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

Published in TACO 2023

Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, Onur Mutlu

Seminar in Computer Architecture, Spring 2023
ETH Zürich
Cristina Morad
Executive Summary

Motivation: Commodity DRAM based PuM techniques improve the performance and energy efficiency of computing systems at no additional DRAM hardware cost.

Problem: Challenges of integrating these PuM techniques into real systems are not solved. General-purpose computing systems, special-purpose testing platforms, and system simulators cannot be used to efficiently study system integration challenges.

Goal: Design and implement a flexible framework that can be used to:
- solve system integration challenges
- analyze trade-offs of end-to-end implementations of commodity DRAM-based-PuM techniques

Key idea: PiDRAM, an FPGA-based framework that enables:
- system integration studies
- end-to-end evaluations of PuM techniques using real unmodified DRAM chips

Evaluation: End-to-end integration of one PuM technique on PiDRAM’s FPGA prototype.

Case Study #1 – RowClone: In-DRAM bulk data copy operations
- 119x speedup for copy operations compared to CPU-copy with system support
- 198 lines of Verilog and 565 lines of C++ code over PiDRAM’s flexible codebase
Outline

Background

- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM

- Overview
- Hardware & Software Components
- Execution of a PuM Operation
- FPGA Prototype

Case Studies

- Case Study #1 – RowClone

Conclusion
DRAM Organization
DRAM Operation

- Wordline Drivers
- Cache line
- READ
- READ
- READ

**DRAM Command Sequence**

- **tRAS** (Activation Latency)
- **tRP** (Precharge Latency)
- **tRCD** (Access Latency)

- **ACT R0**
- **RD**
- **RD**
- **RD**
- **PRE R0**
- **ACT R1**
- **RD**
- **RD**
- **RD**

**time**
Prior works show that tRCD, tRAS and tRP can be violated.
Prior works show that tRCD, tRAS and tRP can be violated. 

**Benefit from violating time parameters:**

**Reduced Access Latency, Parallelism, Locality Exploitation**
Outline

Background
- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM
- Overview
- Hardware & Software Components
- Execution of a PuM Operation
- FPGA Prototype

Case Studies
- Case Study #1 – RowClone

Conclusion
PuM Techniques (I)

• Take advantage of operational principles of memory to perform **bulk data movement and computation in memory**
  • Can **exploit internal connectivity** to move data
  • Can **exploit analog computation capability**
Prior works show a variety of in-DRAM computation mechanism (i.e., PuM techniques).
PuM Techniques (III)

Prior works show a variety of in-DRAM computation mechanism (i.e., PuM techniques)

Great potential to improve system performance & energy efficiency

Provide low-cost security primitives

PreLatPUF: Exploiting DRAM Latency Variations for Generating Robust Device Signatures

D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput

PLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation

RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data

Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM

- Vivek Seshadri
- Yoongu Kim
- Chris Fallin
- Donghyuk Lee
- Rachata Ausavarungnirun
- Gennady Pekhimenko
- Yixin Luo
- Boon-Yew Ong

- Alser
- Ivetra

- Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, Todd C. Mowry

1 Microsoft Research 2 NVIDIA Research 3 TOBB University of Economics and Technology 4 Carnegie Mellon University 5 Intel Pittsburgh 6 ETH Zurich

- Jerome S. Kim
- Minesh Patel
- Hasan Hassan
- Lois Orosa
- Onur Mutlu

1 Carnegie Mellon University 2 ETH Zurich 3 TOBB University of Economics and Technology 4 The University of Toronto
Row-Copy: Key Idea

1. Source row to sense amplifiers
2. Sense amplifiers to destination row
RowClone in Real DRAM Chips

Key Idea: Use carefully created DRAM command sequences

- ACT → PRE → ACT command sequence with greatly reduced DRAM timing parameters
- ComputeDRAM [Gao+, MICRO’19] demonstrates in-DRAM copy operations in real DDR3 chips
Background

- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM

- Overview
  - Hardware & Software Components
  - Execution of a PuM Operation
  - FPGA Prototype

Case Studies

- Case Study #1 – RowClone

Conclusion
# System Support for PuM

<table>
<thead>
<tr>
<th>Application</th>
<th>bulk data initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program/Language</td>
<td>supervisor for basic system support</td>
</tr>
<tr>
<td>System Software</td>
<td>software interface to execute PuM ops.</td>
</tr>
<tr>
<td>SW/HW Interface</td>
<td>control logic for PuM operations</td>
</tr>
<tr>
<td>Micro-architecture</td>
<td>support for custom timing parameters</td>
</tr>
<tr>
<td>Logic</td>
<td></td>
</tr>
<tr>
<td>Devices</td>
<td></td>
</tr>
<tr>
<td>Electrons</td>
<td></td>
</tr>
</tbody>
</table>

**Micro-architecture SW/HW Interface:**
- **Application**
- **Program/Language**
- **System Software**
- **SW/HW Interface**
- **Micro-architecture**
- **Logic**
- **Devices**
- **Electrons**

**DRAM Chip**

- **Row Buffer**
PiDRAM

Bridge the “system gap” with customizable HW/SW components
in doing so, allow users to
rapidly implement PuM techniques, solve system integration challenges, analyze end-to-end implementations

bulk data initialization

- supervisor for basic system support
- software interface to execute PuM ops.
- control logic for PuM operations
- support for custom timing parameters

DRAM Chip
PiDRAM: Key Components

- **Control PuM operation**
  - PuM Operations Controller
  - PuM Operations Library
  - Interface for Applications

- **Memory controller for custom timing parameters**
  - PiDRAM Memory Controller
  - Custom Supervisor Software
  - Supervisor software for basic system support
Outline

Background

  DRAM Organization and Operation
  Commodity DRAM Based PuM Techniques

PiDRAM

  Overview

  **Hardware & Software Components**
  Execution of a PuM Operation
  FPGA Prototype

Case Studies

  Case Study #1 – RowClone

Conclusion
PuM Operations Controller - POC

Decode & execute PiDRAM instructions (e.g., in-DRAM copy)

Receive instructions over memory-mapped interface (portable to other systems with different CPU ISAs)

Simple interface to the PiDRAM memory controller
(i) send request, (ii) wait until completion, (iii) read results
Custom Memory Controller

Perform PuM operations by *violating* DRAM timing parameters

- Support conventional memory operations (e.g., LOAD/STORE)
- One state machine per operation (e.g., LOAD/STORE, in-DRAM copy)

- Controls the physical DDR3 interface
- Receives commands from command scheduler & operates DDR3 pins

Easily replicate a state machine to implement a new operation
PuM Operations Library - pumolib

Contains customizable functions that interface with POC to perform PuM operations in real unmodified DRAM chips

Executes LOAD & STORE requests to communicate with the POC.
PuM Operations Library - pumolib

Contains customizable functions that interface with POC to perform PuM operations in real *unmodified* DRAM chips.

Executes LOAD & STORE requests to communicate with the POC.

---

**Table 1: Pumolib functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_timings</td>
<td>RowClone_T1, RowClone_T2, tRCD</td>
<td>Updates CRF registers with the timing parameters used in RowClone (T1 and T2) and D-RaNGe (tRCD) operations.</td>
</tr>
<tr>
<td>rng_configure</td>
<td>period, address, bit_offsets</td>
<td>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.</td>
</tr>
<tr>
<td>copy_row</td>
<td>source_address, destination_address</td>
<td>Performs a RowClone-Copy operation in DRAM from the source_address to the destination_address.</td>
</tr>
<tr>
<td>activation_failure</td>
<td>address</td>
<td>Induces an activation failure in a DRAM location pointed by the address.</td>
</tr>
<tr>
<td>buf_size</td>
<td>-</td>
<td>Returns the number of random words in the random number buffer.</td>
</tr>
<tr>
<td>rand_dram</td>
<td>-</td>
<td>Returns 32 bits (i.e., random words) from the random number buffer.</td>
</tr>
</tbody>
</table>
Custom Supervisor Software - CSS

Exposes PuM operations to the user application via system calls

Implements the necessary OS primitives (i.e., *virtual memory management, memory allocation* and *alignment*) for end-to-end implementation of PuM techniques

Implements the necessary functions and data structures in the CSS
Outline

Background
  DRAM Organization and Operation
  Commodity DRAM Based PuM Techniques

PiDRAM
  Overview
  Hardware & Software Components
  **Execution of a PuM Operation**
  FPGA Prototype

Case Studies
  Case Study #1 – RowClone

Conclusion
Execution of a PuM Operation

copy_row() function called by the user to perform a RowClone-Copy operation in DRAM

1. Application makes a system call: `copy_row(source, destination)`

2. Custom Supervisor Software calls the `copy_row(source, destination)` pumolib function

```
copy_row(S, D)
```

- **S**: source DRAM row
- **D**: destination DRAM row
Execution of a PuM Operation

copy_row() function called by the user to perform a RowClone-Copy operation in DRAM

3. copy_row(S, D) executes two store instructions in the CPU

4. The first store updates the instruction register with copy_row(S, D)

5. The second store sets the “Start” flag in the flag register

Start (S)  Start the execution of PuM operation
Execution of a PuM Operation

`copy_row()` function called by the user to perform a RowClone-Copy operation in DRAM

6. POC instructs the memory controller to perform RowClone

7. POC resets the “Start” flag, and sets the “Ack” flag

8. PiDRAM memory controller issues commands with violated timing parameters to the DDR3 module
**Execution of a PuM Operation**

`copy_row()` function called by the user to perform a `RowClone-Copy` operation in DRAM

1. The memory controller sets the “Fin.” flag

2. `copy_row(S, D)` periodically checks either “Ack” or “Fin.” flags using LOAD instructions

3. `copy_row(S, D)` returns when the periodically checked flag is set
Execution of a PuM Operation

Data Register is not used in RowClone operations because the result is stored *in memory*

It is used to read true random numbers generated by D-RaNGe
Outline

Background
- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM
- Overview
- Hardware & Software Components
- Execution of a PuM Operation

FPGA Prototype

Case Studies
- Case Study #1 – RowClone

Conclusion
PiDRAM Prototype

PiDRAM is freely available as open-source software on the GitHub repository.
Outline

Background
- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM
- Overview
- Hardware & Software Components
- Execution of a PuM Operation
- FPGA Prototype

Case Studies
- Case Study #1 – RowClone

Conclusion
RowClone has data mapping and alignment requirements that cannot be satisfied by current memory allocation mechanisms.

1-) **Alignment**: Operands must be placed at the same offset to their respective DRAM rows.

2-) **Granularity**: Operands must occupy whole DRAM rows.
RowClone has data mapping and alignment requirements that cannot be satisfied by current memory allocation mechanisms.

<table>
<thead>
<tr>
<th>Source 1</th>
<th>Target 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 3</td>
<td>Target 3</td>
</tr>
</tbody>
</table>

**3-) Mapping:** Operands must be placed within the same subarray.
RowClone has data mapping and alignment requirements that cannot be satisfied by current memory allocation mechanisms.

(4) Satisfies all three requirements
### Observation

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Address Space</td>
<td>Physical Address Space</td>
</tr>
</tbody>
</table>

#### Idea:
If we can control VA ➔ PA and we know the PA ➔ DRAM mappings, we can implement a function.

OS has **full control** over VA ➔ PA translation

No control over PA ➔ DRAM address mapping
void* array = alloc_align(int size, int id);

- Distribute the array over multiple banks while occupying rows as fully as possible
- Fallback to malloc(); for remaining data
- If size is too large such that it exceeds the size of available physical memory, alloc_align fails and causes an exception.
A = alloc_align(16*1024, 0);

B = alloc_align(16*1024, 0);

Array A

Virtual Addresses: 0x0000 0x1000 0x2000

Array B

Virtual Addresses: 0x7000

Bank 0 Bank 1 Bank 2

Row 0 Row 1
Memory Coherence

RowClone operates on data in DRAM
Up-to-date data may be in caches → coherency

Implement CLFLUSH in RISC-V rocket

Pros:
• **Realistic**, supported in contemporary architectures
• Reads and writes can hit in the cache. Flush cache lines prior to in-DRAM operations

Cons:
• Instruction overhead: One instruction per cache block
  The use of CLFLUSH involves flushing cache blocks to DRAM, which can incur additional latency and memory access overhead. This may result in a performance impact, especially if frequent cache flushes are required.
Evaluation: Methodology

Test two configurations:

1. **Bare-Metal**: No address translation
2. **No Flush**: OS support, and assume data is always up-to-date in DRAM (NO CLFLUSH)

Table 3: PiDRAM system configuration

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

- Set tRAS from 37.5ns to 10ns
- Set tRP from 13.5ns to 10ns

Available on ZC706 board: pipelined & in-order
Microbenchmark consist of two programs that take argument $N$:

1. \textit{copy}: copies $N$-byte array to another $N$-byte array
2. \textit{init}: initialize $N$-byte array to all zeros

Each program has two versions:

1. \textbf{CPU-copy version}: Copies or initializes data using \underline{memory loads and stores}, without utilizing Row-Clone operations.
2. \textbf{RowClone version}: Utilizes Row-Clone operations to perform the copy or initialization process.
Observation 1: Significant improvement over traditional CPU-copy and CPU-initialization.
1. rcc throughput improvement range: 317.5× to 364.8×.
2. rci throughput improvement range: 172.4× to 182.4×.

Observation 2: Increasing array size leads to increased throughput improvement provided by rcc and rci.
1. Saturation point: The increase in throughput improvement saturates when the array size reaches 1 MiB.
Observation 1:
1. rcc improves copy throughput by 58.3× for 8 KiB arrays and by 118.5× for 8 MiB arrays.
2. rci improves initialization throughput by 31.4× for 8 KiB arrays and by 88.7× for 8 MiB arrays.

Observation 2:
1. The execution time of rcc and rci operations does not increase linearly with the array size.
Observation 1: Assuming 50% are dirty cache block
• rcc provide 3.2x higher throughput over CPU-copy
• rci provide 3.9x higher throughput over CPU-initialization

Observation 2:
As the fraction of dirty cache blocks increases, the throughput improvement provided by rcc and rci decreases.
CLFLUSH operations are **inefficient** in supporting coherence for RowClone operations.

Expect throughput benefits **increase** as coherence between CPU cache and PIM accelerators **improve**.
Outline

Background
  DRAM Organization and Operation
  Commodity DRAM Based PuM Techniques

PiDRAM
  Overview
  Hardware & Software Components
  Execution of a PuM Operation
  FPGA Prototype

Case Studies
  Case Study #1 – RowClone

Conclusion
Conclusion

**Motivation:** Commodity DRAM based PuM techniques improve the performance and energy efficiency of computing systems at no additional DRAM hardware cost.

**Problem:** Challenges of integrating these PuM techniques into real systems are not solved. General-purpose computing systems, special-purpose testing platforms, and system simulators *cannot* be used to efficiently study system integration challenges.

**Goal:** Design and implement a flexible framework that can be used to:
  * solve system integration challenges
  * analyze trade-offs of end-to-end implementations of commodity DRAM-based-PuM techniques

**Key idea:** PiDRAM, an FPGA-based framework that enables:
  * system integration studies
  * end-to-end evaluations of PuM techniques using real unmodified DRAM chips

**Evaluation:** End-to-end integration of one PuM technique on PiDRAM’s FPGA prototype

**Case Study #1 – RowClone:** In-DRAM bulk data copy operations
  * 119x speedup for copy operations compared to CPU-copy with system support
  * 198 lines of Verilog and 565 lines of C++ code over PiDRAM’s flexible codebase
Discussion
**Strength**

- **Flexible** - Offers a high-level extensible DRAM controller framework for other PuM researchers
- Provides a hardware prototype for experimentation and includes system software support for illustrating the capabilities of the prototype.
- As PiDRAM is made available to the research community as open-source software, users can help develop the framework and research new PuM techniques
- **Well-written**
Strength

- **Flexible** - Offers a high-level extensible DRAM controller framework for other PuM researchers
- Provides a hardware prototype for experimentation and includes system software support for illustrating the capabilities of the prototype.
- As software, users can help develop the framework and research new PuM techniques.
- Well-written

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

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

• Memory allocation mechanism limits the amount of data you can allocate in the same subarray
• The infrastructure is not so easy to extend to different systems and memory technologies
• Lack of information about the underlying structure of IRT (Initializer Rows Table)
Discussion 1 – PiM vs PuM (I)

**Processing using Memory (PuM):**
- enable operations within main memory with minimal changes.

**Processing near memory (PNM):**
- involves adding or integrating PIM logic close to or inside the memory

[Diagram: AMBIT]

[Diagram: PiM]

[Diagram: GRIM-Filter]

Discussion 1 – PiM vs Pum (II)

Where would you place PiDRAM?

**PuM:**
- enable operations within main memory with minimal changes.

**PiM**

**PNM:**
- involves adding or integrating PIM logic close to or inside the memory

“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”
To enable `alloc_align()`, we maintain the **SubArray Mapping Table (SAMT)** that contains a list of physical page addresses that point to DRAM rows that are in the same DRAM subarray.

**SAMT Entry**

```
<table>
<thead>
<tr>
<th># Free Rows</th>
<th>Pair 0</th>
<th>Pair 1</th>
<th>...</th>
<th>Pair M</th>
</tr>
</thead>
</table>
```

1. **Retrieve a physical address pointing to a DRAM row in subarray 0**
2. **Update the page table to map programmer-allocated address to subarray 0**
To keep track of different operands that are allocated by `alloc_align()` using the same ID, we use the Allocation ID Table (AIT).
so far...

DRAM

BANK X

R0
R1
R2

SA W

R3
R4
R5

SA Z

BANK Y

R6
R7
R8

SAMT

BANK X – SA W

BANK X – SA Z

BANK Y – SA W

BANK Y – SA Z
### SAMT

<table>
<thead>
<tr>
<th>#free row</th>
<th>List of Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX-SA W</td>
<td>P R0 R1 R2</td>
</tr>
<tr>
<td>BX-SA Z</td>
<td>P R3 R4 R5</td>
</tr>
<tr>
<td>BY-SA W</td>
<td>P R6 R7 R8</td>
</tr>
<tr>
<td>BY-SA Z</td>
<td>null R9 R10 R11</td>
</tr>
</tbody>
</table>

### DRAM

- **BANK X**
  - SA W
    - R0
    - R1
    - R2
  - SA Z
    - R3
    - R4
    - R5

- **BANK Y**
  - SA W
    - R6
    - R7
    - R8
  - SA Z
    - R9
    - R10
    - R11
so far...

1. \( A = \text{alloc_allign}(8*1024, 0) \);
2. \( B = \text{alloc_allign}(8*1024, 0) \);

<table>
<thead>
<tr>
<th>ID 0</th>
<th>AIT</th>
<th>SAMTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BY-SA W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BX-SA W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY-SA W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID 1</th>
<th>AIT</th>
<th>SAMTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BY-SA W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BX-SA Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY-SA W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

### SAMT

<table>
<thead>
<tr>
<th>#free row</th>
<th>List of Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R0, R1, R2</td>
</tr>
<tr>
<td>P</td>
<td>R3, R4, R5</td>
</tr>
<tr>
<td>P</td>
<td>R6, R7, R8</td>
</tr>
<tr>
<td>null</td>
<td>R9, R10, R11</td>
</tr>
</tbody>
</table>

### DRAM

- **BANK X**
  - SA W: R0, R1, R2
  - SA Z: R3, R4, R5

- **BANK Y**
  - SA W: R7, R8
  - SA Z: R9, R10, R11

---

2. \( B = \text{alloc_allign}(8*1024, 0) \);
Discussion-2 (I)

- Assume that a subarray can store 32 KiB
- We want to allocate two arrays each with size 32 KiB; i.e., a total of 64 KiB
- **Problem**: what if we need to allocate more data than what we can store in a subarray?

```plaintext
A = alloc_align(128*1024, 0);
B = alloc_align(128*1024, 0);
```
Discussion-2 (II)

\[ A = \text{alloc\_align}(16\times1024, 0); \]
\[ B = \text{alloc\_align}(16\times1024, 0); \]
When we want to allocate some arrays, we check if:

\[ \frac{\text{size}_{\text{subarray}}}{2} = \text{size}_{\text{array}} \]

if yes, we continue with the normal allocation; else, we divide the array in chunks until the condition is satisfied.

- AIT grows dynamically
- Different IDs can share same SAMT Entries
- Each entry can contain several SAMTE to different subarrays
In PiDRAM Prototype we have this parameters:

- The **CPU** in the rocket-chip operates at **50 MHz** (i.e., the clock has a period of 20 ns).
- The **DRAM module** is clocked at **400 MHz** (i.e., the clock has a period of 2.5 ns).

**What is the problem?**

Hint: For every CPU clock cycle, the DRAM module completes eight clock cycles!

**This is UNREALISTIC!**

Average frequency of CPU and Memory in today's system:

- **CPU**: range of 2 GHz to 4 GHz
- **Memory**: commonly use DDR4 which ranges from 2133 MHz to 3200 MHz

Basically, the evaluations done are optimistic!
How do you solve the problem?

**Fetch Stage:** Retrieves the instruction from memory using the current program counter value.

**PC Delay Stage:** Holds the program counter value for \(x\) cycle.

**Decode Stage:** Decodes the fetched instruction and prepares the control signals for subsequent stages.
Discussion-4

The infrastructure is not so easy to extend to different systems and memory technologies

- **Hardware Dependencies:**
  PiDRAM's infrastructure is tightly coupled with the specific hardware components it was designed for, such as the memory controller and FPGA boards.

- **Memory Controller Modification:**
  The design is modular and uses well-defined interfaces, but making modifications still requires a deep understanding of the memory controller's operation and careful implementation.

  Make the PiDRAM infrastructure more configurable and parameterizable.
  Including options for different memory technologies, timing parameters, and memory controller configurations.

  ! This is not a solution!
  The infrastructure would still be difficult to extend!
Talk about PiDRAM

Rocket Chip

Open-source SoC design generator

Composed of many SoC component generators

- Generator: Chisel/Scala code that builds hardware

Core

\[ \uparrow \]

L1 Cache

\[ \downarrow \]

Core

\[ \uparrow \]

L1 Cache

\[ \downarrow \]

Core Parameters

- Enable VM
- Use ISA Extensions
- Generate mul/div
- Generate FPU

Outputs synthesizable Verilog RTL
Talk about PiDRAM

**PiDRAM Workflow**

1. User application interfaces with the OS via system calls
2. OS uses PuM Operations Library (pumolib) to convey operation related information to the hardware using
3. STORE instructions that target the memory mapped registers of the PuM Operations Controller (POC)
4. POC oversees the execution of a PuM operation (e.g., RowClone, bulk bitwise operations)
5. Scheduler arbitrates between regular (load, store) and PuM operations and issues DRAM commands with custom timings
Talk about D-RaNGe

D-RaNGe: Access Pattern

- To maximize the bits that are accessed immediately following activation, we alternate in distinct access order.

Accessing cache lines containing more RNG cells will result in more random values.
THANK YOU

Nisa Bostancı
PhD Student
DRAM | Hardware/Software Cooperation | Heterogeneous Memory Systems

Ataberk Olgun
PhD Student
DRAM | Hardware/Software Cooperation | Hardware Security
Thank You For The Attention
Cristina Morad

Question?
Accessing a DRAM Cell

- **wordline**
- **capacitor**
- **access transistor**
- **Sense Amp**
- **enable**
- **bitline**
Accessing a DRAM Cell

1. Enable wordline
2. Connects cell to bitline
3. Cell loses charge
4. Deviation in bitline voltage
5. Enable sense amp
6. Sense Amp

\[ \frac{1}{2} V_{DD} + \delta \]

\[ V_{DD} \]

\[ \frac{1}{2} V_{DD} \]
### Table 2: Various known PuM techniques that can be studied using PiDRAM. PuM techniques we implement in this work are highlighted in bold.

<table>
<thead>
<tr>
<th>PuM Technique</th>
<th>Description</th>
<th>Integration Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputeDRAM-based [44]</td>
<td>Bulk data-copy and initialization</td>
<td>(i) memory allocation and alignment mechanisms that map source &amp; destination operands of a copy operation into same DRAM subarray; (ii) memory coherence, i.e., source operand must be up-to-date in DRAM.</td>
</tr>
<tr>
<td>RowClone [148]</td>
<td>within DRAM</td>
<td></td>
</tr>
<tr>
<td>D-RaNGe [83]</td>
<td>True random number generation using DRAM</td>
<td>(i) periodic generation of true random numbers; (ii) memory scheduling policies that minimize the interference caused by random number requests.</td>
</tr>
<tr>
<td>ComputeDRAM-based Ambit [145]</td>
<td>Bitwise operations in DRAM</td>
<td>(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.</td>
</tr>
<tr>
<td>SIMDGRAM [54]</td>
<td>Arithmetic operations in DRAM</td>
<td>(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.</td>
</tr>
<tr>
<td>DL-PUF [82]</td>
<td>Physical unclonable functions in DRAM</td>
<td>memory scheduling policies that minimize the interference caused by generating PUF responses.</td>
</tr>
<tr>
<td>QUAC-TRNG [123] and Talukder+ [13]</td>
<td>True random number generation using DRAM</td>
<td>(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.</td>
</tr>
</tbody>
</table>
In-DRAM TRNG: Key Idea (D-RaNGe)

High % chance to fail with reduced access latency

50% chance to fail

Low % chance to fail with reduced access latency

Commodity DRAM chips can already perform D-RaNGe
Rocket Chip

Open-source System-on-Chip (SoC) design generator

Composed of many SoC component generators that can be customized to create different SoC variations

Written in Chisel, a hardware design language (HDL) embedded in Scala

Rocket Chip's Chisel generators build hardware and output synthesizable Verilog RTL

Standardized interfaces allow easy component swapping without changing the hardware source code.
• DRAM-internal address mapping schemes translate from logical to physical DRAM addresses.
  • Logical addresses are used by the memory controller
  • Physical addresses are internal to the DRAM chip

• DRAM-internal address mapping schemes serve to:
  i. enable post-manufacturing row repair techniques to assign defective DRAM rows to alternative ones; and
  ii. help DRAM manufacturers organize DRAM internals in a cost-efficient and reliable manner.

• DRAM-internal address mapping can vary substantially between DRAM chips.
  • Consecutive logical DRAM row addresses may not point to physical DRAM rows in the same subarray.
RowClone Success Rate

Determine if certain row address pairs are consistently successful in the RowClone operation, indicating that are mapped to the same DRAM subarray

Observation:
- for a set of \{src, dest\} DRAM row address pairs, RowClone operations repeatedly succeed with 100% probability

Assumption:
- these pairs of DRAM row addresses are mapped to the same subarray

Identifying pairs consist of three steps:
  i. **initialize** src an dest rows with random data,
  ii. **perform RowClone** operation from src row to dest row, and
  iii. **compare** the data in the dest row with the src row.

Repeat **1000 times**, and if every iteration is successful then store pair in SAMT
RowClone Success Rate

Determine if certain row address pairs are consistently successful in the RowClone operation, indicating that are mapped to the same DRAM subarray.

- Initialize with random data
- Compare src & dest rows

\[ \text{success rate} = \frac{\# \text{ of bits that differ}}{\# \text{ of bits stored in a row}} = \begin{cases} 100\% , & \text{if no difference} \\ < 100\% , & \text{otherwise} \end{cases} \]
1. CSS uses the RowClone success rate experiment to identify DRAM rows in the same subarray and initializes the Subarray Mapping Table (SAMT).
2. The programmer utilizes the `alloc_align()` to allocate two 128 KiB arrays, A and B, with the same allocation ID (0), intending to copy data from array A to array B.
CSS allocates contiguous ranges of virtual addresses for arrays A and B, and then divides these virtual address ranges into page-sized memory blocks.
4. The CSS assigns consecutive memory blocks to consecutive DRAM banks and accesses the Allocation ID Table (AIT) using the allocation ID for each memory block.
5. By accessing the AIT, CSS retrieves the subarray ID that points to a SAMTE that corresponds to the subarray that contains the arrays allocated using the allocation ID.
6. CSS accesses the SAMTE to retrieve two physical addresses that point to the same DRAM row. CSS maps a memory block and its sibling memory block to these two physical addresses, such that they are mapped to the first and the second halves of the same DRAM row.
7. 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, ensuring that the memory blocks are mapped to the same DRAM row.
8. traverses the page table to find the physical address of each block (i.e., the address of a DRAM row) – rcc
Other Workloads

forkbench (copy-heavy workload)

- 9% execution time reduction by in-DRAM initialization
  - 17% of compile’s execution time is spent on initialization

compile (initialization-heavy workload)

SPEC2006 libquantum

- 1.3% end-to-end execution time reduction
  - 2.3% of libquantum’s time is spent on initialization
Outline

Background
- DRAM Organization and Operation
- Commodity DRAM Based PuM Techniques

PiDRAM
- Overview
- Hardware & Software Components
- Execution of a PuM Operation
- FPGA Prototype

Case Studies
- Case Study #1 – RowClone
- Case Study #2 – D-RaNGe

Conclusion
Key Idea

- A cell’s latency failure probability is inherently related to random process variation from manufacturing.
- We can extract random values by observing DRAM cells’ latency failure probabilities.

High % chance to fail with reduced $t_{RCD}$

Low % chance to fail with reduced $t_{RCD}$

The key idea is to extract random values by sampling DRAM cells that fail truly randomly.
Implementation

Identify four DRAM cells that fail randomly when accessed with a reduced $\text{t}_{\text{RCD}}$ (**RNG Cell**) in a cache block.

- When accessing an RNG Cell with a reduced $\text{t}_{\text{RCD}}$, the values read will be truly random values.

**RNG Cell**

Random values when accessed with $\text{t}_{\text{RCD}}$ reduced by 50%
1. Periodically **accesses** a DRAM cache block with reduced tRCD.
2. **Reads** four TRNG cells in the cache block.
3. **Stores** four bits read from the TRNG cells in a 1 KiB random number buffer.

Reserve register to configure:
1. **TRNG period** (in ns)
2. tRCD
3. **physical location** of TRNG cell
Pumolib function:
1. `buf_size()` : Returns the number of random words (4 bytes) available in the buffer.
2. `rand_dram()` : Returns one random word read from the buffer.

2. Functions retrieve informations needed by accessing data register with LOAD instruction

1. Update data register
   - with number of random words available
   - with a random word read from the random number buffer
Run microbenchmark that consist of a loop that:

i. Check the availability of random numbers using buf_size(), and

ii. Reads random number using rand_dram().

The TRNG throughput decreases from **8.30 Mb/s at a 220 ns** TRNG period to **1.90 Mb/s at a 1000 ns** TRNG period.