PiDRAM: An FPGA-based Framework for End-to-end Evaluation of Processing-in-DRAM Techniques

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

§ETH Zürich †TOBB University of Economics and Technology

1. Background
DRAM-based main memory is used in nearly all computing systems as a major component. Modern memory-intensive workloads have increasing memory bandwidth, latency, and capacity requirements. However, DRAM vendors often prioritize memory capacity scaling over latency and bandwidth [1–4]. As a result, main memory is an increasingly worsening bottleneck in computing systems [3,5–9].

One way of overcoming the main memory bottleneck is to move computation near memory, a paradigm known as processing-in-memory (PiM) [6,7,10–150]. 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 [3,5,9].

PiM techniques provide significant performance benefits and energy savings by exploiting the high internal bit-level parallelism of DRAM for different arithmetic [50,52,71,73–78,151] and data movement operations [49,51,68,69]. Recent works [72,75,76,151,152] show that some of these PiM techniques can already be supported in contemporary, off-the-shelf DRAM chips with modifications only to the memory controller. Given that DRAM is a dominant memory technology, such commodity DRAM based PiM techniques provide a promising way to improve the performance and energy efficiency of existing and future systems at no additional DRAM hardware cost.

2. Motivation
Integration of PiM techniques in a real system imposes non-trivial challenges that require further research to find appropriate solutions. For example, in-DRAM copy and initialization techniques [51,72] that copy data in bulk (e.g., one or multiple memory pages) require modifications related to memory management that affect various different parts of the system. First, in-DRAM copy and initialization techniques have specific memory allocation and alignment requirements (e.g., page-granularity source and destination operands should be allocated and aligned at row granularity within the same DRAM subarray). These requirements are not satisfied by existing system-level memory allocation primitives (e.g., malloc [153], posix_memalign [154]). Second, the source operands (usually one or more pages) of a copy operation must be up to date in DRAM (i.e., it should not have dirty copies in CPU caches). Existing cache coherence management operations (e.g., the x86 CLFLUSH instruction [155]) cannot efficiently evict page-granularity source operands from the caches [85,156], as cache coherence management operations need to query the coherence states of all cache blocks of the source operands (i.e., even the coherence states of the cache blocks that are not present in CPU caches are queried) [156].

2.1. Limitations of the State-of-the-Art
System integration challenges of PiM techniques 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 [157]), or system simulators (e.g., gem5 [158,159], Ramulator [160,161], Ramulator-PIM [162], zsim [163]). First, many commodity DRAM based PiM techniques rely on non-standard DDRx operation, where manufacturer-recommended timing parameters for DDRx commands are violated [72,75,76,151,152] (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 [1,76,157,164,165], which is required to integrate these PiM techniques into real systems. Second, prior works [72,75,76] show that the reliability of commodity DRAM based PiM techniques is highly dependent on environmental conditions such as temperature and voltage fluctuations. These effects are exacerbated by the non-standard behavior of PiM techniques in real DRAM chips. Although special purpose testing platforms (e.g., SoftMC [157]) can be used to conduct reliability studies, these platforms do not model an end-to-end computing system, where system integration of PiM techniques can be studied. Third, system simulators (e.g., gem5 [158,159], Ramulator [160,161], Ramulator-PIM [162], zsim [163]) can model end-to-end computing systems. However, they (i) do not model DRAM operation beyond 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 PiM techniques are integrated into real systems (e.g., proprietary and chip-specific DRAM internal address mapping [166,167]), and (iii) cannot support studies on the reliability of PiM techniques since system simulators do not model environmental conditions.

1Commodity DRAM based PiM techniques are PiM techniques that can already be supported in existing off-the-shelf DRAM chips with hardware modifications only to the memory controller.

2We are especially interested in PiM techniques that do not require any modification to the DRAM chips or the DRAM interface.
3. PiDRAM Framework

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

PiDRAM facilitates system integration studies of commodity DRAM based PiM techniques by providing a set of four customizable hardware and software components that can be used as a common basis to enable system support for such techniques in real systems. Figure 1 presents an overview of PiDRAM’s components.

3.1. 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 PiM techniques. The PiM Operations Controller (POC). POC decodes and executes PiDRAM instructions (e.g., RowClone-Copy [51]) that are used by the programmer to perform PiM 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 way, we improve the portability of the framework and facilitate porting it to systems that employ different instruction set architectures. Second, POC communicates with the memory controller to perform PiM operations in the DRAM chip over a simple hardware interface. To do so, POC (i) requests the memory controller to perform a PiM operation, (ii) waits until the memory controller performs the operation, and (iii) receives the result of the PiM 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.

PiDRAM Memory Controller. PiDRAM’s memory controller provides an easy-to-extend basis for commodity DRAM based PiM techniques that require issuing DRAM commands with violated timing parameters [72, 75, 76, 151, 152]. The memory controller is designed modularly and requires easy-to-make modifications to its scheduler to implement new PiM techniques. For instance, our modular design enables supporting RowClone [51] operations using only 60 additional lines of Verilog code on top of the baseline custom memory controller’s scheduler that implements conventional DRAM operations [2, 167–176] (e.g., activate, precharge, read, write).

3.2. 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 PiM framework.

Extensible Software Library (pimolib). The extensible library (PiM operations library) allows system designers to implement software support for PiM techniques. Pimolib contains customizable functions that interface with the POC to perform PiM operations in real unmodified DRAM chips. A typical function in pimolib performs a PiM operation in four steps: It (i) writes a PiDRAM instruction to the POC’s instruction register, (ii) sets the Start flag in the POC’s flag register, (iii) waits for the POC to set the Ack flag in the POC’s flag register, and (iv) reads the result of the PiM operation from the POC’s data register (e.g., the true random number after performing an in-DRAM true random number generation operation, Section 4.2).

Custom Supervisor Software. The supervisor software contains the necessary OS primitives (e.g., memory management, allocation, and alignment specialized for RowClone [51]). This facilitates developing end-to-end integration of PiM 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 4.2) that can satisfy the memory allocation constraints of RowClone [51,177].

3.3. General PiDRAM Execution Workflow

We describe the general workflow for a PiDRAM operation (e.g., RowClone-Copy [51], random number generation using DRaNGe [76]) in Figure 2 over an example copy() function that is called by the user to perform a RowClone-Copy operation in DRAM.

Figure 1: PiDRAM overview. Modified hardware (in green) and software (in blue) components. Unmodified components are in gray. A pimolib function executes load and store instructions in the CPU to perform PiM operations (in red). We use yellow to highlight the key hardware structures that are controlled by the user to perform PiM operations.
The user makes a system call to the custom supervisor software ① that in turn calls the copy(source, destination) function in the pilolib ②. The function executes two store instructions in the RISC-V core ③. The first store instruction updates the instruction register with the 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 the POC. When the Start flag is set, the 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 PiM 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 2) in the flag register to logic-1 ⑧, indicating the end of execution for the last PiM 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 ⑨. 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 PiM operation (in this example, RowClone-Copy). ⑩

Figure 2: Workflow for a PiDRAM RowClone-Copy operation.

The key idea of the memory management mechanism is to allocate memory management that satisfies the memory allocation and alignment requirements of RowClone [51, 177]. To support RowClone, we (i) customize PiDRAM’s memory controller to issue carefully-engineered sequences of DRAM commands that perform copy operations in DRAM, and (ii) extend the custom supervisor software to implement a new memory management mechanism that satisfies the memory allocation and alignment requirements of RowClone [51, 177].

We demonstrate a prototype of PiDRAM on an FPGA-based platform (Xilinx ZC706 [178]) that implements an open-source RISC-V system (Rocket Chip [179]). Our custom supervisor software extends the RISC-V PK [180] to support the necessary OS primitives on PiDRAM’s prototype. We perform in-DRAM operations in real unmodified DDR3 DRAM chips that are connected to the FPGA-based platform. Figure 3 shows our FPGA prototype.

Figure 3: PiDRAM’s FPGA prototype.

4. FPGA Prototype & Case Studies

PiDRAM allows (i) interfacing with and performing in-DRAM operations on real unmodified DRAM chips, (ii) observing the effects of environmental conditions on PiM techniques, and (iii) conducting end-to-end system-level studies on PiM techniques using real DRAM chips. We demonstrate the versatility and ease-of-use of PiDRAM by prototyping it on an FPGA board and conducting two case studies of PiM computation.

4.1. FPGA Prototype

To demonstrate the flexibility and ease of use of PiDRAM, we implement two PiM techniques on unmodified DRAM chips by violating timing parameters: (1) RowClone [51], an in-DRAM copy and initialization mechanism (using command sequences proposed by ComputeDRAM [72]), and (2) D-RaNGe [76], an in-DRAM true random number generator based on DRAM activation-latency failures induced by violation of manufacturer-recommended DRAM timing parameters.

To support RowClone, we (i) customize PiDRAM’s memory controller to issue carefully-engineered sequences of DRAM commands that perform copy operations in DRAM, and (ii) extend the custom supervisor software to implement a new memory management mechanism that satisfies the memory allocation and alignment requirements of RowClone [51, 177]. The key idea of the memory management mechanism is to allocate physical addresses in the same DRAM subarray [49–51, 167] for the source and destination operands (i.e., rows) of the copy operation. We develop a new methodology that checks if RowClone operations on randomly-generated physical addresses succeed or fail, to find physical row addresses that are in the same DRAM subarray. We explain this methodology and our techniques in more detail in our arXiv paper [177].
To support D-RaNGe, we (i) make simple modifications to the PiDRAM memory controller to implement a new storage structure that buffers random numbers (i.e., a random number buffer) and (ii) extend the software library with functions that retrieve data from the random number buffer and supply it to the user program.

5. Key Results & Contributions

Our evaluation of RowClone shows that an end-to-end implementation of RowClone achieves (i) $118.5 \times$ speedup for copy and $88.7 \times$ speedup for initialization operations, assuming data in DRAM is up to date and the source operand is not cached in CPU caches (i.e., no cache coherence management operations are required) and (ii) $14.6 \times$ speedup for copy and $12.6 \times$ initialization operations assuming data in DRAM is up to date but the source operand has cached copies in CPU caches (i.e., a cache coherence management operation must be performed for every cache block of the source operand) over CPU-based copy (i.e., conventional \texttt{memcpy} [181]) and initialization (i.e., conventional \texttt{calloc} [182]) operations.

We make two observations from our evaluation of D-RaNGe. First, end-to-end integration of D-RaNGe into a real system provides true random numbers at low latency. Our implementation can generate a 4-bit random number in 220 ns. Second, an end-to-end integration of D-RaNGe into a real system generates true random numbers at high throughput. Our implementation achieves 8.30 Mb/s sustained throughput. PiDRAM’s D-RaNGe implementation can be optimized to generate random numbers more frequently (i.e., at higher throughput). We leave such optimizations to PiDRAM’s D-RaNGe implementation for future work.

Over the Verilog and C++ codebase provided by PiDRAM, integrating RowClone end-to-end in the full system takes 198 lines of Verilog and 565 lines of C++ code and integrating D-RaNGe end-to-end in the full system takes 190 lines of Verilog and 78 lines of C++ code.

Our contributions are as follows:

- We develop PiDRAM, the first flexible, end-to-end, and open source PiM technique (D-RaNGe [76]). Our implementation provides a solid foundation for future work on system integration of DRAM-based PiM security primitives (e.g., PUFs [75,152], TRNGs [76,151,152]), implemented using real unmodified DRAM chips.
- PiDRAM is open sourced and freely available on Github: https://github.com/CMU-SAFARI/PiDRAM.

6. Summary

We introduce the first flexible, end-to-end, and open source in-DRAM computation framework, called PiDRAM. Unlike existing evaluation frameworks and prior work, PiDRAM allows us to conduct end-to-end full system studies of commodity DRAM based PiM techniques that use real unmodified DRAM chips. We demonstrate PiDRAM’s versatility and ease of use by prototyping it on an FPGA board and conducting case studies of two state-of-the-art PiM techniques for in-DRAM data copy & initialization (RowClone [51]) and in-DRAM true random number generation (D-RaNGe [76]). We show that an end-to-end implementation of (i) RowClone greatly improves data copy and initialization performance at the full system level, and (ii) D-RaNGe provides true random numbers to applications at high throughput. We hope and believe that researchers and industry will benefit from and build on PiDRAM to better evaluate PiM techniques and their benefits in real systems.

Acknowledgements

We thank the SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners, including Google, Huawei, Intel, Microsoft, and VMware. We acknowledge support from the Semiconductor Research Corporation and the ETH Future Computing Laboratory.

This invited extended abstract is a summary version of our prior work [177]. A presentation that describes this work can be found at [183].

References

[1] K. K. Chang et al., “Understanding Latency Variation in Modern DRAM Chips: Experimenetal Characterization, Analysis, and Optimization,” in SIGMETRICS, 2016.
[2] D. Lee et al., “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” in HPCA, 2013.
[3] O. Mutlu et al., “A Modern Primer on Processing in Memory,” Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann, 2021.
[4] M. Patel et al., “A Case for Transparent Reliability in DRAM Systems,” arXiv:2204.10378, 2022.
[5] S. Ghose et al., “Processing-in-Memory: A Workload-driven Perspective,” IBM JRD, 2019.
[6] O. Mutlu, “Memory Scaling: A Systems Architecture Perspective,” IMF, 2013.
[7] O. Mutlu et al., “Research Problems and Opportunities in Memory Systems,” SURFACES, 2014.
[8] O. Mutlu, “Main Memory Scaling: Challenges and Solution Directions,” 2015, Invited Book Chapter in More than Moore Technologies for Next Generation Computer Design, pp. 127-153, Springer.
[9] O. Mutlu et al., “Processing Data Where It Makes Sense: Enabling In-Memory Computation,” Micron, 2019.
[10] I. Fernandez et al., “NATS: A Near-Data Processing Accelerator for Time Series Analysis,” in ICCD, 2020.
[11] D. S. Cali et al., “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis,” in MICRO, 2020.
[12] J. S. Kim et al., “GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies,” BMC Genomics, 2018.
