the area of computer architecture and real-time systems to improve the throughput of the system and/or guarantee timing requirements through novel scheduling algorithms. Consequently, comprehensive simulators are highly demanded since they provide an infrastructure for development of new ideas effectively without re-implementing the other parts of the hardware. Although there has been several proposals for off-chip memory device simulators, there is a shortage in their MC counterparts. In this article, we propose MCsim, an extensible and cycle-accurate MC simulator. Designed as an integrable environment, MCsim is able to run as a trace-based simulator as well as provide an interface to connect with external CPU and memory device simulators.

Index Terms—Memory control and access, DRAM, Simulation

1 Introduction

With the emergence of chip multiprocessors, the shared off-chip Dynamic Random Access Memory (DRAM) represents a significant bottleneck for many parallel workloads. Due to the complex internal behavior of DRAM devices, the performance of the main memory subsystem is highly dependant on the arbitration scheme employed by the Memory Controller (MC). For this reason, there is significant past and on-going interest in optimizing the MC design. In particular, the high-performance community has proposed several designs [1], [2], [3] that attempt to balance the inherent trade-off between DRAM throughput and fairness with respect to multiple memory requestors (cores and I/O devices). At the same time, a large number of predictable MC designs for safety-critical systems [4], [5] have been recently proposed to provide tight bounds on worst-case memory access latency.

Most proposed scheduling techniques at the MC in the literature are evaluated using an in-house simulator implemented from scratch, which involves enormous efforts while hindering reproducibility. Available standalone DRAM simulators [6], [7], [8], [9] tend to lack modularity and extensibility and support neither recent controller designs nor new memory standards. In particular, we note that proposed predictable MCs present a large variability in their designs, which is ill-captured by the aforementioned simulators. Existing device simulators also do not provide the capability to differentiate service to different requestors (or cores). Commonly used full-system simulators [10], [11] include a DRAM subsystem; however, for simulation speed, they rely on a fast but simplified/inaccurate model of the DRAM [12], e.g., using fixed-latency [13], or non-cycle accurate models [14]. Finally, Ramulator [15] is a modular and extensible DRAM simulator, but it was designed to explicitly model the structure and behavior of DRAM devices, while the underlying DRAM MC is still a relatively simplified model.

Designing an MC simulator from scratch increases the time required to test and validate new ideas and complicates the process of comparing competing designs. To address such issues, we contribute MCsim: a modular, extensible, and cycle-accurate object-oriented simulation framework for DRAM MCs. Specifically, we focus on rapid implementation, testing, and evaluation of new memory scheduling policies.

Every novel policy optimizes a specific module of the MC to meet a certain design goal such as delivering a higher performance or providing a tighter latency bound. These optimizations can manifest in one or more of the MC modules: command generation schemes, request scheduling mechanisms, command scheduling policies, and different requestor criticalities. In addition, a specific policy's design choice represents a dependency among various MC's modules which results in an involved analysis and development process. This complexity further motivates us to design MCsim such that it provides a modular and configurable architecture, which enables the designer to develop, test, and analyze a specific module of the MC design without affecting the other entities, and hence, eases the modeling of new MC policies. MCsim is designed based on the observation that even though different MCs employ widely different scheduling schemes, they still process memory requests by a set of common functions that are used to implement standard hardware blocks and processing flows. These functions tend to contribute the majority of the code in any simulator, and thus, can be reused across different designs.

We prove the extensibility of MCsim by successfully implementing multiple commercial-of-the-shelf (COTS), high-performance, and predictable MCs. Note that the average Lines-of-Code (LOC) required to develop 14 MCs in MCsim is only 133 per controller, demonstrating minimal effort towards implementing new policies. MCsim can be built on any platform supporting `C++11`, and it can be integrated with any detailed DRAM device model. We integrate MCsim with Ramulator as an example of a validated and open-source device simulator.

TABLE 1
Comparison of capabilities between device simulators and MCsim.

Capability	DRAMsim2 [6]	Ramulator [15]	MCsim
Modular structure	✗	✗	✓
Queuing configurability	Partial	✗	✓
Device specific simulator	✓	✓	✗
Trace-based simulation	✓	✓	✓
Full-system integration	✓	✓	✓
Supported MCs	4	4	14

Table 1 shows the capabilities of MCsim compared to DRAM device simulators. Notice that, in addition to the trace-based execution mode in MCsim, the interfaces facilitate the ability to connect MCsim to CPU simulator such as gem5 [10] and MacSim [11] through the provided library. The simulator code is available at [16].

In the rest of the paper, we describe the overall and detailed design of MCsim, focusing on how we streamline the design of new policies by capturing the generality among existing MC designs.

2 Architectural Design

MCsim employs a modular, expansible, configurable, and integrable design; Figure 1 illustrates the major hardware blocks implemented in the framework. MCsim consists of an address translator (address mapping), which maps requests to physical memory cells, a command generator that converts requests into access commands, and request and command schedulers which determine the order of request/command execution.

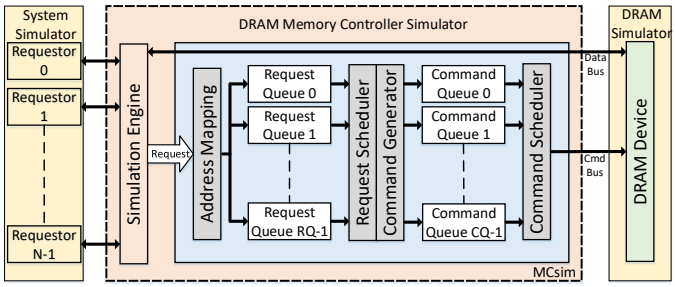

Fig. 1. Generalized MC architecture and major blocks.

Modularity: each block is constructed independently, and the encapsulated data is accessed through a simple interface. In this manner, changes to the behavior of a particular block do not impact the other blocks in the system. The specific algorithms implemented by these blocks must be customized based on the MC design. **Expansibility:** MCsim exploits the benefits of inheritance and polymorphism by providing virtual function interfaces, which minimize the amount of code required to extend the functionality of each block. **Configurability:** an MC simulator must include queues to connect the hardware blocks and temporarily store requests and commands. Rather than fixing the structure of the queues as in most other MC simulators, MCsim provides an easy to configure and modular queue structure. Since DRAM devices are organized in hierarchy levels (e.g., channels, ranks, bank groups, banks), the configurable queue structure allows the designer to construct them according to any DRAM level. **Integrability:** as shown in Figure 1, MCsim employs a generalized interface that can be accessed by any external system simulator to send memory requests and employs an abstract DRAM interface for the DRAM device model, so that the framework is not tied to any specific memory device type. For the device interface, we currently connect MCsim to Ramulator as the preferred device simulator since it supports a wide variety of DRAM standards. Note that a researcher can even implement his own device model and interface it directly to MCsim as far as it adheres to the MCsim interface. For the CPU system interface, MCsim can run as a trace-based simulator. It also provides an interface to connect two of the commonly used simulators, namely gem5 [10] and MacSim [11].

3 Configuration and Simulation Engine

A specific MC is built by a configuration file (`.ini`) to define the structure of the queues as well as the operation of each hardware block. As an example in Pseudo Code 1, we show the configuration for the ORP controller [4], which requires per-requestor buffers and applies `DIRECT` request arbitration, `OPEN` command generation and a specific `ORP` command scheduling policy. MCsim also enables to configure the address mapping based on all possible different permutations of the DRAM device hierarchy, which allows the user to assign the mapping schemes flexibly. Each digit represents the corresponding hierarchy level, and the permutation determines the order of decoding. The permutation of the address bits can change the performance of a task based on how the data is allocated.

```
1  // Rank[0],BankGroup[1],Bank[2],SubArray[3],Row[4],Col[5]
2  AddressMapping=012345 // order of address translation
3  RequestBuffer=0000    // request queue per-level
4  ReqPerREQ=1           // request queue per-requestor
5  WriteBuffer=0         // dedicated queue for write requests
6  CommandBuffer=0000    // command queue per-level
7  CmdPerREQ=1           // command queue per-requestor
8  // scheduler Based on Keys
9  RequestScheduler='DIRECT'// employ "DIRECT" request scheduler
10 CommandGenerator='OPEN'  // employ "OPEN" command generator
11 CommandScheduler='ORP'   // employ "ORP" command scheduler
12 // queue structure scheme: 0000 --> Channel, 1000 --> Rank
13 // 0100 --> BankGroup, 0010 --> Bank, 0001 --> SubArray
```

Pseudo Code 1. Configuration Parameters for ORP

The request and command queues are constructed based on the selected DRAM hierarchy level. The bits value for each DRAM level is shown in lines 12 and 13 of Pseudo Code 1. These schemes are used to build the configured number of queues and also provide a flexible hardware structure to support most predictable DRAM scheduling policies. The structure of request and command queues is depicted in Figure 2. There are three separate buffers for request queues: first, a general buffer is used to store any incoming request; second, a set of buffers can be configured using the `ReqPerREQ` parameter to separate requests by individual requestors; and last, a write buffer can be enabled via the `WriteBuffer` parameter to separate write requests of any requestor from read requests. By providing these configurations, MCsim can support request schedulers that arbitrate among DRAM hierarchies, requestor IDs, type of requests, or all of the above. For example, some MCs schedulers arbitrate among requestors (cores) regardless of the DRAM location of a request [4]. Therefore, an individual buffer is created for each requestor. Other MCs arbitrate among DRAM banks rather than requestors and require a per-bank queue [4], [17]. The request arbitration can also be performed on two levels. For instance, DCmc requires a request queue per bank and performs round-robin (RR) among requestors in each bank. Typical high-performance arbiters employing First-Ready First-Come-First-Serve (FR-FCFS) arbitration in addition to a separate arbitration between read and write requests. The command queue shown in Figure 2 is similar to the request queue and can be configured based on two parameters. `CmdPerREQ` is used to separate commands for

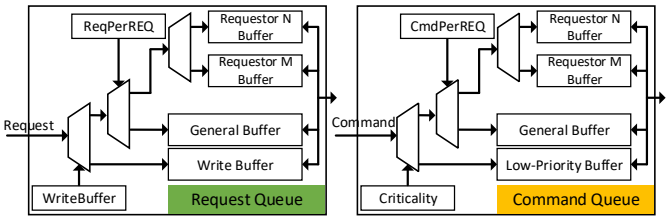

Fig. 2. Request and command queue structures per resource level.

different requestors. Rather than having separate buffers for read and write requests, the command queue has separate command buffers for commands with different criticalities. Criticality reflects the priority of requests. Assigning different priorities for requests is a common theme both in predictable systems as well as COTS platforms. In a case of an MC with two priorities (for instance critical vs. non-critical requests), `General Buffer` is only employed for high requestors, while the `Low-Priority Buffer` stores lower-priority commands. The sub-classes of `RequestScheduler`, `CommandGenerator`, and `CommandScheduler` are selected based on their names. The string name of a subclass must be defined in `schedulerRegister.h` to notify which subclass will be used according to the names.

4 Detailed System Design

Throughout this section, we explain the detailed functionality and implementation of hardware blocks as well as their interactions according to the MCsim class diagram in Figure 3.

4.1 Top-Level Memory Controller

The top-level MemoryController is responsible for controlling the interaction between each internal hardware block and managing the requests and data flow between external memory requests and memory devices. In order to differentiate the requirements of each requestor in a system, we consider a requestor to be either a critical (high priority) or non-critical (low priority). Thus, a *requestorCriticalTable* can be configured by the user to indicate the criticality of each requestor. Then, the table can be used by any hardware blocks

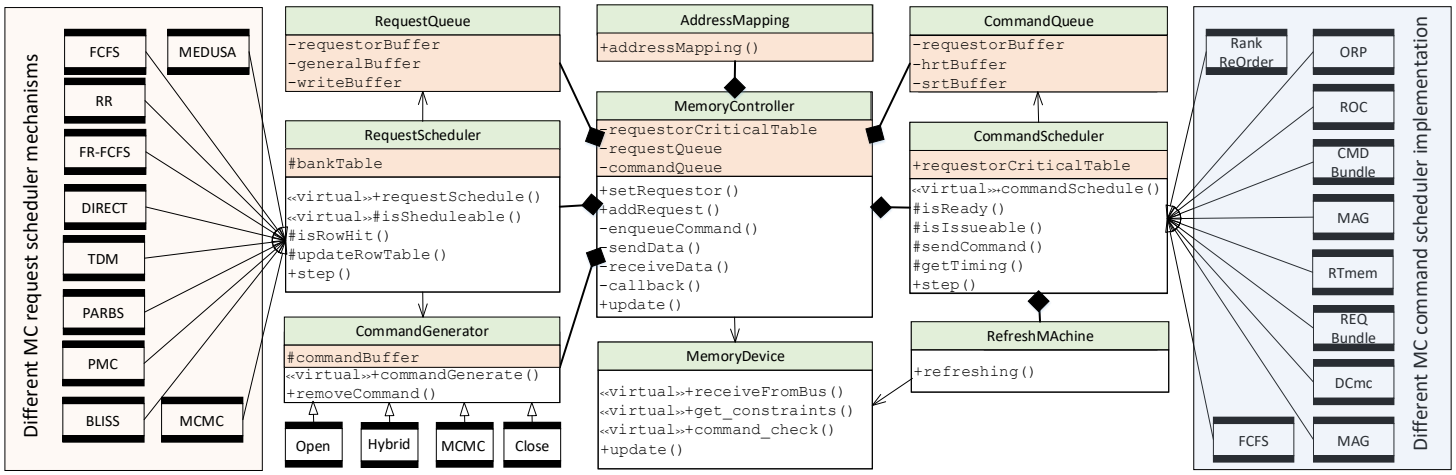

Fig. 3. MCsim class diagram representing the main functional blocks in the simulator.

to make scheduling decisions among requests or commands based on the criticality of the requestors.

MemoryController receives new memory requests from requestors through `addRequest(ID, Address, Type, Size, Data)` and sends complete requests back to the requestors through `callback(Request)` provided by the simulation engine. MemoryController is also responsible for inserting requests and commands into their corresponding queues. Once there are available data that can be transmitted through the data bus, MemoryController communicates with the DRAM device through `receiveData(Data)` and `sendData(Data)` functions.

The `step()` function triggers the proceeding of each of the internal hardware blocks. According to Figure 3, the `requestSchedule()` function of RequestScheduler is first called to select requests that can be converted into commands. All commands generated by the CommandGenerator are en-queued into back-end command queues. Finally, the `commandSchedule()` function of CommandScheduler is called to issue an available command to the DRAM device. Since the data bus and the command bus are separated, available data can be sent or received in parallel with the command.

4.2 Functional Hardware Blocks

4.2.1 AddressMapping

The interface `addressMapping(request)` takes an incoming request and assigns a physical memory location to the request. The location of a request is determined by shifting the memory level bits in the order of the mapping scheme used in the configuration file.

4.2.2 RequestScheduler

The request scheduler is connected with both request and command queues because the arbitration may not only depend on the available requests in a request queue, but also on the status of corresponding command queues. For example, RTMem only allows a new request to be scheduled if there is no activate command in any of the command queues.

A *bankTable* is used to track the currently active row in the row buffer of each bank. It determines if a selected request is targeting an open or close row. The *bankTable* is accessed by `isRowHit(Request)` before a request is sent to command generator and updated by `updateRowTable(Rank,Bank,Row)` once the request is converted into commands.

4.2.3 CommandGenerator

The abstract class CommandGenerator has a virtual interface `commandGenerate(Request, isOpen)` which is called by requestScheduler to decode a request into a set of DRAM commands. This procedure is done based on the status of the row and generation pattern, including open, close, or hybrid page policies. All generated commands in one cycle are temporally stored in a command buffer, which is later accessed by the top-level memory controller. We separate requestScheduler and commandGenerator to allow development of each of those components separately.

4.2.4 CommandScheduler

The command scheduler is connected to the command queues and the DRAM device interface. It contains a `requestorCriticalTable` for each command queue to record the criticality of each requestor. `cmdQueueTimer` tracks the minimum number of clock cycles that any commands must wait before being issued. The table is updated once a command is issued to DRAM devices, and the counter for each command decremented every clock cycle. As an example, we show ORP scheduling mechanism in Pseudo Code 2. Ready commands from each requestor command buffer are first pushed to a FIFO buffer according to their criticality. When there is a CAS command in the critical FIFO that cannot be issued to the device, the CAS command will block all the other CAS commands (by *CASBlock*) in the FIFO, but not the commands of other types. If there is no command issueable from the high-priority FIFO, the scheduler tries to schedule a command from the low-priority FIFO.

4.2.5 MemorySystem

To support a broad range of DRAM standards, MCsim has a general interface to access DRAM information provided by the user through three virtual functions: 1) `GET_CONSTRAINT(name)`: the MC retrieves the timing constraint values for a selected DRAM device from the device simulator. 2) `CHECK_COMMAND(Cmd)`: once a command is selected from command scheduler, this function is used to determine whether the command can be issued in the current clock; 3) `RECEIVE_COMMAND(Cmd)`: behaves as an interface to the command bus; it is called by `sendCommand(Cmd)` in commandScheduler to issue a command through the command bus. Notice that, if the implementation of a certain MC design requires changes in the device itself, some modifications also need to be done in MCsim which would be straightforward due to the modularity of the simulator.

```
Function scheduleCommand_ORP()
    for('each requestorBuffer in a channel commandQueue')
        if('requestorBuffer is not empty')
            get front Cmd from the requestorBuffer;
                if('isReady(Cmd)')
                    if('HRT requestor')
                        push Cmd into FIFO;
                    else
                        push Cmd into SRT–FIFO;
    CASblock = false;
    for('every Cmd in FIFO from the front of the queue')
        if('CASblock is true and Cmd is CAS')
            continue;
            if('issIssueable(Cmd)')
                sendCommand (Cmd);
                return Cmd;
            else if('Cmd is CAS')
                CASblock = true;
    for('every Cmd in the SRT–FIFO from front of the queue')
        if('isIssueable(Cmd)')
            sendCommand(Cmd);
            return Cmd;
    return NULL;
```

Pseudo Code 2. ORP Command Scheduler

5 Evaluation and Validation

1) **Lines-of-Code (LOC):** To show the effectiveness of MCsim we implement 10 different MC scheduling techniques at the request-level and 11 ones at the command-level as Figure 3 illustrates. We observe that the maximum amount of controller-specific code in MCsim is less than 11%, and the largest amount of code required to implement any controller is 432. In Table 2, we report the LOC required to implement each MC. In addition, we also implemented a device based refresh mechanism (per-bank refresh [18]) that only adds 65 LOC to MCsim.

TABLE 2
Simulation time (sec) of MCs for different simulators. ✓represents the ability to distinguish among different requestors in each MC.

Controller	REQ	Simulator	LOC	$RunTime_{seq}$	$RunTime_{rand}$
FR-FCFS	✗	MCsim	32	3.91	8.7
	✗	Ramulator#	309	18.97	31.01
	✗	Ramulator	243	6.04	9.12
	✗	DRAMsim2	356	3.28	8.28
BLISS	✓	MCsim	78	32.01	75.71
	✓	Ramulator#	335	293.32	422.52
PARBS	✓	MCsim	138	46.95	115.66
	✓	Ramulator#	424	172.62	372.21
ORP	✓	MCsim	93	44.80	74.83
	✓	Standalone	542	103.76	726.31
RTMem	✓	MCsim	96	86.69	98.16
	✓	Standalone	1910	50.12	51.24
MEDUSA	✓	MCsim	123	33.78	98.30
CMDBundle	✓	MCsim	169	37.95	73.27
REQBundle	✓	MCsim	432	52.27	57.49
MAG	✓	MCsim	94	40.31	71.16
MCMC	✓	MCsim	182	63.61	66.95
DCmc	✓	MCsim	85	42.81	71.35
AMC	✓	MCsim	98	81.90	92.38
PMC	✓	MCsim	65	93.97	105.52
ROC	✓	MCsim	175	40.38	72.54

2) **Simulation Time (RunTime):** We compare MCsim with existing DRAM and MC simulators, which are open-sourced and can run as a standalone package with inputs trace, making it viable to provide a fair and reproducible comparison. We employ two synthetic memory traces containing one million requests each, including 90% read and 10% write requests since reads are more critical in general. The `seq` trace is constructed such that it accesses rows consecutively, which tends to access open rows in the device. The `rand` trace is created with completely randomized address locations in order to stress the controllers with close requests. Since DDR3 device is supported in all the simulators, we run each simulator on our host (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 16GB RAM, and the Linux kernel is 4.15) with DDR3 1600H device with the same DRAM structures and timing constraints and make sure each simulator has the same system parameters. We configured all device simulators to employ FR-FCFS scheduling. The simulation time are shown in Table 2. Regarding FR-FCFS, the run time is presented, such that each simulator finishes all the requests in the trace. For the rest of the controllers that are concerned about fairness, we duplicate the trace into eight such that each requestor accesses its own trace, which consists of 1 million requests as used previously. We conclude that MCsim provides comparable simulation time to the device simulators, as well as the greatest extensibility that enables us to develop new controllers. Notice that the extra lines of code for implementing a new controller is small; however, on the downside, the generalization feature of MCsim might slow down the simulation speed (specifically, this is the case for RTMem).

3) **Behavioral Validation:** We validate MCsim using the regression test suite provided by the DRAM subsystem validation tool, MCXplore [19]. This regression suite covers a wide range of controller parameters such as hit ratio, read/write ratio, and interleaving. We validate the correctness of MCsim by comparing the issuing times of commands with the JEDEC standard's dictated constraints. We also compare each policy with its counterpart in a publicly available simulator as Table 2 shows. Results confirm that MCsim conforms to the standard and the behavior of each policy matches that of the corresponding simulator.

6 Conclusion

In this paper, we introduce MCsim, an extensible cycle-accurate simulator for MC designs. MCsim provide flexibility such that it can run in a trace-based mode or integrate to an external system simulator with the provided libraries. We expect MCsim to significantly accelerate the design and testing of novel MCs.

References

[1] L. Subramanian *et al.*, "Bliss: Balancing performance, fairness and complexity in memory access schedyuling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 3071–3087, 2016.

[2] S. Rixner *et al.*, "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 128–138.

[3] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[4] D. Guo *et al.*, "A comparative study of predictable DRAM controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, 2018.

[5] R. Mirosanlou *et al.*, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 82–94.

[6] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[7] N. Chatterjee *et al.*, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[8] M. K. Jeong *et al.*, "Drsim: A platform for flexible dram system research," *Accessed in: http://lph. ece. utexas. edu/public/DrSim*, 2012.

[9] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2012, pp. 392–397.

[10] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] H. Kim *et al.*, "Macsim: Simulator for heterogeneous architecture," 2012.

[12] S. Li *et al.*, "Rethinking cycle accurate dram simulation," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 184–191.

[13] S. Srinivasan *et al.*, "Cmp memory modeling: How much does accuracy matter?" in *Citeseer*, 2009.

[14] A. Hansson *et al.*, "Simulating dram controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 201–210.

[15] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.

[16] "Mcsim: An extensible dram memory controller simulator," https://github.com/uwuser/MCsim.

[17] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[18] K. K.-W. Chang *et al.*, "Improving dram performance by parallelizing refreshes with accesses," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 356–367.

[19] M. Hassan and H. Patel, "Mcxplore: Automating the validation process of dram memory controller designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1050–1063, 2017.