ASPLOS 2023 Tutorial

Real-world Processing-in-Memory Systems for Modern Workloads

Processing-Using-Memory

Exploiting the Analog Operational Properties of Memory Components

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH Zürich

SAFARI

Sunday, March 26, 2023

Two PIM Approaches

5.2. Two Approaches: Processing Using Memory (PUM) vs. Processing Near Memory (PNM)

Many recent works take advantage of the memory technology innovations that we discuss in Section 5.1 to enable and implement PIM. We find that these works generally take one of two approaches, which are categorized in Table 1: (1) *processing using memory* or (2) *processing near memory*. We briefly describe each approach here. Sections 6 and 7 will provide example approaches and more detail for both.

Table 1: Summary of enabling technologies for the two approaches to PIM used by recent works. Adapted from [341] and extended.

Approach	Example Enabling Technologies
Processing Using Memory	SRAM DRAM Phase-change memory (PCM) Magnetic RAM (MRAM) Resistive RAM (RRAM)/memristors
Processing Near Memory	Logic layers in 3D-stacked memory Silicon interposers Logic in memory controllers Logic in memory chips (e.g., near bank) Logic in memory modules Logic near caches Logic near/in storage devices

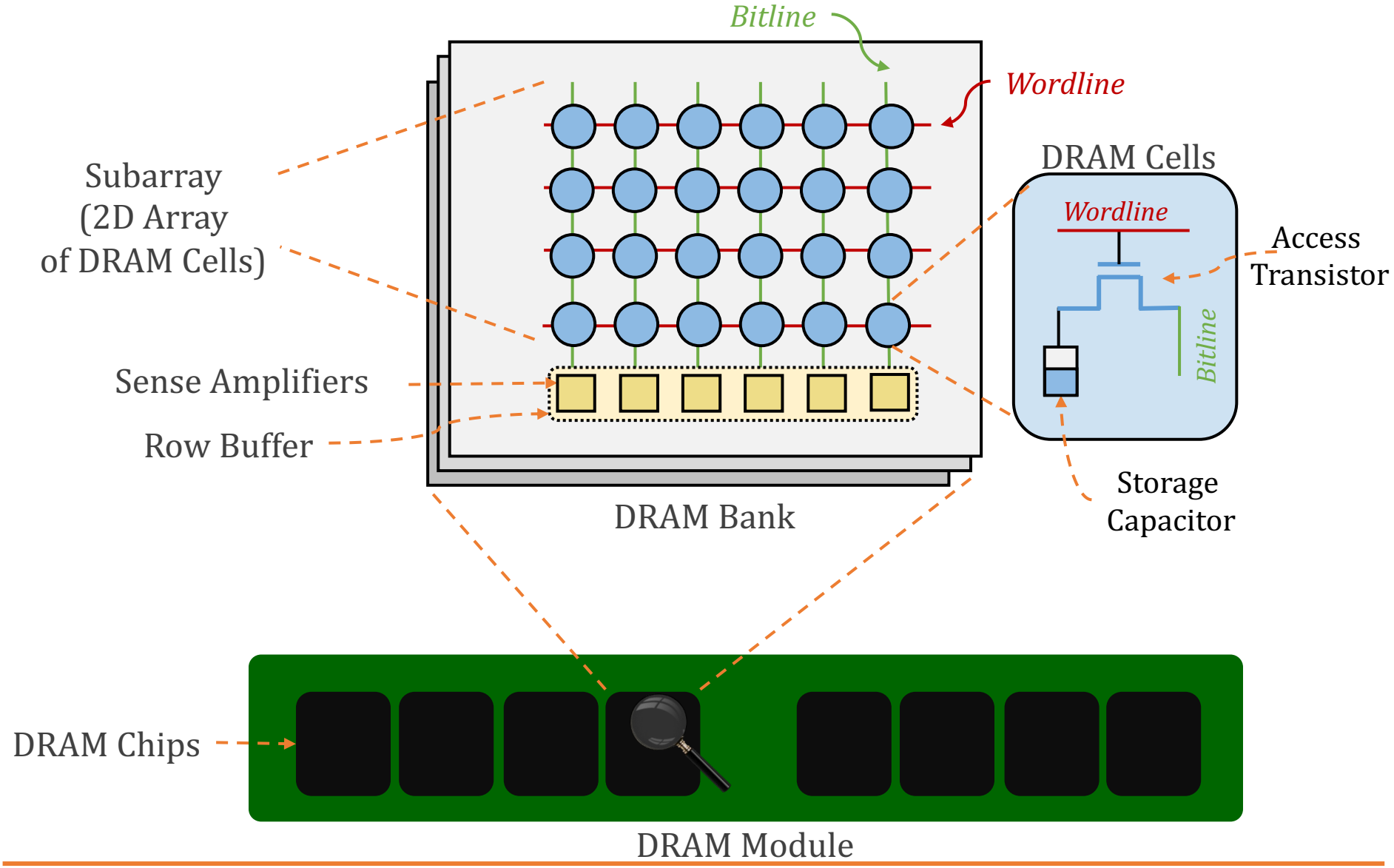
Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun,
"A Modern Primer on Processing in Memory"
Invited Book Chapter in ***Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann***, Springer, to be published in 2021.
[Tutorial Video on "Memory-Centric Computing Systems" (1 hour 51 minutes)]

Processing-Using-Memory (PUM)

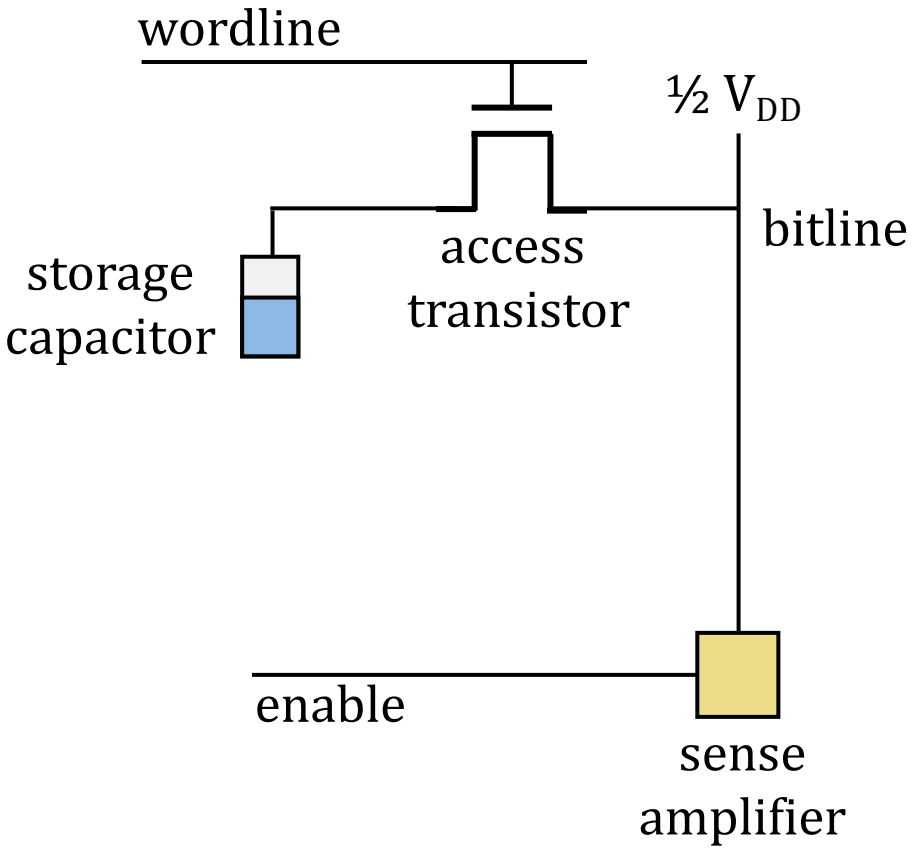
- PUM: Exploits analog operational principles of the memory circuitry to perform computation
  - Exploits internal connectivity to move data
  - Leverages the large internal bandwidth and parallelism available inside the memory arrays
- A common approach for PUM architectures is to perform bulk bitwise operations
  - Simple logical operations (e.g., AND, OR, XOR)
  - More complex operations (e.g., addition, multiplication)

DRAM Operation

Inside a DRAM Chip



DRAM Cell Operation

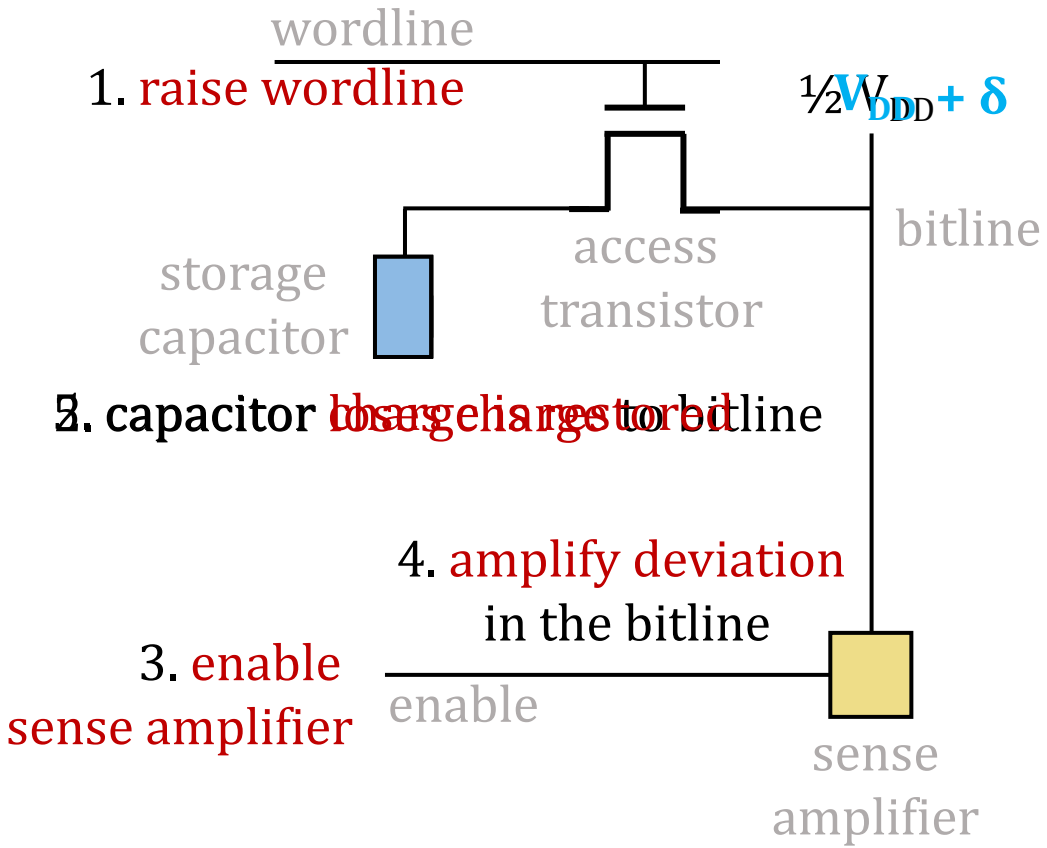


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation (I)

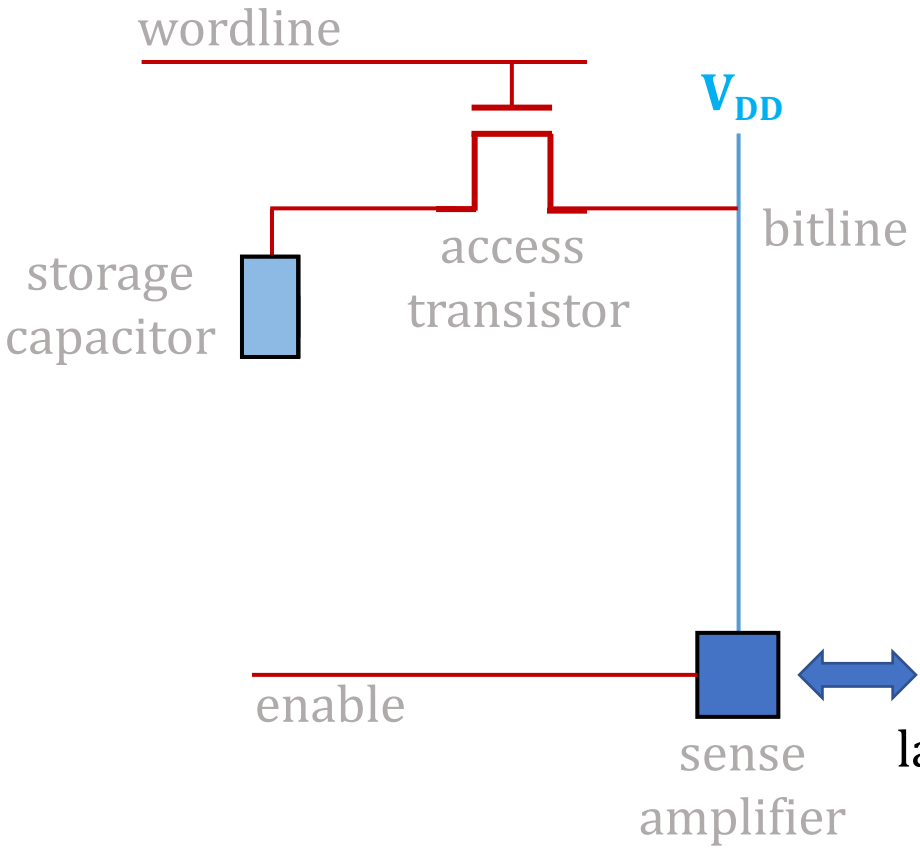


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation (II)



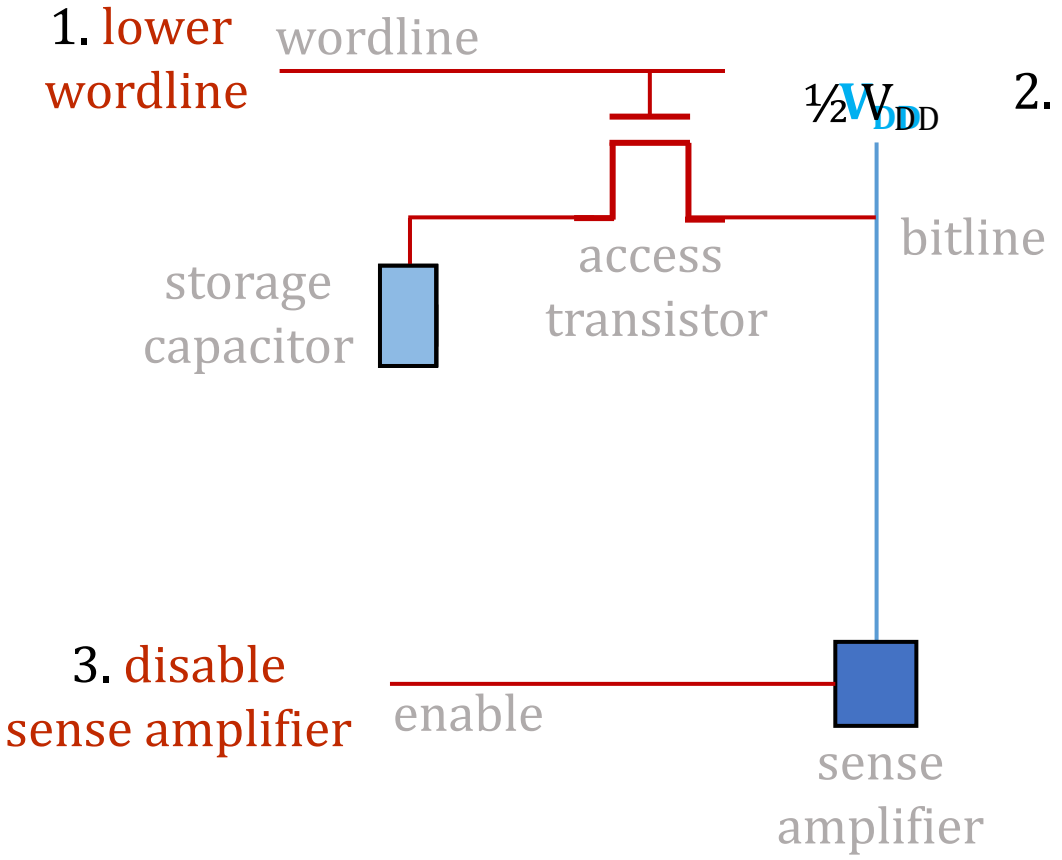
1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

read/write charge latched in sense amplifier

DRAM Cell Operation (III)



1. ACTIVATE (ACT)

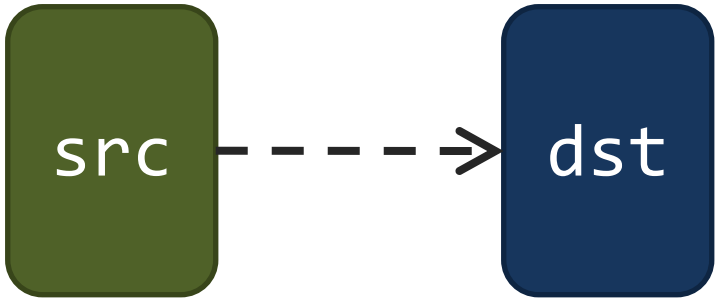
2. READ/WRITE

3. PRECHARGE (PRE)

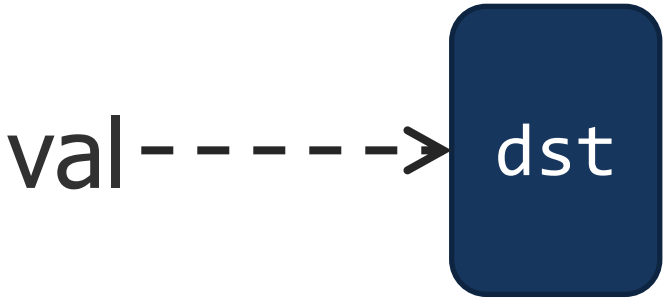
Starting Simple:
Data Copy and Initialization

Starting Simple: Data Copy and Initialization

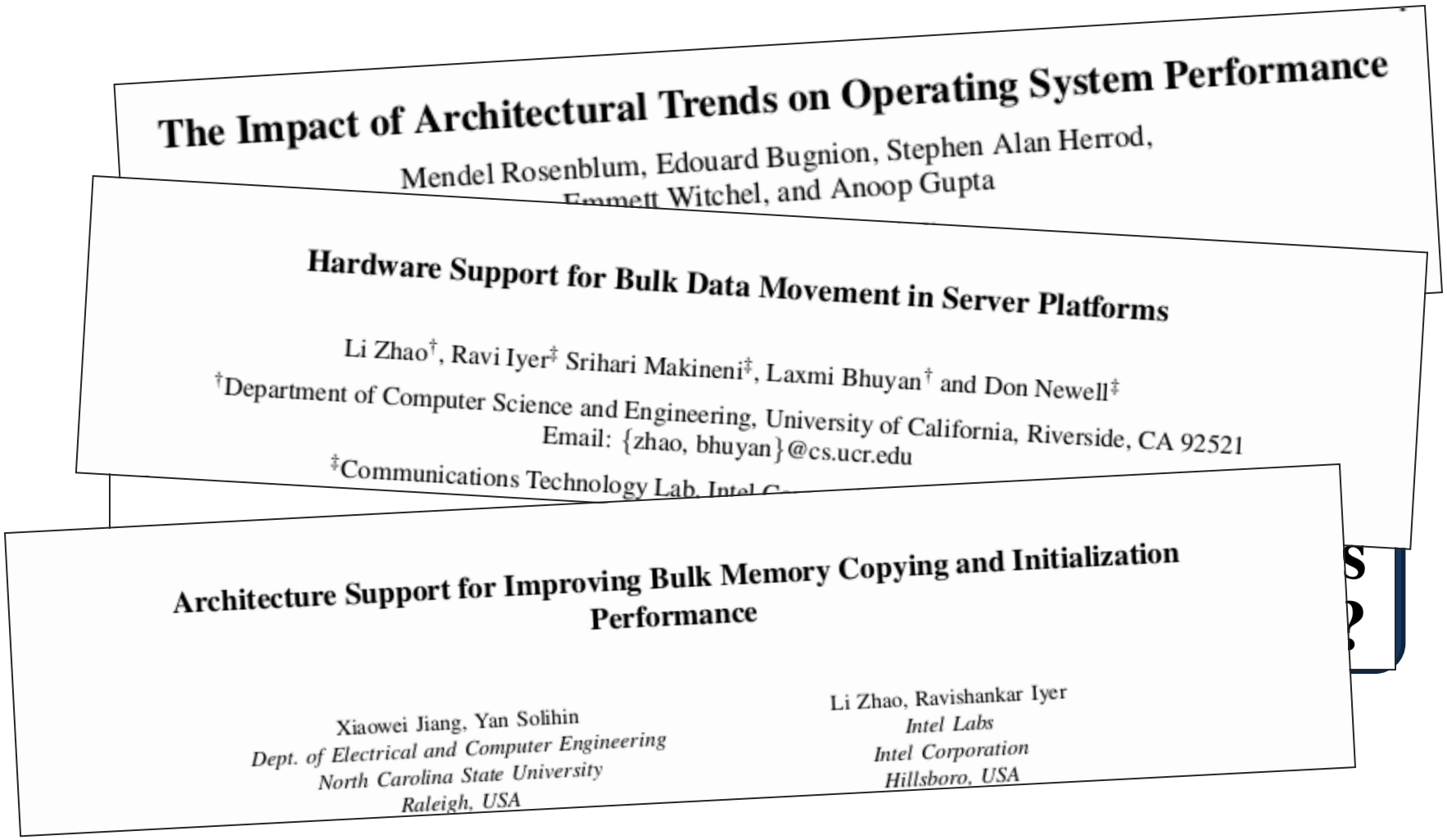
Bulk Data Copy



Bulk Data Initialization



Bulk Data Copy and Initialization



The Impact of Architectural Trends on Operating System Performance

Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod,
Emmett Witchel, and Anoop Gupta

Hardware Support for Bulk Data Movement in Server Platforms

Li Zhao†, Ravi Iyer‡ Srihari Makineni‡, Laxmi Bhuyan† and Don Newell‡

†Department of Computer Science and Engineering, University of California, Riverside, CA 92521
Email: {zhao, bhuyan}@cs.ucr.edu

‡Communications Technology Lab, Intel C...

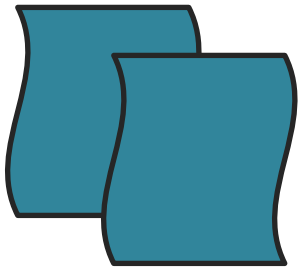
Architecture Support for Improving Bulk Memory Copying and Initialization Performance

Xiaowei Jiang, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA

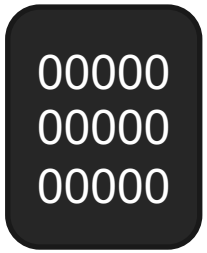
Li Zhao, Ravishankar Iyer
Intel Labs
Intel Corporation
Hillsboro, USA

Starting Simple: Data Copy and Initialization

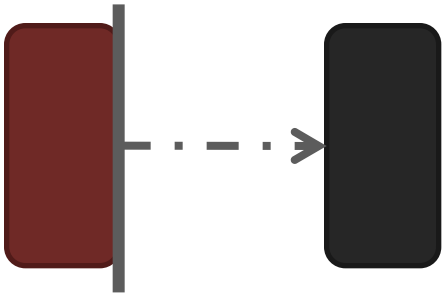
memmove & memcpy: 5% cycles in Google's datacenter [Kanev+ ISCA'15]



Forking

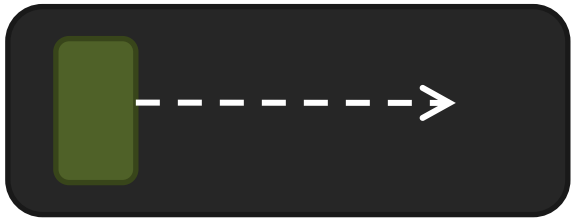


Zero initialization (e.g., security)



Checkpointing

VM Cloning Deduplication

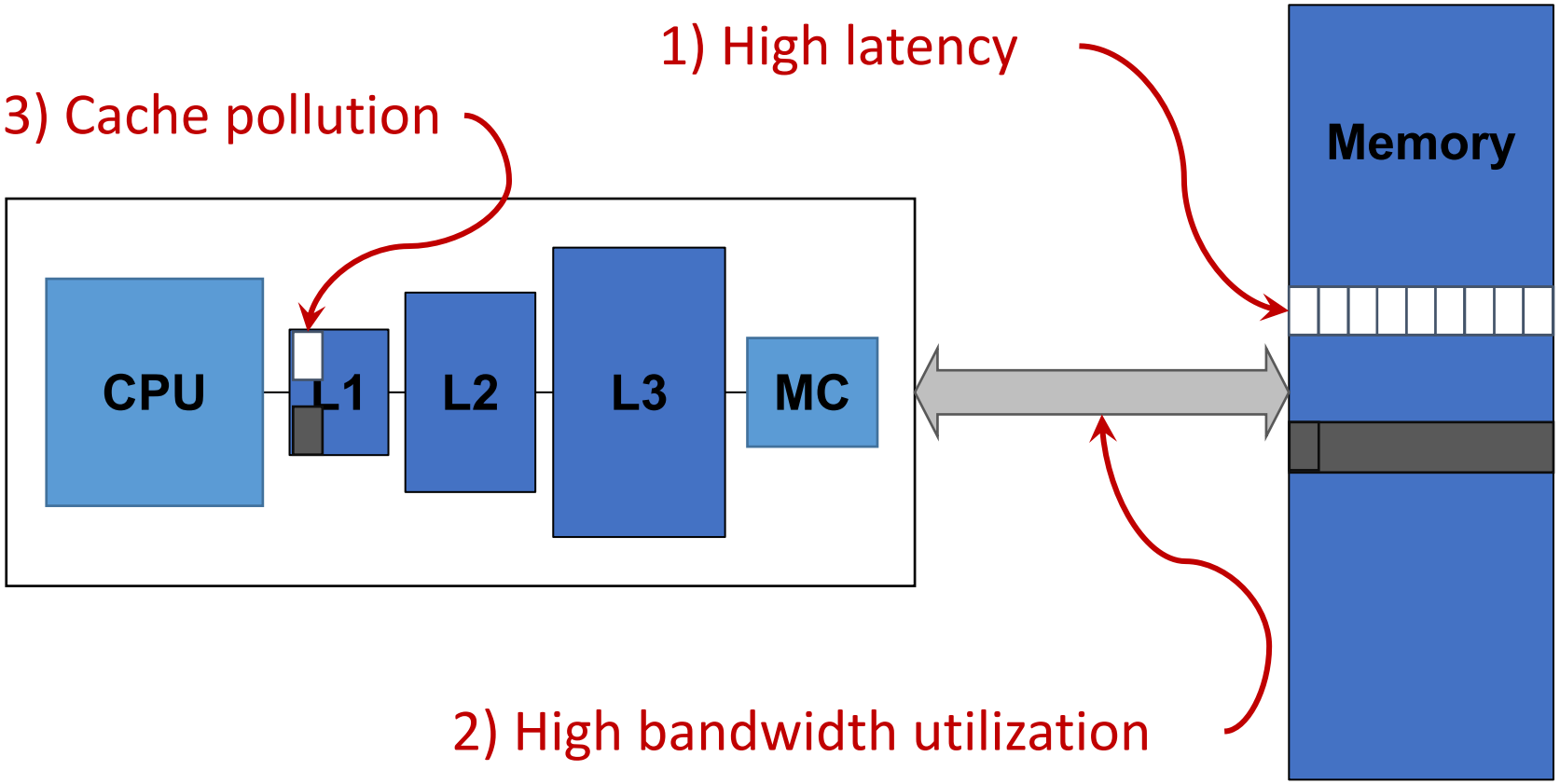


Page Migration

• • •

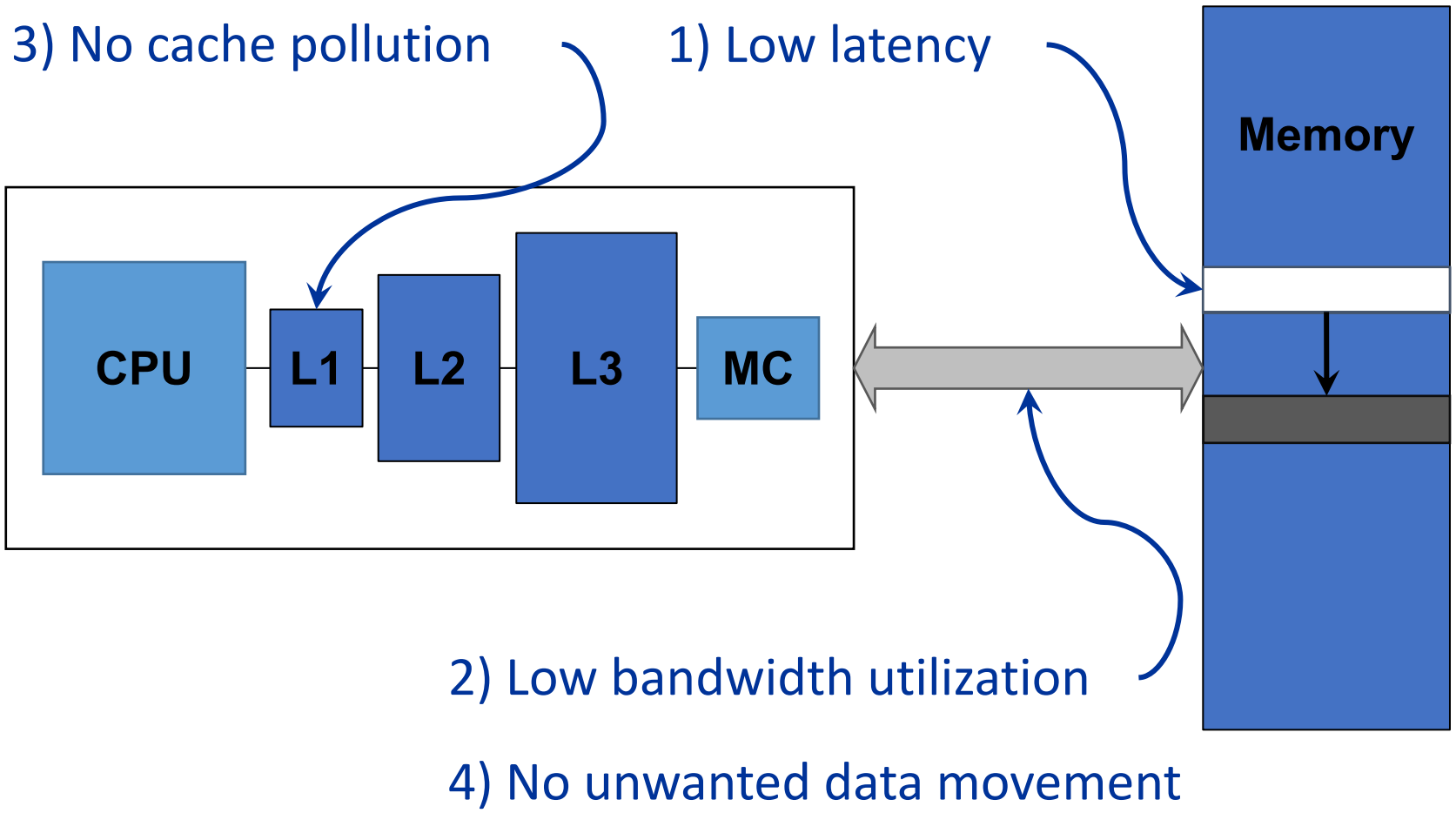
Many more

Today's Systems: Bulk Data Copy



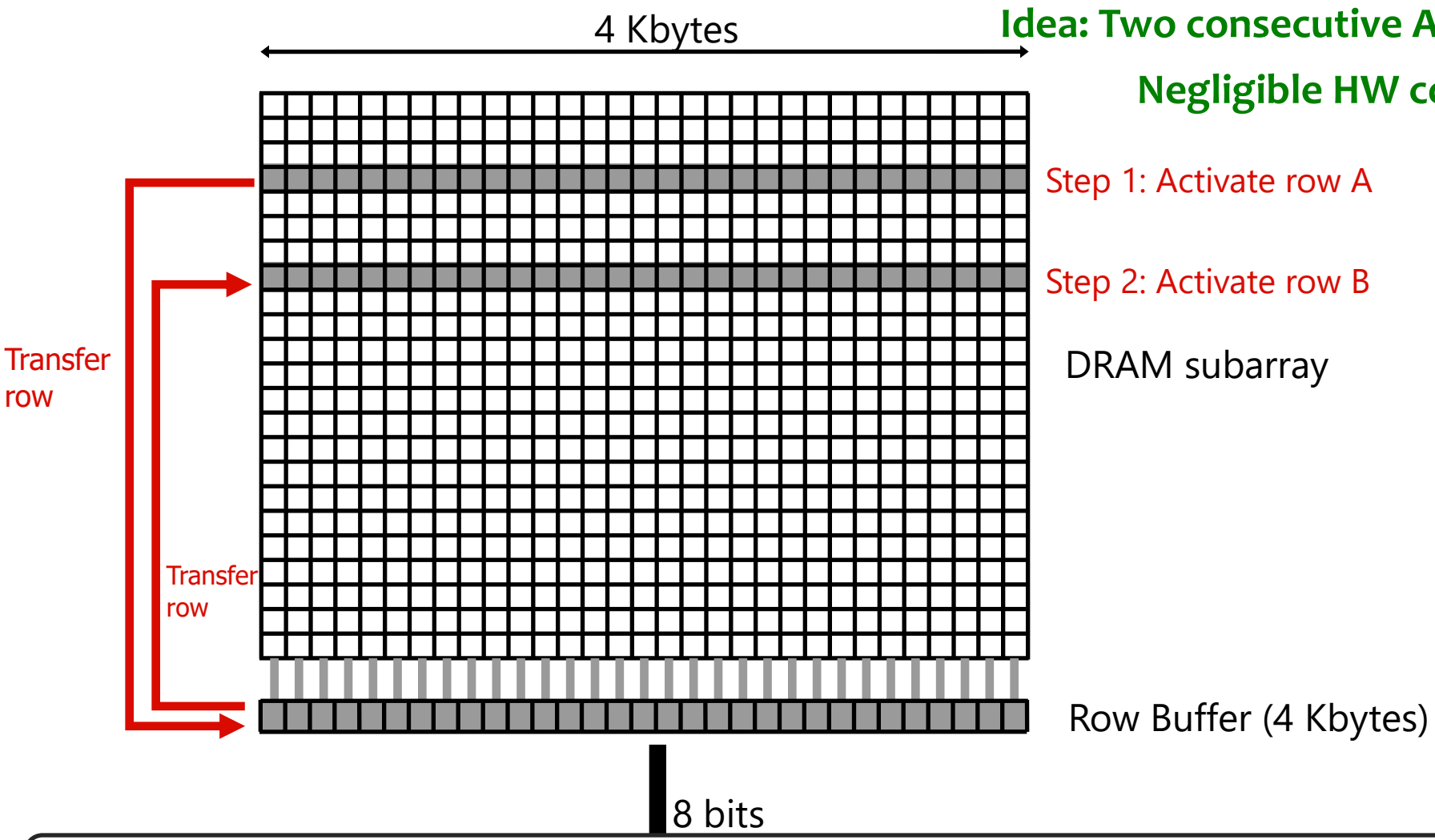
1046ns, 3.6uJ (for 4KB page copy via DMA)

Future Systems: In-Memory Copy



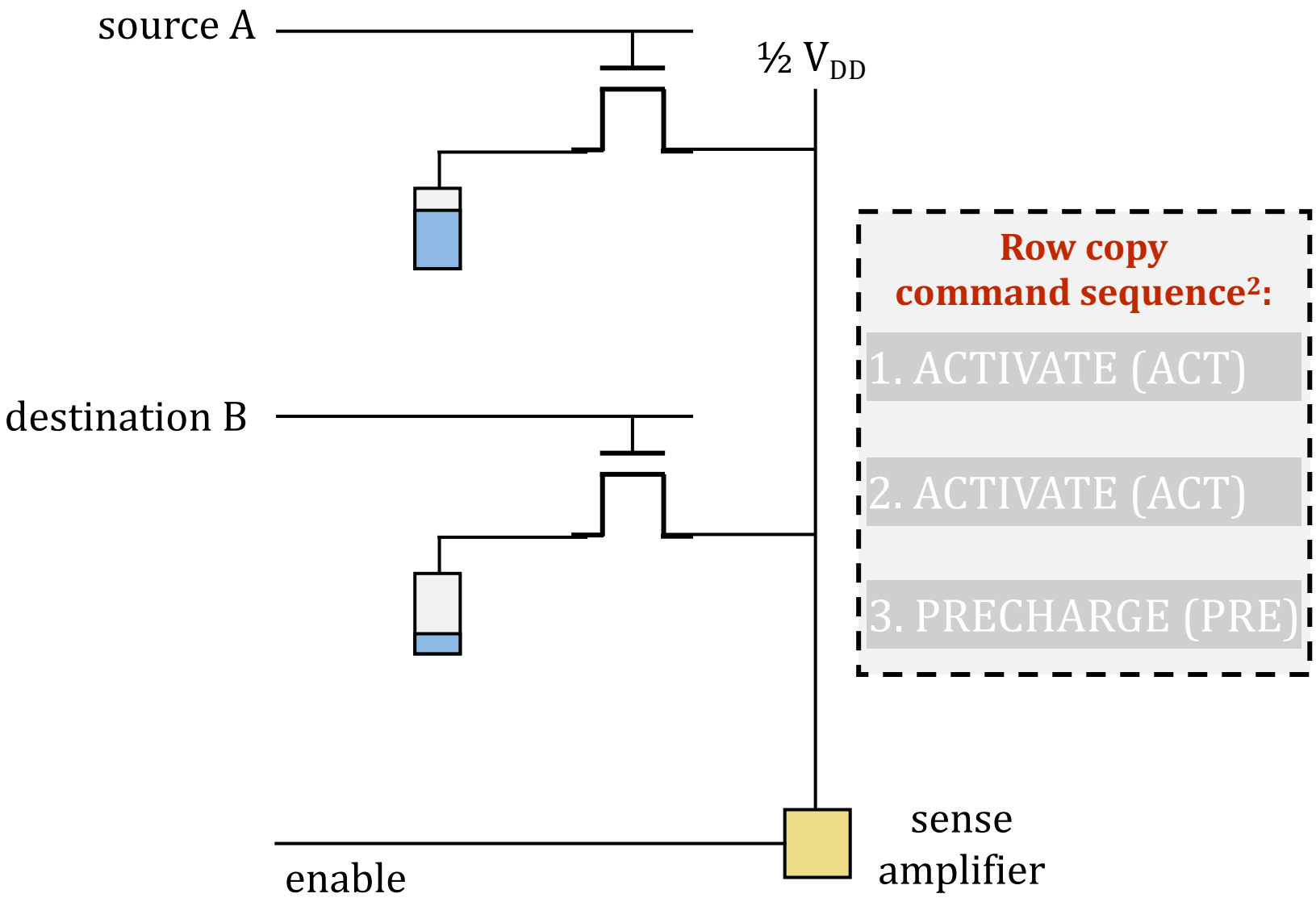
1046ns, 3.6uJ → 90ns, 0.04uJ

RowClone: In-DRAM Row Copy

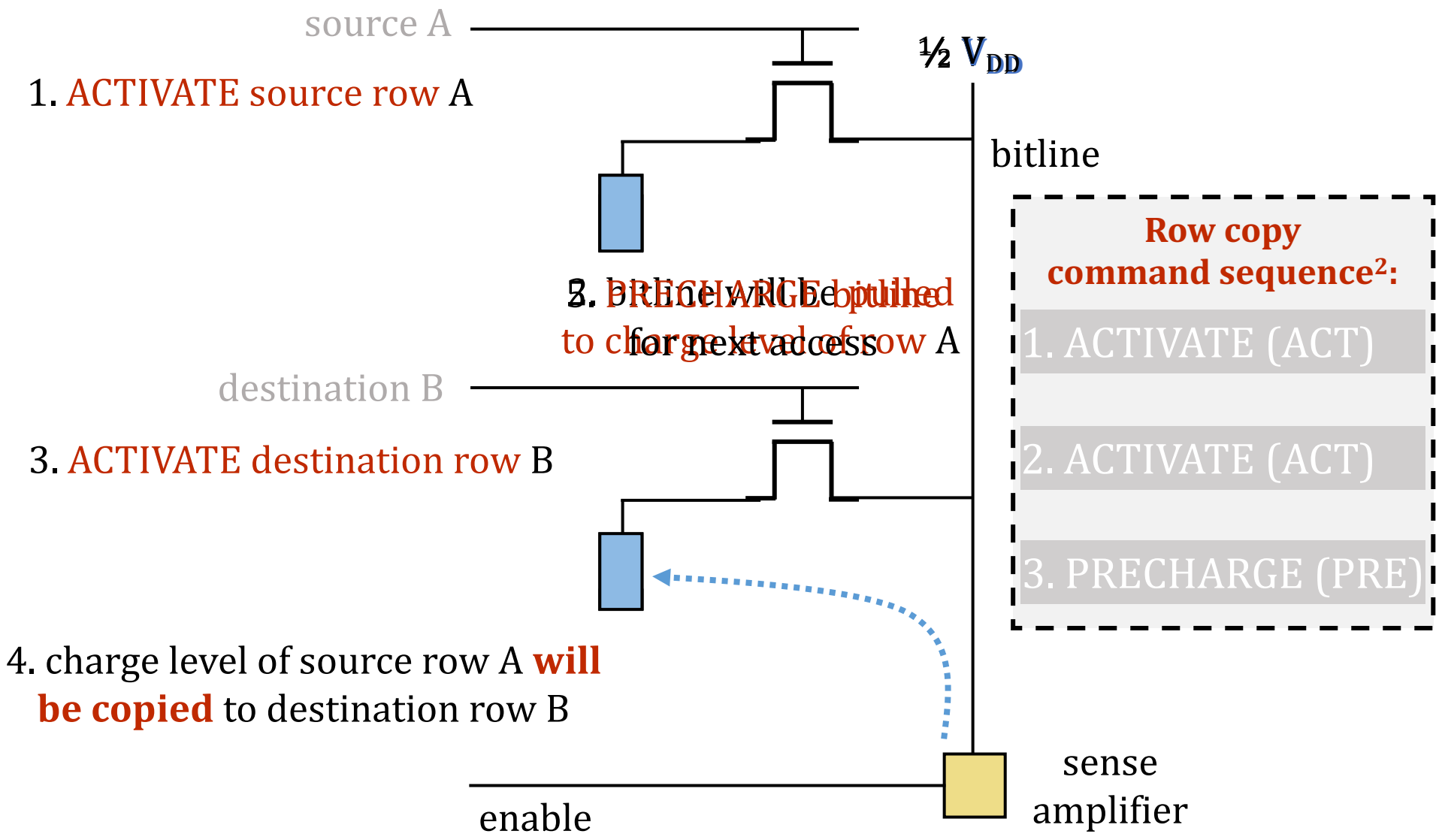


11.6X latency reduction, 74X energy reduction

RowClone: In-DRAM Row Copy (I)

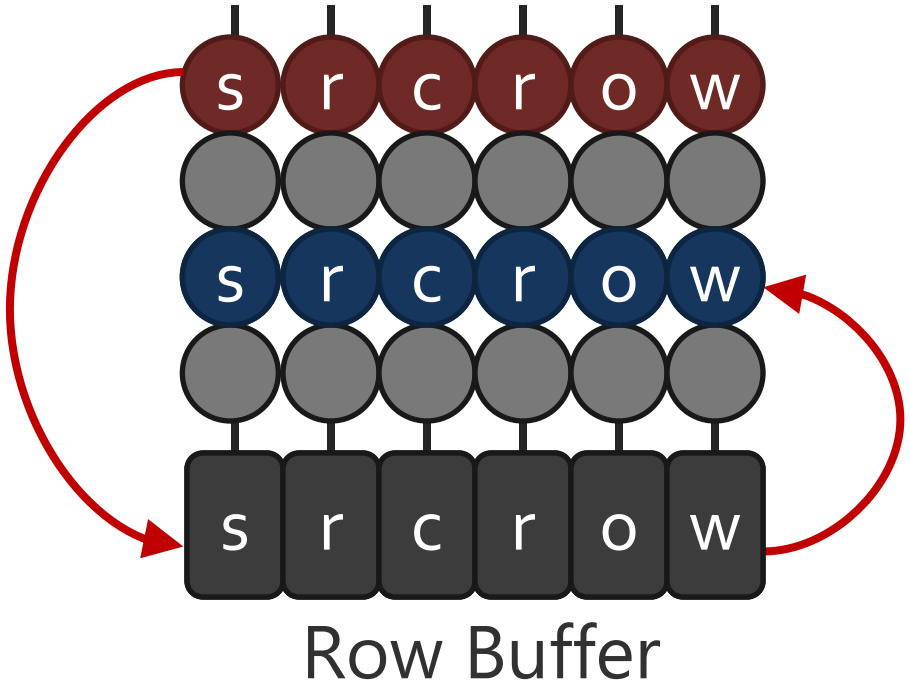


RowClone: In-DRAM Row Copy (II)



[2] V. Seshadri et al., *"RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"*, MICRO, 2013

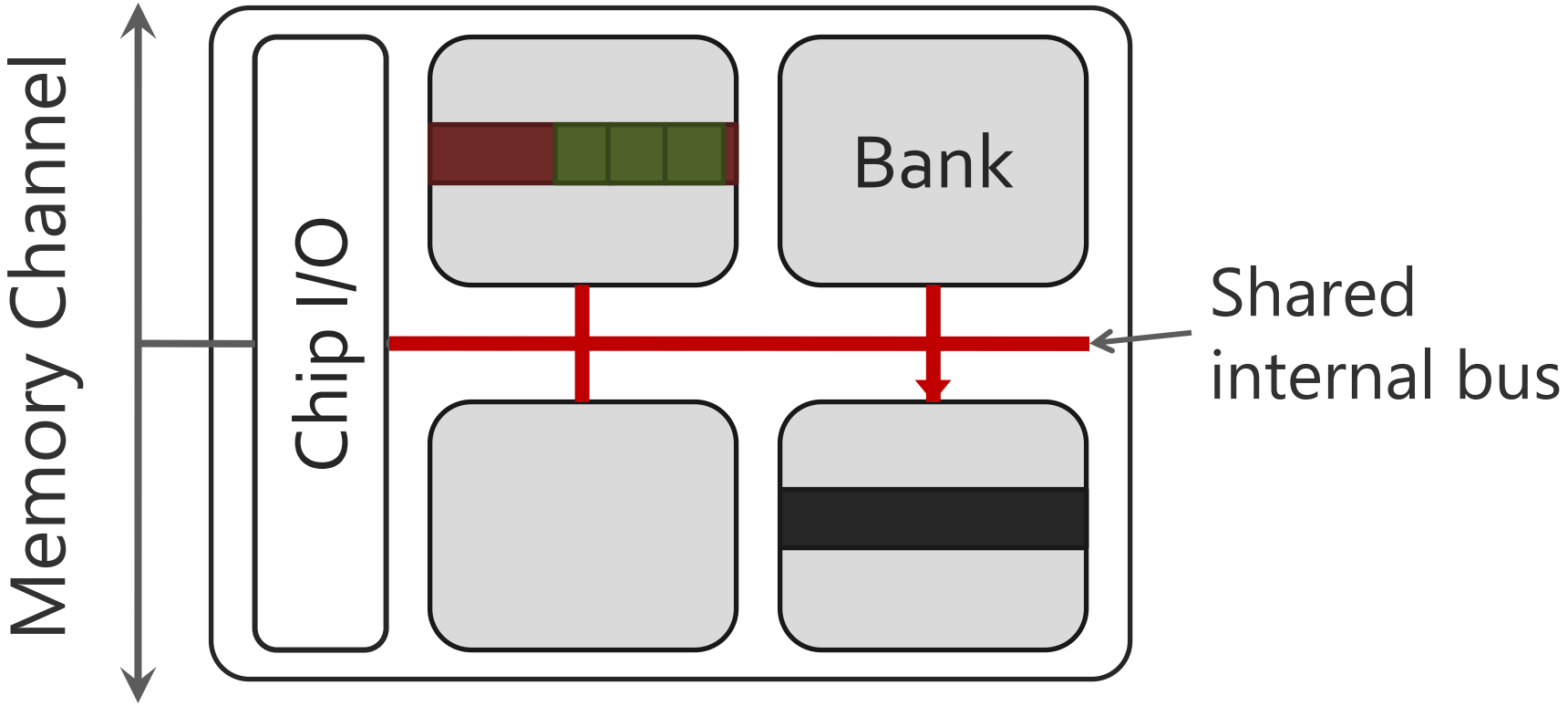
RowClone: Intra-Subarray (II)



1. **Activate** src row (copy data from src to row buffer)

2. **Activate** dst row (disconnect src from row buffer, connect dst – copy data from row buffer to dst)

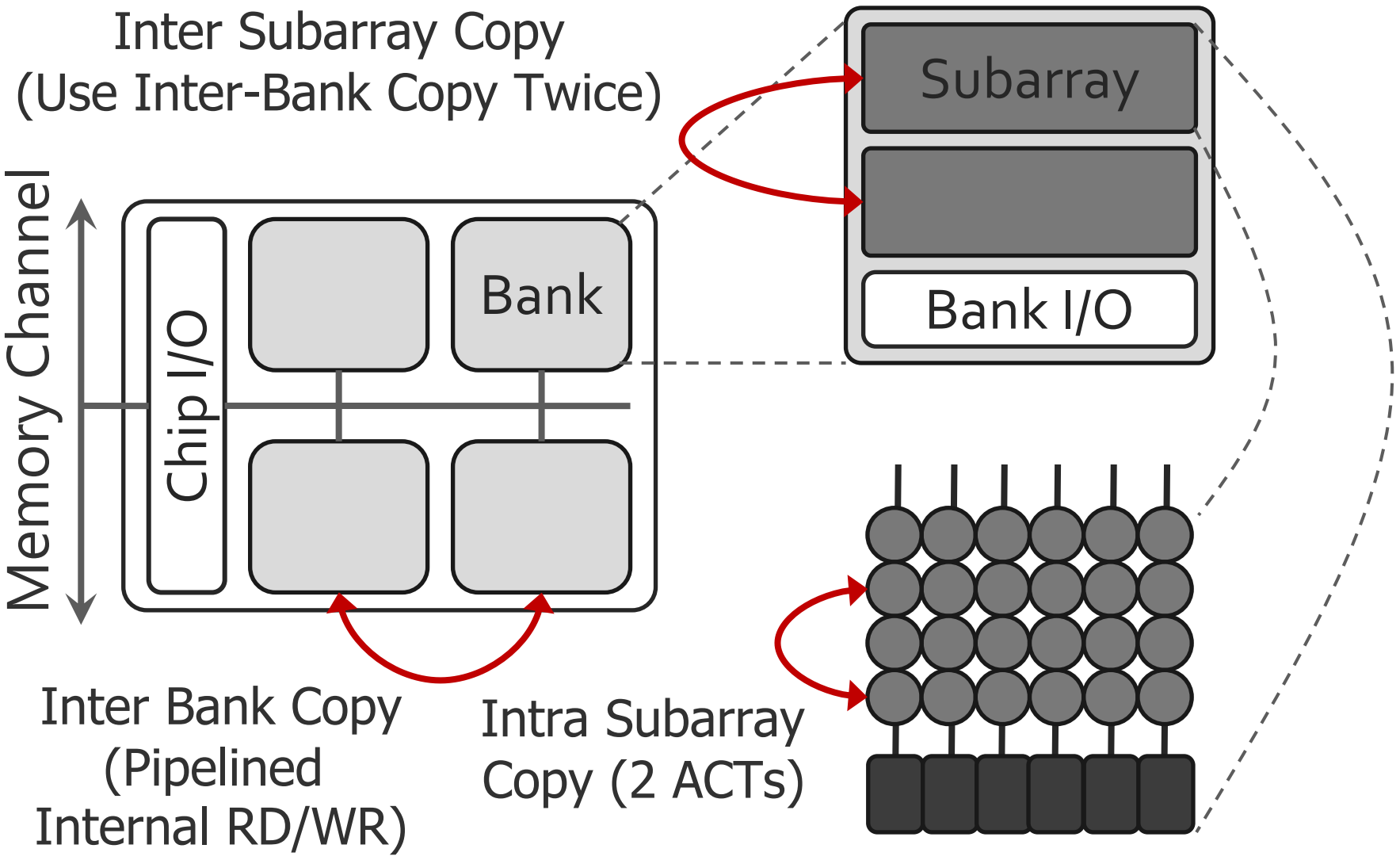
RowClone: Inter-Bank



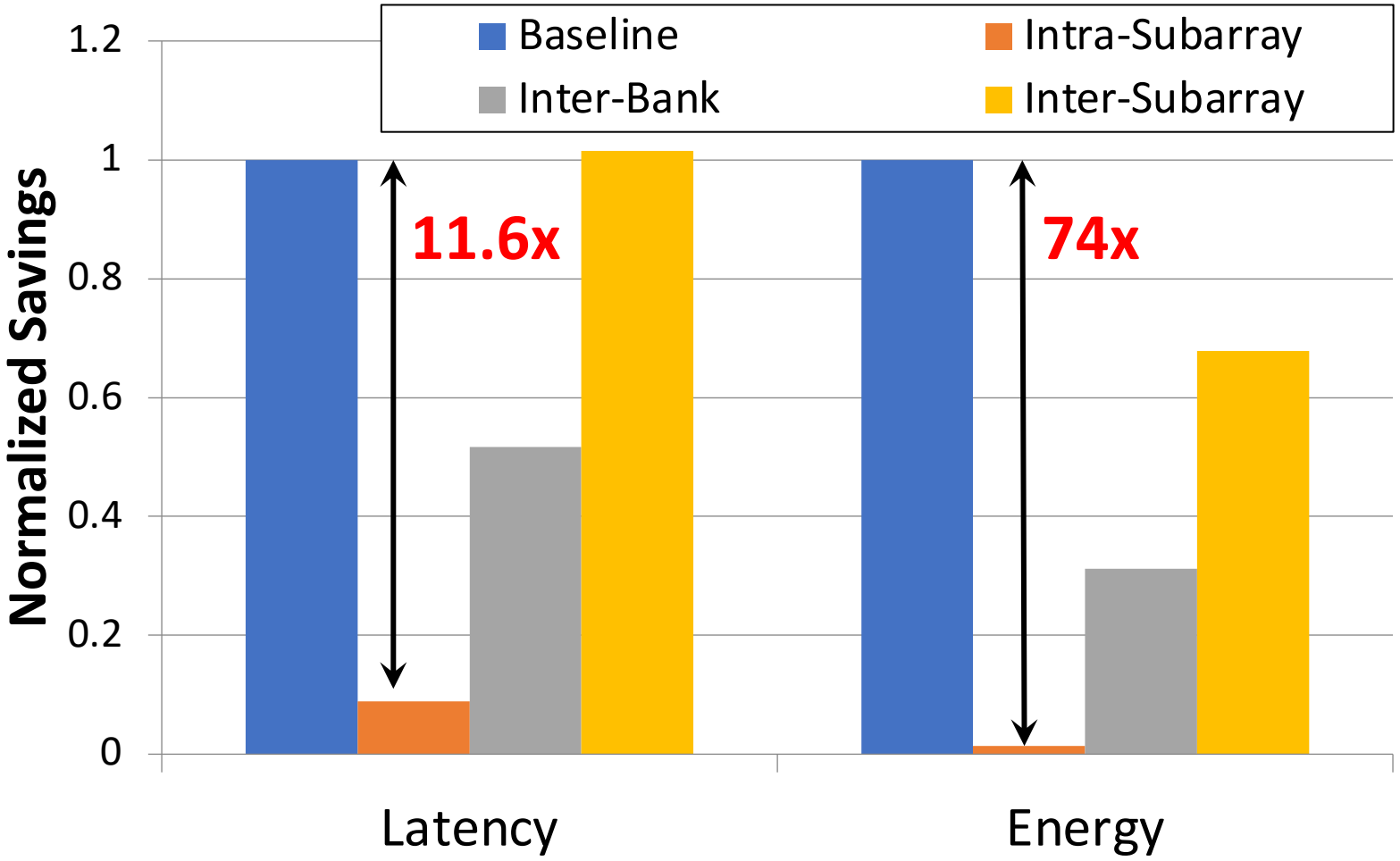
Overlap the latency of the read and the write
1.9X latency reduction, 3.2X energy reduction

Generalized RowClone

0.01% area cost

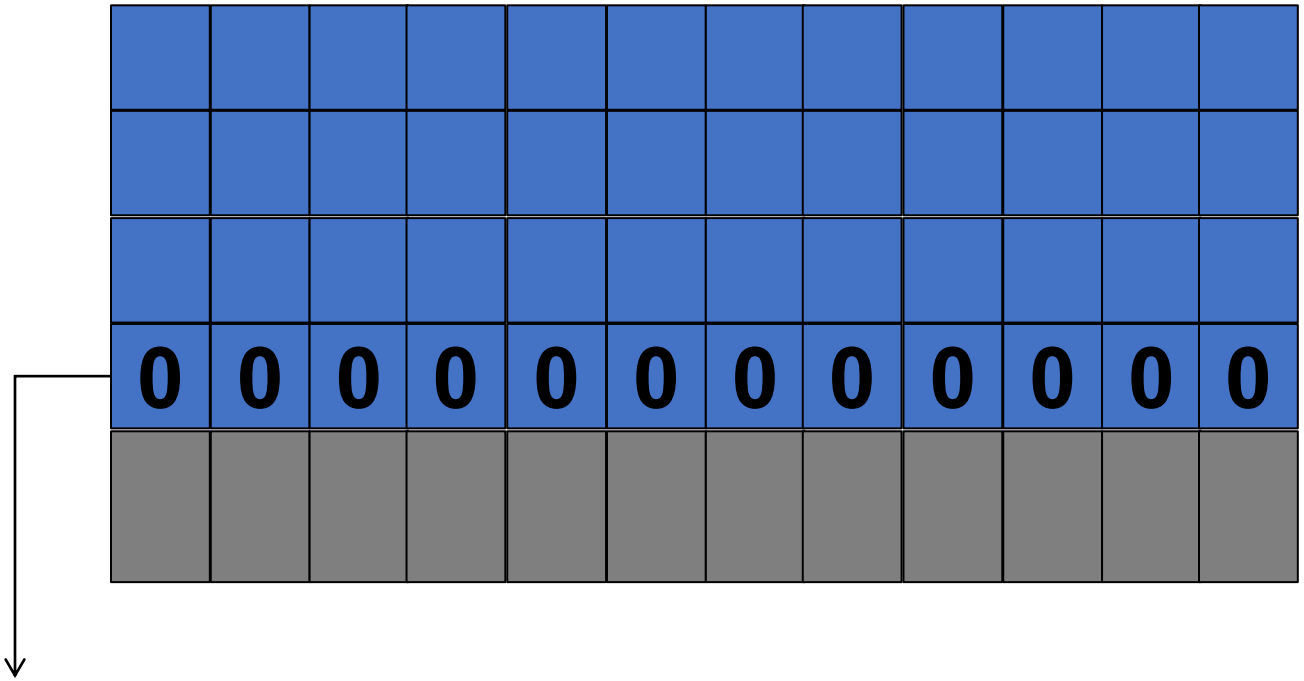


RowClone: Latency and Energy Savings



Seshadri et al., "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," MICRO 2013

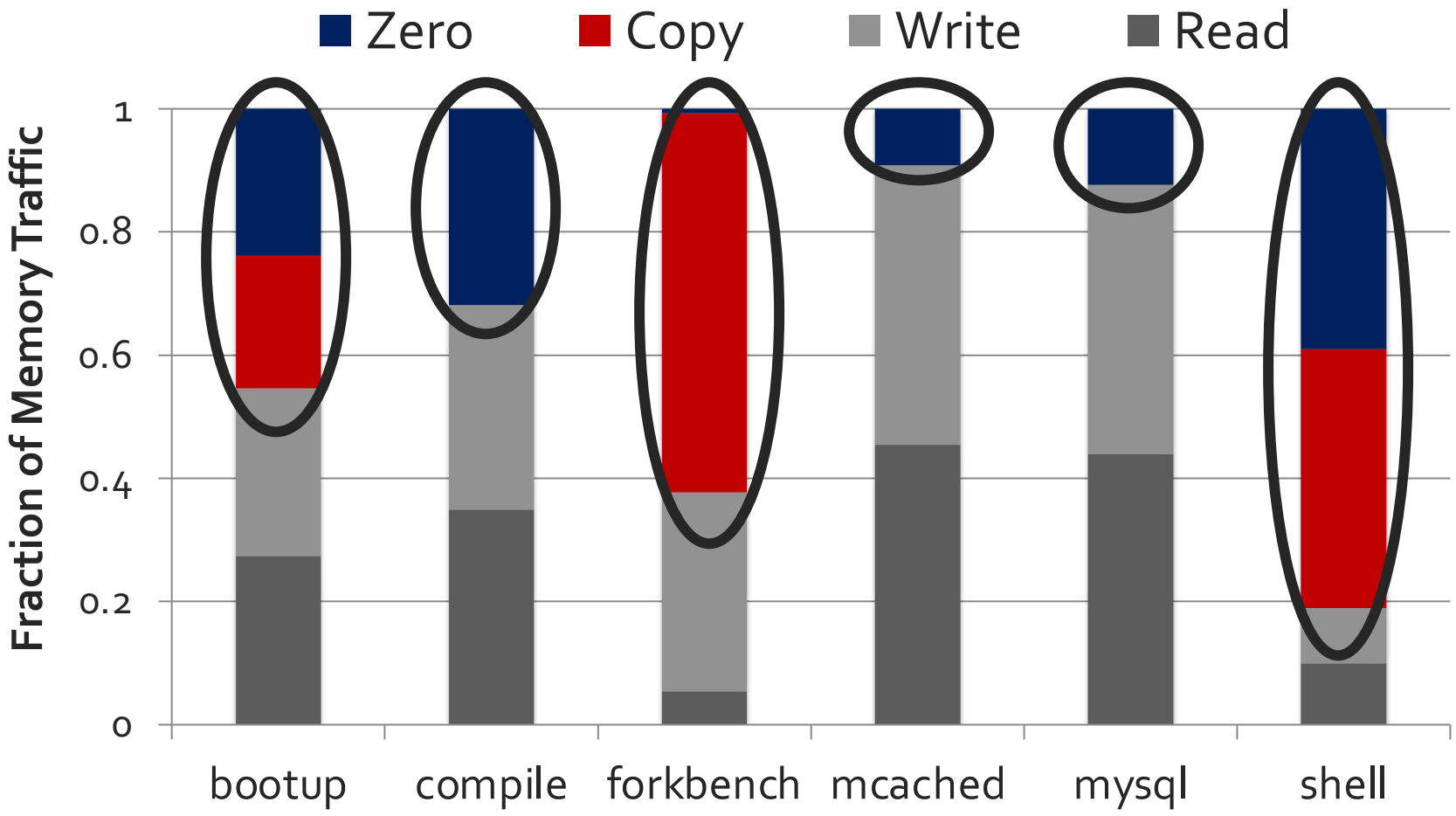
RowClone: Fast Row Initialization



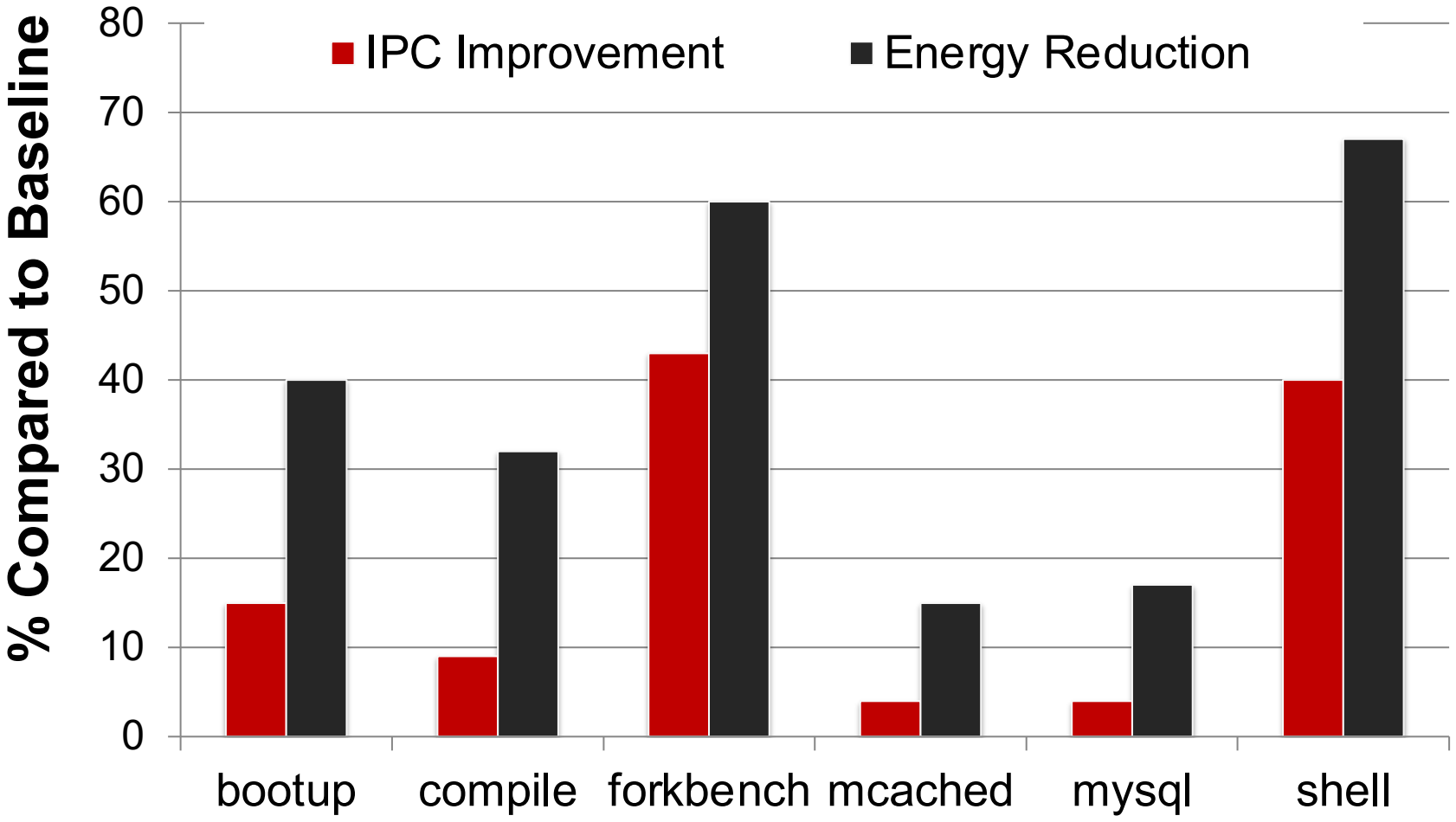
Fix a row at Zero

(0.5% loss in capacity)

Copy and Initialization in Workloads



RowClone: Application Performance



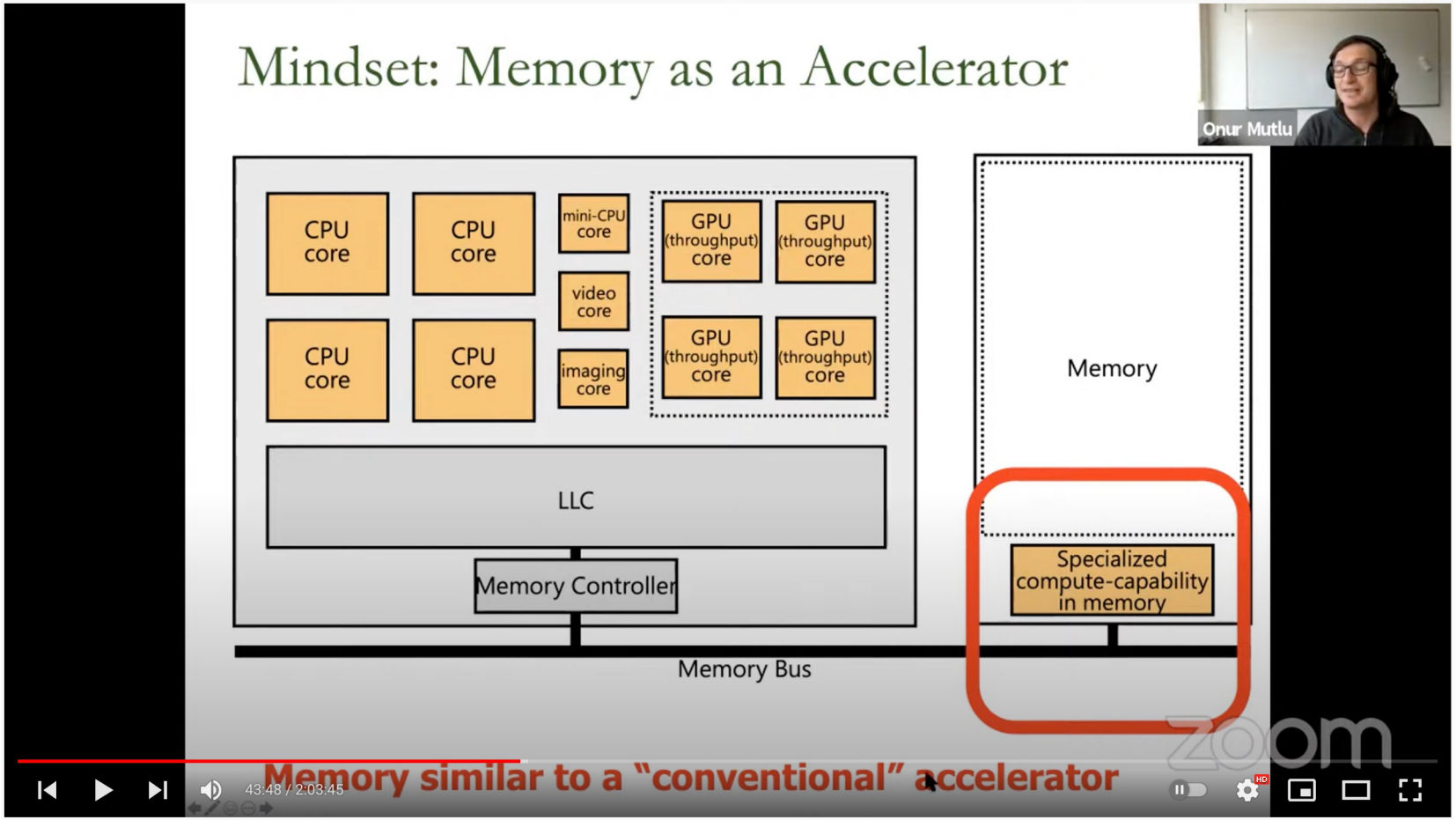
More on RowClone

- Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry,
  "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"
  Proceedings of the 46th International Symposium on Microarchitecture (***MICRO***), Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]

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

Vivek Seshadri vseshadr@cs.cmu.edu
Yoongu Kim yoongukim@cmu.edu
Chris Fallin* cfallin@c1f.net
Donghyuk Lee donghyuk1@cmu.edu
Rachata Ausavarungnirun rachata@cmu.edu
Gennady Pekhimenko gpekhime@cs.cmu.edu
Yixin Luo yixinluo@andrew.cmu.edu
Onur Mutlu onur@cmu.edu
Phillip B. Gibbons† phillip.b.gibbons@intel.com
Michael A. Kozuch† michael.a.kozuch@intel.com
Todd C. Mowry tcm@cs.cmu.edu
Carnegie Mellon University †Intel Pittsburgh

Lecture on RowClone & Processing using DRAM



DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING (D-ITET)

Seminar in Computer Arch. - Meeting 3: RowClone: In-Memory Data Copy and Initialization (Fall 2021)

292 views • Streamed live on Oct 7, 2021

21 0 SHARE SAVE ...

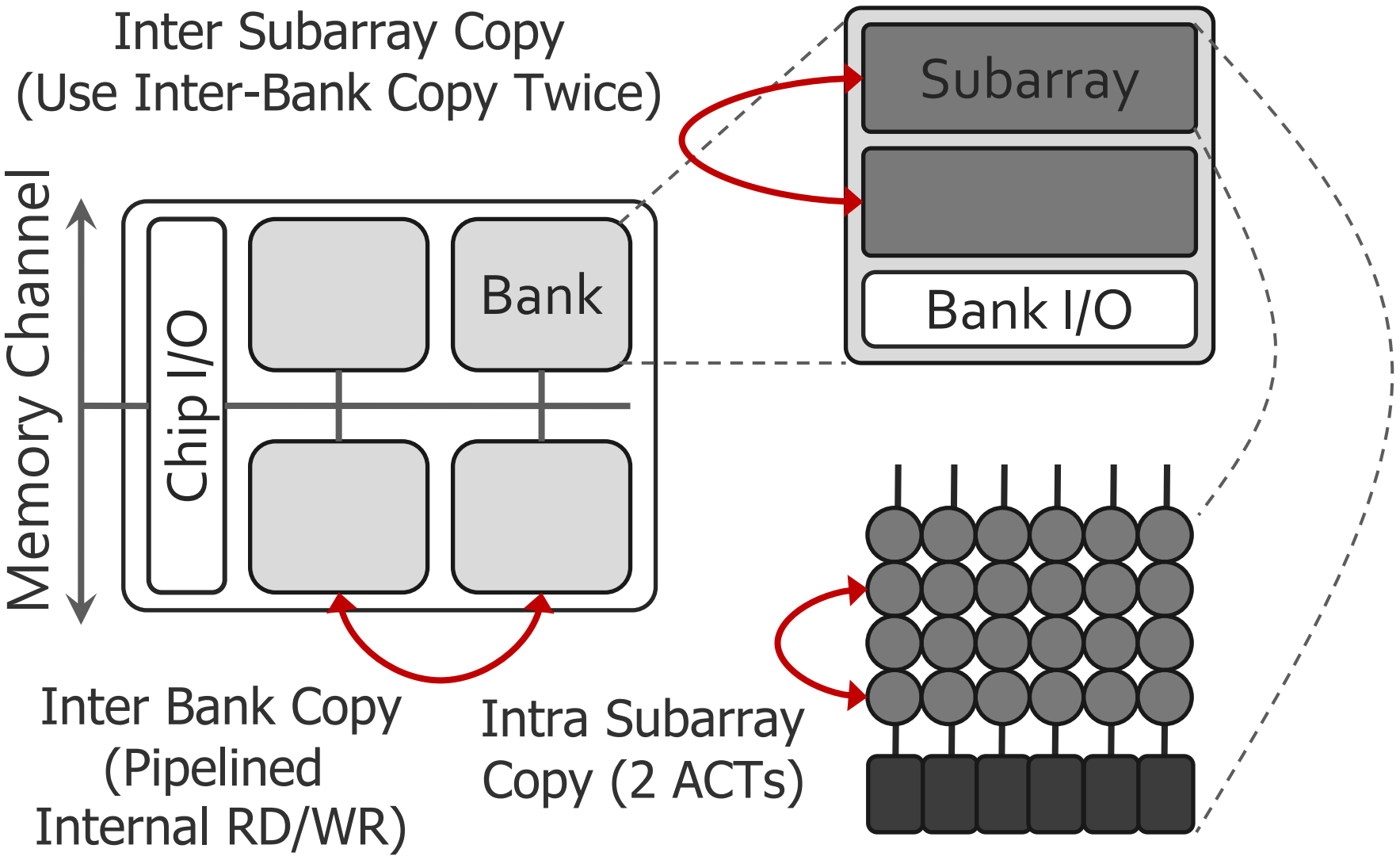
Onur Mutlu Lectures

19.1K subscribers

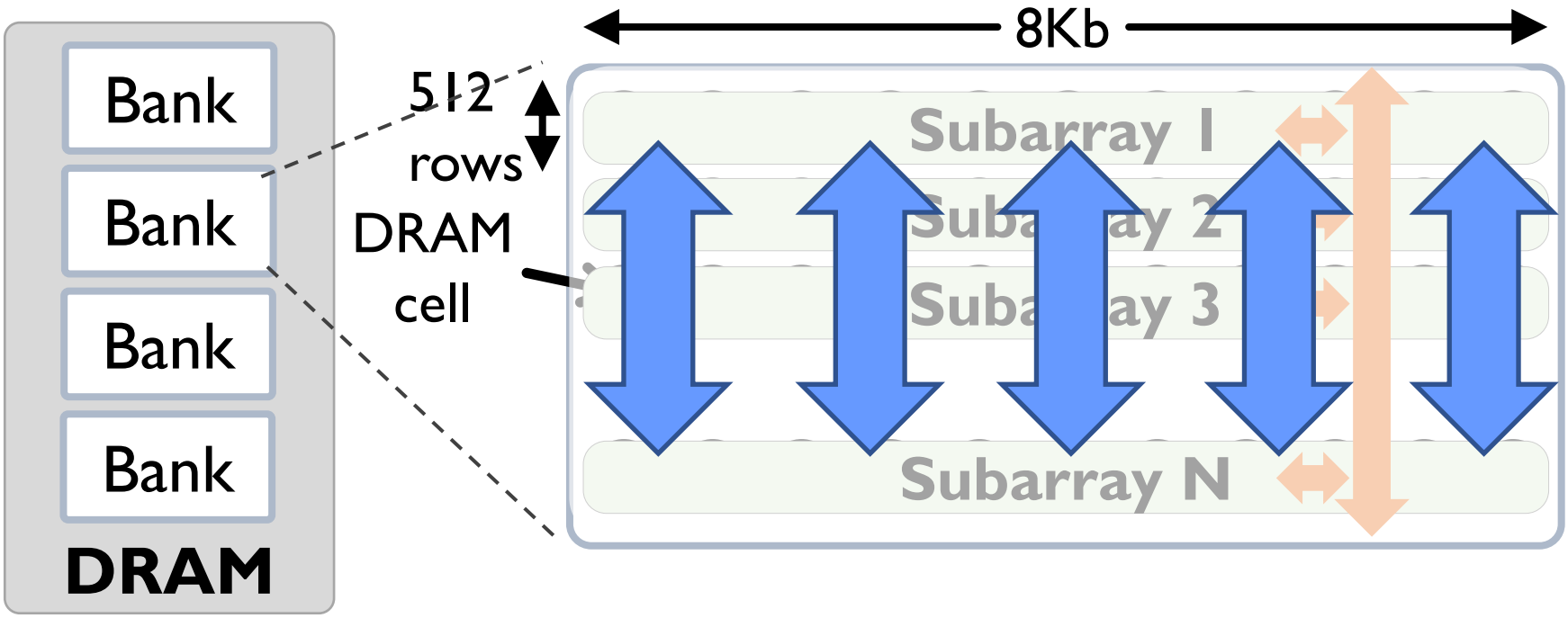
SUBSCRIBED

Generalized RowClone

0.01% area cost



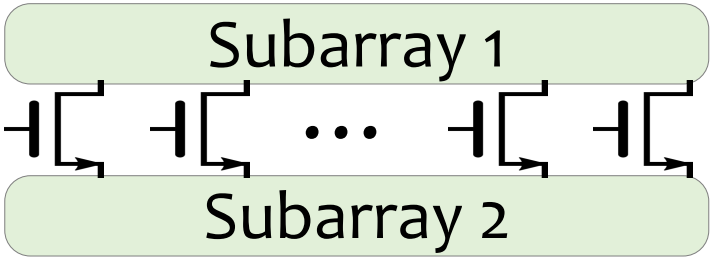
Moving Data Inside DRAM?



Goal: Provide a new substrate to enable wide connectivity between subarrays

Key Idea and Applications

- **Low-cost Inter-linked subarrays (LISA)**
  - Fast bulk data movement between subarrays
  - Wide datapath via isolation transistors: 0.8% DRAM chip area



- LISA is a **versatile substrate** → new applications

  Fast bulk data copy: Copy latency 1.363ms→0.148ms (9.2x)
  → 66% speedup, -55% DRAM energy

  In-DRAM caching: Hot data access latency 48.7ns→21.5ns (2.2x)
  → 5% speedup

  Fast precharge: Precharge latency 13.1ns→5.0ns (2.6x)
  → 8% speedup

More on LISA

- Kevin K. Chang, Prashant J. Nair, Saugata Ghose, Donghyuk Lee, Moinuddin K. Qureshi, and Onur Mutlu,
  "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM"
  Proceedings of the 22nd International Symposium on High-Performance Computer Architecture (***HPCA***), Barcelona, Spain, March 2016.
  [Slides (pptx) (pdf)]
  [Source Code]

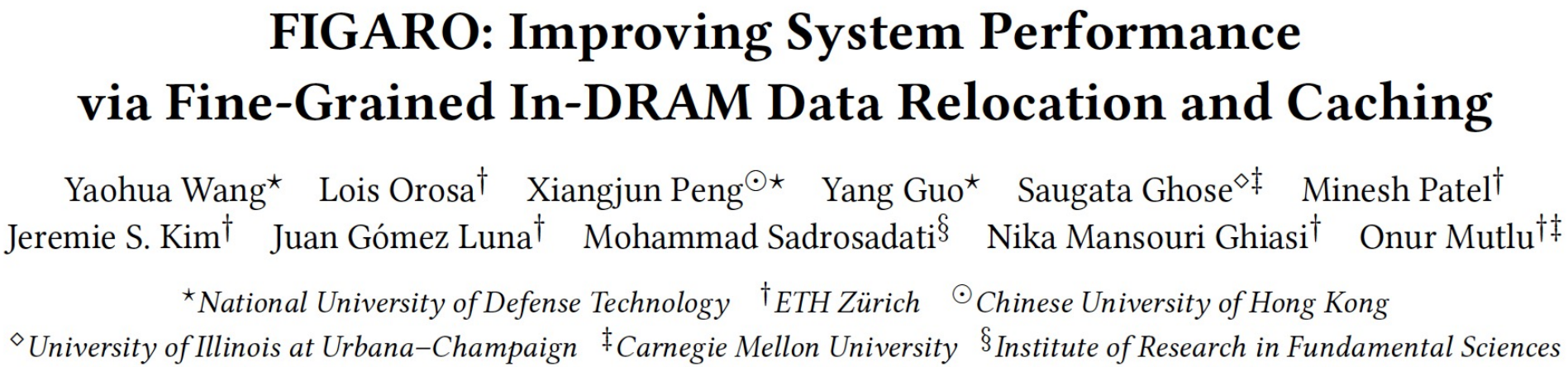
Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM

Kevin K. Chang†, Prashant J. Nair*, Donghyuk Lee†, Saugata Ghose†, Moinuddin K. Qureshi*, and Onur Mutlu†

†*Carnegie Mellon University* **Georgia Institute of Technology*

FIGARO: Fine-Grained In-DRAM Copy

- Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S. Kim, Juan Gómez Luna, Mohammad Sadrosadati, Nika Mansouri Ghiasi, and Onur Mutlu,
  "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching"
  Proceedings of the 53rd International Symposium on Microarchitecture (***MICRO***), Virtual, October 2020.



FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching

Yaohua Wang[⋆] Lois Orosa[†] Xiangjun Peng[⊙⋆] Yang Guo[⋆] Saugata Ghose[◇‡] Minesh Patel[†]
Jeremie S. Kim[†] Juan Gómez Luna[†] Mohammad Sadrosadati[§] Nika Mansouri Ghiasi[†] Onur Mutlu[†‡]

[⋆]*National University of Defense Technology* [†]*ETH Zürich* [⊙]*Chinese University of Hong Kong*
[◇]*University of Illinois at Urbana–Champaign* [‡]*Carnegie Mellon University* [§]*Institute of Research in Fundamental Sciences*

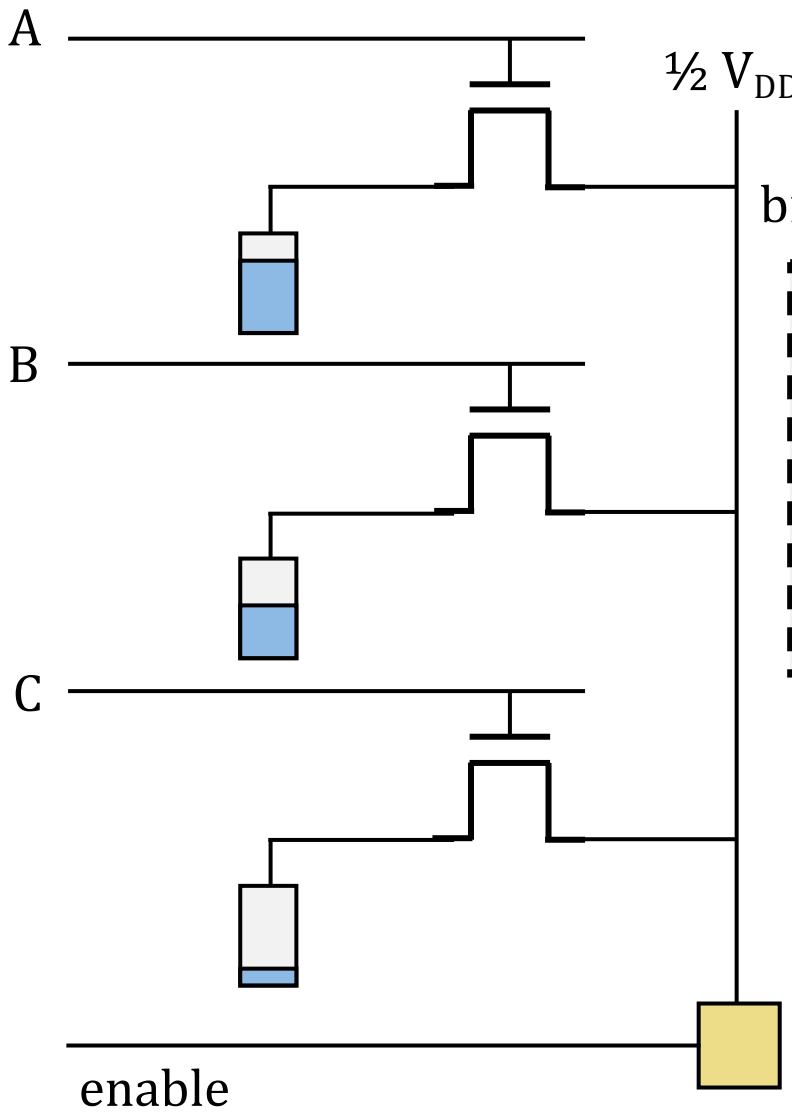
In-Memory Bulk Bitwise Operations

In-Memory Bulk Bitwise Operations

- 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
  - Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," MICRO 2017.

- New memory technologies enable even more opportunities
  - Memristors, resistive RAM, phase change memory, STT-MRAM, …
  - Can operate on data with minimal movement

Triple-Row Activation: Majority Function (I)



Majority function command sequence[3]:

1. ACTIVATE (ACT)
2. PRECHARGE (PRE)

[3] V. Seshadri et al., "*Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology*", MICRO, 2017

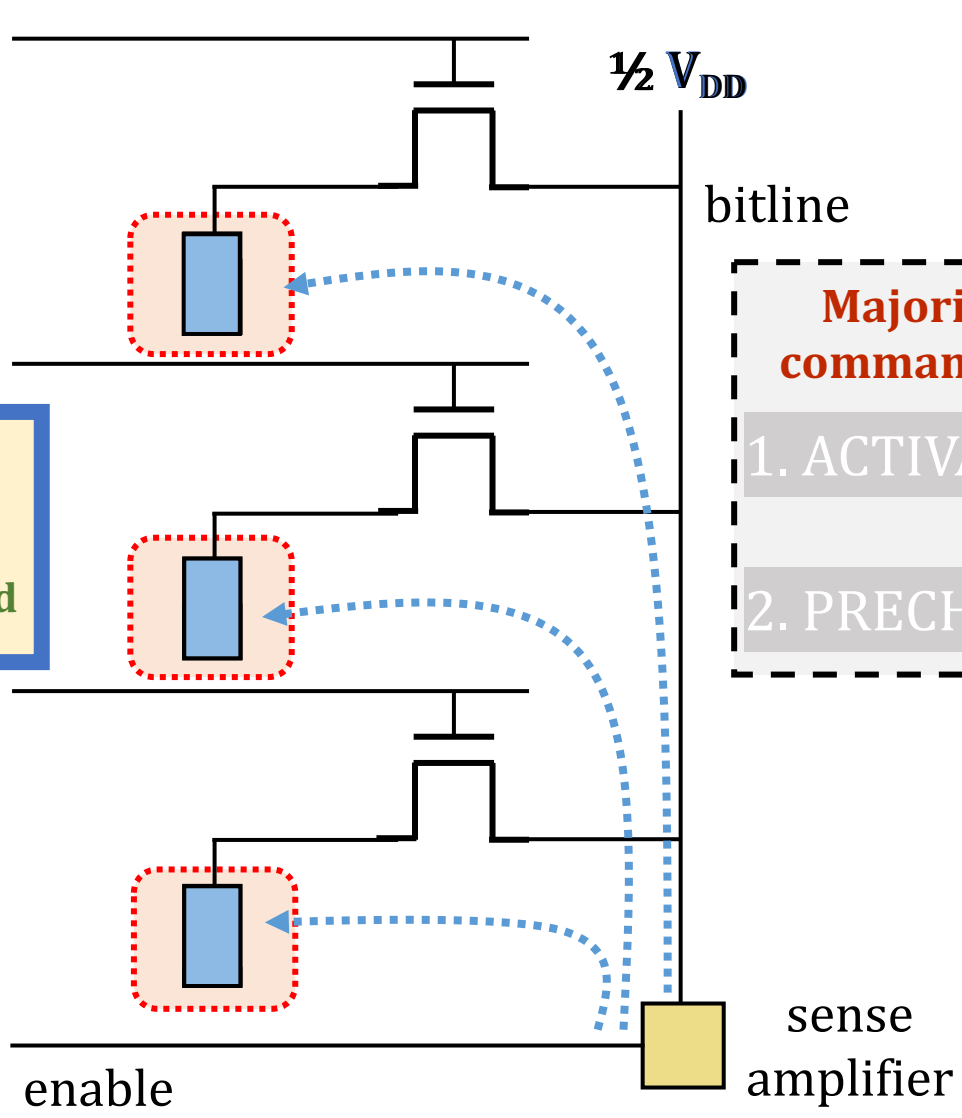
Triple-Row Activation: Majority Function (II)

1. ACTIVATE three rows simultaneously
→ **triple-row activation**

MAJ(A, B, C) =
MAJ(V_{dd}, V_{dd}, 0) = V_{dd}

3. values in cells A, B, C will be overwritten with the majority output

4. PRECHARGE bitline for next access

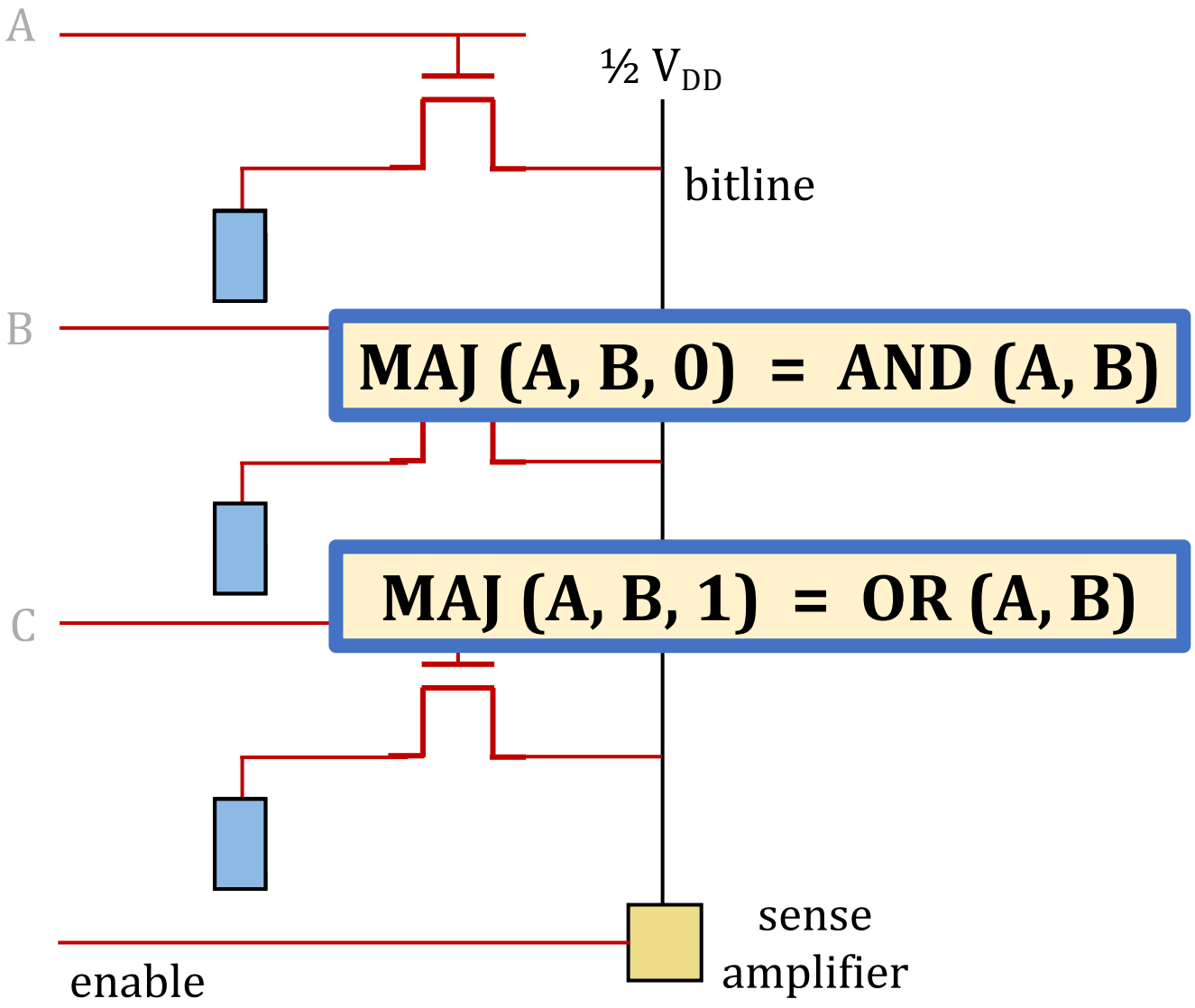


Majority function command sequence[3]:

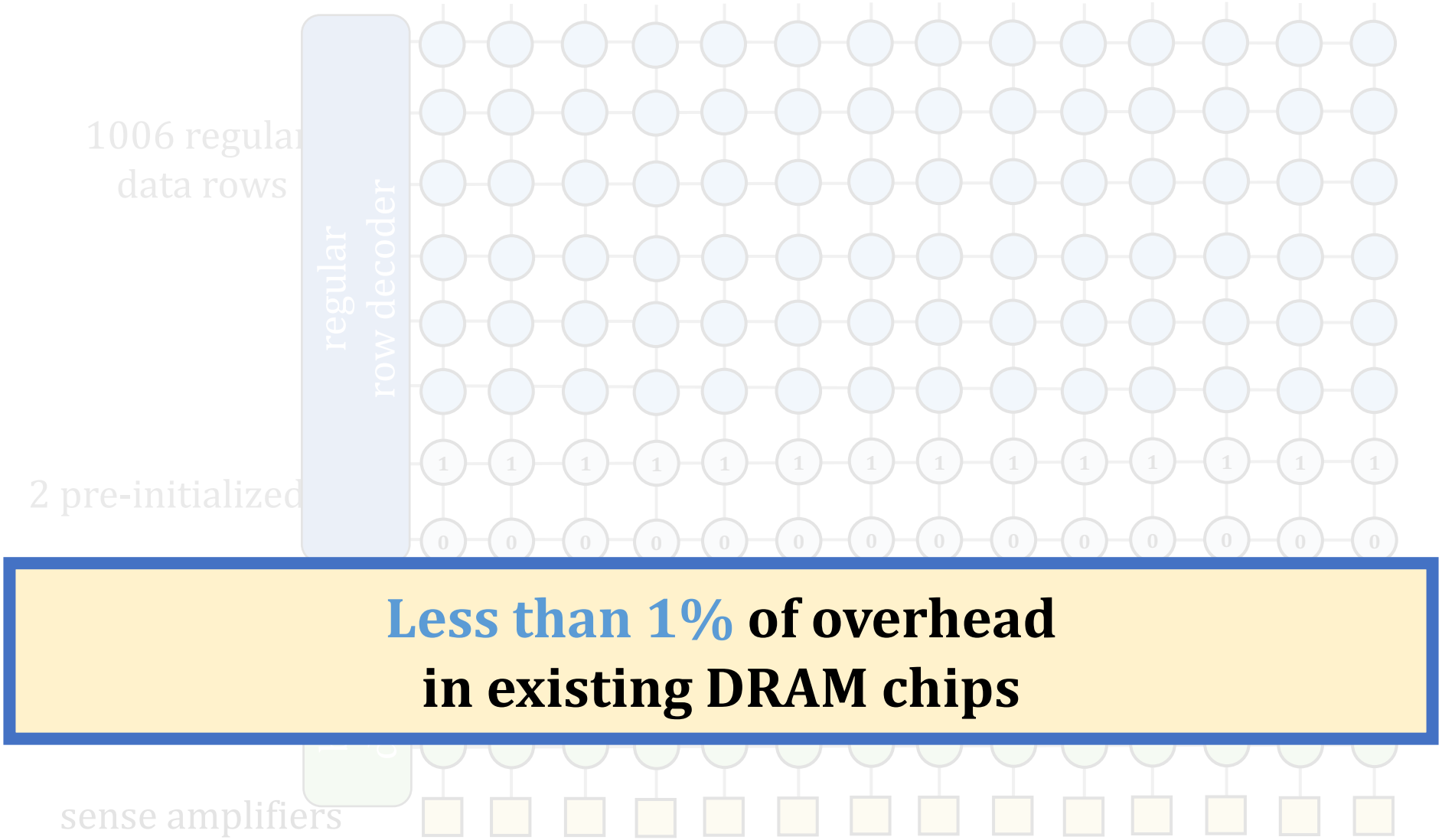
1. ACTIVATE (ACT)
2. PRECHARGE (PRE)

[3] V. Seshadri et al., "*Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology*", MICRO, 2017

Ambit: In-DRAM Bulk Bitwise AND/OR



Ambit: Subarray Organization



Less than 1% of overhead in existing DRAM chips

In-DRAM Bulk Bitwise AND/OR Operation

- BULKAND A, B → C
- Semantics: Perform a bitwise AND of two rows A and B and store the result in row C

- R0 – reserved zero row, R1 – reserved one row
- D1, D2, D3 – Designated rows for triple activation

1. RowClone A into D1
2. RowClone B into D2
3. RowClone R0 into D3
4. ACTIVATE D1,D2,D3
5. RowClone Result into C

More on In-DRAM Bulk AND/OR

- 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"
  IEEE Computer Architecture Letters (***CAL***), April 2015.

Fast Bulk Bitwise AND and OR in DRAM

Vivek Seshadri*, Kevin Hsieh*, Amirali Boroumand*, Donghyuk Lee*,
Michael A. Kozuch†, Onur Mutlu*, Phillip B. Gibbons†, Todd C. Mowry*

*Carnegie Mellon University †Intel Pittsburgh

In-DRAM NOT: Dual Contact Cell

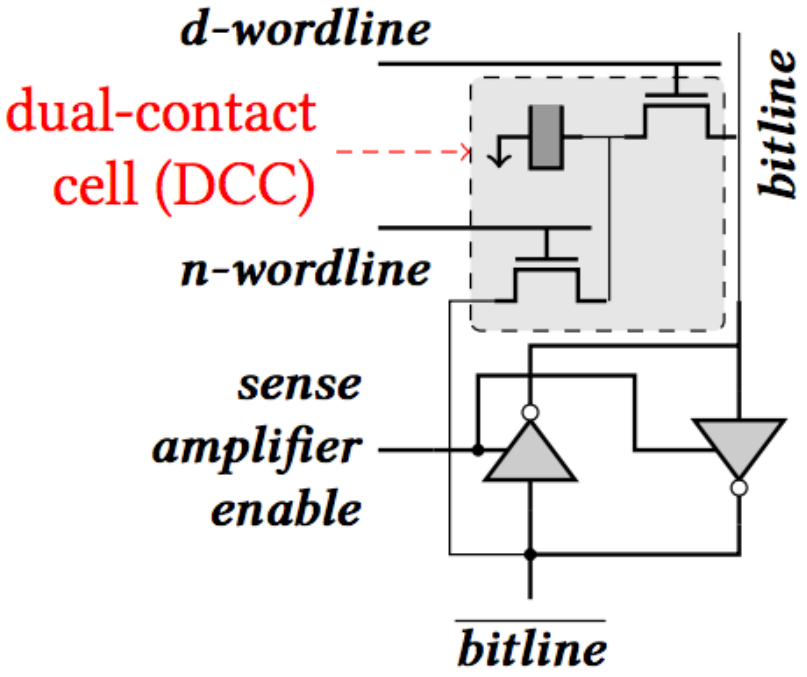


Figure 5: A dual-contact cell connected to both ends of a sense amplifier

Idea:

Feed the negated value in the sense amplifier into a special row

Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," MICRO 2017

In-DRAM NOT Operation

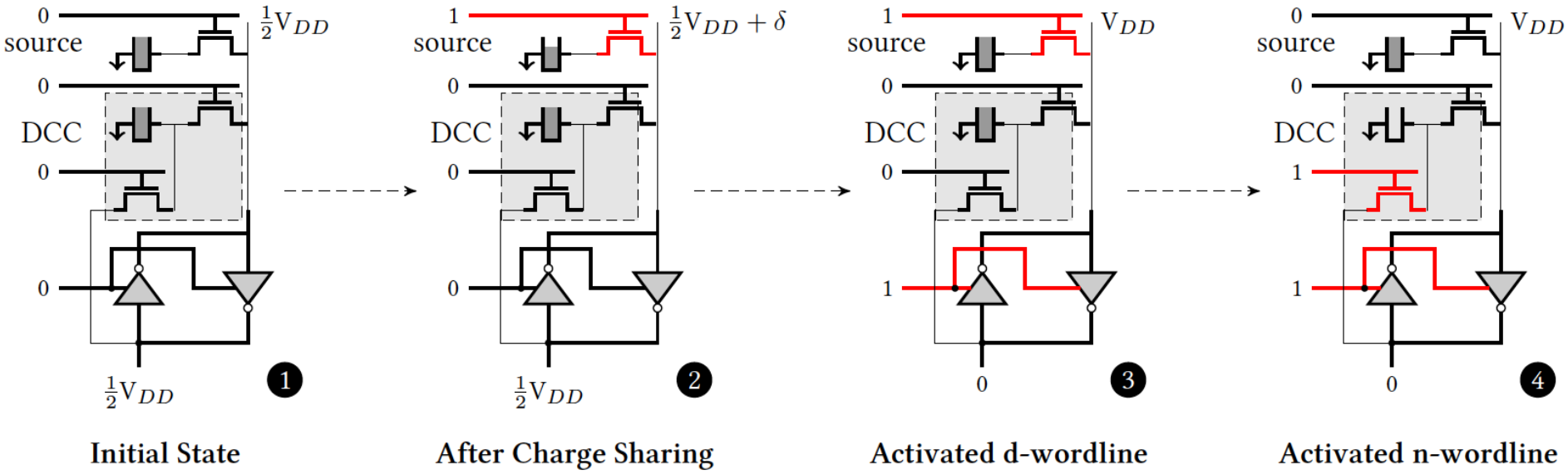


Figure 5: Bitwise NOT using a dual contact capacitor

Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," MICRO 2017

Performance: In-DRAM Bitwise Operations

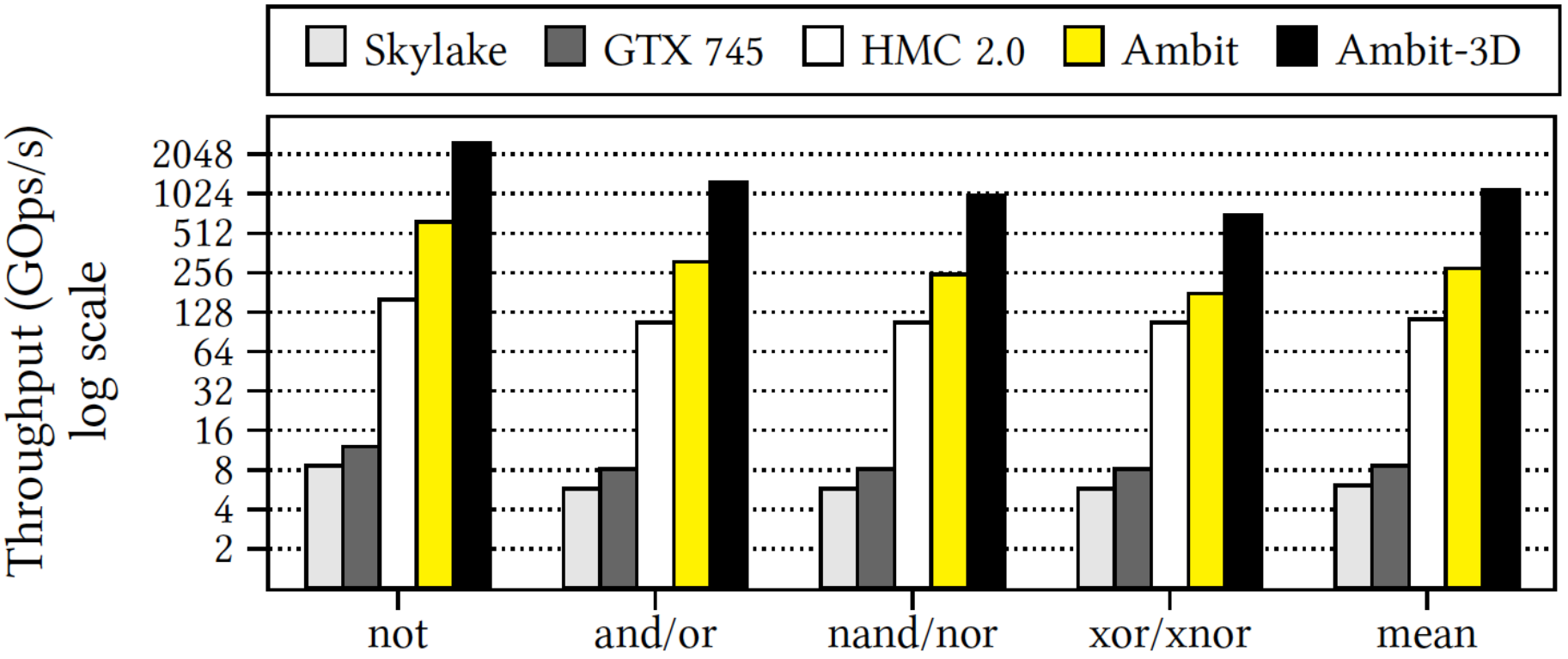


Figure 9: Throughput of bitwise operations on various systems.

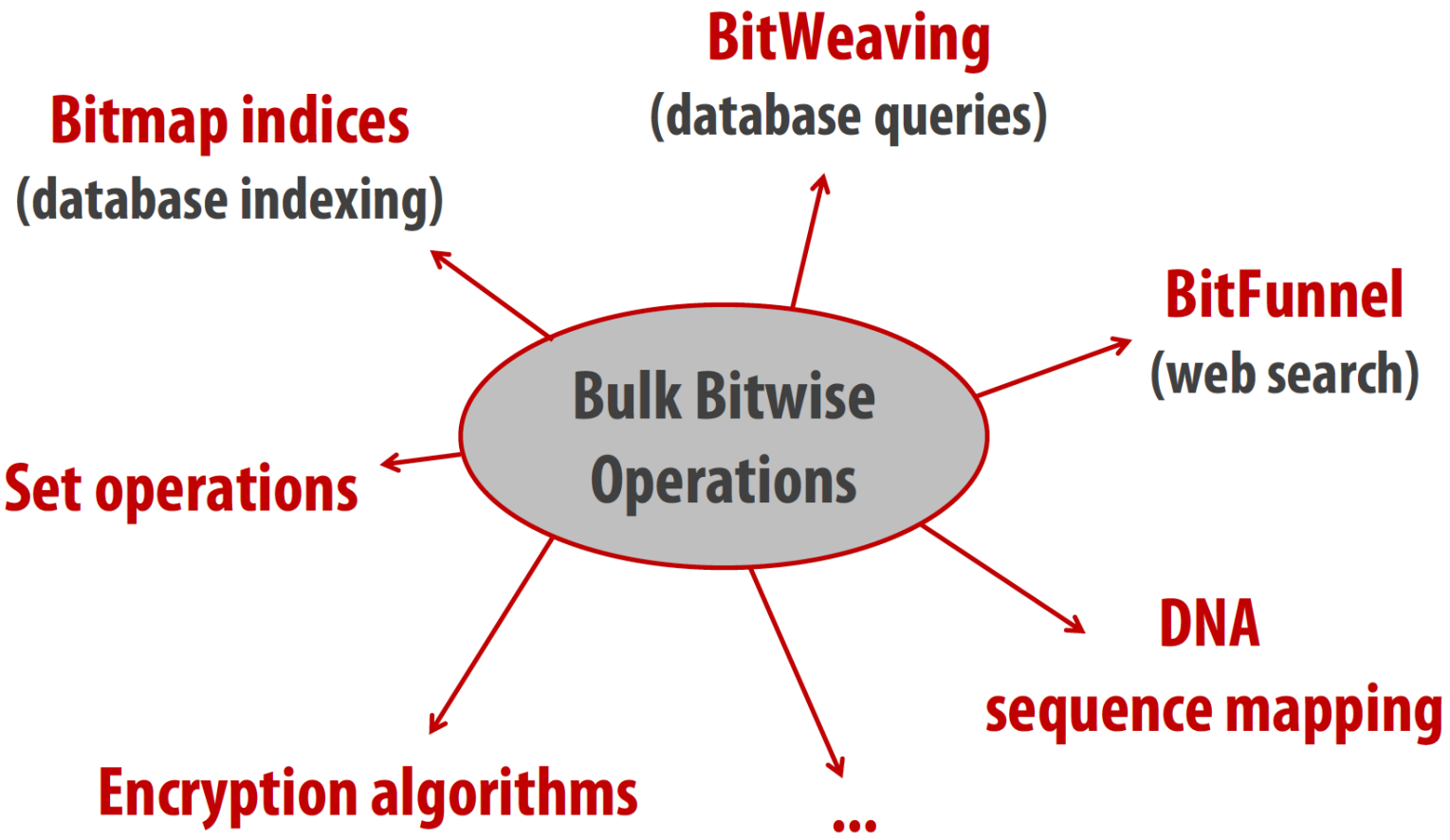
Energy of In-DRAM Bitwise Operations

	Design	not	and/or	nand/nor	xor/xnor
DRAM &	**DDR3**	93.7	137.9	137.9	137.9
Channel Energy	**Ambit**	1.6	3.2	4.0	5.5
(nJ/KB)	(↓)	59.5X	43.9X	35.1X	25.1X

Table 3: Energy of bitwise operations. (↓) indicates energy reduction of Ambit over the traditional DDR3-based design.

Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," MICRO 2017

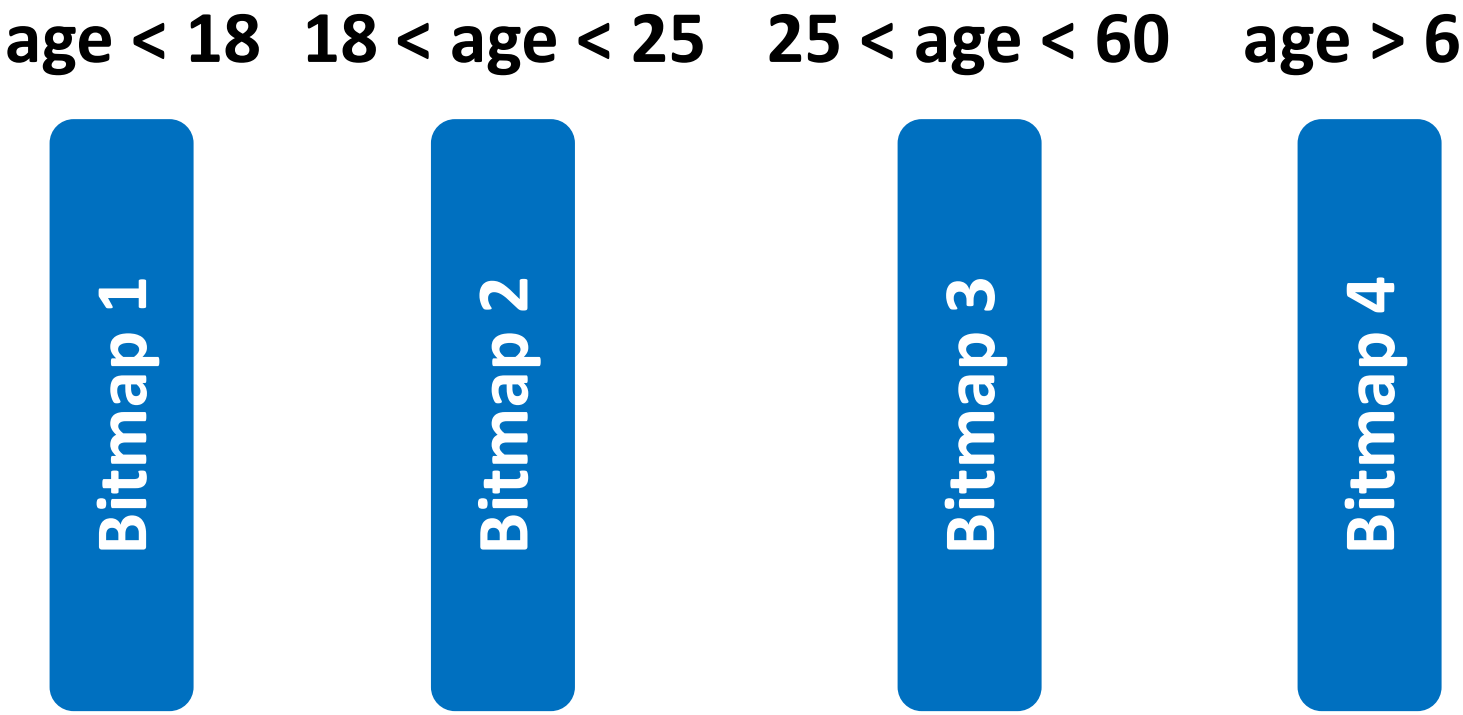
Bulk Bitwise Operations in Workloads



[1] Li and Patel, BitWeaving, SIGMOD 2013
[2] Goodwin+, BitFunnel, SIGIR 2017

Example Data Structure: Bitmap Index

- Alternative to B-tree and its variants
- Efficient for performing *range queries* and *joins*
- Many bitwise operations to perform a query



Performance: Bitmap Index on Ambit

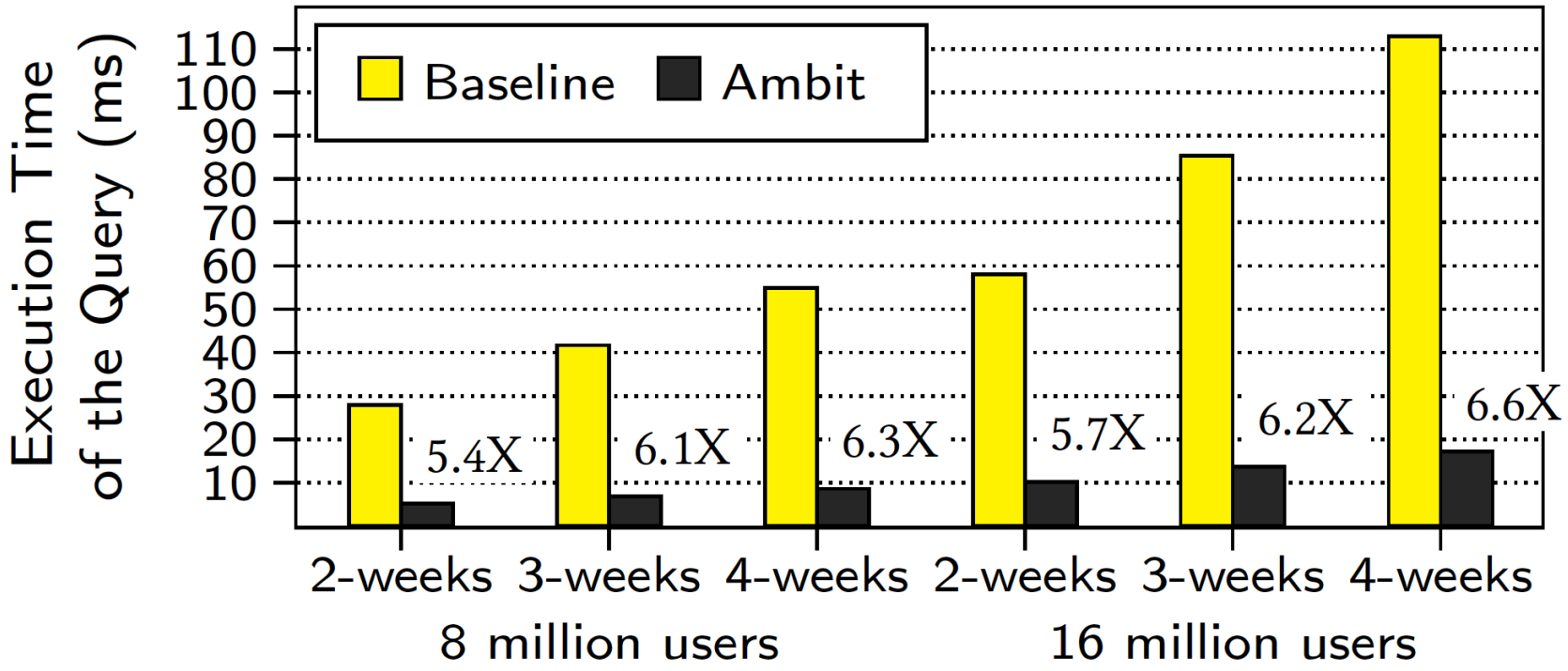


Figure 10: Bitmap index performance. The value above each bar indicates the reduction in execution time due to Ambit.

>5.4-6.6X Performance Improvement

Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," MICRO 2017

Performance: BitWeaving on Ambit

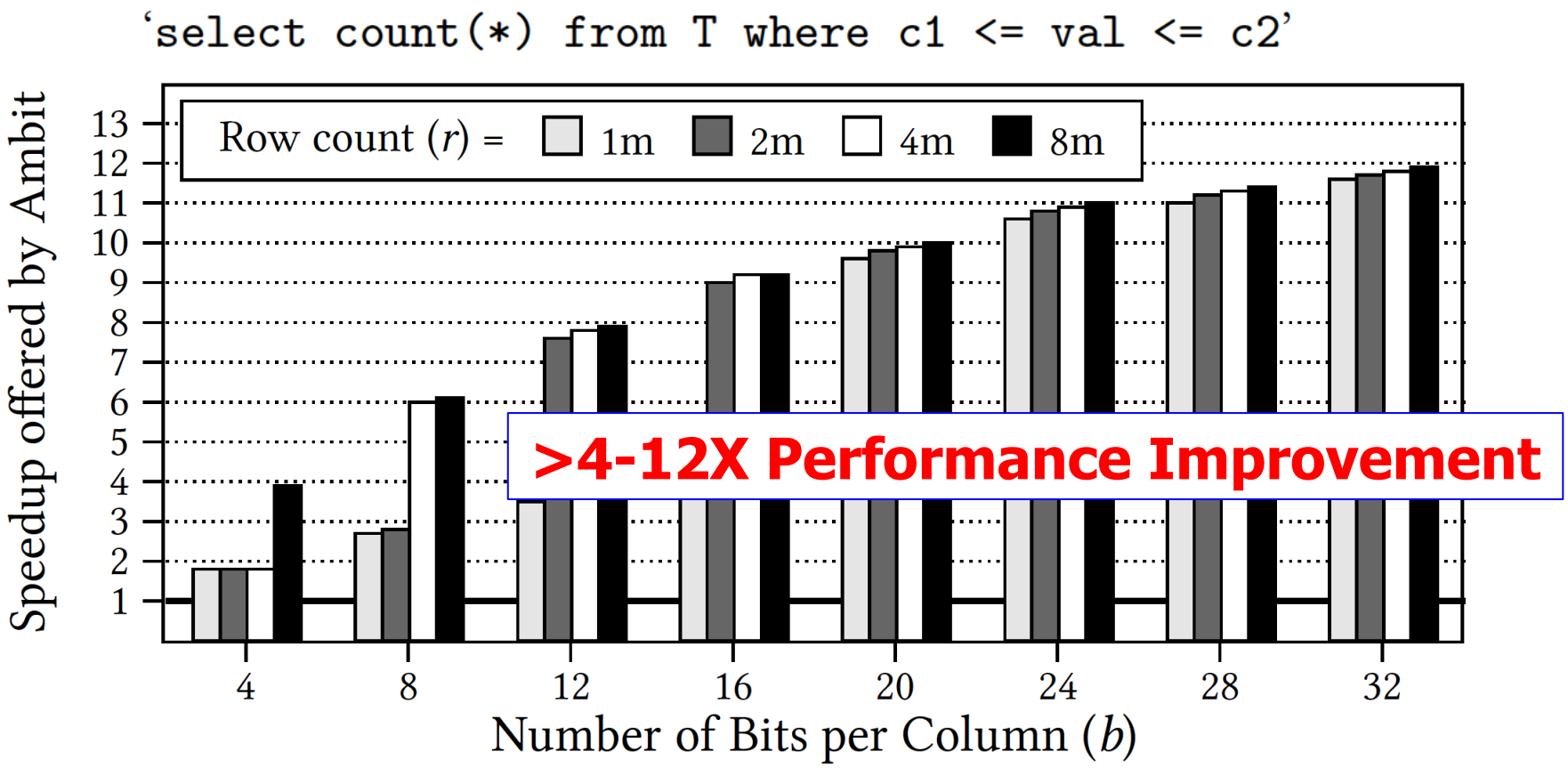


Figure 11: Speedup offered by Ambit over baseline CPU with SIMD for BitWeaving

Seshadri+, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," MICRO 2017

More on In-DRAM Bitwise Operations

- Vivek Seshadri et al., "**Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology**," MICRO 2017.

Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology

Vivek Seshadri[1,5] Donghyuk Lee[2,5] Thomas Mullins[3,5] Hasan Hassan[4] Amirali Boroumand[5]
Jeremie Kim[4,5] Michael A. Kozuch[3] Onur Mutlu[4,5] Phillip B. Gibbons[5] Todd C. Mowry[5]

[1]Microsoft Research India [2]NVIDIA Research [3]Intel [4]ETH Zürich [5]Carnegie Mellon University

More on In-DRAM Bulk Bitwise Execution

- Vivek Seshadri and Onur Mutlu,
  "In-DRAM Bulk Bitwise Execution Engine"
  Invited Book Chapter in Advances in Computers, 2020.
  [Preliminary arXiv version]

In-DRAM Bulk Bitwise Execution Engine

Vivek Seshadri
Microsoft Research India
visesha@microsoft.com

Onur Mutlu
ETH Zürich
onur.mutlu@inf.ethz.ch

A Framework for PUM in DRAM

PUM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)
  - Not widely applicable
- Support a **limited** set of operations
  - Lack the flexibility to support new operations
- Require **significant changes** to the DRAM
  - Costly (e.g., area, power)

PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)

Need a framework that aids general adoption of PuM, by:

- Efficiently implementing complex operations

- Providing flexibility to support new operations

Our Goal

Goal: Design a PuM framework that

- Efficiently implements complex operations
- Provides the flexibility to support new desired operations
- Minimally changes the DRAM architecture

Key Idea

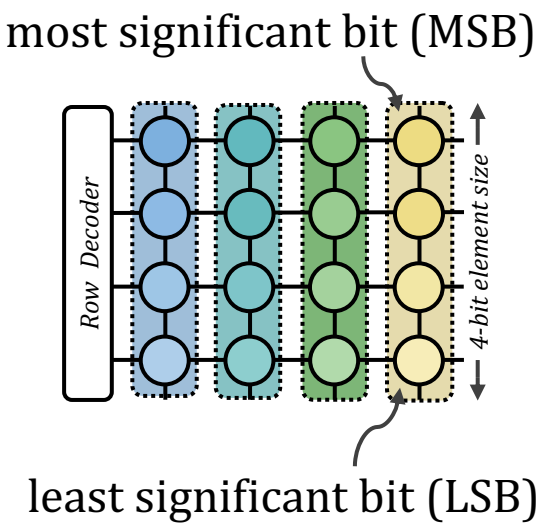
- **SIMDRAM:** An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:

  - Efficiently computing complex operations in DRAM
  - Providing the ability to implement arbitrary operations as required
  - Using an in-DRAM massively-parallel SIMD substrate that requires minimal changes to DRAM architecture

SIMDRAM: PUM Substrate

- SIMDRAM framework is built around a DRAM substrate that enables two techniques:

(1) Vertical data layout

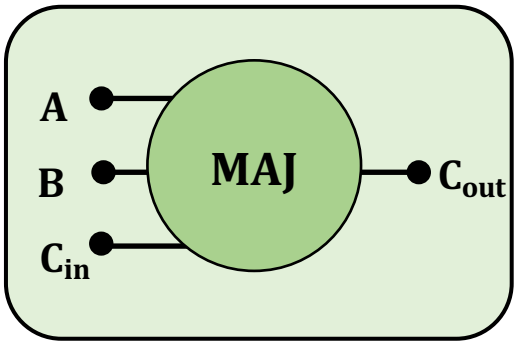


Pros compared to the conventional horizontal layout:

- Implicit shift operation
- Massive parallelism

(2) Majority-based computation

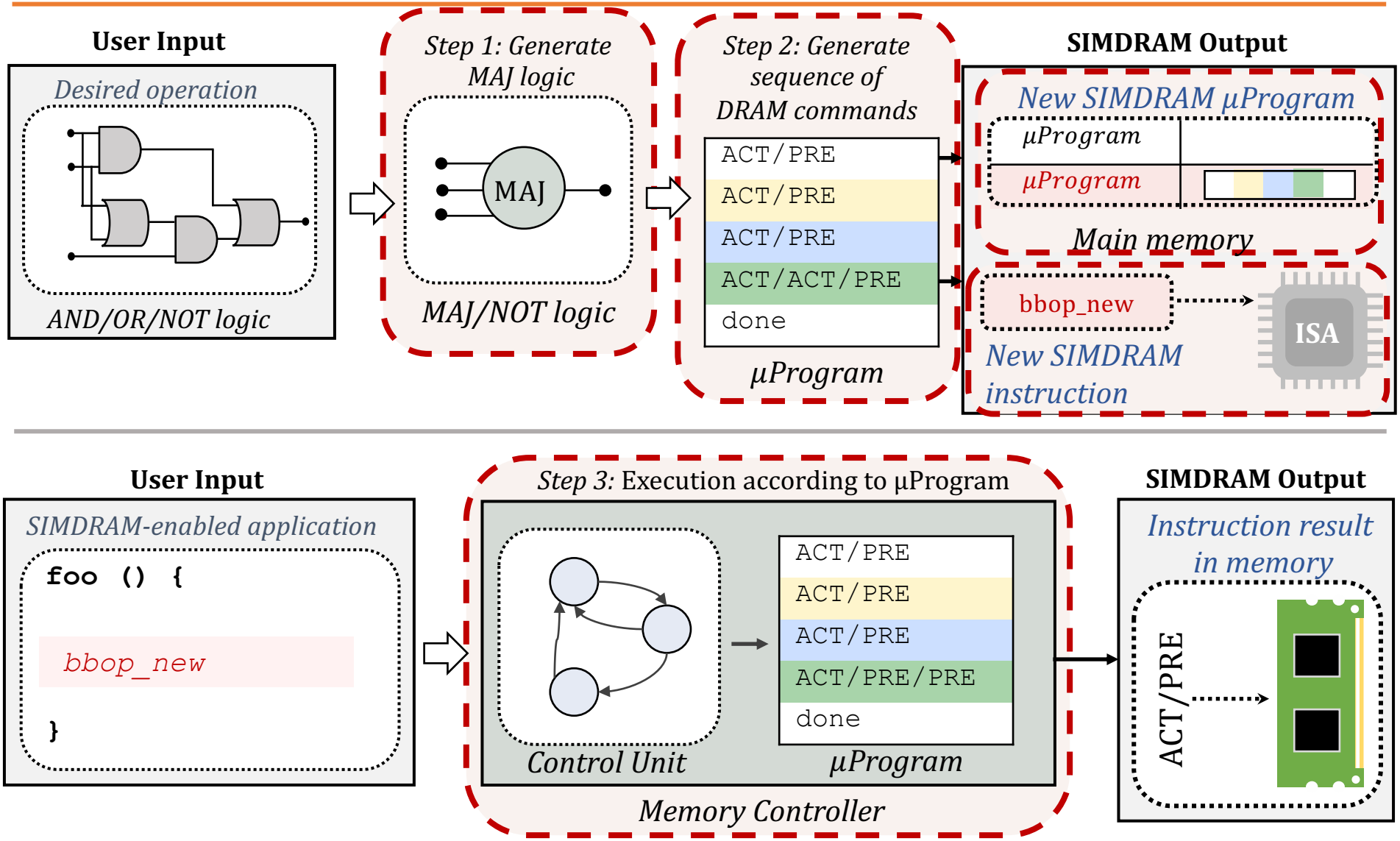
$C_{out} = AB + AC_{in} + BC_{in}$



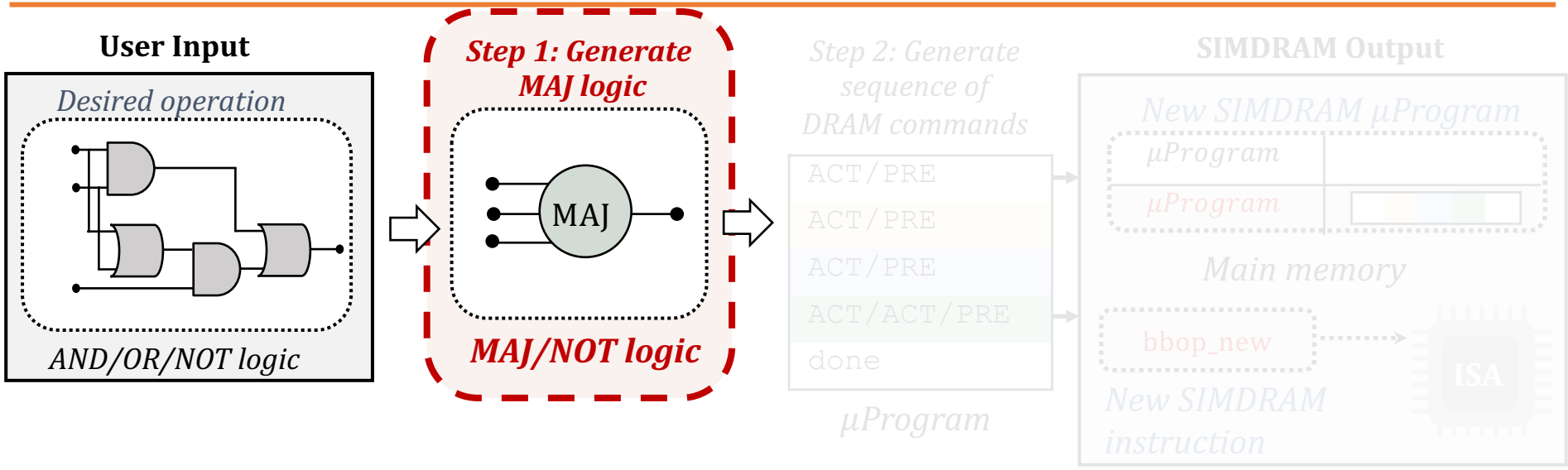
Pros compared to AND/OR/NOT-based computation:

- Higher performance
- Higher throughput
- Lower energy consumption

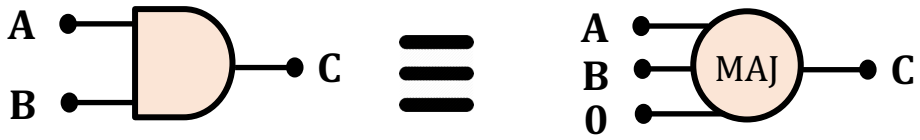
SIMDRAM Framework



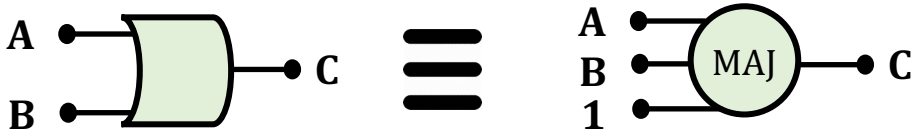
SIMDRAM Framework: Step 1



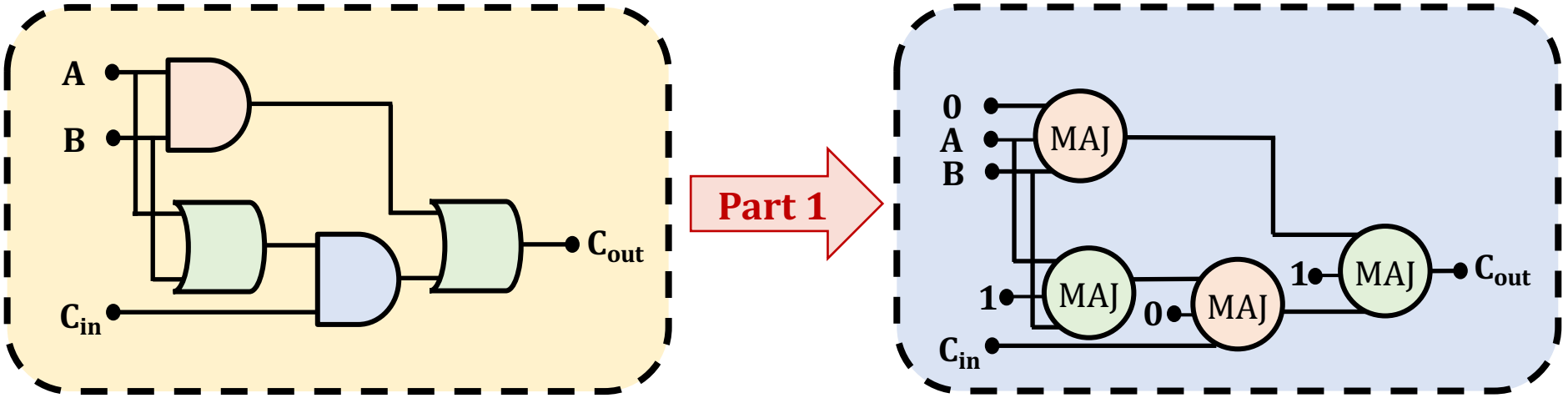
Step 1: Naïve MAJ/NOT Implementation



output is “1” only when A = B = “1”

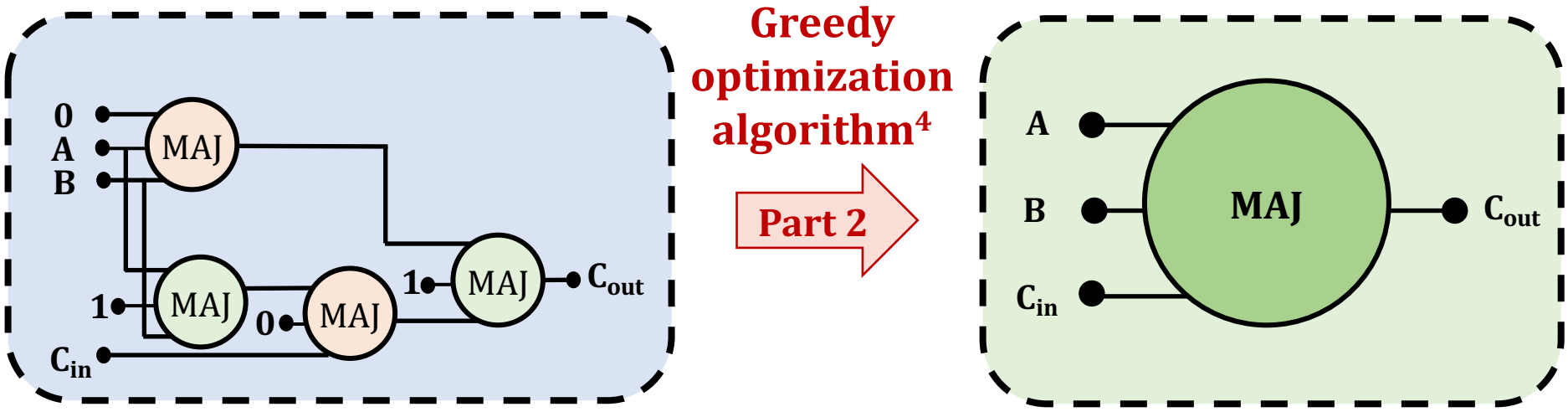


output is “0” only when A = B = “0”



Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit

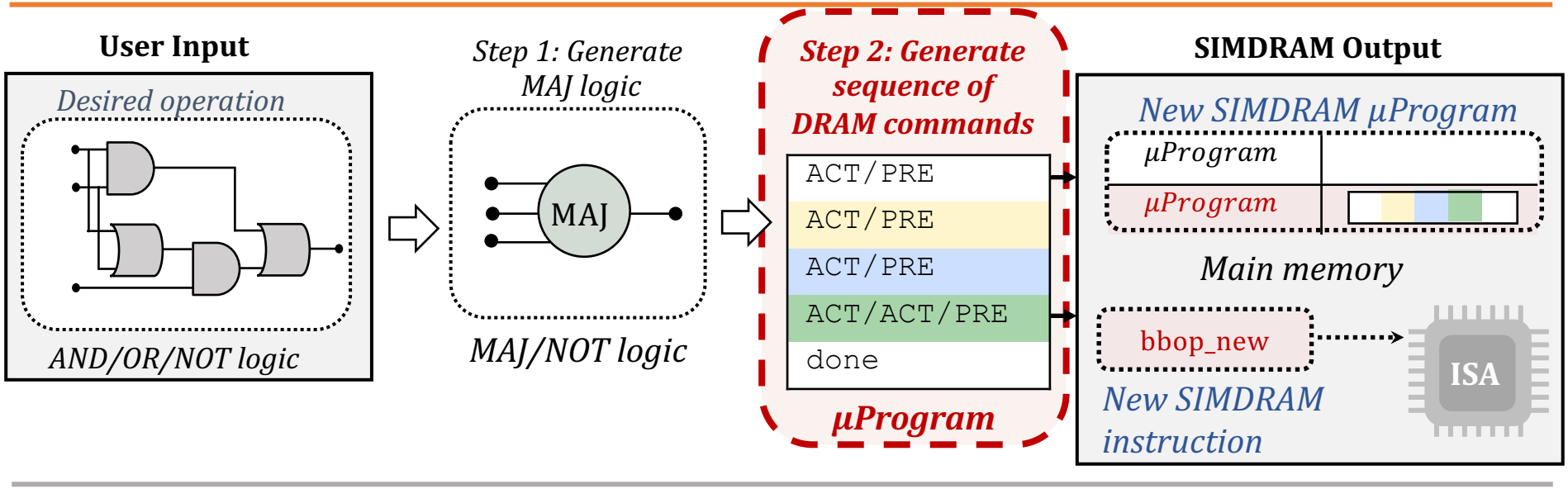
Step 1: Efficient MAJ/NOT Implementation



Step 1 generates an optimized MAJ/NOT-implementation of the desired operation

[4] L. Amarù et al, "*Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization*", DAC, 2014.

SIMDRAM Framework: Step 2



Step 2: µProgram Generation

- **µProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

- **Goal of Step 2:** To generate the µProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate µProgram

Step 2: μProgram Generation

- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

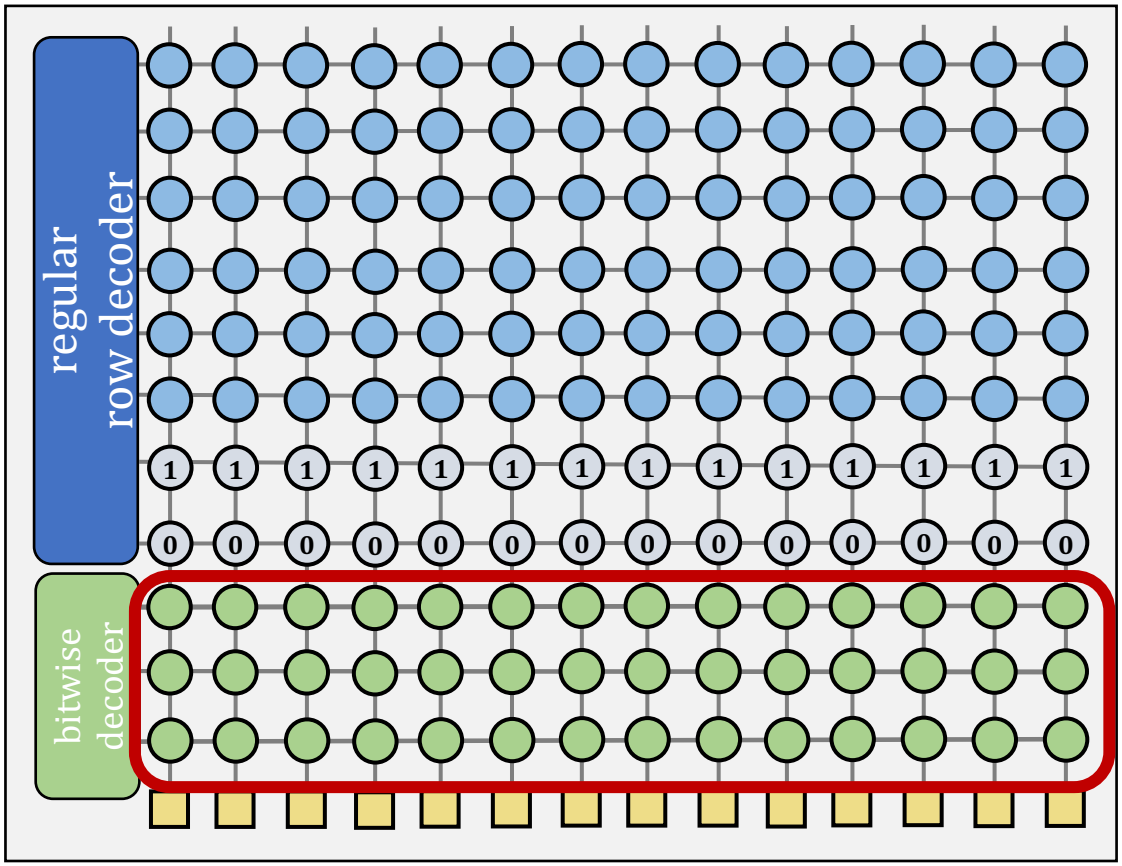
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM



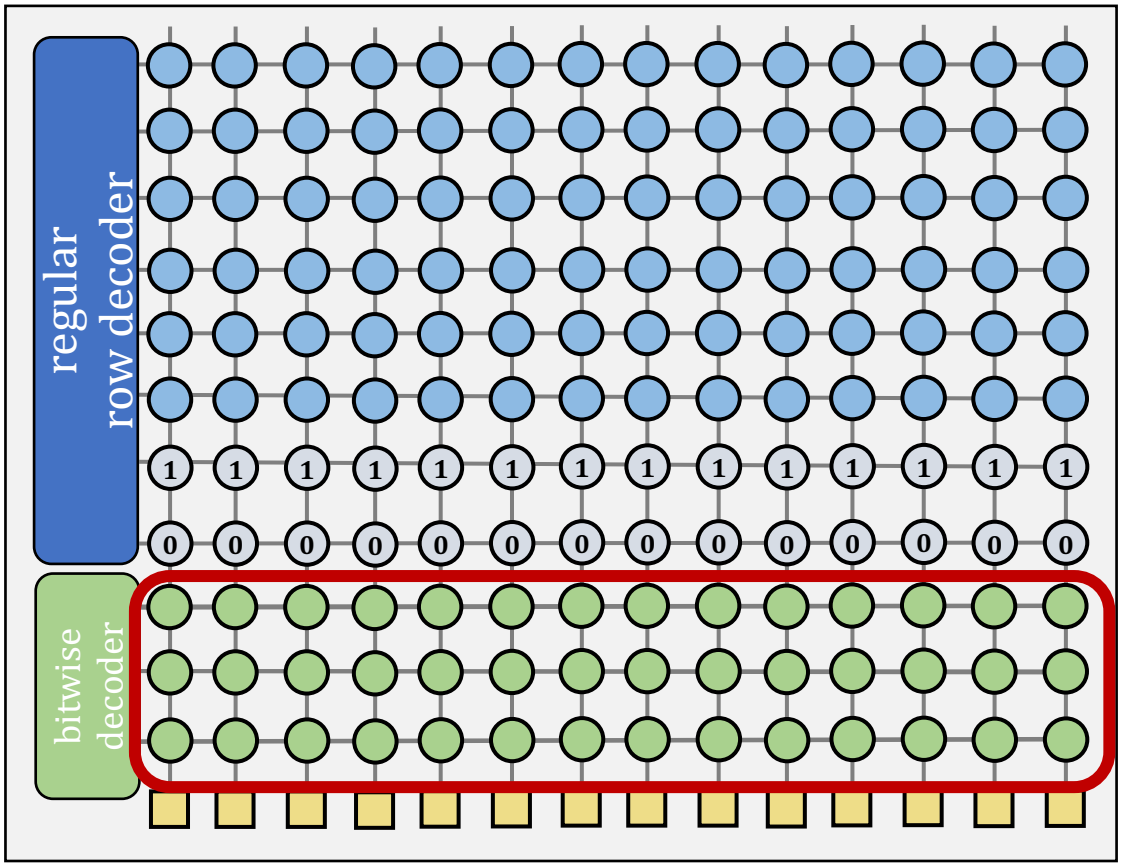
subarray organization

Constraint 1:
Limited number of rows reserved for computation

Compute rows

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM



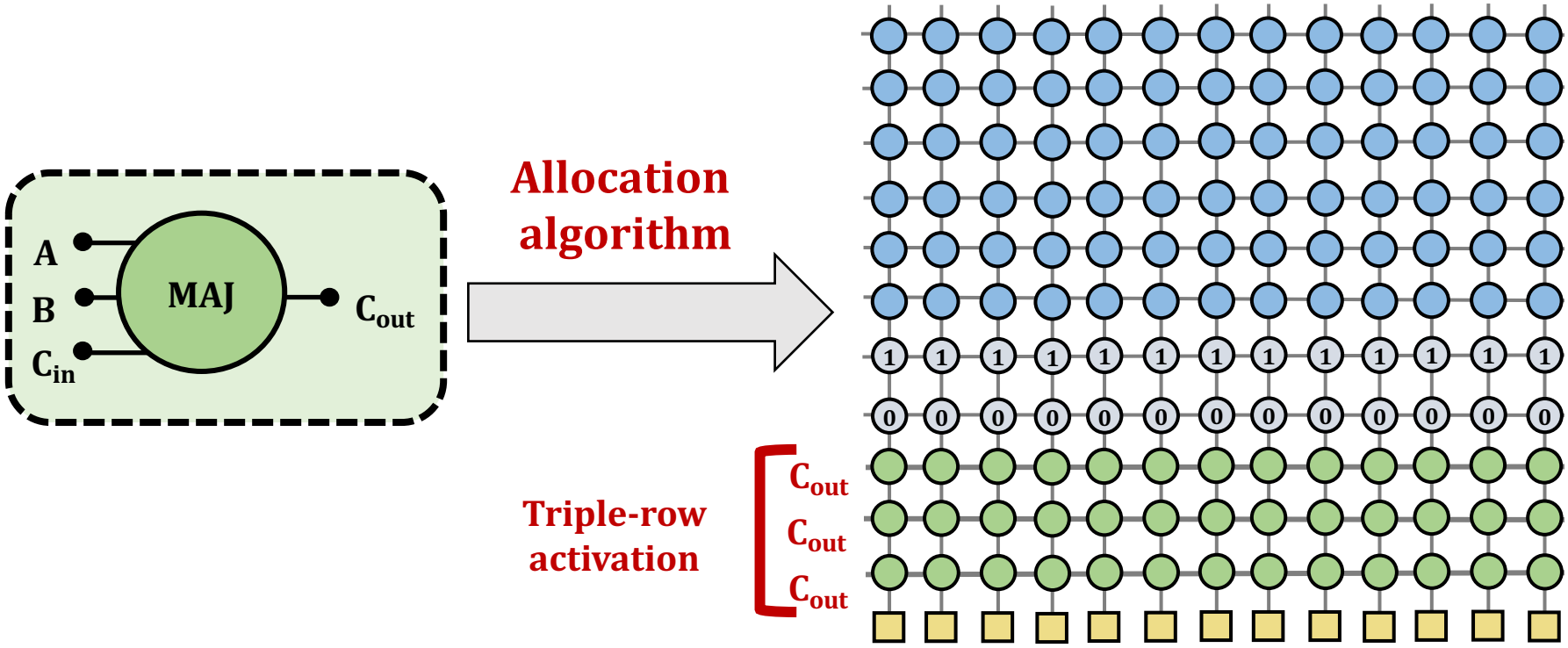
subarray organization

Constraint 2:
Destructive behavior of triple-row activation

Overwritten with MAJ output

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm:
  - Assigns as many inputs as the number of free compute rows
  - All three input rows contain the MAJ output and can be reused



Step 2: μProgram Generation

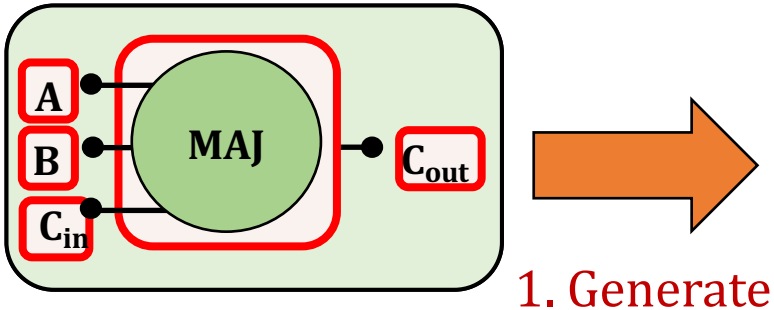
- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

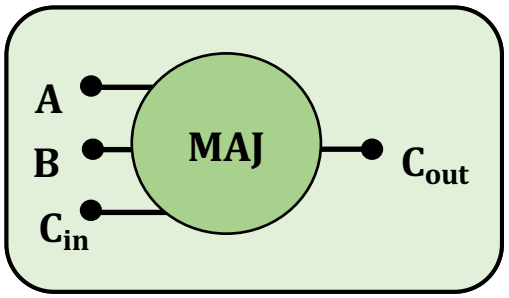
Task 2: Generate μProgram

Task 2: Generate an initial μProgram



1. Generate μProgram

Task 2: Optimize the µProgram



1. Generate µProgram

Initial µProgram

1. Copy A to reserved row **(ACT/ACT/PRE)**

2. Copy B to reserved row **(ACT/ACT/PRE)**

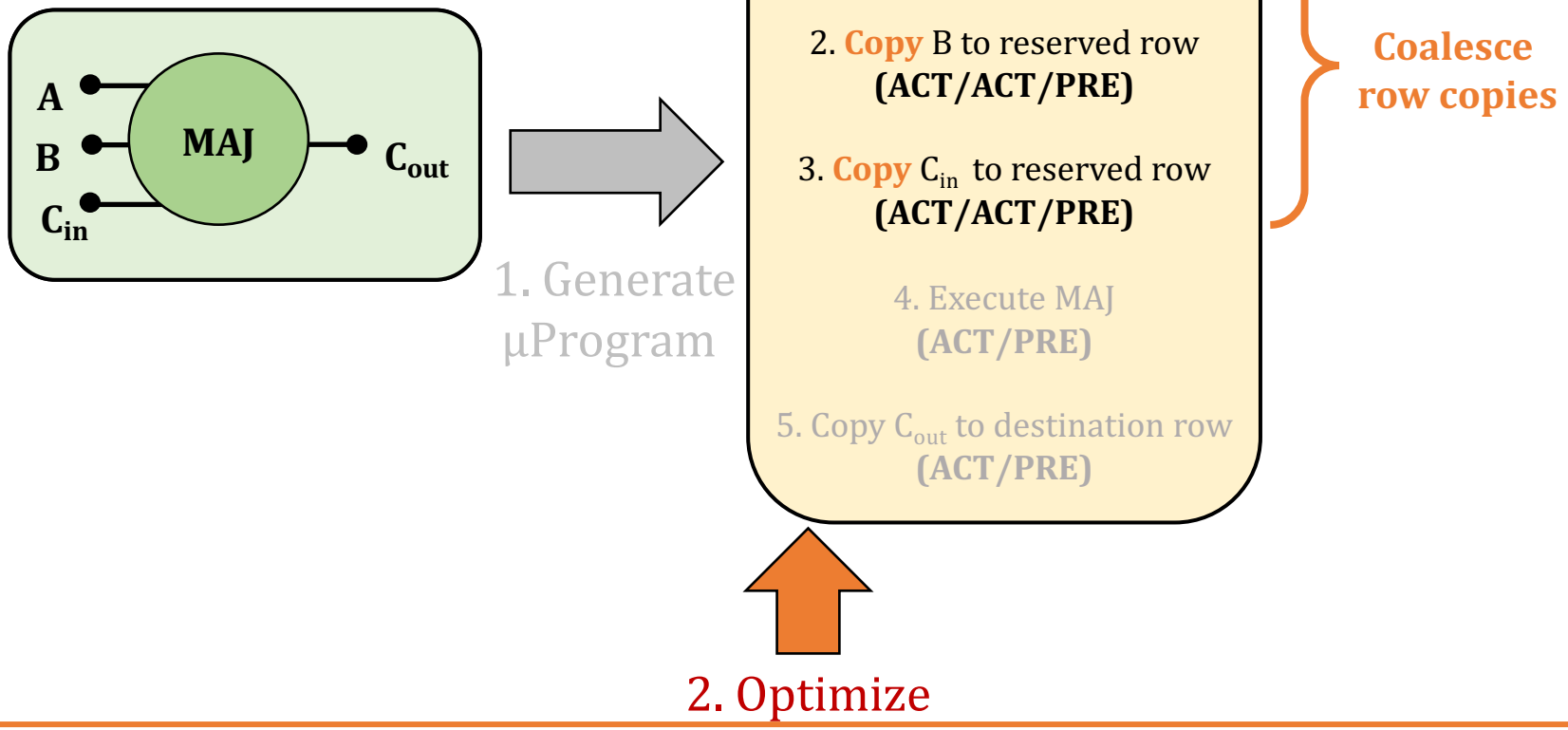
3. Copy C_{in} to reserved row **(ACT/ACT/PRE)**

4. Execute MAJ **(ACT/PRE)**

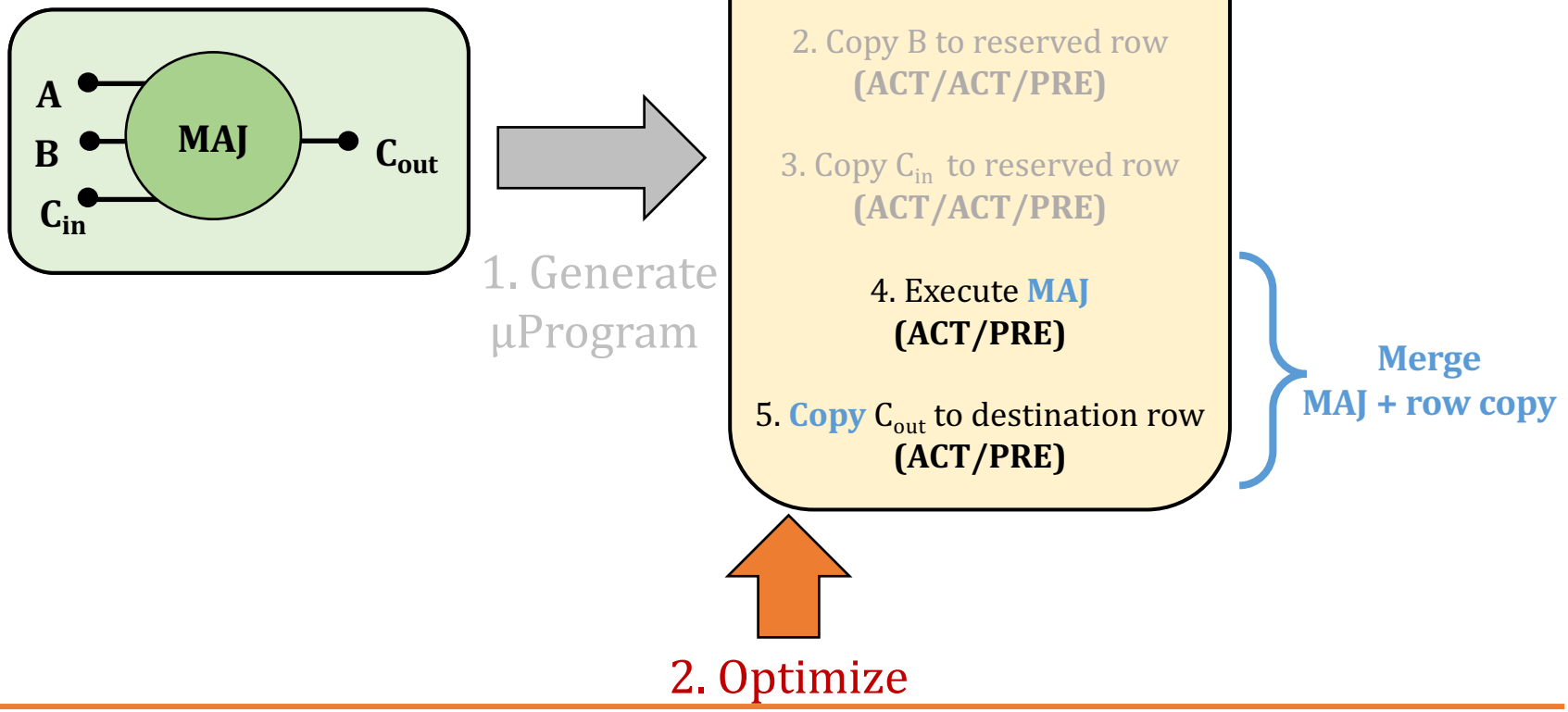
5. Copy C_{out} to destination row **(ACT/PRE)**

2. Optimize

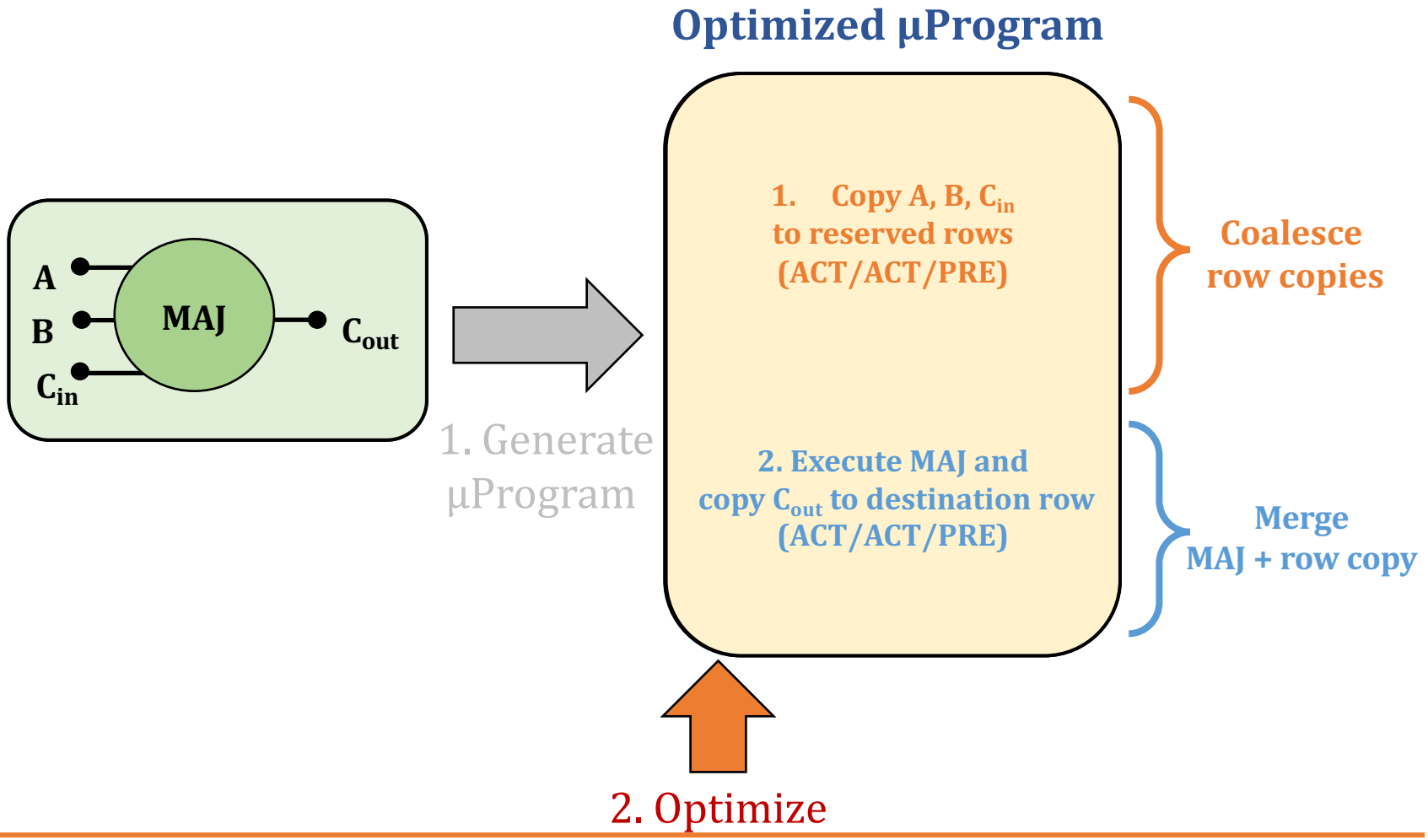
Task 2: Optimize the µProgram



Task 2: Optimize the µProgram

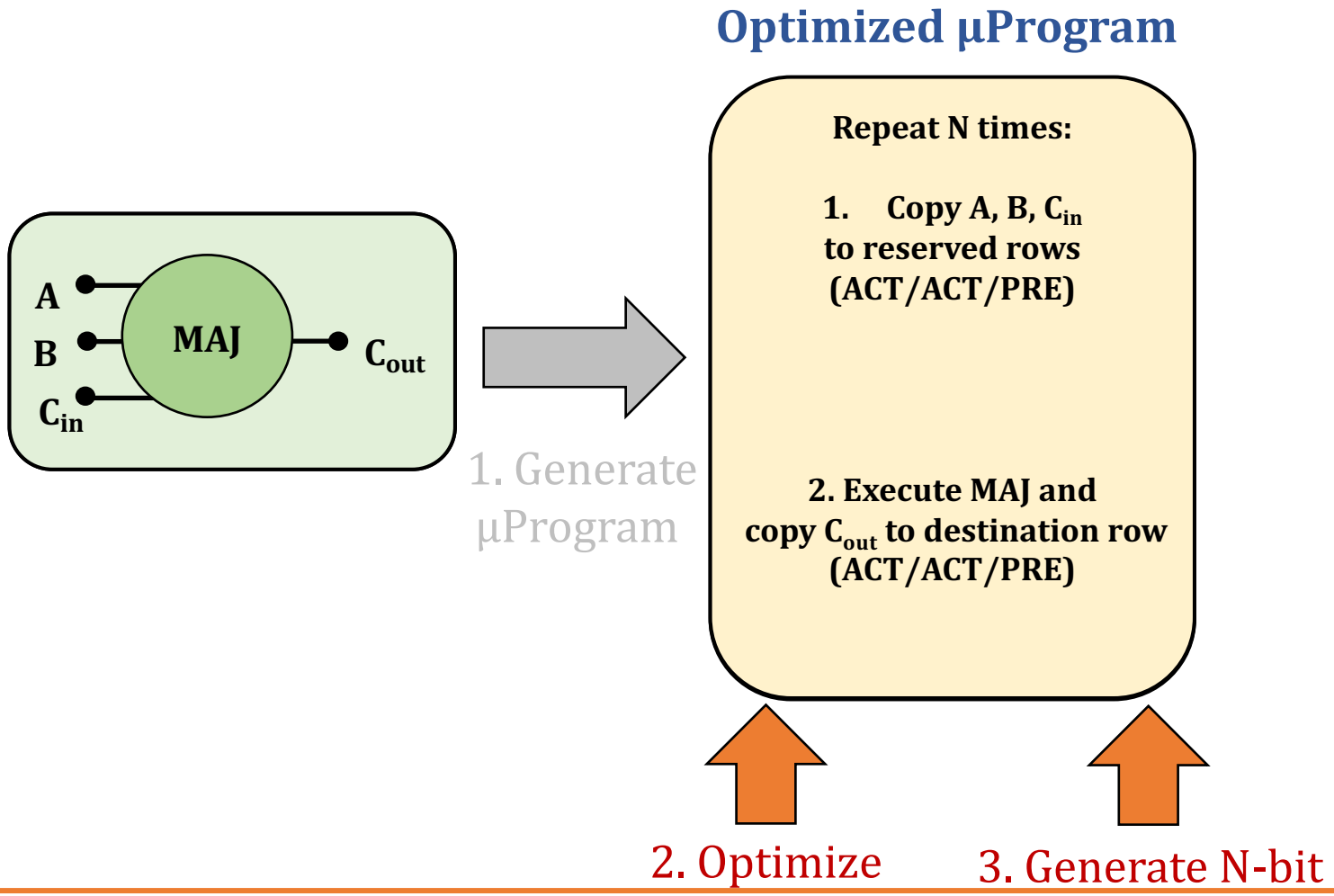


Task 2: Optimize the µProgram



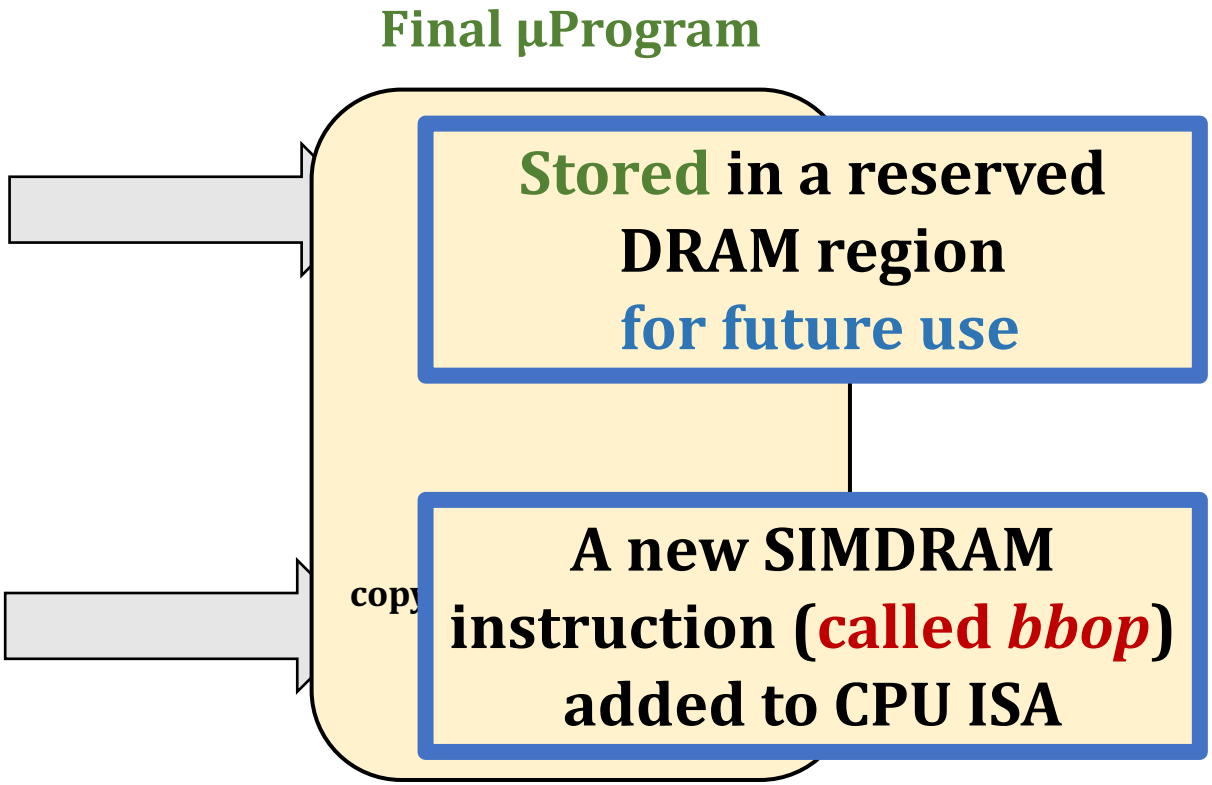
Task 2: Generate N-bit Computation

- **Final μProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion

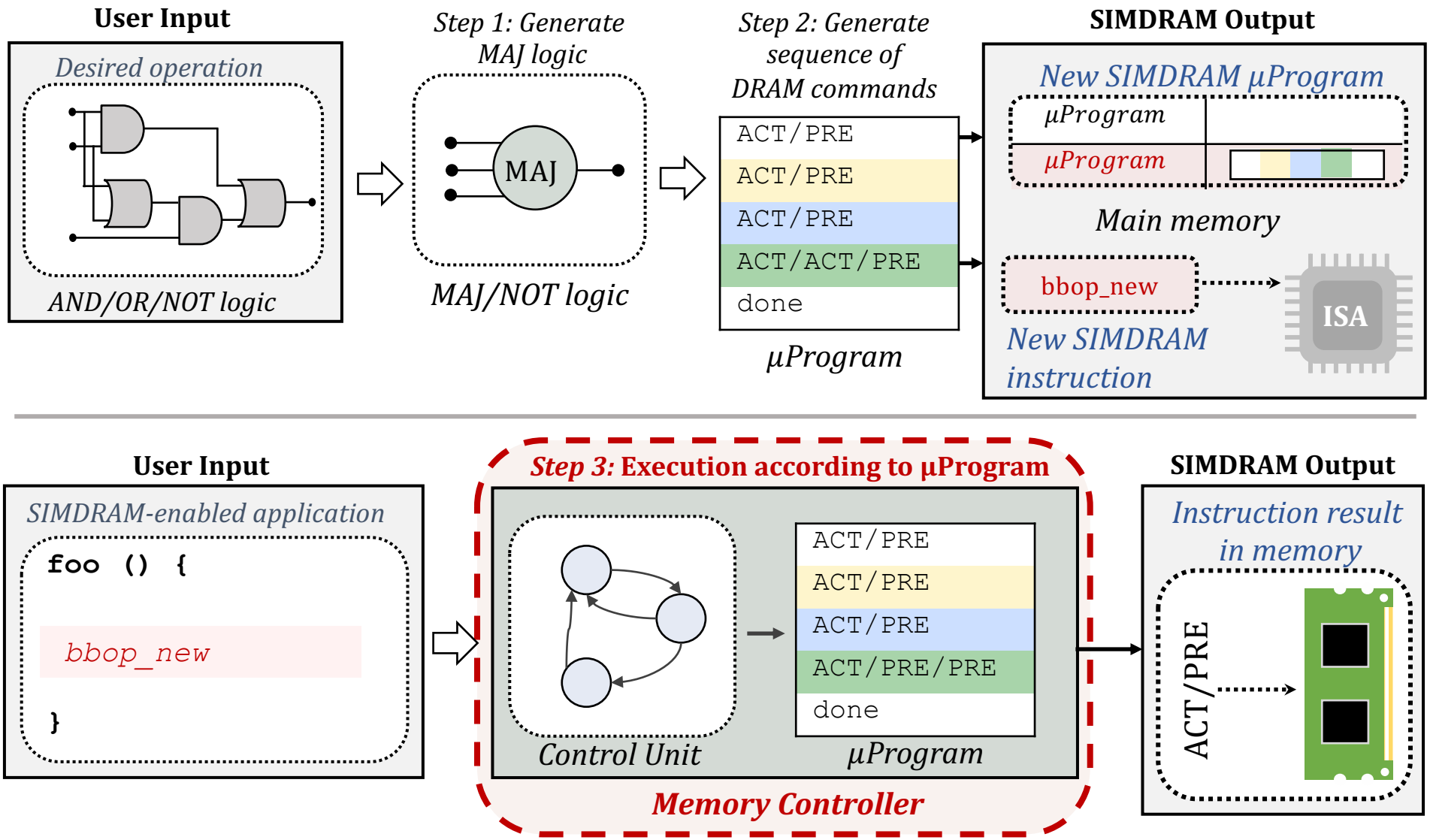


Task 2: Generate µProgram

- **Final µProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion

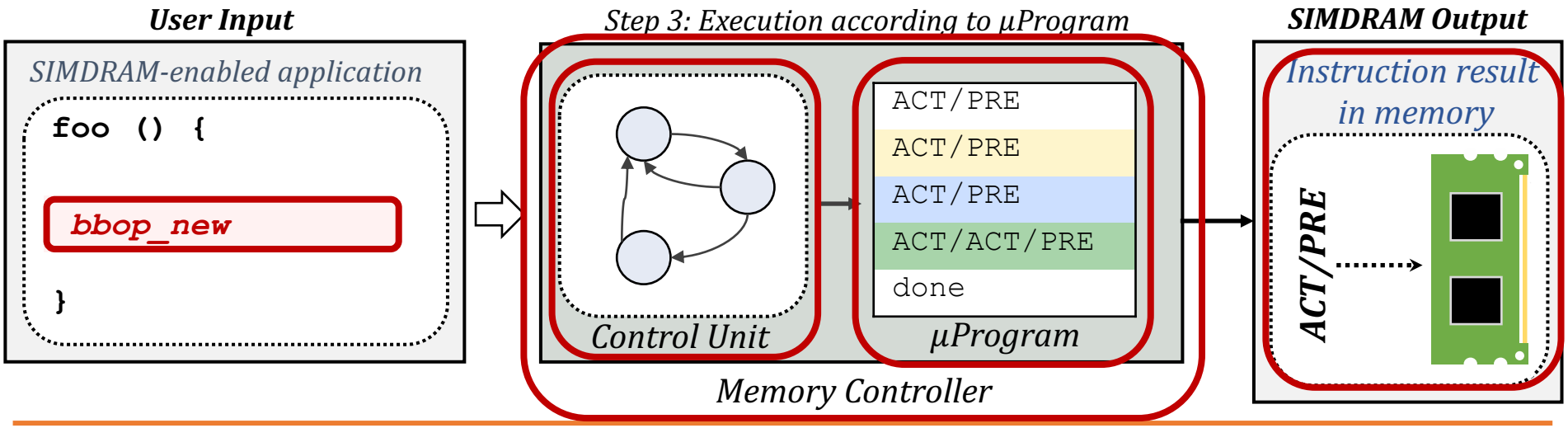


SIMDRAM Framework: Step 3



Step 3: µProgram Execution

- **SIMDRAM control unit:** handles the execution of the µProgram at runtime
- Upon receiving a **bbop instruction**, the control unit:
  1. Loads the µProgram corresponding to SIMDRAM operation
  2. Issues the sequence of DRAM commands (ACT/PRE) stored in the µProgram to SIMDRAM subarrays to perform the in-DRAM operation



System Integration

- Efficiently transposing data
- Programming interface
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework

System Integration

Efficiently transposing data

Programming interface

Handling page faults, address translation, coherence, and interrupts

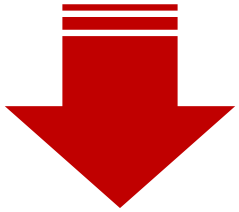
Handling limited subarray size

Security implications

Limitations of our framework

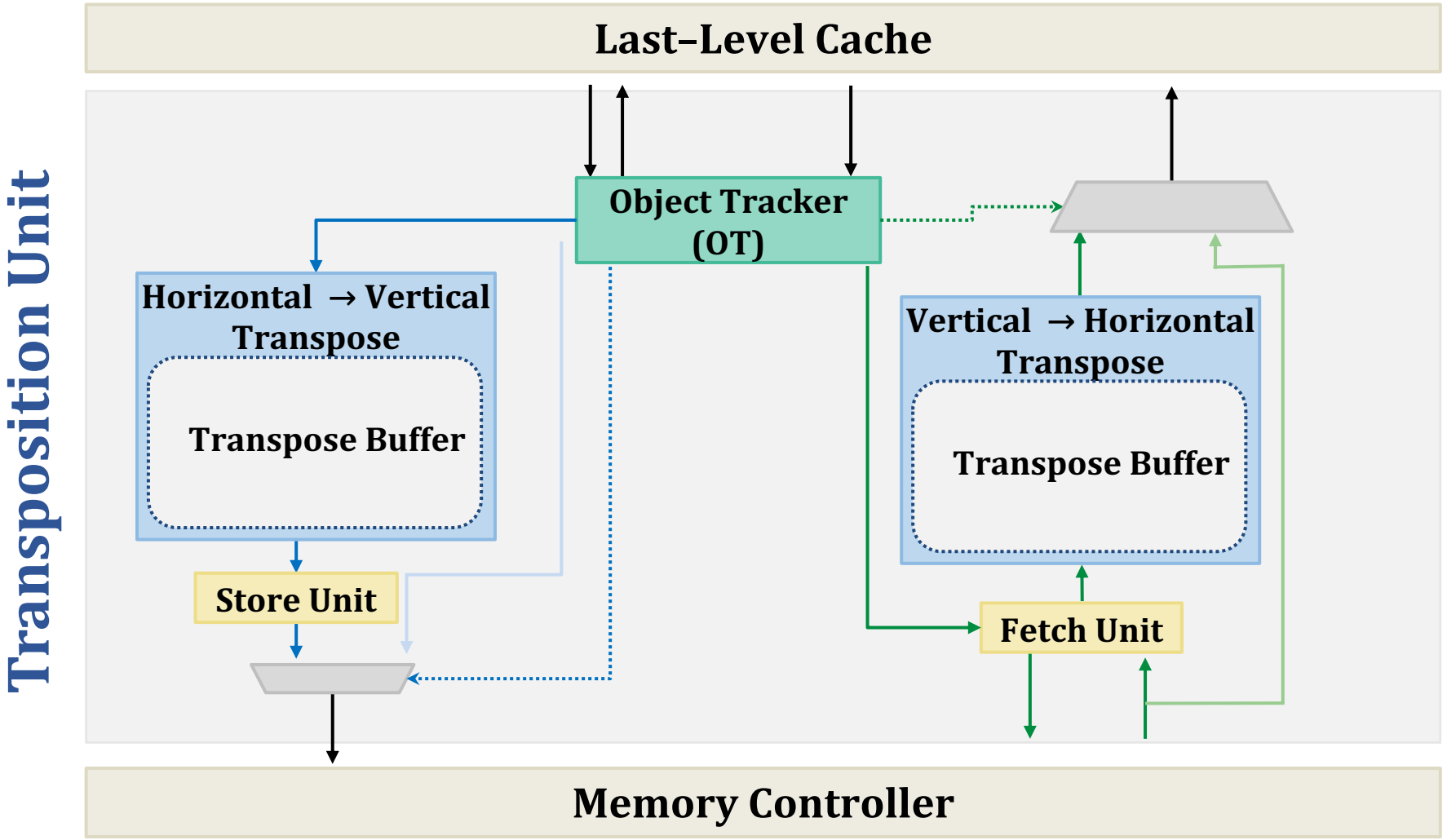
Transposing Data

- SIMDRAM operates on vertically-laid-out data
- Other system components expect data to be laid out horizontally

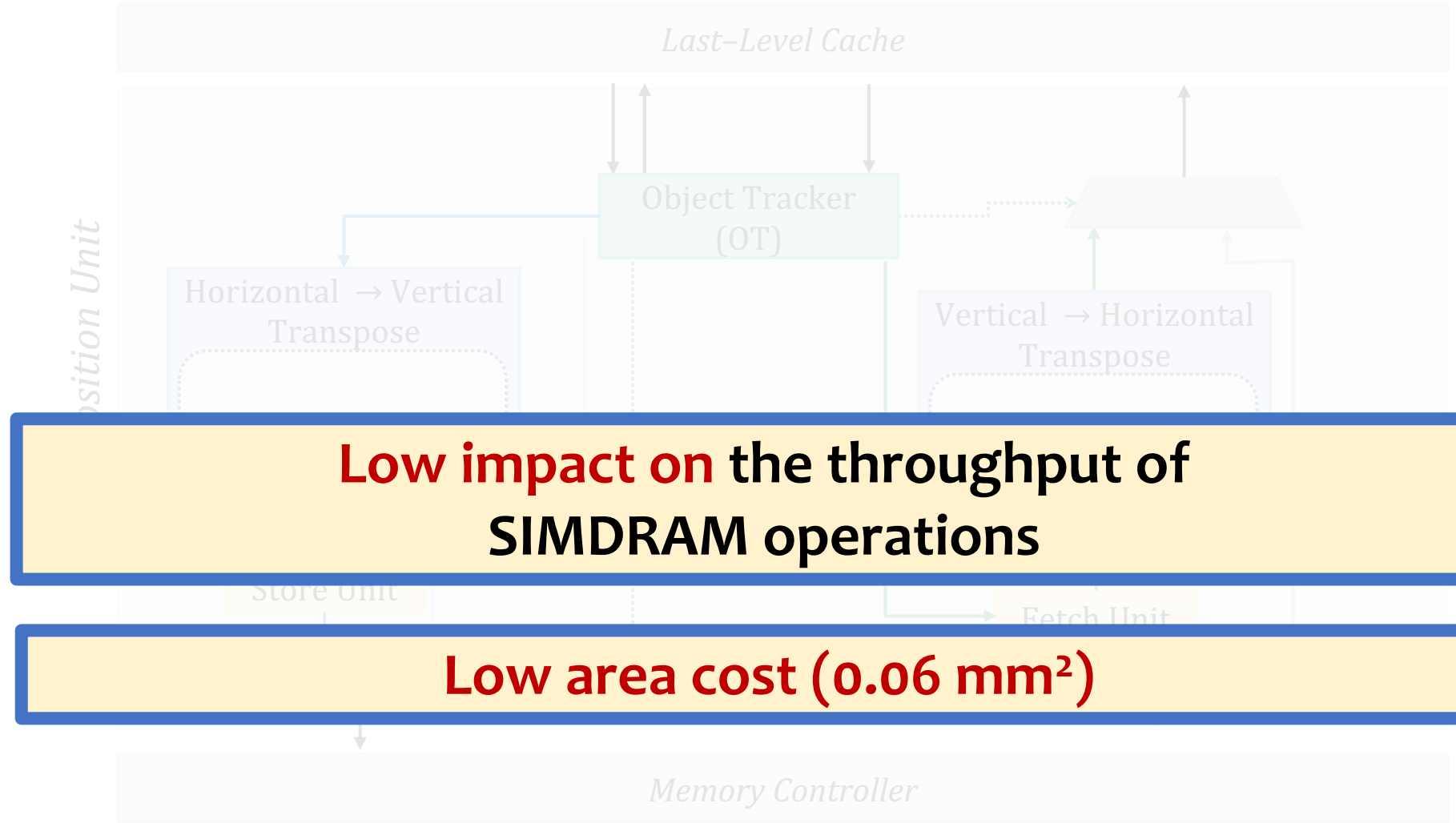


Challenging to share data between SIMDRAM and CPU

Transposition Unit



Efficiently Transposing Data



Low impact on the throughput of SIMDRAM operations

Low area cost (0.06 mm²)

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`
2-Input Operation	`bbop_op dst, src_1, src_2, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`
2-Input Operation	`bbop_op dst, src_1, src_2, size, n`
Predication	`bbop_if_else dst, src_1, src_2, select, size, n`

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

More in the Paper

- Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, Joao Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gomez-Luna, and Onur Mutlu,
  "SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM"
  Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***), Virtual, 2021
  [2-page Extended Abstract]
  [Short Talk Slides (pptx) (pdf)]
  [Talk Slides (pptx) (pdf)]
  [Short Talk Video (5 mins)]
  [Full Talk Video (27 mins)]

SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

*Nastaran Hajinazar[1,2] *Geraldo F. Oliveira[1] Sven Gregorio[1] João Dinis Ferreira[1]
Nika Mansouri Ghiasi[1] Minesh Patel[1] Mohammed Alser[1] Saugata Ghose[3]
Juan Gómez-Luna[1] Onur Mutlu[1]

[1]ETH Zürich [2]Simon Fraser University [3]University of Illinois at Urbana–Champaign

Methodology: Experimental Setup

- **Simulator:** gem5

- **Baselines:**
  - A multi-core CPU (Intel Skylake)
  - A high-end GPU (NVidia Titan V)
  - Ambit: a state-of-the-art in-memory computing mechanism

- **Evaluated SIMDRAM configurations** (all using a DDR4 device):
  - **1-bank:** SIMDRAM exploits 65'536 SIMD lanes (an 8 kB row buffer)
  - **4-banks:** SIMDRAM exploits 262'144 SIMD lanes
  - **16-banks:** SIMDRAM exploits 1'048'576 SIMD lanes

Methodology: Workloads

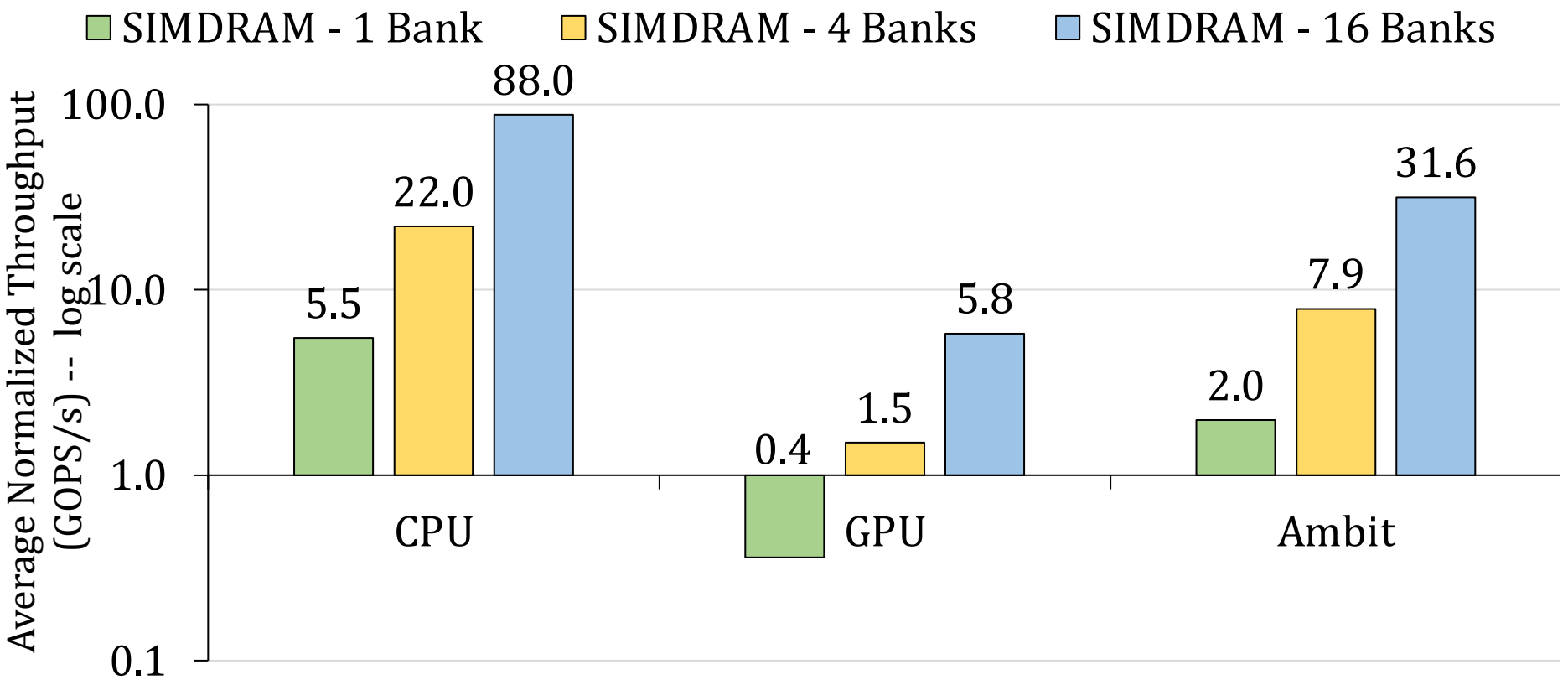
Evaluated:

- 16 complex in-DRAM operations:
  - Absolute
  - Addition/Subtraction
  - BitCount
  - Equality/ Greater/Greater Equal
  - Predication
  - ReLU
  - AND-/OR-/XOR-Reduction
  - Division/Multiplication

- 7 real-world applications
  - BitWeaving (databases)
  - TPH-H (databases)
  - kNN (machine learning)
  - LeNET (neural networks)
  - VGG-13/VGG-16 (neural networks)
  - Brightness (graphics)

Throughput Analysis

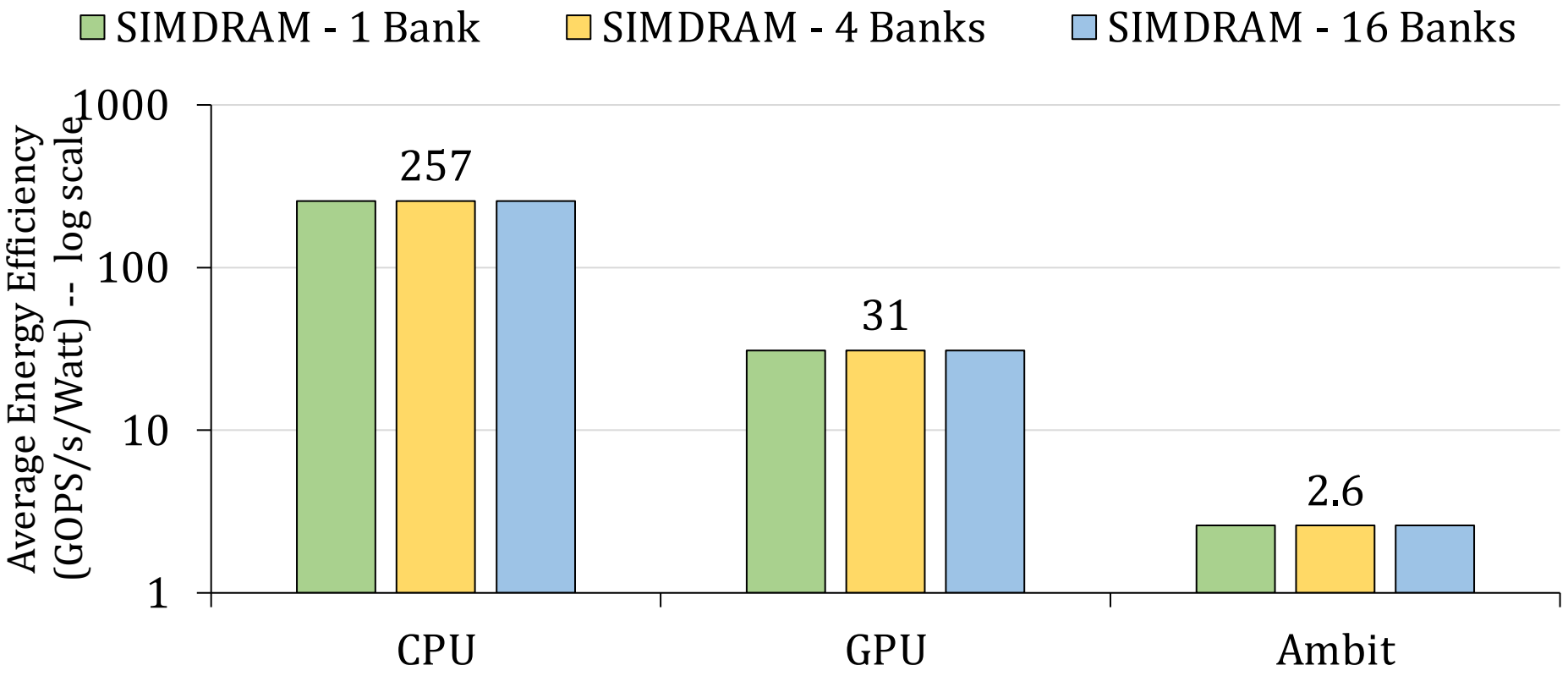
Average normalized throughput across all 16 SIMDRAM operations



SIMDRAM significantly outperforms all state-of-the-art baselines for a wide range of operations

Energy Analysis

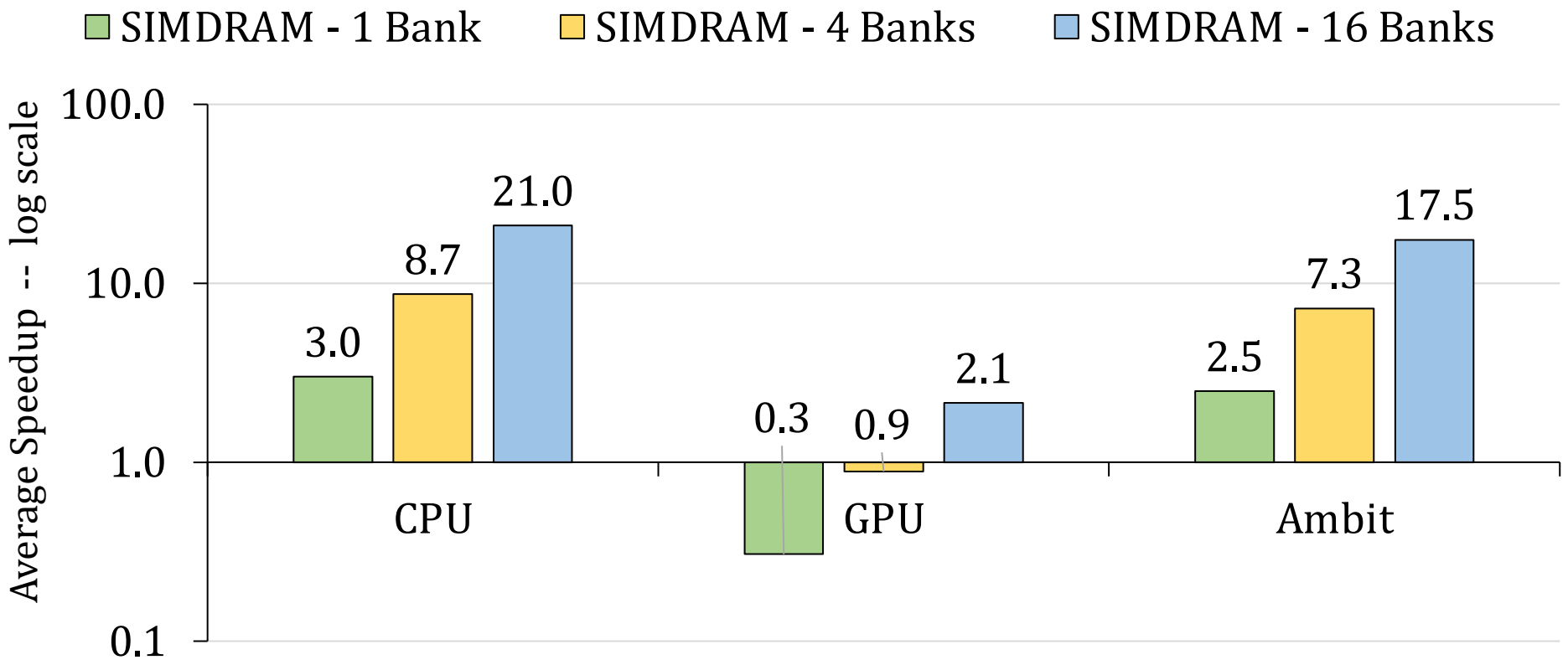
Average normalized energy efficiency across all 16 SIMDRAM operations



SIMDRAM is more energy-efficient than all state-of-the-art baselines for a wide range of operations

Real-World Application

Average speedup across 7 real-world applications



SIMDRAM effectively and efficiently accelerates many commonly-used real-world applications

More in the Paper

- **Evaluation:**
  - Reliability
  - Data movement overhead
  - Data transposition overhead
  - Area overhead
  - Comparison to in-cache computing

SIMDRAM: Conclusion

- **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  1. Efficiently computing complex operations
  2. Providing the ability to implement arbitrary operations as required
  3. Using a massively-parallel in-DRAM SIMD substrate
- **Key Results**: SIMDRAM provides:
  - 88x and 5.8x the throughput and 257x and 31x the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
  - 21x and 2.1x the performance of the CPU and GPU over seven real-world applications
- **Conclusion**: SIMDRAM is a promising PuM framework
  - Can ease the adoption of processing-using-DRAM architectures
  - Improve the performance and efficiency of processing-using-DRAM architectures

Is PUM in DRAM Really Feasible?

RowClone in Off-the-Shelf DRAM Chips

- Idea: Violate DRAM timing parameters to mimic RowClone

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

RowClone & Bitwise Ops in Real DRAM Chips

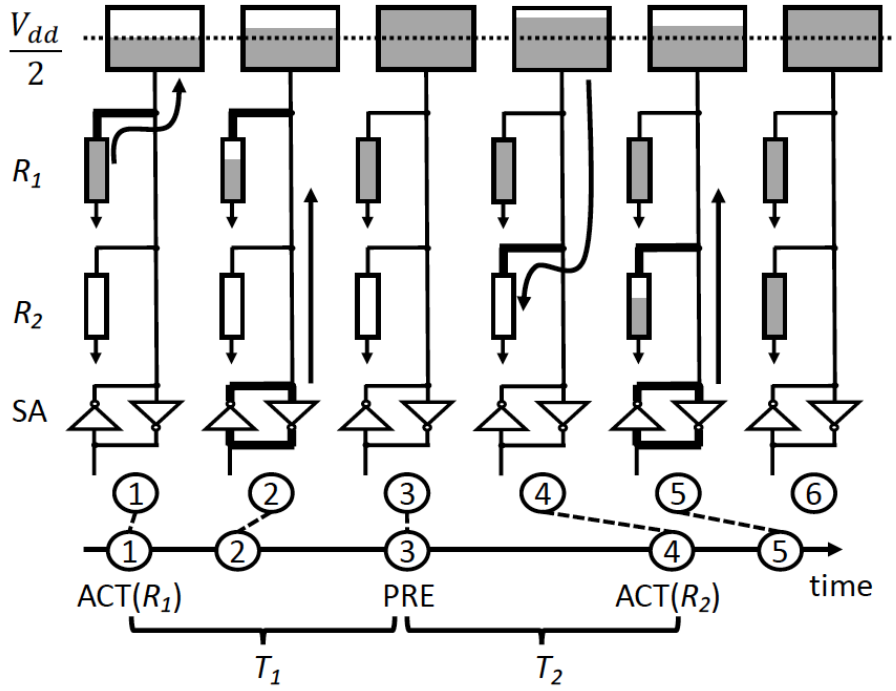


Figure 4: Timeline for a single bit of a column in a row copy operation. The data in R_1 is loaded to the bit-line, and overwrites R_2.

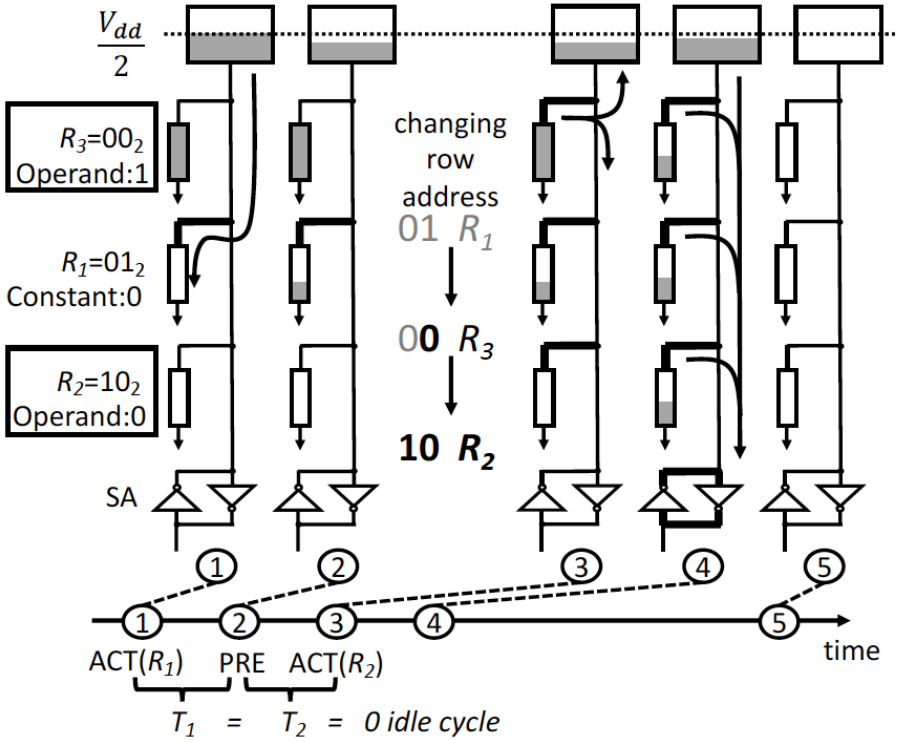
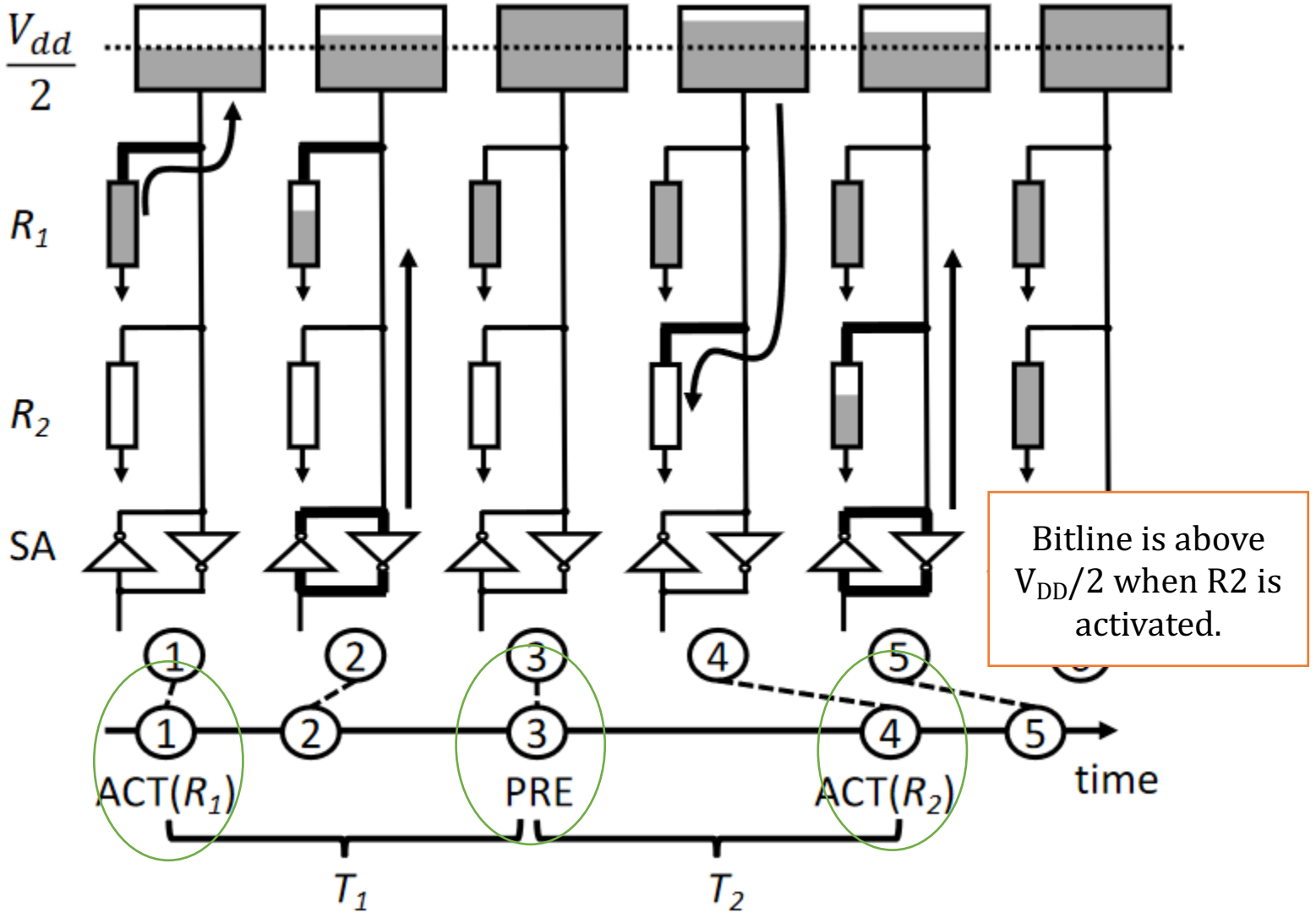
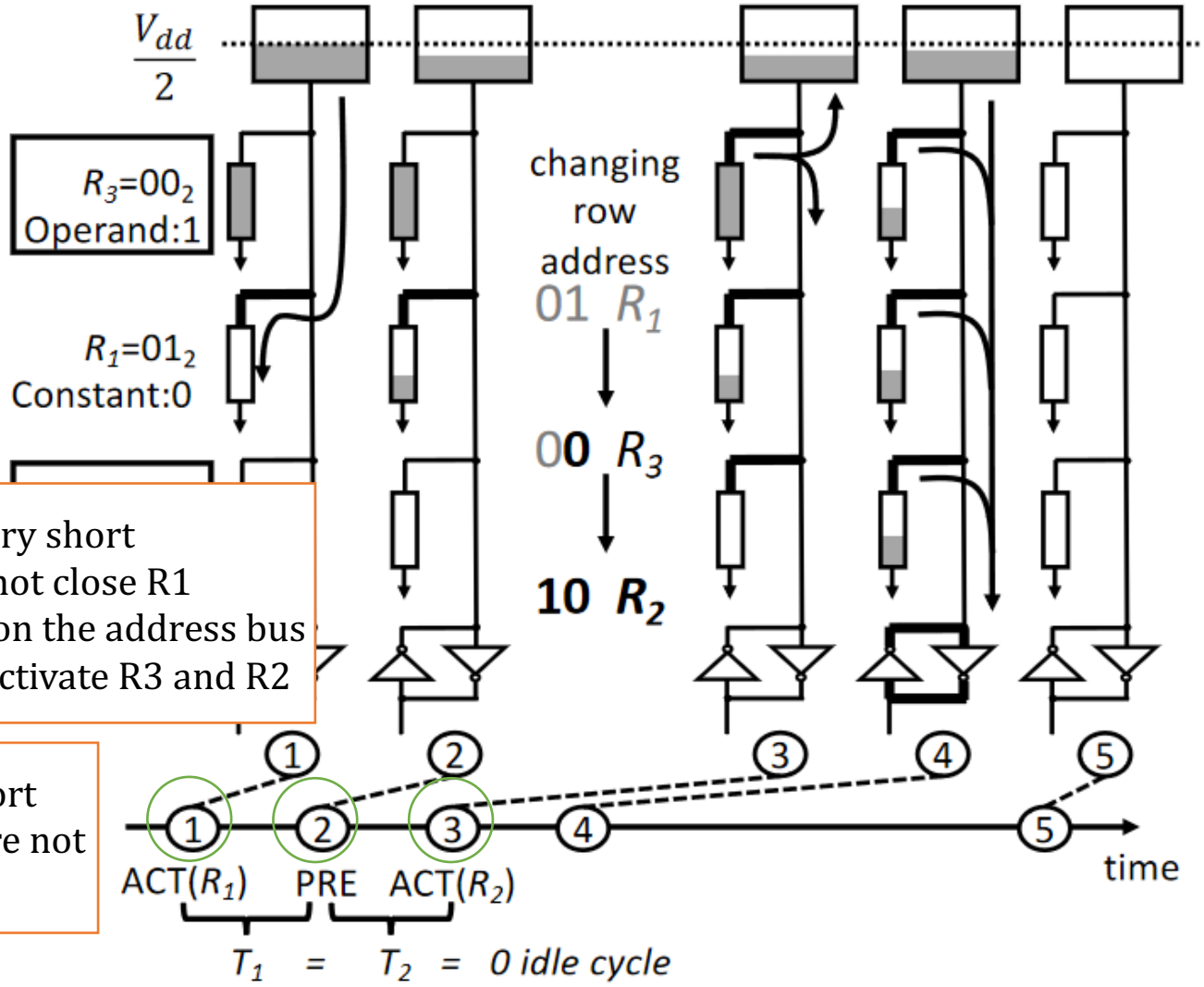


Figure 5: Logical AND in ComputeDRAM. R_1 is loaded with constant zero, and R_2 and R_3 store operands (0 and 1). The result ($0 = 1 \wedge 0$) is finally set in all three rows.

Row Copy in ComputeDRAM



Bitwise AND in ComputeDRAM



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

T1 very short
Sense amps are not activated

Experimental Methodology

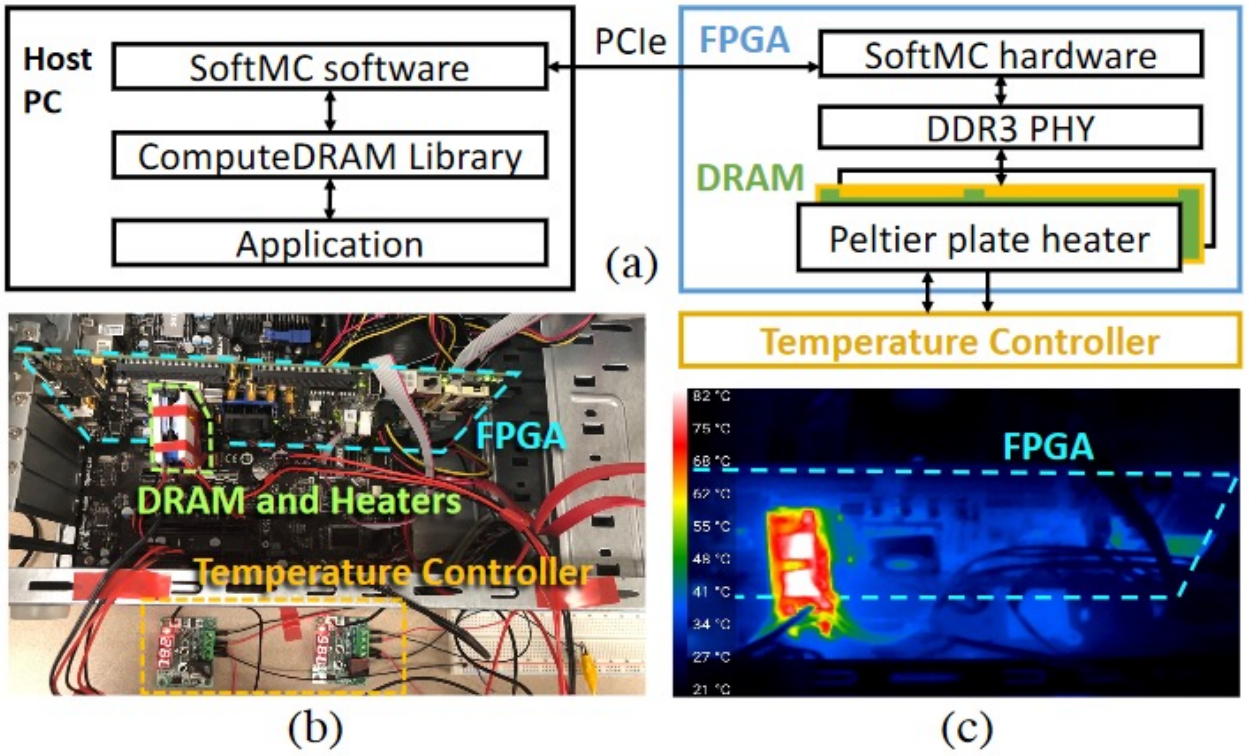


Figure 9: (a) Schematic diagram of our testing framework. (b) Picture of our testbed. (c) Thermal picture when the DRAM is heated to 80 °C.

Experimental Methodology

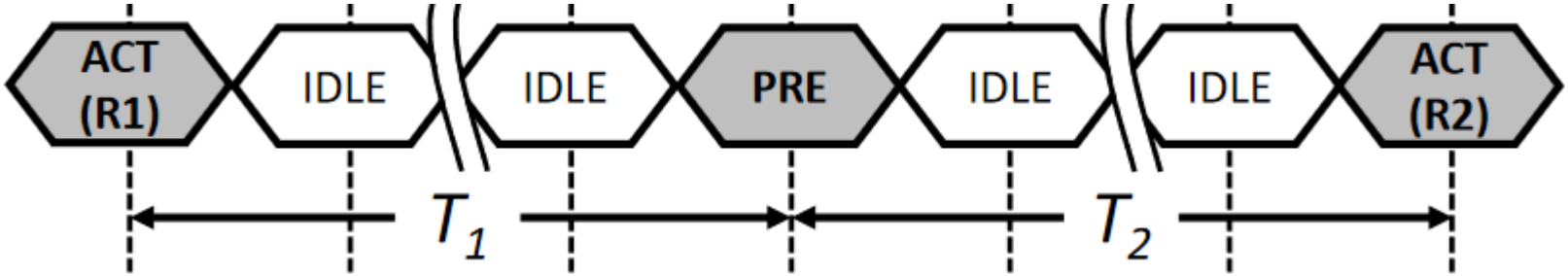
Table 1: Evaluated DRAM modules

Group ID: Vendor_Size_Freq(MHz)	Part Num	# Modules
SKhynix_2G_1333	HMT325S6BFR8C-H9	6
SKhynix_4		2
SKhynix_4		2
SKhynix_4		4
SKhynix_4		2
Samsung_4		2
Samsung_4		2
Micron_2G		2
Micron_2G		2
Elpida_2G_1333	EBJ21UE8BDS0-DJ-F	2
Nanya_4G_1333	NT4GC64B8HG0NS-CG	2
TimeTec_4G_1333	78AP10NUS2R2-4G	2
Corsair_4G_1333	CMSA8GX3M2A1333C9	2

32 DDR3 Modules
~256 DRAM Chips

Proof of Concept

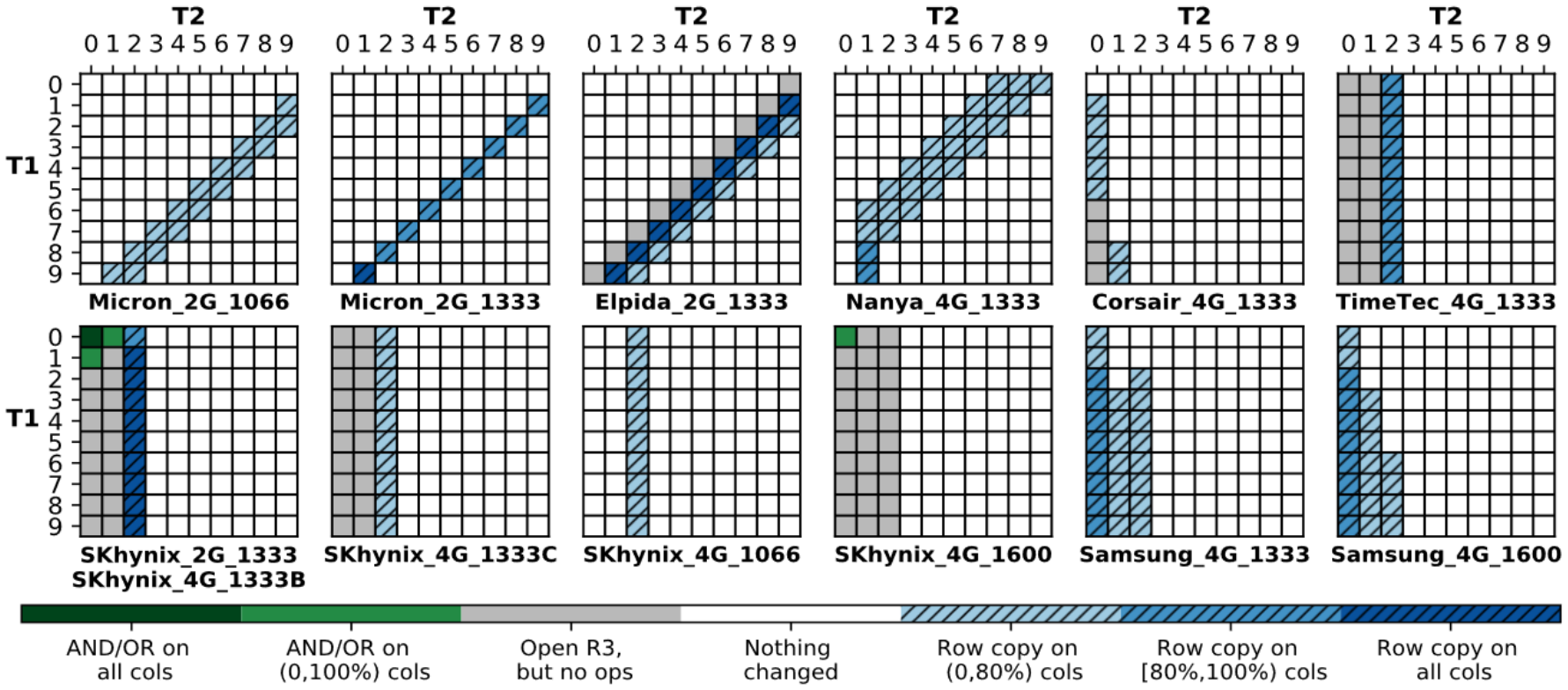
- How they test these memory modules:
  - Vary T_1 and T_2, observe what happens.



SoftMC Experiment

1. Select a random subarray
2. Fill subarray with random data
3. Issue ACT-PRE-ACTs with given T_1 & T_2
4. Read out subarray
5. Find out how many columns in a row support either operation
   - Row-wise success ratio

Proof of Concept



- Each grid represents the success ratio of operations for a specific DDR3 module.

Real Processing-Using-Memory Prototype

- End-to-end RowClone & TRNG using off-the-shelf DRAM chips
- Idea: Violate DRAM timing parameters to mimic RowClone

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 ETÜ [*]BSC

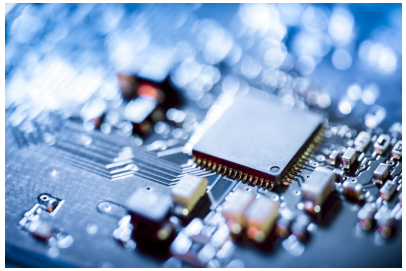
https://arxiv.org/pdf/2111.00082.pdf
https://github.com/cmu-safari/pidram
https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

PiDRAM

Goal: Develop a **flexible** platform to explore **end-to-end** implementations of PuM techniques

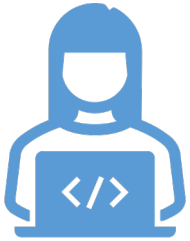
- Enable rapid integration via key components

Hardware



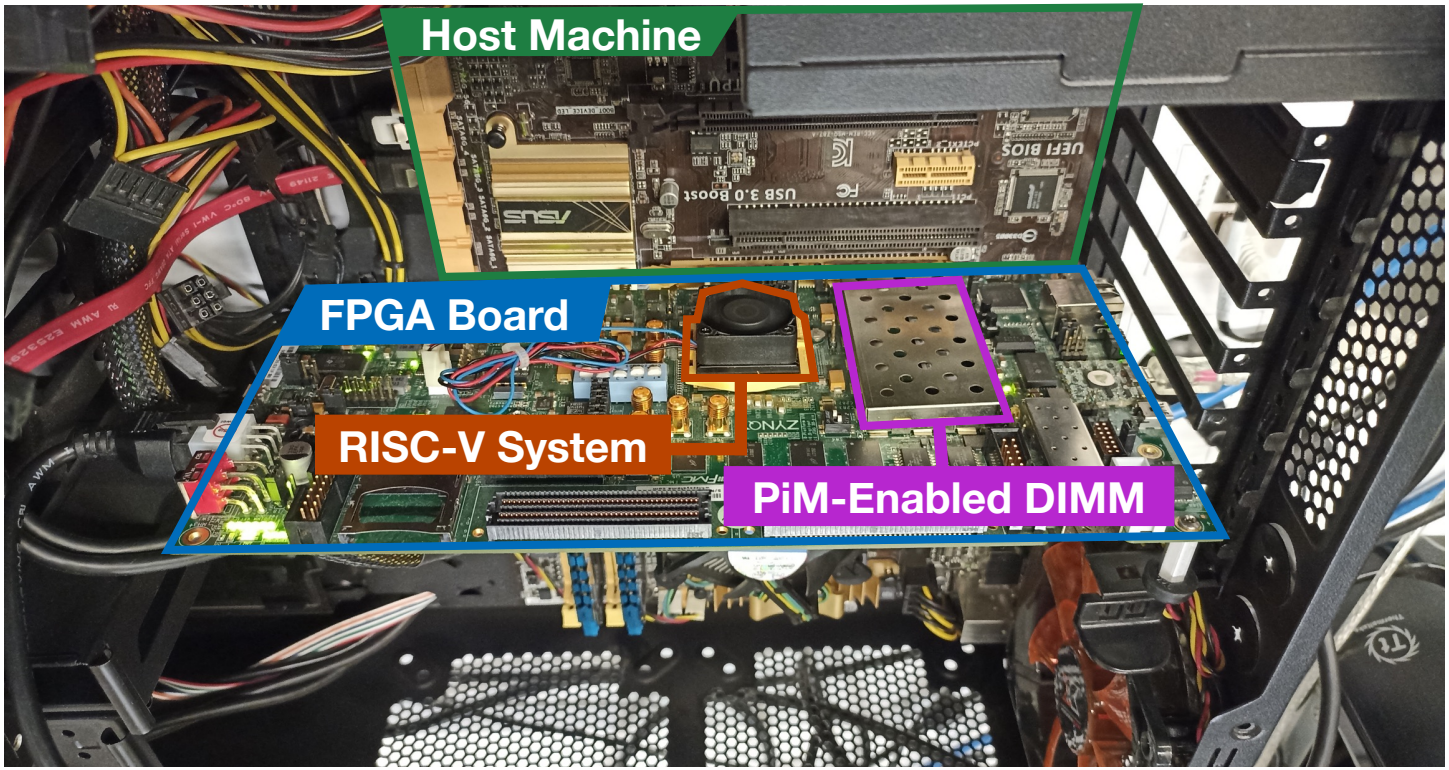
1. Easy-to-extend Memory Controller
2. ISA-transparent PuM Controller

Software



1. Extensible Software Library
2. Custom Supervisor Software

Real Processing-Using-Memory Prototype

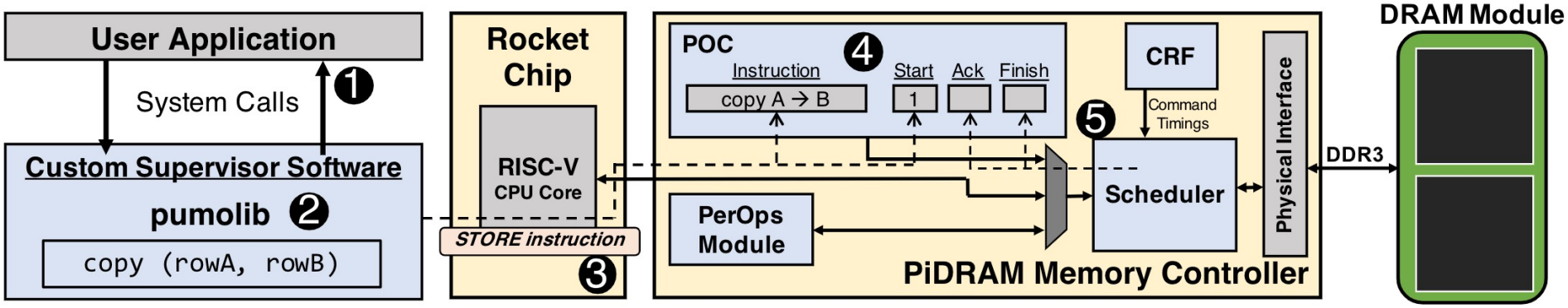


https://arxiv.org/pdf/2111.00082.pdf

https://github.com/cmu-safari/pidram

https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

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

Real Processing-Using-Memory Prototype

README.md

Building a PiDRAM Prototype

To build PiDRAM's prototype on Xilinx ZC706 boards, developers need to use the two sub-projects in this directory. `fpga-zynq` is a repository branched off of UCB-BAR's fpga-zynq repository. We use `fpga-zynq` to generate rocket chip designs that support end-to-end DRAM PuM execution. `controller-hardware` is where we keep the main Vivado project and Verilog sources for PiDRAM's memory controller and the top level system design.

Rebuilding Steps

1. Navigate into `fpga-zynq` and read the README file to understand the overall workflow of the repository
   - Follow the readme in `fpga-zynq/rocket-chip/riscv-tools` to install dependencies
2. Create the Verilog source of the rocket chip design using the `ZynqCopyFPGAConfig`
   - Navigate into zc706, then run `make rocket CONFIG=ZynqCopyFPGAConfig -j<number of cores>`
3. Copy the generated Verilog file (should be under zc706/src) and overwrite the same file in `controller-hardware/source/hdl/impl/rocket-chip`
4. Open the Vivado project in `controller-hardware/Vivado_Project` using Vivado 2016.2
5. Generate a bitstream
6. Copy the bitstream (system_top.bit) to `fpga-zynq/zc706`
7. Use the `./build_script.sh` to generate the new `boot.bin` under `fpga-images-zc706`, you can use this file to program the FPGA using the SD-Card
   - For details, follow the relevant instructions in `fpga-zynq/README.md`

You can run programs compiled with the RISC-V Toolchain supplied within the `fpga-zynq` repository. To install the toolchain, follow the instructions under `fpga-zynq/rocket-chip/riscv-tools`.

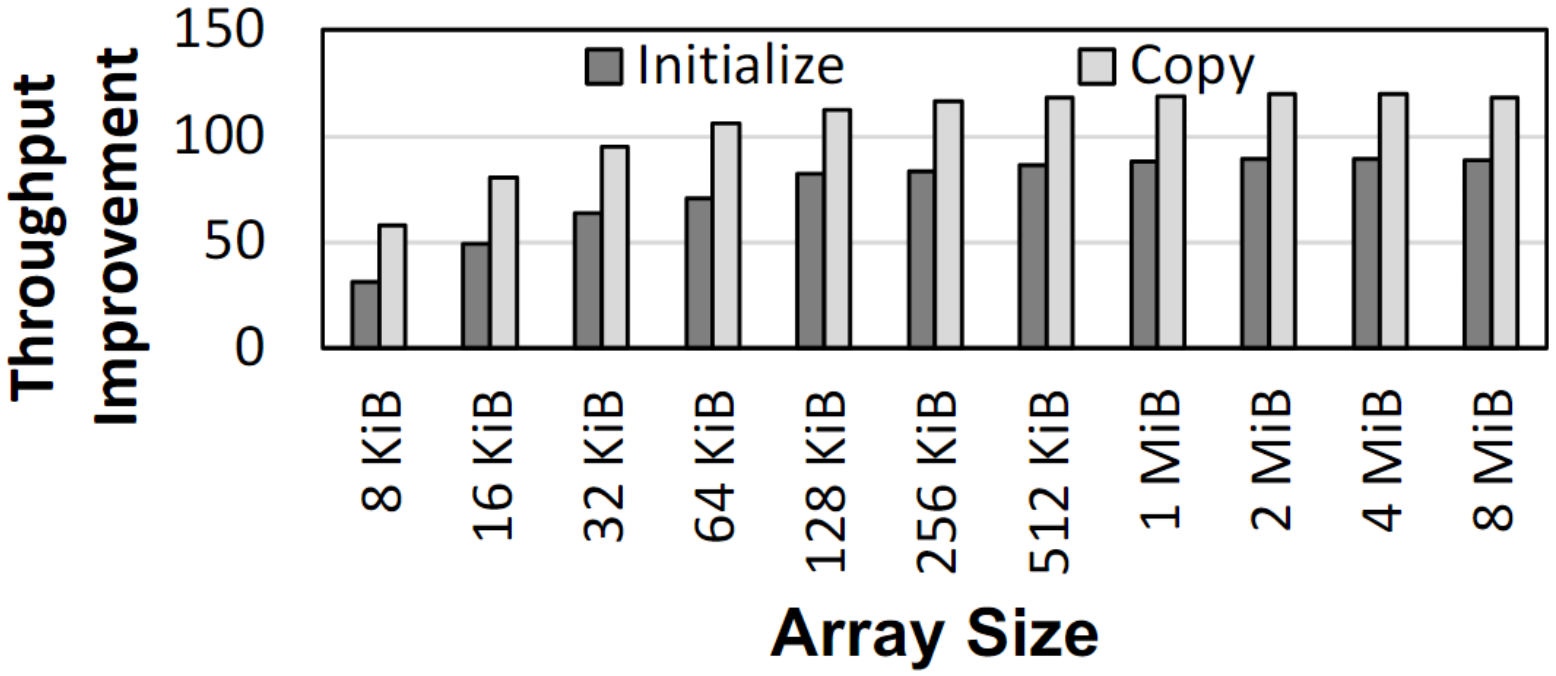
Generating DDR3 Controller IP sources

We cannot provide the sources for the Xilinx PHY IP we use in PiDRAM's memory controller due to licensing issues. We describe here how to regenerate them using Vivado 2016.2. First, you need to generate the IP RTL files:

1- Open IP Catalog
2- Find "Memory Interface Generator (MIG 7 Series)" IP and double click

https://arxiv.org/pdf/2111.00082.pdf
https://github.com/cmu-safari/pidram
https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

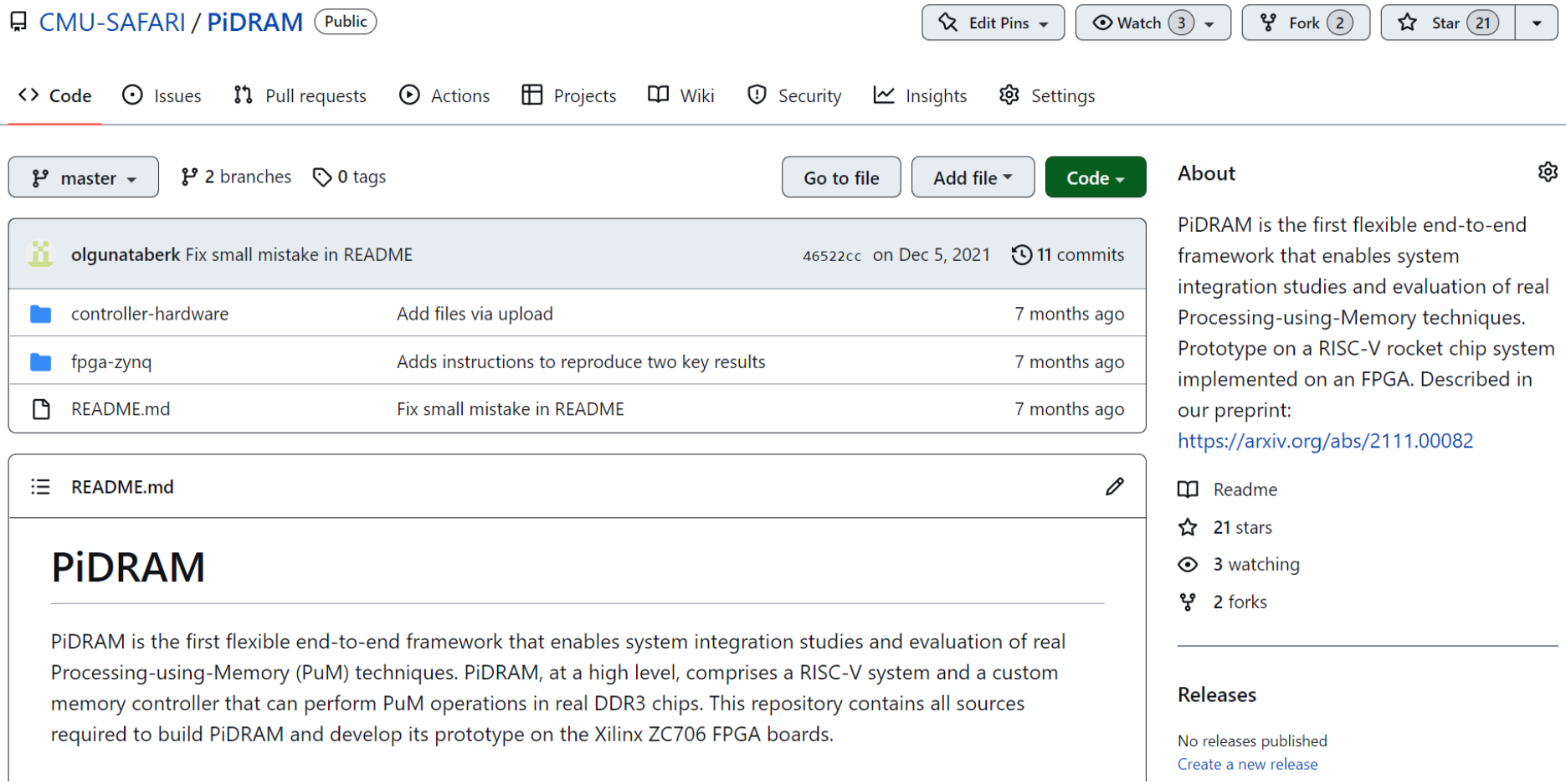
Microbenchmark Copy/Initialization Throughput



In-DRAM Copy and Initialization improve throughput by 119x and 89x

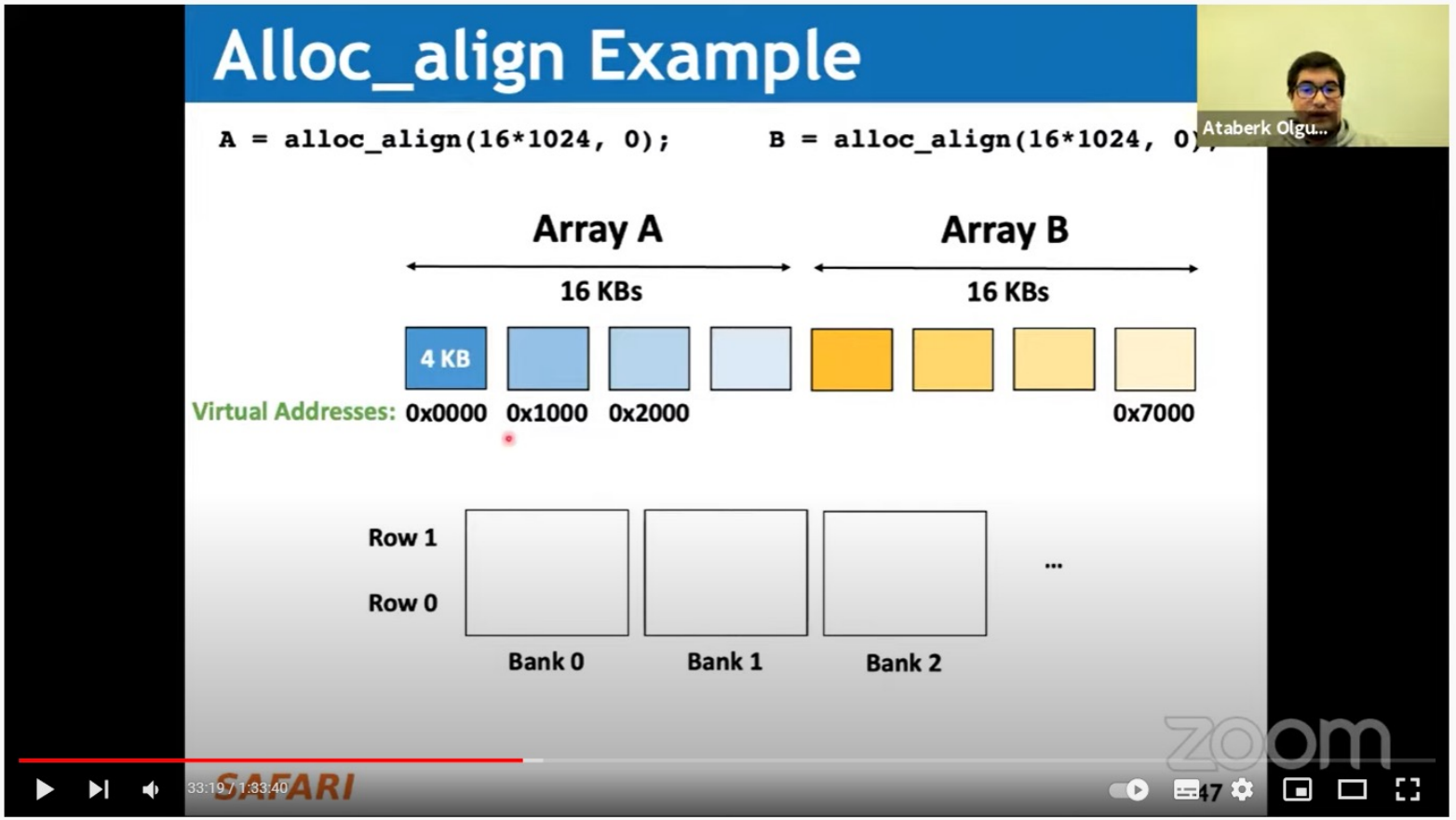
PiDRAM is Open Source

https://github.com/CMU-SAFARI/PiDRAM



PiDRAM: Long Talk and Tutorial

https://youtu.be/s_z_S6FYpC8



PUM
in Non-Volatile Memories

Pinatubo: Row Copy and Bitwise Ops in PCM

Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories

Shuangchen Li[1]*, Cong Xu[2], Qiaosha Zou[1,5], Jishen Zhao[3], Yu Lu[4], and Yuan Xie[1]

University of California, Santa Barbara[1], Hewlett Packard Labs[2]
University of California, Santa Cruz[3], Qualcomm Inc.[4], Huawei Technologies Inc.[5]
{shuangchenli, yuanxie}ece.ucsb.edu[1]

Pinatubo: Row Copy and Bitwise Ops in PCM

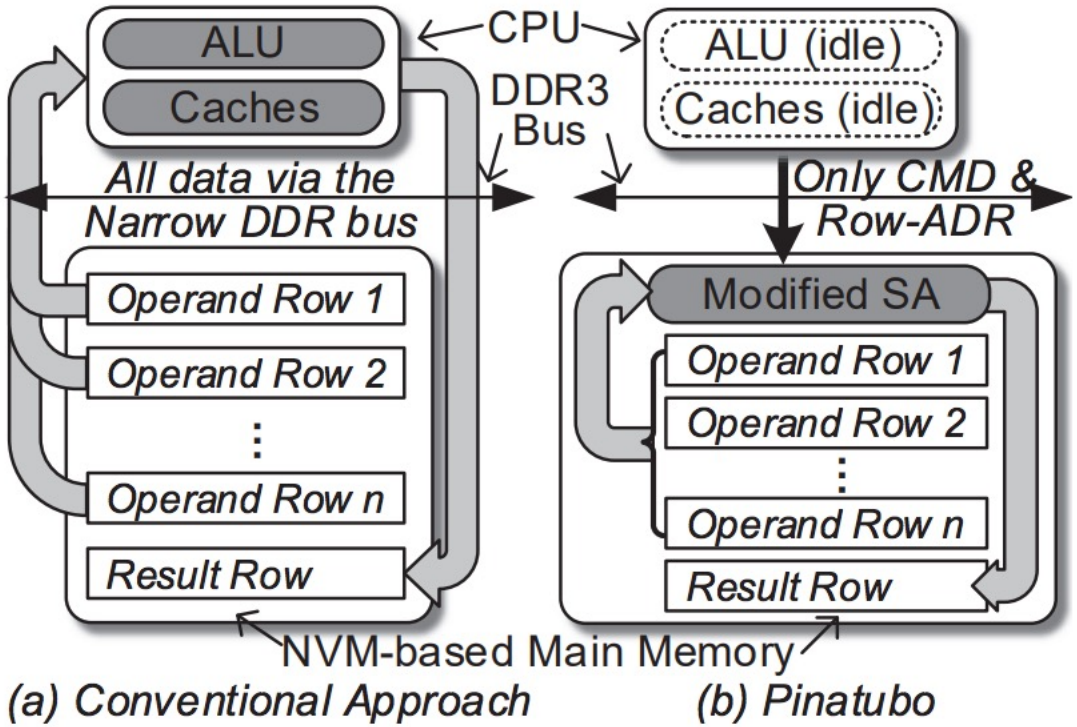
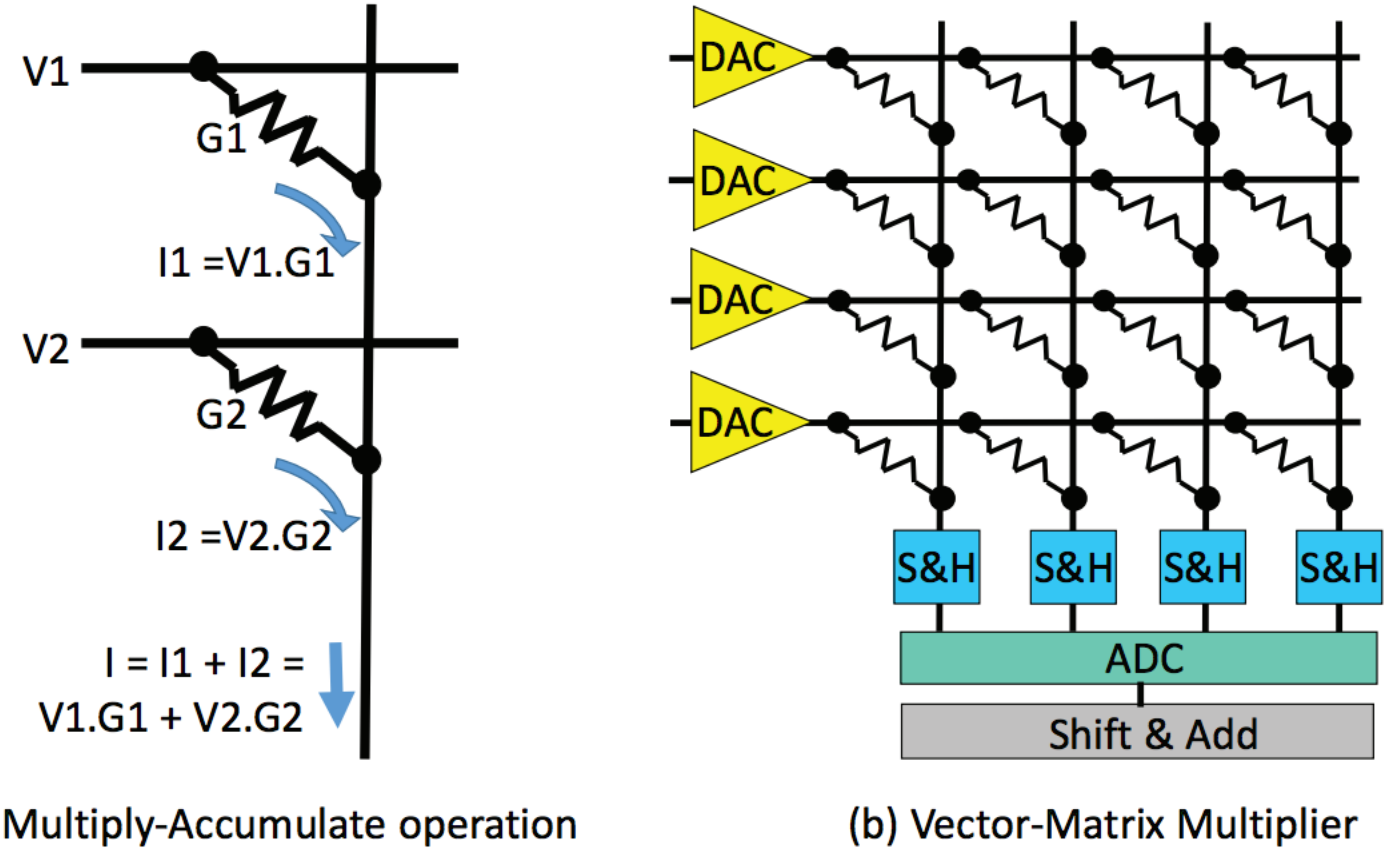


Figure 2: Overview: (a) Computing-centric approach, moving tons of data to CPU and write back. (b) The proposed Pinatubo architecture, performs n-row bitwise operations inside NVM in one step.

In-Memory Crossbar Array Operations

- Some emerging NVM technologies have crossbar array structure
  - Memristors, resistive RAM, phase change mem, STT-MRAM, …
- Crossbar arrays can be used to perform dot product operations using "analog computation capability"
  - Can operate on multiple pieces of data using Kirchoff's laws
    - Bitline current is a sum of products of wordline V x (1 / cell R)
  - Computation is in analog domain inside the crossbar array
- Need peripheral circuitry for D→A and A→D conversion of inputs and outputs

In-Memory Crossbar Computation



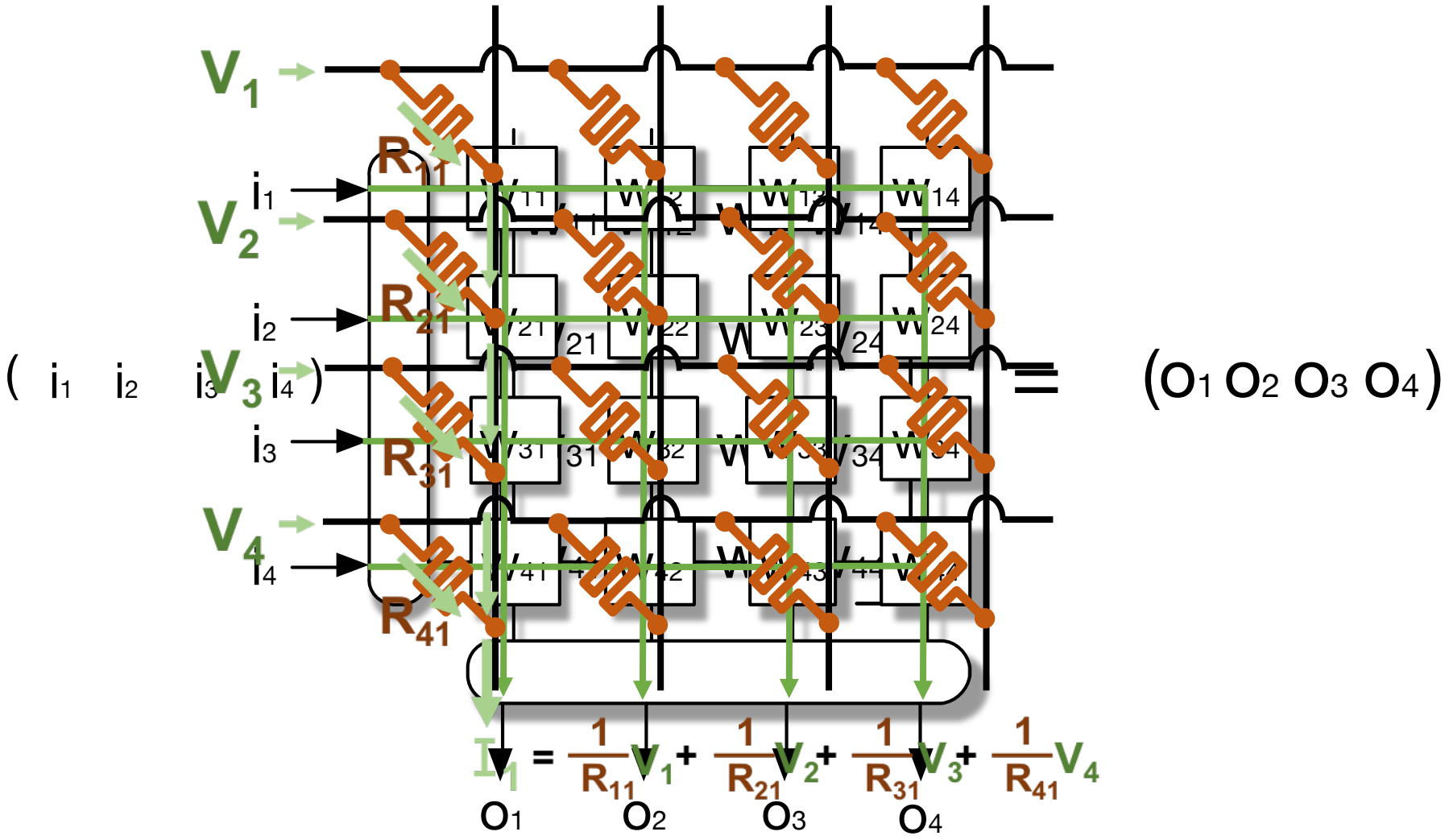
(a) Multiply-Accumulate operation

(b) Vector-Matrix Multiplier

Fig. 1. (a) Using a bitline to perform an analog sum of products operation. (b) A memristor crossbar used as a vector-matrix multiplier.

Shafiee+, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars", ISCA 2016

In-Memory Crossbar Computation



Processing-Using-Memory (PUM)

- PUM: Exploits analog operational principles of the memory circuitry to perform computation
  - Exploits internal connectivity to move data
  - Leverages the large internal bandwidth and parallelism available inside the memory arrays
- A common approach for PUM architectures is to perform bulk bitwise operations
  - Simple logical operations (e.g., AND, OR, XOR)
  - More complex operations (e.g., addition, multiplication)

ASPLOS 2023 Tutorial

Real-world Processing-in-Memory Systems for Modern Workloads

Processing-Using-Memory

Exploiting the Analog Operational Properties of Memory Components

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH Zürich

SAFARI

Sunday, March 26, 2023