SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies

Hasan Hassan[1,2,3] Nandita Vijaykumar[3] Samira Khan[4,3] Saugata Ghose[3] Kevin Chang[3]
Gennady Pekhimenko[5,3] Donghyuk Lee[6,3] Oguz Ergin[2] Onur Mutlu[1,3]

[1]*ETH Zürich* [2]*TOBB University of Economics & Technology* [3]*Carnegie Mellon University*
[4]*University of Virginia* [5]*Microsoft Research* [6]*NVIDIA Research*

DRAM is the primary technology used for main memory in modern systems. Unfortunately, as DRAM scales down to smaller technology nodes, it faces key challenges in both data integrity and latency, which strongly affects overall system reliability and performance. To develop reliable and high-performance DRAM-based main memory in future systems, it is critical to characterize, understand, and analyze various aspects (e.g., reliability, latency) of existing DRAM chips. To enable this, there is a strong need for a publicly-available DRAM testing infrastructure that can flexibly and efficiently test DRAM chips in a manner accessible to both software and hardware developers.

This paper develops the first such infrastructure, SoftMC (Soft Memory Controller)*, an FPGA-based testing platform that can control and test memory modules designed for the commonly-used DDR (Double Data Rate) interface. SoftMC has two key properties:* (i) *it provides* flexibility *to thoroughly control memory behavior or to implement a wide range of mechanisms using DDR commands; and* (ii) *it is* easy to use *as it provides a simple and intuitive high-level programming interface for users, completely hiding the low-level details of the FPGA.*

We demonstrate the capability, flexibility, and programming ease of SoftMC with two example use cases. First, we implement a test that characterizes the retention time of DRAM cells. Experimental results we obtain using SoftMC are consistent with the findings of prior studies on retention time in modern DRAM, which serves as a validation of our infrastructure. Second, we validate two recently-proposed mechanisms, which rely on accessing recently-refreshed or recently-accessed DRAM cells faster than other DRAM cells. Using our infrastructure, we show that the expected latency reduction effect of these mechanisms is not observable in existing DRAM chips, which demonstrates the usefulness of SoftMC in testing new ideas on existing memory modules. We discuss several other use cases of SoftMC, including the ability to characterize emerging non-volatile memory modules that obey the DDR standard. We hope that our open-source release of SoftMC fills a gap in the space of publicly-available experimental memory testing infrastructures and inspires new studies, ideas, and methodologies in memory system design.

1. Introduction

DRAM (Dynamic Random Access Memory) is the predominant technology used to build main memory systems of modern computers. The continued scaling of DRAM process technology has enabled tremendous growth in DRAM density in the last few decades, leading to higher capacity main memories. Unfortunately, as the process technology node scales down to the sub-20 nm feature size range, DRAM technology faces key challenges that critically impact its reliability and performance.

The fundamental challenge with scaling DRAM cells into smaller technology nodes arises from the way DRAM stores data in cells. A DRAM cell consists of a transistor and a capacitor. Data is stored as charge in the capacitor. A DRAM cell cannot retain its data permanently as this capacitor leaks its charge gradually over time. To maintain correct data in DRAM, each cell is periodically refreshed to replenish the charge in the capacitor. At smaller technology nodes, it is becoming increasingly difficult to store and retain enough charge in a cell, causing various reliability and performance issues [60, 61]. Ensuring reliable operation of the DRAM cells is a key challenge in future technology nodes [38, 45, 60, 61, 65, 68, 71].

The fundamental problem of retaining data with less charge in smaller cells directly impacts the reliability and performance of DRAM cells. First, smaller cells placed in close proximity make cells more susceptible to various types of interference. This potentially disrupts DRAM operation by flipping bits in DRAM, resulting in major reliability issues [46, 66, 74, 83, 84, 90], which can lead to system failure [66, 84] or security breaches [25, 46, 82, 85, 86, 95, 98]. Second, it takes longer time to access a cell with less charge [27, 56], and write latency increases as the access transistor size reduces [38]. Thus, smaller cells directly impact DRAM latency, as DRAM access latency is determined by the worst-case (i.e., *slowest*) cell in any chip [17, 56]. DRAM access latency has not improved with technology scaling in the past decade [6, 36, 57, 71], and, in fact, some latencies are expected to increase [38], making memory latency an increasingly critical system performance bottleneck.

As such, there is a significant need for new mechanisms that improve the reliability and performance of DRAM-based main memory systems. In order to design, evaluate, and validate many such mechanisms, it is important to accurately characterize, analyze, and understand DRAM (cell) behavior in terms of reliability and latency. For such an understanding to be accurate, it is critical that the characterization and analysis be based on the *experimental* studies of *real DRAM chips*, since a large number of factors (e.g., various types of cell-to-cell interference [46, 74, 83], inter- and intra-die process variation [17, 18, 56, 58, 75], random effects [29, 61, 91, 103], operating conditions [59, 61], internal organization [30, 43, 61], stored data patterns [43, 44, 61]) concurrently impact the reliability and latency of cells. Many of these phenomena and their interactions cannot be properly modeled (e.g., in simulation or using analytical methods) without rigorous experimental characterization and analysis of real DRAM chips. The need for such experimental characterization and analysis, with the goal of building the understanding necessary to improve the reliability and performance of future DRAM-based main memories at various levels (both software and hardware), motivates the need for a publicly-available DRAM testing infrastructure that can enable system users and designers to characterize real DRAM chips.

Two key features are desirable from such an experimental memory testing infrastructure. First, the infrastructure should be *flexible* enough to test any DRAM operation (supported by the commonly-used DRAM interfaces, e.g., the standard Double Data Rate, or DDR, interface) to characterize cell behavior or evaluate the impact of a mechanism (e.g., adopting different refresh rates for different cells [42, 44, 60, 79, 96]) on real DRAM chips. Second, the infrastructure should be *easy to use*, such

that it is possible for both software and hardware developers to implement new tests or mechanisms without spending significant time and effort. For example, a testing infrastructure that requires circuit-level implementation, detailed knowledge of the physical implementation of DRAM data transfer protocols over the memory channel, or low-level FPGA-programming to modify the infrastructure would severely limit the usability of such a platform to a limited number of experts.

This paper designs, prototypes, and demonstrates the basic capabilities of such a flexible and easy-to-use experimental DRAM testing infrastructure, called *SoftMC (Soft Memory Controller)*. SoftMC is an open-source FPGA-based DRAM testing infrastructure, consisting of a programmable memory controller that can control and test memory modules designed for the commonly-used DDR (Double Data Rate) interface. To this end, SoftMC implements *all* low-level DRAM operations (i.e., DDR commands) available in a typical memory controller (e.g., opening a row in a bank, reading a specific column address, performing a refresh operation, enforcing various timing constraints between commands). Using these low-level operations, SoftMC can test and characterize any (existing or new) DRAM mechanism that uses the existing DDR interface. SoftMC provides a simple and intuitive high-level programming interface that completely hides the low-level details of the FPGA from users. Users implement their test routines or mechanisms in a high-level language that automatically gets translated into the low-level SoftMC memory controller operations in the FPGA.

SoftMC can be used to implement any DRAM test or mechanism consisting of DDR commands, without requiring significant effort. Users can verify whether the test or mechanism works successfully on real DRAM chips by monitoring whether any errors are introduced into the data. The high-level programming interface provides simple routines to verify data integrity at any point during the test. SoftMC offers a wide range of use cases, such as characterizing the effects of variation within a DRAM chip and across DRAM chips, verifying the correctness of new DRAM mechanisms on actual hardware, and experimentally discovering the reliability, retention, and timing characteristics of an unknown or newly-designed DRAM chip (or finding the best specifications for a known DRAM chip). We demonstrate the potential and ease of use of SoftMC by implementing two use cases.

First, we demonstrate the ease of use of SoftMC's high-level interface by implementing a simple retention time test. Using this test, we characterize the retention time behavior of cells in modern DRAM chips. Our test results match the prior experimental studies that characterize DRAM retention time in modern DRAM chips [42, 60, 61, 79], providing a validation of our infrastructure.

Second, we demonstrate the flexibility and capability of SoftMC by validating two recently-proposed DRAM latency reduction mechanisms [27, 87]. These mechanisms exploit the idea that highly-charged DRAM cells can be accessed with low latency. DRAM cells are in a highly-charged state when they are recently refreshed or recently accessed. Our SoftMC-based experimental analysis of 24 real DRAM chips from three major DRAM vendors demonstrates that the expected latency reduction effect of these mechanisms is *not* observable in existing DRAM chips. We discuss the details of our experiments in Section 6.2 and provide reasons as to why the effect is not observable in existing chips. This experiment demonstrates *(i)* the importance of experimentally characterizing real DRAM chips to understand the behavior of DRAM cells, and designing mechanisms that are based on this experimental understanding; and *(ii)* the effectiveness of SoftMC in testing (validating or refuting) new ideas on existing memory modules.

We also discuss several other use cases of SoftMC, including the ability to characterize emerging non-volatile memory modules that obey the DDR standard. We hope that SoftMC inspires other new studies, ideas, and methodologies in memory system design.

This work makes the following major contributions:

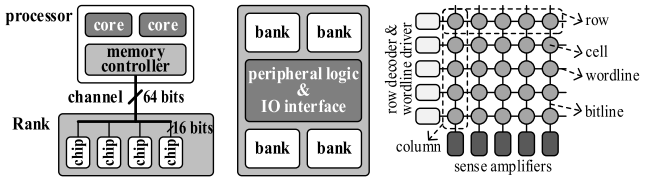
- We introduce SoftMC, the first open-source FPGA-based experimental memory testing infrastructure. SoftMC implements all low-level DRAM operations in a programmable memory controller that is exposed to the user with a flexible and easy-to-use interface, and hence enables the efficient characterization of modern DRAM chips and evaluation of mechanisms built on top of low-level DRAM operations. To our knowledge, SoftMC is the first publicly-available infrastructure that exposes a high-level programming interface to ease memory testing and characterization.
- We provide a prototype implementation of SoftMC with a high-level software interface for users and a low-level FPGA-based implementation of the memory controller. We have released the software interface and the implementation publicly as a freely-available open-source tool [88].
- We demonstrate the capability, flexibility, and programming ease of SoftMC by implementing two example use cases. Our second use case demonstrates the effectiveness of SoftMC as a new tool to test existing or new mechanisms on existing memory chips. Using SoftMC, we demonstrate that the expected effect (i.e., highly-charged DRAM rows can be accessed faster than others) of two recently-proposed mechanisms is not observable in 24 modern DRAM chips from three major manufacturers.

2. Background

In this section, we first provide the necessary basics on DRAM organization and operation. We also provide background on the DDR command interface, which is the major standard for modern DRAM-based memories. For a more detailed description of DRAM operation, we refer the reader to [18, 47, 56, 57, 60].

2.1. DRAM Organization

Figure 1a shows the organization of a DRAM-based memory system. This system consists of *memory channels* that connect the processor to the memory. A channel has one or more *ranks* that share the control and data bus of the channel. A rank typically consists of multiple (typically four to eight) DRAM chips. All DRAM chips in a rank share the command signals such that they operate in lockstep. Each chip has an 8- or 16-bit wide IO bus that contributes to the 64-bit IO bus of the channel. For example, four DRAM chips (with a 16-bit IO bus per chip) are used to build the 64-bit IO bus in the memory channel.



(a) System **(b)** Chip **(c)** Bank

Figure 1: DRAM-based memory system organization

Figure 1b depicts a DRAM chip consisting of multiple *banks*. Each bank's cells can be accessed independently of other banks. Figure 1c shows the key details of a bank, which consists of *(i)* a 2D cell array, *(ii)* a row decoder, and *(iii)* sense amplifiers. A row of cells is connected to the row decoder through a wire called *wordline*, and a column of cells is connected to a sense amplifier through a wire called *bitline*. When accessing a row of cells, one of the wordlines in the cell array is selected by

the row decoder. Then, the corresponding wordline driver raises the wordline, making electrical connections between cells in the row and sense amplifiers through the bitlines. This operation, i.e., accessing a row of cells, is referred to as row *activation*.

2.2. DRAM Operations and Commands

There are four DRAM operations necessary for data access and retention. We describe them and their associated commands and timing constraints, as specified in the DDR standard [34].

Activation. When the memory controller receives a request to a row that is not already activated, it issues an `ACTIVATE` command with a row address to the DRAM. Then, the row decoder determines the wordline that corresponds to the requested row address, and enables that wordline to activate the cells connected to it. The activated cells share their charge with the bitlines that they are connected to. Then, sense amplifiers detect data by observing the perturbation that charge sharing created on the bitlines, and fully *restore* the charge of the activated cells. At the end of activation, sense amplifiers contain the data of the activated row.

Read/Write. Some time after issuing the `ACTIVATE` command, the memory controller issues a column command, i.e., `READ` or `WRITE`, to select a portion of data in the activated row which corresponds to the requested column address. The `tRCD` timing parameter restricts the minimum time interval between an `ACTIVATE` command and a column command, to guarantee that the DRAM chip is ready to transfer data from the sense amplifiers to the memory channel. When the controller performs a `READ` operation after any `WRITE` command is issued to an open row in any bank, the `tWTR` timing parameter needs to be obeyed. Likewise, when the controller performs a `WRITE` operation after any `READ` command is issued to an open row in any bank, it needs to obey the `tRTW` timing parameter. During `tWTR` and `tRTW`, the DDR bus is switched between read and write modes, which is called bus turnaround.

Precharge. If a request to access a new row arrives while another row is currently activated in the same bank, the memory controller first needs to prepare the DRAM bank for the new row access. This operation is called *precharge*, and consists of two steps. First, the activated row in the bank needs to be *deactivated*, by disconnecting and isolating the activated cells. Second, the sense amplifiers (and corresponding bitlines) need to be prepared (i.e., charged to the appropriate voltage levels) to detect data during the next row activation [56, 57]. There are three timing parameters related to the precharge operation. First, `tRAS` specifies the minimum time interval between an `ACTIVATE` command and a `PRECHARGE` command, to ensure that the activated cells are fully restored. Second, `tWR` (i.e., *write recovery time*) specifies the minimum time interval between the end of data transfer caused by a write command and a `PRECHARGE` command, to ensure that the cells updated by the write operation are fully restored. Third, `tRP` specifies the minimum time between a `PRECHARGE` command and an `ACTIVATE` command, to ensure that the precharge operation completes before the next activation.

Refresh. A DRAM cell cannot retain its data permanently due to charge leakage. To ensure data integrity, the charge of the DRAM cell needs to be *refreshed* (i.e., replenished) periodically. The memory controller replenishes cell charge by refreshing each DRAM row periodically (typically every 64 ms). The refresh period is specified by a timing parameter `tREFI`. The refresh operation can be performed at the rank or bank granularity [19] depending on the DDR standard. Prior to issuing a `REFRESH` command to a rank/bank, the memory controller first precharges all activated rows in the DRAM rank or the activated row in the DRAM bank that the refresh operation will be performed on. `tRP` specifies the minimum timing interval between a `PRECHARGE` command and a `REFRESH` command. After `tRP`, the memory controller issues a `REFRESH` command to perform the refresh operation, which delays the subsequent `ACTIVATE` command to the refreshing rank/bank for an interval specified by `tRFC`.

2.2.1. DDR Command Interface. DDR commands are transmitted from the memory controller to the DRAM module across a memory bus. On the memory bus, each command is encoded using five output signals (`CKE`, `CS`, `RAS`, `CAS`, and `WE`). Enabling/disabling these signals corresponds to specific commands (as specified by the DDR standard). First, the `CKE` signal (*clock enable*) determines whether the DRAM is in "standby mode" (ready to be accessed) or "power-down mode". Second, the `CS` (*chip selection*) signal specifies the rank that should receive the issued command. Third, the `RAS` (*row address strobe*)/`CAS` (*column address strobe*) signal is used to generate commands related to DRAM row/column operations. Fourth, the `WE` signal (*write enable*) in combination with `RAS` and `CAS`, generates the specific row/column command. For example, enabling `CAS` and `WE` together generates a `WRITE` command, while enabling only `CAS` indicates a `READ` command.

So far, we have described the major DRAM operations and commands specified by the DDR interface. Next, we motivate the need for a DRAM testing infrastructure that can flexibly issue these DRAM commands via a high-level user interface.

3. Motivation

A publicly-available DRAM testing infrastructure that can characterize real DRAM chips enables new mechanisms to improve DRAM reliability and latency. In this work, we argue that such a testing infrastructure should have two key features to ensure widespread adoption among architects and designers: *(i)* flexibility and *(ii)* ease of use.

Flexibility. As discussed in Section 2, a DRAM module is accessed by issuing specific commands (e.g., `ACTIVATE`, `PRECHARGE`) in a particular sequence with a strict delay between the commands (specified by the timing parameters, e.g., `tRP`, `tRAS`). A DRAM testing infrastructure should implement all low-level DRAM operations (i.e., DDR commands) with tunable timing parameters without any restriction on the ordering of DRAM commands. Such a design enables flexibility at two levels. First, it enables comprehensive testing of *any* DRAM operation with the ability to customize the length of each timing constraint. For example, we can implement a retention test with different refresh intervals to characterize the distribution of retention time in modern DRAM chips. Such a characterization can enable new mechanisms to reduce the number of refresh operations in DRAM, leading to performance and power efficiency improvements. Second, it enables testing of DRAM chips with high-level test programs, which can consist of *any combination of DRAM operations and timings*. Such flexibility is extremely powerful to test the impact of existing or new DRAM mechanisms in real DRAM chips.

Ease of Use. A DRAM testing infrastructure should provide a simple and intuitive programming interface that minimizes programming effort and time. An interface that hides the details of the underlying implementation is accessible to a wide range of users. With such a high-level abstraction, even users that lack hardware design experience should be able to develop DRAM tests.

In this work, we propose and prototype a publicly-available, open-source DRAM testing infrastructure that can enable system users and designers to easily characterize real DRAM chips. Our experimental DRAM testing infrastructure, called *SoftMC (Soft Memory Controller)*, can test DDR-based memory modules with a flexible and easy-to-use interface. In the next section,

we discuss the shortcomings of existing tools and platforms that can be used to test DRAM chips, and explain how SoftMC is designed to avoid these shortcomings.

4. Related Work

No prior DRAM testing infrastructure provides both *flexibility* and *ease of use* properties, which are critical for enabling widespread adoption of the infrastructure. Three different kinds of tools/infrastructure are available today for characterizing DRAM behavior. As we will describe, each kind of tool has some shortcomings. The goal of SoftMC is to eliminate *all* of these shortcomings.

Commercial Testing Infrastructures. A large number of commercial DRAM testing platforms (e.g., [1, 24, 76, 93]) are available in the market. Such platforms are optimized for throughput (i.e., to test as many DRAM chips as possible in a given time period), and generally apply a *fixed test pattern* to the units under test. Thus, since they lack support for flexibility in defining the test routine, these infrastructures are not suitable for detailed DRAM characterization where the goal is to investigate new issues and new ideas. Furthermore, such testing equipment is usually quite expensive, which makes these infrastructures an impractical option for research in academia. Industry may also have internal DRAM development and testing tools, but, to our knowledge, these are proprietary and are unlikely to be made openly available.

We aim for SoftMC to be a low-cost (i.e., free) and flexible open-source alternative to commercial testing equipment that can enable new research directions and mechanisms. For example, prior work [105] recently proposed a random command pattern generator to validate DRAM chips against uncommon yet supported (according to JEDEC specifications) DDR command patterns. Using the test patterns on commercial test equipment, this work demonstrates that specific sequences of commands introduce failures in current DRAM chips. SoftMC flexibly supports the ability to issue an arbitrary command sequence, and therefore can be used as a low-cost method for validating DRAM chips against problems that arise due to command ordering.

FPGA-Based Testing Infrastructures. Several prior works proposed FPGA-based DRAM testing infrastructures [31, 32, 41]. Unfortunately, all of them lack flexibility and/or a simple user interface, and none are open-source. The FPGA-based infrastructure proposed by Huang et al. [32] provides a high-level interface for developing DRAM tests, but the interface is limited to defining only data patterns and march algorithms for the tests. Hou et al. [31] propose an FPGA-based test platform whose capability is limited to analyzing only the data retention time of the DRAM cells. Another work [41] develops a custom memory testing board with an FPGA chip, specifically designed to test memories at a very high data rate. However, it requires low-level knowledge to develop FPGA programs, and even then offers only limited flexibility in defining a test routine. On the other hand, SoftMC aims to provide *full control over all DRAM commands* using a high-level *software interface*, and it is open-source.

PARDIS [5] is a reconfigurable logic (e.g., FPGA) based programmable memory controller meant to be implemented inside microprocessor chips. PARDIS is capable of optimizing memory scheduling algorithms, refresh operations, etc. at run-time based on application characteristics, and can improve system performance and efficiency. However, it does not provide programmability for DRAM commands and timing parameters, and therefore cannot be used for detailed DRAM characterization.

Built-In Self Test (BIST). A BIST mechanism (e.g, [3, 77, 78, 104, 106]) is implemented inside the DRAM chip to enable fixed test patterns and algorithms. Using such an approach, DRAM tests can be performed faster than with other testing platforms. However, BIST has two major flexibility issues, since the testing logic is hard-coded into the hardware: *(i)* BIST offers only a limited number of tests that are fixed at hardware design time. *(ii)* A limited set of DRAM chips, which come with BIST support, can be tested. In contrast, SoftMC allows for the implementation of a wide range of DRAM test routines and supports any off-the-shelf DRAM chip that is compatible with the DDR interface.

We conclude that prior work lacks either the flexibility or the ease-of-use properties that are critical for performing detailed DRAM characterization. To fill the gap left by current infrastructures, we introduce an open-source DRAM testing infrastructure, SoftMC, that fulfills these two properties.

5. SoftMC Design

SoftMC is an FPGA-based open-source programmable memory controller that provides a high-level software interface, which the users can use to initiate any DRAM operation from a host machine. The FPGA component of SoftMC collects the DRAM operation requests incoming from the host machine and executes them on real DRAM chips that are attached to the FPGA board. In this section, we explain in detail the major components of our infrastructure.

5.1. High-Level Design

Figure 2 shows our SoftMC infrastructure. It comprises three major components:

- In the host machine, the *SoftMC API* provides a high-level software interface (in C++) for users to communicate with the SoftMC hardware. The API provides user-level *functions* to 1) send SoftMC *instructions* from the host machine to the hardware and 2) receive data, which the hardware reads from the DRAM, back to the host machine. An *instruction* encodes and specifies an operation that the hardware is capable of performing (see Section 5.3). It is used to communicate the user-level *function* to the SoftMC hardware such that the hardware can execute the necessary operations to satisfy the user-level *function*. The SoftMC API provides several functions (See Section 5.2) that users can call to easily generate an instruction to perform any of the operations supported by the SoftMC infrastructure. For example, the API contains a *genACT()* function, which users call to generate an instruction to activate a row in DRAM.
- The *driver* is responsible for transferring instructions and data between the host machine and the FPGA across a PCIe bus. To implement the driver, we use RIFFA [33]. SoftMC execution is *not* affected by the long transfer latency of the PCIe interface, as our design sends *all of the instructions* over the PCIe bus to the FPGA *before the test routine begins*, and buffers the commands within the FPGA. Therefore, SoftMC guarantees that PCIe-related delays do not affect the precise user-defined timings between each instruction.
- Within the FPGA, the core *SoftMC hardware* queues the instructions, and executes them in an appropriate hardware component. For example, if the hardware receives an instruction that indicates a DRAM command, it sends DDR-compatible signals to the DRAM to issue the command.

5.2. SoftMC API

SoftMC offers users complete control over the DDR-based DRAM memory modules by providing an API (Application Programming Interface) in a high-level programming language (C++). This API consists of several C++ functions that generate *instructions* for the SoftMC hardware, which resides in

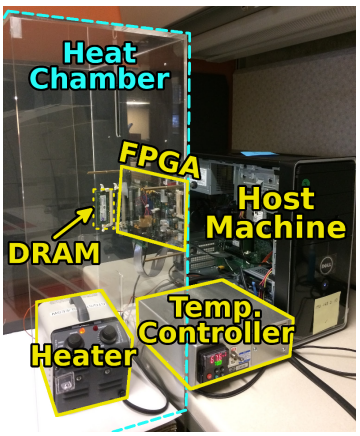


Figure 2: Our SoftMC infrastructure.

the FPGA. These instructions provide support for *(i)* all possible DDR DRAM commands; and *(ii)* managing the SoftMC hardware, e.g., inserting a delay between commands, configuring the auto-refresh mechanism (see Section 5.4). Using the SoftMC API, any sequence of DRAM commands can be easily and flexibly implemented in SoftMC with little effort and time from the programmer.

When users would like to issue a sequence of commands to the DRAM, they *only* need to make SoftMC API function calls. Users can easily generate *instructions* by calling the API *functions*. Users can insert the generated instructions to an *InstructionSequence*, which is a data structure that the SoftMC API provides to keep the instructions in order and easily send them to the SoftMC hardware via the driver. There are three types of API functions: *(i) command functions*, which provide the user the ability to specify DDR-compatible DRAM commands (e.g., `ACTIVATE`, `PRECHARGE`); *(ii)* the *wait function*, which provide the user the ability to specify a time interval between two commands (thereby allowing the user to determine the latencies of each timing parameter); and *(iii) aux functions*, which provide the user with control over various auxiliary operations. Programming the memory controller using such high-level C++ function calls enables users who are inexperienced in hardware design to easily develop sophisticated memory controllers.

Command Functions. For every DDR command, we provide a command function that generates an instruction, which instructs the SoftMC hardware to execute the command. For example, *genACT(b, r)* function generates an instruction that informs the SoftMC controller to issue an `ACTIVATE` to DRAM row *r* in bank *b*. Program 1 shows a short example program that writes to a single cache line (i.e., DRAM column) in a closed row. In Program 1, there are three command functions: *genACT()*, *genWR()*, and *genPRE()*, in lines 2, 4, and 6, which generate SoftMC hardware instructions that inform the SoftMC hardware to issue `ACTIVATE`, `WRITE`, and `PRECHARGE` commands to the DRAM.

```
1  InstructionSequence iseq;
2  iseq.insert(genACT(bank, row));
3  iseq.insert(genWAIT(tRCD));
4  iseq.insert(genWR(bank, col, data));
5  iseq.insert(genWAIT(tCL + tBL + tWR));
6  iseq.insert(genPRE(bank));
7  iseq.insert(genWAIT(tRP));
8  iseq.insert(genEND());
9  iseq.execute(fpga));
```

Program 1: Performing a write operation to a closed row using the SoftMC API.

Wait Function. We provide a single function, *genWAIT(t)*, which generates an instruction to inform the SoftMC hardware to wait *t* DRAM cycles before executing the next instruction. With this one function, the user can implement any timing constraint. For example, in Program 1, the controller should wait for `tRCD` after the `ACTIVATE` before issuing a `WRITE` to the DRAM. We simply add a call to *genWAIT()* on Line 3 to insert the `tRCD` delay. Users can easily provide any value that they want to test for the time interval (e.g., a reduced `tRCD`, to see if errors occur when the timing parameter is reduced).

Aux Functions. There are a number of auxiliary functions that allow the user to configure the SoftMC hardware. One such function, *genBUSDIR(dir)*, allows the user to switch (i.e., turn around) the DRAM bus direction. To reduce the number of IO pins, the DDR interface uses a bi-directional data bus between DRAM and the controller. The bus must be set up to move data from DRAM to the controller before issuing a `READ`, and from the controller to DRAM before issuing a `WRITE`. After *genBUSDIR()* is called, the user should follow it with a *genWAIT()* function, to ensure that the bus is given enough time to complete the turn around. The time interval for this wait corresponds to the `tWTR` and `tRTW` timing parameters in the DDR interface.

We provide an auxiliary function to control auto-refresh behavior. The SoftMC hardware includes logic to *automatically* refresh the DRAM cells after a given time interval (`tREFI`) has passed. Auto-refresh operations are not added to the code by the user, and are instead handled directly within our hardware. In order to adjust the interval at which refresh is performed, the user can invoke the *genREF_CONFIG()* function. To disable the auto-refresh mechanism, the user needs to set the refresh interval to 0 by calling that function. The user can invoke the same function to adjust the number of cycles that specify the refresh latency (`tRFC`). We describe the auto-refresh functionality of the SoftMC hardware in more detail in Section 5.4.

Finally, we provide a function, *genEND()*, which generates an *END* instruction to inform the SoftMC hardware that the end of the instruction sequence has been reached, and that the SoftMC hardware can start executing the sequence. Note that the programmer should call *iseq.execute(fpga)* to send the instruction sequence to the driver, which in turn sends the sequence to the SoftMC hardware. Program 1 shows an example of this in Lines 8–9.

5.3. Instruction Types and Encoding

Figure 3 shows the encodings for the key SoftMC instruction types.[1] All SoftMC instructions are 32 bits wide, where the most-significant 4 bits indicate the type of the instruction.

InstrType				
DDR (4)	*unused* (3)	CKE, CS (2), RAS, CAS, WE (6)	Bank (3)	Addr (16)
WAIT (4)	cycles (28)			
BUSDIR (4)	*unused* (27)			dir (1)
END (4)	*unused* (28)			

Figure 3: Key SoftMC instruction types.

DDR Instruction Type. Recall from Section 2.2.1 that the DDR commands are encoded using several fields (e.g., `RAS`, `CAS`, `WE`) that correspond to the signals of the DDR interface, which sits between the memory controller and the DRAM. The SoftMC instructions generated by our API functions encode these same fields, as shown in the instruction encoding for the DDR type instructions in Figure 3. Thus, DDR type instructions can represent *any* DDR command from the DDR3 specification [34]. For example, the user needs to set `CKE` = 1, `CS` = 0, `RAS` = 0, `CAS` = 1, and `WE` = 1 to create an `ACTIVATE` command to open the row in the bank specified by *Address* and *Bank*, respectively. Other DDR commands can easily be repre-

[1] We refer the reader to the source-code and manual of SoftMC for the encoding of *all* instruction types [88].

sented by setting the appropriate instruction fields according to the DDR3 specification.

Other Instruction Types. Other instructions carry information to the FPGA using the bottom 28 bits of their encoding. For example, the *WAIT* instruction uses these bits to encode the wait latency, i.e., wait cycles. The *BUSDIR* instruction uses the least significant bit of the bottom 28 bits to encode the new direction of the data bus. The *END* instruction requires no additional parameters, using only the *InstrType* field.

5.4. Hardware Architecture

The driver sends the SoftMC instructions across the PCIe bus to the FPGA, where the *SoftMC hardware* resides. Figure 4 shows the five components that make up the *SoftMC hardware*: *(i)* an *instruction receiver*, which accumulates the instructions sent over the PCIe bus; *(ii)* an *instruction dispatcher*, which decodes each instruction into DRAM control signals and address bits, and then sends the decoded bits to the DRAM module through the memory bus; *(iii)* a *read capture module*, which receives data read from the DRAM and sends it to the host machine over PCIe; *(iv)* an *auto-refresh controller*; and *(v)* a *calibration controller*, responsible for ensuring data integrity on the memory bus. SoftMC also includes two interfaces: a *PCIe bus controller* and a *DDR PHY*. Next, we describe each of these components in detail.

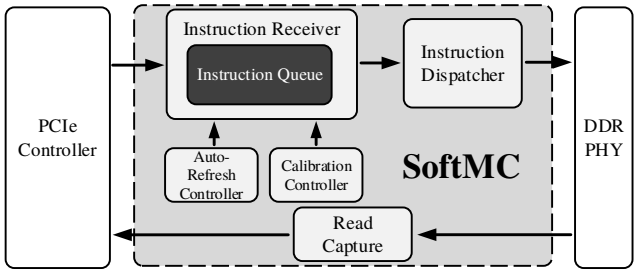


Figure 4: SoftMC hardware architecture.

Instruction Receiver. When the driver sends SoftMC instructions across the PCIe bus, the instruction receiver buffers them in an *instruction queue* until the *END* instruction is received. When the *END* instruction arrives, the instruction receiver notifies the instruction dispatcher that it can start fetching instructions from the instruction queue. Execution ends when the instruction queue is fully drained.

Instruction Dispatcher. The instruction dispatcher fetches an instruction, decodes it, and executes the operation that the instruction indicates, at the corresponding component. For example, if the type of the instruction is *DDR*, the dispatcher decodes the control signals and address bits from the instruction, and then sends them to the DRAM using the DDR PHY. Since the FPGAs operate at a lower frequency than the memory bus, the dispatcher supports fetching and issuing of multiple DDR commands per cycle. As another example, if the instruction type is *WAIT*, it is not sent to the DRAM, but is instead used to count down the number of cycles before the instruction dispatcher can issue the next instruction to the DRAM. Other instruction types (e.g., *BUSDIR, END*) are handled in a similar manner.

Read Capture Module. After a `READ` command is sent to the DRAM module, the read capture module receives the data emitted by the DRAM from the DDR PHY, and sends it to the host machine via the PCIe controller. The read capture module is responsible for ensuring data alignment and ordering, in case the DDR PHY operates on a different clock domain or reorders the data output.

Auto-Refresh Controller. The auto-refresh controller includes two registers, which store the refresh interval (`tREFI`) and refresh latency (`tRFC`) values (See Section 2.2). It issues periodic refresh operations to DRAM, based on the values stored in the registers. If the instruction dispatcher is in the middle of executing an instruction sequence that was received from the host machine, at the time a refresh is scheduled to start, the auto-refresh controller *delays* the refresh operation until the dispatcher finishes the execution of the instruction sequence. Note that a small delay does not pose any problems, as the DDR standard allows for a small amount of flexibility for scheduling refresh operations [19, 34, 69]. However, if the instruction sequence is too long, and delays the refresh operation beyond a critical interval (i.e., more than 8X the `tREFI`), it is the user's responsibility to redesign the test routine and split up the long instruction sequence into smaller sequences.

Calibration Module. This module ensures the data integrity of the memory bus. It periodically issues a command to perform *short ZQ calibration*, which is an operation specified by the DDR standard to calibrate the output driver and on-die termination circuits [20, 35]. SoftMC provides hardware support for *automatic* calibration, in order to ease programmer effort.

PCIe Controller and DDR PHY. The PCIe controller acts as an interface between the host machine and the rest of the SoftMC hardware. The DDR PHY sits between the SoftMC hardware and the DRAM module, and is responsible for initialization and clock synchronization between the FPGA and the DRAM module.

5.5. SoftMC Prototype

We describe the key implementation details of our first prototype of SoftMC.

FPGA Board. The current SoftMC prototype targets a Xilinx ML605 FPGA board [101]. It consists of *(i)* a Virtex-6 FPGA chip, which we use to implement our SoftMC design; and *(ii)* a SO-DIMM slot, where we plug in the real DRAM modules to be tested.

PCIe Interface. The PCIe controller uses *(i)* the Xilinx PCIe Endpoint IP Core [99], which implements low-level operations related to the physical layer of the PCIe bus; and *(ii)* the RIFFA IP Core [33], which provides high-level protocol functions for data communication. The communication between the host machine and the FPGA board is established through a PCIe 2.0 link.

DDR PHY. SoftMC uses the Xilinx DDR PHY IP Core [100] to communicate with the DRAM module. This PHY implements the low-level operations to transfer data over the memory channel to/from the DRAM module. The PHY transmits two DDR commands each memory controller cycle. As a result, our SoftMC implementation fetches and dispatches two instructions from the instruction queue every cycle.

Performance. To evaluate the performance of our prototype, we time the execution of a test routine that performs operations on a particular region of the DRAM and then reads data back from this region, repeating the test on different DRAM regions until the entire DRAM module has been tested. We model the expected overall execution time of the test as:

$$\textit{Expected Execution Time} = (S + E + R) * \frac{\textit{Capacity}}{\textit{Region Size}} \quad (1)$$

where *S (Send)*, *E (Execute)*, and *R (Receive)* are the host-to-FPGA PCIe latency, execution time of the command sequence in the FPGA, and FPGA-to-host PCIe latency, respectively, that is required for each DRAM region. For our prototype, we measure S and R to both be 22 µs on average (with ± 2 µs variation). E varies based on the length of the command sequence. For the sequence given in Program 2, which we use in our first use case (Section 6.1), we calculate E as 16 µs using typical DDR3 timing parameters. Since that command sequence tests only

a single 8KB row, the *Region Size* is 8KB. For an experiment testing a full 4GB memory module completely, we find that our prototype runs Program 2 on the entire module in approximately 31.5 seconds. An at-speed DRAM controller, which, at best, has *S* and *R* equal to zero, would run the same test in approximately 11.5 seconds, i.e., only 2.7x faster. As *E* increases, the performance of our SoftMC prototype gets closer to the performance of an at-speed DRAM controller. We conclude that our SoftMC prototype is fast enough to run test routines that cover an entire memory module.

6. Example Use Cases

Using our SoftMC prototype, we perform two case studies on randomly-selected real DRAM chips from three manufacturers. First, we discuss how a simple retention test can be implemented using SoftMC, and present the experimental results of that test (Section 6.1). Second, we demonstrate how SoftMC can be leveraged to test the expected effect of two recently-proposed mechanisms that aim to reduce DRAM access latency (Section 6.2). Both use cases demonstrate the flexibility and ease of use of SoftMC.

6.1. Retention Time Distribution Study

This test aims to characterize data retention time in different DRAM modules. The retention time of a cell can be determined by testing the cell with different refresh intervals. The cell fails at a refresh interval that is greater than its retention time. In this test, we gradually increase the refresh interval from the default 64 ms and count the number of bytes that have an incorrect value at each refresh interval.

6.1.1. Test Methodology. We perform a simple test to measure the retention time of the cells in a DRAM chip. Our test consists of three steps: *(i)* We write a reference data pattern (e.g. all zeros, or all ones) to an entire row. *(ii)* We wait for the specified refresh interval, so that the row is idle for that time and all cells gradually leak charge. *(iii)* We read data back from the same row and compare it against the reference pattern that we wrote in the first step. Any mismatch indicates that the cell could not hold its data for that duration, which resulted in a bit flip. We count the number of bytes that have bit flips for each test.

We repeat this procedure for all rows in the DRAM module. The read and write operations in the test are issued with the standard timing parameters, to make sure that the only timing delay that affects the cells is the refresh interval.

6.1.2. Implementation with SoftMC.

Writing Data to DRAM. In Program 2, we present the implementation of the first part of our retention time test, where we write data to a row, using the SoftMC API. First, to activate the row, we insert the instruction generated by the *genACT()* function to an instance of the *InstructionSequence* (Lines 1-2). This function is followed by a *genWAIT()* function (Line 3) that ensures that the activation completes with the standard timing parameter `tRCD`. Second, we issue write instructions to write the data pattern in each column of the row. This is implemented in a loop, where, in each iteration, we call *genWR()* (Line 5), followed by a call to *genWAIT()* function (Line 6) that ensures proper delay between two `WRITE` operations. After writing to all columns of the row, we insert another delay (Line 8) to account for the *write recovery* time (Section 2.2). Third, once we have written to all columns, we close the row by precharging it. This is done by the *genPRE()* function (Line 9), followed by a *genWAIT()* function with standard `tRP` timing. Finally, we call the *genEND()* function to indicate the end of the instruction sequence, and send the test program to the FPGA by calling the *execute()* function.

Employing a Specific Refresh Interval. Using SoftMC, we can implement the target refresh interval in two ways. We can use the auto-refresh support provided by the SoftMC hardware, by setting the `tREFI` parameter to our target value, and letting the FPGA take care of the refresh operations. Alternatively, we can disable auto-refresh, and manually control the refresh operations from the software. In this case, the user is responsible for issuing refresh operations at the right time. In this retention test, we disable auto-refresh and use a software clock to determine when we should read back data from the row (i.e., refresh the row).

```
1  InstructionSequence iseq;
2  iseq.insert(genACT(bank, row));
3  iseq.insert(genWAIT(tRCD));
4  for(int col = 0; col < COLUMNS; col++){
5      iseq.insert(genWR(bank, col, data));
6      iseq.insert(genWAIT(tBL));
7  }
8  iseq.insert(genWAIT(tCL + tWR));
9  iseq.insert(genPRE(bank));
10 iseq.insert(genWAIT(tRP));
11 iseq.insert(genEND());
12 iseq.execute(fpga));
```

Program 2: Writing data to a row using the SoftMC API.

Reading Data from DRAM. Reading data back from the DRAM requires steps similar to DRAM writes (presented in Program 2). The only difference is that, instead of issuing a `WRITE` command, we need to issue a `READ` command and enforce read-related timing parameters. In the SoftMC API, this is done by calling the *genRD()* function in place of the *genWR()* function, and specifying the appropriate read-related timing parameters. After the read operation is done, the FPGA sends back the data read from the DRAM module, and the user can access that data using the *fpga_recv()* function provided by the driver.

Based on the intuitive code implementation of the retention test, we conclude that it requires minimal effort to write test programs using the SoftMC API. Our full test is provided in our open-source release [88] and described in our extended technical report [28].

6.1.3. Results. We perform the retention time test at room temperature, using 24 chips from three major manufacturers. We vary the refresh interval from 64 ms to 8192 ms exponentially. Figure 5 shows the results for the test, where the x-axis shows the refresh interval in milliseconds, and the y-axis shows the number of erroneous bytes found in each interval. We make two major observations.

(i) We do not observe any retention failures until we test with a refresh interval of 1 s. This shows that there is a large safety margin for the refresh interval in modern DRAM chips, which is conservatively set to 64 ms by the DDR standard.[2]

[2]DRAM manufacturers perform retention tests that are similar to ours (but with proprietary in-house infrastructures that are not disclosed). Their results are similar to ours [18, 42, 56, 61], showing significant margin for the refresh interval. This margin is added to ensure reliable DRAM operation for the worst-case operating conditions (i.e., worst case temperature) and for worst-case cells, as has been shown by prior works [18, 42, 56, 61].

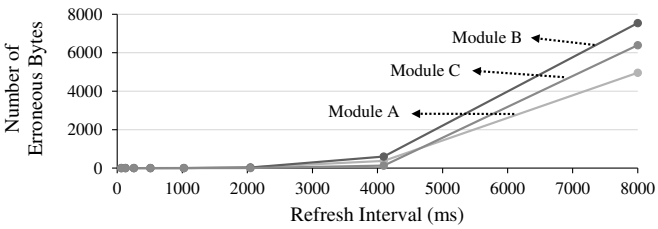


Figure 5: Number of erroneous bytes observed in retention time tests.

(ii) We observe that the number of failures increases exponentially with the increase in refresh interval.

Prior experimental studies on retention time of DRAM cells have reported similar observations as ours [26, 31, 42, 56, 61]. We conclude that SoftMC can easily reproduce prior known DRAM experimental results, validating the correctness of our testing infrastructure and showing its flexibility and ease of use.

6.2. Evaluating the Expected Effect of Two Recently-Proposed Mechanisms in Existing DRAM Chips

Two recently-proposed mechanisms, ChargeCache [27] and NUAT [87], provide low-latency access to highly-charged DRAM cells. They both are based on the key idea that a highly-charged cell can be accessed faster than a cell with less charge [56]. ChargeCache observes that cells belonging to *recently-accessed* DRAM rows are in a highly-charged state and that such rows are likely to be accessed again in the near future. ChargeCache exploits the highly-charged state of these recently-accessed rows to lower the latency for later accesses to them. NUAT observes that *recently-refreshed* cells are in highly-charged state, and thus it lowers the latency for accesses to recently-refreshed rows. Prior to issuing an `ACTIVATE` command to DRAM, both ChargeCache and NUAT determine whether the target row is in a highly-charged state. If so, the memory controller uses reduced `tRCD` and `tRAS` timing parameters to perform the access.

In this section, we evaluate whether or not the expected latency reduction effect of these two works is observable in existing DRAM modules, using SoftMC. We first describe our methodology for evaluating the improvement in the `tRCD` and `tRAS` parameters. We then show the results we obtain using SoftMC, and discuss our observations.

6.2.1. Evaluating DRAM Latency with SoftMC.

Methodology. In our experiments, we use 24 DDR3 chips (i.e., three SO-DIMMs) from three major vendors. To stress DRAM reliability and maximize the amount of cell charge leakage, we raise the test temperature to 80°C (significantly higher than the common-case operating range of 35-55°C [56]) by enclosing our FPGA infrastructure in a temperature-controlled heat chamber (see Figure 2). For all experiments, the temperature within the heat chamber was maintained within 0.5°C of the target 80°C temperature.

To study the impact of charge variation in cells on access latency in existing DRAM chips, which is dominated by the `tRCD` and `tRAS` timing parameters [18, 56, 57] (see Section 2.2), we perform experiments to test the headroom for reducing these parameters. In our experiments, we vary one of the two timing parameters, and test whether the original data can be read back correctly with the reduced timing. If the data that is read out contains errors, this indicates that the timing parameter cannot be reduced to the tested value without inducing errors in the data. We perform the tests using a variety of data patterns (e.g., 0x00, 0xFF, 0xAA, 0x55) because 1) different DRAM cells store information (i.e., 0 or 1) in different states (i.e., charged or empty) [61] and 2) we would like to stress DRAM reliability by increasing the interference between adjacent bitlines [43, 44, 61]. We also perform tests using different refresh intervals, to study whether the variation in charge leakage increases significantly if the time between refreshes increases.

tRCD Test. We measure how highly-charged cells affect the `tRCD` timing parameter, by using a custom `tRCD` value to read data from a row to which we previously wrote a reference data pattern. We adjust the time between writing a reference data pattern and performing the read, to vary the amount of charge stored within the cells of a row. In Figure 6a, we show the command sequence that we use to test whether recently-refreshed DRAM cells can be accessed with a lower `tRCD`, compared to cells that are close to the end of the refresh interval. We perform the write and read operations to each DRAM row one column at a time, to ensure that each read incurs the `tRCD` latency. First (① in Figure 6a), we perform a reference write to the DRAM column under test by issuing `ACTIVATE`, `WRITE`, and `PRECHARGE` successively with the *default* DRAM timing parameters. Next (②), we wait for the duration of a time interval (**T1**), which is the refresh interval in practice, to vary the charge contained in the cells. When we wait longer, we expect the target cells to have less charge at the end of the interval. We cover a wide range of wait intervals, evaluating values between 1 and 512 ms. Finally (③), we read the data from the column that we previously wrote to and compare it with the reference pattern. We perform the read with the custom `tRCD` value for that specific test. We evaluate `tRCD` values ranging from 3 to 6 (default) cycles. Since a `tRCD` of 3 cycles produced errors in every run, we did not perform any experiments with a lower `tRCD`.

We process multiple rows in an interleaved manner (i.e., we write to multiple rows, wait, and then verify their data one after another) in order to further stress the reliability of DRAM [56]. We repeat this process for all DRAM rows to evaluate the entire memory module.

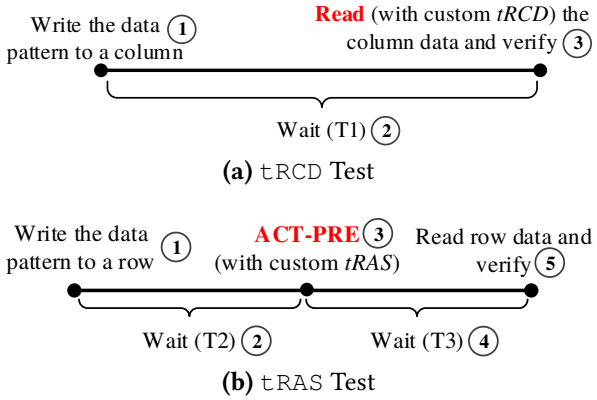


Figure 6: Timelines that illustrate the methodology for testing the improvement of (a) `tRCD` and (b) `tRAS` on highly-charged DRAM cells.

tRAS Test. We measure the effect of accessing highly-charged rows on the `tRAS` timing parameter by issuing the `ACTIVATE` and `PRECHARGE` commands, with a custom `tRAS` value, to a row. We check if that row still contains the same data that it held before the `ACTIVATE-PRECHARGE` command pair was issued. Figure 6b illustrates the methodology for testing the effect of the refresh interval on `tRAS`. First (①), we write the reference data pattern to the selected DRAM row with the default timing parameters. Different from the `tRCD` test, we write to *every column* in the open row (before switching to another row) to save cycles by eliminating a significant amount of `ACTIVATE` and `PRECHARGE` commands, thereby reducing the testing time. Next (②), we wait for the duration of time interval **T2**, during which the DRAM cells lose a certain amount of charge. To refresh the cells (③), we issue an `ACTIVATE-PRECHARGE` command pair associated with a custom `tRAS` value. When the `ACTIVATE-PRECHARGE` pair is issued, the charge in the cells of the target DRAM row may not be fully restored if the wait time is too long or the `tRAS` value is too short, potentially leading to loss of data. Next (④), we wait again for a period of time **T3** to allow the cells to leak a portion of their charge. Finally (⑤), we read the row using the default

timing parameters and test whether it still retains the correct data. Similar to the tRCD test, to stress the reliability of DRAM, we simultaneously perform the tRAS test on multiple DRAM rows.

We would expect, from this experiment, that the data is likely to maintain its integrity when evaluating reduced tRAS with shorter wait times (**T2**), as the higher charge retained in a cell with a short wait time can enable a reliable reduction on tRAS. In contrast, we would expect failures to be more likely when using a reduced tRAS with a longer wait time, because the cells would have a low amount of charge that is not enough to reliably reduce tRAS.

6.2.2. Results. We analyze the results of the tRCD and tRAS tests, for 24 real DRAM chips from different vendors, using the test programs detailed in Section 6.2.1. We evaluate tRCD values ranging from 3 to 6 cycles, and tRAS values ranging from 2 to 14 cycles, where the maximum number for each is the default timing parameter value. For both tests, we evaluate refresh intervals between 8 and 512 ms and measure the number of observed errors during each experiment.

Figures 7 and 8 depict the results for the tRCD test and the tRAS test, respectively, for three DRAM modules (each from a different DRAM vendor). We make three major observations:

*(i) Within the duration of the standard refresh interval (*64 ms*), DRAM cells do not leak a sufficient amount of charge to have a negative impact on DRAM access latency.*[3] For refresh intervals less than or equal to 64 ms, we observe little to no variation in the number of errors induced. Within this refresh interval range, depending on the tRCD or tRAS value, the errors generated are either zero or a constant number. We make the same observation in both the tRCD and tRAS tests for all three DRAM modules.

For all the modules tested, using different data patterns and stressing DRAM operation with temperatures significantly higher than the common-case operating conditions, we can significantly reduce tRCD and tRAS parameters, without observing any errors. We observe errors only when tRCD and tRAS parameters are too small to correctly perform the DRAM access, regardless of the charge amount of the accessed cells.

(ii) The large safety margin employed by the manufacturers protects DRAM against errors even when accessing DRAM cells with low latency. We observe no change in the number of induced errors for tRCD values less than the default of 6 cycles (down to 4 cycles in modules A and B, and 5 cycles in module C). We observe a similar trend in the tRAS test: tRAS can be reduced from the default value of 14 cycles to 5 cycles without increasing the number of induced errors for *any* refresh interval.

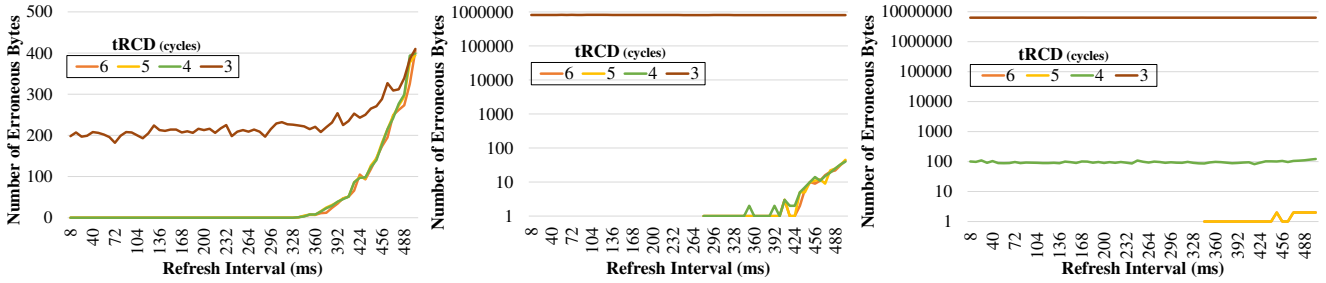
We conclude that even at temperatures much higher than typical operating conditions, there exists a large safety margin for access latency in existing DRAM chips. This demonstrates that DRAM cells are much *stronger* than their timing specifications indicate.[4] In other words, the timing margin in most DRAM cells is very large, given the existing timing parameters.

(iii) The expected effect of ChargeCache and NUAT, that highly-charged cells can be accessed with lower latency, is slightly observable only when very long refresh intervals are used. For each of the tests, we observe a significant increase in the number of errors at refresh intervals that are much higher than the typical refresh interval of 64 ms, demonstrating the variation in charge held by each of the DRAM cells. Based on the assumptions made by ChargeCache and NUAT, we expect that when lower values of tRCD and tRAS are employed, the error rate should increase more rapidly. However, we find that for all but the minimum values of tRCD and tRAS (and for tRCD = 4 for module C), the tRCD and tRAS latencies have almost no impact on the error rate.

We believe that the reason we cannot observe the expected latency reduction effect of ChargeCache and NUAT on exist-

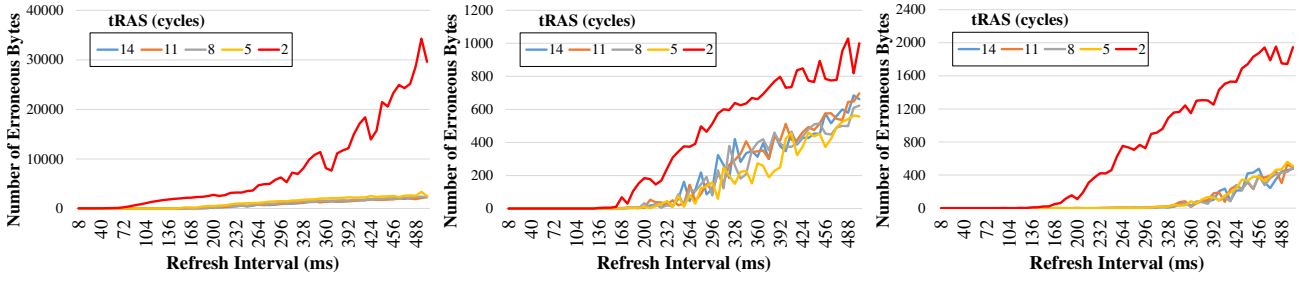
[3]Other studies have shown methods to take advantage of the fact that latencies can be reduced without incurring errors [18, 56].

[4]Similar observations were made by prior work [17, 18, 56].



(a) Module A **(b)** Module B **(c)** Module C

Figure 7: Effect of reducing tRCD on the number of errors at various refresh intervals.



(a) Module A **(b)** Module B **(c)** Module C

Figure 8: Effect of reducing tRAS on the number of errors at various refresh intervals.

ing DRAM modules is due to the internal behavior of existing DRAM chips that does not allow latencies to be reduced beyond a certain point: we cannot *externally control* when the sense amplifier gets enabled, since this is dictated with a fixed latency internally, regardless of the charge amount in the cell. The sense amplifiers are enabled only after charge sharing, which starts by enabling the wordline and lasts until sufficient amount of charge flows from the activated cell into the bitline (See Section 2.2), is expected to complete. Within existing DRAM chips, the expected charge sharing latency (i.e., the time when the sense amplifiers get enabled) is not represented by a timing parameter managed by the memory controller. Instead, the latency is controlled internally within the DRAM using a fixed value [40, 94]. ChargeCache and NUAT require that charge sharing completes in less time, and the sense amplifiers get enabled faster for a highly-charged cell. However, since existing DRAMs provide no way to control the time it takes to enable the sense amplifiers, we cannot harness the potential latency reduction possible for highly-charged cells [94]. Reducing `tRCD` affects the time spent *only after charge sharing*, at which point the bitline voltages exhibit similar behavior regardless of the amount of charge initially stored within the cell. Consequently, we are unable to observe the expected latency reduction effect of ChargeCache and NUAT by simply reducing `tRCD`, even though we believe that the mechanisms are sound and can reduce latency (assuming the behavior of DRAM chips is modified). If the DDR interface exposes a method of controlling the time it takes to enable the sense amplifiers in the future, SoftMC can be easily modified to use the method and fully evaluate the latency reduction effect of ChargeCache and NUAT.

Summary. Overall, we make two major conclusions from the implementation and experimental results of our DRAM latency experiments. First, SoftMC provides a simple and easy-to-use interface to quickly implement tests that characterize modern DRAM chips. Second, SoftMC is an effective tool to validate or refute the expected effect of existing or new mechanisms on existing DRAM chips.

7. Limitations of SoftMC

In the previous section, we presented and discussed some examples demonstrating the benefits of SoftMC. These use cases illustrate the flexibility, ease of use, and capability of SoftMC in characterizing DRAM operations and evaluating new DRAM mechanisms (using the DDR interface). Although our SoftMC prototype already caters for a wide range of use cases, it also has some limitations that arise mainly from its current hardware implementation.

Inability to Evaluate System Performance. Studies where the applications run on the host machine and access memory within the module under test can demonstrate the impact of memory reliability and latency on system performance using real applications. However, such studies are difficult to perform in any FPGA-based DRAM testing infrastructure that connects the host and the FPGA over the PCIe bus (including SoftMC). This is due to the the long latency associated with the PCIe bus (in microseconds [4, 37, 51]; in contrast to DRAM access latency that is within 15-80 ns [57, 89]). Note that this long PCIe bus latency does *not* affect our tests (as explained in Section 5.1), but it would affect the performance of a system that would use SoftMC as the main memory controller. One way to enable performance studies with SoftMC is to add support for trace-based execution, where the traces (i.e., memory access requests) are collected by executing workloads on real systems or simulating them. The host PC can transform the traces into SoftMC instructions and transmit them to SoftMC hardware, where we can buffer the instructions and emulate microarchitectural behavior of a memory controller developed using SoftMC while avoiding the PCIe bus latency. This functionality requires coordination between the system/simulator and SoftMC, and we expect to add such support by enabling coordination between our DRAM simulator, Ramulator [50, 80], and a future release of SoftMC.

Coarse-Grained Timing Resolution. As FPGAs are significantly slower than the memory bus, the minimum time interval between two consecutive commands that the FPGA can send to the memory bus is limited by the FPGA frequency. In our current prototype, the DDR interface of the ML605 operates at 400MHz, allowing SoftMC to issue two consecutive commands with a minimum interval of 2.5 ns. Therefore, the timing parameters in SoftMC can be changed only at the granularity of 2.5 ns. However, it is possible to support finer-granularity resolution with faster FPGA boards [102].

Limitation on the Number of Instructions Stored in the FPGA. The number of instructions in one test program (that is executed atomically in SoftMC) is limited by the size of the *Instruction Queue* in SoftMC. To keep FPGA resource usage low, the size of the *Instruction Queue* in the current SoftMC prototype is limited to 8192 instructions. In the future, instead of increasing the size of the *Instruction Queue*, which would increase resource utilization in the FPGA, we plan to extend the SoftMC hardware with control flow instructions to support arbitrary-length tests. Adding control flow instructions would enable loops that can iterate over large structures (e.g., DRAM rows, columns) in hardware with a small number of SoftMC instructions, avoiding the need for the host machine to issue a large number of instructions in a loop-unrolled manner. Such control flow support would further improve the ease of use of SoftMC.

8. Research Directions Enabled by SoftMC

We believe SoftMC can enable many new studies of the behavior of DRAM and other memories. We briefly describe several examples in this section.

More Characterization of DRAM. The SoftMC DRAM testing infrastructure can test any DRAM mechanism consisting of low-level DDR commands. Therefore, it enables a wide range of characterization and analysis studies of real DRAM modules that would otherwise not have been possible without such an infrastructure. We discuss three such example research directions.

First, as DRAM scales down to smaller technology nodes, it faces key challenges in both reliability and latency [38, 43–45, 60, 61, 65, 68, 71]. Unfortunately, there is no comprehensive experimental study that characterizes and analyzes the trends in DRAM cell operations and behavior with technology scaling across various DRAM generations. The SoftMC infrastructure can help us answer various questions to this end: How are the cell characteristics, reliability, and latency changing with different generations of technology nodes? Do all DRAM operations and cells get affected by scaling at the same rate? Which DRAM operations are getting worse?

Second, aging-related failures in DRAM can potentially affect the reliability and availability of systems in the field [66, 84]. However, the causes, characteristics, and impact of *aging* have remained largely unstudied. Using SoftMC, it is possible to devise controlled experiments to analyze and characterize aging. The SoftMC infrastructure can help us answer questions such as: How prevalent are aging-related failures? What types of usage accelerate aging? How can we design architectural techniques that can slow down the aging process?

Third, prior works show that the failure rate of DRAM modules in large data centers is significant, largely affecting the cost and downtime in data centers [64, 66, 84, 90]. Unfortunately, there is no study that analyzes DRAM modules that have failed in the field to determine the common causes of failure. Our

SoftMC infrastructure can test faulty DRAM modules and help answer various research questions: What are the dominant types of DRAM failures at runtime? Are failures correlated to any location or specific structure in DRAM? Do all chips from the same generation exhibit the same failure characteristics?

Characterization of Non-Volatile Memory. The SoftMC infrastructure can test any chip compatible with the DDR interface. Such a design makes the scope of the chips that can be tested by SoftMC go well beyond just DRAM. With the emergence of byte addressable non-volatile memories (e.g., PCM [53–55, 81], STT-RAM [39, 52], ReRAM [2, 97]), several vendors are working towards manufacturing DDR-compatible non-volatile memory chips at a large scale [23, 67]. When these chips become commercially available, it will be critical to characterize and analyze them in order to understand, exploit, and/or correct their behavior. We believe that SoftMC can be seamlessly used to characterize these chips, and can help enable future mechanisms for NVM.

SoftMC will hopefully enable other works that build on it in various ways. For example, future work can extend the infrastructure to enable researchers to analyze memory scheduling (e.g., [22, 48, 49, 70, 72, 73, 92]) and memory power management [20, 21] mechanisms, and allow them to develop new mechanisms using a programmable memory controller and real workloads. We conclude that characterization with SoftMC enables a wide range of research directions in DDR-compatible memory chips (DRAM or NVM), leading to better understanding of these technologies and helping to develop mechanisms that improve the reliability and performance of future memory systems.

9. Other Related Work

Although no prior work provides an open-source DRAM testing infrastructure similar to SoftMC, infrastructures for testing other types of memories have been developed. Cai et al. [8] developed a platform for characterizing NAND flash memory. They propose a flash controller, implemented on an FPGA, to quickly characterize error patterns of existing flash memory chips. They expose the functions of the flash translation layer (i.e., the flash chip interface) to the software developer via the host machine connected to the FPGA board, similar to how we expose the DDR interface to the user in SoftMC. Many works [7, 9–16, 62, 63] use this flash memory testing infrastructure to study various aspects of flash chips.

Our prior works [18, 42–44, 46, 56, 58, 61] developed and used FPGA-based infrastructures for a wide range of DRAM studies. Liu et al. [61] and Khan et al. [42] analyzed the data retention behavior of modern DRAM chips and proposed mechanisms for mitigating retention failures. Khan et al. [43, 44] studied data-dependent failures in DRAM, and developed techniques for efficiently detecting and handling them. Lee et al. [56, 58] analyzed latency characteristics of modern DRAM chips and proposed mechanisms for latency reduction. Kim et al. [46] discovered a new reliability issue in existing DRAM, called *RowHammer*, which can lead to security breaches [25, 82, 85, 86, 95, 98]. Chang et al. [18] used SoftMC to characterize latency variation across DRAM cells for fundamental DRAM operations (e.g., activation, precharge). SoftMC evolved out of these previous infrastructures, to address the need to make the infrastructure flexible and easy to use.

10. Conclusion

This work introduces the first publicly-available FPGA-based DRAM testing infrastructure, *SoftMC* (Soft Memory Controller), which provides a programmable memory controller with a flexible and easy-to-use software interface. SoftMC enables the flexibility to test any standard DRAM operation and any (existing or new) mechanism comprising of such operations. It provides an intuitive high-level software interface for the user to invoke low-level DRAM operations, in order to minimize programming effort and time. We provide a prototype implementation of SoftMC, and we have released it publicly as a freely-available open-source tool [88].

We demonstrate the capability, flexibility, and programming ease of SoftMC by implementing two example use cases. Our experimental analyses demonstrate the effectiveness of SoftMC as a new tool to *(i)* perform detailed characterization of various DRAM parameters (e.g., refresh interval and access latency) as well as the relationships between them, and *(ii)* test the expected effects of existing or new mechanisms (e.g., whether or not highly-charged cells can be accessed faster in existing DRAM chips). We believe and hope that SoftMC, with its flexibility and ease of use, can enable many other studies, ideas and methodologies in the design of future memory systems, by making memory control and characterization easily accessible to a wide range of software and hardware developers.

Acknowledgments

We thank the reviewers, the SAFARI group members, and Shigeki Tomishima from Intel for their feedback. We acknowledge the generous support of Google, Intel, Nvidia, Samsung, and VMware. This work is supported in part by NSF grants 1212962, 1320531, and 1409723, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

[1] Advantest, "V6000 Memory Platform," https://www.advantest.com/products/ic-test-systems/v6000-memory.
[2] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proc. IEEE*, 2010.
[3] P. Bernardi *et al.*, "A Programmable BIST for DRAM Testing and Diagnosis," in *ITC*, 2010.
[4] R. Bittner *et al.*, "Direct GPU/FPGA Communication via PCI Express," *Cluster Computing*, 2014.
[5] M. N. Bojnordi and E. Ipek, "PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards," in *ISCA*, 2012.
[6] S. Borkar and A. A. Chien, "The Future of Microprocessors," in *CACM*, 2011.
[7] Y. Cai *et al.*, "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques," in *HPCA*, 2017.
[8] Y. Cai *et al.*, "FPGA-Based Solid-State Drive Prototyping Platform," in *FCCM*, 2011.
[9] Y. Cai *et al.*, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
[10] Y. Cai *et al.*, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.
[11] Y. Cai *et al.*, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *DSN*, 2015.
[12] Y. Cai *et al.*, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.
[13] Y. Cai *et al.*, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," in *ICCD*, 2013.
[14] Y. Cai *et al.*, "Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime," in *ICCD*, 2012.
[15] Y. Cai *et al.*, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," *ITJ*, 2013.
[16] Y. Cai *et al.*, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
[17] K. Chandrasekar *et al.*, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
[18] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[19] K. K.-w. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[20] H. David *et al.*, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *ICAC*, 2011.
[21] Q. Deng *et al.*, "MemScale: Active Low-Power Modes for Main Memory," in *ASPLOS*, 2011.
[22] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[23] EverSpin, "ST-MRAM," https://www.everspin.com/mram-replaces-dram.
[24] FuturePlus, "FS2800 DDR Detective," http://www.futureplus.com/DDR-Detective-Standalone/summary-2800.html.

[25] D. Gruss *et al.*, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *arXiv*, 2015.
[26] T. Hamamoto *et al.*, "On the Retention Time Distribution of Dynamic Random Access Memory (DRAM)," *Electron Devices*, 1998.
[27] H. Hassan *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[28] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Comprehensive Experimental Characterization of Existing DRAM Chips," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. 2017-001, 2017.
[29] P. Hazucha and C. Svensson, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate," *TNS*, 2000.
[30] H. Hidaka *et al.*, "Twisted Bit-Line Architectures for Multi-Megabit DRAMs," *JSSC*, 1989.
[31] C. Hou *et al.*, "An FPGA-Based Test Platform for Analyzing Data Retention Time Distribution of DRAMs," in *VLSI-DAT*, 2013.
[32] J. Huang *et al.*, "An FPGA-Based Re-Configurable Functional Tester for Memory Chips," in *ATS*, 2000.
[33] M. Jacobsen *et al.*, "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," *TRETS*, 2015.
[34] JEDEC, "DDR3 SDRAM Standard," *JESD79-3*, 2007.
[35] JEDEC, "Low Power Double Data Rate 3 (LPDDR4)," Standard No. JESD209-4, 2014.
[36] T. S. Jung, "Memory Technology and Solutions Roadmap," http://www.sec.co.kr/images/corp/ir/irevent/techforum_01.pdf, 2005.
[37] E. Kadric *et al.*, "An FPGA Implementation for a High-Speed Optical Link with a PCIe Interface," in *SoC*, 2012.
[38] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
[39] T. Kawahara *et al.*, "2 Mb SPRAM (SPin-Transfer Torque RAM) with bit-by-bit bi-directional current write and parallelizing-direction current read," *JSSC*, 2008.
[40] B. Keeth, "DRAM Circuit Design: Fundamental and High-Speed Topics." John Wiley & Sons, 2008, pp. 26–31.
[41] D. Keezer *et al.*, "An FPGA-Based ATE Extension Module for Low-Cost Multi-GHz Memory Test," in *ETS*, 2015.
[42] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[43] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[44] S. Khan *et al.*, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
[45] K. Kim, "Technology for Sub-50nm DRAM and NAND Flash Manufacturing," in *IEDM*, 2005.
[46] Y. Kim *et al.*, "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[47] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[48] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[49] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[50] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2015.
[51] M. J. Koop *et al.*, "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate Infiniband," in *HOTI*, 2008.
[52] E. Kultursay *et al.*, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
[53] B. C. Lee *et al.*, "Phase Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
[54] B. C. Lee *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
[55] B. C. Lee *et al.*, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, 2010.
[56] D. Lee *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[57] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[58] D. Lee *et al.*, "Reducing DRAM Latency by Exploiting Design-Induced Latency Variation in Modern DRAM Chips," *arXiv*, 2016.
[59] M. Lee and K. W. Park, "A Mechanism for Dependence of Refresh Time on Data Pattern in DRAM," *EDL*, 2010.
[60] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[61] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[62] Y. Luo *et al.*, "WARM: Improving NAND Flash Memory Lifetime with Write-Hotness Aware Retention Management," in *MSST*, 2015.
[63] Y. Luo *et al.*, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
[64] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
[65] J. Mandelman *et al.*, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," *IBM JRD*, 2002.
[66] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[67] Micron, "3D XPoint Memory," http://www.micron.com/about/innovations/3d-xpoint-technology, 2016.
[68] W. Mueller *et al.*, "Challenges for the DRAM Cell Scaling to 40nm," in *IEDM*, 2005.
[69] J. Mukundan *et al.*, "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems," in *ISCA*, 2013.
[70] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[71] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
[72] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[73] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[74] Y. Nakagome *et al.*, "The Impact of Data-Line Interference Noise on DRAM Scaling," *JSSC*, 1988.
[75] S. Nassif, "Delay Variability: Sources, Impacts and Trends," in *ISSCC*, 2000.
[76] Nickel Electronics, "DRAM Memory Testing," https://www.nickelelectronics.com/memory-testing/.
[77] B. Querbach *et al.*, "A Reusable BIST with Software Assisted Repair Technology for Improved Memory and IO Debug, Validation and Test Time," in *ITC*, 2014.
[78] B. Querbach *et al.*, "Architecture of a Reusable BIST Engine for Detection and Auto Correction of Memory Failures and for IO Debug, Validation, Link Training, and Power Optimization on 14nm SOC," *IEEE D&T*, 2016.
[79] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[80] Ramulator Source Code, https://github.com/CMU-SAFARI/ramulator.
[81] S. Raoux *et al.*, "Phase-Change Random Access Memory: A Scalable Technology," *IBM JRD*, 2008.
[82] K. Razavi *et al.*, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Sec.*, 2016.
[83] M. Redeker *et al.*, "An Investigation into Crosstalk Noise in DRAM Structures," in *MTDT*, 2002.
[84] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS*, 2009.
[85] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.
[86] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," in *Black Hat*, 2015.
[87] W. Shin *et al.*, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
[88] SoftMC Source Code, https://github.com/CMU-SAFARI/SoftMC.
[89] Y. Son *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," *ISCA*, 2013.
[90] V. Sridharan *et al.*, "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults," in *SC*, 2013.
[91] G. R. Srinivasan *et al.*, "Accurate, Predictive Modeling of Soft Error Rate due to Cosmic Rays and Chip Alpha Radiation," in *IRPS*, 1994.
[92] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[93] Teradyne, "Magnum Memory Test System," http://www.teradyne.com/products/semiconductor-test/magnum.
[94] S. Tomishima, Personal Communication, Dec. 2016.
[95] V. van der Veen *et al.*, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.
[96] R. Venkatesan *et al.*, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.
[97] H.-S. P. Wong *et al.*, "Metal-Oxide RRAM," *Proc. IEEE*, 2012.
[98] Y. Xiao *et al.*, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Sec.*, 2016.
[99] Xilinx, "Virtex-6 FPGA Integrated Block for PCI Express," http://www.xilinx.com/support/documentation/user_guides/v6_pcie_ug517.pdf.
[100] Xilinx, "Virtex-6 FPGA Memory Interface Solutions," http://www.xilinx.com/support/documentation/ip_documentation/mig/v3_92/ug406.pdf.
[101] Xilinx, *ML605 Hardware User Guide*, Oct. 2012.
[102] Xilinx, *VC709 FPGA User Guide*, Aug. 2016.
[103] D. Yaney *et al.*, "A Meta-Stable Leakage Phenomenon in DRAM Charge Storage - Variable Hold Time," in *IEDM*, 1987.
[104] C. Yang *et al.*, "A Hybrid Built-In Self-Test Scheme for DRAMs," in *VLSI-DAT*, 2015.
[105] H. Yang *et al.*, "Random Pattern Generation for Post-Silicon Validation of DDR3 SDRAM," in *VTS*, 2015.
[106] Y. You and J. Hayes, "A Self-Testing Dynamic RAM Chip," *TED*, 1985.