PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM

Ataberk Olgun[§] Juan Gómez Luna[§] Konstantinos Kanellopoulos[§] Behzad Salami[§]
Hasan Hassan[§] Oğuz Ergin[†] Onur Mutlu[§]

[§]*ETH Zürich* [†]*TOBB University of Economics and Technology*

Commodity DRAM based processing-using-memory (PuM) techniques that are supported by off-the-shelf DRAM chips present an opportunity for alleviating the data movement bottleneck at low cost. However, system integration of these techniques imposes non-trivial challenges that are yet to be solved. Potential solutions to the integration challenges require appropriate tools to develop any necessary hardware and software components. Unfortunately, current proprietary computing systems, specialized DRAM-testing platforms, or system simulators do not provide the flexibility and/or the holistic system view that is necessary to properly evaluate and deal with the integration challenges of commodity DRAM based PuM techniques.

We design and develop PiDRAM, the first flexible end-to-end framework that enables system integration studies and evaluation of real, commodity DRAM based PuM techniques. PiDRAM provides software and hardware components to rapidly integrate PuM techniques across the whole system software and hardware stack. We implement PiDRAM on an FPGA-based RISC-V system. To demonstrate the flexibility and ease of use of PiDRAM, we implement and evaluate two state-of-the-art commodity DRAM based PuM techniques: (i) in-DRAM copy and initialization (RowClone) and (ii) in-DRAM true random number generation (D-RaNGe). We describe how we solve key integration challenges to make such techniques work and be effective on a real-system prototype, including memory allocation, alignment, and coherence. We observe that end-to-end RowClone speeds up bulk copy and initialization operations by 14.6× and 12.6×, respectively over conventional CPU copy, even when coherence is supported with inefficient cache flush operations. Over PiDRAM's extensible codebase, integrating both RowClone and D-RaNGe end-to-end on a real RISC-V system prototype takes only 388 lines of Verilog code and 643 lines of C++ code.

1. Introduction

Main memory is a major performance and energy bottleneck in computing systems [48, 120]. One way of overcoming the main memory bottleneck is to move computation into/near memory, a paradigm known as *processing-in-memory* (PiM) [120]. PiM reduces memory latency between the memory units and the compute units, enables the compute units to exploit the large internal bandwidth within memory devices, and reduces the overall power consumption of the system by eliminating the need for transferring data over power-hungry off-chip interfaces [48, 120].

Recent works propose a variety of PiM techniques to alleviate the data movement problem. One set of techniques propose to place compute logic *near* memory arrays (e.g., processing capability in the memory controller, logic layer of 3D-stacked memory, or near the memory array within the memory chip) [2–4, 12, 20–22, 25, 31, 34, 37, 39, 45–47, 51, 53, 57, 58, 66, 67, 79, 80, 84, 111, 118, 121, 131, 133, 153, 162, 173, 176–178]. These techniques are called *processing-near-memory* (PnM) techniques [120]. Another set of techniques propose to leverage analog properties of memory (e.g., SRAM, DRAM, and NVM) operation to perform computation in different ways (e.g., leveraging non-deterministic behavior in memory array operation to generate random numbers, performing bitwise operations within the memory array by exploiting analog charge sharing properties of DRAM operation) [1, 5–9, 17, 19, 24, 28, 32, 36, 42–44, 54–56, 69, 73, 82, 83, 91–93, 102–104, 114, 134, 137, 145, 148, 152, 156, 160, 163, 168, 171, 172]. These techniques are known as *processing-using-memory* (PuM) techniques [120].

A subset of PuM proposals devise mechanisms that enable computation using DRAM arrays [5, 9, 28, 32, 44, 54, 82, 83, 103, 134, 145, 148, 160, 168]. These mechanisms provide significant performance benefits and energy savings by exploiting the high internal bit-level parallelism of DRAM for (1) bulk data copy and initialization operations at row granularity [1, 28, 134, 148, 160], (2) bitwise operations [6–8, 103, 104, 114, 143–145, 147, 149, 168], (3) arithmetic operations [1, 9, 17, 32, 36, 42, 43, 55, 56, 73, 91–93, 102, 103, 152, 163, 171], and (4) security primitives (e.g., true random number generation [83], physical unclonable functions [82, 126]). Recent works [44, 82, 83] show that some of these PuM mechanisms can already be reliably supported in contemporary, off-the-shelf DRAM chips.[1] Given that DRAM is the dominant main memory technology, these commodity DRAM based PuM techniques[2] provide a promising way to improve the performance and energy efficiency of existing and future systems at *no additional DRAM hardware cost.*

Integration of these PuM mechanisms in a real system imposes non-trivial challenges that require further research to find appropriate solutions. For example, in-DRAM bulk data copy and initialization techniques [28, 145] require modifications to memory management that affect different parts of the system. First, these techniques have specific memory allocation and alignment requirements (e.g., page-granularity source and destination operand arrays should be allocated and aligned in the same DRAM subarray) that are *not* satisfied

[1]We are especially interested in PiM techniques that do *not* require any modification to the DRAM chips or the DRAM interface.

[2]Commodity DRAM based PuM techniques are PuM techniques that can already be supported in existing off-the-shelf DRAM chips without *any* modification to DRAM chips or DRAM interfaces.

by existing memory allocation primitives (e.g., `malloc` [106], `posix_memalign` [108]). Second, in-DRAM copy requires efficient handling of memory coherence, such that the contents of the source operand in DRAM are up-to-date.

None of these system integration challenges of PuM mechanisms can be efficiently studied in existing general-purpose computing systems (e.g., personal computers, cloud computers, embedded systems), special-purpose testing platforms (e.g., SoftMC [60]), or system simulators (e.g., gem5 [18, 132], Ramulator [90, 138], Ramulator-PIM [140], zsim [141], DAMOVSim [125, 139], and other simulators [35, 169, 170, 175]). First, many commodity DRAM based PuM mechanisms in DRAM rely on non-standard DDRx operation, where timing parameters for DDRx commands are violated [44, 82, 83] (or otherwise new DRAM commands are added, which requires new chip designs and interfaces). Existing general-purpose computing systems do *not* permit dynamically changing DDRx timing parameters, which is required to integrate these PuM mechanisms into real systems. Second, prior works show that the reliability of commodity DRAM based PuM mechanisms is highly dependent on environmental conditions such as temperature and voltage fluctuations [82, 83] and process variation. These effects are exacerbated by the non-standard behavior of PuM mechanisms in real DRAM devices. Although special-purpose testing platforms (e.g., SoftMC [60]) can be used to conduct reliability studies, these platforms do *not* model an end-to-end computing system, where system integration of PuM mechanisms can be studied. System simulators (e.g., those aforementioned) can model end-to-end computing systems. However, they (i) do *not* model DRAM operation that violates manufacturer-recommended timing parameters, (ii) do *not* have a way of interfacing with real DRAM chips that embody undisclosed and unique characteristics that have implications on how PuM techniques are integrated into real systems (e.g., proprietary and chip-specific DRAM internal address mapping [30, 89, 130]), and (iii) *cannot* support characterization studies on the reliability of PuM mechanisms since system simulators do not model environmental conditions and process variation.

Our goal is to design and implement a flexible real-system platform that can be used to solve system integration challenges and analyze trade-offs of end-to-end implementations of commodity DRAM based PuM mechanisms. To this end, we develop ***P**rocessing-**in**-**DRAM*** (PiDRAM) framework, the first flexible, end-to-end, and open source framework that enables system integration studies and evaluation of real PuM techniques using real unmodified DRAM devices.

PiDRAM facilitates system integration studies of new commodity DRAM based PuM mechanisms by providing four customizable hardware and software components that can be used as a common basis to enable system support for such mechanisms in real systems. PiDRAM contains two main *hardware* components. First, a custom, easy-to-extend *memory controller* allows for implementing new DRAM command sequences that perform PuM operations. For example, the memory controller can be extended with a single state machine in its hardware description to implement a new DDRx command sequence with user-defined timing parameters to implement a new PuM technique (i.e., perform a new PuM operation). Second, an *ISA-transparent controller (**P**u**M** **O**perations **C**ontroller, POC)* supervises PuM execution. POC exposes the PuM operations to the software components of PiDRAM over a memory-mapped interface to the processor, allowing the programmer to perform PuM operations using the PiDRAM framework by executing conventional LOAD/STORE instructions. The memory-mapped interface allows PiDRAM to be easily ported to systems that implement different instruction set architectures. PiDRAM contains two main *software* components. First, an *extensible library* allows system designers to implement software support for PuM mechanisms. This library contains customizable functions that communicate with POC to perform PuM operations. Second, a custom *supervisor software* contains the necessary OS primitives (e.g., memory management) to enable end-to-end implementations of commodity DRAM based PuM techniques.

We demonstrate a prototype of PiDRAM on an FPGA-based RISC-V system [11]. To demonstrate the flexibility and ease of use of PiDRAM, we implement two prominent PuM techniques: (1) *RowClone* [148], an in-DRAM data copy and initialization technique, and (2) *D-RaNGe* [83], an in-DRAM true random number generation technique based on activation-latency failures. In order to support RowClone (Section 5), (i) we customize the PiDRAM memory controller to issue carefully-engineered sequences of DRAM commands that perform data copy (and initialization) operations in DRAM, and (ii) we extend the custom supervisor software to implement a new memory management mechanism that satisfies the memory allocation and alignment requirements of RowClone. For D-RaNGe (Section 6), we extend (i) the PiDRAM memory controller with a new state machine that periodically performs DRAM accesses with reduced activation latencies to generate random numbers [83] and a new hardware *random number buffer* that stores the generated random numbers, and (ii) the custom supervisor software with a function that retrieves the random numbers from the hardware buffer to the user program. Our end-to-end evaluation of (i) RowClone demonstrates up to 14.6× speedup for bulk copy and 12.6× initialization operations over CPU copy (i.e., conventional `memcpy`), even when coherence is satisfied using inefficient cache flush operations, and (ii) D-RaNGe demonstrates that an end-to-end integration of D-RaNGe can provide true random numbers at high throughput (8.30 Mb/s) and low latency (4-bit random number in 220 ns), even without any hardware or software optimizations. Implementing both PuM techniques over the Verilog and C++ codebase provided by PiDRAM requires only 388 lines of Verilog code and 643 lines of C++ code.

Our contributions are as follows:

- We develop PiDRAM, the first flexible framework that enables end-to-end integration and evaluation of PuM mechanisms using real unmodified DRAM chips.
- We develop a prototype of PiDRAM on an FPGA-based platform. To demonstrate the ease-of-use and evaluation benefits

of PiDRAM, we implement two state-of-the-art DRAM-based PuM mechanisms, RowClone and D-RaNGe, and evaluate them on PiDRAM's prototype using unmodified DDR3 chips.

- We devise a new memory management mechanism that satisfies the memory allocation and alignment requirements of RowClone. We demonstrate that our mechanism enables RowClone end-to-end in the full system, and provides significant performance improvements over traditional CPU-based copy and initialization operations (`memcpy` [107] and `calloc` [105]) as demonstrated on our PiDRAM prototype.
- We implement and evaluate a state-of-the-art DRAM-based true random number generation technique (D-RaNGe). Our implementation provides a solid foundation for future work on system integration of DRAM-based PuM security primitives (e.g., PUFs [13, 82], TRNGs [13, 123, 124]), implemented using real unmodified DRAM chips.

2. Background

We provide the relevant background on DRAM organization, DRAM operation and commodity DRAM based PuM techniques. We refer the reader to prior works for more comprehensive background about DRAM organization and operation [26, 29, 49, 50, 87, 89, 95, 99, 100, 113, 123, 128].

2.1. DRAM Background

DRAM-based main memory is organized hierarchically. Fig. 1 (top) depicts this organization. A processor is connected to one or more memory channels (DDRx in the figure) ❶. Each channel has its own command, address, and data buses. Multiple memory modules can be plugged into a single channel. Each module contains several DRAM chips ❷. Each chip contains multiple DRAM banks that can be accessed independently ❸. Data transfers between DRAM memory modules and processors occur at cache block granularity. The cache block size is typically 64 bytes in current systems.

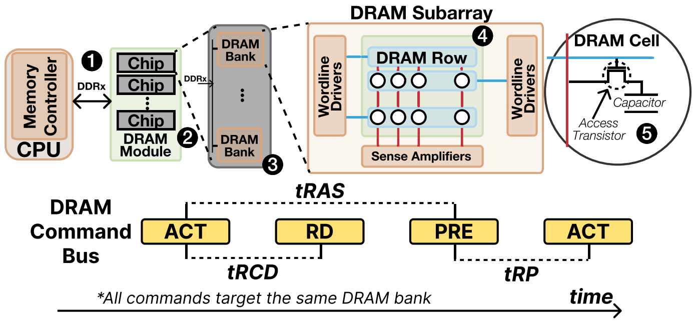

Figure 1: DRAM organization (top). Timing diagram of DRAM commands (bottom).

Inside a DRAM bank, DRAM cells are laid out as a two dimensional array of wordlines (i.e., DRAM rows) and bitlines (i.e., DRAM columns) ❹. Wordlines are depicted in blue and bitlines are depicted in red in Fig. 1. Wordline drivers drive the wordlines and sense amplifers read the values on the bitlines. A DRAM cell is connected to a bitline via an access transistor ❺. When enabled, an access transistor allows charge to flow between a DRAM cell and the cell's bitline.

DRAM Operation. When all DRAM rows in a bank are closed, DRAM bitlines are precharged to a reference voltage level of $\frac{V_{DD}}{2}$. The memory controller sends an activate (ACT) command to the DRAM module to drive a DRAM wordline (i.e., enable a DRAM row). Enabling a DRAM row starts the charge sharing process. Each DRAM cell connected to the DRAM row starts sharing its charge with its bitline. This causes the bitline voltage to deviate from $\frac{V_{DD}}{2}$ (i.e., the charge in the cell perturbs the bitline voltage). The sense amplifier senses the deviation in the bitline and amplifies the voltage of the bitline either to V_{DD} or to 0. As such, an ACT command copies one DRAM row to the sense amplifiers (i.e., row buffer). The memory controller can send READ/WRITE commands to transfer data from/to the sense amplifier array. Once the memory controller needs to access another DRAM row, the memory controller can close the enabled DRAM row by sending a precharge (PRE) command on the command bus. The PRE command first disconnects DRAM cells from their bitlines by disabling the enabled wordline and then precharges the bitlines to $\frac{V_{DD}}{2}$.

DRAM Timing Parameters. DRAM datasheets specify a set of timing parameters that define the minimum time window between valid combinations of DRAM commands [26, 27, 81, 98]. The memory controller must wait for tRCD, tRAS, and tRP nanoseconds between successive ACT $\rightarrow$ RD, ACT $\rightarrow$ ACT, and PRE $\rightarrow$ ACT commands, respectively (Figure 1, bottom). Prior works show that these timing parameters can be violated (e.g., successive ACT $\rightarrow$ RD commands may be issued with a shorter time window than tRCD) to improve DRAM access latency [26, 27, 81, 96, 98], implement physical unclonable functions [13, 82, 126], generate true random numbers [83, 123, 124], copy data [44, 148], and perform bitwise AND/OR operations [44, 143–145, 149] in commodity DRAM devices.

DRAM Internal Address Mapping. DRAM manufacturers use DRAM-internal address mapping schemes [30, 89, 130] to translate from logical (e.g., row, bank, column) DRAM addresses that are used by the memory controller to physical DRAM addresses that are internal to the DRAM chip (e.g., the physical position of a DRAM row within the chip). These schemes allow (i) post-manufacturing row repair techniques to map erroneous DRAM rows to redundant DRAM rows and (ii) DRAM manufacturers to organize DRAM internals in a cost-efficient and reliable way [76, 159]. DRAM-internal address mapping schemes can be substantially different across different DRAM chips [15, 30, 63, 70, 75–77, 88, 96, 110, 127, 129, 130, 142]. Thus, consecutive logical DRAM row addresses might not point to physical DRAM rows in the same subarray.

2.2. PuM Techniques

Prior work proposes a variety of in-DRAM computation mechanisms (i.e., PuM techniques) that (i) have great potential to improve system performance and energy efficiency [9, 28, 40, 54, 144, 145, 147–151] or (ii) can provide low-cost security primitives [13, 14, 82, 83, 124, 126]. A subset of these in-DRAM computation mechanisms are demonstrated on real DRAM chips [13, 44, 82, 83, 124, 126]. We describe the major relevant prior works briefly:

RowClone [148] is a low-cost DRAM architecture that can perform bulk data movement operations (e.g., copy, initialization) inside DRAM chips at high performance and low energy.
Ambit [144, 145, 147, 150, 151] is a new DRAM substrate that can perform (i) bitwise majority (and thus bitwise AND/OR) operations across three DRAM rows by simultaneously activating three DRAM rows and (ii) bitwise NOT operations on a DRAM row using 2-transistor 1-capacitor DRAM cells [72, 112].
ComputeDRAM [44] demonstrates in-DRAM copy (previously proposed by RowClone [148]) and bitwise AND/OR operations (previously proposed by Ambit [145]) on real DDR3 chips. ComputeDRAM performs in-DRAM operations by issuing carefully-engineered, valid sequences of DRAM commands with violated tRAS and tRP timing parameters (i.e., by not obeying manufacturer-recommended timing parameters defined in DRAM chip specifications [116]). By issuing command sequences with violated timing parameters, ComputeDRAM activates two or three DRAM rows in a DRAM bank in quick succession (i.e., performs two or three row activations). ComputeDRAM leverages (i) two row activations to transfer data between two DRAM rows and (ii) three row activations to perform the majority function in real unmodified DRAM chips.
D-RaNGe [83] is a state-of-the-art high-throughput DRAM-based true random number generation technique. D-RaNGe leverages the randomness in DRAM activation (tRCD) failures as its entropy source. D-RaNGe extracts random bits from DRAM cells that fail with 50% probability when accessed with a reduced (i.e., violated) tRCD. D-RaNGe demonstrates high-quality true random number generation on a vast number of real DRAM chips across multiple generations.
QUAC-TRNG [124] demonstrates that four DRAM rows can be activated in a quick succession using an ACT-PRE-ACT command sequence (called QUAC) with violated tRAS and tRP timing parameters in real DDR4 DRAM chips. QUAC-TRNG uses QUAC to generate true random numbers at high throughput and low latency.

3. Motivation

Implementing DRAM-based PuM techniques and integrating them into a real system requires modifications across the hardware and software stack. End-to-end implementations of PuM techniques require proper tools that (i) are flexible, to enable rapid development of PuM techniques and (ii) support real DRAM devices, to correctly observe the effects of reduced DRAM timing operations that are fundamental to enabling commodity DRAM based PuM in real unmodified DRAM devices. Existing general-purpose computers, specialized DRAM testing platforms (e.g., those aforementioned, Section 1) cannot be used to study end-to-end implementations of commodity DRAM based PuM techniques.

First, implementing new DDRx command sequences that perform PuM operations requires modifications to the memory controller. Existing general purpose computers do not support customizations to the memory controller to dynamically modify manufacturer-recommended DRAM timing parameters [27, 60, 81, 83, 98]. This hinders the possibility of studying end-to-end implementations of PuM techniques on such platforms. Second, PuM techniques impose data mapping and allocation requirements (Section 5) that are not satisfied by current memory management and allocation mechanisms (e.g., malloc [106]). Current OS memory management schemes must be augmented to satisfy these requirements. Existing specialized DRAM testing platforms (e.g., SoftMC [60]) do not have system support to enable this. By design, these platforms are not built for system integration. Hence, it is difficult to evaluate system-level mechanisms that enable PuM techniques on DRAM testing platforms. Third, system simulators (i) do *not* model DRAM operation that violates manufacturer-recommended timing parameters, (ii) do *not* have a way of interfacing with real DRAM chips that embody undisclosed and unique characteristics that have implications on how PuM techniques are integrated into real systems (e.g., proprietary and chip-specific DRAM internal address mapping [30, 89, 130]) that influence PuM operations, and (iii) *cannot* support studies on the reliability of PuM techniques since system simulators do *not* model environmental conditions and process variation. We summarize the limitations of the relevant experimental platforms later in Table 4.

Our **goal** is to develop a flexible end-to-end framework that enables rapid system integration of commodity DRAM based PuM techniques and facilitates studies on end-to-end full-system implementations of PuM techniques using real DRAM devices. To this end, we develop PiDRAM.

4. PiDRAM

Implementing commodity DRAM based PuM techniques end-to-end requires developing new hardware (HW) and software (SW) components or augmenting existing components with new functionality (e.g., memory allocation for RowClone requires a new memory allocation routine in the OS, Section 5.1). To ease the process of modifying various components across the hardware and software stack to implement new PuM techniques, PiDRAM provides key HW and SW components. Figure 2 presents an overview of the HW and SW components of the PiDRAM framework. Later in Section 4.3, we describe the general workflow for executing a PuM operation on PiDRAM.

4.1. Hardware Components

PiDRAM comprises two key hardware components. Both of these components are designed with the goal to provide a flexible and easy to use framework for evaluating PuM techniques.

❶ **PuM Operations Controller (POC).** POC decodes and executes PiDRAM instructions (e.g., RowClone-Copy [148] that are used by the programmer to perform PuM operations. POC communicates with the rest of the system over two well-defined interfaces. First, it communicates with the CPU over a memory-mapped interface, where the CPU can send data to or receive data from POC using memory store and load instructions. The CPU accesses the memory-mapped registers (*instruction*, *data*, and *flag* registers) in POC to execute in-DRAM operations. This improves the portability of the

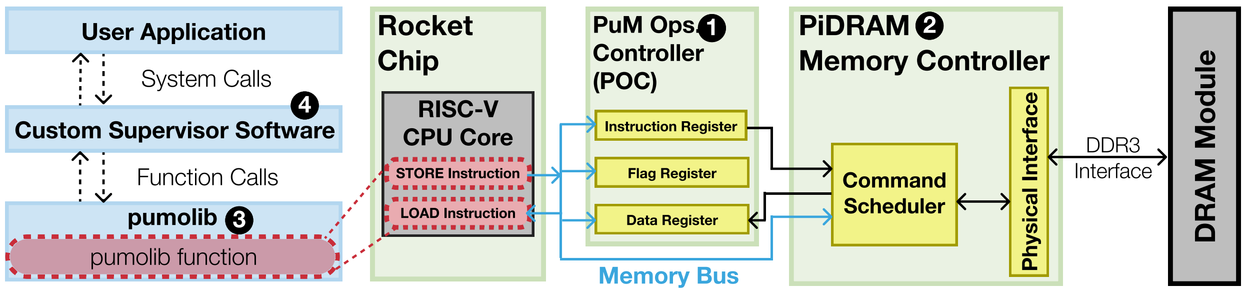

Figure 2: PiDRAM overview. Modified hardware (in green) and software (in blue) components. Unmodified components are in gray. A pumolib function executes load and store instructions in the CPU to perform PuM operations (in red). We use yellow to highlight the key hardware structures that are controlled by the user to perform PuM operations.

framework and facilitates porting the framework to systems that employ different instruction set architectures. Second, POC communicates with the memory controller to perform PuM operations in the DRAM chip over a simple hardware interface. To do so, POC (i) requests the memory controller to perform a PuM operation, (ii) waits until the memory controller performs the operation, and (iii) receives the result of the PuM operation from the memory controller. The CPU can read the result of the operation by executing load instructions that target the *data* register in POC.

❷ **Custom Memory Controller.** PiDRAM's memory controller provides an easy-to-extend basis for commodity DRAM based PuM techniques that require issuing DRAM commands with violated timing parameters [13, 44, 82, 83, 124]. The memory controller is designed modularly and requires easy-to-make modifications to its scheduler to implement new PuM techniques. For instance, our modular design enables supporting RowClone operations (Section 5) in just 60 lines of Verilog code on top of the baseline custom memory controller's scheduler that implements conventional DRAM operations (e.g., read, write).

The custom memory controller employs three key submodules to facilitate the implementation of new PuM techniques. (i) The *Periodic Operations Module* periodically issues DDR3 refresh [117] and interface maintenance commands [52]. (ii) A simple *DDR3 Command Scheduler* supports conventional DRAM operations (e.g., activate, precharge, read, and write). This scheduler applies an open-bank policy (i.e., DRAM banks are left open following a DRAM row activation) to exploit temporal locality in memory accesses to the DRAM module. LOAD/STORE memory requests are simply handled by the command scheduler in a latency-optimized way. Thus, new modules that are implemented to provide new PuM functionality (e.g., a state machine that controls the execution of a new PuM operation) in the custom memory controller do not compromise the performance of LOAD/STORE memory requests. (iii) The *Configuration Register File* (CRF) comprises 16 user-programmable registers that store the violated timing parameters used for DDRx sequences that trigger PuM operations (e.g., activation latency used in generating true random numbers using D-RaNGe [83], see Section 6) and miscellaneous parameters for PuM implementations (e.g., true random number generation period for D-RaNGe, see Section 6). In our implementation, CRF stores only the timing parameters used for performing PuM operations (e.g., RowClone and D-RaNGe). We do not store every standard DDRx timing parameter (i.e., non-violated, which are used exactly as defined as in DRAM chip specifications) in the CRF. Instead these timings are embedded in the command scheduler.

4.2. Software Components

PiDRAM comprises two key software components that complement and control PiDRAM's hardware components to provide a flexible and easy to use end-to-end PuM framework.

❸ **PuM Operations Library (pumolib).** The extensible library (*PuM o*perations *lib*rary) allows system designers to implement software support for PuM techniques. Pumolib contains customizable functions that interface with POC to perform PuM operations in real unmodified DRAM chips. The customizable functions hide the hardware implementation details of PuM techniques implemented in PiDRAM from software developers (that use pimolib). For example, although we expose PuM techniques to software via memory LOAD-/STORE operations (POC is exposed as a memory-mapped module, Section 4.1), PuM techniques can also be exposed via specialized instructions provided by ISA extensions. Pumolib hides such implementation details from the user of the library and contributes to the modular design of the framework.

We implement a general protocol that defines how programmers express the information required to execute PuM op-

Table 1: Pumolib functions

Function	Arguments	Description
set_timings	RowClone_T1, RowClone_T2, tRCD	Updates CRF registers with the timing parameters used in RowClone (*T1* and *T2*) and D-RaNGe (*tRCD*) operations.
rng_configure	period, address, bit_offsets	Updates CRF registers, configuring the random number generator to to access the DRAM cache block at *address* every *period* cycles and collect the bits at *bit_offsets* from the cache block.
copy_row	source_address, destination_address	Performs a RowClone-Copy operation in DRAM from the *source_address* to the *destination_address*.
activation_failure	address	Induces an activation failure in a DRAM location pointed by the *address*.
buf_size	-	Returns the number of random words in the random number buffer.
rand_dram	-	Returns 32 bits (i.e., random words) from the random number buffer.

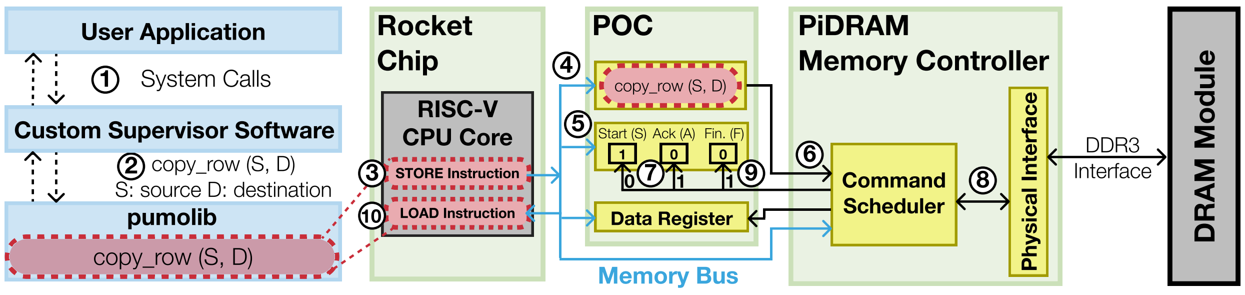

Figure 3: Workflow for a PiDRAM RowClone-Copy operation

erations to the PuM operations controller (POC). A typical function in pumolib performs a PuM operation in four steps: It (i) writes a PiDRAM instruction to the POC's *instruction* register, (ii) sets the *Start* flag in POC's *flag* register, (iii) waits for the POC to set the *Ack* flag in POC's *flag* register, and (iv) reads the result of the PuM operation from POC's *data* register (e.g., the true random number after performing an in-DRAM true random number generation operation, Section 6). We list the currently implemented pumolib functions in Table 1.

❹ **Custom Supervisor Software.** PiDRAM provides a custom supervisor software that implements the necessary OS primitives (i.e., virtual memory management, memory allocation and alignment) for end-to-end implementation of PuM techniques. This facilitates developing end-to-end integration of PuM techniques in the system as these techniques require modifications across the software stack. For example, integrating RowClone end-to-end in the full system requires a new memory allocation mechanism (Section 5.1) that can satisfy the memory allocation constraints of RowClone [148]. Thus, we implement the necessary functions and data structures in the custom supervisor software to implement an allocation mechanism that satisfies RowClone's constraints. This allows PiDRAM to be extended easily to implement support for new PuM techniques that share similar memory allocation constraints (e.g., Ambit [145], SIMDRAM [54], and QUAC-TRNG [124], as shown in Table 2).

4.3. Execution of a PuM Operation

We describe the general workflow for a PiDRAM operation (e.g., RowClone-Copy [148], random number generation using D-RaNGe [83]) in Figure 3 over an example `copy_row()` function that is called by the user to perform a RowClone-Copy operation in DRAM.

The user makes a system call to the custom supervisor software ① that in turn calls the `copy_row(source, destination)` function in the pumolib ②. The function executes two store instructions in the RISC-V core ③. The first store instruction updates the *instruction* register with the copy_row instruction (i.e., the instruction that performs a RowClone-Copy operation in DRAM) ④ and the second store instruction sets the Start flag in the flag register to logic-1 ⑤ in POC. When the Start flag is set, POC instructs the PiDRAM memory controller to perform a RowClone-Copy operation using violated timing parameters ⑥. The POC waits until the memory controller starts executing the operation, after which it sets the Start flag to logic-0 and the Ack flag to logic-1 ⑦, indicating that it started the execution of the PuM operation. The PiDRAM memory controller performs the RowClone-Copy operation by issuing a set of DRAM commands with violated timing parameters ⑧. When the last DRAM command is issued, the memory controller sets the Finish flag (denoted as Fin. in Figure 3) in the flag register to logic-1 ⑨, indicating the end of execution for the last PuM operation that the memory controller acknowledged. The copy function periodically checks either the Ack or the Finish flag in the flag register (depending on a user-supplied argument) by executing load instructions that target the flag register ⑩. When the periodically checked flag is set, the copy function returns. This way, the copy function optionally blocks until the start (i.e., the Ack flag is set) or the end (i.e., the Finish flag is set) of the execution of the PuM operation (in this example, RowClone-Copy).[3]

4.4. Use Cases

PiDRAM is primarily designed to study end-to-end implementations of commodity DRAM based PuM techniques [13, 44, 82, 83, 123] on real systems. Beyond commodity DRAM based PuM techniques, many prior works propose minor modifications to DRAM arrays to enable various arithmetic [9, 32, 40, 54] and bitwise operations [9, 145, 147, 149, 151] and security primitives [126]. These PuM techniques share common memory allocation and coherence requirements (Section 5.1) that must be satisfied to enable their end-to-end integration into a real system. PiDRAM facilitates the implementation of PuM techniques and enables rapid exploration of such integration challenges on a real DRAM-based system. Table 2 describes some of the PuM case studies PiDRAM can enable.

Other than providing an easy-to-use basis for end-to-end implementations of commodity DRAM based PuM techniques, PiDRAM can be easily extended with a programmable microprocessor placed near the memory controller to study system integration challenges of Processing-near-Memory (PnM) techniques (e.g., efficient pointer chasing [57, 58, 65], general-purpose compute [158], machine learning [74, 86, 94, 101, 122], databases [21, 22, 97], graph processing [16]).

[3]The data register is not used in a RowClone-Copy [148] operation because the result of the RowClone-Copy operation is stored *in memory* (i.e., the source memory row is copied to the destination memory row). The data register is used in a D-RaNGe [83] operation, as described in Section 6. When used, the command scheduler stores the random numbers generated by the D-RaNGe operation in the data register. To read the generated random number, we implement a pumolib function called `rand_dram()` that executes load instructions in the RISC-V core to retrieve the random number from the data register in POC.

4.5. PiDRAM's HW & SW Components: Summary

We identify and build two hardware components (PuM Operations Controller and Custom Memory Controller) and two software components (PuM Operations Library, Custom Supervisor Software) as key components that are commonly required by end-to-end PuM implementaions. We reuse these key components to implement two different PuM mechanisms (RowClone in Section 5 and D-RaNGe in Section 6) in PiDRAM. The key components can be reused in the same way to implement other PuM mechanisms (e.g., the ones in Table 2). However, reusing a component does not mean that the component can simply be instantiated in a system and the system will be able to perform PuM operations immediately.

We acknowledge that these components require modifications to implement new PuM techniques in PiDRAM and possibly to integrate PiDRAM into other systems. In fact, we quantify the degree of these modifications in our RowClone and D-RaNGe case studies. We show that the key components form a useful and easy-to-extend basis for PuM techniques with our Verilog and C code complexity analyses for both use cases (Sections 5.5.1 and 6.2).

4.6. PiDRAM Prototype

We develop a prototype of the PiDRAM framework on an FPGA-based platform. We use the Xilinx ZC706 FPGA board [167] to interface with real DDR3 modules. Xilinx provides a DDR3 PHY IP [164] that exposes a low-level "DFI" interface [33] to the DDR3 module on the board. We use this interface to issue DRAM commmands to the DDR3 module. We use the existing RISC-V based SoC generator, Rocket Chip [11], to generate the RISC-V hardware system. Our custom supervisor software extends the RISC-V Proxy Kernel [136] to support the necessary OS primitives on PiDRAM's prototype. Figure 4 shows our prototype.

Simulation Infrastructure. To aid the users in testing the correctness of any modifications they make on top of PiDRAM, we provide the developers with a Verilog simulation environment that injects regular READ/WRITE commands and custom commands (e.g., update the Configurable Register File (CRF), perform RowClone-Copy, generate random numbers) to the memory controller. When used in conjunction with the Micron

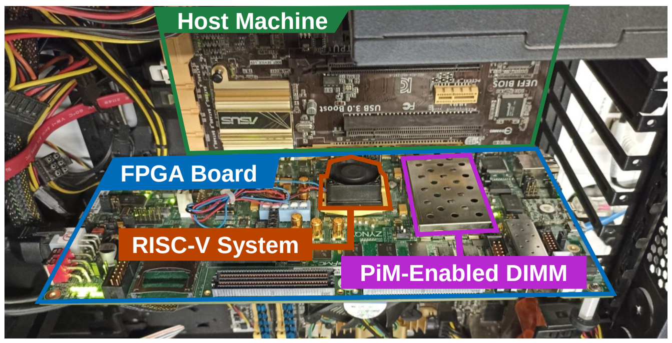

Figure 4: PiDRAM's FPGA prototype

DDR3 Verilog model provided by Xilinx [164], the simulation environment can help the developers to easily understand if something unexpected is happening in their implementation (e.g., if timing parameters are violated).

Open Source Repository. We make PiDRAM freely available to the research community as open source software at `https://github.com/CMU-SAFARI/PiDRAM`. Our repository includes the full PiDRAM prototype that has RowClone (Section 5) and D-RaNGe (Section 6) implemented end-to-end on the RISC-V system.

5. Case Study #1: End-to-end RowClone

We implement support for ComputeDRAM-based (i.e., using carefully-engineered sequences of valid DRAM commands with violated timing parameters) RowClone (in-DRAM copy-/initialization) operations on PiDRAM to conduct a detailed study of the challenges associated with implementing RowClone end-to-end on a real system. None of the relevant prior works [44, 54, 143, 145, 148, 149, 151, 160] provide a clear description or a real system demonstration of a working memory allocation mechanism that can be implemented in a real operating system to expose RowClone capability to the programmer.

5.1. Implementation Challenges

Data Mapping. RowClone has data mapping and alignment requirements that cannot be satisfied by current memory allocation mechanisms (e.g., malloc [106]). We identify four major issues that complicate the process of implementing support

Table 2: Various known PuM techniques that can be studied using PiDRAM. PuM techniques we implement in this work are highlighted in bold.

PuM Technique	Description	Integration Challenges
ComputeDRAM-based [44] RowClone [148]	Bulk data-copy and initialization within DRAM	(i) *memory allocation and alignment mechanisms* that map source & destination operands of a copy operation into same DRAM subarray; (ii) *memory coherence*, i.e., source operand must be up-to-date in DRAM.
D-RaNGe [83]	True random number generation using DRAM	(i) periodic generation of true random numbers; (ii) *memory scheduling policies* that minimize the interference caused by random number requests.
ComputeDRAM-based [44] Ambit [145]	Bitwise operations in DRAM	(i) *memory allocation and alignment mechanisms* that map operands of a bitwise operation into same DRAM subarray; (ii) *memory coherence*, i.e., operands of the bitwise operations must be up-to-date in DRAM.
SIMDRAM [54]	Arithmetic operations in DRAM	(i) *memory allocation and alignment mechanisms* that map operands of an arithmetic operation into same DRAM subarray; (ii) *memory coherence*, i.e., operands of the arithmetic operations must be up-to-date in DRAM; (iii) *bit transposition*, i.e., operand bits must be laid out vertically in a single DRAM bitline.
DL-PUF [82]	Physical unclonable functions in DRAM	*memory scheduling policies* that minimize the interference caused by generating PUF responses.
QUAC-TRNG [123] and Talukder+ [13]	True random number generation using DRAM	(i) periodic generation of true random numbers; (ii) *memory scheduling policies* that minimize the interference caused by random number requests; (iii) efficient integration of the SHA-256 cryptographic hash function.

for RowClone in real systems. First, the source and destination operands (i.e., page (4 KiB)-sized arrays) of the copy operation must reside in the same DRAM subarray. We refer to this as the *mapping* problem. Second, the source and destination operands must be aligned to DRAM rows. We refer to this as the *alignment* problem. Third, the size of the copied data must be a multiple of the DRAM row size. The size constraint defines the granularity at which we can perform bulk-copy operations using RowClone. We refer to this as the *granularity* problem. Fourth, RowClone must operate on up-to-date data that resides in main memory. Modern systems employ caches to exploit locality in memory accesses and reduce memory latency. Thus, cache blocks (typically 64 B) of either the source or the destination operands of the RowClone operation may have cache block copies present in the cache hierarchy. Before performing RowClone, the cached copies of pieces of both source and destination operands must be invalidated and written back to main memory. We refer to this as the *memory coherence* problem.

We explain the data mapping and alignment requirements of RowClone using Figure 5. The figure depicts a simplified version of a DRAM chip with two banks and two subarrays. The operand Source 1 cannot be copied to the operand Target 1 as the operands do not satisfy the *granularity* constraint (❶). Performing such a copy operation would overwrite the remaining (i.e., non-Target 1) data in Target 1's DRAM row with the remaining (i.e., non-Source 1) data in Source 1's DRAM row. Source 2 cannot be copied to Target 2 as Target 2 is not *aligned* to its DRAM row (❷). Source 3 cannot be copied to Target 3, as these operands are not *mapped* to the same DRAM subarray (❸). In contrast, Source 4 can be copied to Target 4 using in-DRAM copy because these operands are (i) *mapped* to the same DRAM subarray, (ii) aligned to their DRAM rows and (iii) occupy their rows completely (i.e., the operands have sizes equal to DRAM row size) (❹).

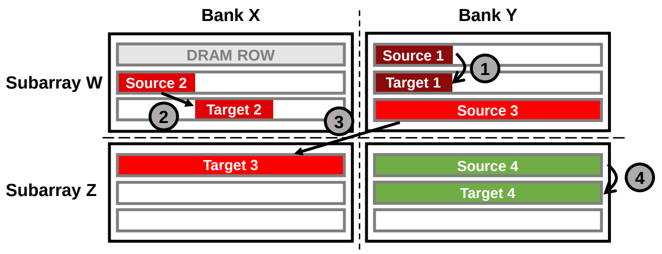

Figure 5: A DRAM chip with two banks and two subarrays. Only one operation (i.e., operation ❹) can succeed as its operands satisfy all of *mapping*, *alignment* and *granularity* constraints.

5.2. Memory Allocation Mechanism

Computing systems employ various layers of address mappings that obfuscate the DRAM row-bank-column address mapping from the programmer [30, 61], which makes allocating source and target operands as depicted in Figure 5-(❹) difficult. DRAM manufacturers employ DRAM-internal address mapping schemes (Section 2.1) that translate from logical (e.g., memory-controller-visible DRAM row, bank, column) addresses to physical DRAM addresses. General-purpose processors use complex functions to map physical addresses to DDRx addresses (e.g., DRAM banks, rows and columns) [62]. The operating system (OS) maps virtual addresses to physical addresses to provide isolation between multiple processes. Only these virtual addresses are exposed to the programmer. Without control over the virtual address $\rightarrow$ DRAM address mapping, the programmer *cannot* easily place data in a way that satisfies the mapping and alignment requirements of an in-DRAM copy operation.

We implement a new memory allocation mechanism that can perform memory allocation for RowClone (in-DRAM copy/initialization) operations. This mechanism enables page-granularity RowClone operations (i.e., a virtual page can be copied to another virtual page using RowClone) *without* introducing any changes to the programming model. The mechanism places the operands of RowClone operations in the same DRAM subarray while maximizing the bank-level parallelism in regular DRAM accesses (reads & writes) to these operands (such that the commonly-performed streaming accesses to these operands benefit from bank-level parallelism in DRAM). Figure 6 depicts an overview of our memory allocation mechanism.

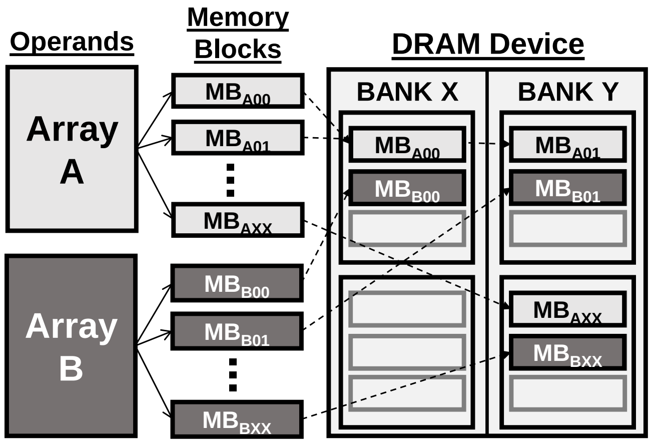

Figure 6: Overview of our memory allocation mechanism

At a high level, our memory allocation mechanism (i) splits the source and destination operands into page-sized virtually-addressed memory blocks, (ii) allocates two physical pages in different DRAM rows in the same DRAM subarray, (iii) assigns these physical pages to virtual pages that correspond to the source and destination memory blocks at the same index such that the source block can be copied to the destination block using RowClone. We repeat this process until we exhaust the page-sized memory blocks. As the mechanism processes subsequent page-sized memory blocks of the two operands, it allocates physical pages from a different DRAM bank to maximize bank-level parallelism in streaming accesses to these operands.

To overcome the *mapping*, *alignment*, and *granularity* problems, we implement our memory management mechanism in the custom supervisor software of PiDRAM. We expose the allocation mechanism using the `alloc_align(N, ID)` system call. The system call returns a pointer to a contiguous array of *N* bytes in the virtual address space (i.e., one operand). Multiple calls with the same *ID* to `alloc_align(N, ID)` place the allocated arrays in the same subarray in DRAM, such that they can be copied from one to another using RowClone. If *N* is too large such that it exceeds the size of available physical memory, `alloc_align` fails and causes an exception. Our implementation of RowClone requires application developers to directly use `alloc_align` to allocate data instead of `malloc` and similar function calls.

The custom supervisor software maintains three key structures to make `alloc_align()` work: (i) Subarray Mapping Table (SAMT), (ii) Allocation ID Table (AIT), and (iii) Initializer Rows Table (IRT).

1) Subarray Mapping Table (SAMT). We use the **S**ubarray **Ma**pping **T**able (SAMT) to maintain a list of physical page addresses that point to DRAM rows that are in the same DRAM subarray. `alloc_align()` queries SAMT to find physical addresses that map to rows in one subarray.

SAMT contains the physical pages that point to DRAM rows in each subarray. SAMT is indexed using subarray identifiers (SA IDs) in the range *[0, number of subarrays)*. SAMT contains an entry for every subarray. An entry consists of two elements: (i) the number of free physical address tuples and (ii) a list of physical address tuples. Each tuple in the list contains two physical addresses that respectively point to the first and second halves of the same DRAM row. The list of tuples contains all the physical addresses that point to DRAM rows in the DRAM subarray indexed by the SAMT entry. We allocate free physical pages listed in an entry and assign them to the virtual pages (i.e., memory blocks) that make up the row-copy operands (i.e., arrays) allocated by `alloc_align()`. We slightly modify our high-level memory allocation mechanism to allow for two memory blocks (4 KiB virtually-addressed pages) of an array to be placed in the same DRAM row, as the page size in our system is 4 KiB, and the size of a DRAM row is 8 KiB. We call two memory blocks in the same operand that are placed in the same DRAM row *sibling memory blocks* (also called sibling pages). The parameter *N* of the `alloc_align()` call defines this relationship: We designate memory blocks that are precisely *N/2* bytes apart as *sibling memory blocks*.

Finding the DRAM Rows in a Subarray. Finding the DRAM row addresses that belong to the same subarray is not straightforward due to DRAM-internal mapping schemes employed by DRAM manufacturers (Section 5.1). It is extremely difficult to learn which DRAM address (i.e., bank-row-column) is actually mapped to a physical location (e.g., a subarray) in the DRAM device, as these mappings are not exposed through publicly accessible datasheets or standard definitions [71, 116, 130]. We make the key observation that the entire mapping scheme need *not* be available to successfully perform RowClone operations.

We observe that for a set of *{source, destination}* DRAM row address pairs, RowClone operations repeatedly succeed with a 100% probability. We hypothesize that these pairs of DRAM row addresses are mapped to the same DRAM subarray. We identify these row address pairs by conducting a *RowClone success rate* experiment where we repeatedly perform RowClone operations between every *source, destination* row address pair in a DRAM bank. Our experiment works in three steps: we (i) initialize both the source and the destination row with random data, (ii) perform a RowClone operation from the source to the destination row, and (iii) compare the data in the destination row with the source row. RowClone success rate is calculated as the number of bits that differ between the source and destination rows' data divided by the number of bits stored in a row (8 KiB in our prototype). If there is no difference between the source and the destination rows' data (i.e., the RowClone success rate for the source and the destination row is 100%), we infer that the RowClone operation was successful. We repeat the experiment for 1000 iterations for each row address pair and if every iteration is successful, we store the address pair in the SAMT, indicating that the row address pair is mapped to different rows in the same DRAM subarray. The same RowClone success rate experiment could be conducted in other systems that are based on PiDRAM or in a PiDRAM prototype that uses a different DRAM module. Since the RowClone success rate experiment is a one-time process, its overheads (e.g., time taken to iterate over all DRAM rows using our experiment) are amortized over the lifetime of such a system.

2) Allocation ID Table (AIT). To keep track of different operands that are allocated by `alloc_align` using the same *ID* (used to place different arrays in the same subarray), we use the **A**llocation **I**D **T**able (AIT). AIT entries are indexed by *allocation ID*s (the parameter *ID* of the *alloc_align* call). Each AIT entry stores a pointer to an SAMT entry. The SAMT entry pointed by the AIT entry contains the set of physical addresses that were allocated using the same *allocation ID*. AIT entries are used by the `alloc_align` function to find which DRAM subarray can be used to allocate DRAM rows from, such that the newly allocated array can be copied to other arrays allocated using the same *ID*.

3) Initializer Rows Table (IRT). To find which row in a DRAM subarray can be used as the source operand in zero-initialization (RowClone-Initialize) operations, we maintain the **I**nitializer **R**ows **T**able (IRT). The IRT is indexed using physical page numbers. RowCopy-Initialize operations query the IRT to obtain the physical address of the DRAM row that is initialized with zeros and that belong to the same subarray as the destination operand (i.e., the DRAM row to be initialized with zeros).

Figure 7 describes how `alloc_align()` works over an end-to-end example. Using the RowClone success rate experiment (described above), the custom supervisor software (CSS for short) finds the DRAM rows that are in the same subarray (❶) and initializes the Subarray Mapping Table (SAMT). The programmer allocates two 128 KiB arrays, A and B, via `alloc_align()` using the same *allocation id* (**0**), with the intent to copy from A to B (❷). CSS allocates contiguous ranges of virtual addresses to A and B, and then splits the virtual address ranges into page-sized memory blocks (❸). CSS assigns consecutive memory blocks to consecutive DRAM banks and accesses the Allocation ID Table (AIT) with the *allocation id* (❹) for each memory block. By accessing the AIT, CSS retrieves the *subarray id* that points to a SAMT entry. The SAMT entry corresponds to the subarray that contains the arrays that are allocated using the *allocation id* (❺). CSS accesses the SAMT entry to retrieve two physical addresses that point to the same DRAM row. CSS maps a memory block and its *sibling memory block* (i.e., the memory block that is N/2 bytes away from this memory block, where N is the *size* argument of the `alloc_align()` call) to these two physical addresses, such that they are mapped to the first and the second halves of the same DRAM row (❻). Once allocated, these physical addresses are pinned to main memory and cannot be swapped out to storage. Finally, CSS updates the page table with the physical addresses to map the memory blocks to the same DRAM row (❼).

5.3. Maintaining Memory Coherence

Since memory instructions update the cached copies of data (Section 5.1), a naive implementation of RowClone can potentially operate on stale data because cached copies of RowClone operands can be modified by CPU store instructions. Thus, we need to ensure memory coherence to prevent RowClone from operating on stale data.

We implement a new custom RISC-V instruction, called *CLFLUSH*, to flush dirty cache blocks to DRAM (RISC-V does not implement any cache management operations [161]) so as to ensure RowClone operates on up-to-date data. A CLFLUSH instruction flushes (invalidates) a physically addressed dirty (clean) cache block. CLFLUSH or other cache management operations with similar semantics are supported in X86 [68] and ARM architectures [10]. Thus, the CLFLUSH instruction (that we implement) provides a minimally invasive solution (i.e., it requires no changes to the specification of commercial ISAs) to the memory coherence problem.

We modify the non-blocking data cache and the Rocket core modules (defined in *NBDCache.scala* and *rocket.scala* in Rocket Chip [11], respectively) to implement CLFLUSH. We modify the RISC-V GNU compiler toolchain [135] to expose CLFLUSH as an instruction to C/C++ applications. Before executing a RowClone Copy or Initialization operation (see Section 5.4), the custom supervisor software flushes (invalidates) the cache blocks of the source (destination) row of the RowClone operation using CLFLUSH.

5.4. RowClone-Copy and RowClone-Initialize

We support the RowClone-Copy and RowClone-Initialize operations in our custom supervisor software via two functions: (i) RowClone-Copy, `rcc(void *dest, void *src, int size)` and (ii) RowClone-Initialize, `rci(void* dest, int size)`. `rcc` copies *size* number of contiguous bytes in the virtual address space starting from the *src* memory address to the *dest* memory address. `rci` initializes *size* number of contiguous bytes in the virtual address space starting from the *dest* memory address. We expose `rcc` and `rci` to user-level programs using system calls defined in the custom supervisor software.

`rcc` (i) splits the source and destination operands into page-aligned, page-sized blocks, (ii) traverses the page table (Figure 7-❽) to find the physical address of each block (i.e., the address of a DRAM row), (iii) flushes all cache blocks corresponding to the source operand and invalidates all cache blocks corresponding to the destination operand, and (iv) performs a RowClone operation from the source row to the destination row using pumolib's `copy_row()` function.

`rci` (i) splits the destination operand into page-aligned, page-sized blocks, (ii) traverses the page table to find the physical address of the destination operand, (iii) queries the Initializer Rows Table (IRT, see Section 5.2) to obtain the physical address of the initializer row (i.e., source operand), (iv) invalidates the cache blocks corresponding to the destination operand, and (v) performs a RowClone operation from the initializer row to the destination row using using pumolib's `copy_row()` function.

5.5. Evaluation

We evaluate our solutions for the challenges in implementing RowClone end-to-end on a real system using PiDRAM. We modify the custom memory controller to implement DRAM command sequences ($ACT \rightarrow PRE \rightarrow ACT$) to trigger RowClone operations. We set the $tRAS$ and tRP parameters to 10 ns (below the manufacturer-recommended $37.5\,\mathrm{ns}$ for tRAS and $13.5\,\mathrm{ns}$ for tRP [117]). We modify our custom supervisor software to implement our memory allocation mechanism and add support for RowClone-Copy (`rcc`) and RowClone-Initialize (`rci`) operations.

5.5.1. Experimental Methodology. Table 3 describes the configuration of the components in our system. We use the pipelined and in-order Rocket core with 16 KiB L1 data cache and 4-entry TLB as the main processor of our system. We use the 1 GiB DDR3 module available on the ZC706 board as the main memory where we conduct PuM operations.

Table 3: PiDRAM system configuration

CPU: 50 MHz; in-order Rocket core [11]; **TLB** 4 entries DTLB; LRU policy
L1 Data Cache: 16 KiB, 4-way; 64 B line; random replacement policy
DRAM Memory: 1 GiB DDR3; 800MT/s; single rank; 8 KiB row size

Implementing RowClone requires an additional 198 lines of Verilog code over PiDRAM's existing Verilog design. We add 43 and 522 lines of C code to pumolib and to our custom supervisor software, respectively, to implement RowClone in the software components.

Table 8 describes the mapping scheme we use in our custom memory controller to translate from physical to DRAM row-bank-column addresses. We map physical addresses to DRAM columns, banks, and rows from lower-order bits to higher-order bits to exploit the bank-level parallelism in memory accesses to consecutive physical pages. We note that our memory management mechanism is compatible with other physical address $\rightarrow$ DRAM address mappings [62]. For example, for a mapping scheme where page offset bits (physical address (PA) [11:0]) include all or a subset of the bank address bits, a single RowClone operand (i.e., a 4 KiB page) would be split across

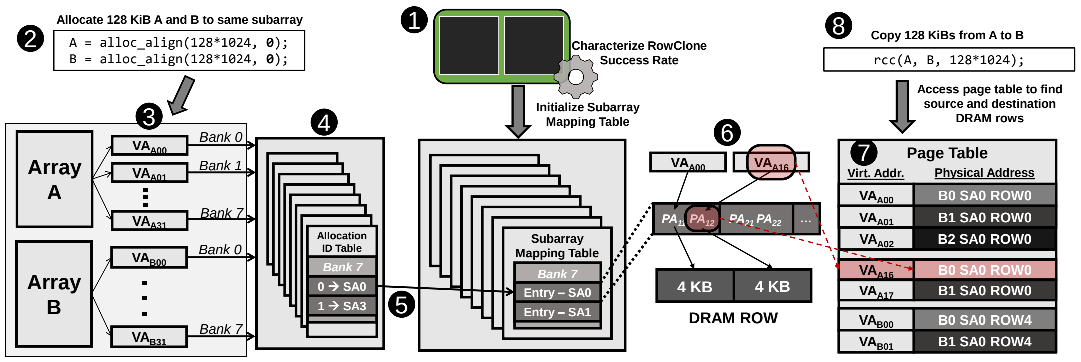

Figure 7: `Alloc_align() and RowClone-Copy (rcc, see Section 5.4) workflow.`

multiple DRAM banks. This only coarsens the granularity of RowClone operations as the sibling pages that must be copied in unison, to satisfy the granularity constraint, increases. We expect that for other complex or unknown physical address $\rightarrow$ DRAM address mapping schemes, the characterization of the DRAM device for RowClone success rate would take longer. In the worst case, DRAM row addresses that belong to the same DRAM subarray can be found by testing all combinations of physical addresses for their RowClone success rate.

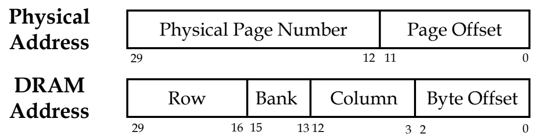

Figure 8: Physical address to DRAM address mapping in PiDRAM. Byte offset is used to address the byte in the DRAM burst.

We evaluate `rcc` and `rci` operations under two configurations to understand the copy/initialization throughput improvements provided by `rcc` and `rci` over traditional CPU-copy operations performed by the Rocket core, and to understand the overheads introduced by end-to-end support for commodity DRAM based PuM operations. We test two configurations: (i) *Bare-Metal*, to find the maximum RowClone throughput our implementation provides solely using pumolib, and (ii) *No Flush*, to understand the benefits our end-to-end implementation (i.e., with system support) of RowClone can provide in copy/initialization throughput when data in DRAM is up-to-date (i.e., when no coherence operations are needed).

Bare-Metal. We assume that RowClone operations always target data that is allocated correctly in DRAM (i.e., there is no overhead introduced by address translation, IRT accesses, and CLFLUSH operations). We directly issue RowClone operations via pumolib using physical addresses. Traditional CPU-copy operations (executed on the Rocket core) also use physical addresses.

No Flush. We assume that the programmer uses the `alloc_align` function to allocate the operands of RowClone operations. We use a version of `rcc` and `rci` system calls that do not use CLFLUSH to flush cache blocks of source and destination operands of RowClone operations. We run the *No Flush* configuration on our custom supervisor software. Both `rcc` and `rci`, and traditional CPU-copy operations use virtual addresses.

5.5.2. Workloads. For the two configurations, we run a microbenchmark that consists of two programs, *copy* and *init*, on our prototype. Both programs take the argument N, where *copy* copies an N-byte array to another N-byte array and *init* initializes an N-byte array to all zeros. Both programs have two versions: (i) CPU-copy, which copies/initializes data using memory loads and stores, (ii) RowClone, which uses RowClone operations to perform copy/initialization. All programs use `alloc_align` to allocate data. The performance results we present in this section are the average of a 1000 runs. To maintain the same initial system state for both CPU-copy and RowClone, we flush all cache blocks before each one of the 1000 runs. We run each program for array sizes (*N*) that are powers of two and $8\ KiB < N < 8\ MiB$, and find the average copy/initialization throughput across all 1000 runs (by measuring the # of elapsed CPU cycles to execute copy/initialization operations) for CPU-copy, RowClone-Copy (`rcc`), and RowClone-Initialize (`rci`).[4]

We analyze the overheads of CLFLUSH operations on copy-/initialization throughput that `rcc` and `rci` can provide. We measure the execution time of CLFLUSH operations in our prototype to find how many CPU cycles it takes to flush a (i) dirty and (ii) clean cache block on average across 1000 measurements. We simulate various scenarios (described in Figure 11) where we assume a certain fraction of the operands of RowClone operations are cached and dirty.

5.5.3. Bare-Metal RowClone. Figure 9 shows the throughput improvement provided by `rcc` and `rci` for *copy* and *initialize* over CPU-copy and CPU-initialization for increasing array sizes.

We make two major observations. First, we observe that `rcc` and `rci` provide significant throughput improvement over traditional CPU-copy and CPU-initialization. The throughput improvement provided by `rcc` ranges from 317.5× (for 8 KiB arrays) to 364.8× (for 8 MiB arrays). The throughput improvement provided by `rci` ranges from 172.4× to 182.4×. Second, the throughput improvement provided by `rcc` and `rci`

[4]We tested RowClone operations using `alloc_align()` with up to 8 MiB of allocation size since we observed diminishing returns on performance improvement provided by RowClone operations on larger array sizes.

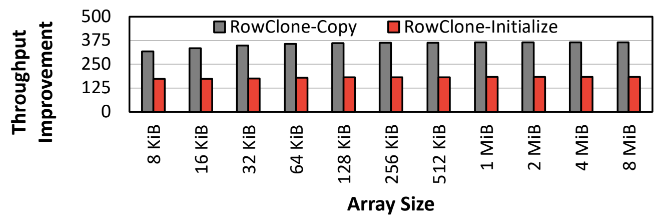

Figure 9: RowClone-Copy and RowClone-Initialize over traditional CPU-copy and -initialization for the Bare-Metal configuration

increases as the array size increases. This increase saturates when the array size reaches 1 MiB. The load/store instructions used by CPU-copy and CPU-initialization access the operands in a streaming manner. The eviction of dirty cache blocks (i.e., the destination operands of copy and initialization operations) interfere with other memory requests on the memory bus.[5] We attribute the observed saturation at 1 MiB array size to the interference on the memory bus.

5.5.4. No Flush RowClone. We analyze the overhead in copy/initialization throughput introduced by system support (Section 5.2). Figure 10 shows the throughput improvement of copy and initialization provided in the *No Flush* configuration by `rcc` and `rci` operations.

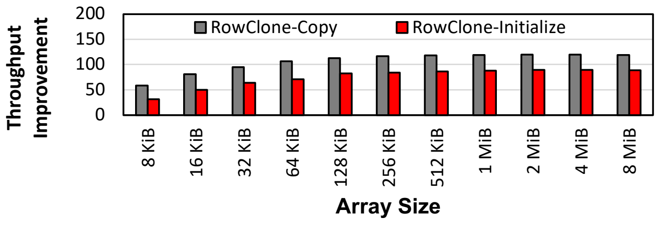

Figure 10: Throughput improvement provided by RowClone-Copy and RowClone-Initialize over traditional CPU-copy and -initialization for the NoFlush configuration.

We make two major observations: First, `rcc` improves the copy throughput by 58.3× for 8 KiB and by 118.5× for 8 MiB arrays, whereas `rci` improves initialization throughput by 31.4× for 8 KiB and by 88.7× for 8 MiB arrays. Second, we observe that the throughput improvement provided by `rcc` and `rci` improves *non-linearly* as the array size increases. The execution time (in Rocket core clock cycles) of `rcc` and `rci` operations (not shown in Figure 10) *does not increase linearly* with the array size. For example, the execution time of `rcc` is 397 and 584 cycles at 8 KiB and 16 KiB array sizes, respectively, resulting in a 1.47 × increase in execution time between 8 KiB and 16 KiB array sizes. However, the execution time of `rcc` is 92,656 and 187,335 cycles at 4 MiB and 8 MiB array sizes, respectively, resulting in a 2.02 × increase in execution time between 4 MiB and 8 MiB array sizes. We make similar observations on the execution time of `rci`. For every RowClone operation, `rcc` and `rci` walk the page table to find the physical addresses corresponding to the source (`rcc`) and the destination (`rcc` and `rci`) operands. We attribute the non-linear increase in `rcc` and `rci`'s execution time to (i) the locality exploited by the Rocket core in accesses to the page table and (ii) the diminishing constant cost in the execution time of both `rcc` and `rci` due to common instructions executed to perform a system call.

5.5.5. CLFLUSH Overhead. We find that our implementation of CLFLUSH takes 45 Rocket core clock cycles to flush a dirty cache block and 6 Rocket core cycles to invalidate a clean cache block. We estimate the throughput improvement of `rcc` and `rci` including the CLFLUSH overhead. We assume that all cache blocks of the source and destination operands are cached, and that a fraction of the all cached cache blocks is dirty (quantified on the x-axis). We do not include the overhead of accessing the data (e.g., by using *load* instructions) *after* the data gets copied in DRAM. Figure 11 shows the estimated improvement in copy and initialization throughput that `rcc` and `rci` provide for 8 MiB arrays.

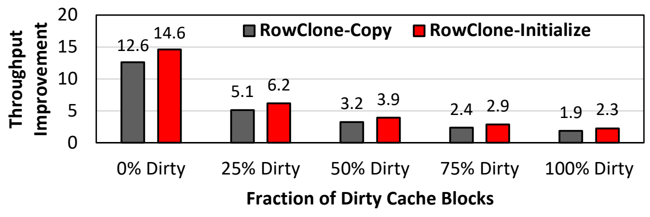

Figure 11: Throughput improvement provided by `rcc` and `rci` with CLFLUSH over Rocket's CPU-copy.

We make three major observations. First, even with inefficient cache flush operations, `rcc` and `rci` provide 3.2× and 3.9× higher throughput over the CPU-copy and CPU-initialization operations, assuming 50% of the cache blocks of the 8 MiB source operand are dirty, respectively. Second, as the fraction of dirty cache blocks increases, the throughput improvement provided by both `rcc` and `rci` decreases (down to 1.9× for `rcc` and 2.3× for `rci` for 100% dirty cache block fraction). Third, we observe that `rci` can provide better throughput improvement compared to `rcc` when we include the CLFLUSH overhead. This is because `rci` flushes cache blocks of one operand (destination), whereas `rcc` flushes cache blocks of both operands (source and destination).

We do not study the distribution of dirty cache block fractions in real applications as that is not the goal of our CLFLUSH overhead analysis. However, if a large dirty cache block fraction causes severe overhead in a real application, the system designer or the user of the system would likely decide not to offload the operation to PuM (i.e., performing `rcc` operations instead of CPU-Copy). PiDRAM's prototype can be useful for studies on different PuM system integration aspects, including such offloading decisions.

We observe that the CLFLUSH operations are inefficient in supporting coherence for RowClone operations. Even so, we see that RowClone-Copy and RowClone-Initialization provides throughput improvements ranging from 1.9× to 14.6×. We expect the throughput improvement benefits to increase

[5]Because the data cache in our prototype employs random replacement policy, as the array size increases, the fraction of cache evictions among all memory requests also increases, causing larger interference on the memory bus (i.e., more memory requests to satisfy all cache evictions). The interference saturates at 1 MiB array size.

as coherence between the CPU caches and PIM accelerators become more efficient with new techniques [21, 22, 146].

5.5.6. Real Workload Study. The benefit of `rcc` and `rci` on a full application depends on what fraction of execution time is spent on bulk data copy and initialization. We demonstrate the benefit of `rcc` and `rci` on *forkbench* [148] and *compile* [148] workloads with varying fractions of time spent on bulk data copy and initialization, to show that our infrastructure can enable end-to-end execution and estimation of benefits on real workloads.[6] We especially study *forkbench* in detail to demonstrate how the benefits vary with the time spent on data copying in the baseline for this workload.

Forkbench first allocates N memory pages and copies data to these pages from a buffer in the process's memory and then accesses 32K random cache blocks within the newly allocated pages to emulate a workload that frequently spawns new processes. We evaluate *forkbench* under varying bulk data copy sizes where we sweep N from 8 to 2048 in increasing powers of two.

Compile first zero-allocates (`calloc` or `rci`) two pages (8 KiBs) and then executes a number of arithmetic and memory instructions to operate on the zero-allocated data. We carefully develop the *compile* microbenchmark to maintain a realistic ratio between the number of arithmetic and memory instructions executed and zero-allocation function calls made, which we obtain by profiling *gcc* [109]. We use the *No-Flush* configuration of our RowClone implementation for both *forkbench* and *compile*.

Figure 12 plots the speedup provided by `rcc` over the CPU-copy (bars, left y-axis) baseline, and the proportion of time spent on `memcpy` functions by the CPU-copy baseline (blue curve, right y-axis), for various configurations of *forkbench* on the x-axis.

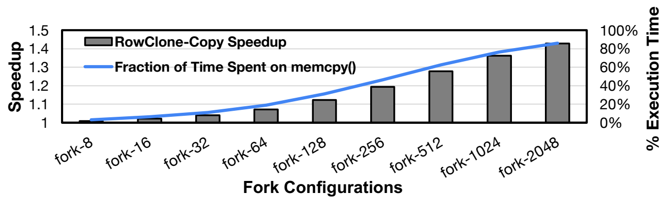

Figure 12: *Forkbench* speedup (bars, left y-axis) and time spent on `memcpy` by the CPU baseline (curve, right y-axis)

Forkbench. We observe that RowClone-Copy can significantly improve the performance of *forkbench* by up to 42.9%. RowClone-Copy's performance improvement increases as the number of pages copied increase. This is because the copy operations accelerated by `rcc` contribute a larger amount to the total execution time of the workload. The `memcpy` function calls take 86% of the CPU-copy baseline's time during *forkbench* execution for N = 2048.

[6]A full workload study (i.e., with system calls to a full operating system such as Linux) of *forkbench* and *compile* is out of the scope of this paper. Our infrastructure currently cannot execute all possible workloads due to the limited library and system call functionality provided by the RISC-V Proxy Kernel [136].

Compile. RowClone-Initialize improves the performance of *compile* by 9%. Only an estimated 17% of the execution time of *compile* is used for zero-allocation by the CPU-initialization baseline, `rci` reduces the overhead of zero-allocation by (i) performing in-DRAM bulk-initialization and (ii) executing a smaller number of instructions.

Libquantum. To demonstrate that PiDRAM can run real workloads, we run a SPEC2006 [154] workload (libquantum). We modify the `calloc` (allocates and zero initializes memory) function call to allocate data using `alloc_align`, and initialize data using `rci` for allocations that are larger than 8 KiBs.

Using `rci` to bulk initialize data in libquantum improves end-to-end application performance by 1.3% (compared to the baseline that uses CPU-Initialization). This improvement is brought by `rci`, which initializes a total amount of 512 KiBs of memory[7] using RowClone operations. We note that the proportion of store instructions executed by libquantum to initialize arrays in the CPU-initialization baseline is only 0.2% of all dynamic instructions in the libquantum workload which amounts to an estimated 2.3% of the total runtime of libquantum. Thus, the 1.3% end-to-end performance improvement provided by `rci`, which can ideally speed up only 2.3% of the total runtime, is reasonable, and we expect it to increase with the initialization intensity of workloads.

Summary. We conclude from our evaluation that end-to-end implementations of RowClone (i) can be efficiently supported in real systems by employing memory allocation mechanisms that satisfy the memory *alignment*, *mapping*, *granularity* requirements (Section 5.1) of RowClone operations, (ii) can greatly improve copy/initialization throughput in real systems, and (iii) require cache coherence mechanisms (e.g., PIM-optimized coherence management [21, 22, 146]) that can flush dirty cache blocks of RowClone operands efficiently to achieve optimal copy/initialization throughput improvement. PiDRAM can be used to estimate end-to-end workload execution benefits provided by RowClone operations. Our experiments using libquantum, forkbench, and compile show that (i) PiDRAM can run real workloads, (ii) our end-to-end implementation of RowClone operates correctly, and (iii) RowClone can improve the performance of real workloads in a real system, even when inefficient CLFLUSH operations are used to maintain memory coherence.

6. Case Study #2: End-to-end D-RaNGe

Prior work on DRAM-based random number generation techniques [13, 83, 123] do not integrate and evaluate their techniques end-to-end in a real system. We evaluate one DRAM-based true random number generation technique, D-RaNGe [83], end-to-end using PiDRAM. We implement support for D-RaNGe in PiDRAM by enabling access to DRAM with reduced activation latency (i.e., $tRCD$ set to values lower than manufacturer recommendations).

[7]In libquantum, there are 16 calls to `calloc` that exceed the 8 KiB allocation size. We only bulk initialize data using `rci` for these 16 calls.

6.1. D-RaNGe Implementation

We implement a simple version of D-RaNGe in PiDRAM. PiDRAM's D-RaNGe controller collects true random numbers from four DRAM cells in the same DRAM cache block inside one DRAM bank. We implement the D-RaNGe controller within the Periodic Operations Module (Section 4.1). The D-RaNGe controller (i) periodically accesses a DRAM cache block with reduced tRCD, (ii) reads four of the TRNG DRAM cells in the cache block, (iii) stores the four bits read from the TRNG cells in a 1 KiB random number buffer. We reserve multiple configuration registers in the configuration register file (CRF) to configure (i) the TRNG period (in nanoseconds) used by the D-RaNGe controller to periodically generate random numbers by accessing DRAM with reduced activation latency while the buffer is not full (the D-RaNGe controller accesses DRAM every TRNG period), (ii) the timing parameter ($tRCD$) used to induce activation latency failures and (iii) the physical location (DRAM bank, row, column addresses, and bit offset within the DRAM column) of the TRNG cells to read. We implement two pumolib functions: (i) `buf_size()`, which returns the number of random words (4-bytes) available in the buffer, and (ii) `rand_dram()`, which returns one random word that is read from the buffer. The two functions first execute PiDRAM instructions in the POC that update the data register either with (i) the number of random words available (when buf_size() is called) or (ii) a random word read from the random number buffer (when rand_dram() is called). The two functions then access the data register using LOAD instructions to retrieve either the size of the random number buffer or a random number. The application developer reads true random numbers using these two functions in pumolib.

Random Cell Characterization. D-RaNGe requires the system designer to characterize the DRAM module for activation latency failures to find DRAM cells that fail with a 50% probability (i.e., randomly) when accessed with reduced $tRCD$. Following the methodology presented in [83], the system designer can characterize a DRAM device or use an automated procedure to find cells that fail with a 50% probability. In PiDRAM, we implement reduced latency access to DRAM by (i) extending the scheduler of the custom memory controller and (ii) adding a pumolib function `activation_failure(address)` which induces an activation failure on the DRAM cache block pointed by the `address` parameter.

6.2. Evaluation and Results

Experimental Methodology. We run a microbenchmark to understand the effect of the TRNG period on true random number generation throughput observed by a program running on the Rocket core. The microbenchmark consists of a loop that (i) checks the availability of random numbers using `buf_size()` and (ii) reads a random number from the buffer using `rand_dram()`. We execute the microbenchmark until we read one million bytes of random numbers.

Results. The D-RaNGe controller can perform reduced-latency accesses frequently, every 220 ns. Figure 13 depicts the TRNG throughput observed by the microbenchmark for

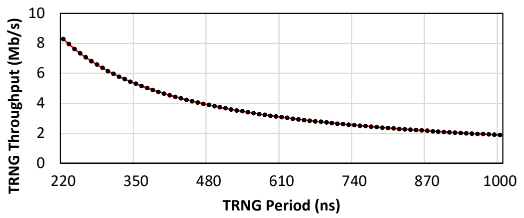

Figure 13: TRNG throughput observed by our microbenchmark for TRNG periods ranging from 220 ns to 1000 ns

TRNG periods in the range [220 ns, 1000 ns] with increments of 10 ns. We observe that the TRNG throughput decreases from 8.30 Mb/s at 220 ns TRNG period to 1.90 Mb/s at 1000 ns TRNG period. D-RaNGe [83] reports 25.2 Mb/s TRNG throughput using a single DRAM bank when there are four random cells in a cache block. PiDRAM's D-RaNGe controller can be optimized to generate random numbers more frequently to match D-RaNGe's observed maximum throughput.[8] We leave such optimizations to PiDRAM's D-RaNGe controller for future work.

Including the modifications to the custom memory controller and pumolib, implementing D-RaNGe and reduced-latency DRAM access requires an additional **190** lines of Verilog and **74** lines of C code over PiDRAM's existing codebase. We conclude that our D-RaNGe implementation (i) provides a basis for PiDRAM developers to study end-to-end implementations of DRAM-based true random number generators, (ii) shows that PiDRAM's hardware and software components facilitate the implementation of new commodity DRAM based PuM techniques, specifically those that are related to security. Our reduced-latency DRAM access implementation provides a basis for other PuM techniques for security purposes, such as the DRAM-latency physical unclonable functions (DL-PUF [82]) and QUAC-TRNG [124] (Section 4.4). We leave further exploration on end-to-end implementations of D-RaNGe, DL-PUF, and QUAC-TRNG, as well as end-to-end analyses of the security benefits they provide using PiDRAM for future work.

7. Extending PiDRAM

We briefly describe the modifications required to extend PiDRAM (i) with new DRAM commands and DRAM timing parameters, (ii) with new case studies, and (iii) to support new FPGA boards.

New DRAM Commands and Timing Parameters. Implementing new DRAM commands or modifying DRAM timing parameters require modifications to PiDRAM's memory controller. This is straightforward as PiDRAM's memory controller's Verilog design is modular and uses well-defined interfaces: It is composed of multiple modules that perform separate tasks. For example, the memory request scheduler comprises

[8]D-RaNGe has a smaller true random number generation (TRNG) latency (i.e., takes a smaller amount of time to generate a 4-bit random number) than PiDRAM. PiDRAM has a larger TRNG latency due to (i) discrepancies in the data path (i.e., on-chip interconnect) in D-RaNGe's simulated system and PiDRAM's prototype and (ii) the TRNG period of the D-RaNGe controller (D-RaNGe controller performs a reduced $tRCD$ access only as frequently as one every 220 ns). The D-RaNGe controller can be optimized further to reduce the TRNG period by down to the DRAM row cycle time (tRC standard timing parameter, typically 45 ns [117]).

two main components: (1) *command timer*, and (2) *command scheduler*. To serve LOAD and STORE memory requests, the command scheduler maintains state (e.g., which row is active) for every bank. The command scheduler selects the next DRAM command to satisfy the LOAD or STORE memory request and queries the command timer with the selected DRAM command. The command timer checks for all possible standard DRAM timing constraints and outputs a valid bit if the selected command can be issued in that FPGA clock cycle. To extend the memory controller with a new standard DRAM command (e.g., to implement a newer standard like DDR4 or DDR5), a PiDRAM developer simply needs to (i) add a new timing constraint by replicating the logic in the command timer and (ii) extend the command scheduler to correctly maintain the bank state.

New Case Studies. Implementing new techniques (e.g., those that are listed in Table 2) to perform new case studies requires modifications to PiDRAM's hardware and software components. We describe the required modifications over an example ComputeDRAM-based in-DRAM bitwise operations case study.

To implement ComputeDRAM-based in-DRAM bitwise operations, the developers need to (i) extend the *custom command scheduler* in PiDRAM's memory controller with a new state machine that schedules new DRAM command sequences (ACT-PRE-ACT) with an appropriate set of violated timing parameters (our ComputeDRAM-based in-DRAM copy implementation provides a solid basis for this), (ii) expose the functionality to the processor by implementing new PiDRAM instructions in the PuM controller (e.g., by replicating and customizing the existing logic for decoding and executing RowClone operations), (iii) and make modifications to the software library to expose the new instruction to the programmer (e.g., by replicating the copy_row function's behavior, described in Table 1).

Porting to New FPGA Boards. Developing new PiDRAM prototypes on different FPGA boards could require modifications to design constraints (e.g., top level input/outputs to physical FPGA pins) and the DDRx PHY IP depending on the FPGA board. Modifying design constraints is a straightforward task involving looking up the FPGA manufacturer datasheets and modifying design constraint files [165]. Manufacturers may provide different DDRx PHY IPs for different FPGAs. Fortunately, these IPs typically expose similar (based on the DFI standard [33]) interfaces to user hardware (in our case, to PiDRAM's memory controller). Thus, other PiDRAM prototypes on different FPGA boards can be developed with small yet careful modifications to the ZC706 prototype design we provide.

8. Related Work

To our knowledge, this is the first work to develop a flexible, open-source framework that enables integration and evaluation of commodity DRAM based processing-using-memory (PuM) techniques on real DRAM chips by providing the necessary hardware and software components. We demonstrate the first end-to-end implementation of RowClone and D-RaNGe using real DRAM chips. We compare the features of PiDRAM with other state-of-the-art prototyping and evaluation platforms in Table 4 and discuss them below. The four features we use for comparison are:

1. **Interface with real DRAM chips:** The platform allows running experiments using real DRAM chips.
2. **Flexible memory controller (MC) for PuM:** The platform provides a flexible memory controller that can easily be extended to perform (e.g., as in PiDRAM) or emulate (e.g., as in PiMulator [119]) new PuM operations.
3. **System software support:** The platform provides support for running system software such as operating systems or supervisor software (e.g., RISC-V PK [136]).
4. **Open-source:** The platform is available as open source software.

Silent-PIM [78]. Silent-PIM proposes a new DRAM design that incorporates processing units capable of vector arithmetic computation. Silent-PIM's goal is to evaluate PIM techniques on a *new, PIM-capable* DRAM device using standard DRAM commands (e.g., as defined in DDR4 [71]); it does not provide an evaluation platform or prototype. In contrast, PiDRAM is designed for researchers to rapidly integrate and evaluate PuM techniques that use *real DRAM devices*. PiDRAM provides key hardware and software components that facilitate end-to-end implementations of PuM techniques.

SoftMC [52, 60] SoftMC is an FPGA-based DRAM testing infrastructure. SoftMC can issue arbitrary sequences of DDR3 commands to real DRAM modules. SoftMC is widely used in prior work that studies the performance, reliability and security of real DRAM chips [13, 14, 28, 38, 41, 50, 59, 77, 83, 85, 96, 127, 155]. SoftMC is built to test DRAM modules, *not* to study end-to-end implementations of PuM techniques. Thus, SoftMC (i) does *not* support application execution on a real system, and (ii) *cannot* use DRAM modules as main memory. While SoftMC is useful in studies that perform exhaustive search on all possible sequences of DRAM commands to potentially uncover undocumented DRAM behavior (e.g., ComputeDRAM [44], QUAC-TRNG [123]), PiDRAM is developed to study end-to-end implementations of PuM techniques. PiDRAM provides an FPGA-based prototype that comprises a RISC-V system and supports using DRAM modules both for storing data (i.e., as main memory) and performing PuM computation.

ComputeDRAM [44]. ComputeDRAM partially demonstrates that two DRAM-based state-of-the-art PuM techniques, RowClone [148] and Ambit [145], are already possible on real off-the-shelf DDR3 chips. ComputeDRAM uses SoftMC to demonstrate in-DRAM copy and bitwise AND/OR operations on real DDR3 chips. ComputeDRAM's goal is *not* to develop a framework to facilitate end-to-end implementations of PuM techniques. Therefore, it does *not* provide (i) a flexible memory controller for PuM or, (ii) support for system software. PiDRAM provides the necessary software and hardware components to facilitate end-to-end implementations of PuM techniques.

MEG [174]. MEG is an open-source system emulation platform for enabling FPGA-based operation interfacing with High-Bandwidth Memory (HBM). MEG aims to efficiently re-

Table 4: Comparison of PiDRAM with related state-of-the-art prototyping and evaluation platforms

Platforms	Interface with real DRAM chips	Flexible MC for PuM	System software support	Open-source
Silent-PIM [78]	✗	✗	✓	✗
SoftMC [60]	✓(DDR3)	✗	✗	✓
ComputeDRAM [44]	✓(DDR3)	✗	✗	✗
MEG [174]	✓(HBM)	✗	✓	✓
PiMulator [119]	✗	✓	✗	✓
Commercial platforms (e.g., ZYNQ [166])	✓(DDR3/4)	✗	✓	✗
Simulators [18, 35, 90, 132, 140, 169, 170, 175]	✗	✓	✓(potentially)	✓
PiDRAM (this work)	✓(DDR3)	✓	✓	✓

trieve data from HBM and perform the computation in the host processor implemented as a soft core on the FPGA. Unlike PiDRAM, MEG does *not* implement a flexible memory controller that is capable of performing PuM operations. We demonstrate the flexibility of PiDRAM by implementing two state-of-the-art PuM techniques [83, 148]. We believe MEG and PiDRAM can be combined to get the functionality and prototyping power of both works.

PiMulator [119]. PiMulator is an open-source PiM emulation platform. PiMulator implements a main memory and a PiM model using SystemVerilog, allowing FPGA emulation of PiM architectures. PiMulator enables easy emulation of new PiM techniques. However, it does *not* allow end-to-end execution of workloads that use PiM techniques and it does not provide the user with full control over the DRAM interface.

Commercial Platforms (e.g., ZYNQ [166]). Some commercial platforms implement CPU-FPGA heterogeneous computing systems. A memory controller and necessary hardware-software modules are provided to access DRAM as the main memory in such systems. However, in such systems, (i) there is *no* support for PuM mechanisms, and (ii) the entire hardware-software stack is closed-source. PiDRAM can be integrated into these systems, using the closed-source computing system as the main processor. Our prototype utilizes an open-source system-on-chip (Rocket Chip [11]) as the main processor, which enables developers to study architectural and microarchitectural aspects of PuM techniques (e.g., data allocation and coherence mechanisms). Such studies cannot be conducted using closed-source computing systems.

Simulators. Many prior works propose full-system (e.g., [18, 132]), trace-based (e.g., [64, 90, 140, 169, 170, 175]), and instrumentation-based (e.g., [35, 64, 169]) simulators that can be used to evaluate PuM techniques. Although useful, these simulators do not model DRAM behavior and cannot integrate proprietary device characteristics (e.g., DRAM internal address mapping) into their simulations, without conducting a rigorous characterization study. Moreover, the effects of environmental conditions (e.g., temperature, voltage) on DRAM chips are unlikely to be modeled on accurate, full-system simulators as it would require excessive computation, which would negatively impact the already poor performance (200K instructions per second) of full system simulators [141]. In contrast, PiDRAM interfaces with real DRAM devices and its prototype achieves a 50 MHz clock speed (and can be improved further) which lets PiDRAM execute > 10M instructions per second (assuming < 5 cycles per instruction). PiDRAM can be used to study end-to-end implementations of PuM techniques and explore solutions that take into account the effects related to the environmental conditions of real DRAM devices. Future versions of PiDRAM could be easily extended (e.g., with real hardware that allows controlling DRAM temperature and voltage [115, 157]) to experiment with different DRAM temperature and voltage levels to better understand the effects of these environmental conditions on the reliability of PuM operations. Using PiDRAM, experiments that require executing real workloads can take an order of magnitude shorter wall clock time compared to using full-system simulators.

Other Related Work. Prior works (see Section 2.2) (i) propose or (ii) demonstrate using real DRAM chips, several DRAM-based PuM techniques that can perform computation [9, 28, 40, 54, 144, 145, 147, 150, 151], move data [148, 160], or implement security primitives [13, 14, 82, 83, 124, 126] in memory. SIMDRAM [54] develops a framework that provides a programming interface to perform in-DRAM computation using the majority operation. DR-STRANGE [23] proposes an end-to-end system design for DRAM-based true random number generators. None of these works provide an end-to-end in-DRAM computation framework that is integrated into a real system using real DRAM chips.

We conclude that existing platforms cannot substitute PiDRAM in studying commodity DRAM based PuM techniques end-to-end.

9. Conclusion

We develop PiDRAM, a flexible and open-source prototyping framework for integrating and evaluating end-to-end commodity DRAM based processing-using-memory (PuM) techniques. PiDRAM comprises the necessary hardware and software structures to facilitate end-to-end implementation of PuM techniques. We build an FPGA-based prototype of PiDRAM along with an open-source RISC-V system and enable computation on real DRAM chips. Using PiDRAM, we implement and evaluate RowClone (in-DRAM data copy and initialization) and D-RaNGe (in-DRAM true random number generation) end-to-end in the entire real system. Our results show that RowClone significantly improves data copy and initialization throughput in a real system on real workloads, and efficient cache coherence mechanisms are needed to maximize RowClone's potential benefits. Our implementation of D-RaNGe requires small additions to PiDRAM's codebase and provides true random numbers at high throughput and with low latency. We conclude that unlike existing prototyping and evaluation platforms, PiDRAM enables (i) easy integration of existing and new PuM techniques end-to-end in a real system and (ii) novel studies on end-to-end implementations of PuM techniques using real DRAM chips. PiDRAM is freely available as an open-source tool for researchers and designers in both academia and industry to experiment with and build on.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *HPCA*, 2017.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[4] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.

[5] M. F. Ali, A. Jaiswal, and K. Roy, "In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology," in *TCAS-I*, 2019.

[6] S. Angizi, Z. He, and D. Fan, "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-efficient Logic Computation," in *DAC*, 2018.

[7] S. Angizi, A. S. Rakin, and D. Fan, "CMP-PIM: An Energy-efficient Comparator-based Processing-in-Memory Neural Network Accelerator," in *DAC*, 2018.

[8] S. Angizi, J. Sun, W. Zhang, and D. Fan, "AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM," in *DAC*, 2019.

[9] S. Angizi and D. Fan, "Graphide: A Graph Processing Accelerator Leveraging In-DRAM-Computing," in *GLSVLSI*, 2019.

[10] ARM, "Cache Maintenance Operations," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0246/h/programmers-model/register-descriptions/cache-maintenance-operations

[11] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. R. Hauser, A. M. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moretó, A. Ou, D. A. Patterson, B. H. Richards, C. Schmidt, S. M. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," ser. Technical Report No. UCB/EECS-2016-17, 2016.

[12] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016.

[13] B. M. S. Bahar Talukder, J. Kerns, B. Ray, T. Morris, and M. T. Rahman, "Exploiting DRAM Latency Variations for Generating True Random Numbers," in *ICCE*, 2019.

[14] B. M. S. Bahar Talukder, B. Ray, D. Forte, and M. T. Rahman, "PreLatPUF: Exploiting DRAM Latency Variations for Generating Robust Device Signatures," in *IEEE Access*, 2019.

[15] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.

[16] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.

[17] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW Architecture for In-Memory Computing," in *DATE*, 2017.

[18] N. Binkert, B. Beckman, A. Saidi, G. Black, and A. Basu, "The gem5 Simulator," *CAN*, 2011.

[19] J. Borghetti, G. Snider, P. Kuekes, J. J. Yang, D. Stewart, and S. Williams, "Memristive Switches Enable Stateful Logic Operations via Material Implication," in *Nature*, 2010.

[20] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[21] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "CoNDA: Efficient Cache Coherence Support for near-Data Accelerators," in *ISCA*, 2019.

[22] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," in *CAL*, 2016.

[23] F. Bostanci, A. Olgun, L. Orosa, A. Yaglikci, J. S. Kim, H. Hassan, O. Ergin, and O. Mutlu, "DR-STRaNGe: End-to-End System Design for DRAM-based True Random Number Generators," in *HPCA*, 2022.

[24] G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, L. L. Sanches, I. Boybat, M. L. Gallo, K. Moon, J. Woo, H. Hwang, and Y. Leblebici, "Neuromorphic Computing Using Non-volatile Memory," in *Advances in Physics: X*, 2017.

[25] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.

[26] K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2017.

[27] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[28] 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 *HPCA*, 2016.

[29] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[30] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *S&P*, 2020.

[31] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing," in *TCAD*, 2018.

[32] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: a DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018.

[33] DFI Group, *DFI 5.0 Specification*, July 2018.

[34] M. P. Drumond Lages De Oliveira, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel Obando, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *ISCA*, 2017.

[35] B. E. Forlin, P. C. Santos, A. E. Becker, M. A. Alves, and L. Carro, "Sim2PIM: A Complete Simulation Framework for Processing-in-Memory," in *JSA*, 2022.

[36] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.

[37] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.

[38] M. Farmani, M. Tehranipoor, and F. Rahman, "RHAT: Efficient RowHammer-Aware Test for Modern DRAM Modules," in *ETS*, 2021.

[39] I. Fernandez, R. Quislant, C. Giannoula, M. Alser, J. Gomez-Luna, E. Gutierrez, O. Plata, and O. Mutlu, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.

[40] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori *et al.*, "pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation," arXiv:2104.07699, 2021.

[41] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[42] D. Fujiki, S. Mahlke, and R. Das, "Duality Cache for Data Parallel Acceleration," in *ISCA*, 2019.

[43] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The Programmable Logic-in-Memory (PLiM) Computer," in *DATE*, 2016.

[44] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.

[45] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.

[46] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.

[47] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[48] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-Memory: A Workload-driven Perspective," in *IBM JRD*, 2019.

[49] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, and O. Mutlu, "Demystifying Complex Workload-DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.

[50] S. Ghose, A. G. Yaglikçi, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.

[51] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.

[52] S. R. Group, "SoftMC v1.0 – GitHub Repository," 2021. [Online]. Available: https://github.com/CMU-SAFARI/SoftMC

[53] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *ISCA*, 2016.

[54] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[55] S. Hamdioui, S. Kvatinsky, and e. a. G. Cauwenberghs, "Memristor for Computing: Myth or Reality?" in *DATE*, 2017.

[56] S. Hamdioui, L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor Based Computation-in-Memory Architecture for Data-intensive Applications," in *DATE*, 2015.

[57] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.

[58] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.

[59] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications," arXiv:2110.10603, 2021.

[60] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.

[61] C. Helm, S. Akiyama, and K. Taura, "Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters," in *MASCOTS*, 2020.

[62] M. Hillenbrand, "Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet," 2017.

[63] M. Horiguchi, "Redundancy Techniques for High-Density DRAMs," in *ISIS*, 1997.

[64] HPS Research Group, "Scarab – Github Repository," 2022. [Online]. Available: https://github.com/hpsresearchgroup/scarab

[65] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[66] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Conner, N. Vijaykumar, O. Mutlu, and S. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[67] Y. Huang, L. Zheng, P. Yao, J. Zhao, X. Liao, H. Jin, and J. Xue, "A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing," in *IPDPS*, 2020.
[68] Intel, "Intel 64 and IA-32 Architectures Software Developer Manuals," 2011. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[69] Intel, "Taking Neuromorphic Computing to the Next Level with Loihi 2," Technology Brief, 2022.
[70] K. Itoh, *VLSI Memory Chip Design*. Springer, 2001.
[71] JEDEC, "DDR4," *JEDEC Standard JESD79–4*, 2012.
[72] H. B. Kang and S. K. Hong, "One-Transistor Type DRAM," US Patent 7701751, 2009.
[73] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM," in *ICASSP*, 2014.
[74] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," in *IEEE Micro*, 2021.
[75] B. Keeth and R. Baker, *DRAM Circuit Design: A Tutorial*. Wiley, 2001.
[76] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM," in *DSN*, 2016.
[77] S. Khan, C. Wilkerson, Z. Wang, A. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[78] C. H. Kim, W. J. Lee, Y. Paik, K. Kwon, S. Y. Kim, I. Park, and S. W. Kim, "Silent-PIM: Realizing the Processing-in-Memory Computing with Standard Memory Requests," *TPDS*, 2021.
[79] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[80] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *SC*, 2017.
[81] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[82] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
[83] J. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.
[84] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," in *BMC Genomics*, 2018.
[85] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.
[86] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah *et al.*, "Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond," in *Hot Chips*, 2021.
[87] Y. Kim, "Architectural Techniques to Enhance DRAM Scaling," Ph.D. dissertation, Carnegie Mellon University, 2015.
[88] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[89] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[90] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2015.
[91] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC—Memristor-Aided Logic," in *IEEE TCAS II: Express Briefs*, 2014.
[92] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-Based IMPLY Logic Design Procedure," in *ICCD*, 2011.
[93] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," in *TVLSI*, 2014.
[94] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[95] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon University, 2016.
[96] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[97] D. Lee, J. So, M. AHN, J.-G. Lee, J. Kim, J. Cho, R. Oliver, V. C. Thummala, R. s. JV, S. S. Upadhya *et al.*, "Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM)," in *DaMoN*, 2022.
[98] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
[99] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[100] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[101] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.
[102] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic Operations in Memory Using a Memristive Akers Array," in *Microelectronics Journal*, 2014.
[103] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.
[104] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.
[105] Linux man-pages Project, "calloc(3p) — Linux manual page," https://man7.org/linux/man-pages/man3/calloc.3p.html, 2022.
[106] Linux man-pages Project, "malloc(3) — Linux manual page," https://man7.org/linux/man-pages/man3/malloc.3.html, 2022.
[107] Linux man-pages Project, "memcpy(3) — Linux manual page," https://man7.org/linux/man-pages/man3/memcpy.3.html, 2022.
[108] Linux man-pages Project, "posix_memalign(3) — Linux manual page," https://man7.org/linux/man-pages/man3/posix_memalign.3.html, 2022.
[109] Linux Wiki, "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, 2021.
[110] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[111] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.
[112] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
[113] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yaglikci, L. Orosa, J. Park, and O. Mutlu, "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in *ISCA*, 2020.
[114] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," in *IBM JRD*, 2002.
[115] Maxwell, "FT20X," https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf, 2022.
[116] Micron, "DDR4 SDRAM Datasheet," 2016.
[117] Micron, "DDR3 SDRAM: MT41J128M8," Data Sheet, 2018.
[118] A. Morad, L. Yavits, and R. Ginosar, "GP-SIMD Processing-in-Memory," in *ACM TACO*, 2015.
[119] S. Mosanu, M. N. Sakib, T. II, E. Cukurtas, A. Ahmed, P. Ivanov, S. Khan, K. Skadron, and M. Stan, "PiMulator: a Fast and Flexible Processing-in-Memory Emulation Platform," in *DATE*, 2022.
[120] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
[121] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[122] D. Niu, S. Li, Y. Wang, W. Han, Z. Zhang, Y. Guan, T. Guan, F. Sun, F. Xue, L. Duan *et al.*, "184QPS/W 64Mb/mm 2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
[123] A. Olgun, M. Patel, A. G. Yaglikci, H. Luo, J. S. Kim, N. Bostanci, N. Vijaykumar, O. Ergin, and O. Mutlu, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips," arXiv:2105.08955, 2021.
[124] A. Olgun, M. Patel, A. G. Yağlıkçı, H. Luo, J. S. Kim, F. Nisa Bostancı, N. Vijaykumar, O. Ergin, and O. Mutlu, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips," in *ISCA*, 2021.
[125] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," in *IEEE Access*, 2021.
[126] L. Orosa, Y. Wang, M. Sadrosadati, J. S. Kim, M. Patel, I. Puddu, H. Luo, K. Razavi, J. Gómez-Luna, H. Hassan, N. Mansouri-Ghiasi, S. Ghose, and O. Mutlu, "CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations," in *ISCA*, 2021.
[127] L. Orosa, A. G. Yaglikci, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chipsand Implications on Future Attacks and Defenses," in *MICRO*, 2021.
[128] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[129] M. Patel, J. S. Kim, T. Shahroodi, H. Hassan, and O. Mutlu, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics," in *MICRO*, 2020.
[130] M. Patel, T. Shahroodi, A. Manglik, A. G. Yaglikci, A. Olgun, H. Luo, and O. Mutlu, "A Case for Transparent Reliability in DRAM Systems," arXiv:2204.10378, 2022.
[131] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[132] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," in *CAL*, Jan 2015.

[133] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.

[134] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," in *CAL*, 2020.

[135] RISC-V, "RISC-V GNU Compiler Toolchain," 2021. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

[136] RISC-V, "RISC-V proxy kernel," 2022. [Online]. Available: https://github.com/riscv/riscv-pk

[137] R. Ronen, A. Eliahu, O. Leitersdorf, N. Peled, K. Korgaonkar, A. Chattopadhyay, B. Perach, and S. Kvatinsky, "The Bitlet Model: A Parameterized Analytical Model to Compare PIM and CPU Systems," in *J. Emerg. Technol. Comput. Syst.*, 2022.

[138] SAFARI Research Group, "Ramulator: A DRAM Simulator – GitHub Repository," https://github.com/CMU-SAFARI/ramulator/, 2015.

[139] SAFARI Research Group, "DAMOV – GitHub Repository," https://github.com/CMU-SAFARI/DAMOV, 2021.

[140] SAFARI Research Group, "Ramulator-PIM: A Processing-in-Memory Simulation Framework – GitHub Repository," 2021. [Online]. Available: https://github.com/CMU-SAFARI/ramulator-pim

[141] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA*, 2013.

[142] S. Saroiu, A. Wolman, and L. Cojocar, "The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses," in *IRPS*, 2022.

[143] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2016.

[144] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988, 2016.

[145] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

[146] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.

[147] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," in *CAL*, 2015.

[148] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[149] V. Seshadri and O. Mutlu, "The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR," arXiv:1610.09603, 2016.

[150] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers, Volume 106*, 2017.

[151] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822, 2020.

[152] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.

[153] G. Singh, J. Gomez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stujik, O. Mutlu, and H. Corporaal, "NAPEL: Near-memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.

[154] Standard Performance Evaluation Corp., "SPEC CPU 2006," 2006. [Online]. Available: http://www.spec.org/cpu2006

[155] B. S. B. Talukder, V. Menon, B. Ray, T. Neal, and M. Rahman, "Towards the Avoidance of Counterfeit Memory: Identifying the DRAM Origin," in *HOST*, 2020.

[156] E. Testa, M. Soeken, O. Zografos, L. Amaru, P. Raghavan, R. Lauwereins, P.-E. Gaillardon, and G. De Micheli, "Inversion Optimization in Majority-Inverter Graphs," in *NANOARCH*, 2016.

[157] TTi, "PL & PL-P Series DC Power Supplies Data Sheet - Issue 5," https://resources.aimtti.com/datasheets/AIM-PL+PL-P_series_DC_power_supplies_data_sheet-Iss5.pdf, 2022.

[158] UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator," 2018.

[159] A. van de Goor and I. Schanstra, "Address and data scrambling: Causes and impact on memory tests," in *IEEE International Workshop on Electronic Design, Test and Applications*, 2002.

[160] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, and O. Mutlu, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.

[161] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual," 2021. [Online]. Available: https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf

[162] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.

[163] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Fast Boolean Logic Mapped on Memristor Crossbar," in *ICCD*, 2015.

[164] Xilinx, *7 Series FPGAs Memory Interface Solutions*, March 2011.

[165] Xilinx, *Vivado Design Suite: Using Constraints*, November 2021.

[166] Xilinx, "Xilinx Ultrascale+ MPSoC," 2021. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[167] Xilinx, "Xilinx Zynq-7000 SoC ZC706 Evaluation Kit," 2021. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html

[168] X. Xin, Y. Zhang, and J. Yang, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *HPCA*, 2020.

[169] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "PIMSim: A Flexible and Detailed Processing-in-Memory Simulator," in *IEEE CAL*, 2019.

[170] C. Yu, S. Liu, and S. Khan, "MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator," in *IEEE CAL*, 2021.

[171] J. Yu, H. A. D. Nguyen, L. Xie, M. Taouil, and S. Hamdioui, "Memristive Devices for Computation-in-Memory," in *DATE*, 2018.

[172] Y. Zha, E. Nowak, and J. Li, "Liquid Silicon: A Nonvolatile Fully Programmable Processing-In-Memory Processor with Monolithically Integrated ReRAM for Big Data/Machine Learning Applications," in *Symposium on VLSI Circuits*, 2019.

[173] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

[174] J. Zhang, Y. Zha, N. Beckwith, B. Liu, and J. Li, "MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory," in *TRETS*, 2020.

[175] L. Zhang and L. Shen, "PIM-HBMSim: A Processing in Memory Simulator Based on High Bandwidth Memory," in *CICA*, 2022.

[176] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.

[177] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.

[178] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-based Graph Processing," in *MICRO*, 2019.