DRAM Bender: An Extensible and Versatile FPGA-Based Infrastructure to Easily Test State-of-the-Art DRAM Chips

Ataberk Olgun\textsuperscript{a}, Graduate Student Member, IEEE, Hasan Hassan, A. Giray Yağılıkçı, Yahya Can Tuğrul\textsuperscript{a}, Lois Orosa\textsuperscript{b}, Member, IEEE, Haocong Luo, Minesh Patel\textsuperscript{b}, Öğuz Ergin, and Onur Mutlu\textsuperscript{b}, Fellow, IEEE

Abstract—To understand and improve DRAM performance, reliability, security, and energy efficiency, prior works study characteristics of commodity DRAM chips. Unfortunately, state-of-the-art open source infrastructures capable of conducting such studies are obsolete, poorly supported, or difficult to use, or their inflexibility limits the types of studies they can conduct. We propose DRAM Bender, a new FPGA-based infrastructure that enables experimental studies on state-of-the-art DRAM chips. DRAM Bender offers three key features at the same time. First, DRAM Bender enables directly interfacing with a DRAM chip through its low-level interface. This allows users to issue DRAM commands in arbitrary order and with fine-grained time intervals compared to other open source infrastructures. Second, DRAM Bender exposes easy-to-use C++ and Python programming interfaces, allowing users to quickly and easily develop different types of DRAM experiments. Third, DRAM Bender is easily extensible. The modular design of DRAM Bender allows extending it to: 1) support existing and emerging DRAM interfaces and 2) run on new commercial or custom FPGA boards with little effort. To demonstrate that DRAM Bender is a versatile infrastructure, we conduct three case studies, two of which lead to new observations about the DRAM RowHammer vulnerability. In particular, we show that data patterns supported by DRAM Bender uncover a larger set of bit-flips on a victim row than those commonly used by prior work. We demonstrate the extensibility of DRAM Bender by implementing it on five different FPGAs with DDR4 and DDR3 support. DRAM Bender is freely and openly available at https://github.com/CMU-SAFARI/DRAM-Bender.

Index Terms—DRAM, energy, FPGA, memory systems, open source, performance, reliability, security, testing.

I. INTRODUCTION

DRAM [19] is the dominant technology used in building the main memory of computer systems as it provides low access latency and can be manufactured at low cost per bit. Challenges in DRAM technology scaling render continuous improvement of DRAM performance, energy efficiency, security, and reliability difficult [91], [92]. To improve DRAM in all aspects and overcome DRAM scaling challenges, it is critical to experimentally understand the operation and characteristics of real DRAM chips. As such, it is critical to develop experimental testing infrastructures to efficiently and easily test real state-of-the-art DRAM chips. We refer to these infrastructures as DRAM testing infrastructures. These infrastructures enable at least two major possibilities.

Understanding DRAM Scaling Trends: Technology node scaling affects various DRAM characteristics, such as DRAM cell capacitance and capacitive crosstalk, negatively impacting the reliability, latency, and security characteristics of DRAM [52], [81], [91]. Therefore, it is important to thoroughly analyze these effects to understand how real DRAM chip characteristics change with technology scaling. Using DRAM testing infrastructures, prior studies uncover insights that inspire new mechanisms/methodologies for improving DRAM access latency [13], [14], [16], [18], [32], [61], [70], [71], [72], [130], [139], improving DRAM energy efficiency [17], [30], [77], [111], [116], understanding and mitigating the Rowhammer problem [64], [65], [92], [93], [107], [152], and understanding DRAM data retention failures [29], [57], [60], [76], [77], [111], [116], [136]. Such understanding is very difficult, if not impossible, to develop without testing real chips.

Uncovering Undocumented Functionality: Standard DRAM interfaces define constraints (e.g., timing parameters) that the memory controller obeys to guarantee correct DRAM operation. These constraints, by design, preclude the discovery of various undocumented operations that the DRAM technology or the chip design may inherently be capable of. For example, recent works demonstrate techniques that enable several in-DRAM computation primitives (e.g., bulk data copy [23], [102], [125], bitwise operations [23], [124], [126], true random number generation [6], [63], [104], and physical unclonable functions [6], [62], [106], [132]) using off-the-shelf DRAM devices by violating the DRAM timing parameters.

Manuscript received 4 December 2022; revised 23 March 2023; accepted 6 May 2023. Date of publication 1 June 2023; date of current version 12 December 2023. This work was supported in part by the Google Security and Privacy Research Award and in part by the Microsoft Swiss Joint Research Center. This article was recommended by Associate Editor X. Li. (Corresponding author: Ataberk Olgun.)
Such undocumented DRAM behavior is possible to discover and demonstrate only by testing real DRAM chips using infrastructures that can violate DRAM timing parameters.

**Limitations of Existing Infrastructures:** DRAM timing parameters cannot be freely modified or violated in widely available computer systems. In such systems, the CPU sends memory requests (e.g., load/store) to the memory controller to access DRAM. The memory controller translates the memory request to appropriate low-level DRAM commands that strictly comply with the interface specification (i.e., DRAM standard) of the DRAM chip (i.e., the memory controller does not violate DRAM timing parameters). Thus, the two major research directions cannot be explored using typical computer systems as such exploration would require modifications to existing memory controllers.

To overcome the limitations of existing computer systems, prior works develop DRAM testing infrastructures that allow issuing DRAM commands with modified/violated timing parameters to DRAM chips. Unfortunately, the two existing FPGA-based open source DRAM testing infrastructures (SoftMC [34] and LiteX RowHammer tester (LRT) [2]) suffer from critical issues that prevent researchers from easily pursuing the above research directions. These infrastructures: 1) do not fully expose the DRAM interface to test programs (i.e., impose restrictions on the DRAM interface that limit studies that can be performed); 2) are fairly difficult to use; or 3) are difficult to extend to support new DRAM interfaces.

**Interface Restrictions:** Both SoftMC and LRT expose a restricted view of the DRAM interface to test programs due to the limited capability provided by their instruction set architectures (ISAs). First, SoftMC imposes restrictions on data patterns that can be used in initializing DRAM rows. Second, SoftMC imposes a nondeterministic delay in the order of microseconds between different parts of a DRAM test program. Third, LRT imposes 10 ns of delay between successive DRAM commands (i.e., a DRAM timing parameter can only be violated down to 10 ns and many standard DRAM timing parameters [44], [48], [49] cannot even be violated). These restrictions limit the types of experimental studies supported by both platforms.

**Ease of Use:** SoftMC and LRT are difficult to use. First, SoftMC 1) is only prototyped on a discontinued FPGA board and 2) has run-time dependencies that are obsolete and challenging to set up. Therefore, obtaining or setting up a system that readily supports SoftMC to conduct experimental studies is difficult. Second, LRT requires users to implement a new hardware module that post-processes experimental data read from DRAM (by using a hardware description language (HDL) such as Verilog) for every new type of experiment. Developing a hardware module is considerably difficult compared to developing a computer program for postprocessing and requires expertise in HDL. As such, LRT cannot be easily used as a general DRAM testing infrastructure.

**Extensibility:** SoftMC 1) supports only DDR3 and has a hardware design that is not modular and 2) has a deprecated development environment that consists of outdated hardware design tools. Therefore, it cannot be easily modified to support DDR4 or other current and future DRAM interfaces.

**Our Goal** is to develop an infrastructure that is superior to past infrastructures by overcoming their shortcomings. To this end, we design and develop DRAM Bender. DRAM Bender 1) fully exposes the DRAM interface to test programs through its nonrestrictive ISA; 2) provides easy-to-use C++ and Python programming interfaces, and open source working prototypes of itself on five different modern FPGA boards; and 3) has a modular design and provides the necessary HW/SW components to facilitate the integration of new DRAM standards. Table I presents a qualitative comparison of DRAM Bender against the two other open source DRAM testing infrastructures available today.

**Key Results:** We 1) evaluate DRAM Bender’s versatility in types of experimental studies on DRAM (Section IV); 2) demonstrate DRAM Bender’s ease of use (Sections IV-A and IV-B); and 3) evaluate DRAM Bender’s extensibility to new DRAM interfaces (Section III-D2). First, leveraging DRAM Bender’s versatility in conducting experimental studies on DRAM, we uncover two new insights into the RowHammer problem [65], [95] (Sections IV-A and IV-B) and demonstrate that processing-using-memory capability (i.e., bulk bitwise execution capability) exists in real DDR4 chips (Section IV-C). Our results show that: 1) the effectiveness of a double-sided RowHammer attack at causing bit-flips highly depends on the order in which the two aggressor rows are activated and precharged and 2) data patterns supported by DRAM Bender can uncover a larger set of bit-flips in a victim row compared to the data patterns commonly used by prior work. Second, we demonstrate DRAM Bender’s ease of use by showing that it can be used to perform a RowHammer experiment in 12 lines of C++ code and a bulk bitwise AND/OR operation in just three lines of C++ code, in working prototypes of DRAM Bender on five different modern FPGA boards. Third, we demonstrate that DRAM Bender is easy to extend (Section III-D2).1 The required source code modifications to port DRAM Bender to another FPGA board with a different DRAM interface are small (only 230 lines of additional Verilog and 30 lines of additional C++ code).2

1We have more detailed results and explanations for all three case studies in an extended version of this article [101], which also discusses new research & development directions based on DRAM Bender.

2On top of the initial implementation of DRAM Bender with DDR4 support that comprises approximately 3400 lines of Verilog and 2000 lines of C++ code.
DRAM Bender: a) fully exposes the DRAM interface; b) is easy to use; and c) is easy to extend and modify. DRAM Bender is the only general open source testing infrastructure to easily test DDR4 chips today.

2) We make two new observations on RowHammer-induced bit-flips, a major reliability, safety, and security problem in modern DRAM chips. These observations can be used to construct better RowHammer attacks and defenses.

3) We demonstrate that contemporary off-the-shelf DDR4 devices are capable of performing in-DRAM bitwise Majority, AND, and OR operations.

4) We demonstrate the extensibility of DRAM Bender by developing its prototypes for five FPGA boards, which have either DDR4 DIMM/SODIMM or DDR3 SODIMM slots.

5) We open source DRAM Bender [118] to facilitate research on understanding the operation and characteristics of real DRAM chips. We also open source [119], [120] two prior works, U-TRR [33], and QUAC-TRNG [104], as example works DRAM Bender has enabled.

II. BACKGROUND AND MOTIVATION

We provide a concise background on DRAM organization, cell access, and timing parameters in this section. We refer the reader to many prior works for more detailed background on these subjects [14], [15], [16], [17], [21], [25], [30], [31], [32], [34], [40], [53], [57], [58], [59], [61], [62], [63], [64], [65], [66], [67], [70], [72], [73], [76], [77], [79], [90], [96], [97], [100], [106], [109], [110], [111], [116], [125], [126], [127], [128], [131], [139], [140], [159].

A. DRAM Organization

DRAM-based main memory is typically organized hierarchically, as depicted in Fig. 1. A memory controller in the CPU is connected to a DRAM module over a DRAM channel. Each module on a DRAM channel comprises multiple DRAM chips, where a set of chips that operate concurrently is referred to as a DRAM rank. A DRAM chip is partitioned into multiple DRAM banks that operate independently. A collection of DRAM cells, row decoders, and row buffers (sense amplifier arrays) constitute a DRAM bank.

DRAM cells that are laid out onto the same wordline are accessed simultaneously, and referred to as a row. Each cell in a row is connected to a different bitline over an access transistor. A bitline connects one DRAM cell from each row to a sense amplifier in the row buffer. A DRAM cell stores data as charge in a capacitor. A sense amplifier can read or manipulate a DRAM cell’s value by sampling or driving a bitline.

B. Accessing DRAM Cell

Initially, the bitline is precharged to a reference voltage level (V_{DD}/2). The memory controller sends an activate (ACT) command to a row to enable the wordline. This starts the charge-sharing process, where the DRAM cell on the wordline shares its charge with its bitline. The sense amplifier samples the bitline voltage and detects a deviation toward V_{DD} or 0, depending on the value stored in the DRAM cell, and amplifies it. The DRAM cell can then be read or its value updated using READ/WRITE commands until the bank is finally precharged (PRE) to access another row.

C. DRAM Timing Parameters

Each DRAM standard defines a set of timing parameters. A timing parameter specifies the minimum time between two DRAM commands to guarantee correct operation. We describe two relevant timing parameters:

tRP: Time window between an ACT and the next PRE command. During tRP, charge of the DRAM cells on the open DRAM row is fully restored.

C. RowHammer

RowHammer, first introduced in 2014 [65], is a circuit-level phenomenon in DRAM devices, where frequent activation of an aggressor DRAM row induces bit-flips on DRAM cells in nearby victim rows. RowHammer is an important DRAM security/reliability/safety vulnerability that becomes more severe as DRAM devices scale to smaller technology nodes [64], [65], [107]. A RowHammer attack can flip critical bits in memory and can be used to escalate privilege and break confidentiality in real systems (as shown in, e.g., [27], [69], [123], and [135]).

F. FPGA-Based DRAM Testing Infrastructures

SoftMC [34], [121], and LRT [2] are the only two open source FPGA-based DRAM testing infrastructures that exist today.

SoftMC [34], [121] is an open source DDR3 testing infrastructure. SoftMC issues DDR3 command traces to the DDR3 device under test. A host machine transmits these command traces to the FPGA board that implements SoftMC.

LRT [2] is a RowHammer testing infrastructure that is primarily designed to conduct RowHammer experiments on
DDR3/4 and LPDDR4 chips. LRT receives DRAM command sequences from the host machine and issues them to the DRAM device.

Both SoftMC and LRT suffer from multiple shortcomings that make them inadequate to satisfy the requirements of a versatile, easy-to-use, and extensible FPGA-based DRAM testing infrastructure. We describe these shortcomings in detail:

Shortcomings of SoftMC: First, SoftMC imposes restrictions on DDR3 data transfers. By replicating an 8-bit data pattern provided by the user, SoftMC initializes DRAM cache blocks (512-bits) with repetitive data. This prevents its users from exploring the large search space of data patterns that can be used in experimental DRAM studies (Section IV-B). Second, due to its DDR3 command trace-based execution model, SoftMC imposes a nondeterministic delay on the order of a few microseconds between DDR3 command traces that are executed back-to-back. This prevents a user from running a class of experiments (e.g., energy measurements) that require precise command timings over a long period of time (i.e., on the order of seconds or minutes, where SoftMC needs to execute multiple DRAM command sequences and induces multiple nondeterministic delays on the order of microseconds). Third, SoftMC is fairly difficult to use and extend. It is only prototyped on the ML605 FPGA board [145], which is no longer manufactured. SoftMC’s development environment contains obsolete and discontinued software and hardware components [42], [146]. Fourth, SoftMC’s DDR3 interface implementation is not decoupled from its remaining hardware via a well-defined interface (i.e., SoftMC is hard-coded to work with DDR3 modules). Thus, implementing new DRAM standards (e.g., DDR4) requires major intrusive modifications to SoftMC’s hardware design.

Shortcomings of LRT: First, LRT does not support timing delays between successive DRAM commands smaller than 10 ns [75]. This greatly limits the DRAM experiment space supported by LRT. For example, many prior works that evaluate DRAM under reduced activation latency [14], [23], [61], [62], [63] and under greatly reduced tRAS-tRP [23], [104] could not conduct their experiments using LRT. Second, LRT can transfer data between the host machine and the FPGA board at only up to 1.9 Mb/s [3]. A slow communication channel between the host machine and the FPGA board can greatly slow down experiments, as DRAM experiments often require a large amount of data to be transferred to the host machine for analysis (Section III-D1). To alleviate the overheads of data transfer and enable practical testing times for RowHammer experiments, LRT implements a specialized hardware module for postprocessing. However, it would be impractical to develop a specialized module for each type of DRAM experiment. Thus, LRT is difficult to use as a general DRAM testing platform.

G. Motivation for DRAM Bender

None of the existing open source DRAM testing infrastructures can be used to easily and comprehensively test state-of-the-art, DDR4 DRAM chips. Therefore, it is essential to develop an open source, versatile, easy-to-use, and extensible DRAM testing infrastructure to enable experimental studies on new DRAM chips. Based on this requirement and our evaluation of the existing DRAM testing infrastructures, we define three key features that a DRAM testing infrastructure should provide.

A. DRAM Bender ISA

The DRAM Bender ISA (Table II) exposes key functionalities that the user needs to write a DRAM test program. For example, these functionalities include issuing DRAM command sequences with control flow (e.g., in a loop) and dynamically modifying the wide (e.g., 512-bits) data word that is written to DRAM with the execution of a WRITE command. There are two types of instructions in the DRAM Bender ISA: 1) RISC-like regular instructions that operate on the registers of the programmable core (e.g., arithmetic instructions and...
is as large as the transfer size of the DRAM interface (e.g., 512 bits for an 8-chip DDR4 module [48]). The wide-data register can only be manipulated using the LOWD instruction, and is used by the WRITE command. The wide-data register allows users to initialize DRAM devices with arbitrary data patterns at the data transfer granularity (e.g., 512 bits in a DDR3/4 module). In contrast, a prior testing infrastructure [34] allows the users to only specify 8-bit data patterns, which are replicated 64 times to form the 512-bit transfer data, significantly limiting the data patterns that can be used in DRAM experiments (e.g., the prior testing infrastructure cannot randomly initialize DRAM chips).

### B. DRAM Bender Hardware Modules

The hardware components of DRAM Bender are: 1) a specialized programmable core that implements the DRAM Bender ISA; 2) a modular memory controller subsystem, and (iii) multiple memory elements that are used to store the DRAM Bender program and data; and 3) buffer the data read back from the DRAM device under test.

**1) Programmable Core:** DRAM Bender comprises a programmable core that implements the DRAM Bender ISA. To enable the programmable core to achieve high frequencies that are required to interface with DRAM devices (under normal operating conditions, the minimum operating frequency for DDR4 devices is 666 MHz [48]), we design it as an inorder core with five pipeline stages: 1) fetch; 2) decode; and 3) three execute stages. The fetch stage fetches a single instruction (72 bits) from the instruction memory. The decode stage decodes the instruction into either an execute micro-operation (µ-op) or into four DRAM µ-ops. The execute stages are separated into two pipelines: 1) execute pipeline, which executes regular µ-ops and 2) DRAM pipeline, which executes four DRAM µ-ops per cycle.

**DRAM µ-ops:** Each DRAM µ-op corresponds to a DRAM command. To execute a DRAM µ-op, DRAM Bender first reads the GPRs to obtain the address operands (e.g., bank address of a PRE command) of the corresponding DRAM command and then issues the command to the DRAM device over the physical interface. The timing parameters of a DRAM command sequence are indirectly specified by either regular SLEEP instructions or DRAM NOP µ-ops.

**Pipeline Dependencies:** Both pipelines read from and write to the register file at the second execute stage to maintain a simple design. The only exception to this is LOAD instructions, which write to the register file at the third execute stage for better design timing closure during implementation. Control-flow instructions (i.e., branches) simply stall the pipeline until they are resolved at the third execute stage. Since the penalty of control-flow instructions is deterministic (six clock cycles), stalling the pipeline does not affect the user-specified timing parameters of the DRAM command sequences in an unpredictable way.

2) **DRAM Interface Adapter:** DRAM Bender implements the DRAM interface adapter, which acts as an intermediary between the DRAM pipeline and the physical DRAM interface. The DRAM pipeline simply outputs a one-hot encoded signal for each DRAM command in a DRAM instruction. For example, a DRAM instruction with an ACT as its first and a WRITE as its third command would set the least-significant bit of the dram_act signal to “logic-1” and the third least-significant bit of the dram_write signal to “logic-1,” while leaving all other signals at “logic-0.” These signals are picked up by the DRAM adapter module.
which essentially translates from this simplified interface (one-hot encoded signals) to a more complex PHY interface. The adapter is decoupled from the rest of the design such that it can be easily modified to integrate new DRAM interfaces.

Developers can easily extend DRAM Bender to support another DRAM interface by: 1) defining new one-hot encoded signals that are set with the execution of new DRAM commands introduced in the new interface and 2) modifying the adapter to match with the new DRAM PHY interface.

3) Memory Elements: DRAM Bender comprises multiple memory elements that store programs, program data, and data read back from DRAM during an experiment. The instruction memory stores the DRAM Bender program and it is initialized by the frontend module, which receives data over the high-speed interface (e.g., PCIe) from a host machine. The data scratchpad stores program data and it can be accessed and manipulated using LD and ST instructions.

4) Periodic Operation Scheduler: To maintain correct operation on the DRAM interface, DRAM devices typically need to receive a set of commands periodically. The periodic operations scheduler (POS) performs these operations either with predefined (periodic READ, ZQ5) or configurable (periodic refresh) periods. POS stores three DRAM Bender programs in its memory to conduct periodic READ, ZQ5 calibration, and refresh operations. It arbitrates between these programs to correctly schedule the required periodic operation to the DRAM device.

5) Readback FIFO: Typically, there is a discrepancy between the bandwidth of the high-speed host-FPGA interface and the DRAM interface on FPGA boards because the DRAM interface can be clocked at higher speeds. To alleviate the negative effects (e.g., pipeline stalls) due to this discrepancy on experiments conducted using DRAM Bender, we buffer the data read back from the DRAM device in a FIFO. The readback FIFO buffers data read from the DRAM device as it is sent to the host machine over the high-speed interface.

Avoiding Readback Stalls: The readback FIFO may fill up during the execution of a DRAM Bender program, in which case the host machine must read the FIFO to allow the upcoming READ commands to execute. DRAM Bender adopts a preemptive mechanism to stall programs at a timing-non-critical point (i.e., while executing regular instructions) to prevent the user-specified timing parameters of DRAM command sequences from being violated. In other words, DRAM Bender never stalls a program while executing a continuous sequence of DRAM commands.

DRAM Bender stalls a program prior to the execution of a continuous DRAM command sequence that contains READ commands (DRAM command sequences without a READ command do not cause readback stalls). The API automatically inserts into DRAM Bender programs hints which specify the number of READ commands that will be executed in the next continuous DRAM command sequence. DRAM Bender lets such command sequences execute only if there is enough space within the readback FIFO, and stalls them otherwise.

C. DRAM Bender Software Modules

DRAM Bender contains two major software modules: 1) the API and 2) program debugger.

1) Application Programmer Interface: DRAM Bender provides an easy-to-use and nonrestrictive API implemented in C++. This API is extensible with a modular design that allows developers to easily integrate new DRAM interfaces to DRAM Bender, and adapt DRAM Bender to new FPGA boards. The API is composed of three components: 1) the program class defines how DRAM Bender programs are created; 2) the platform class exposes the FPGA-based platform to the user via key functions; and 3) the board class implements wrapper functions that communicate with the FPGA board over that FPGA board’s interface (e.g., PCIe drivers). DRAM Bender also implements a Python interface that provides the same functionality as the C++ API using pybind11 [113]. The Python interface allows: 1) agile development and verification of DRAM Bender programs and 2) simple workflow for easy integration of popular data analysis and visualization frameworks (e.g., [134] and [141]). We describe the two components in the API in more detail.

Program: The program class encapsulates functions that are used to create DRAM Bender programs. A DRAM Bender program is internally represented as a list of instructions by the API. The user constructs this list by appending instructions to the list one by one. Users can also append labels to a program which can be used as branch targets of control-flow instructions. This convention allows the user to easily create DRAM Bender programs. To demonstrate the ease of use, Listing 1 depicts a C++ code example that uses the DRAM Bender API to create a DRAM Bender program that reads from a DRAM row. Our open source repository [118] contains other C++ code examples (e.g., for characterizing data retention time) that the users can benefit from or build upon.

The program class defines a set of functions that are based on the template appendInstruction>. Each append function either inserts a regular instruction or a DRAM command into the program. append functions that insert regular instructions have RISC-like argument orderings, e.g., appendLI(R5, 0) (line 5) inserts a load immediate instruction that sets the value of R5 to 0. The arguments of append functions that insert DRAM commands into the program (lines 10, 12, and 14) are ordered as: 1) address register identifier (one or two); 2) address register increment flag (one or two); 3) other flags; and 4) an optional delay (highlighted in red in Listing 1).

Using the optional delay, users can easily create DRAM command sequences with arbitrary timing parameters. The address register increment flag allows for access patterns with arbitrary strides. As an example, the arguments of the appendACT function (line 10) in Listing 1 are: 1) R5, BAR; 2) false, do not increment BAR by bank address stride register; 3) R4, row address register (RAR); 4) false, do not increment RAR by row address stride register; and 5) wait for 11 additional DRAM command bus cycles after issuing the ACT command, in order from left to right. In the appendREAD function, the user reads all DRAM cache

Listing 1. Example DRAM Bender program that reads a row.
blocks in a DRAM row by setting the column address register increment flag, without having to execute ADD instructions between READ commands. This allows the user to schedule DRAM commands with as tight timing delays as possible.

**Platform:** The platform class defines methods to: 1) execute programs on the infrastructure; 2) manipulate the state of the platform (e.g., reset and enable auto-refresh); and 3) retrieve data from the platform over the high-speed interface.

**Board:** The board class defines how data is transferred between the host machine and the FPGA infrastructure over the high-speed interface. It exposes two key functions to the platform class, `sendData` and `receiveData`, which are overloaded by the code that implements the FPGA board-specific interface (e.g., Xilinx XDMA [149]). `sendData` is used by the platform class to send a program to the FPGA, and `receiveData` is used to read data back from the DRAM device under test.

2) Program Debugger: DRAM Bender provides an easy-to-use program debugger. The debugger is used to analyze and debug DRAM Bender programs generated by the API without access to the hardware setup. It uses the Vivado Simulator [147] to run timing-critical simulations of the hardware setup, which includes a DRAM device model. Using the DRAM model, the debugger can inform the user of every timing violation, which enables the user to easily understand if their program runs correctly (i.e., it does not violate DRAM timing parameters unexpectedly). The tight integration of the debugger and the Vivado Simulator allows users to view waveform diagrams and select which hardware signals to debug conveniently, using Vivado’s graphical user interface (GUI).

### D. DRAM Bender Prototypes

We implement DRAM Bender on five different FPGA boards that support different DRAM standards (DDR3 and DDR4) and DRAM modules with different form factors (SODIMM, RDIMM, and UDIMM), to demonstrate the extensibility of DRAM Bender. Different modules can contain DRAM chips built using different packaging technologies (e.g., FBGA 78 and FBGA 96). A DRAM Bender user can find out what packaging technologies are used in chips on a DRAM module by inspecting the relevant module manufacturer specifications. Therefore, a DRAM Bender user can test different DRAM packages. Table III describes the details of DRAM Bender prototypes on different boards. Table IV shows the FPGA resource utilization for each DRAM Bender prototype.

In all prototypes, the instruction memory stores 2048 instructions, the scratchpad stores 1024 words (32-bits), and the read-back FIFO is 32 KiB (i.e., buffers up to 512 DRAM data transfers). DRAM Bender supports a DRAM timing resolution (i.e., the minimum distance between two consecutive DRAM commands) of 1.5 ns in DDR4 boards and a DRAM timing resolution of 2.5 ns in the DDR3 board.5

**Controlled Environment:** To conduct experiments in a controlled environment, our infrastructure incorporates a programmable Maxwell FT200 [83] temperature controller [Fig. 4(a)] connected to silicone rubber heaters attached to both sides of the DRAM module [Fig. 4(b)]. To measure the actual temperature of DRAM chips, we use a thermocouple, which we place between the rubber heaters and the DRAM chips. We connect the heater pads and the thermocouple to the temperature controller. The temperature controller keeps the temperature stable via implementing a closed-loop PID controller.6

1) Hardware IP Cores: We make use of several Xilinx IP components across our prototypes.

**Xilinx XDMA PCIe Engine:** DRAM experiments can result in terabytes of data that is analyzed by the user program that runs on the host machine. Therefore, it is important to provide a high-speed host machine ↔ FPGA board interface. We implement PCIe as the high-speed interface in our prototypes as it has high throughput (several Gb/s [112]) and it is widely available among popular FPGA boards. We use the Xilinx DMA/Bridge Subsystem for PCIe Express IP [149] to communicate with the host machine over the PCIe interface. The IP is responsible for bringing up the PCIe connection with the attached host machine, and driving the physical PCIe signals with the data it receives from DRAM Bender hardware (data read from DRAM using READ commands) over the standard AXI4-Stream interface [4].

---

5We configure our prototypes such that the DRAM interface operates at the minimum standard frequency. This allows for DRAM Bender’s hardware to satisfy the tight FPGA timing constraints in a subset of the FPGA boards (Bittware XUPP35 and Xilinx ZC706) with fairly old and resource-bound FPGAs. When necessary, DRAM Bender can be implemented on other FPGA boards with newer FPGAs to satisfy a finer timing resolution out of the box (e.g., at least as fine as 1.25 ns in Bittware XUPV VH [9]).

6In our experiments, we typically use the temperature range of [30 °C and 85 °C]. Based on the data sheets [83], [117] of the temperature controller and the sensor, we expect the maximum error in temperature measurements to be within ±1.11 °C.
Xilinx DDR3/4 PHY: We issue DRAM commands to the Xilinx DDR3/4 PHY IP [150] over a low-level DFI interface. The IP receives multiple DRAM commands per FPGA clock cycle and performs the required operations to drive the physical DRAM interface signals (e.g., DQ and DQS) at high speeds, such as serializing multiple DRAM commands it receives to a higher frequency domain (e.g., the DDRx interface). By default, the IP imposes a constraint on the ordering of DDRx READ and WRITE commands. We modify the IP’s Verilog source to remove this constraint.

FPGA designs typically use a DFI-like interface with a manufacturer-provided IP to communicate with memory devices [38], [142]. DRAM Bender can be easily adapted to support other memory standards by interfacing with manufacturer-provided IPs built for these memory standards (e.g., HBM2 [148] and RLDRAM [37]). In fact, a recent work [103] adapts DRAM Bender to interface with modern HBM2 chips.

The IPs we use in our prototypes implement standardized interfaces to communicate with the DRAM Bender hardware. DRAM Bender can be relatively easily ported to new prototypes that use different IP (e.g., open source IP), given that they use the same common standard interfaces, e.g., AXI4, PHY interface. In particular, Intel provides a PCIe IP [39] and an external memory interface IP [38] for developers to use in their Intel FPGA designs. These IPs expose hardware interfaces similar to Xilinx’s PCIe and DDR3/4 PHY IPs.

2) DDR3 Changes: To show that DRAM Bender can be easily extended to support other DRAM interfaces and FPGA boards, we summarize the changes required for DRAM Bender to support DDR3 using the Xilinx ZC706 board [143]. First, we modify the DRAM interface adapter (Section III-B2) to correctly translate from DRAM pipeline signals to Xilinx DDR3 PHY [144] signals. Second, the PCIe connection is slower in the ZC706 board (5.0 GT/s) compared to the other boards we use for DDR4 (8.0 GT/s). As a consequence, the AXI interface between the frontend and the Xilinx XDMA IP has a twice as narrow (128-bit) data bus. We make small modifications to the frontend module (Section III-B3) to account for this change in the data bus. Third, we slightly modify the sendData function in the API to correctly partition instructions into packets sent over the PCIe bus. Overall, our changes introduce only additional lines of Verilog code in the hardware description and 30 additional lines of C++ code in the API. Our Github repository includes all ZC706 DDR3 design sources [118].

E. Experiment Workflow

To show how instructions and data flow between DRAM Bender components during an experiment, we describe a typical experiment workflow when using the DRAM Bender infrastructure. Fig. 5 summarizes the workflow.

The user 1) creates a platform object; 2) initializes the DRAM Bender platform using the platform.initialize() function; 3) creates a program object; 4) adds instructions to the program object using append<Instruction> functions; 5) programs DRAM Bender by supplying the program object to the platform.execute() function; and 6) retrieves data from the FPGA board by using the platform.receiveData() function to analyze output data.

IV. Use Cases

To demonstrate that DRAM Bender is a versatile DRAM testing infrastructure, we conduct three new case studies that result in new observations: 1) we investigate the effects of the order in which the two aggressor rows are activated and precharged (i.e., the interleaving pattern of activations to the aggressor rows) on RowHammer [65], [98]; 2) we investigate the effects of data patterns on RowHammer and show that DRAM Bender can be used to uncover new insights into the RowHammer vulnerability; and 3) we demonstrate for the first time that processing-using-memory capability exists in off-the-shelf DDR4 chips. The source code for all three case studies is available on our Github repository [118]. An extended version of this paper [101] presents more detailed results and explanations for all three case studies.

Experimental Setup: In all case studies, we use Alveo U200 [151] boards that are set up as described in Section III-D. We keep the temperature of the DRAM chips that we test at 50°C. We disable all DRAM self-regulation events (e.g., DRAM Refresh [48]) to: 1) minimize all possible sources of interference [64] and 2) disable proprietary in-DRAM RowHammer protection mechanisms (e.g., target row refresh (TRR) [21], [33]). We disable all forms of rank-level error-correction codes (ECCs) to make sure ECC does not obscure RowHammer-induced bit-flips. Table V lists the DRAM modules from three different manufacturers that we use in our experiments. We test one memory module (8 DRAM chips) from each manufacturer.

TABLE V

| Manufacturer | Standard     | Chip Density | Org. |
|--------------|--------------|--------------|------|
| A            | DDR3 RDIMM   | 8Gb          | 1Rx4 |
| B            | DDR4 RDIMM   | 4Gb          | 1Rx8 |
| C            | DDR3 UDIMM   | 6Gb          | 2Rx8 |

A. Study #1: RowHammer: Interleaving Pattern of Activations

To demonstrate DRAM Bender’s ease of use for developing new experiments that uncover new insights, we conduct a comprehensive experiment to study the effects of the interleaving pattern of activations to the aggressor rows on RowHammer bit-flips. Fig. 6 depicts the double-sided RowHammer attack [64], [65], [107], [123] in our case study. We pick five consecutive rows V1, A1, V2, A2, V3, and rapidly activate the aggressor rows (A1 and A2) and record the bit-flips observed in the victim rows (V1, V2, and V3). The interleaving pattern describes how many times one aggressor row is hammered before switching to the hammering of the other aggressor row in one iteration of the attack.
Methodology: We use the checkered data pattern as it is reported to have the highest average RowHammer error coverage across DDR4 modules from three manufacturers in [64]. This data pattern initialises aggressor rows A1 and A2 (see Fig. 6) with “0xAA” values and victim rows V1, V2, and V3 with “0x55” values. We activate aggressors in an interleaved manner. In each iteration, we 1) activate the first aggressor row T times and 2) activate the second aggressor row T times. We sweep the value of T and observe the number of bit-flips in the victim rows. We issue a total number of 1 M ACT commands regardless of the value of T (e.g., the attack finishes in 8 iterations if T = 64 K and in 512 K iterations if T = 1). We create a double-sided RowHammer attack program using the DRAM Bender API. Listing 2 depicts the key code portion that performs an iteration of the attack, which we repeatedly perform until we issue 1 M ACT commands. We sweep T through the range [1, 64 K], incrementing by a power of two after each iteration (i.e., we set T to 1, 2, 4, ..., 64 K) to perform the double-sided RowHammer attack in a sequential (T = 1), and in a cascaded manner (T > 1).

We use two metrics to demonstrate the effects of the interleaving pattern of activations in a RowHammer attack: 1) average bit-flips per row is the number of bit-flips in a DRAM bank averaged across all DRAM rows and 2) HC first is the number of ACT commands issued per aggressor row in a RowHammer attack to induce the first bit-flip in a particular DRAM row.

Results: Fig. 7 shows the number of bit-flips we observe in different modules as we sweep T from 1 to 64 K. Each bar represents the number of bit-flips we observe in one victim row (e.g., gold bar is the sandwiched row, V2) normalized to the number of bit-flips we observe in the same row at T = 64 K, for all rows in a DRAM bank.

We make two new observations from Fig. 7. First, the number of bit-flips in the sandwiched row (V2) significantly increases as T approaches one. Since the total number of ACT commands we issue (i.e., hammer count) remains the same, this observation suggests that the interleaving pattern of the ACT commands to the two aggressor rows in a double-sided RowHammer attack greatly affects the effectiveness of the attack. We observe on average across all double-sided victim (V2) rows (31.9, 9.9, 71.2) bit-flips at T = 64 K, and (314.8, 50.7, 604.9) bit-flips at T = 1 for manufacturers (A, B, C), respectively. The double-sided RowHammer attack more strongly resembles a single-sided attack as T increases. Prior works (e.g., [5] and [123]) show that single-sided attacks are less effective in inducing RowHammer bit-flips than double-sided attacks, and this could explain why our attack’s effectiveness degrades as T increases.

Second, we observe that the number of bit-flips induced in the nearby victim rows, V1 and V3, also increases as T approaches one. We hypothesize that a combination of two phenomena explains this second observation.

1) V1 and V3 become less susceptible to RowHammer-induced bit-flips as their immediate neighbor aggressor row is hammered in a cascaded manner (i.e., T approaches 64 K). A cascaded activation pattern causes the aggressor row to stay precharged for a longer time (one aggressor stays precharged while the other aggressor is being hammered). This might reduce the overall electron injection rate between the aggressor and victim rows [24], [138] and thus the RowHammer vulnerability of the victim row.

2) The RowHammer blast radius effect [65], in which an aggressor row can induce bit-flips in victim rows that are not its immediate neighbors (e.g., A2 and V1), is strengthened as the attack alternates between aggressor rows more frequently (i.e., T approaches one).

Fig. 8 shows the number of ACT commands that is required to observe the first RowHammer bit-flip (HC first) in the sandwiched DRAM row (V2) for each DRAM manufacturer using different values of T, normalized to HC first for T = 1. For each T, we find the row with the smallest HC first in a DRAM bank. We observe an HC first of 99, 80, and 16 K at T = 1, and 130, 108, and 23 K at T = 64 K for manufacturers A, B, and C, respectively. Hence, the interleaving pattern of the double-sided RowHammer attack affects the HC first of a DRAM row. As the attack’s interleaving pattern becomes more cascaded (i.e., T approaches 64 K), HC first increases. RowHammer defense mechanisms (e.g., [51], [65], [74], [108], [122], [129], [154], and [156]) may leverage this observation to render double-sided RowHammer attacks less effective by scheduling the aggressor row accesses in a cascaded manner to the DRAM module.

Listing 2. DRAM Bender program that performs an iteration of the double-sided RowHammer attack.

```python
1. p.appendACT(hammerCount, 0);
2. p.appendLabel("HAMMER");
3. p.appendACT(hammer, false, A), false, tRAb);  
4. p.appendPRE(hammer, false, true, tRAb);
5. p.appendADDI(hammerCount, hammerCount, 1);  
6. p.appendACT(hammer, true, "0x55");  
7. p.appendACT(hammerCount, 0);  
8. p.appendLabel("HOMER");  
9. p.appendACT(hammer, false, A, false, tRAb);  
10. p.appendPRE(hammer, false, true, tRAb);
11. p.appendADDI(hammerCount, hammerCount, 1);  
12. p.appendACT(hammer, true, "0x55");
```
Conclusion: Based on our new observations enabled by DRAM Bender, we conclude that the interleaving pattern of the aggressor row activations in a double-sided RowHammer attack significantly affects: 1) the number of RowHammer-induced bit-flips and 2) the number of ACT commands required to induce the first RowHammer-induced bit-flip in a DRAM bank.

B. Study #2: RowHammer: Data Patterns

To demonstrate that DRAM Bender’s nonrestrictive data interface is useful for generating new insights, we conduct a comprehensive experiment to study the effect of data patterns on RowHammer-induced bit-flips.

Methodology: We run a DRAM Bender test where we perform a double-sided RowHammer attack on a victim row and observe the bit-flips that occur in a DRAM cache block (512 bits) in the victim row. Our test code: 1) initializes the aggressor rows using either a) a SoftMC or b) a DRAM Bender data pattern; 2) conducts double-sided RowHammer on the victim row; and 3) records the positions of the bits that flip in one cache block in the victim row. We run two variations of the test code to observe the bit-flips induced by 1) SoftMC data patterns and 2) DRAM Bender data patterns. First, we use the limited 256 8-bit data patterns supported by SoftMC, used in many prior works to conduct RowHammer experiments [33], [64], [65], [78], [105], [107]. Second, we use the 256 randomly-generated 512-bit DRAM Bender data patterns. We run the test code 100 times and aggregate all bit-flips across the 100 runs of the experiment to collect the complete set of RowHammer failures (e.g., a bit-flip that occurs at least once during the 100 rounds of testing is recorded as a bit-flip). We test 24 randomly selected DRAM cache blocks in 24 DRAM rows from each DRAM module to maintain a reasonable testing time. We repeat each test code variation twice, once by initializing the victim row with an all-zeros and once by initializing the victim row with an all-ones data pattern, to cover cells that flip from 1 → 0 and 0 → 1.

Results: We identify at least one additional cell that flips from either 1 → 0 or 0 → 1 when we initialize the aggressor rows with random 512-bit data patterns compared to when we initialize the aggressor rows with SoftMC data patterns on every victim row that we test in modules from all three DRAM manufacturers. We conclude that DRAM Bender can identify RowHammer-susceptible DRAM cells that cannot be identified using the SoftMC data patterns used in prior work. This information could be useful for strengthening RowHammer attacks that need to induce bit-flips in a particular cell [69].

C. Study #3: In-DRAM Bitwise Operations

Recent works propose processing-using-memory (PIM) techniques that can greatly improve system performance and reduce energy consumption [26], [28], [82], [94], [102], [124], [125], [126]. A recent work demonstrates that bitwise AND/OR operations can be performed in off-the-shelf DDR3 chips [23]. However, it is unclear if commonly used newer-generation off-the-shelf DDR4 chips support these operations. Our goal in this section is to leverage DRAM Bender’s support for fine-grained control over DRAM timing parameters and demonstrate that bitwise AND/OR operations can also be performed in off-the-shelf DDR4 chips. In our experiments, we perform bitwise operations by issuing a sequence of ACT → PRE → ACT commands with violated timing parameters. We characterize our DRAM modules to obtain the bit error rate (BER) (i.e., the fraction of bitlines that cannot correctly perform the bitwise operation) of the performed bitwise operations.

Majority Operation: To perform bitwise AND/OR operations, we use the majority function across DRAM cells on the same bitline [126] in a small portion of a DRAM array. We refer the reader to ComputeDRAM [23] for a detailed description of how real chips can perform majority operations.

Methodology: We perform bitwise AND/OR operations by issuing DRAM command sequences to DRAM modules in quick succession by violating standard DRAM timing parameters. Listing 3 depicts the key code block that we use to generate DRAM Bender programs for our experiment.

We send the first ACT command to row address R1, and the second ACT command to row address R2. We select R1 and R2 such that they correspond to the second and the third row in a contiguous set of four DRAM rows (e.g., row addresses 0, 1, 2, 3), which we call a DRAM segment. For example, the first row in the second DRAM segment in a bank has the row address 4. Thus, R1 and R2 are 5 and 6 when we want to perform bitwise AND/OR in the second segment. In a segment, we initialize the second row with all-ones and place the operands in the first and the third rows to perform an OR operation, and initialize the first row with all-zeros and place the operands in the second and the third rows to perform an AND operation [23]. We test a total number of 8 K segments in one DRAM bank and calculate the BER of the bitwise AND/OR operations for each segment. We sweep the values of tRAS (N in Listing 3) and tRP (M in Listing 3) from 1.5 ns to 15 ns with increments of 1.5 ns, and test a total number of 100 different tRAS and tRP combinations.

Results: We make the key observation from our experiments that we can perform valid majority AND/OR operations (i.e., we observe that multiple rows are activated with the execution of the ACT → PRE → ACT command sequence) on DRAM chips from one DRAM manufacturer (i.e., SK Hynix) when we use the following (tRAS, tRP) timing combinations: (1.5, 1.5), (1.5, 3.0), and (3.0, 1.5) ns.

Fig. 9 depicts the proportion of rows that can perform AND/OR operations with smaller BER than what is displayed on the x-axis. Each bar is composed of three stacks: 1) the blue stack represents the number of segments that can perform AND operations, but not OR operations; 2) the green stack represents the number that can perform OR operations, but not AND operations; and 3) the gold stack represents the number that can perform both AND/OR operations, all proportional to the number of segments that can correctly perform either one or both of the operations. There is heterogeneity in the BER in terms of the operation (AND/OR) being performed. We find that 1) AND operations produce more reliable results than OR operations. Thirty five segments support only AND operations at less than 3% BER and the smallest observed BER is 1.9% for AND operations and 2) the number of segments that support AND/OR operations significantly increases as BER increases: 160 segments support AND/OR with < 5%

Listing 3. DRAM Bender code segment to perform a bitwise majority operation.

1. p. action=ACT [BANK, false, R1, false, 30];
2. p. action=ACT [BANK, false, false, R1];
3. p. action=ACT [BANK, false, false, R2];
D. Research Enabled by DRAM Bender

DRAM Bender has already enabled several prior works on read disturbance characterization [21], [33], [64], [78], [105], [107], [152], evaluation of existing RowHammer mitigations [21], [33], true random number generation [104], uncovering undocumented functionality in DRAM [153], and approximate DRAM for efficient deep neural network inference [68]. Parts of DRAM Bender’s hardware design were reused in developing an end-to-end processing-in-DRAM framework [102] and an HBM2 DRAM testing infrastructure [103].

All of these works have provided significant contributions and advanced the state-of-the-art in their respective areas. Particularly, the works that evaluate RowHammer mitigation mechanisms used in real DDR4 DRAM chips demonstrated that these DRAM chips are not RowHammer-free, contrary to major DRAM manufacturers’ claims [21], [33] and thus enabled significant changes in industry [98].

The artifacts of two of the prior works that use DRAM Bender, U-TRR [33], and QUAC-TRNG [104], are already freely and openly available [119], [120]. The open source artifacts of other prior work [21], [64], [68], [107], [152], [153] that used DRAM Bender are work-in-progress and we plan to release them as part of our future work.

We believe that these prior works demonstrate the versatility and ease of use of DRAM Bender in enabling impactful research. We hope that such versatility and ease of use accelerates the adoption of DRAM Bender by researchers in academia and industry to perform experiments on state-of-the-art DRAM chips.

V. RELATED WORK

DRAM Bender enables experimental studies on state-of-the-art DRAM chips by providing an open, easy-to-use, and extensible DRAM testing infrastructure. We already extensively discuss the shortcomings of existing open source DRAM testing infrastructures in Section II-F. We now compare DRAM Bender against other infrastructures and platforms.

Besides SoftMC [34] and LRT [2], there exist other infrastructures used for testing DRAM chips [1], [11], [22], [35], [36], [54], [80], [99], [133]. Unfortunately, those infrastructures suffer from at least one of the following three shortcomings: 1) support for limited and fixed test patterns [1], [22], [99], [133]; 2) poor usability due to lack of flexibility and open source code [35], [36], [54]; and 3) lack of programmability [11]. DRAM chips include a mechanism called Built-In Self-Test (BIST) (e.g., [7], [41], [114], [115], [155], and [157]) to conduct fixed test patterns. Unfortunately, BIST 1) provides only a limited number of fixed tests that are not programmable; 2) lacks flexibility and visibility to perform many experiments; and 3) is implemented only in a limited set of DRAM chips [34]. In contrast, as we demonstrate, DRAM Bender provides an extensible design and an easy-to-use API. DRAM Bender fully exposes the underlying DRAM interface to the user to allow precisely crafting a desired access pattern. DRAM Bender users can conduct tests using either a supported FPGA board and DRAM standard out of the box or a new FPGA board and a new DRAM standard with relatively simple modifications.

Prior works develop other open source FPGA-based frameworks for: 1) evaluating processing-in-memory (PIM) techniques on real DRAM chips on real systems [102]; 2) facilitating the emulation of PIM architectures and prototyping new PIM techniques [89]; and 3) evaluating HBM-based processing-near-memory (PNM) architectures [158]. Although useful in conducting real system evaluation of PIM and PNM techniques using contemporary workloads (e.g., performance and energy), unlike DRAM Bender, these frameworks cannot be used to study the performance, reliability, and security characteristics of real, state-of-the-art DRAM chips.

VI. CONCLUSION

We develop a new open source DRAM testing infrastructure that is versatile, extensible, and easy to use. Unlike existing open source infrastructures, DRAM Bender provides both 1) a nonrestrictive interface to DRAM devices and 2) an extensible design. Its nonrestrictive API allows users to issue DRAM commands in an arbitrary order and at arbitrary times. It comprises a modular design that enables easily integrating new DRAM interfaces and porting to different FPGA boards. We demonstrate DRAM Bender’s: 1) versatility and ease of use by conducting three new studies that show how DRAM Bender successfully enables new insights into the characteristics of modern DDR4 DRAM chips and 2) extensibility by porting it to five FPGA boards with either DDR3 or DDR4 support. We hope that with its versatility, ease of use, and extensible design, DRAM Bender will be the mainstream infrastructure used in experimental DRAM studies and will be useful in developing new mechanisms and methodologies that improve DRAM security, reliability, performance, and efficiency.

REFERENCES

[1] “V6000 memory platform.” Advantest. Accessed: Jun. 13, 2023. [Online]. Available: https://www.advantest.com/products/ic-test-systems/v6000-memory
[2] “LiteX row hammer tester—GitHub repository.” Antmicro. Accessed: Jun. 13, 2023. [Online]. Available: https://github.com/antmicro/liteX-rowhammer-tester
[3] “Why is EtherBone bridge so slow?” Antmicro. Accessed: Jun. 13, 2023. [Online]. Available: https://github.com/antmicro/liteX-rowhammer-tester/issues/4
[4] AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite

[5] Z. B. Aweke et al., “ANVIL: Software-based protection against next-generation Rowhammer attacks,” in Proc. ASPLOS, 2016, pp. 743–755.

[6] M. M. B. Taller, J. Kerns, B. Ray, T. Morris, and M. T. Rahman, “In-Bitstream DRAM latency variations for generating true random numbers,” in Proc. ICCE, 2019, pp. 1–6.

[7] B. Keeth and R. Baker, “Tramp: An open-source design for enabling experimental DRAM studies,” in Proc. ISCA, 2012, pp. 13–24.

[8] H. Hassan et al., “SoftMC: A flexible and practical open-source infrastructure for enabling experimental DRAM studies,” in Proc. HPCA, 2017, pp. 241–252.

[9] H. Hassan et al., “ChargeCache: Reducing DRAM latency by exploiting row access locality,” in Proc. HPCA, 2016, pp. 581–593.

[10] H. Hassan et al., “SelfMC: A reusable integration framework for FPGA accelerators,” in Proc. TRETIS, 2015, pp. 1–8.

[11] K. Chandrasekar et al. “DRAMPower: Open-source DRAM power & energy estimation tool.” Accessed: Jun. 13, 2023. [Online]. Available: http://www.drampower.info/

[12] K. K. Chang, “Understanding and improving the latency of DRAM-based memory systems,” Ph.D. dissertation, Dep. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2017.

[13] K. K. Chang et al., “Understanding latency variation in modern DRAM chips: Experimental Characterization, analysis, and optimization,” in Proc. SIGMETRICS, 2016, pp. 323–334.

[14] K. K. Chang, “Improving DRAM performance by Parallelizing refreshes with accesses,” in Proc. HPCA, 2014, pp. 356–367.

[15] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-cost inter-linked Subarrays (LISA): Enabling fast inter-Subarray data movement in DRAM,” in Proc. HPCA, 2016, pp. 556–568.

[16] K. K. Chang et al., “Understanding reduced-voltage operation in modern DRAM devices: Experimental characterization, analysis, and mechanisms,” in Proc. SIGMETRICS, 2017, p. 52.

[17] J. Choi et al., “Multiple clone row DRAM: A low latency and area optimized DRAM design,” in Proc. ISCA, 2015, pp. 223–234.

[18] R. H. Dennard, “Field-effect transistor memory,” U.S. Patent 3 387 286, 1968.

[19] DFI 5.0 Specification, DFI Group, Hong Kong, Jul. 2018.

[20] F. Gao, G. Tziantzioulis, and D. Wentzlaff, “ComputeDRAM: In-memory architecture,” in Proc. SIGMETRICS, 2018, pp. 27–40.

[21] C. Hou, J.-F. Li, C.-Y. Lo, D.-M. Kwai, Y.-F. Chou, and C.-W. Wu, “An FPGA-based test platform for Analyzing data retention time distribution of DRAMS,” in Proc. VLSI-DAT, 2013, pp. 1–4.

[22] J. S. Kim et al., “Revisiting RowHammer: An experimental analysis of modern devices and mitigation techniques,” in Proc. ISCA, 2014, pp. 519–532.

[23] J. S. Kim et al., “Detecting and mitigating data-dependent DRAM failures by exploiting current memory content,” in Proc. ICCD, 2018, pp. 27–40.

[24] J. S. Kim et al., “Revisiting RowHammer: An experimental analysis of modern devices and mitigation techniques,” in Proc. ISCA, 2020, pp. 519–532.

[25] S. Khan et al., “An FPGA-based ATE extension module for low-cost multi-GHz memory test,” in Proc. ETS, 2015, pp. 1–6.

[26] S. Khan, C. Wilkerson, D. Lee, and O. Mutlu, “The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study,” in Proc. SIGMETRICS, 2014, pp. 519–532.

[27] S. Khan, D. Lee, and O. Mutlu, “PALBROR: An efficient system-level technique to detect data-dependent failures in DRAM,” in Proc. DSN, 2016, pp. 239–250.

[28] S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, and O. Mutlu, “A case for memory content-based detection and mitigation of data-dependent failures in DRAM,” IEEE Comput. Archit. Lett., vol. 16, no. 2, pp. 88–93, Jul./Dec. 2017.

[29] J. S. Kim et al., “The DRAM latency PUF: Quickly evaluating physical Uncloneable functions by exploiting the latency-reliability tradeoff in modern commodity DRAM devices,” in Proc. HPCA, 2018, pp. 194–207.

[30] J. S. Kim, D. Lee, H. Hassan, and O. Mutlu, “Detecting and mitigating data-dependent DRAM failures by exploiting current memory content,” in Proc. MICRO, 2017, pp. 27–40.

[31] J. S. Kim, D. Lee, H. Hassan, and O. Mutlu, “Solar-DRAM: Reducing DRAM access latency by exploiting the variation in local bitlines,” in Proc. ICCD, 2018, pp. 282–291.

[32] J. S. Kim, D. Lee, and O. Mutlu, “PARBOR: An efficient system-level technique to detect data-dependent failures in DRAM,” in Proc. DSN, 2016, pp. 239–250.
Haocong Luo received the master’s degree from ETH Zurich, Zürich, Switzerland, in 2023, where he is currently pursuing the Ph.D. degree.

His research interests include secure and reliable memory systems, computer architecture, and system evaluation.

Minesh Patel received the Ph.D. degree from ETH Zurich, Zürich, Switzerland, in 2022.

His research interests include architecture, compiler, and runtime support for workloads on cutting-edge systems; system design, analysis, modeling, and evaluation; and building dependable, safe, and secure systems.

O˘guz Ergin received the B.S. degree in electrical and electronics engineering from Middle East Technical University, Ankara, Türkiye, in 2000, and the M.S. and Ph.D. degrees in computer science from the State University of New York at Binghamton, Binghamton, NY, USA, in 2003 and 2005, respectively.

He is a Professor with the Department of Computer Engineering, TOBB University of Economics and Technology, Ankara, Türkiye.

He is working on energy-efficient, reliable, and high-performance computer architectures.

Onur Mutlu (Fellow, IEEE) received the B.S. degree in computer engineering and psychology from the University of Michigan, Ann Arbor, MI, USA, in 2000, and the M.S. and Ph.D. degrees in ECE from the University of Texas at Austin, Austin, TX, USA, in 2002 and 2006, respectively.

He is a Professor of Computer Science with ETH Zurich, Zürich, Switzerland. He is also a Faculty Member with Carnegie Mellon University, Pittsburgh, PA, USA, where he previously held the Strecker Early Career Professorship. His current broader research interests are in computer architecture, systems, hardware security, and bioinformatics. A variety of techniques he, along with his group and collaborators, has invented over the years have influenced industry and have been employed in commercial microprocessors and memory/storage systems. He started the Computer Architecture Group with Microsoft Research, Redmond, WA, USA, from 2006 to 2009, and held various product and research positions with Intel Corporation, Santa Clara, CA, USA, Advanced Micro Devices, Santa Clara, VMware, Palo Alto, CA, USA, and Google, Mountain View, CA, USA. His computer architecture and digital logic design course lectures and materials are freely available on YouTube (https://www.youtube.com/OnurMutluLectures), and his research group makes a wide variety of software and hardware artifacts freely available online (https://safari.ethz.ch/). For more information, please see https://people.inf.ethz.ch/omutlu/.

Dr. Mutlu received the Huawei OlympusMons Award for Storage Systems Research, the Google Security and Privacy Research Award, the Intel Outstanding Researcher Award, the IEEE High Performance Computer Architecture Test of Time Award, the NVMW Persistent Impact Prize, the IEEE Computer Society Edward J. McCluskey Technical Achievement Award, the ACM SIGARCH Maurice Wilkes Award, the Inaugural IEEE Computer Society Young Computer Architect Award, the U.S. National Science Foundation CAREER Award, the Carnegie Mellon University Ladd Research Award, the faculty partnership awards from various companies, and a healthy number of best paper or “Top Pick” paper recognitions at various computer systems, architecture, and security venues. He is an ACM Fellow “for contributions to computer architecture research, especially in memory systems;” an IEEE Fellow for “contributions to computer architecture research and practice;” and an Elected Member of the Academy of Europe.