PiDRAM
A Framework for End-to-end Integration of Processing-using-memory

P&S Processing-in-Memory Tutorial
09.11.21

Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oğuz Ergin, Onur Mutlu
Executive Summary

**Motivation:** Recent works propose in-DRAM computation primitives with great potential to improve performance and energy consumption of computing systems.

**Problem:** These works are developed in limited environments (e.g., simulators, characterization platforms) where many parts of the system are ignored:
- The challenges in integrating these primitives into a system cannot be fully explored in these environments.

**Goal:** Develop a flexible platform to explore end-to-end implementations of current and future processing-in-memory (PuM) techniques.

**Key idea:** To build an FPGA-based infrastructure that supports in-DRAM operations and has system support.

SAFARI
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Accessing a DRAM Cell

- wordline
- capacitor
- access transistor
- enable
- bitline

[Seshadri+ MICRO’17]
Accessing a DRAM Cell

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

Sense Amp deviation in bitline voltage connects cell to bitline.

[Seshadri+ MICRO’17]

SAFARI
DRAM Operation

DRAM Command Sequence

\[ t_{RAS} \] (Activation Latency) \[ t_{RP} \] (Precharge Latency)

SAFARI [Kim+ HPCA'19]
• Take advantage of operational principles of memory to perform \textit{bulk data movement and computation in memory}
  • Can \textit{exploit internal connectivity} to move data
  • Can \textit{exploit analog computation capability}

• Examples: RowClone, In-DRAM AND/OR, D-RaNGe, ...
  • \textit{RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data} (Seshadri et al., MICRO 2013)
  • "\textit{Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology}” (Seshadri et al., MICRO 2017)
  • “\textit{D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput}” (Kim et al., HPCA 2019)
  • ...
**RowClone**

High Energy
(3600nJ to copy 4KB)

High latency
(1046ns to copy 4KB)
RowClone-FPM: Mechanism

1. Source row to row buffer
2. Row buffer to destination row
RowClone-FPM: Bitline Operation (I)

Data gets copied

\[ V_{DD}/2 \delta \]

\[ V_{DD}/2 + \delta \]

src 0

dst 0

S Sense Amplifier (Row Buffer)

Data gets copied

\[ V_{DD}/2 \delta \]

\[ V_{DD}/2 + \delta \]
Enable fast bulk-data copy with small overhead
In-DRAM Bitwise AND/OR

- We can support **in-DRAM COPY, ZERO, AND, OR, NOT, MAJ**
- At low cost
- Using inherent analog computation capability of DRAM
  - Idea: activating multiple rows performs computation
- 30-60X performance and energy improvement

Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry,
"Fast Bulk Bitwise AND and OR in DRAM"
In-DRAM AND/OR: Triple Row Activation

Final State
\[ AB + BC + AC \]

\[ C(A + B) + \sim C(AB) \]

Demonstrates RowClone and AND/OR in real chips

- **Violate** DRAM timing parameters: tRAS, tRP
  - Induce undefined behavior

**ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs**

Fei Gao  
feig@princeton.edu  
Department of Electrical Engineering  
Princeton University

Georgios Tziantzioulis  
georgios.tziantzioulis@princeton.edu  
Department of Electrical Engineering  
Princeton University

David Wentzlaff  
wentzlaf@princeton.edu  
Department of Electrical Engineering  
Princeton University

Figure 4: Timeline for a single bit of a column in a row copy operation. The data in $R_1$ is loaded to the bit-line, and overwrites $R_2$.

Figure 5: Logical AND in ComputeDRAM. $R_1$ is loaded with constant zero, and $R_2$ and $R_3$ store operands (0 and 1). The result ($0 = 1 \land 0$) is finally set in all three rows.
Bitline is above $V_{DD}/2$ when R2 is activated.
Bitwise AND in ComputeDRAM

- $V_{dd}/2$
- $R_3 = 00_2$
  - Operand: 1
- $R_1 = 01_2$
  - Constant: 0

T2 very short
PRE cannot close R1
R3 will appear on the address bus
ACT(R2) will activate R3 and R2

ACT(R1)  PRE  ACT(R2)

$T_1 = T_2 = 0$ idle cycle
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Rocket Chip

Open-source SoC design *generator*

Composed of many SoC component *generators*

- Generator: Chisel/Scala code that builds hardware

Outputs *synthesizable* Verilog RTL

- Enable VM
- Use ISA Extensions
- Generate muldiv
- Generate FPU
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Goal: Develop a **flexible** platform to explore end-to-end implementations of PuM techniques
- Enable rapid integration via key components
**Goal:** Develop a **flexible** platform to explore **end-to-end** implementations of PuM techniques

- Enable rapid integration via key components

**Hardware**

1. Easy-to-extend Memory Controller
2. ISA-transparent PuM Controller

**Software**

1. Extensible Software Library
2. Custom Supervisor Software
Current real PuM techniques require issuing DRAM command sequences with violated timings

- Extensible memory controller facilitates implementation of such DRAM command sequences through modular design

- New command sequences require only designing new state machines
Memory controller employs three submodules to further ease developer effort

1. **Periodic Operations Module (PerOps Module):**
   DRAM periodic refresh and bus maintenance operations

2. **Scheduler:** DRAM RD/WR operations, open-bank policy

3. **Configuration Register File (CRF):**
   Store timing information, access using LD/ST instructions from the CPU
PiDRAM: PuM Controller (I)

PuM Operations Controller (POC)

Provide ISA-transparent control for PuM operations
- Connected as a memory-mapped module
  - Hierarchically, resides within the memory controller
- Simple Interface: Offload 128-bit instructions

[Diagram showing the connection between the Rocket Chip, PuM Operations Controller, CRF, Scheduler, and Physical Interface.]
PiDRAM: PuM Controller (II)

Currently implements five instructions

- **RowClone**
  - Reserved
  - Copy: 69 68
  - Dst. Row: 64 63
  - Src. Row: 32 31
  - Cache Block: 0

- **Activation Failure**
  - Reserved
  - RLRD: 69 68
  - Unused: 64 63
  - Cache Block: 32 31

- **Read Random Number (RN)**
  - Reserved
  - RRN: 69 68
  - Unused: 64 63

- **Read RN Buffer Size**
  - Reserved
  - RRNS: 69 68
  - Unused: 64 63

- **Write to configuration regs.**
  - Reserved
  - WCR: 69 68
  - Address: 64 63
  - Data: 32 31

Reserved bits for other commands

- Instruction size is **configurable**
PiDRAM: Software Library

**Pumolib**: Expose PuM operations to the user while abstracting the hardware implementation details

```c
static inline void copy_row(char *source, char *target)
{
  volatile uint64_t *ptr = (uint64_t*) IMOC_INST_UPPER;
  uint64_t imo_op = 0x1;
  uint32_t source_row_addr = (uint32_t) source;
  uint64_t target_row_addr = (uint32_t) target;
  uint64_t inst_lower = source_row_addr |
                        (target_row_addr << 32);
  uint64_t inst_upper = imo_op << (IMO_OP_OFS);

  *ptr = inst_upper;
  *(ptr+1) = inst_lower;
  *(ptr+2) = (uint64_t) 0x1;
  while(*ptr+2 != 0x2);
}
```

- **Prepare PuM instruction**
- **Offload the instruction and block until ACK’d**
Simple, easy-to-hack OS to integrate PuM techniques end-to-end:
- Virtual memory management
- Memory allocation

OS based on RISC-V proxy kernel

Future work: Integrate pumolib into Linux
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
PiDRAM FPGA Prototype

Xilinx ZC706

Single core RISC-V CPU @ 50MHz in-order, single-issue
16KB 4-way L1 D$_1$
4KB I$_1$

Compute-Enabled DIMM
RISC-V System
• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
RowClone-FPM has memory mapping requirements

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

2-) Granularity: Operands must occupy whole DRAM rows
RowClone-FPM has memory mapping requirements

3-) Mapping: Operands must be placed within the same subarray
(4) Satisfies all three requirements
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
    • Allocation Mechanism
    • Memory Coherence
    • Evaluation

• Installing and Using PiDRAM
New memory allocation mechanism to satisfy these three requirements: `alloc_align()`

```c
void* array = alloc_align(int size);
```

Optimize physical address allocation to `array` for `size` byte large copy operations so that RowClone can be used most effectively.

### Example (Row = 8 KiB)

```c
char *A = alloc_align(A, 4096*3);
```

We can at least copy 8 KiB using RowClone.
Data Mapping (II)

OS has **full control** over VA $\rightarrow$ PA translation

**No control** over PA $\rightarrow$ DRAM address mapping

**Idea:** If we can **control** VA $\rightarrow$ PA and we **know** the PA $\rightarrow$ DRAM mappings, we can implement *alloc_align*
**Alloc_align (I)**

```c
void* array = alloc_align(int size);
```

How to lay out an array onto DRAM?

I. Distribute the array over multiple banks while occupying rows as fully as possible
II. Fallback to `malloc()` for remaining data

**Assumptions:**

(i) We know the **DDRX address mapping** in our system
   - Reverse-engineer: (i) Check for RowClone, (ii) do Rowhammer
(ii) We know which **DRAM rows** are in the same subarray
   - Characterize pairs of rows for RowClone success rate
Other versions of alloc_align can be implemented to align multiple arrays in DRAM (for RowClone-Copy)

```c
alloc_align(int size, int id);
```

`id` = operand RowClone identifier

Operands with the same id are placed into the same subarray
PiDRAM Address Mapping

Our Configuration:
SODIMM: MT8JTF12864HZ-1G6G1
# of Rows = 16K
# of Banks = 8
# of Columns = 1K
Row Size = 8 KB
Total Size = 1 GB

Physical to DRAM address mapping (configurable)
Alloc_align Example

A = alloc_align(16*1024, 0);  B = alloc_align(16*1024, 0);

Array A

16 KBs

4 KB

Virtual Addresses: 0x0000 0x1000 0x2000

Array B

16 KBs

Bank 0
Bank 1
Bank 2

Row 0
Row 1

...
SubArray Mapping Table (SAMT)
A table that collects DRAM rows that map to the same subarray under common entries as physical page address pairs

SAMT Entry

<table>
<thead>
<tr>
<th>Bank 0 - SA 0</th>
<th>Bank 0 - SA 1</th>
<th>...</th>
<th>Bank 7 - SA N</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMT Entry</td>
<td>SAMT Entry</td>
<td>...</td>
<td>SAMT Entry</td>
</tr>
</tbody>
</table>

# Free Rows | Pair 0 | Pair 1 | ... | Pair M

We assume that DRAM row $\rightarrow$ DRAM subarray mapping is arbitrary
Alloc_align - Structures (II)

Reserve one address pair in every SAMT entry for initialization

<table>
<thead>
<tr>
<th># Free Rows</th>
<th>Pair 0</th>
<th>Pair 1</th>
<th>Pair M 0000...</th>
</tr>
</thead>
</table>

SIM, Subarray Id Map
Mapping from page numbers to subarray indices

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Subarray ID (SAMT idx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN 0</td>
<td>0</td>
</tr>
<tr>
<td>PPN 1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PPN 256</td>
<td>1</td>
</tr>
</tbody>
</table>
How to use SMT

\texttt{alloc\_align(int size)}:

1. Find out how many banks to utilize
   - Spread allocation across many banks

2. Split array into 4 KB blocks
   - 4 KB == 1 page
   - \( NB = \# \text{ of 4 KB blocks in A} \)

3. For all \textit{blocks} \( A[i] \) where \( i < \frac{NB}{2} \):
   1. Pick a bank.
   2. Select a SAMT entry (SAMTE) with \( \geq 1 \) free address pairs
   3. Select one pair from SAMTE
   4. Assign physical pages in the pair to virtual pages \( A[i] \) and \( A[i+\frac{NB}{2}] \)
RowClone-Initialize (RCI)

\[
\text{rci}(\text{char* } A, \text{ int } N) \rightarrow \text{Initialize } A \text{ with } N \text{ 0s.}
\]

1. Split \( A \) into 4KB blocks

2. For all \textit{blocks} \( A[i] \) where \( i < \text{NB}/2 \):
   1. Translate from \( A[i] \) to \( PA_1 \) → use as \textit{destination} address
   2. Access \textit{SIM} with \( PA_1 \) to obtain its \textit{subarray id}
   3. Access \textit{SAMT} to get the address of the \textit{all-zero row}, \( PA_{zero} \)
   4. Perform RowClone from \( PA_{zero} \) to \( PA_1 \)
      - This will automatically copy zeros to \( A[i+\text{NB}/2] \) too

(Subarray ID Map) \rightarrow \text{SA ID} \rightarrow \text{Subarray Mapping Table} \rightarrow \text{(PA}_{\text{ZERO})\ Source row that contains all 0s}
Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Memory Coherence (I)

RowClone, AMBIT 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
Other mechanisms that can alleviate the overheads

Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"The Dirty-Block Index"
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Source Code]
Other mechanisms that can alleviate the overheads

Outline

• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
Evaluation: Methodology

Table 2: PiDRAM system configuration

| CPU: 50 MHz; in-order Rocket core [16]; 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        |

Test two configurations:

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

Workloads:

- **Microbenchmarks**: CPU-Copy, RowClone-Copy
- **SPEC2006**: libquantum
RowClone-Copy provides over 365x performance improvement over rocket CPU-Copy.
RCC and RCI (with system support) improve copy throughput by 119x and 89x, respectively.
CLFLUSH dramatically reduces the potential improvement

PiDRAM enables the study of better coherence mechanisms (e.g., DBI)
Evaluation - Summary

• **PiDRAM** can run real workloads end-to-end:
  • Replace *libquantum calloc()* with *alloc_align()* and *rci()*: 1.3% performance improvement
• **RowClone-Copy** can greatly improve bulk data copy performance with (119x) and without (365x) system support
• **RowClone** requires *efficient coherency management mechanisms* to achieve its potential copy throughput improvement
PiDRAM implements RowClone and D-RaNGe

**AMBIT, SIMDRAM**

- Memory allocation and alignment
- Memory coherence
- Transposition (SIMDRAM places data vertically)

**DLPUF, QUAC-TRNG**

- Memory scheduling policies
- Efficient post-processing integration
### Table 1: PuM techniques that can be studied using PiDRAM. PuM techniques that 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><strong>RowClone [102]</strong></td>
<td>Bulk data-copy and initialization within DRAM</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 &amp; destination operands must be up-to-date in DRAM.</td>
</tr>
<tr>
<td><strong>D-RaNGe [67]</strong></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><strong>Ambit [99]</strong></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><strong>SIMDRAM [46]</strong></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><strong>DL-PUF [66]</strong></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><strong>QUAC-TRNG [89]</strong></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>
PiDRAM enables end-to-end integration of PuM techniques via
(1) Interface with real DRAM
(2) Providing a flexible memory controller design
(3) Open source design
(4) Support for system software
• Background
  • DRAM Organization
  • Processing-using-Memory
  • Rocket Chip SoC Generator

• Overview of PiDRAM
  • Hardware & Software Components
  • Prototype

• Case Study #1 - RowClone
  • Challenges
  • Allocation Mechanism
  • Memory Coherence
  • Evaluation

• Installing and Using PiDRAM
BACKUP SLIDES
Alloc_align and RCC

2. Allocate 128 KiB A and B to same subarray
   A = alloc_align(128*1024, θ);
   B = alloc_align(128*1024, θ);

3. Arrays are split into 4KB blocks

4. Allocation ID Table
   - Bank 7
   - 0 \(\rightarrow\) SA0
   - 1 \(\rightarrow\) SA3

5. Subarray Mapping Table
   - Bank 7
   - Entry – SA0
   - Entry – SA1

6. DRAM ROW
   - Consecutive blocks are assigned to DRAM rows in different DRAM banks

7. Page Table
<table>
<thead>
<tr>
<th>Virt. Addr.</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA_{A00}</td>
<td>B0 SA0 ROW0</td>
</tr>
<tr>
<td>VA_{A01}</td>
<td>B1 SA0 ROW0</td>
</tr>
<tr>
<td>VA_{A02}</td>
<td>B2 SA0 ROW0</td>
</tr>
<tr>
<td>VA_{A16}</td>
<td>B0 SA0 ROW0</td>
</tr>
<tr>
<td>VA_{A17}</td>
<td>B1 SA0 ROW0</td>
</tr>
<tr>
<td>VA_{B00}</td>
<td>B0 SA0 ROW4</td>
</tr>
<tr>
<td>VA_{B01}</td>
<td>B1 SA0 ROW4</td>
</tr>
</tbody>
</table>

8. Copy 128 KiBs from A to B
   rcc(A, B, 128*1024);

   Access page table to find source and destination DRAM rows