

# Methodologies, Workloads, and Tools for Processing-in-Memory: Enabling the Adoption of Data-Centric Architectures

**Geraldo F. Oliveira**

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**ISVLSI  
2022**

**SAFARI**

**ETH** *zürich*

**I** UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

Evaluation

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

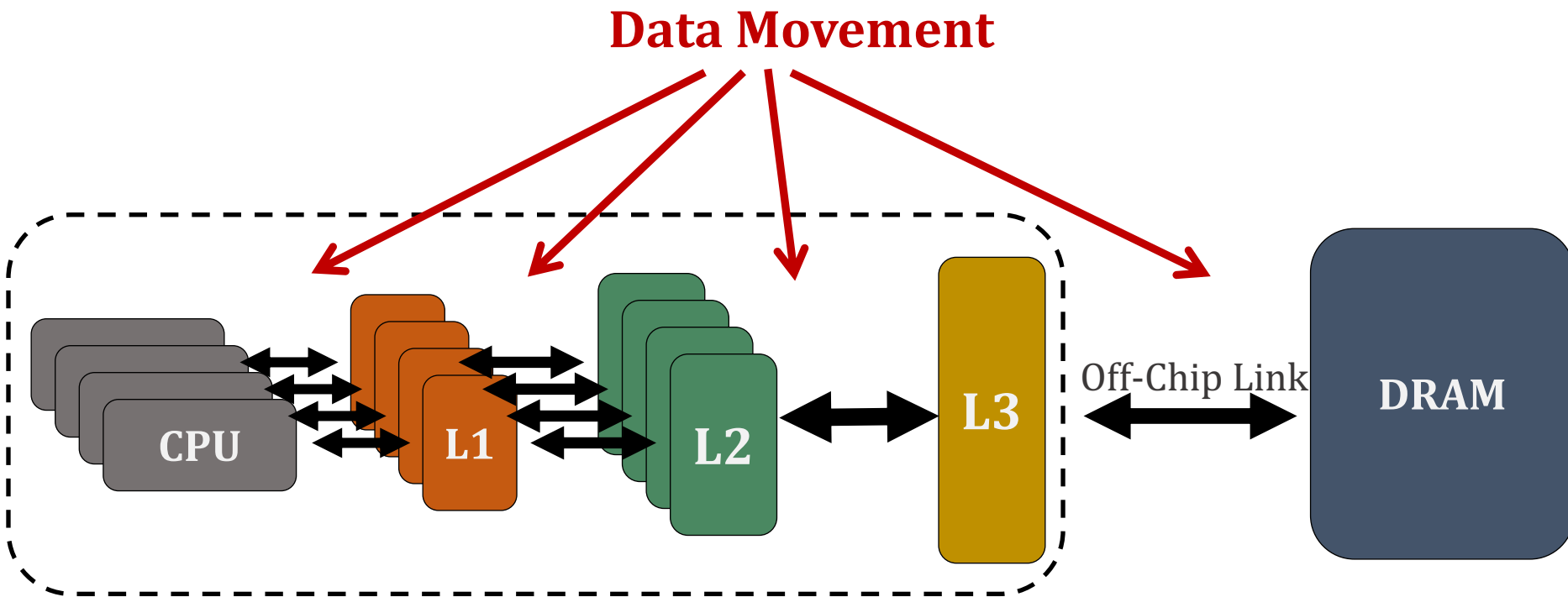
## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

Evaluation

# Data Movement Bottlenecks (1/2)

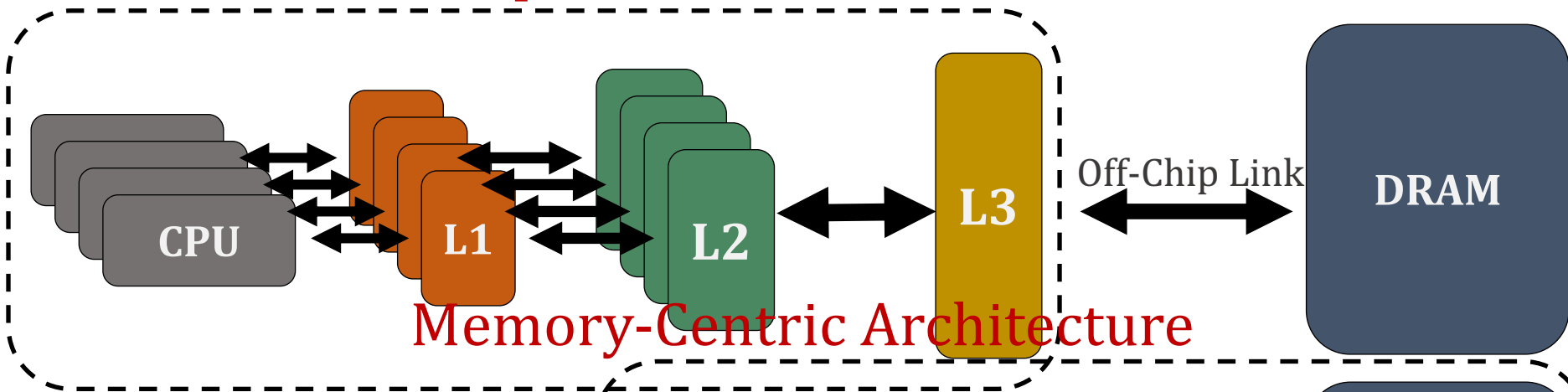


**Data movement bottlenecks** happen because of:

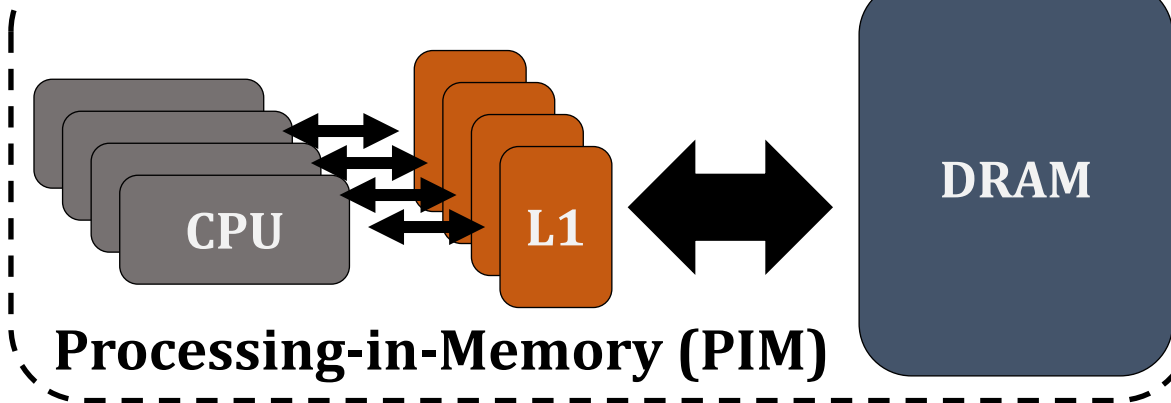
- Not enough data **locality** → ineffective use of the cache hierarchy
- Not enough **memory bandwidth**
- High average **memory access time**

# Data Movement Bottlenecks (2/2)

## Compute-Centric Architecture



## Memory-Centric Architecture



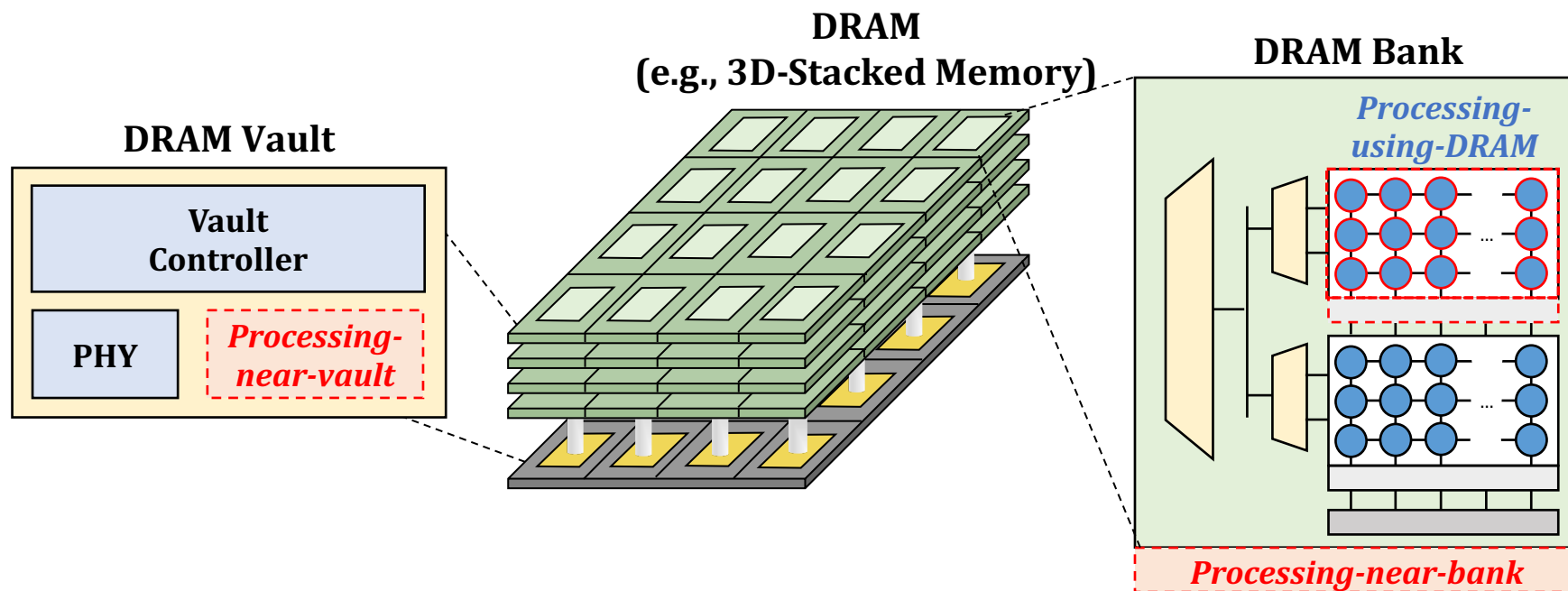
## Processing-in-Memory (PIM)

- Abundant DRAM bandwidth
- Shorter average memory access time

# Processing-in-Memory: Taxonomy

## Two main approaches for Processing-in-Memory:

- 1 **Processing-near-Memory**: PIM logic is added to the **same die as memory** or to the **logic layer** of 3D-stacked memory
- 2 **Processing-using-Memory**: uses the **operational principles of memory cells** to perform computation



# Processing-in-Memory: Challenges

**The lack of tools and system support for PIM architectures limit the adoption of PIM system**

**To fully support PIM systems, we need to develop:**

- 1 Workload characterization methodologies and benchmark suites targeting PIM architectures**
- 2 Frameworks that can facilitate the implementation of complex operations and algorithms using PIM primitives**
- 3 Compiler support and compiler optimizations targeting PIM architectures**
- 4 Operating system support for PIM-aware virtual memory, memory management, data allocation and mapping**
- 5 Efficient data coherence and consistency mechanisms**

# In this Work

The lack of tools and system support for PIM architectures limit the adoption of PIM system

To fully support PIM systems, we need to develop:

- 1 **Workload characterization methodologies and benchmark suites** targeting PIM architectures
- 2 **Frameworks** that can facilitate the implementation of **complex operations and algorithms** using PIM primitives
- 3 Compiler support and compiler optimizations targeting PIM architectures
- 4 Operating system support for PIM-aware virtual memory, memory management, data allocation and mapping
- 5 Efficient data coherence and consistency mechanisms

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

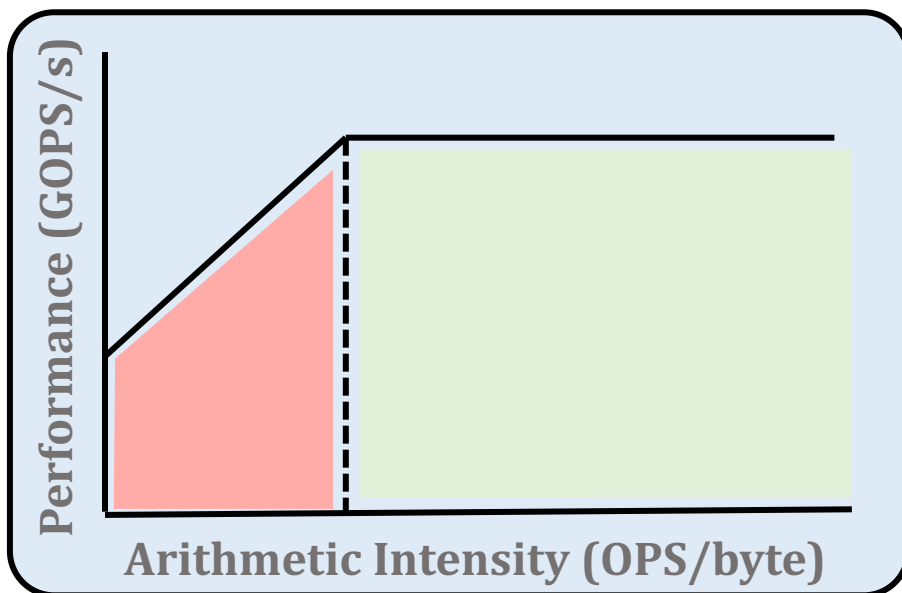
System Integration

Evaluation

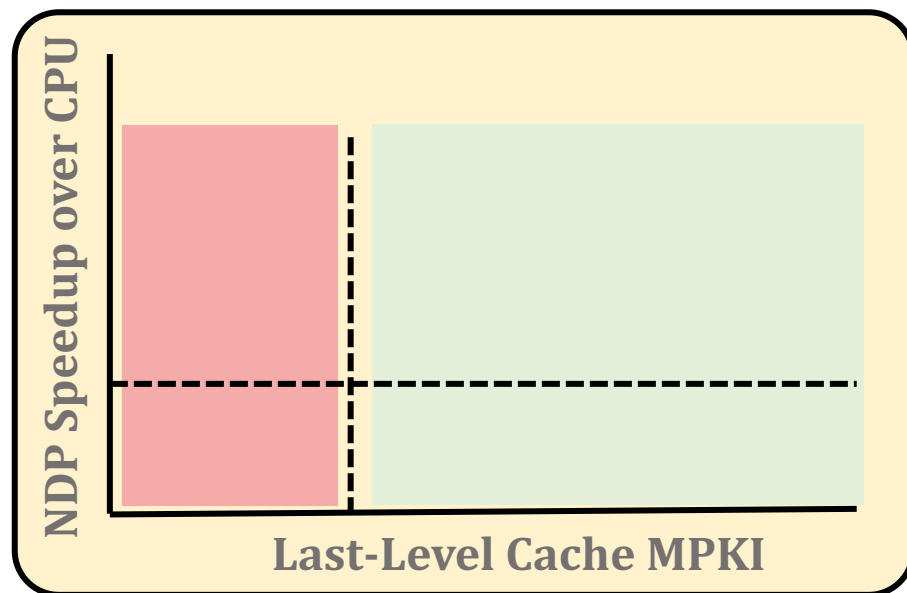
# Identifying Memory Bottlenecks

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP
- Existing approaches are not comprehensive enough

**Roofline model**



**High LLC MPKI**

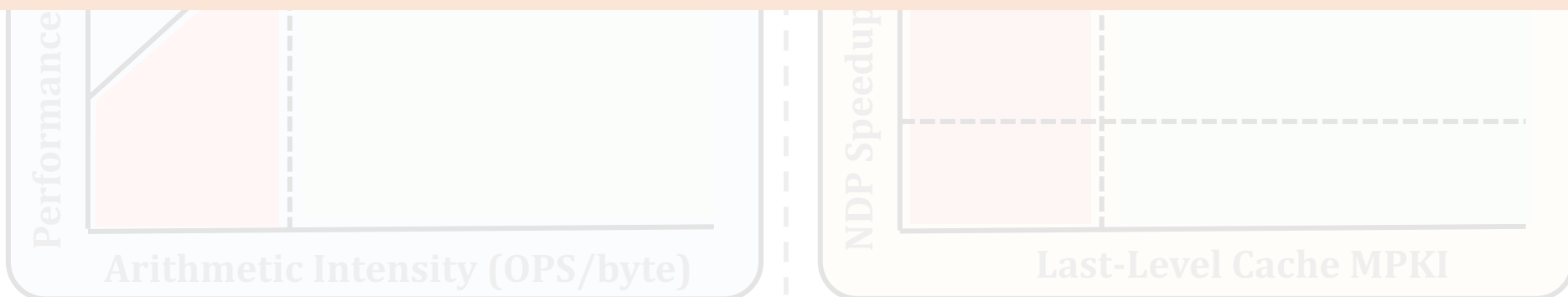


# The Problem

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

No available methodology can comprehensively:

- **identify** data movement bottlenecks
- **correlate** them with the **most suitable** data movement mitigation mechanism



# Our Goal

- **Our Goal:** develop a methodology to:
  - methodically identify sources of data movement bottlenecks
  - comprehensively compare compute- and memory-centric data movement mitigation techniques

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

### Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

Evaluation

# Key Approach

- New **workload characterization methodology** to analyze:
  - data movement bottlenecks
  - suitability of different data movement mitigation mechanisms
- Two main profiling strategies:

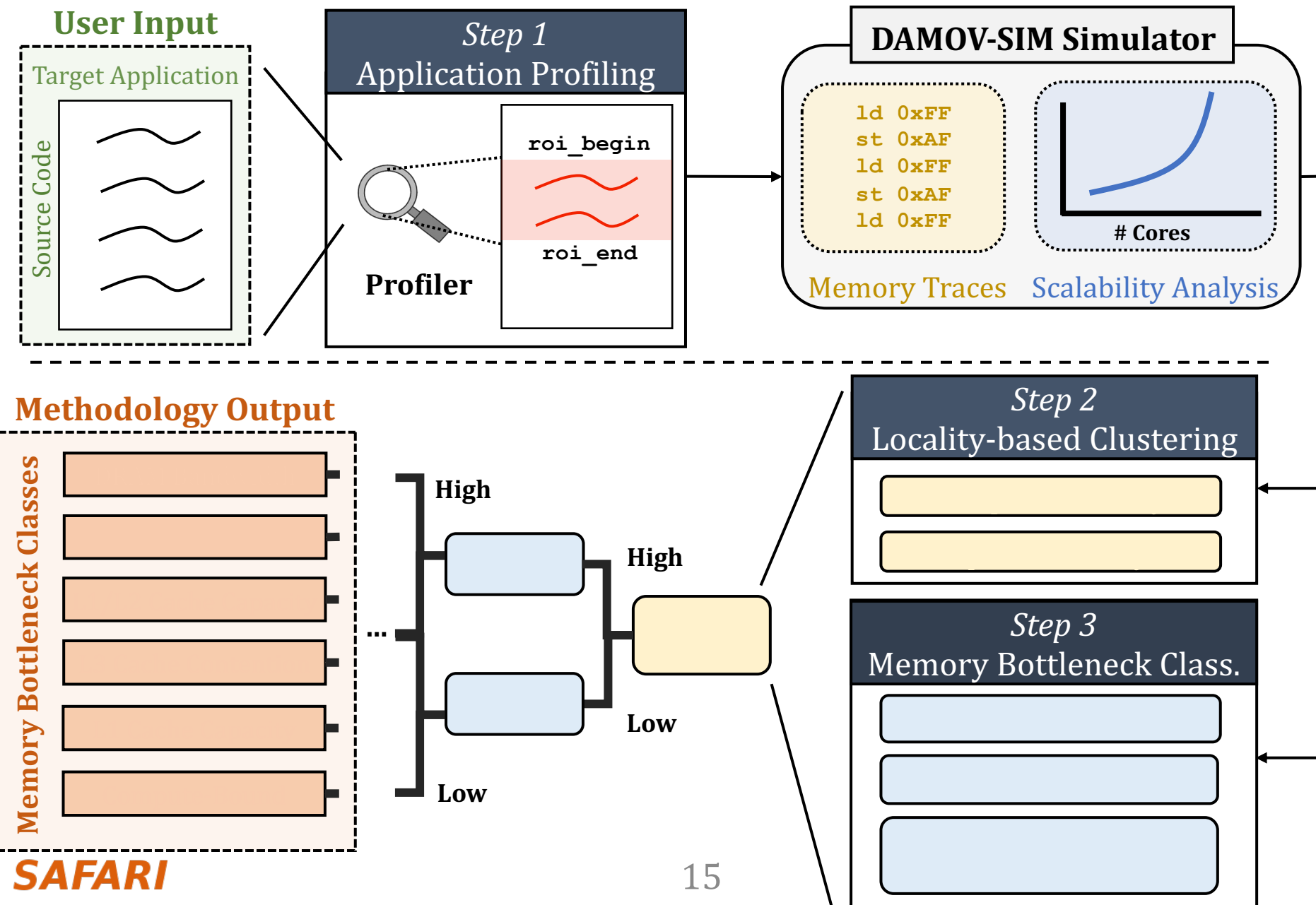
## **Architecture-independent profiling:**

characterizes the memory behavior **independently**  
of the underlying **hardware**

## **Architecture-dependent profiling:**

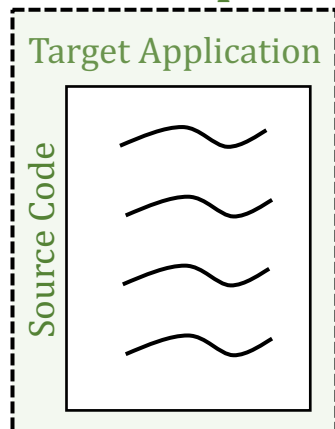
evaluates the **impact of the system configuration**  
on the memory behavior

# Methodology Overview

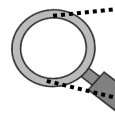


# Methodology Overview

## User Input



## Step 1 Application Profiling



Profiler

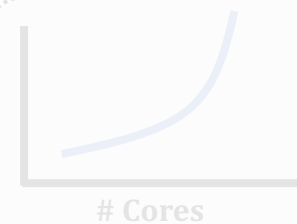
roi\_begin

roi\_end

## DAMOV-SIM Simulator

```
ld 0xFF
st 0xAF
ld 0xFF
st 0xAF
ld 0xFF
```

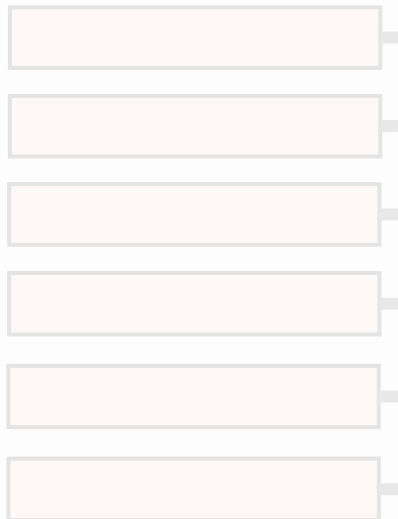
Memory Traces



Scalability Analysis

## Methodology Output

Memory Bottleneck Classes



**SAFARI**

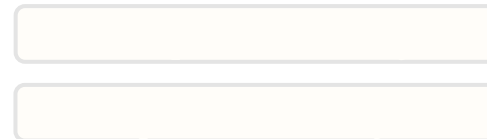
High

High

Low

Low

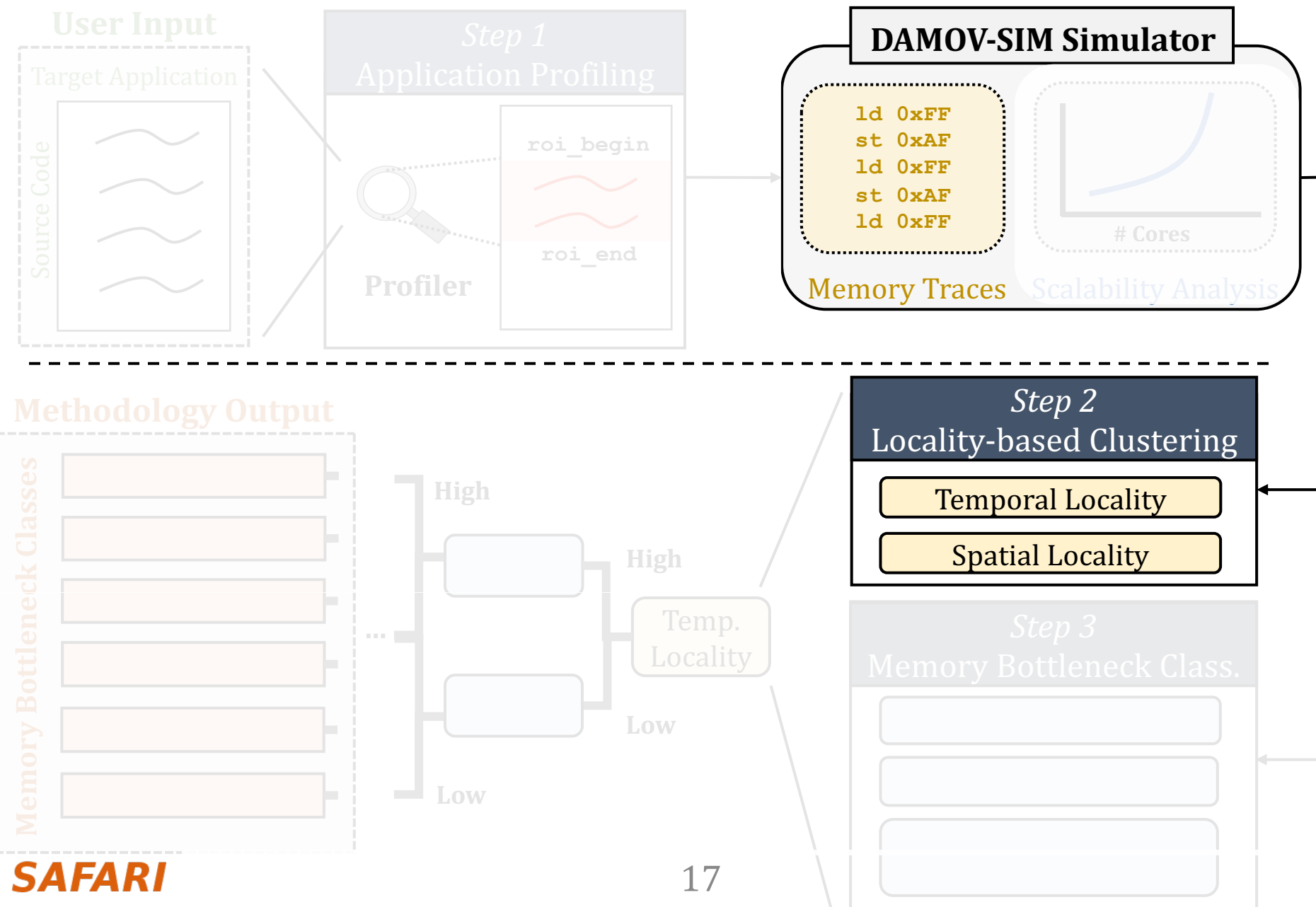
## Step 2 Locality-based Clustering



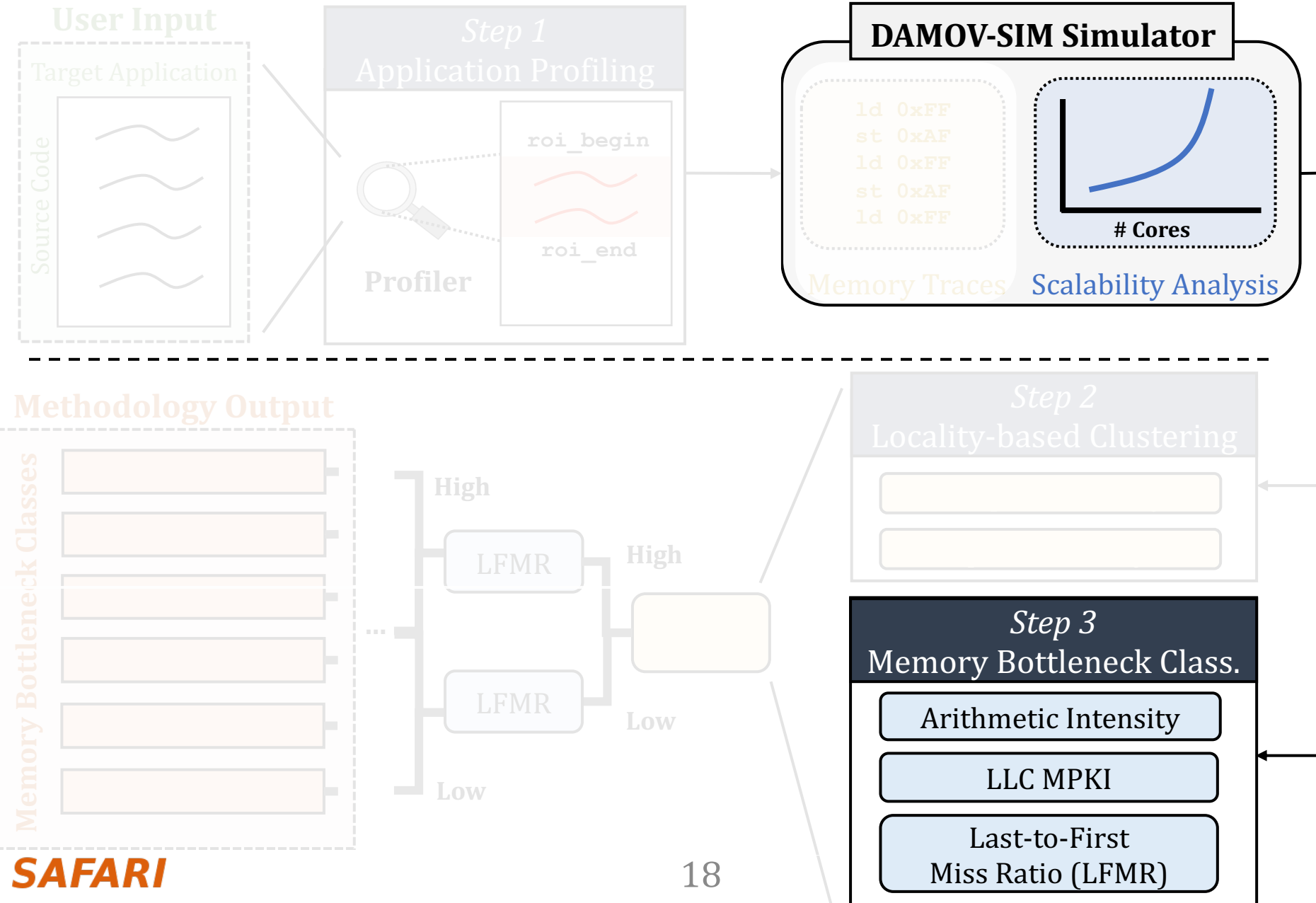
## Step 3 Memory Bottleneck Class.



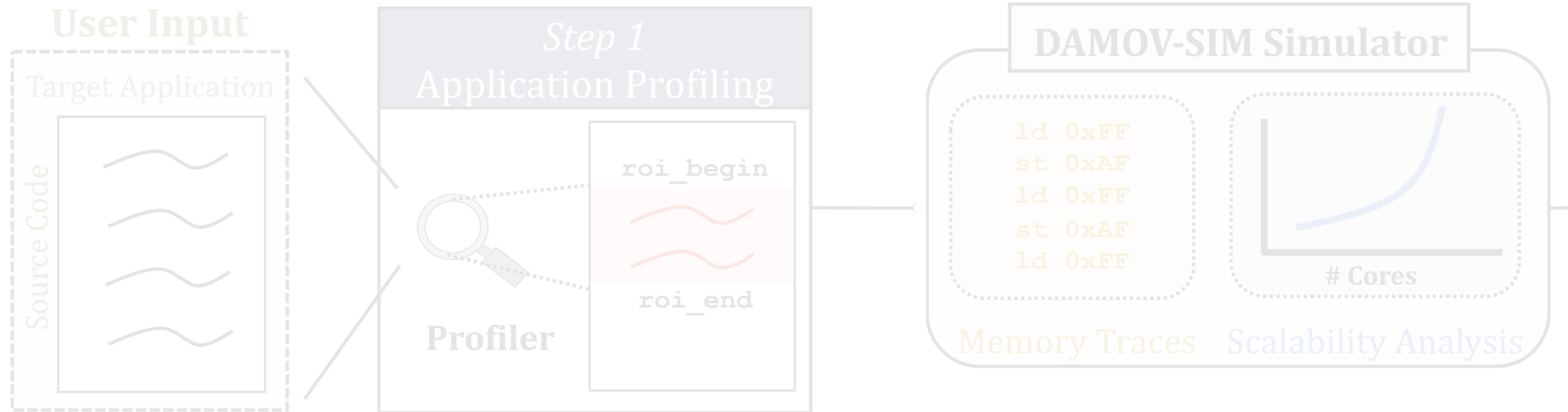
# Methodology Overview



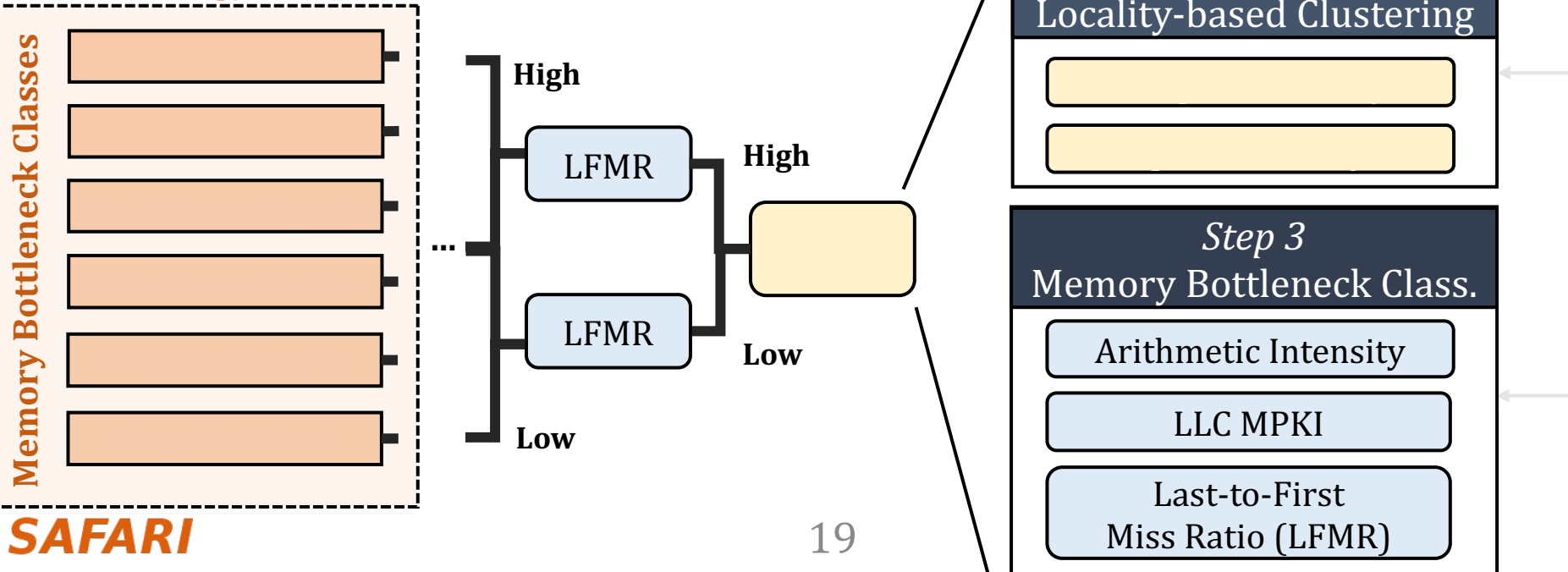
# Methodology Overview



# Methodology Overview



## Methodology Output



# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

**Application Profiling**

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

Evaluation

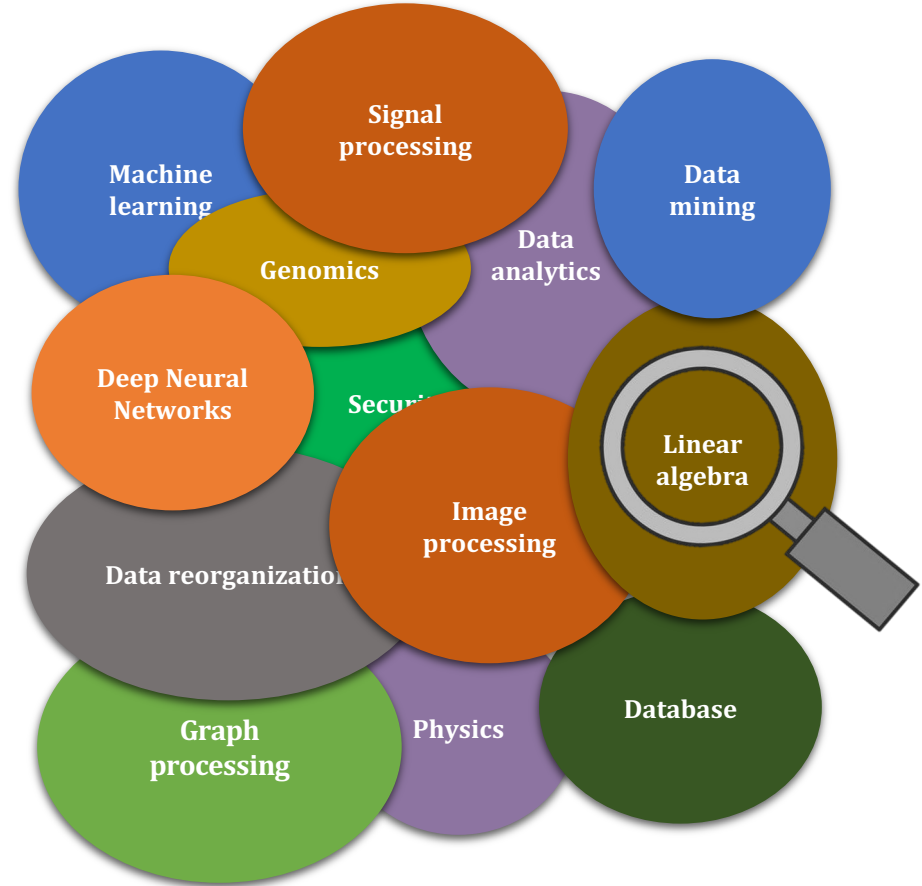
# Step 1: Application Profiling

- We analyze 345 applications from distinct domains:

- Graph Processing
- Deep Neural Networks
- Physics
- High-Performance Computing
- Genomics
- Machine Learning
- Databases
- Data Reorganization
- Image Processing
- Map-Reduce
- Benchmarking
- Linear Algebra

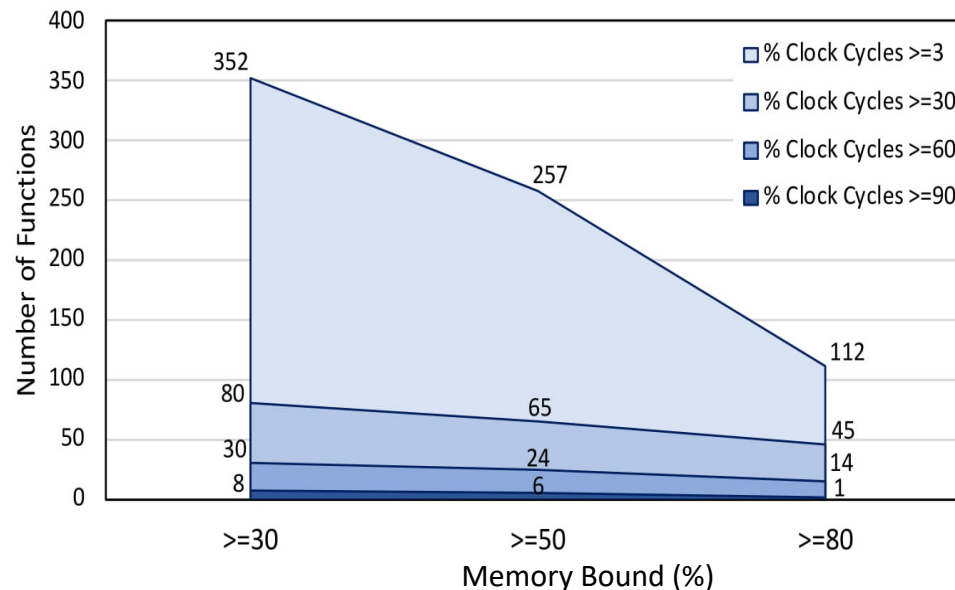
...

**SAFARI**



# Memory Bound Functions

- We analyze 345 applications from distinct domains
- **Selection criteria:** clock cycles > 3% and Memory Bound > 30%



- We find 144 functions from a total of 77K functions and select:
  - 44 functions → apply steps 2 and 3
  - 100 functions → validation

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

**Locality-Based Clustering**

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

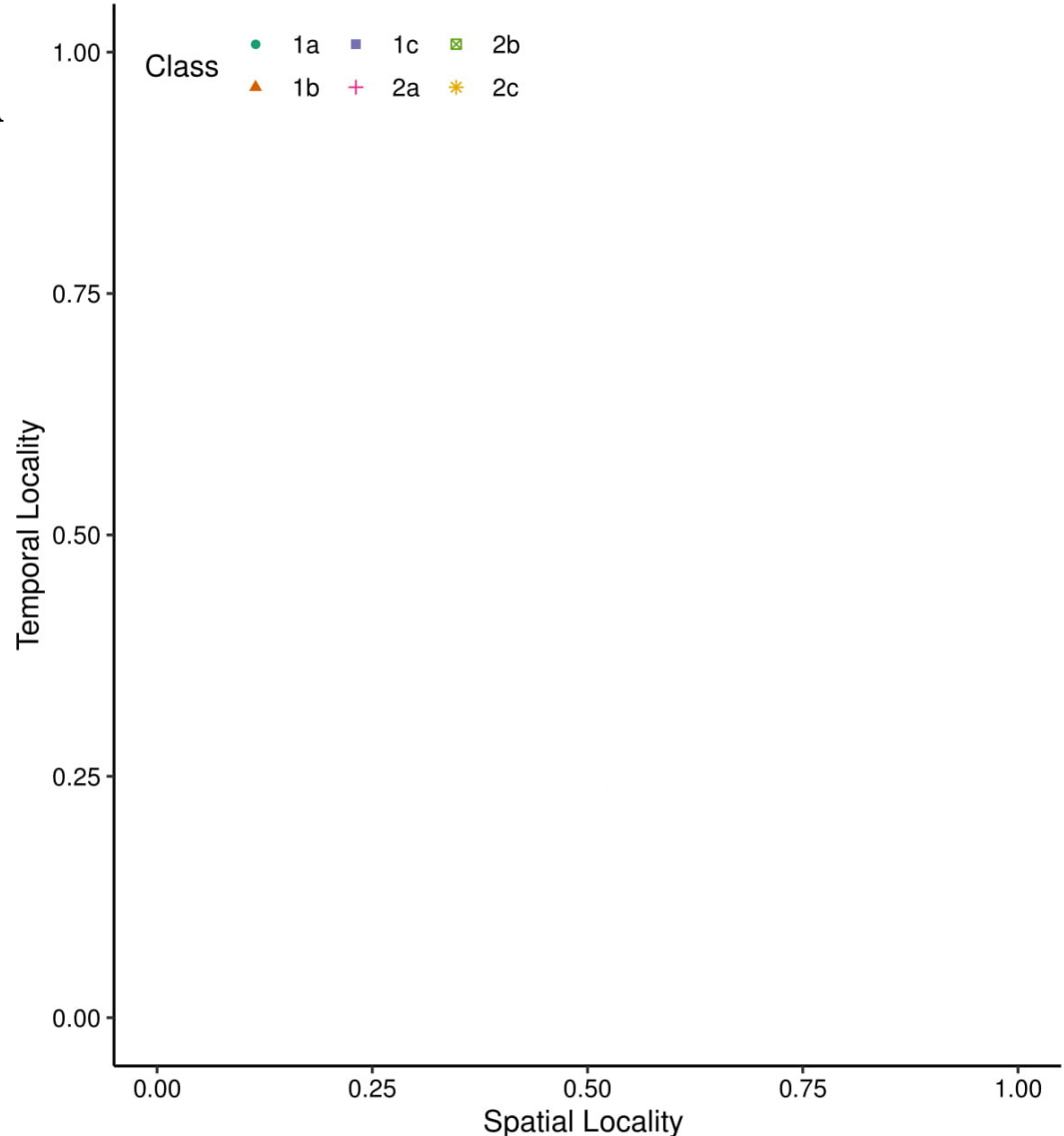
System Integration

Evaluation

# Step 2: Locality-Based Clustering

We use K-means to cluster the applications across both **spatial and temporal locality**, forming two groups

1. Low locality applications (in orange)
2. High locality applications (in blue)



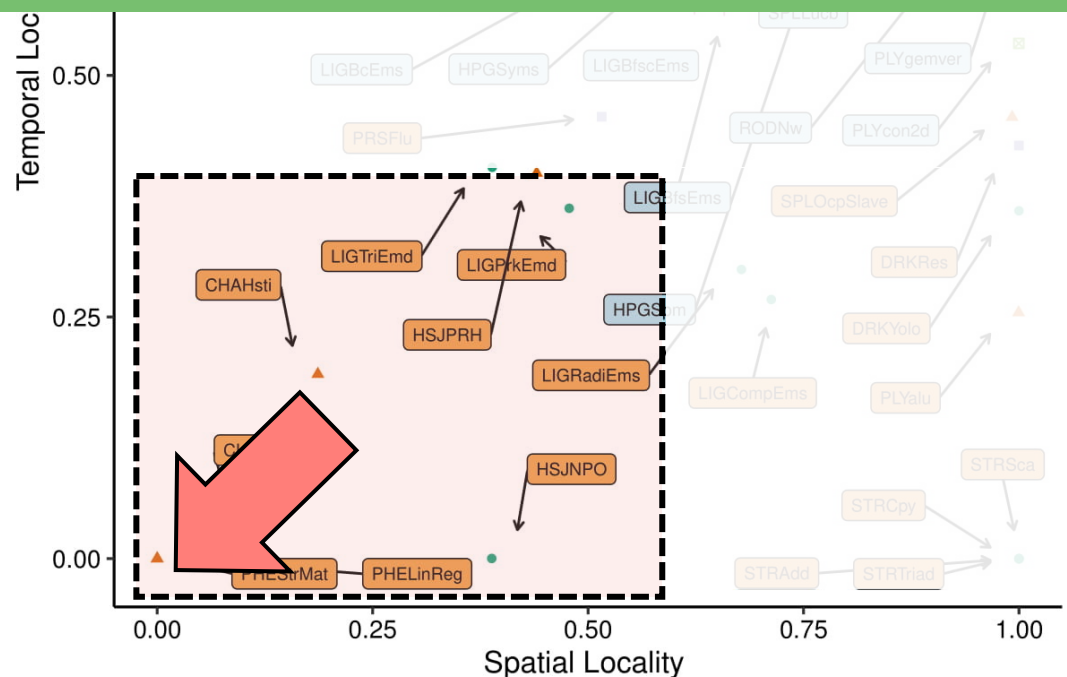
## Step 2: Locality-Based Clustering



The closer a function is to the **bottom-left corner**

→ less likely it is to **take advantage** of  
a deep cache hierarchy

2. High locality applications (in blue)



# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

**Memory Bottleneck Analysis**

DAMOV Benchmark Suite

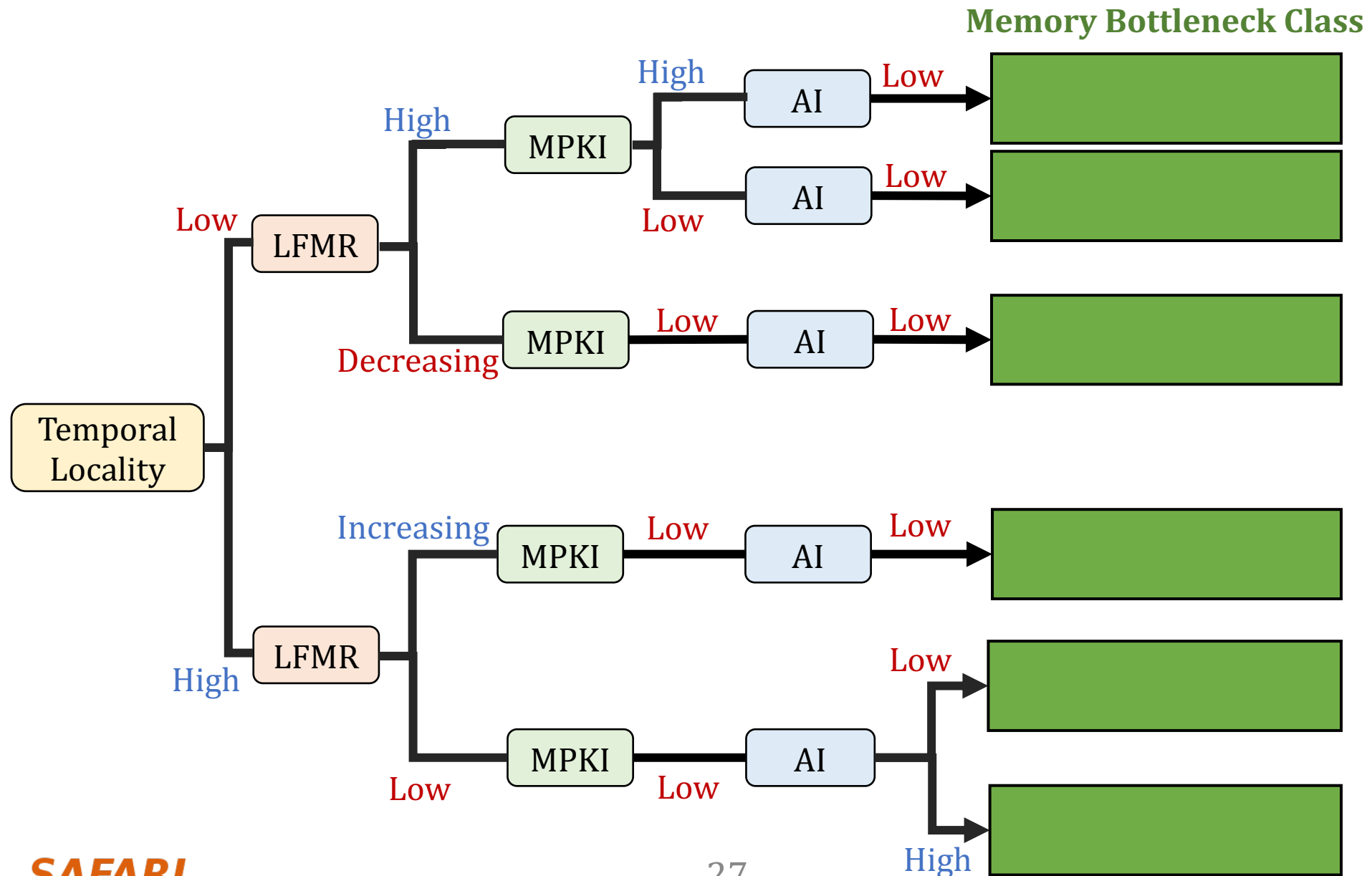
## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

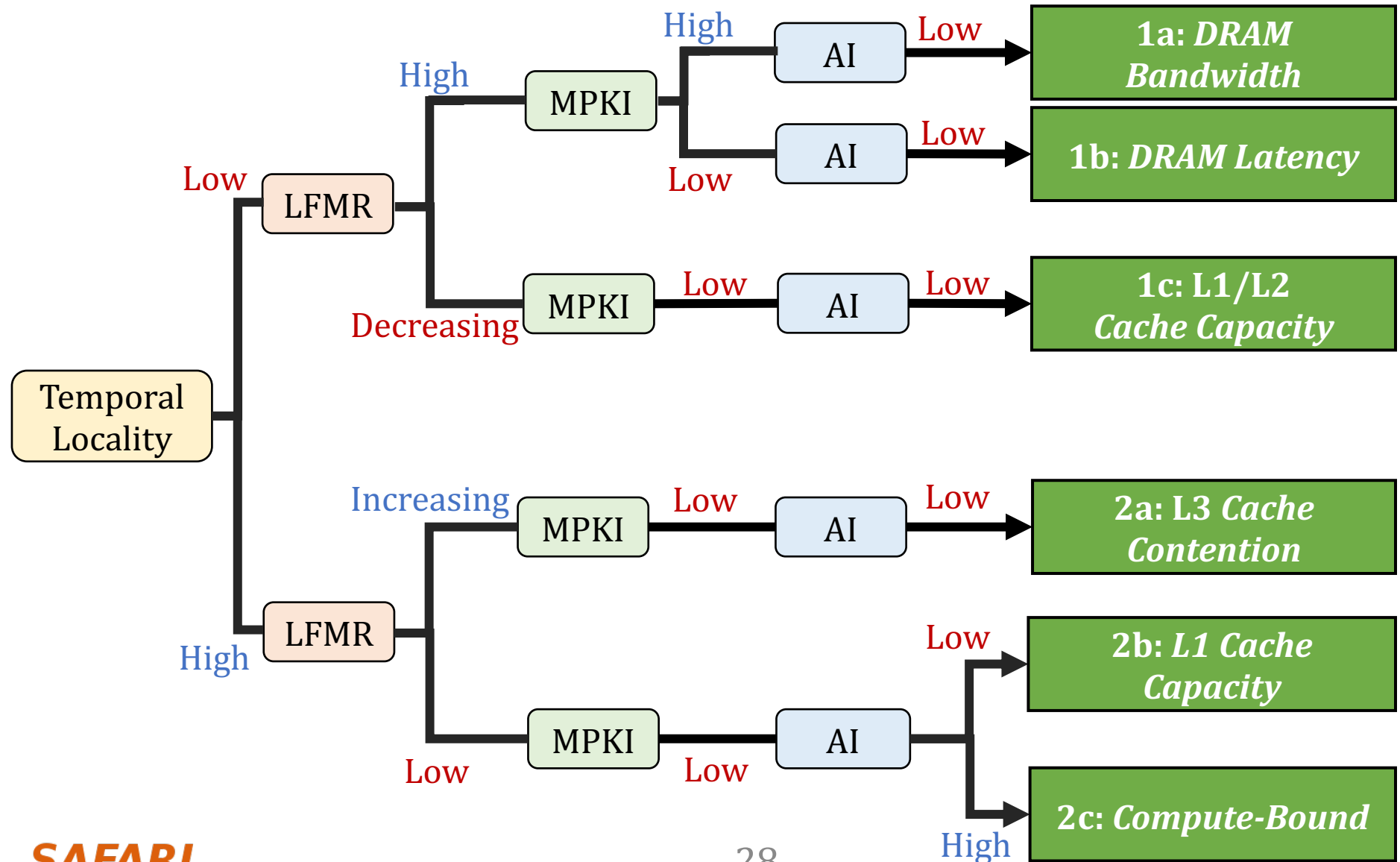
Evaluation

# Step 3: Memory Bottleneck Analysis



# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Step 3: Memory Bottleneck Analysis

**Six classes of  
data movement bottlenecks:**

each class  $\leftrightarrow$  data movement  
mitigation mechanism

## Memory Bottleneck Class

1a: *DRAM  
Bandwidth*

1b: *DRAM Latency*

1c: *L1/L2  
Cache Capacity*

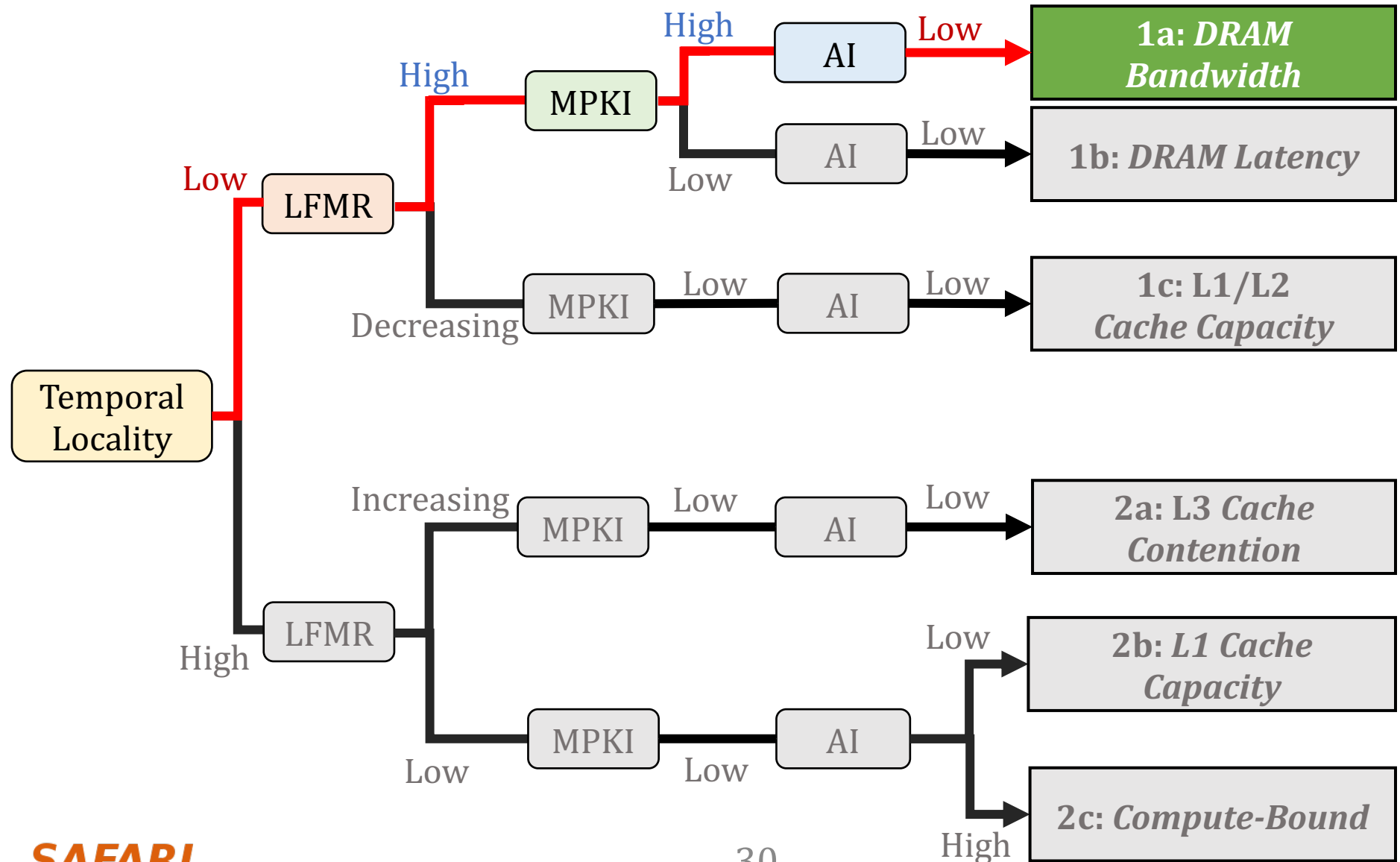
2a: *L3 Cache  
Contention*

2b: *L1 Cache  
Capacity*

2c: *Compute-Bound*

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 1a: DRAM Bandwidth Bound (1/2)

- High MPKI → **high memory pressure**
- Host scales well until **bandwidth saturates**
- NDP scales **without saturating** alongside attained bandwidth

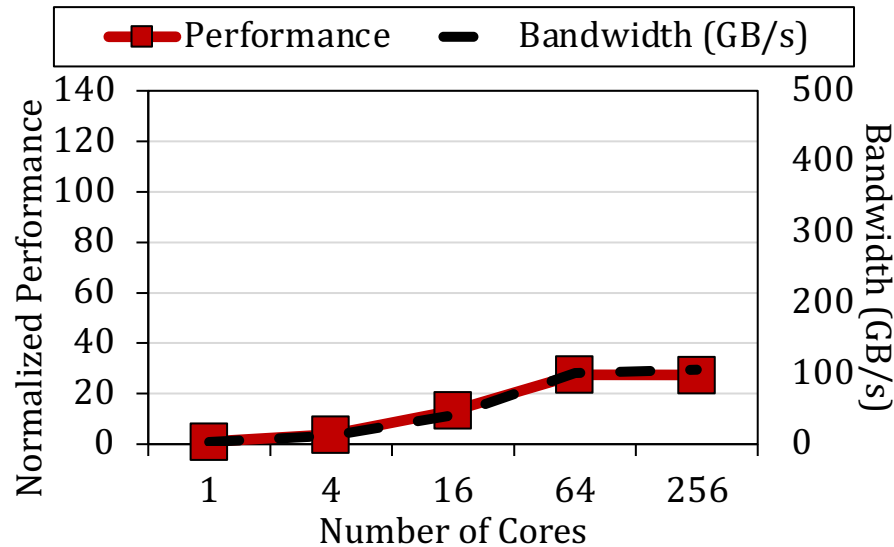
Temp. Loc: *low*

LFMR: *high*

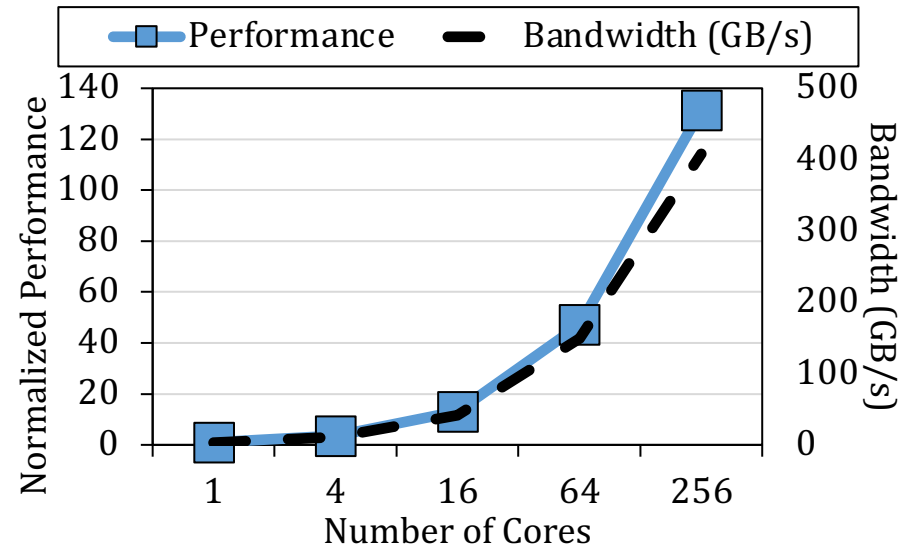
MPKI: *high*

AI: *low*

## Host



## NDP



## DRAM bandwidth bound applications:

NDP does better because of the **higher internal DRAM bandwidth**

# Class 1a: DRAM Bandwidth Bound (2/2)

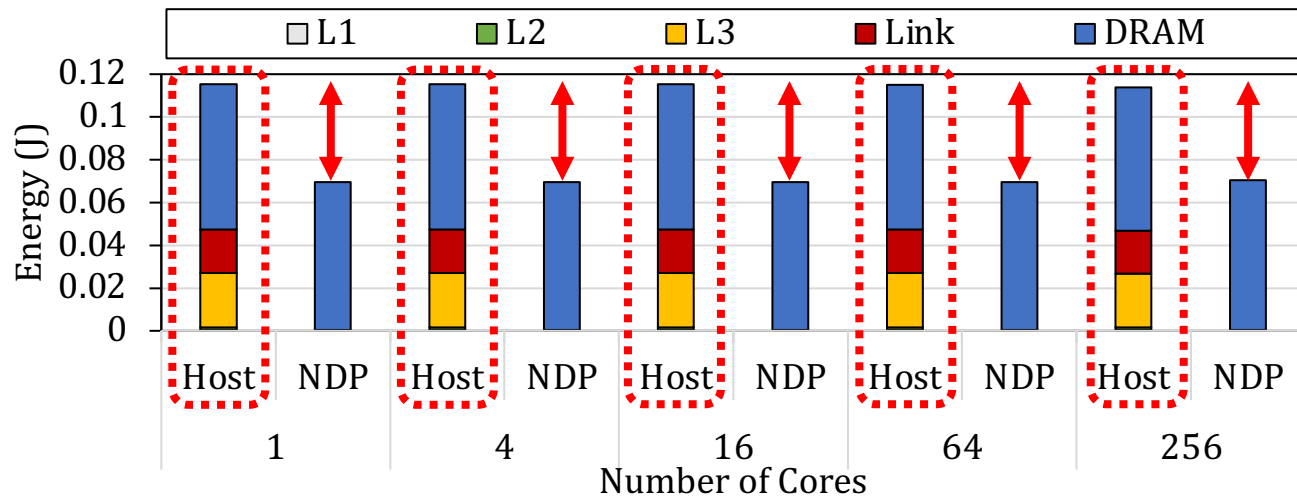
- High LFMR → **L2 and L3 caches are inefficient**
- Host's energy consumption is dominated by **cache look-ups and off-chip data transfers**
- NDP provides **large system energy reduction** since it does not access L2, L3, and off-chip links

Temp. Loc: *low*

LFMR: *high*

MPKI: *high*

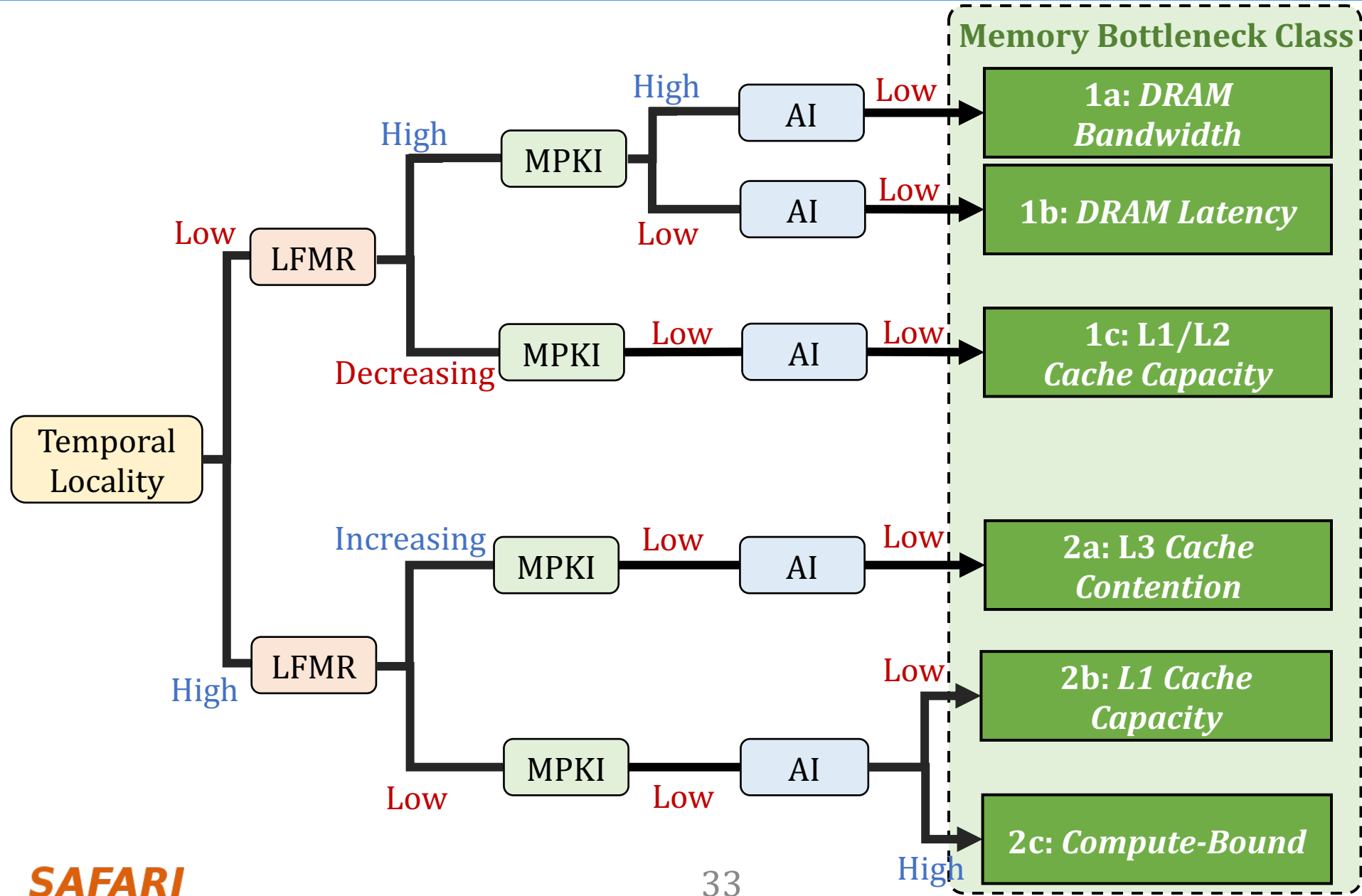
AI: *low*



**DRAM bandwidth bound applications:**

NDP does better because it eliminates off-chip I/O traffic

# Step 3: Memory Bottleneck Analysis



# Step 3: Memory Bottleneck Analysis

Memory Bottleneck Class



## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

GERALDO F. OLIVEIRA<sup>1</sup>, JUAN GÓMEZ-LUNA<sup>1</sup>, LOIS OROSA<sup>1</sup>, SAUGATA GHOSE<sup>2</sup>, NANDITA VIJAYKUMAR<sup>3</sup>, IVAN FERNANDEZ<sup>1,4</sup>, MOHAMMAD SADROSADATI<sup>1</sup>, and ONUR MUTLU<sup>1</sup>

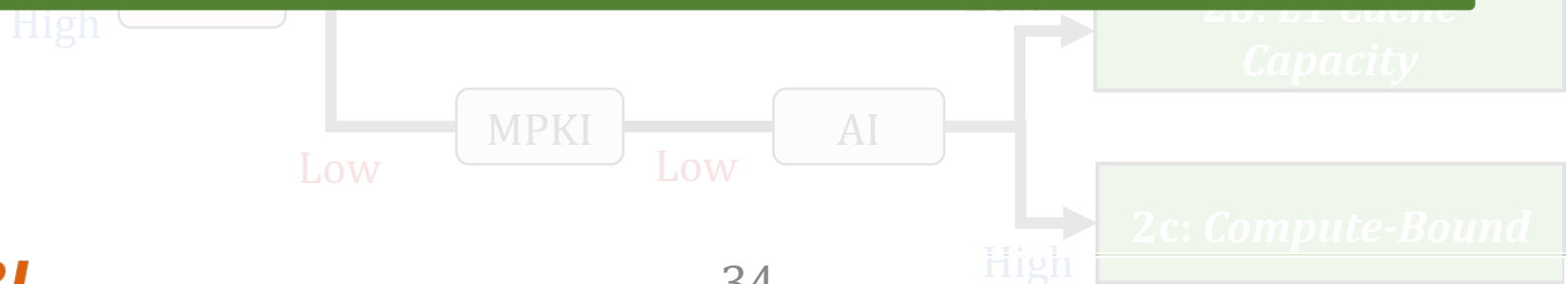
<sup>1</sup>ETH Zürich, Switzerland

<sup>2</sup>University of Illinois Urbana-Champaign, USA

<sup>3</sup>University of Toronto, Canada

<sup>4</sup>University of Malaga, Spain

Corresponding author: Geraldo F. Oliveira (e-mail: geraldod@inf.ethz.ch).



# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

**DAMOV Benchmark Suite**

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

Evaluation

# DAMOV is Open-Source

- We open-source our benchmark suite and our toolchain

CMU-SAFARI / DAMOV

<> Code Issues Pull requests Actions Projects Security Insights Settings

main 1 branch 0 tags

Go to file

Add file

Code

About



omutlu Update README.md

ce1b4ea 17 days ago 5 commits

simulator

Cleaning

19 days ago

README.md

Update README.md

17 days ago

get\_workloads.sh

DAMOV -- first commit

19 days ago

README.md



## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing. Described by Oliveira et al. (preliminary version at <https://arxiv.org/pdf/2105.03725.pdf>)

Readme

Releases

No releases published  
[Create a new release](#)

Packages

No packages published  
[Publish your first package](#)

Languages



DAMOV-SIM

DAMOV  
Benchmark

# DAMOV is Open-Source

- We open-source our benchmark suite and our toolchain

CMU-SAFARI / DAMOV

<> Code Issues Pull requests Actions Projects Security Insights Settings

main 1 branch 0 tags

Go to file

Add file

Code

About

DAMOV is a benchmark suite and a methodical framework targeting the

omutlu Update README.md

celb4ea 17 days ago 5 commits

## Get DAMOV at:

<https://github.com/CMU-SAFARI/DAMOV>

## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.

### Releases

No releases published  
[Create a new release](#)

### Packages

No packages published  
[Publish your first package](#)

### Languages

# Conclusion

- **Problem**: Data movement is a major bottleneck in modern systems. However, it is **unclear** how to identify:
  - **different sources** of data movement bottlenecks
  - the **most suitable** mitigation technique (e.g., caching, prefetching, near-data processing) for a given data movement bottleneck
- **Goals**:
  1. Design a methodology to **identify** sources of data movement bottlenecks
  2. **Compare** compute- and memory-centric data movement mitigation techniques
- **Key Approach**: Perform a large-scale application characterization to identify **key metrics** that reveal the sources to data movement bottlenecks
- **Key Contributions**:
  - **Experimental characterization** of 77K functions across 345 applications
  - A **methodology** to characterize applications based on data movement bottlenecks and their relation with different data movement mitigation techniques
  - **DAMOV**: a **benchmark suite** with **144 functions** for data movement studies
    - Get DAMOV at: <https://github.com/CMU-SAFARI/DAMOV>
  - **Four case-studies** to highlight DAMOV's applicability to open research problems

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

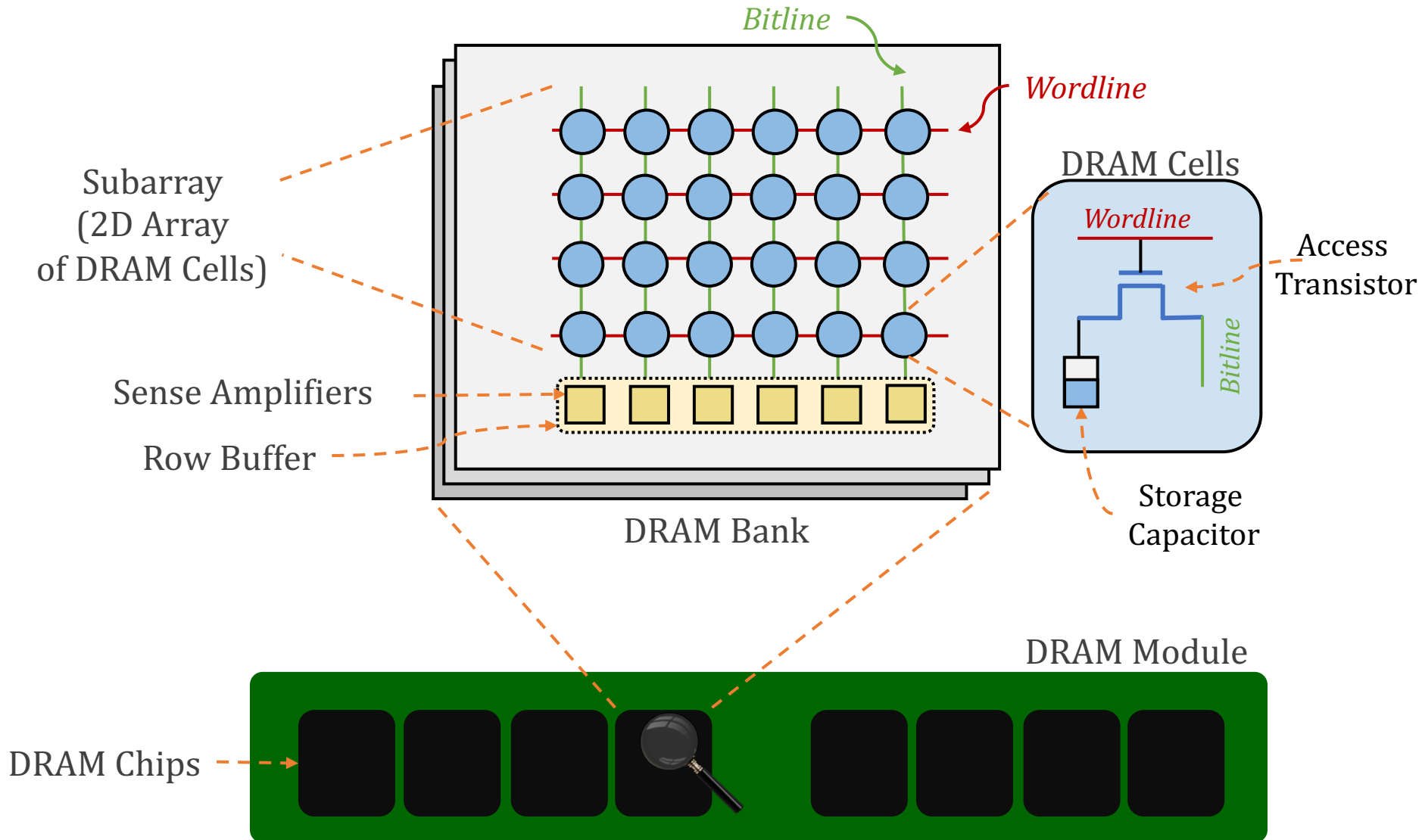
## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

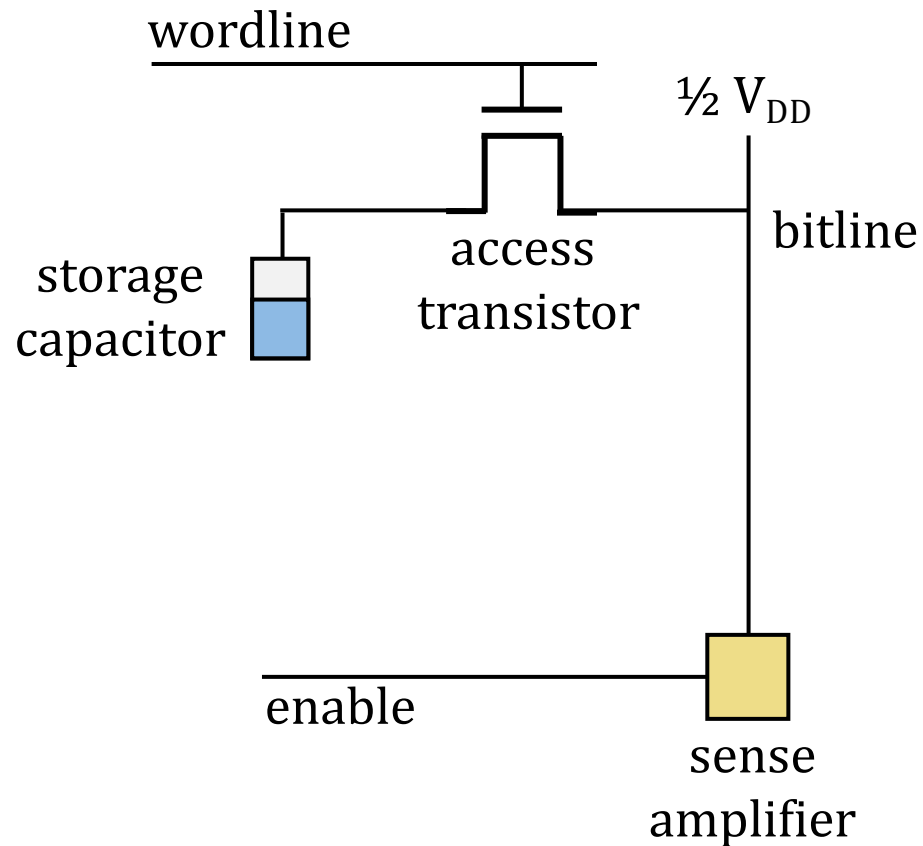
System Integration

Evaluation

# Inside a DRAM Chip



# DRAM Cell Operation

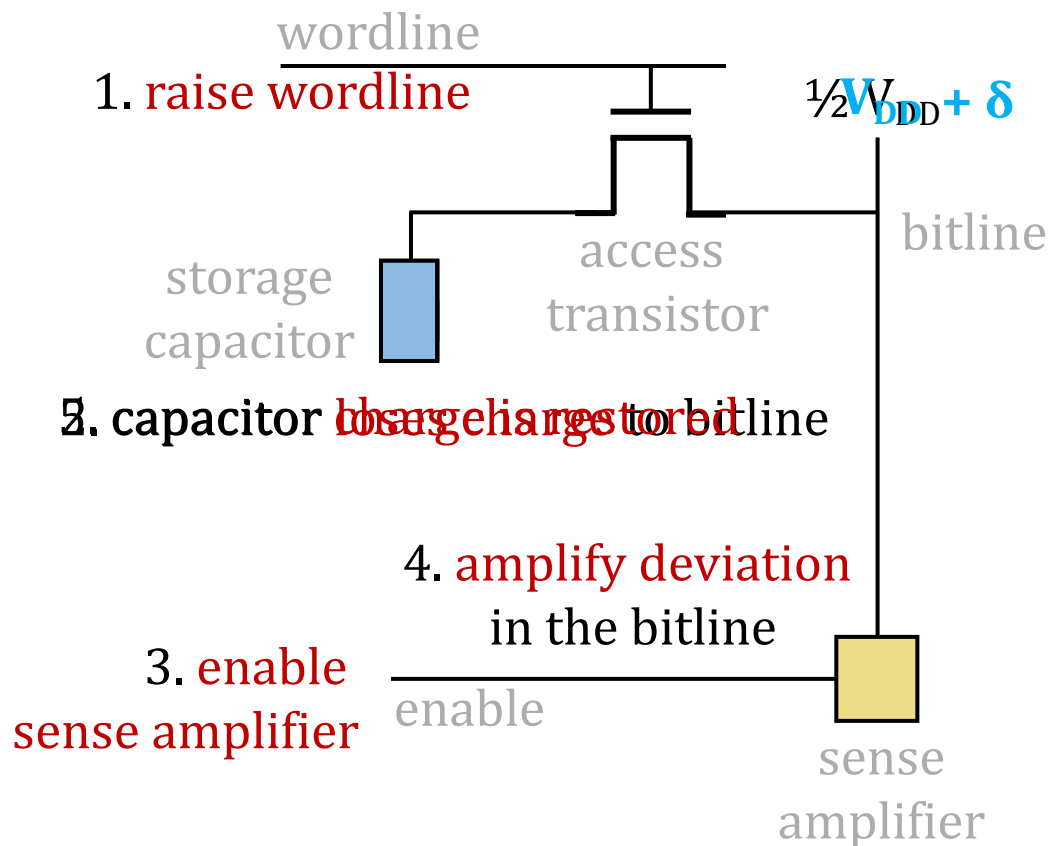


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

# DRAM Cell Operation (1/3)

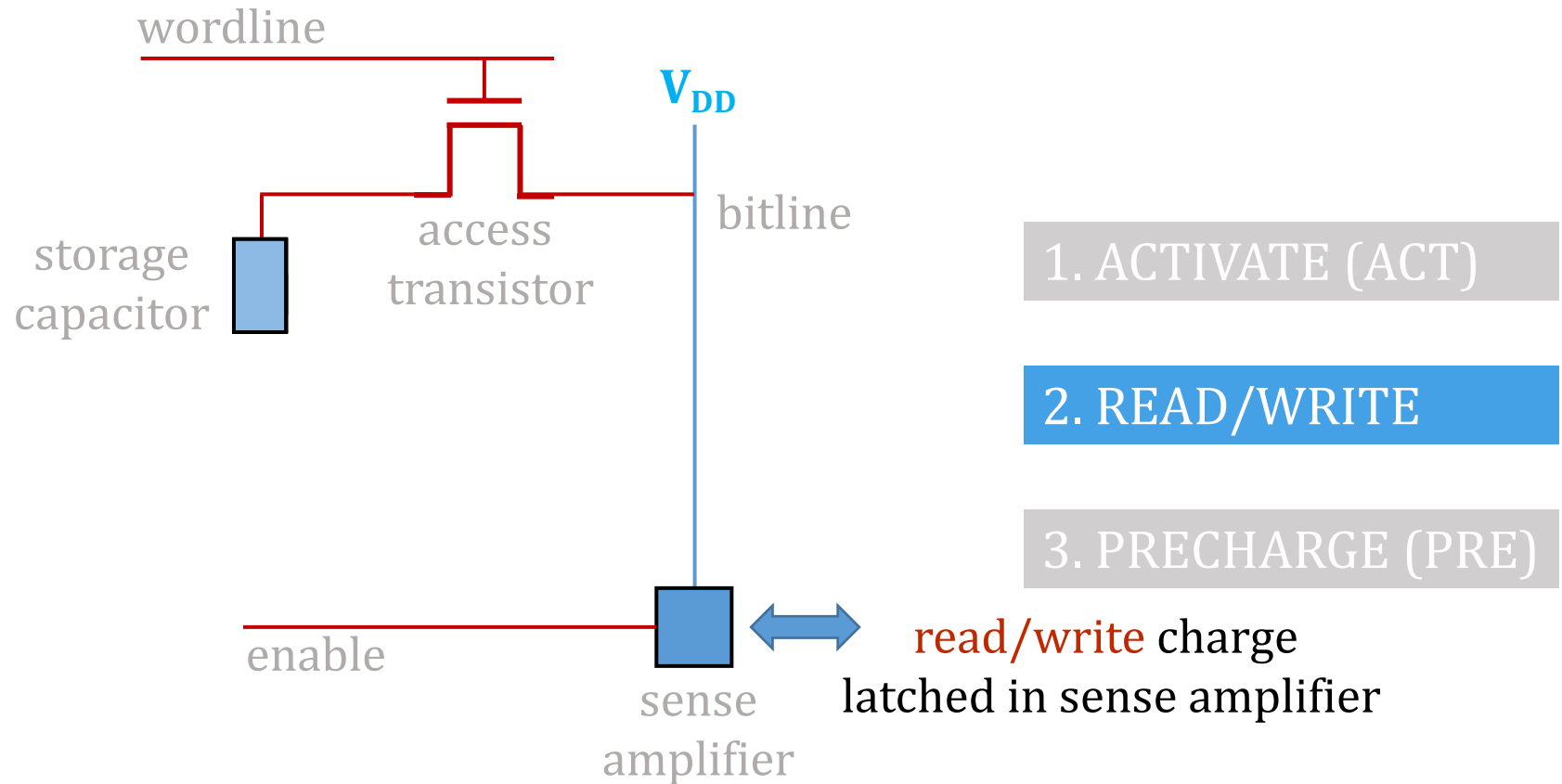


1. ACTIVATE (ACT)

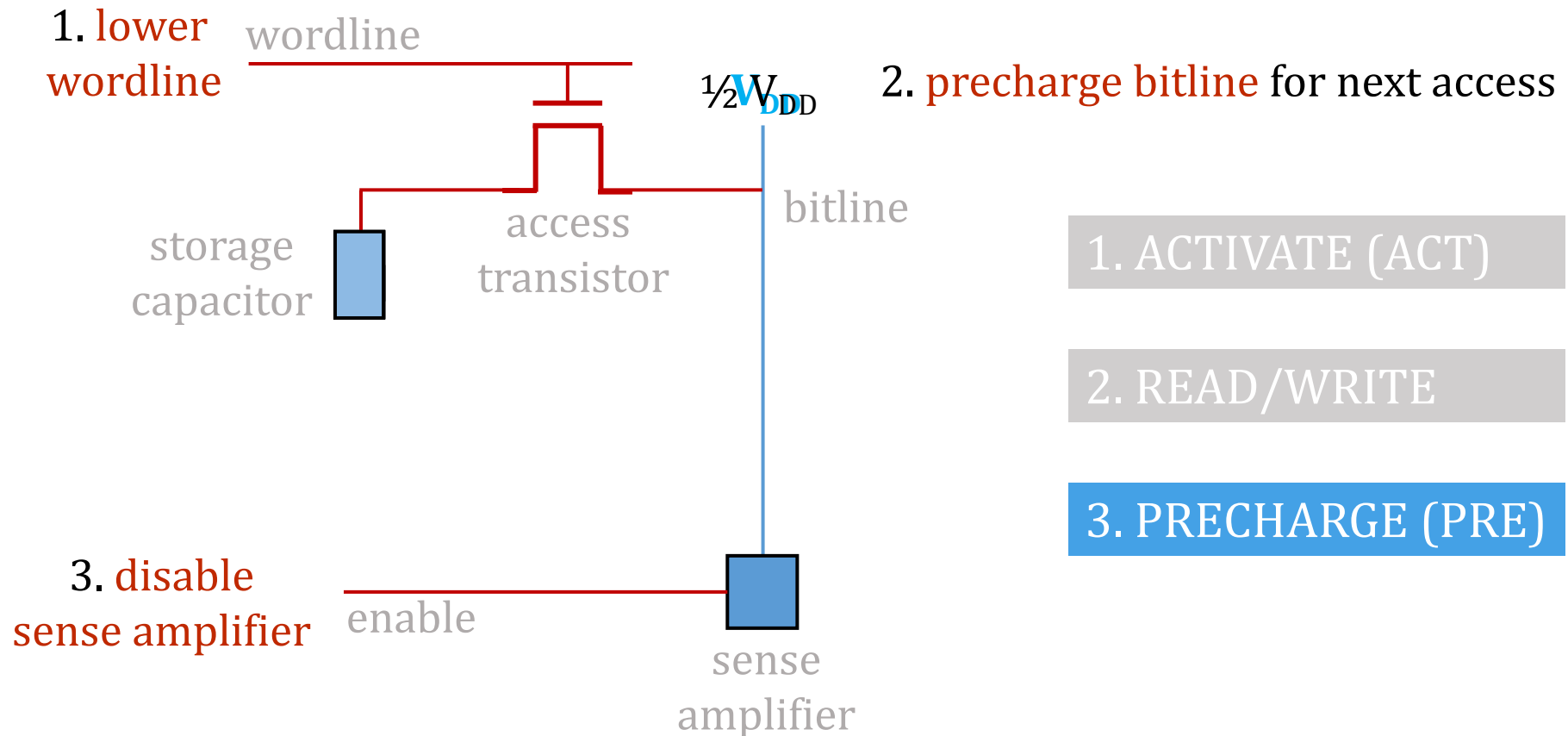
2. READ/WRITE

3. PRECHARGE (PRE)

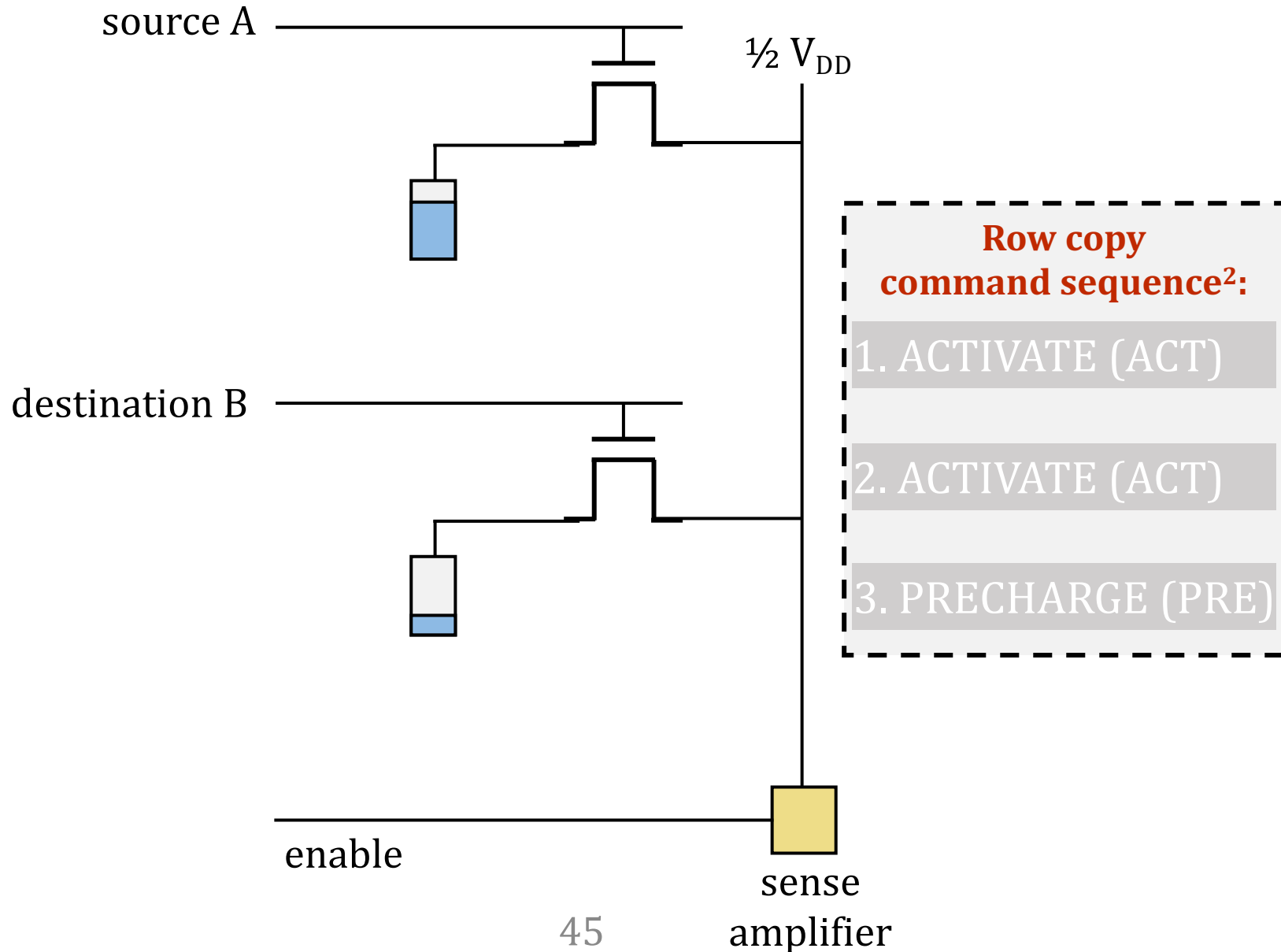
# DRAM Cell Operation (2/3)



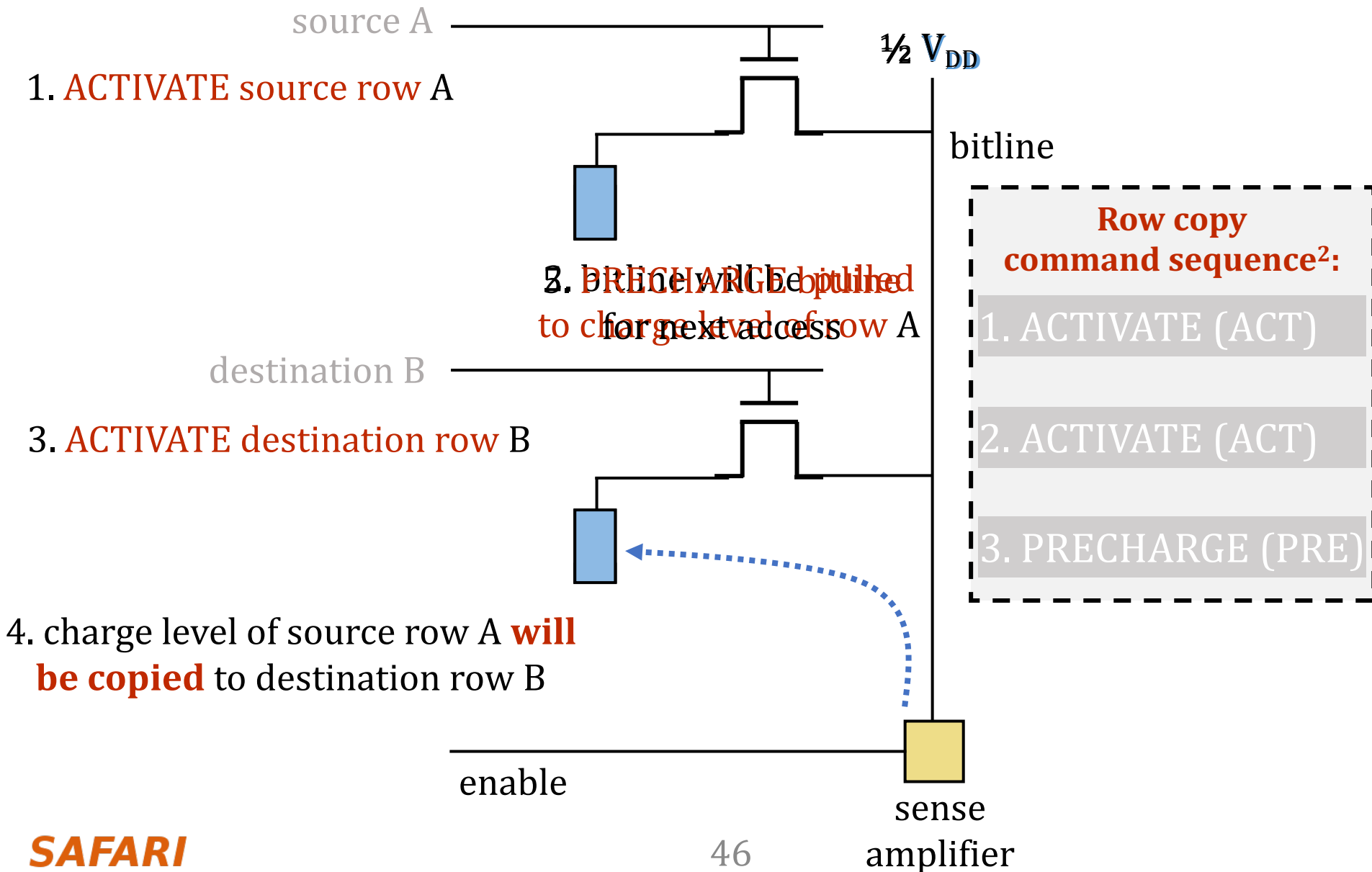
# DRAM Cell Operation (3/3)



# RowClone: In-DRAM Row Copy (1/2)

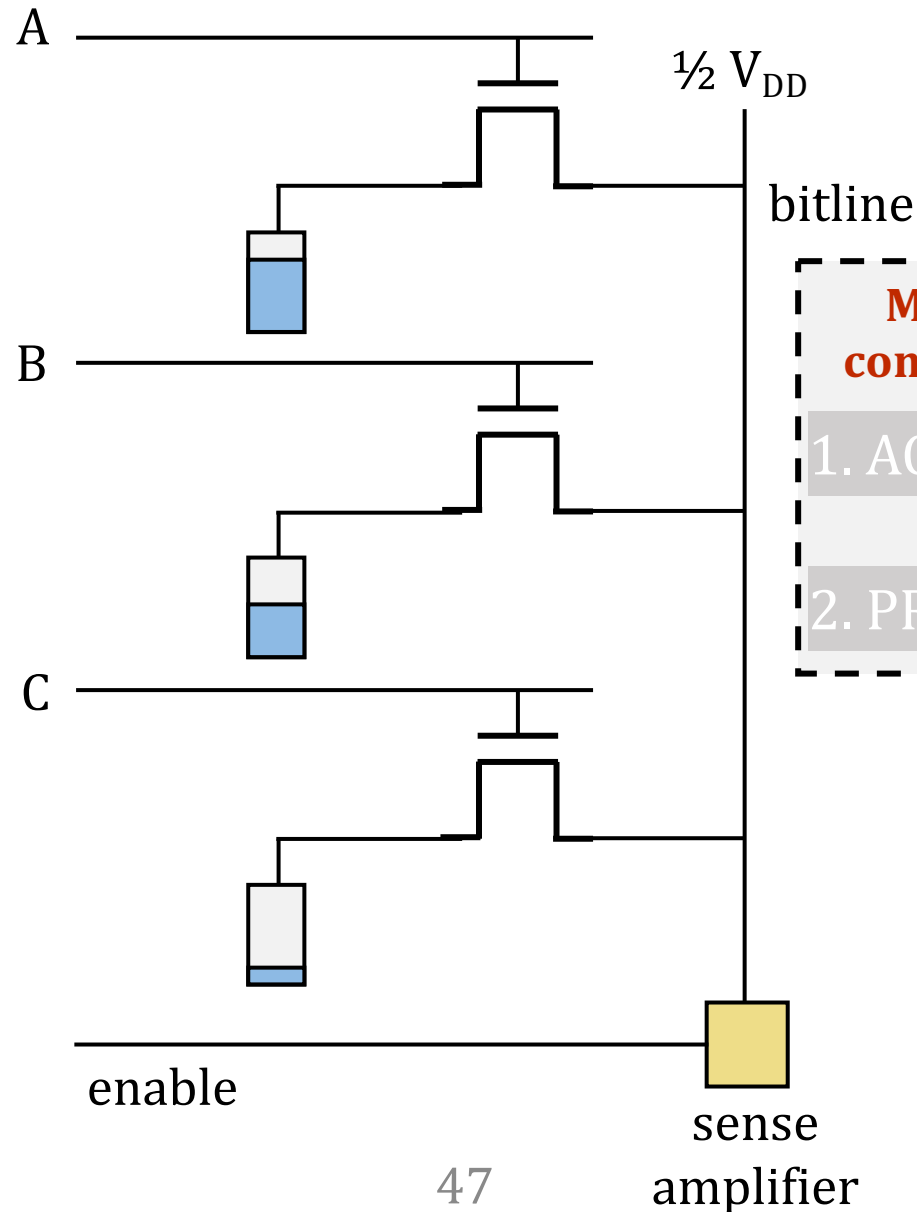


# RowClone: In-DRAM Row Copy (2/2)



<sup>2</sup> V. Seshadri et al., "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization", MICRO, 2013

# Triple-Row Activation: Majority Function



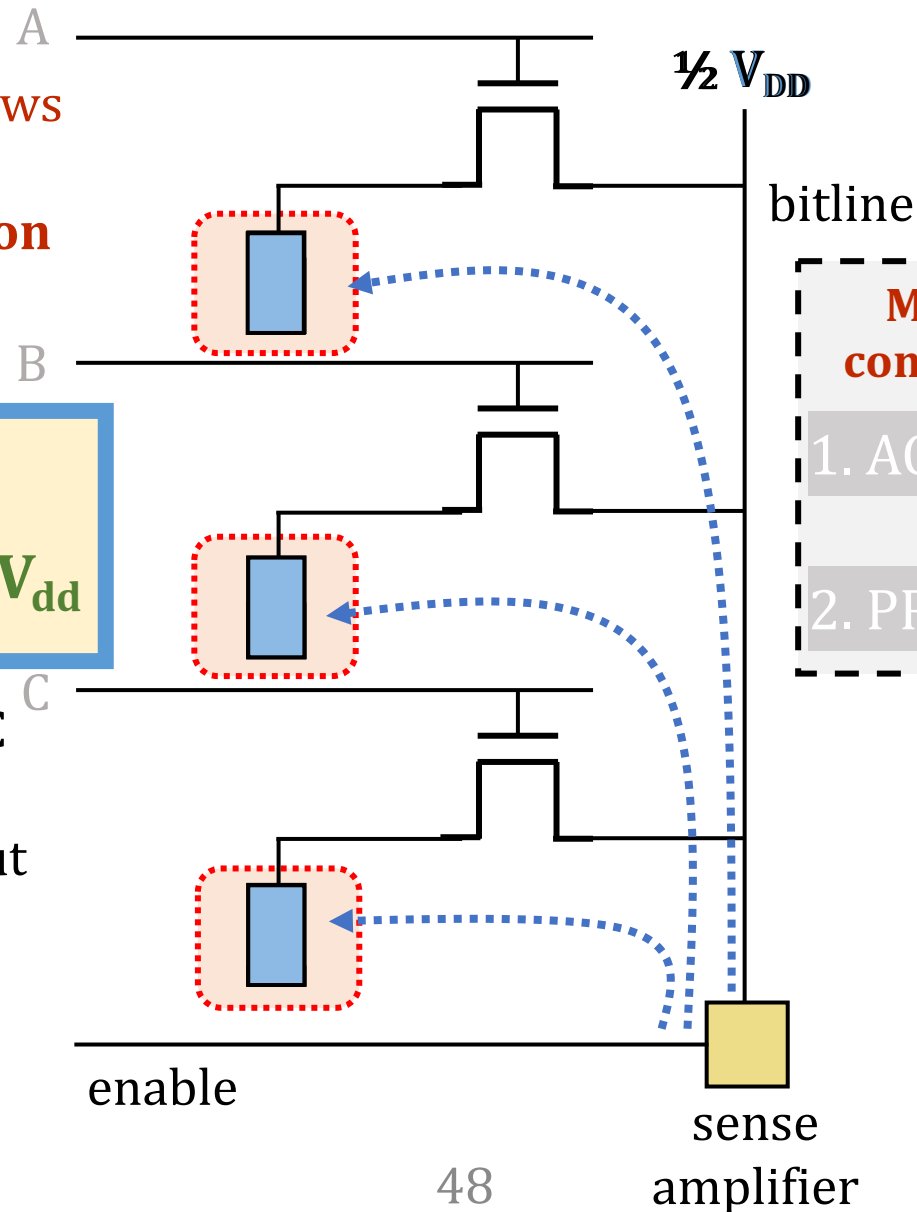
# Triple-Row Activation: Majority Function

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

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

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

4. **PRECHARGE** bitline for next access

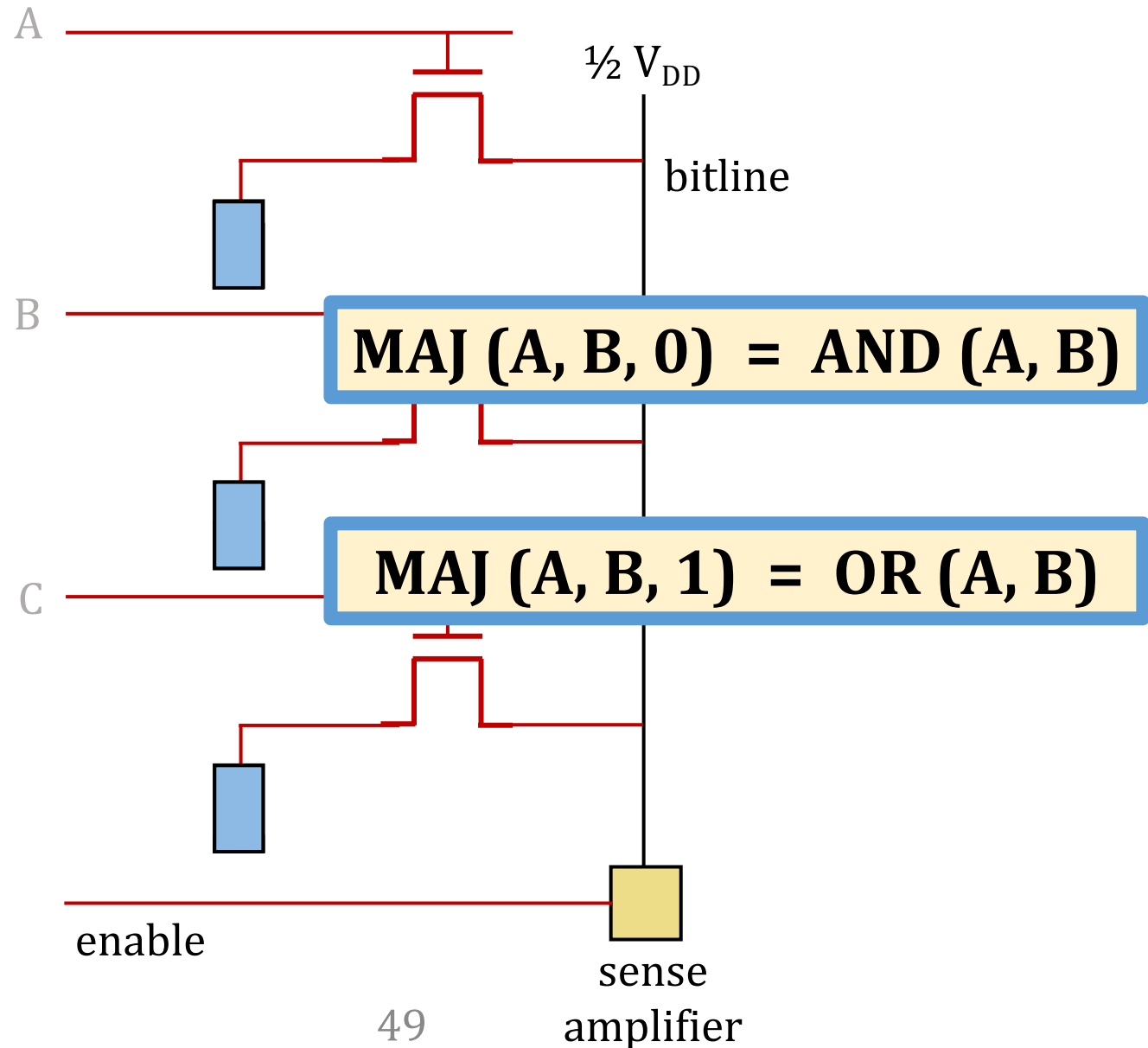


**Majority function command sequence<sup>3</sup>:**

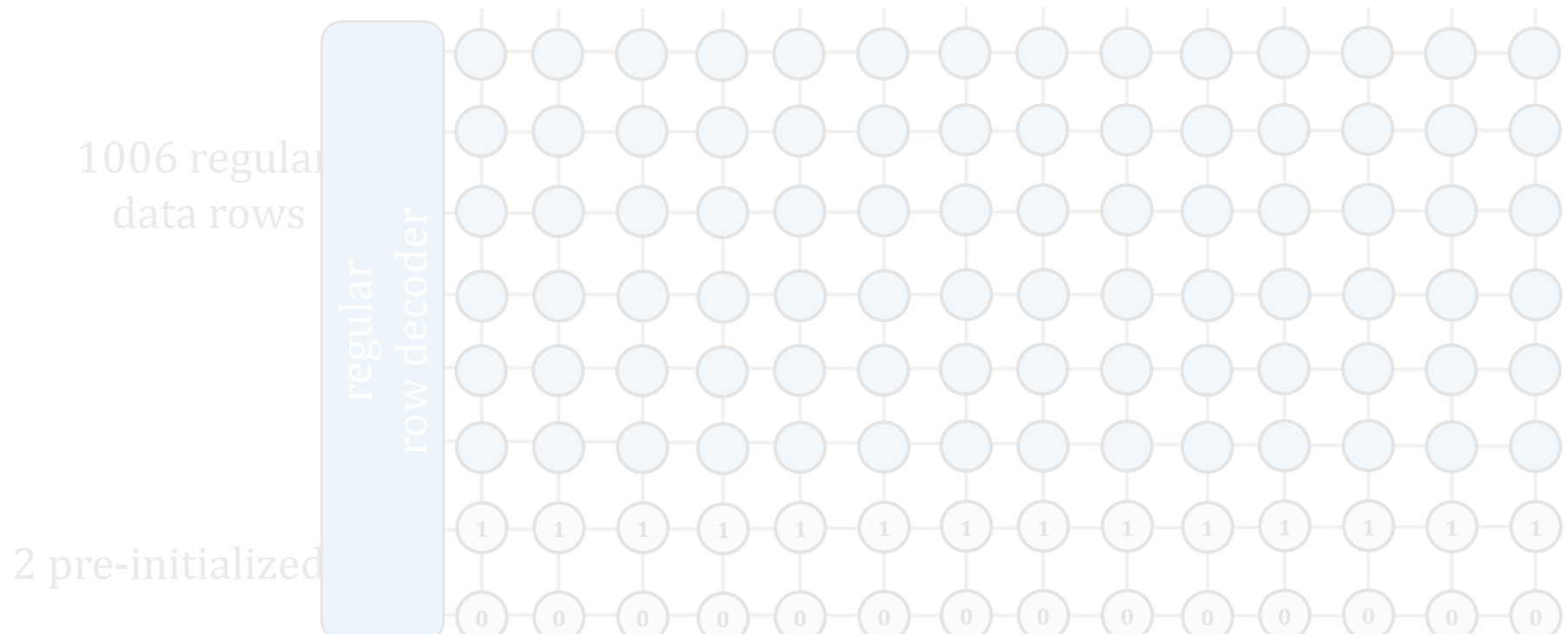
1. **ACTIVATE (ACT)**

2. **PRECHARGE (PRE)**

# Ambit: In-DRAM Bulk Bitwise AND/OR



# Ambit: Subarray Organization



**Less than 1% of overhead  
in existing DRAM chips**

# 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**

- Costly (e.g., area, power)

# 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

# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

**SIMDRAM Framework**

System Integration

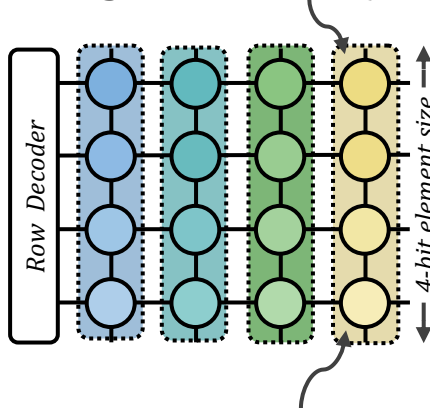
Evaluation

# SIMDRAM: PuM Substrate

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

## (1) Vertical data layout

most significant bit (MSB)



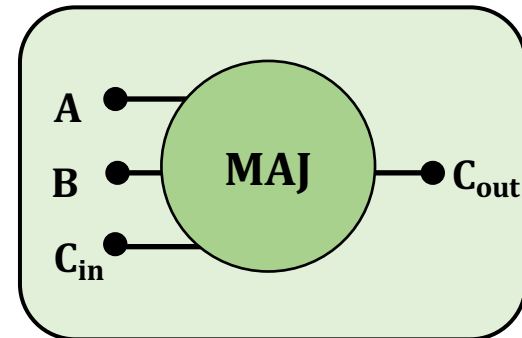
least significant bit (LSB)

**Pros compared to the conventional **horizontal layout**:**

- Implicit shift operation
- Massive parallelism

## (2) Majority-based computation

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



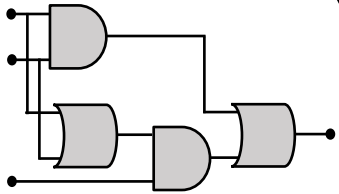
**Pros compared to **AND/OR/NOT-based** computation:**

- Higher performance
- Higher throughput
- Lower energy consumption

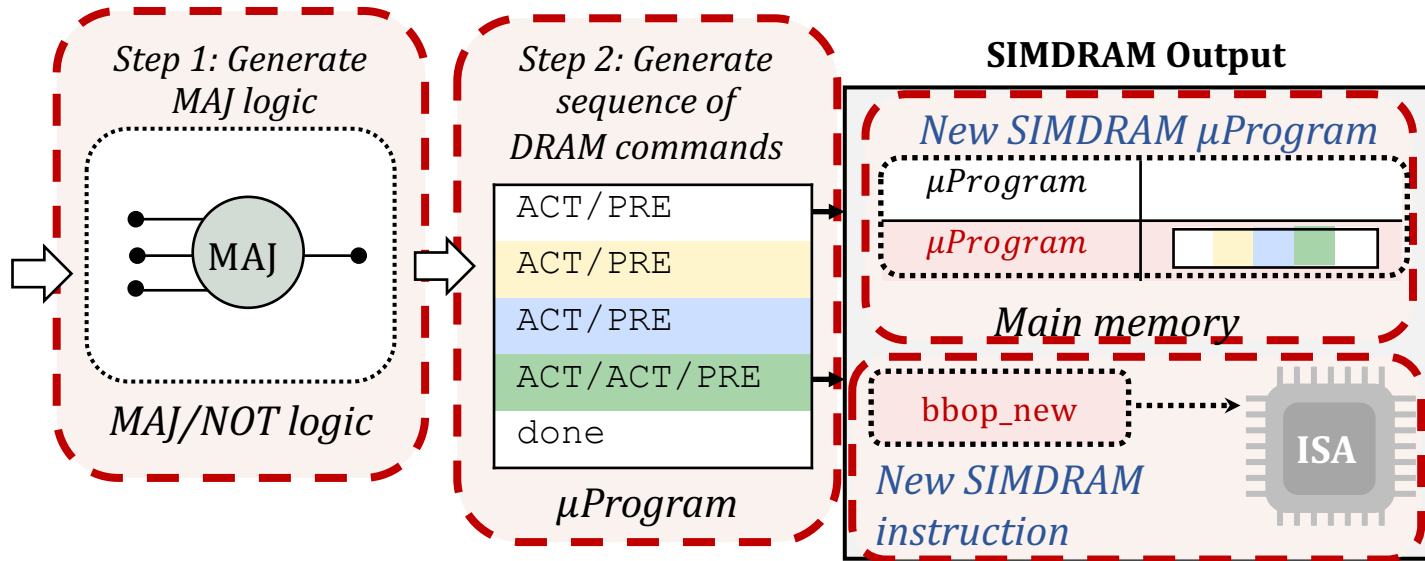
# SIMDRAM Framework

## User Input

*Desired operation*



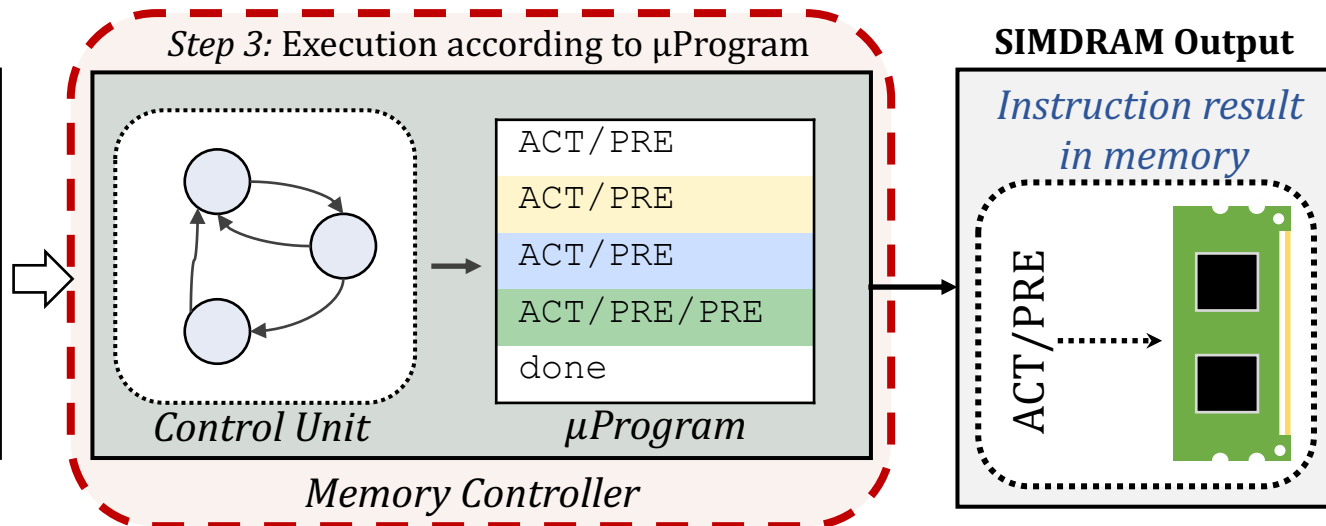
*AND/OR/NOT logic*



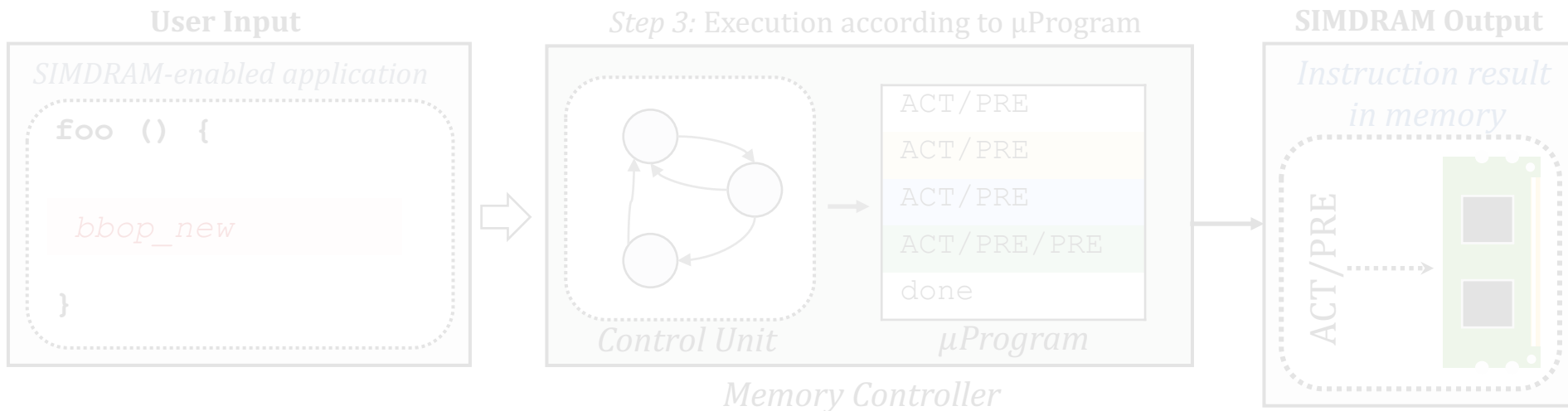
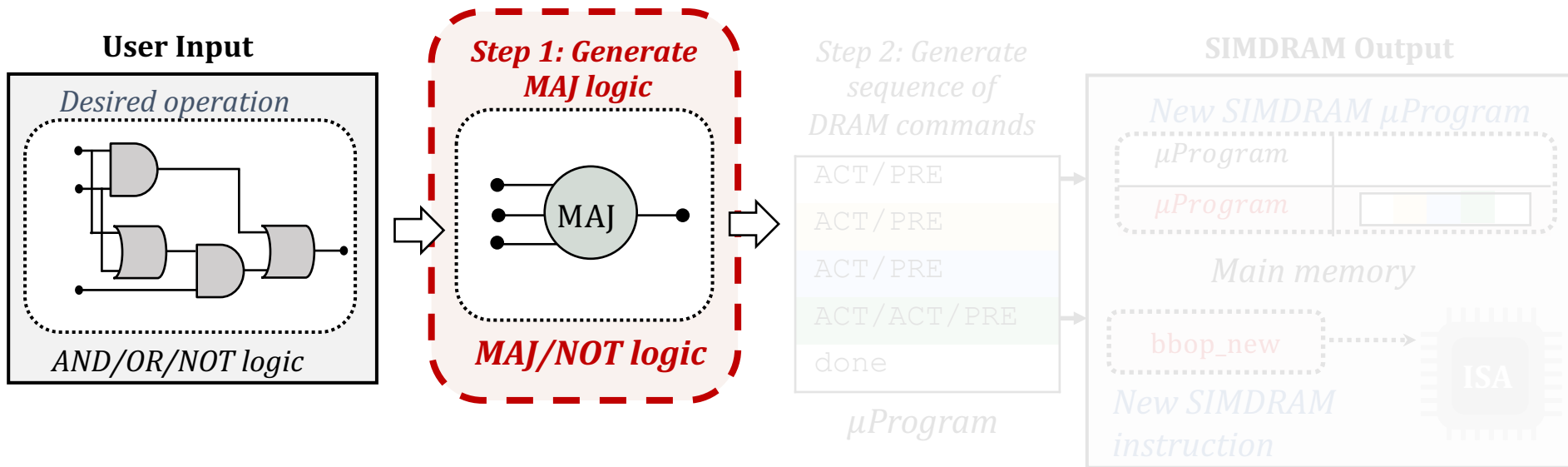
## User Input

*SIMDRAM-enabled application*

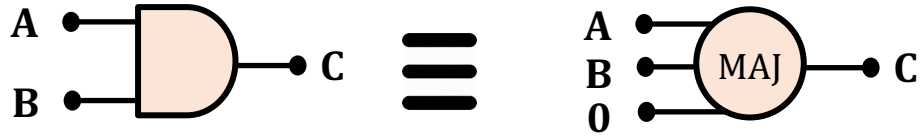
```
foo () {  
    bbop_new  
}
```



# SIMDRAM Framework: Step 1



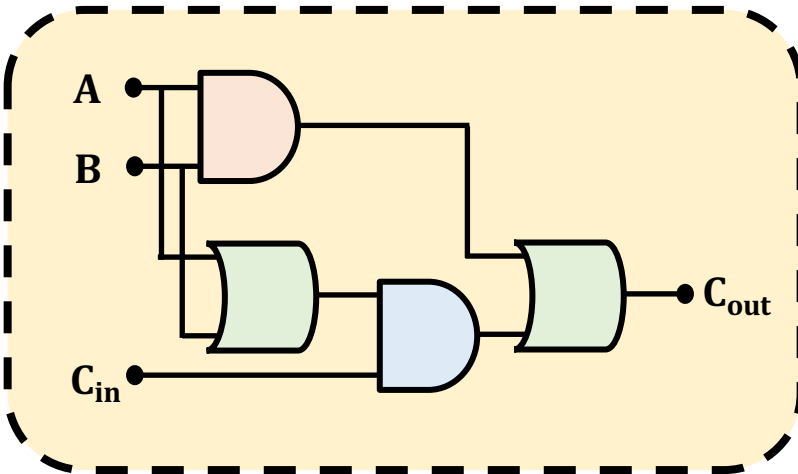
# Step 1: Naïve MAJ/NOT Implementation



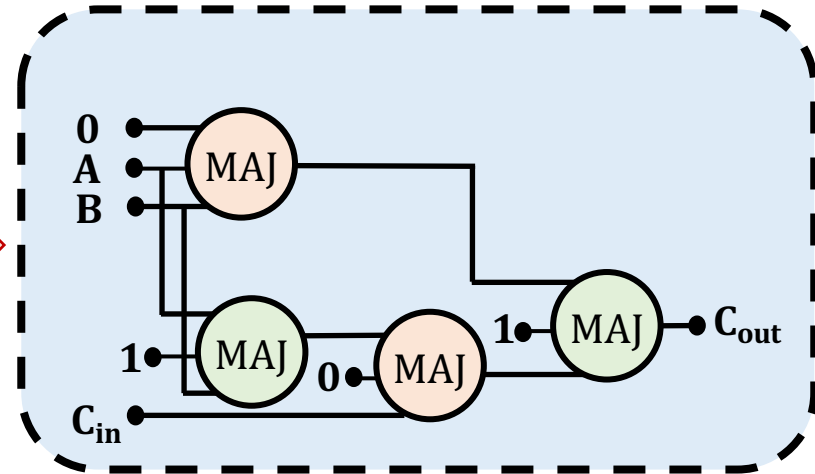
output is "1" only when  $A = B = "1"$



output is "0" only when  $A = B = "0"$

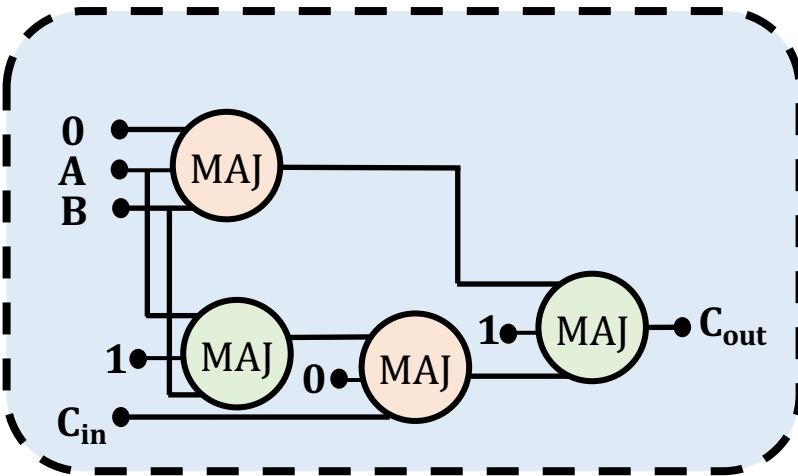


Part 1



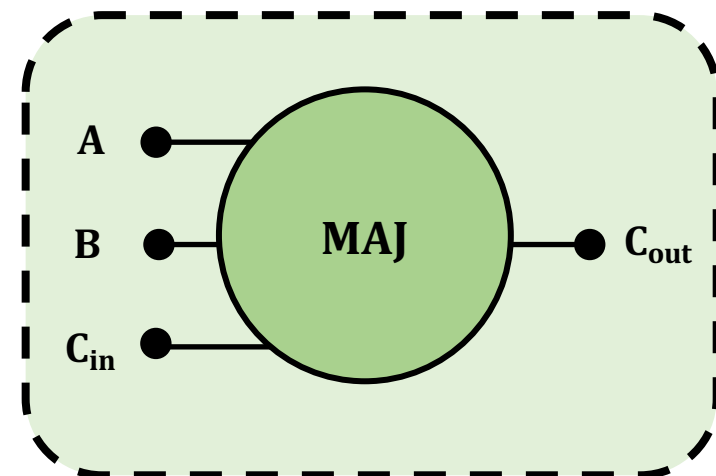
**Naïvely** converting **AND/OR/NOT-implementation** to **MAJ/NOT-implementation** leads to an **unoptimized circuit**

# Step 1: Efficient MAJ/NOT Implementation



Greedy  
optimization  
algorithm<sup>4</sup>

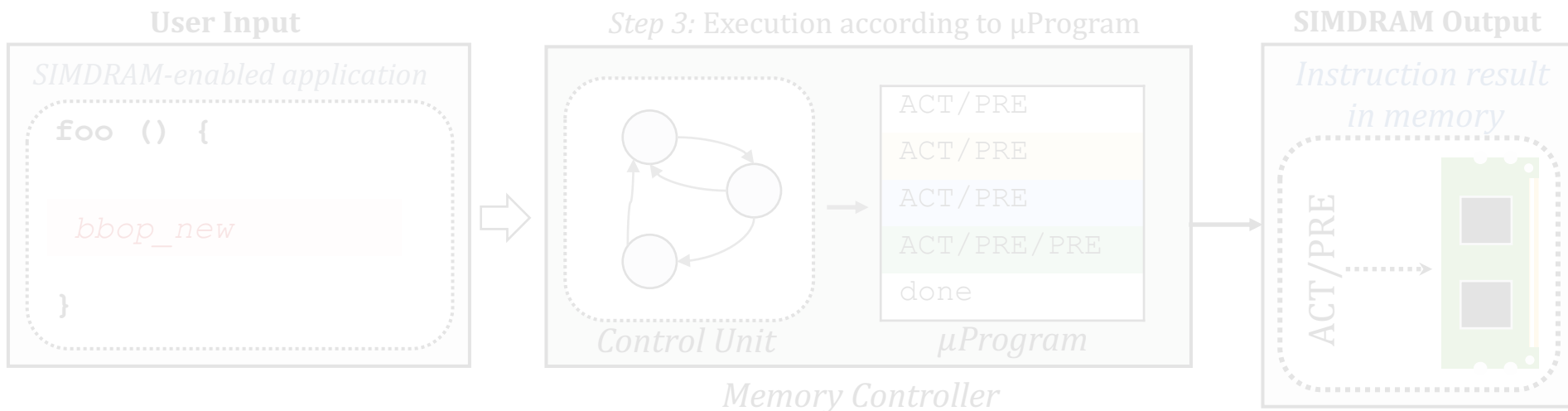
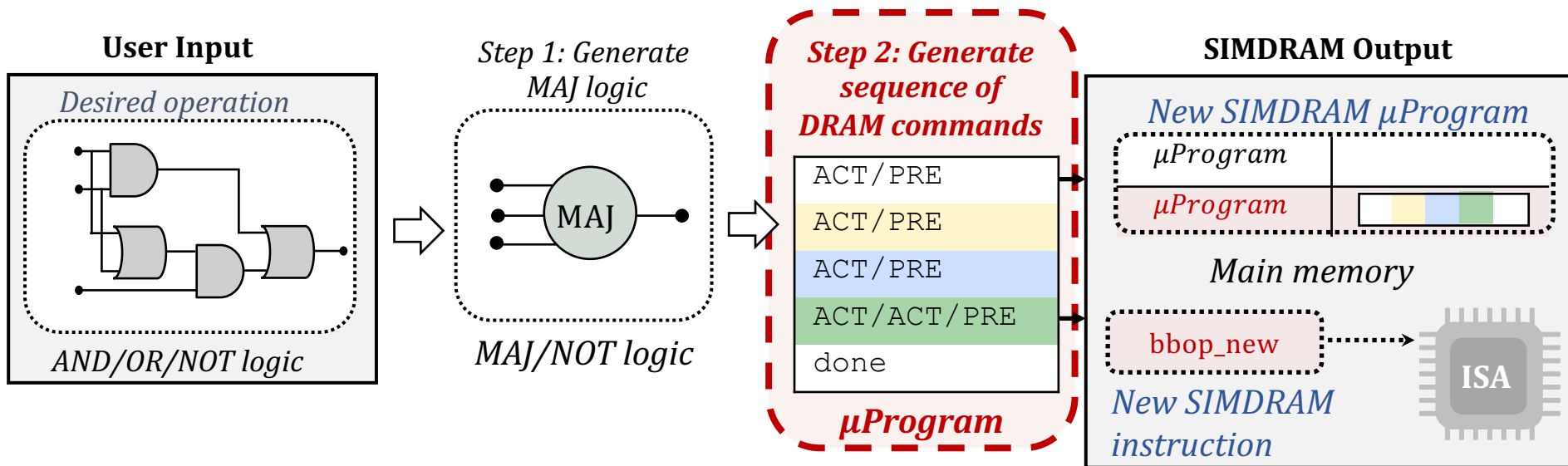
Part 2



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

<sup>4</sup> 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: $\mu$ Program Generation

- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate  $\mu$ Program

## Step 2: $\mu$ Program Generation

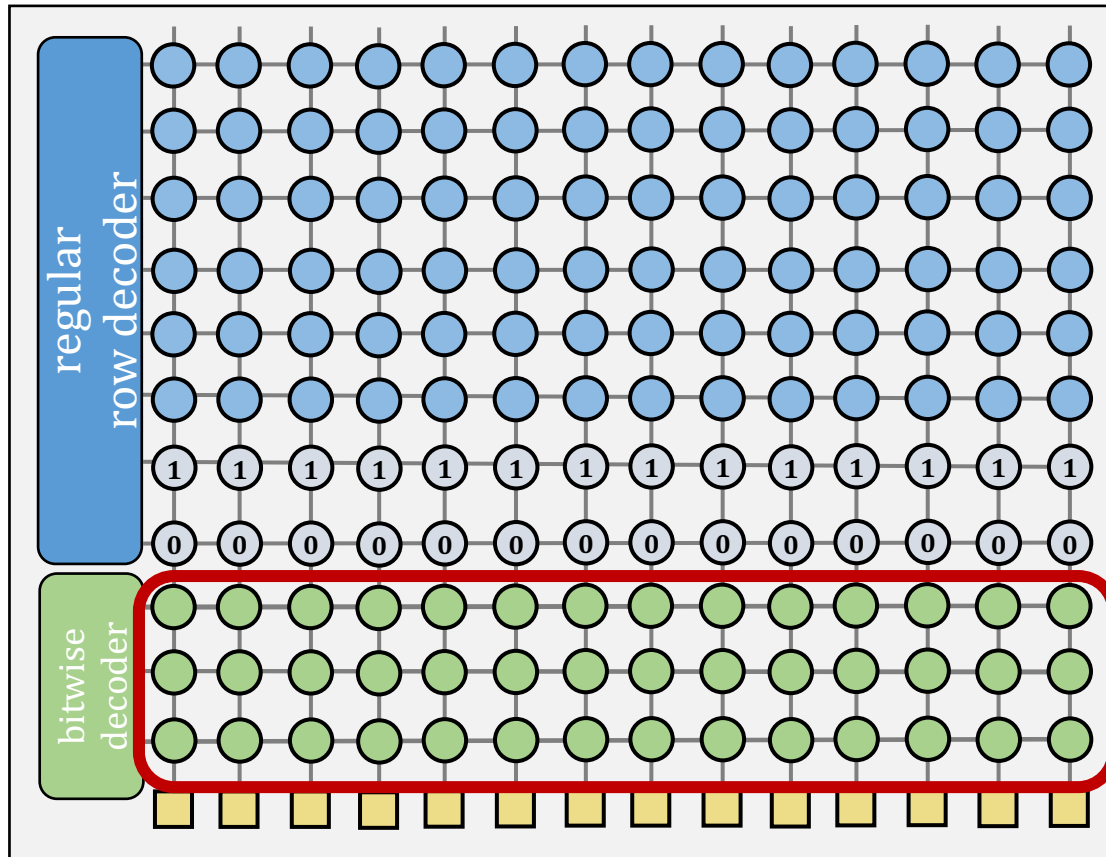
- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

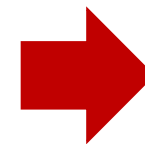
Task 2: Generate  $\mu$ Program

# Task 1: Allocating DRAM Rows to Operands

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



**Constraint 1:**  
**Limited** number of rows  
reserved for computation

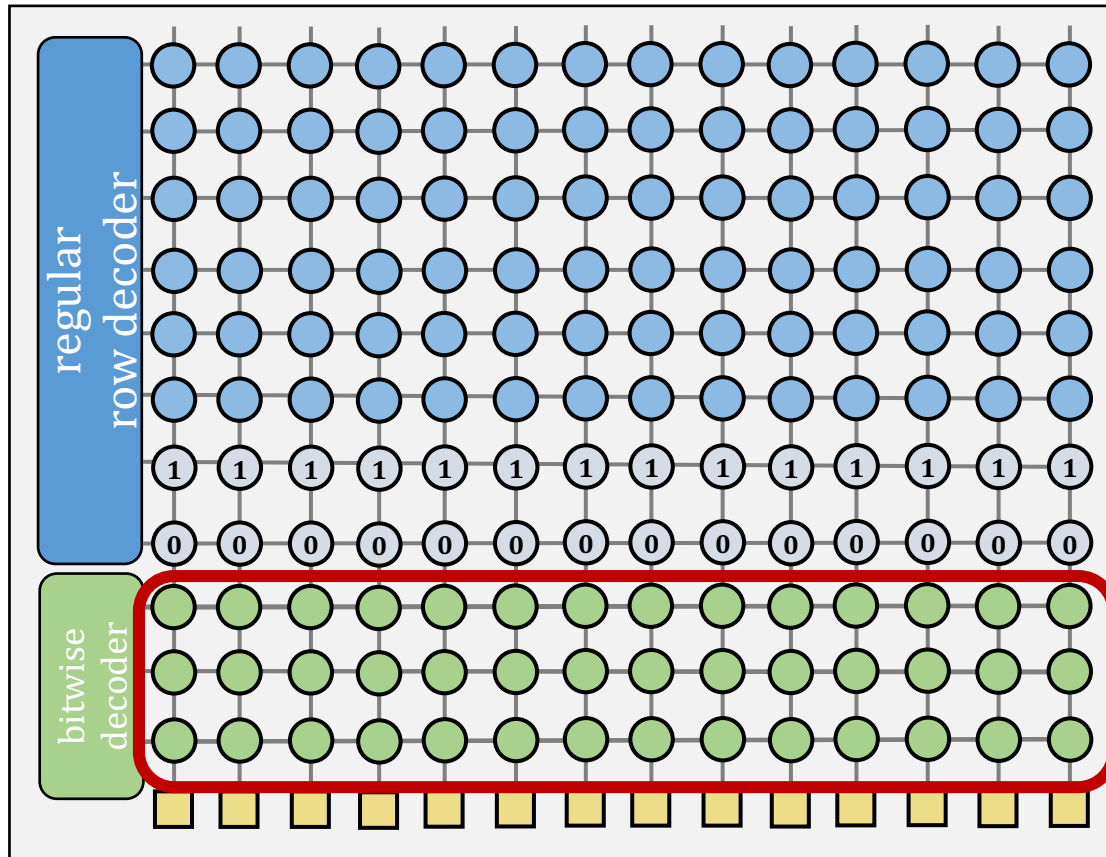


**Compute  
rows**

**subarray organization**

# Task 1: Allocating DRAM Rows to Operands

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



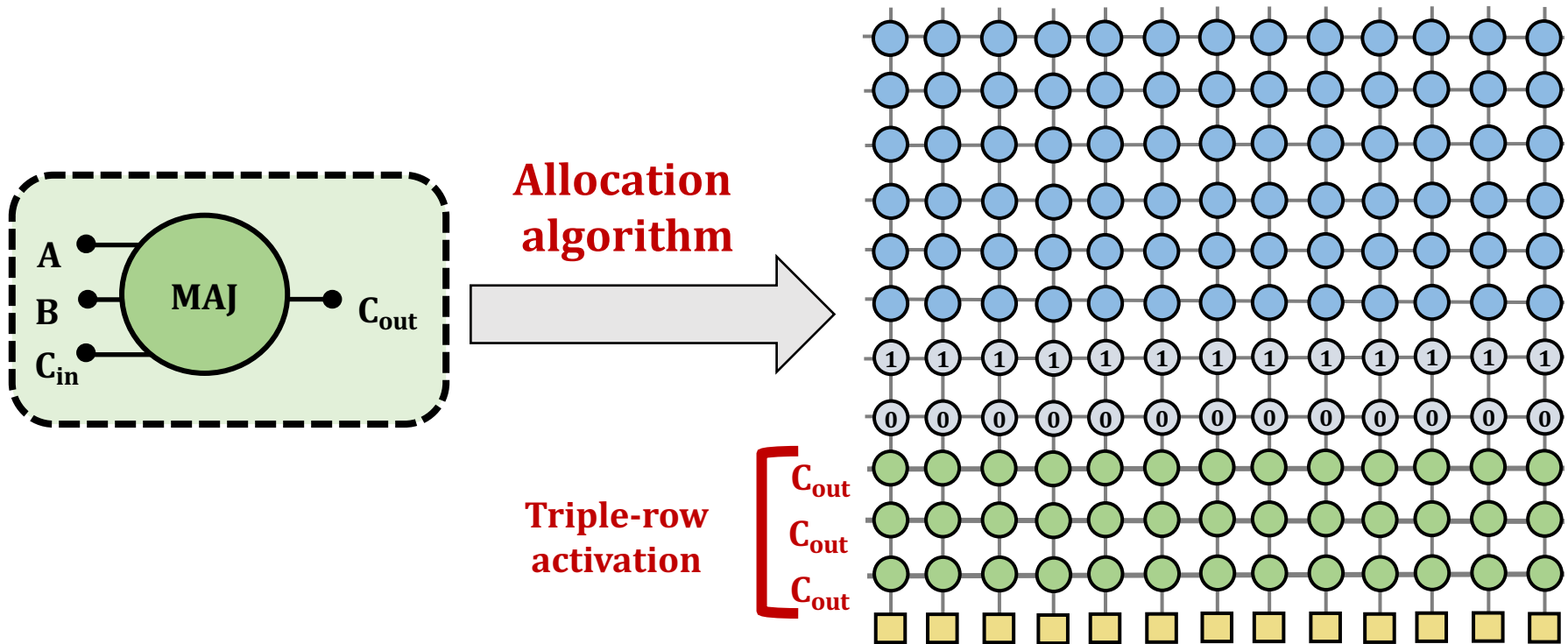
**Constraint 2:**  
**Destructive** behavior  
of triple-row activation

➔ **Overwritten  
with MAJ output**

**subarray organization**

# 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**



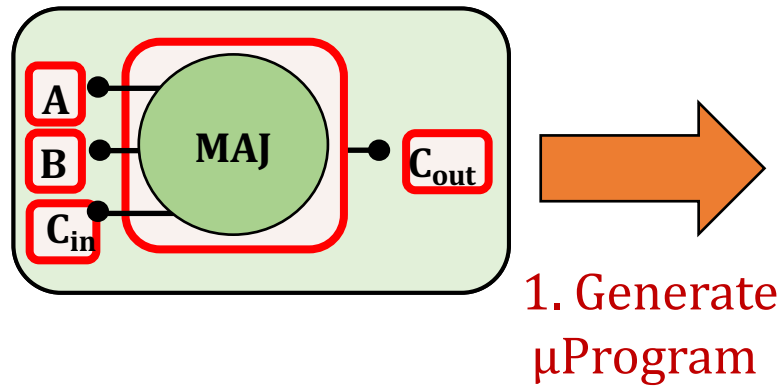
## Step 2: $\mu$ Program Generation

- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

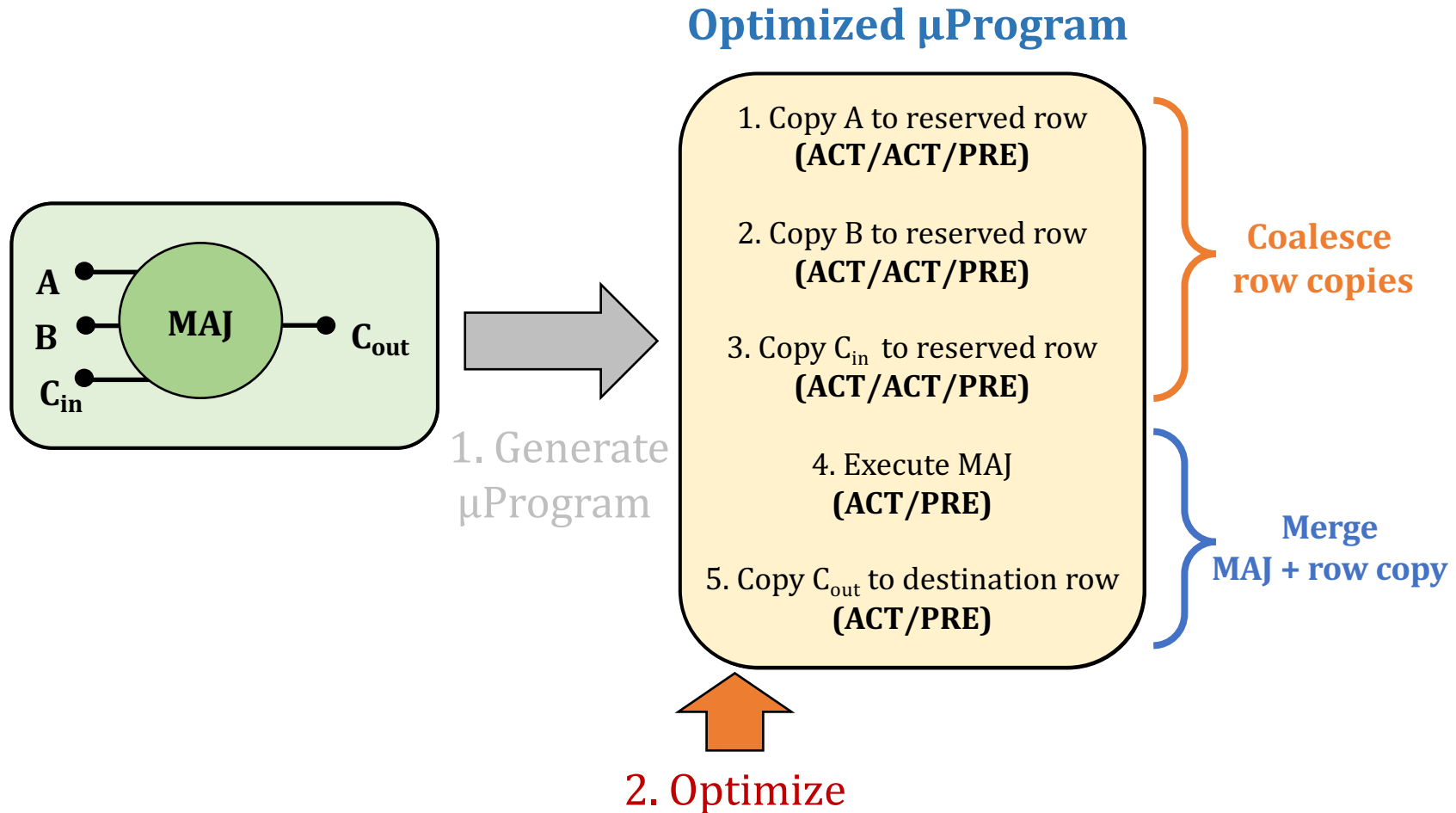
Task 1: Allocate DRAM rows to the operands

Task 2: Generate  $\mu$ Program

# Task 2: Generate an initial $\mu$ Program

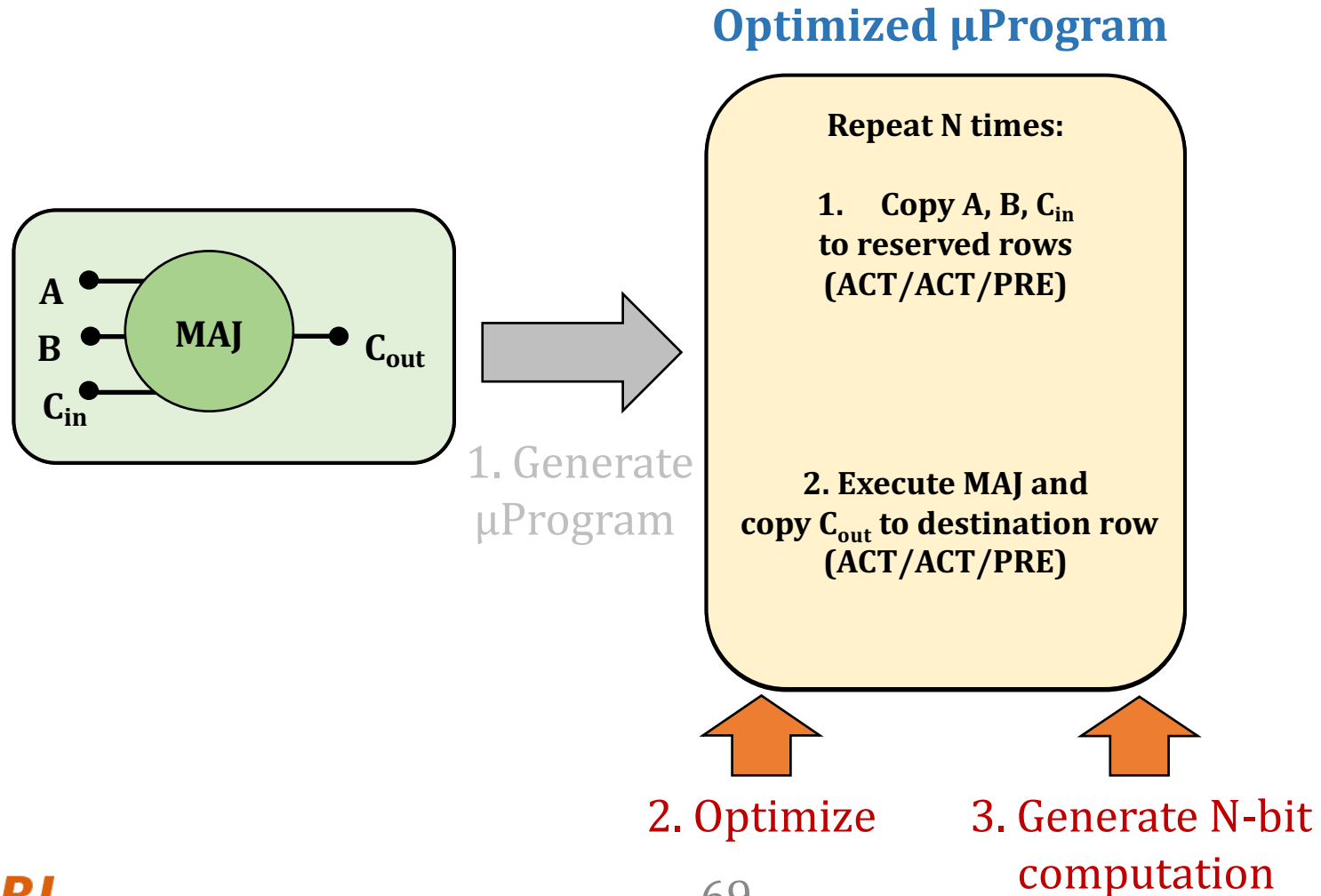


# Task 2: Optimize the $\mu$ Program



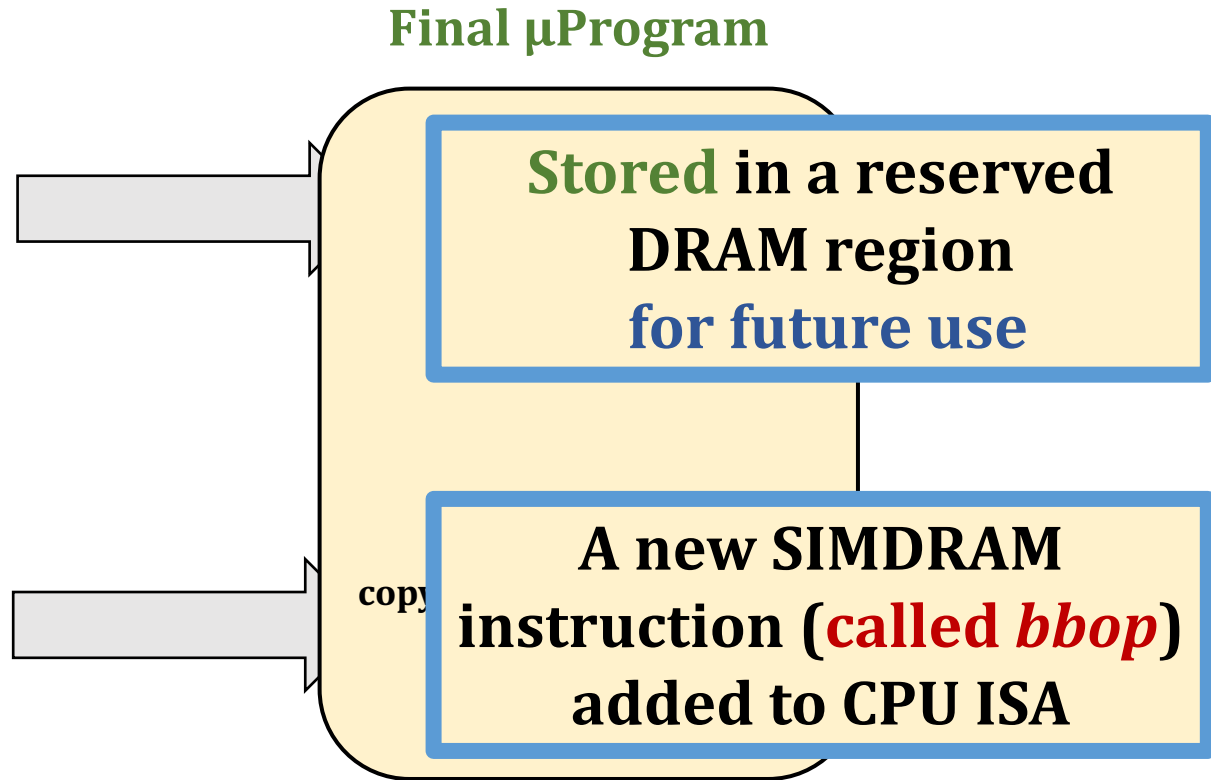
# Task 2: Generate N-bit Computation

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



# Task 2: Generate $\mu$ Program

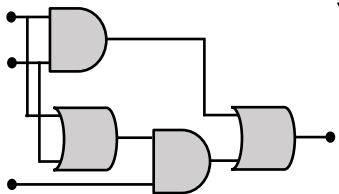
- **Final  $\mu$ Program** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion



# SIMDRAM Framework: Step 3

## User Input

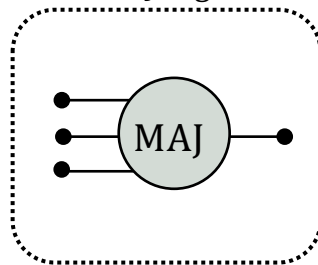
*Desired operation*



*AND/OR/NOT logic*



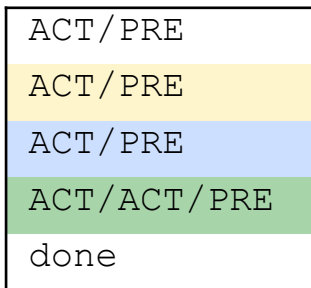
*Step 1: Generate  
MAJ logic*



*MAJ/NOT logic*



*Step 2: Generate  
sequence of  
DRAM commands*



*μProgram*

## SIMDRAM Output

*New SIMD RAM μProgram*

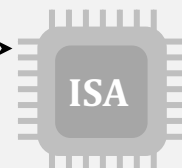
*μProgram*

*μProgram*

*Main memory*

*bbop\_new*

*New SIMD RAM  
instruction*



## User Input

*SIMDRAM-enabled application*

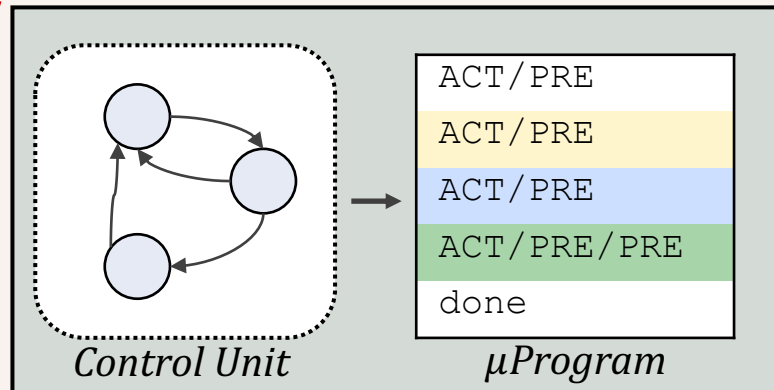
```
foo () {
```

*bbop\_new*

```
}
```



*Step 3: Execution according to μProgram*



*Control Unit*

*μProgram*

*Memory Controller*

## SIMDRAM Output

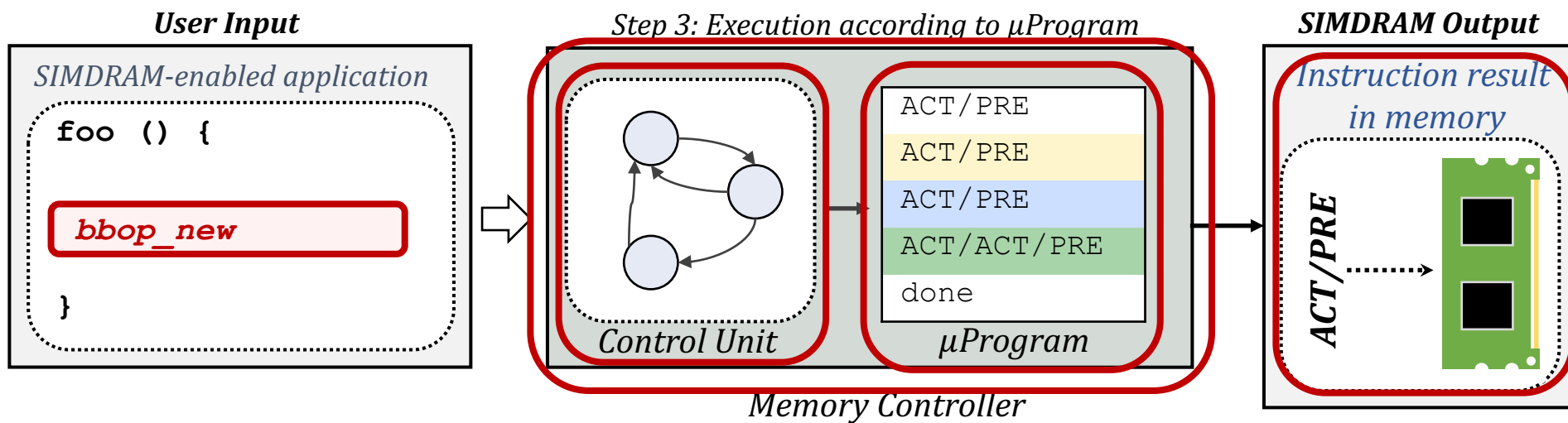
*Instruction result  
in memory*

*ACT/PRE*



# Step 3: $\mu$ Program Execution

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



# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

**System Integration**

Evaluation

# 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

# More in the Paper

## **SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM**

\*Nastaran Hajinazar<sup>1,2</sup>

Nika Mansouri Ghiasi<sup>1</sup>

<sup>1</sup>ETH Zürich

\*Geraldo F. Oliveira<sup>1</sup>

Minesh Patel<sup>1</sup>

Juan Gómez-Luna<sup>1</sup>

<sup>2</sup>Simon Fraser University

Sven Gregorio<sup>1</sup>

Mohammed Alser<sup>1</sup>

Onur Mutlu<sup>1</sup>

<sup>3</sup>University of Illinois at Urbana-Champaign

João Dinis Ferreira<sup>1</sup>

Saugata Ghose<sup>3</sup>

Handling page faults, address translation,  
coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework



# Outline

## 1. Introduction

## 2. Identifying Memory Bottlenecks

Methodology Overview

Application Profiling

Locality-Based Clustering

Memory Bottleneck Analysis

DAMOV Benchmark Suite

## 3. Enabling Complex Operations using DRAM

SIMDRAM Framework

System Integration

**Evaluation**

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

# Methodology: Workloads

## Evaluated:

- 16 complex in-DRAM operations:

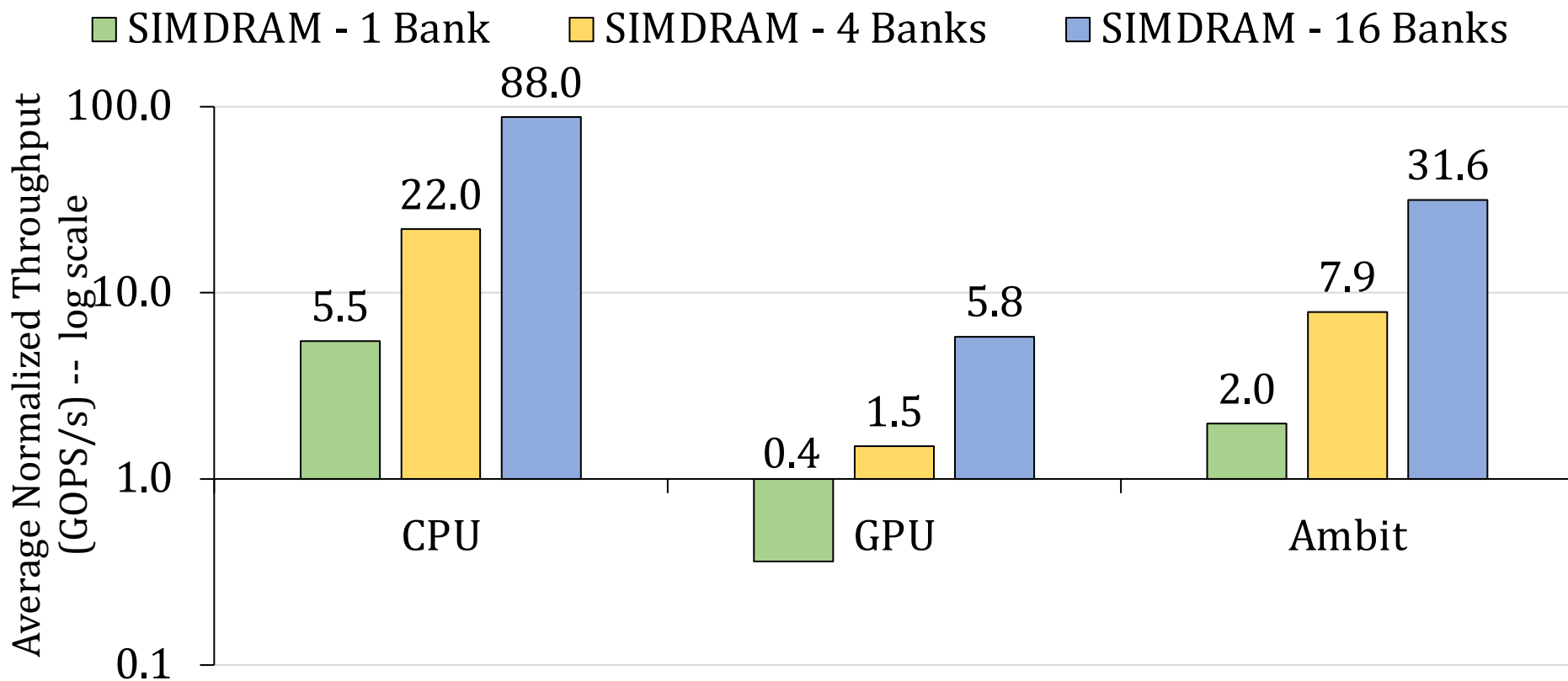
- Absolute
- Addition/Subtraction
- BitCount
- Equality/ Greater/Greater Equal
- Predication
- ReLU
- AND-/OR-/XOR-Reduction
- Division/Multiplication

- 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

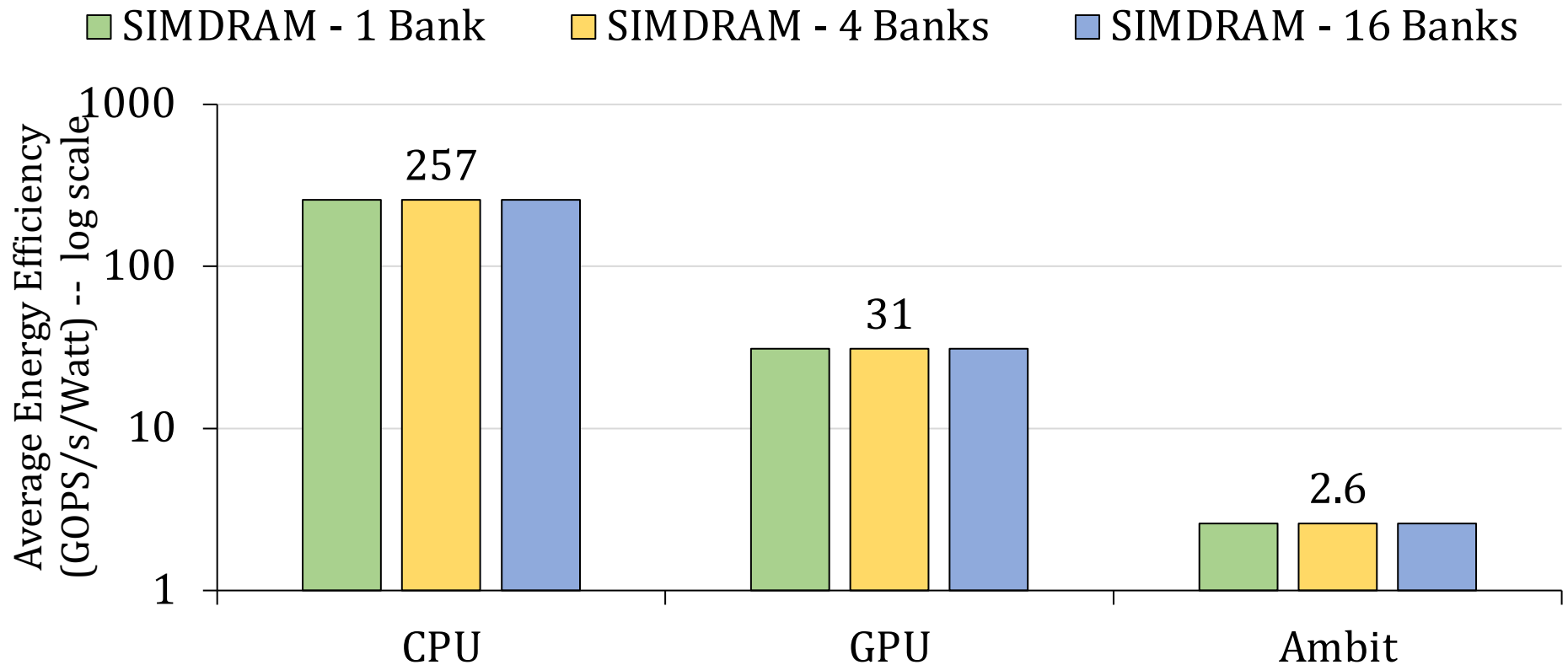
Average normalized throughput across all 16 SIMD RAM operations



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

# Energy Analysis

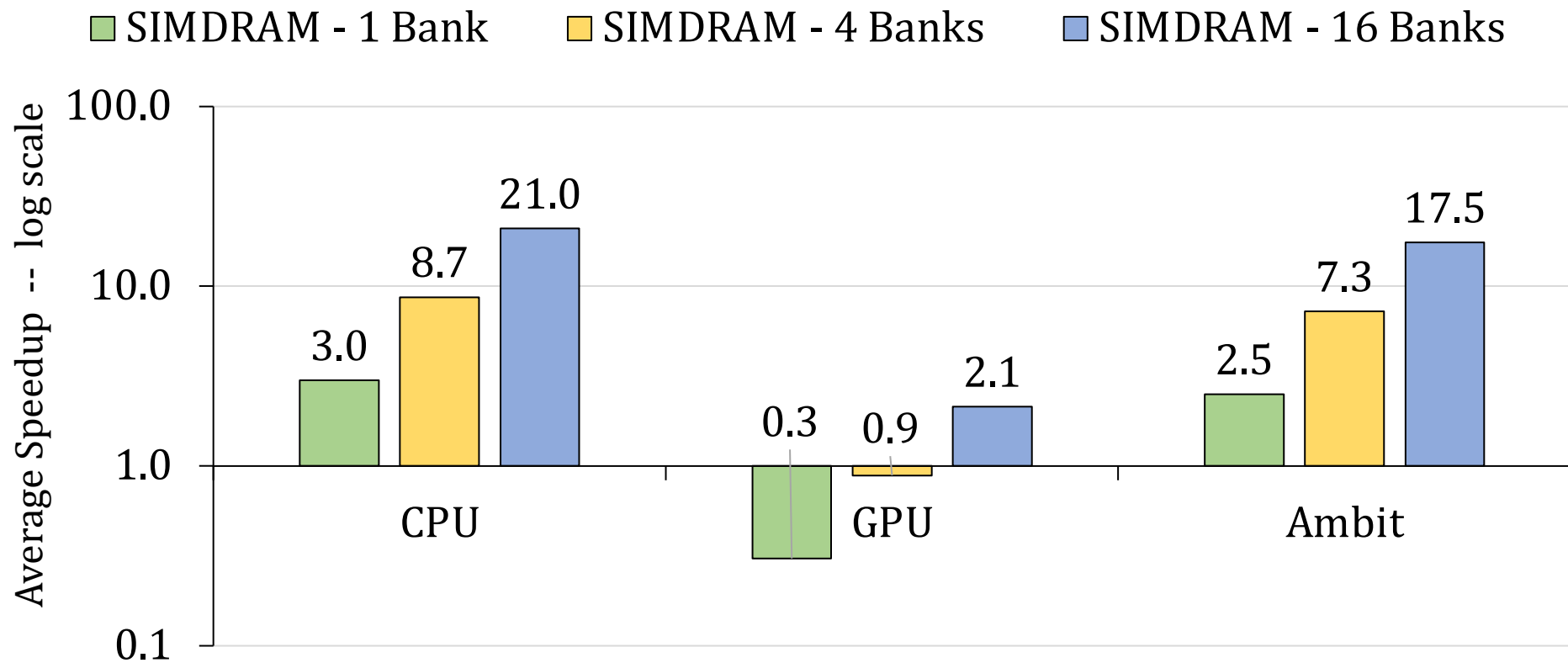
Average normalized energy efficiency across all 16 SIMD RAM operations



**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**

# Conclusion

- **Motivation**: Processing-using-Memory (PuM) architectures can effectively perform bulk bitwise computation
- **Problem**: Existing PuM architectures are not widely applicable
  - Support only a limited and specific set of operations
  - Lack the flexibility to support new operations
  - Require significant changes to the DRAM subarray
- **Goals**: Design a processing-using-DRAM framework that:
  - Efficiently implements complex operations
  - Provides the flexibility to support new desired operations
  - Minimally changes the DRAM architecture
- **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

# Methodologies, Workloads, and Tools for Processing-in-Memory: Enabling the Adoption of Data-Centric Architectures

**Geraldo F. Oliveira**

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**ISVLSI  
2022**

**SAFARI**

**ETH** *zürich*

**I** UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

P&S Ramulator  
29.04.2022

**Geraldo F. Oliveira**

Juan Gómez-Luna   Lois Orosa   Saugata Ghose

Nandita Vijaykumar   Ivan Fernandez   Mohammad Sadrosadati

Onur Mutlu

**SAFARI**

**ETH** *Zürich*



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN



UNIVERSITY OF  
**TORONTO**



UNIVERSIDAD  
DE MÁLAGA

# Executive Summary

- **Problem**: Data movement is a major bottleneck in modern systems. However, it is **unclear** how to identify:
  - **different sources** of data movement bottlenecks
  - the **most suitable** mitigation technique (e.g., caching, prefetching, near-data processing) for a given data movement bottleneck
- **Goals**:
  1. Design a methodology to **identify** sources of data movement bottlenecks
  2. **Compare** compute- and memory-centric data movement mitigation techniques
- **Key Approach**: Perform a large-scale application characterization to identify **key metrics** that reveal the sources of data movement bottlenecks
- **Key Contributions**:
  - **Experimental characterization** of 77K functions across 345 applications
  - A **methodology** to characterize applications based on data movement bottlenecks and their relation with different data movement mitigation techniques
  - **DAMOV**: a **benchmark suite** with **144 functions** for data movement studies
  - **Four case-studies** to highlight DAMOV's applicability to open research problems

# Outline

1. Data Movement Bottlenecks
2. Methodology Overview
3. Application Profiling
4. Locality-Based Clustering
5. Memory Bottleneck Analysis
6. Case Studies

# Outline

## 1. Data Movement Bottlenecks

## 2. Methodology Overview

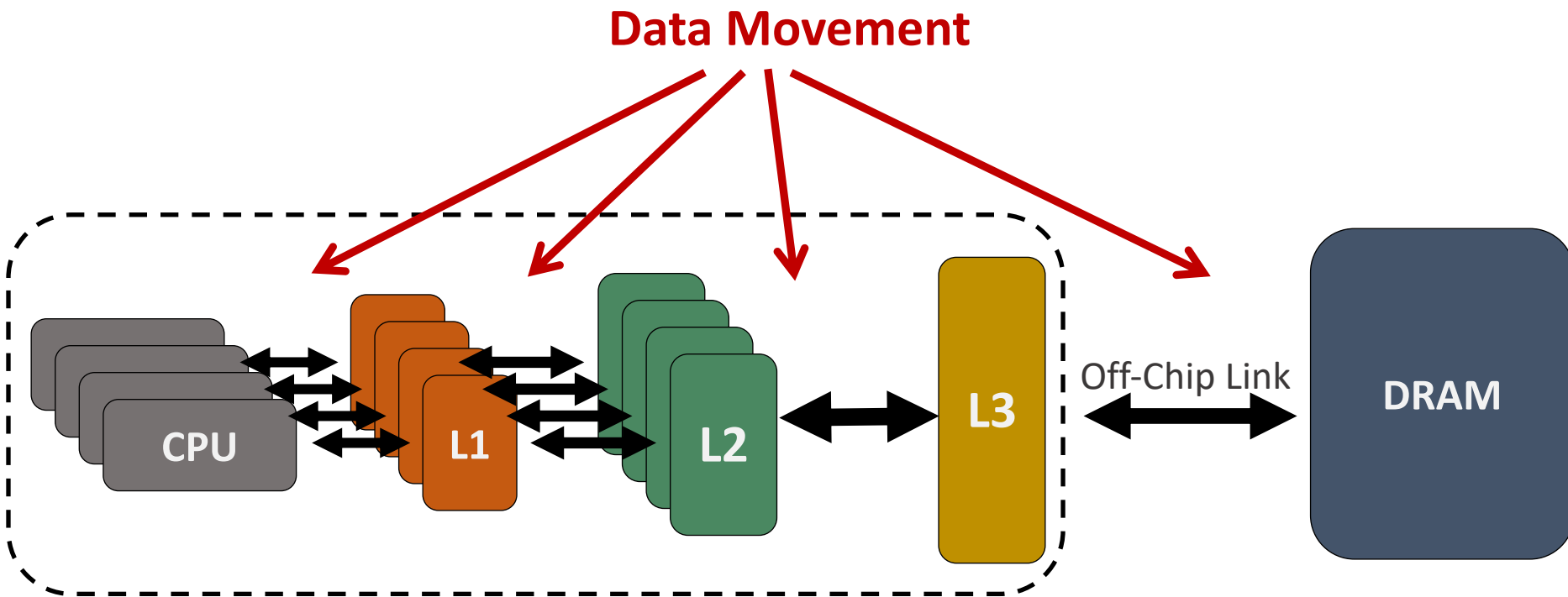
## 3. Application Profiling

## 4. Locality-Based Clustering

## 5. Memory Bottleneck Analysis

## 6. Case Studies

# Data Movement Bottlenecks (1/2)

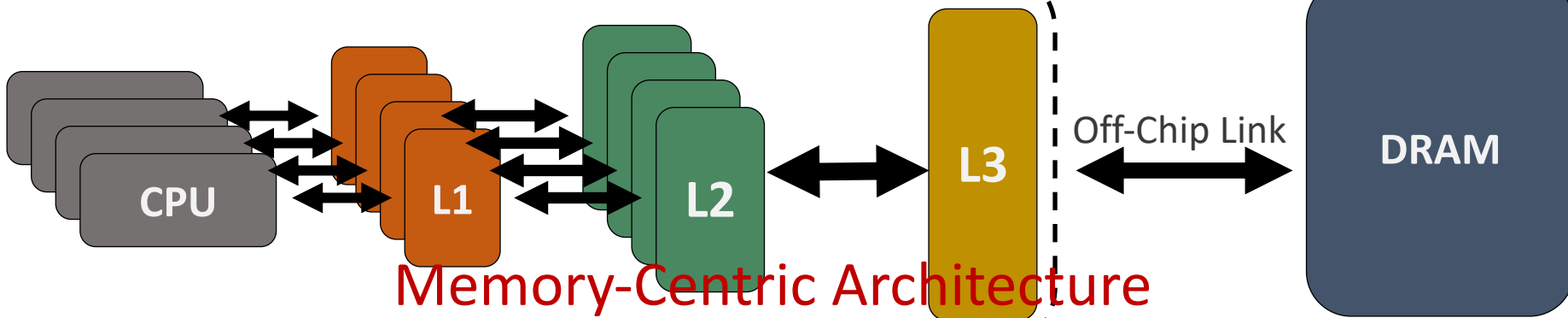


**Data movement bottlenecks** happen because of:

- Not enough data **locality** → ineffective use of the cache hierarchy
- Not enough **memory bandwidth**
- High average **memory access time**

# Data Movement Bottlenecks (2/2)

## Compute-Centric Architecture

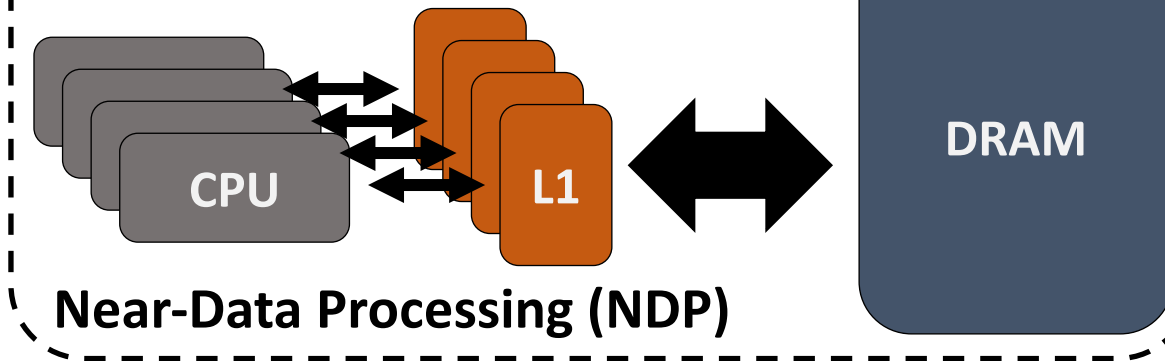


## Memory-Centric Architecture

- Abundant DRAM bandwidth

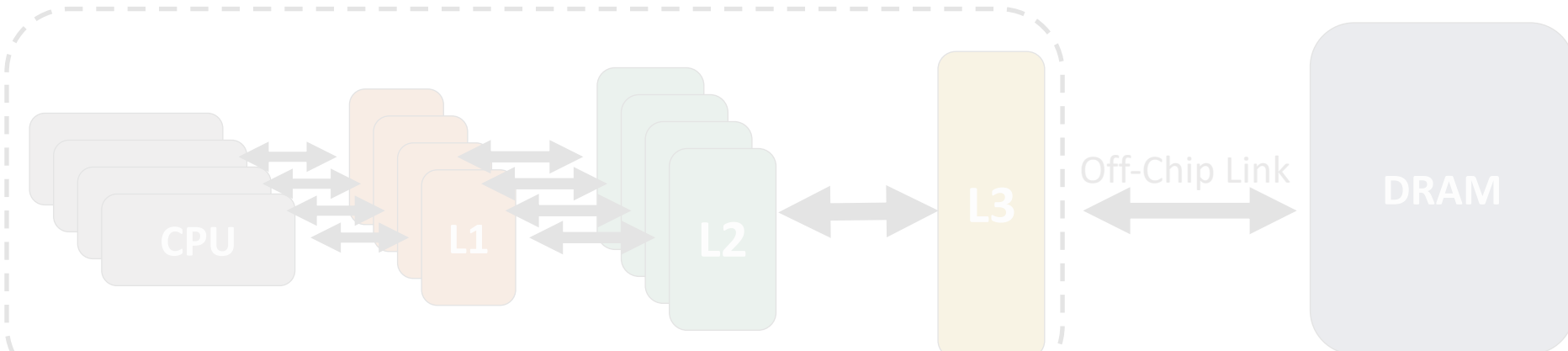


- Shorter average memory access time



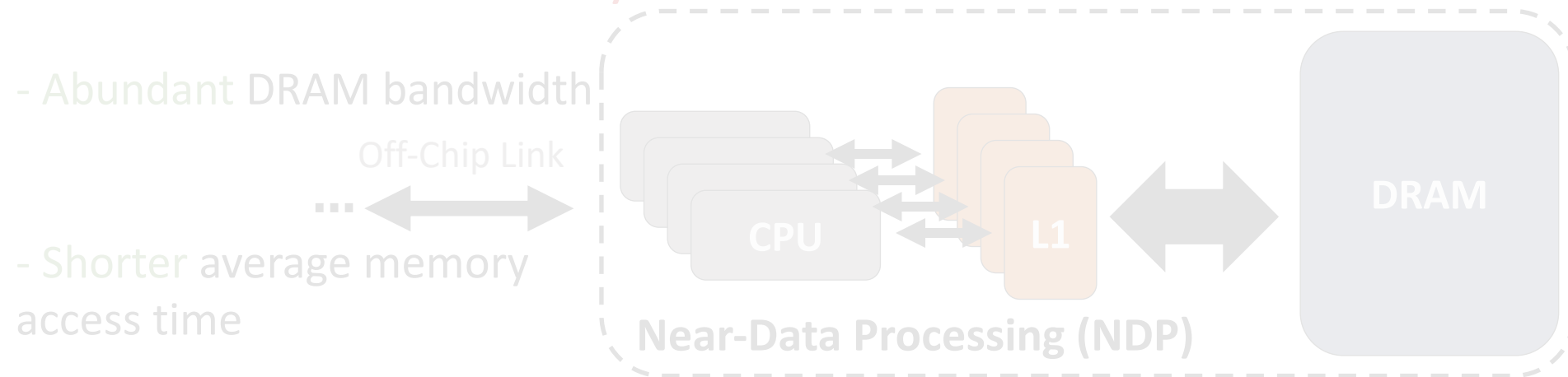
# Near-Data Processing (1/2)

## Compute-Centric Architecture



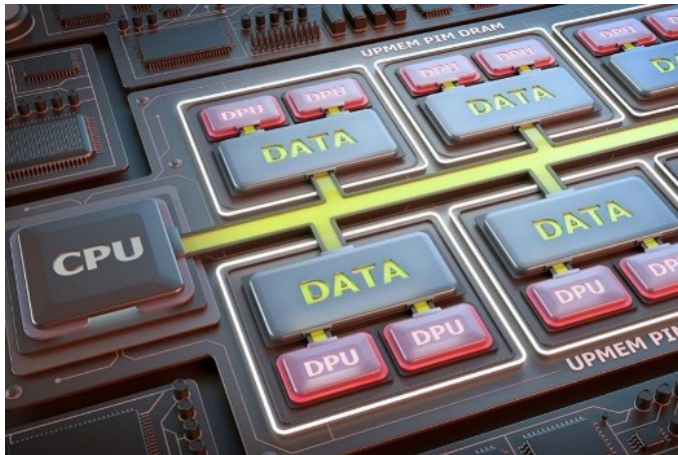
The goal of Near-Data Processing (NDP) is  
**to mitigate data movement**

## Memory-Centric Architecture



# Near-Data Processing (2/2)

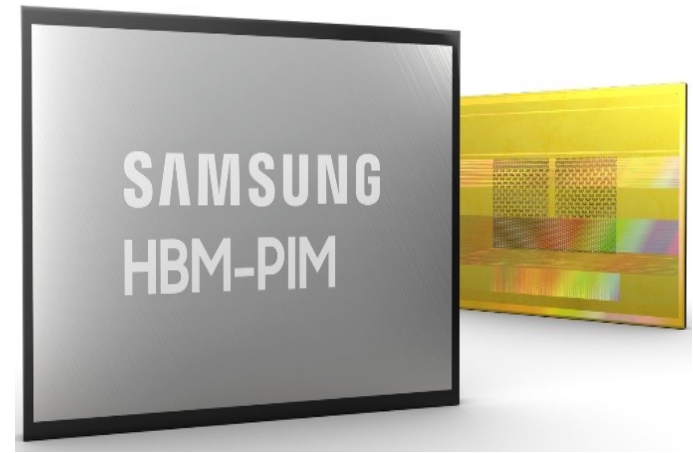
## UPMEM (2019)



Near-DRAM-banks processing  
for general-purpose computing

**0.9 TOPS compute throughput<sup>1</sup>**

## Samsung FIMDRAM (2021)

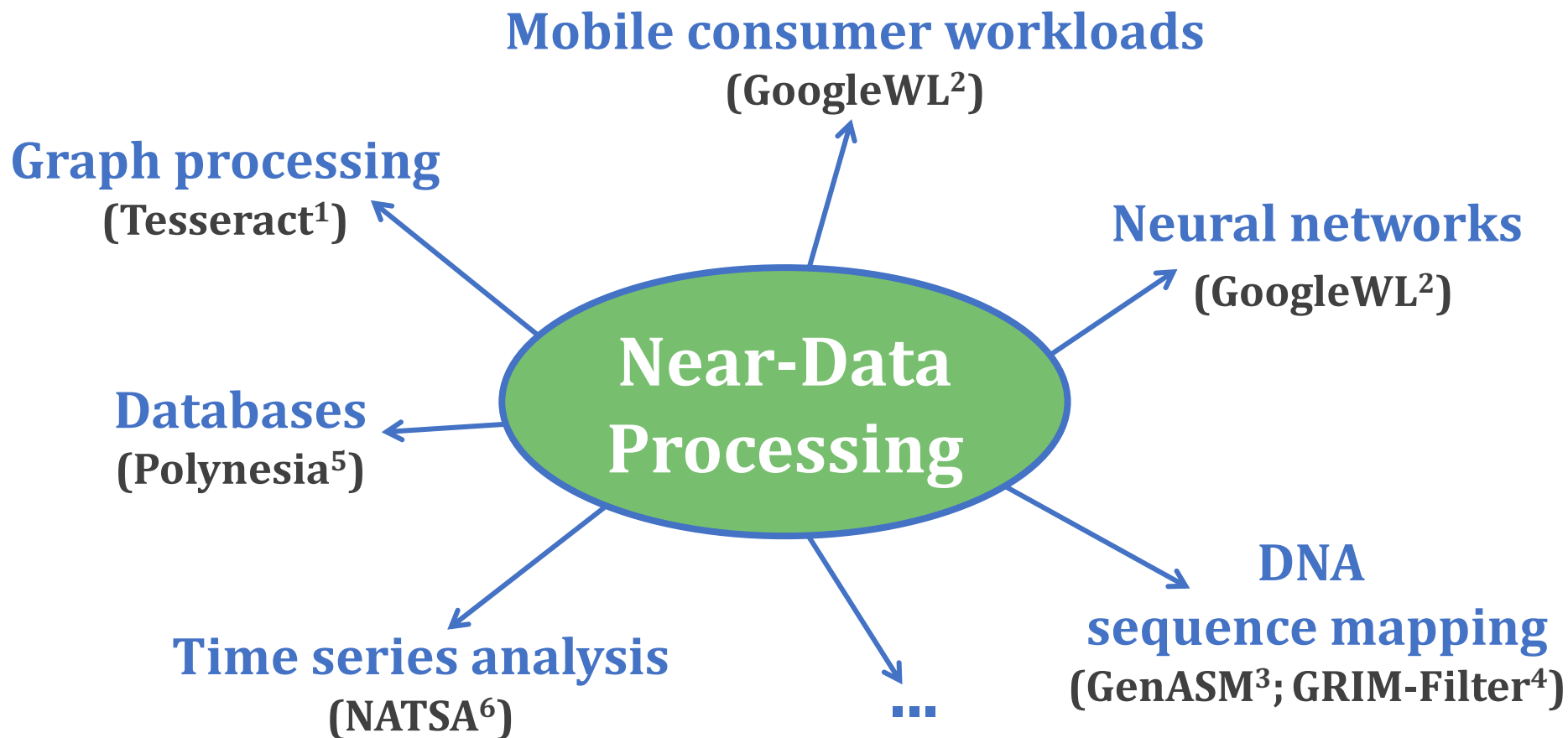


Near-DRAM-banks processing  
for neural networks

**1.2 TFLOPS compute throughput<sup>2</sup>**

The goal of Near-Data Processing (NDP) is  
**to mitigate data movement**

# When to Employ Near-Data Processing?



[1] Ahn+, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," ISCA, 2015

[2] Boroumand+, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ASPLOS, 2018

[3] Cali+, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," MICRO, 2020

[4] Kim+, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," BMC Genomics, 2018

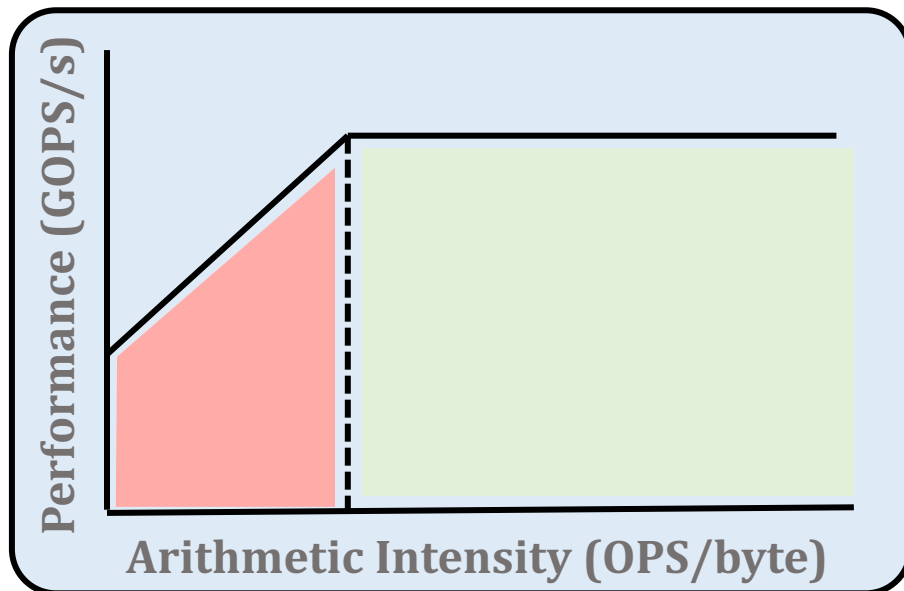
[5] Boroumand+, "Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design," arXiv:2103.00798 [cs.AR], 2021

[6] Fernandez+, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," ICCD, 2020

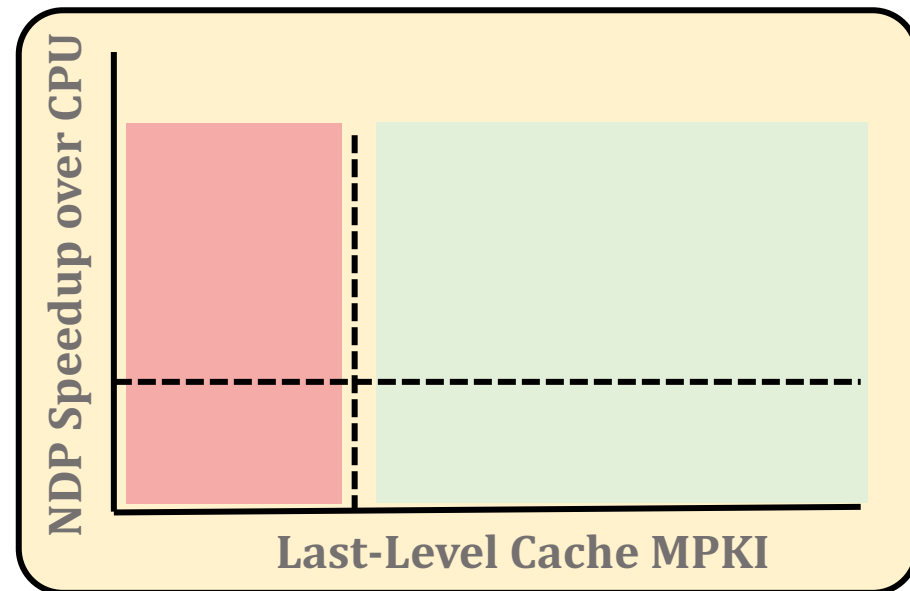
# Identifying Memory Bottlenecks

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP
- Existing approaches are not comprehensive enough

**Roofline model**

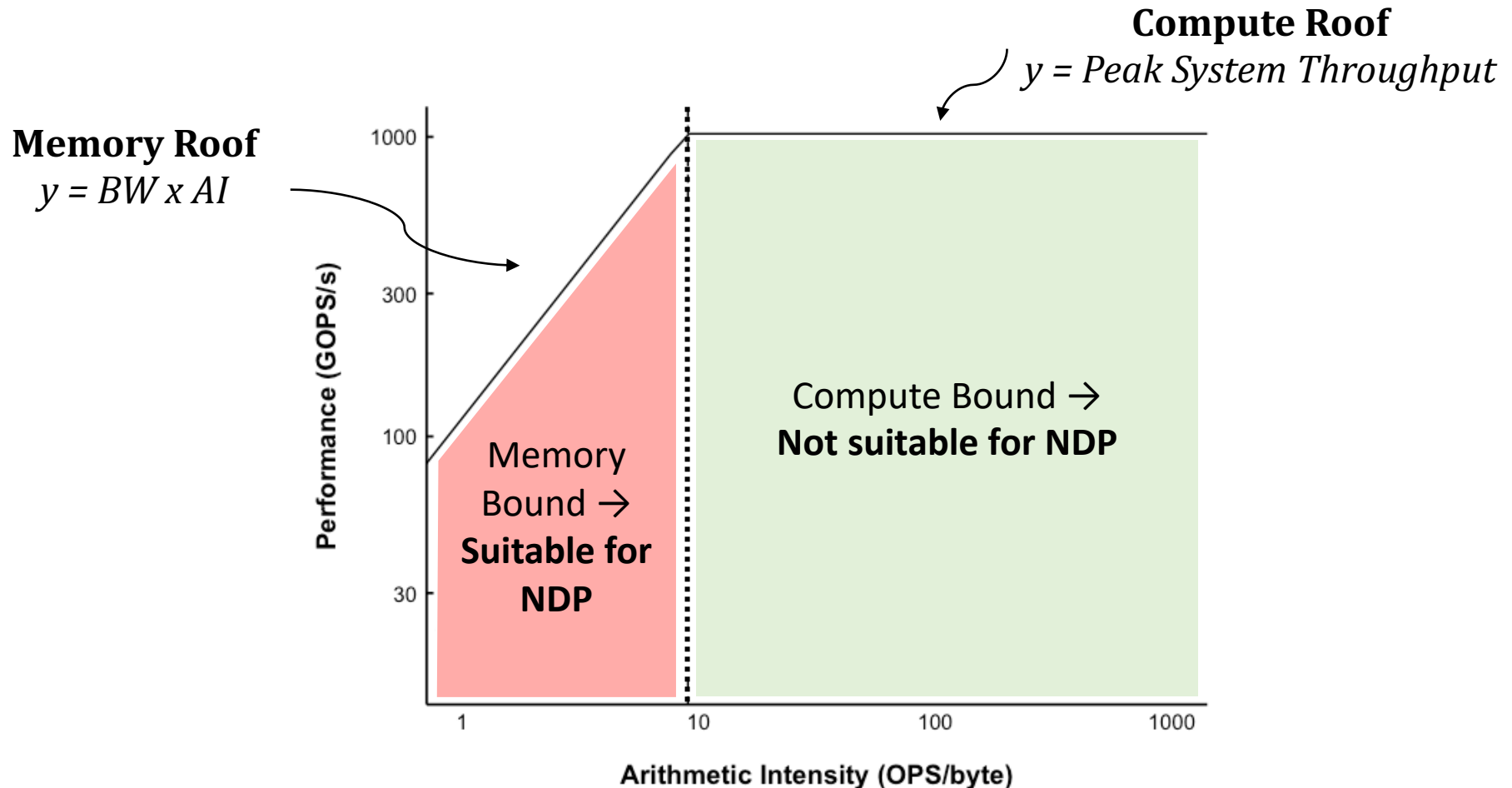


**High LLC MPKI**



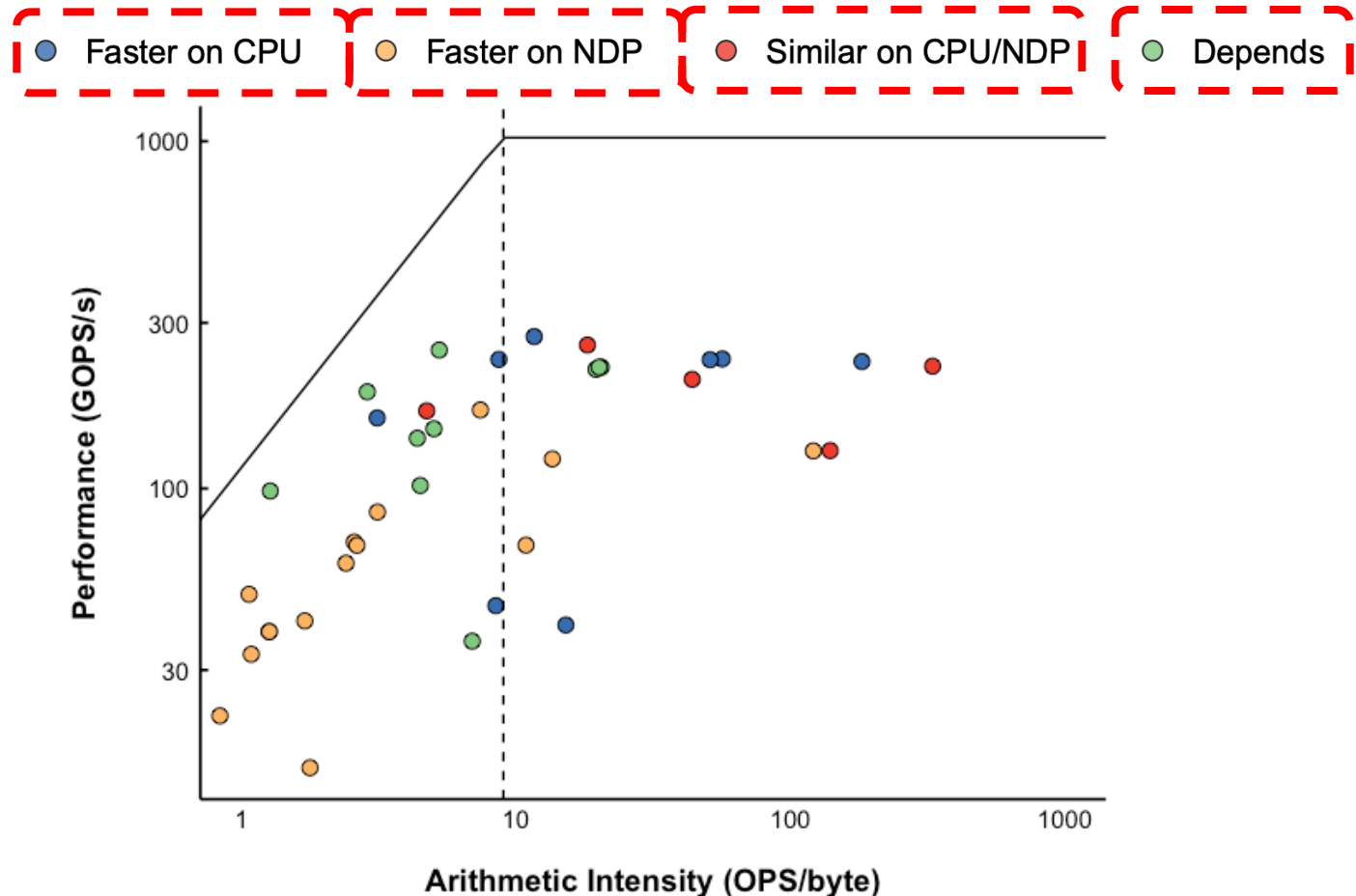
# Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units



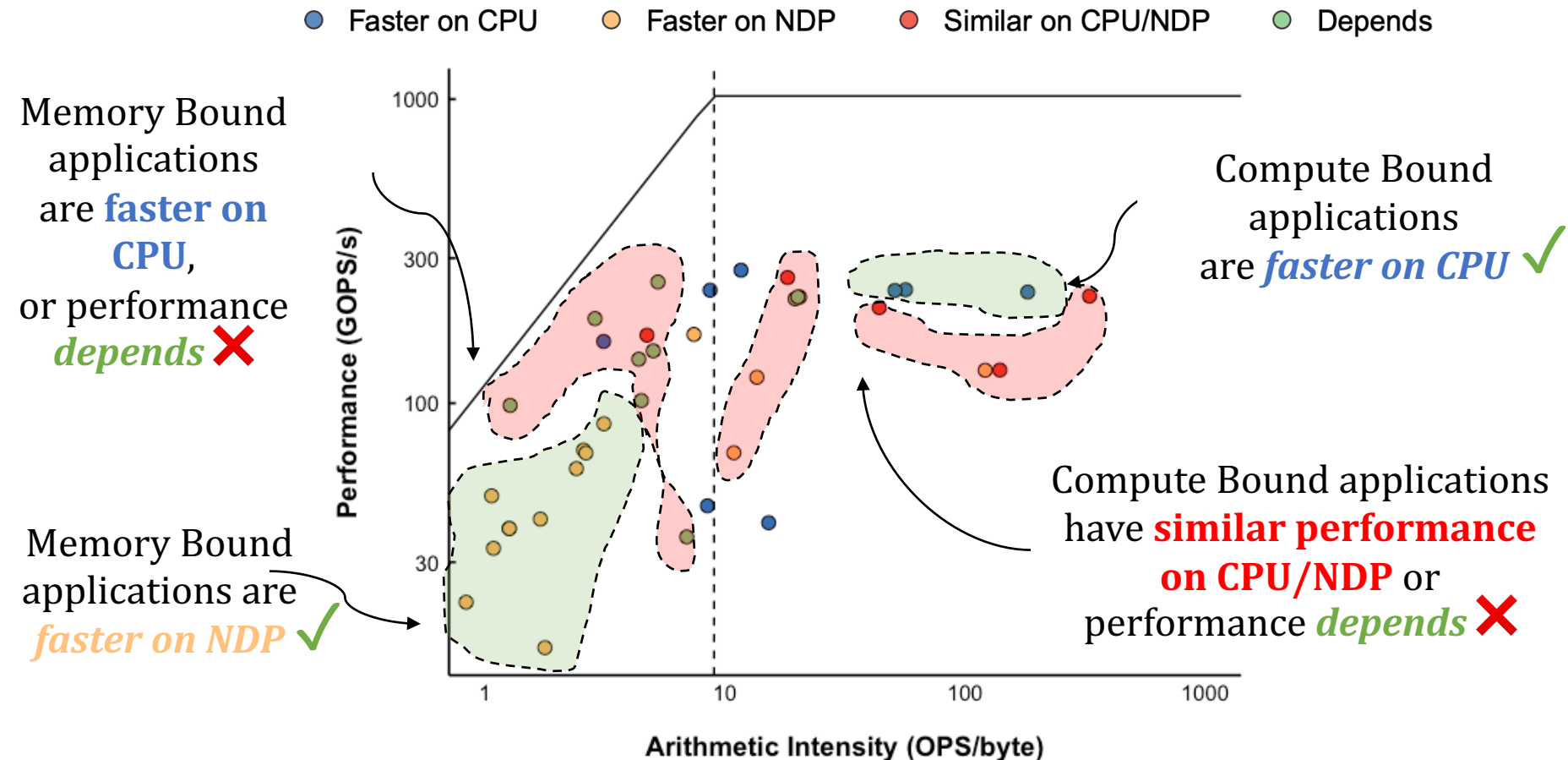
# Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units



# Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units



# Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units

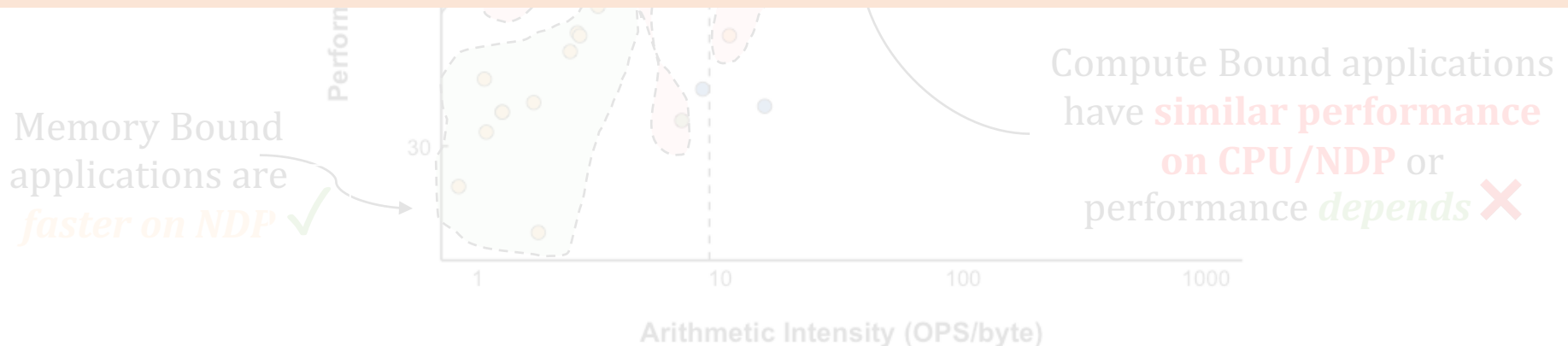
● Faster on CPU

● Faster on NDP

● Similar on CPU/NDP

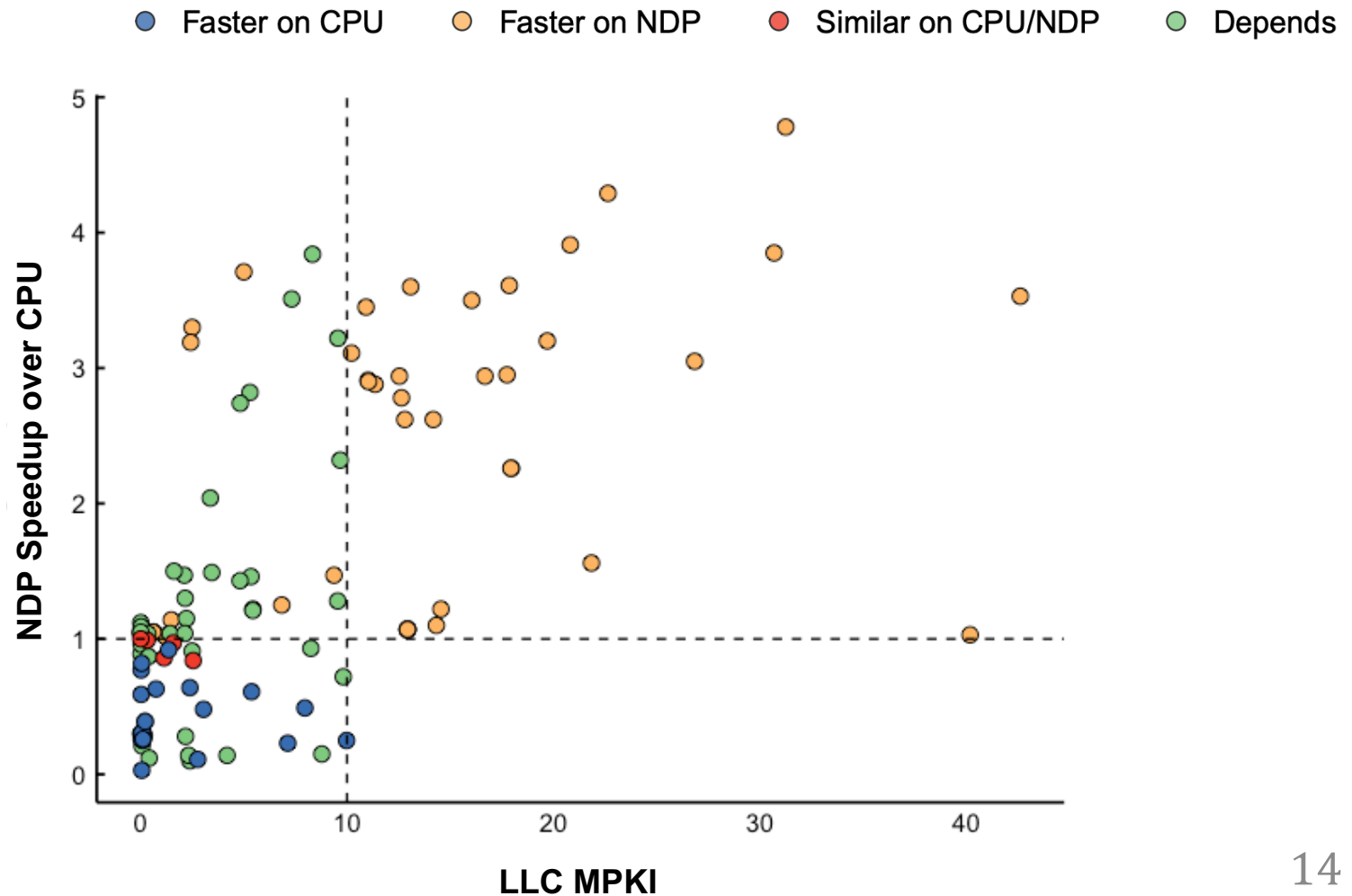
● Depends

**Roofline model does not accurately account for the NDP suitability of memory-bound applications**



# Limitations of Prior Approaches (2/2)

- Application with a last-level cache **MPKI > 10**  
→ **memory intensive** and **benefits from NDP**



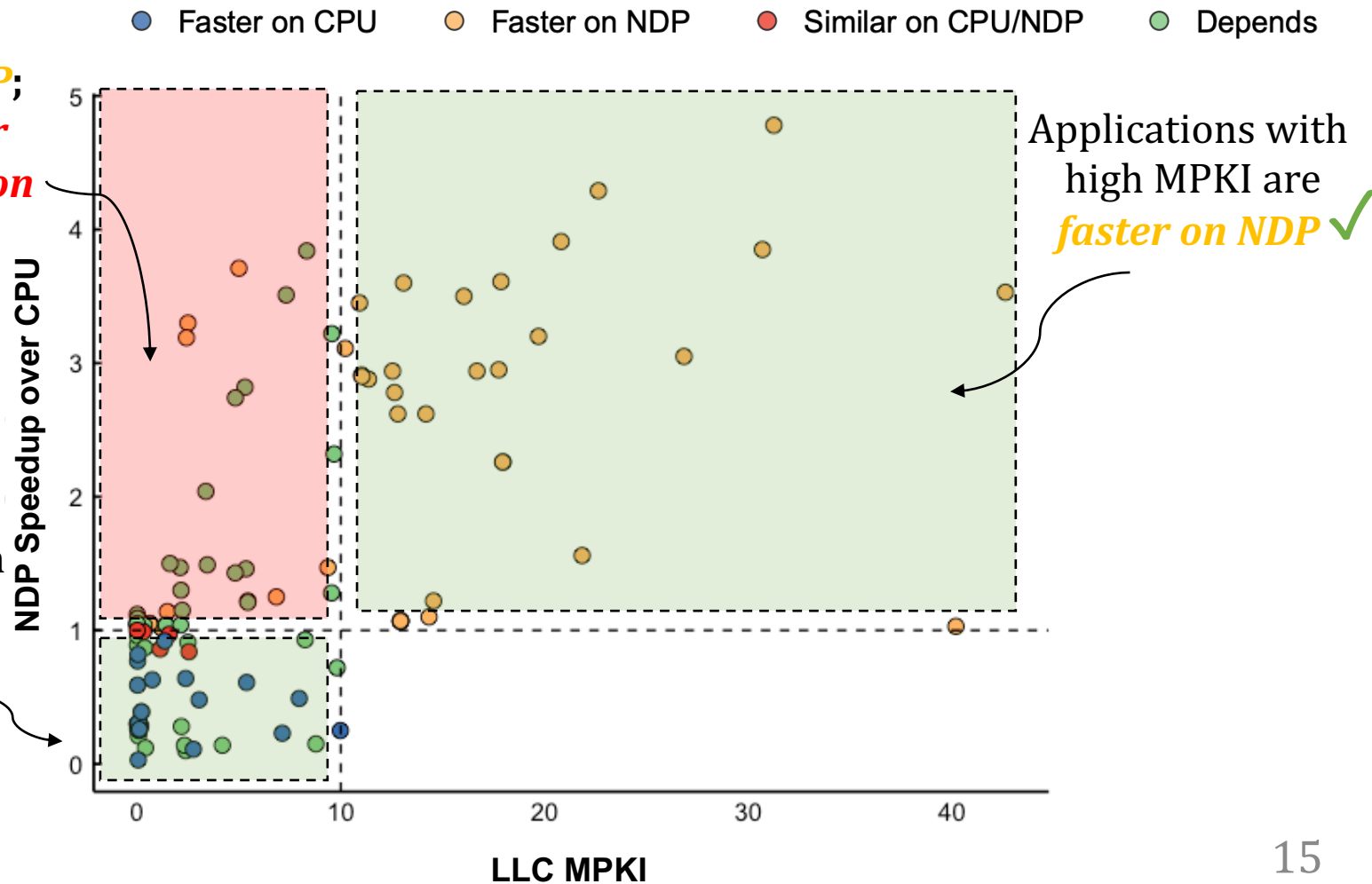
# Limitations of Prior Approaches (2/2)

- Application with a last-level cache **MPKI > 10**  
→ **memory intensive** and **benefits from NDP**

Applications with low  
MPKI can be

*faster on NDP*;  
have *similar*  
*performance on*  
*CPU/NDP* or;  
performance  
can *depends*  
✗

Applications with  
low MPKI are  
*faster on CPU*  
✓



# Limitations of Prior Approaches (2/2)

- Application with a last-level cache MPKI > 10  
→ **memory intensive** and **benefits from NDP**

Applications with low  
MPKI can be  
*faster on NDP*;

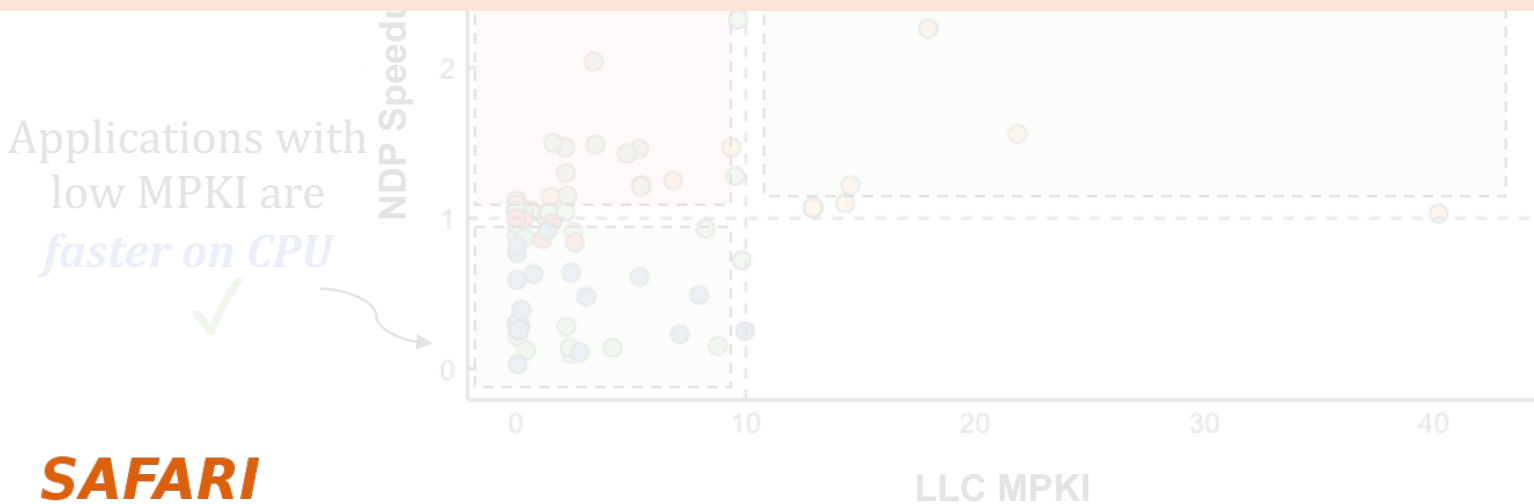
● Faster on CPU

● Faster on NDP

● Similar on CPU/NDP

● Depends

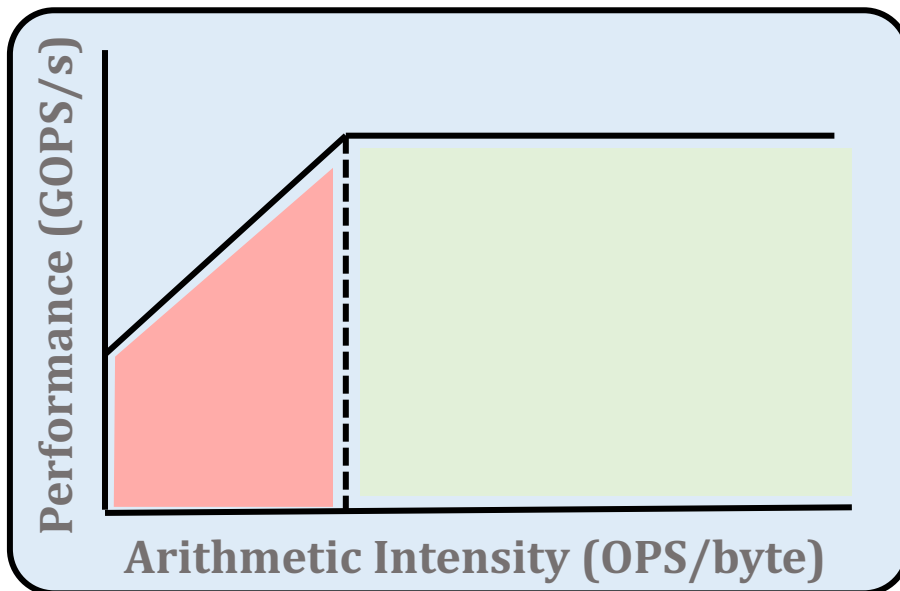
**LLC MPKI does not accurately account  
for the NDP suitability of memory-bound applications**



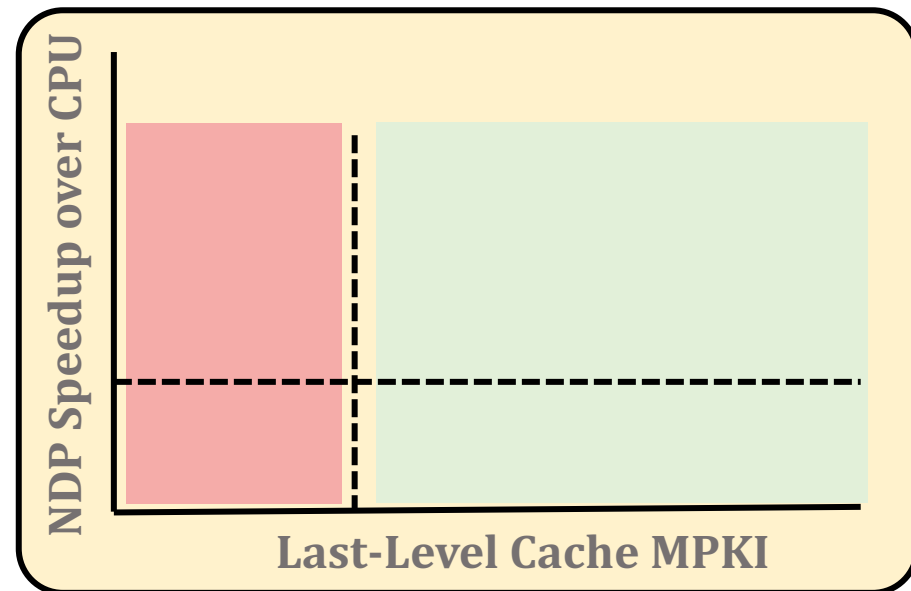
# Identifying Memory Bottlenecks

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP
- Existing approaches are not comprehensive enough

**Roofline model**



**High LLC MPKI**

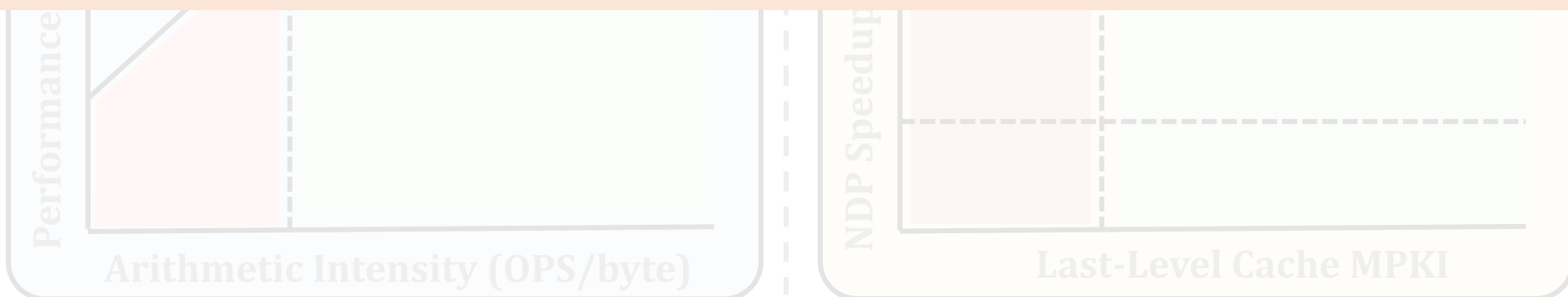


# The Problem

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

No available methodology can comprehensively:

- **identify** data movement bottlenecks
- **correlate** them with the **most suitable** data movement mitigation mechanism



# Our Goal

- **Our Goal:** develop a methodology to:
  - **methodically identify** sources of data movement bottlenecks
  - **comprehensively compare** compute- and memory-centric data movement mitigation techniques

# Outline

1. Data Movement Bottlenecks

**2. Methodology Overview**

3. Application Profiling

4. Locality-Based Clustering

5. Memory Bottleneck Analysis

6. Case Studies

# Key Approach

- New **workload characterization methodology** to analyze:
  - data movement bottlenecks
  - suitability of different data movement mitigation mechanisms
- Two main profiling strategies:

## **Architecture-independent profiling:**

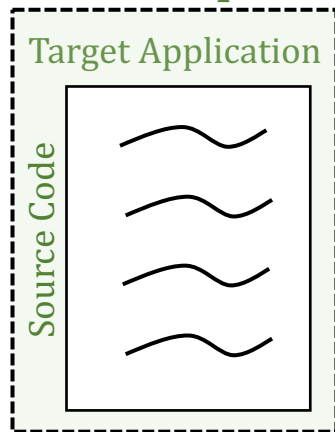
characterizes the memory behavior **independently**  
of the underlying **hardware**

## **Architecture-dependent profiling:**

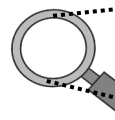
evaluates the **impact of the system configuration**  
on the memory behavior

# Methodology Overview

## User Input



## Step 1 Application Profiling



Profiler

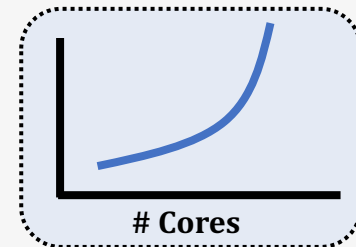
roi\_begin

roi\_end

## DAMOV-SIM Simulator

```
ld 0xFF
st 0xAF
ld 0xFF
st 0xAF
ld 0xFF
```

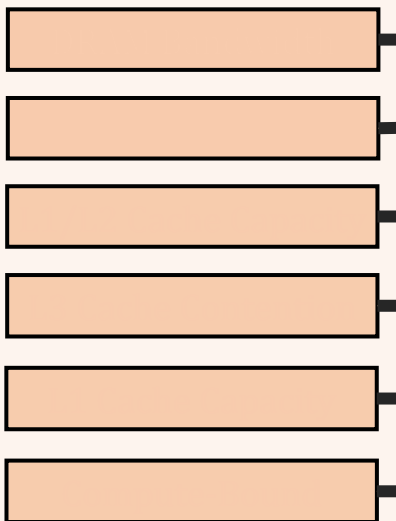
Memory Traces



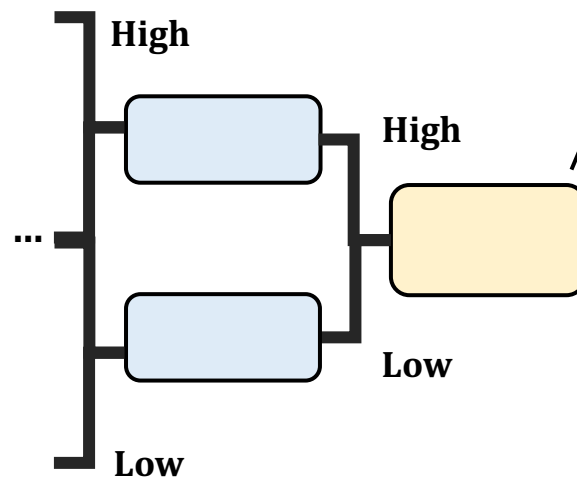
Scalability Analysis

## Methodology Output

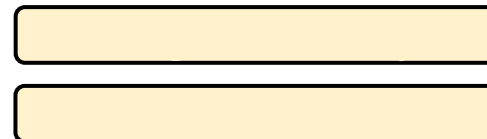
Memory Bottleneck Classes



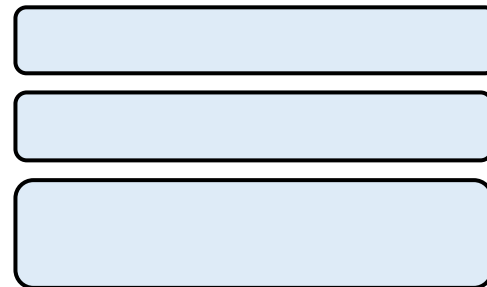
**SAFARI**



## Step 2 Locality-based Clustering

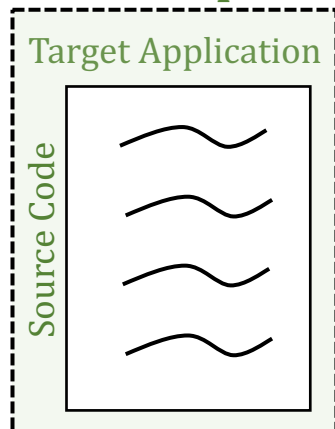


## Step 3 Memory Bottleneck Class.

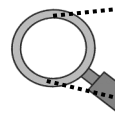


# Methodology Overview

## User Input



## Step 1 Application Profiling



Profiler

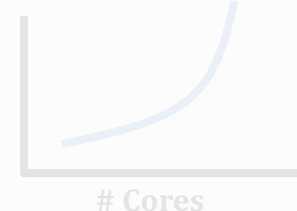
roi\_begin

roi\_end

## DAMOV-SIM Simulator

```
ld 0xFF
st 0xAF
ld 0xFF
st 0xAF
ld 0xFF
```

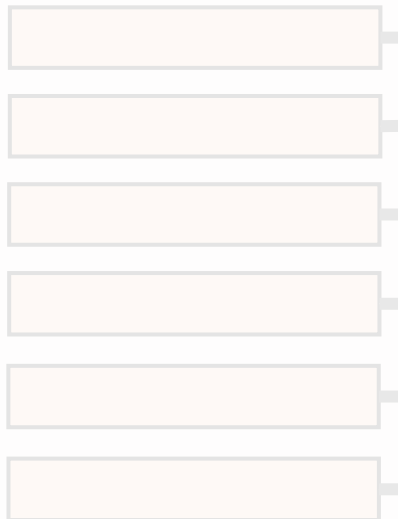
Memory Traces



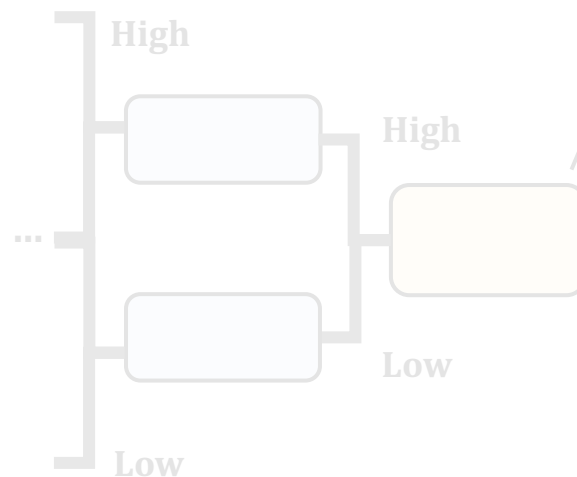
Scalability Analysis

## Methodology Output

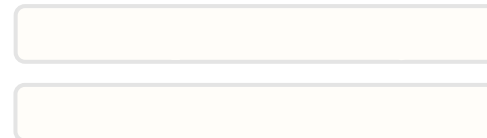
Memory Bottleneck Classes



**SAFARI**



## Step 2 Locality-based Clustering

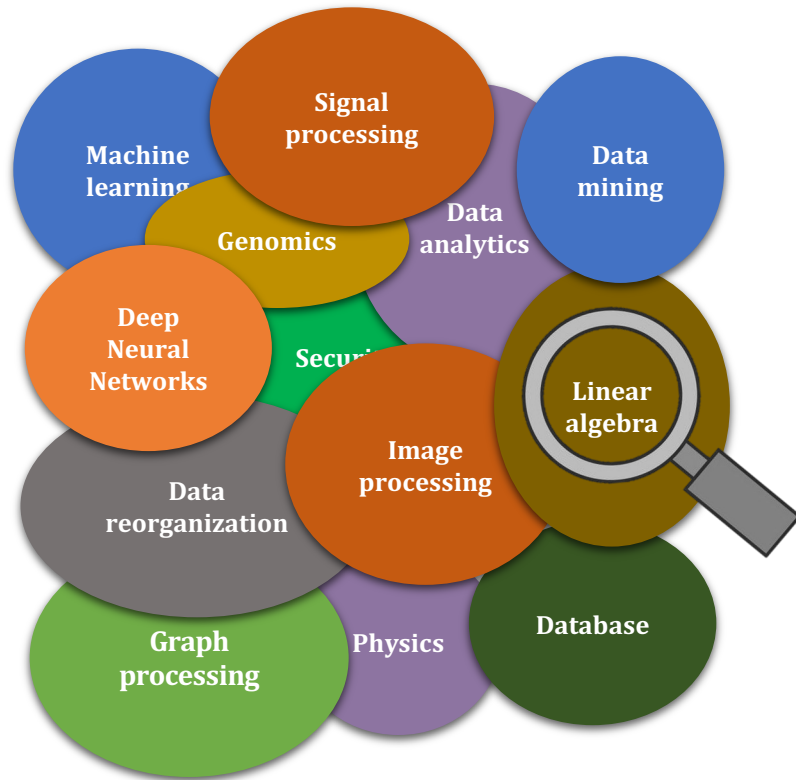


## Step 3 Memory Bottleneck Class.

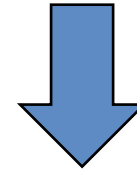


# Step 1: Application Profiling

Goal: Identify **application functions** that suffer from **data movement bottlenecks**

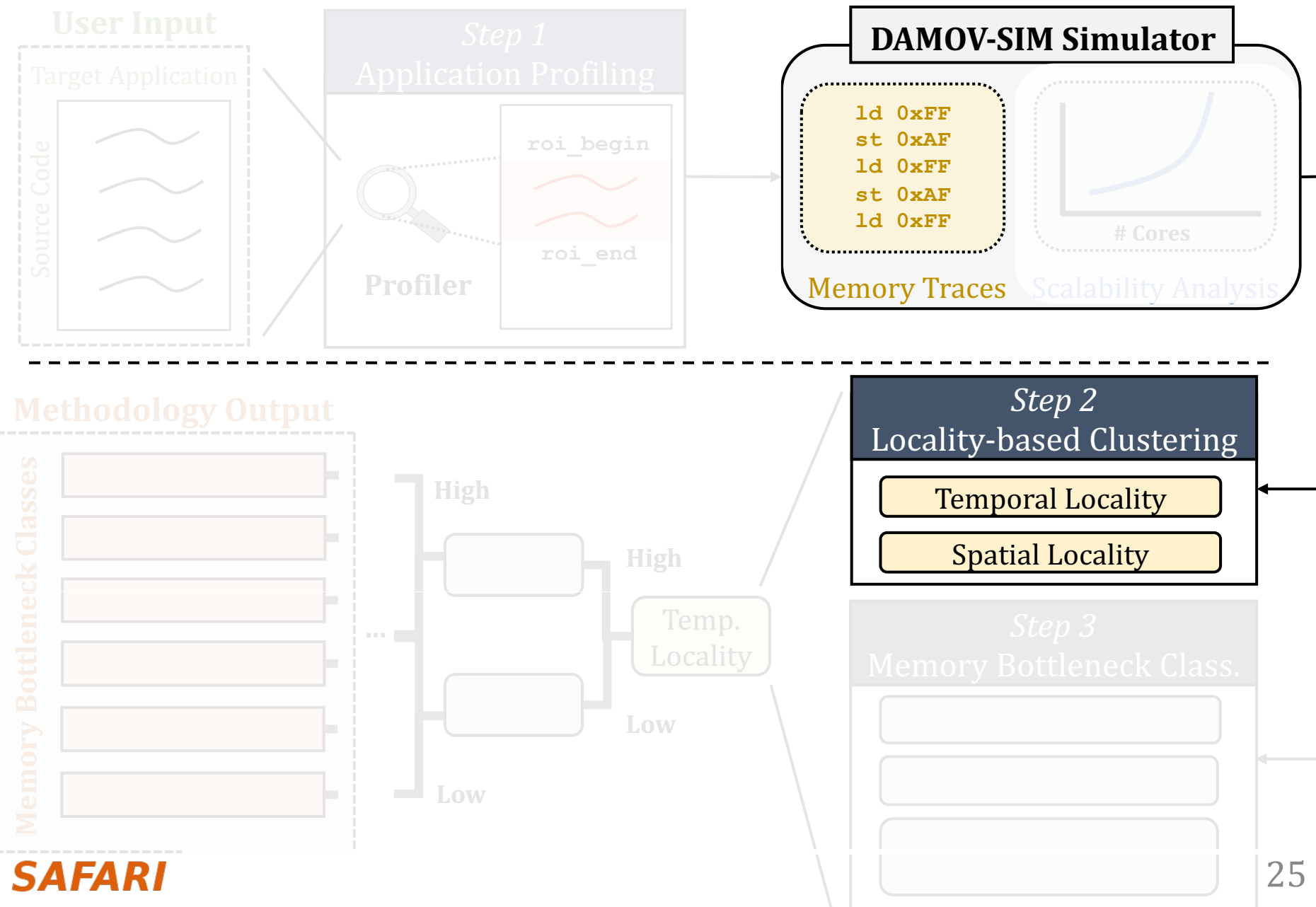


Hardware Profiling Tool:  
Intel VTune



**MemoryBound:**  
CPU is stalled due to load/store

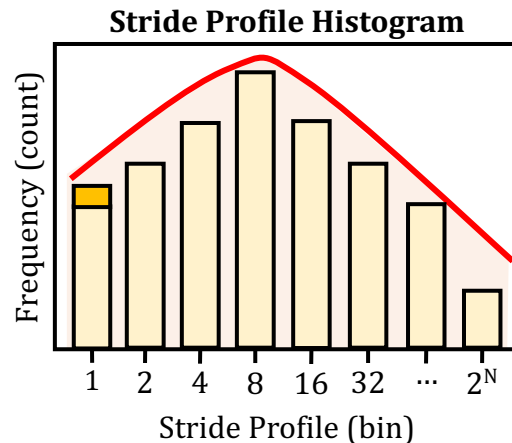
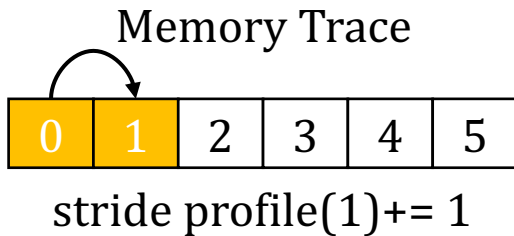
# Methodology Overview



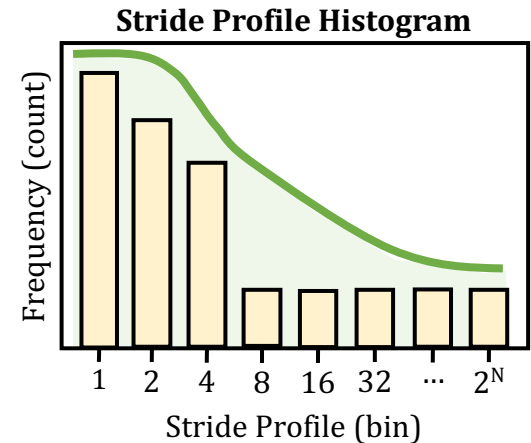
# Step 2: Locality-Based Clustering

- **Goal:** analyze application's memory characteristics

## Spatial Locality<sup>7</sup>



**Low spatial locality**

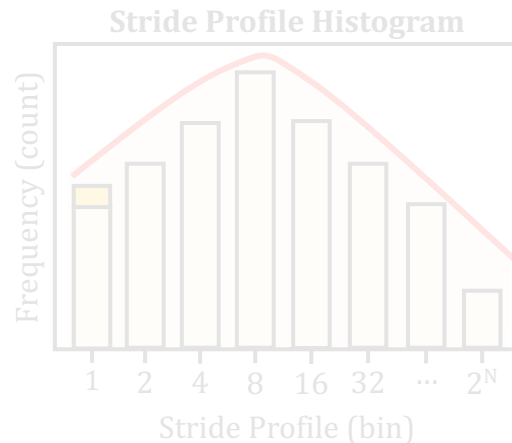
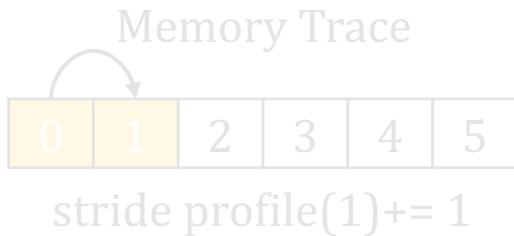


**High spatial locality**

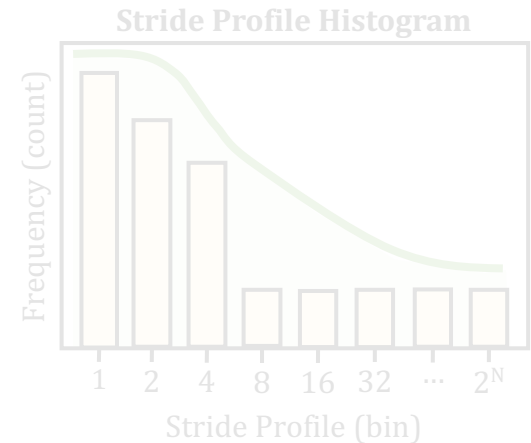
# Step 2: Locality-Based Clustering

- **Goal:** analyze application's memory characteristics

## Spatial Locality<sup>7</sup>

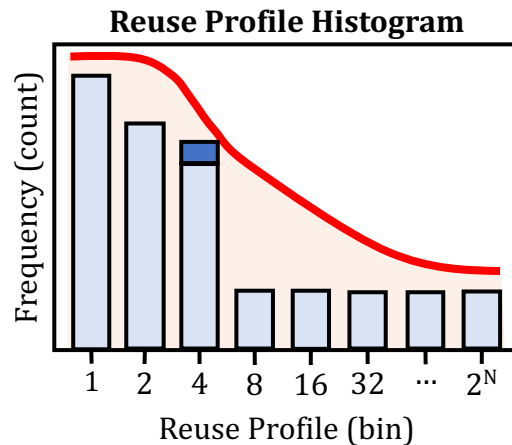
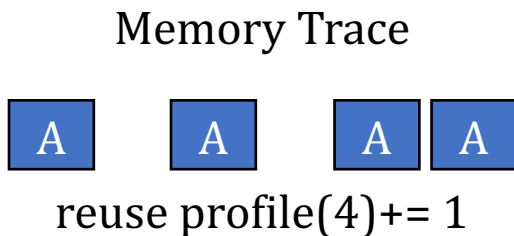


Low spatial locality

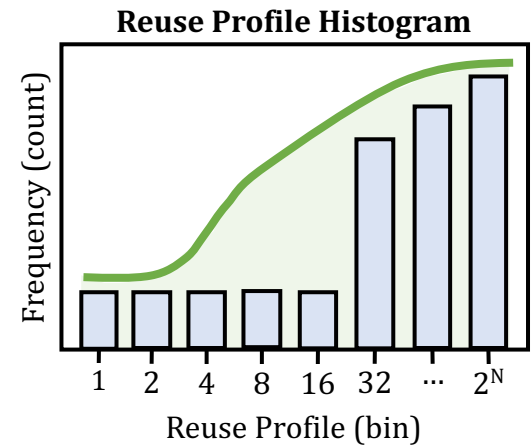


High spatial locality

## Temporal Locality<sup>7</sup>

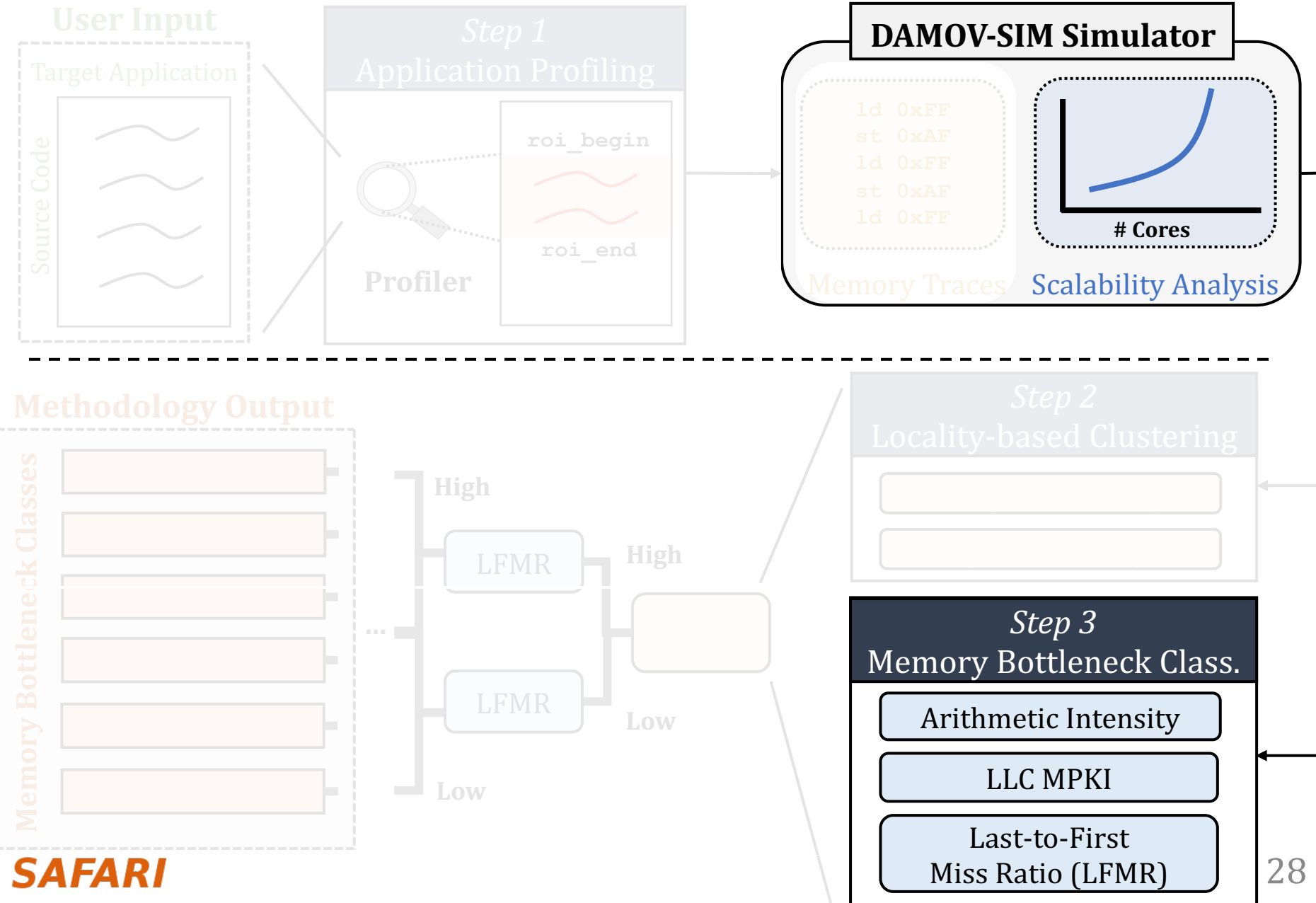


Low temporal locality



High temporal locality

# Methodology Overview



# Step 3: Memory Bottleneck Classification (1/2)

## Arithmetic Intensity (AI)

- floating-point/arithmetic operations per L1 cache lines accessed  
→ shows **computational intensity** per memory request

## LLC Misses-per-Kilo-Instructions (MPKI)

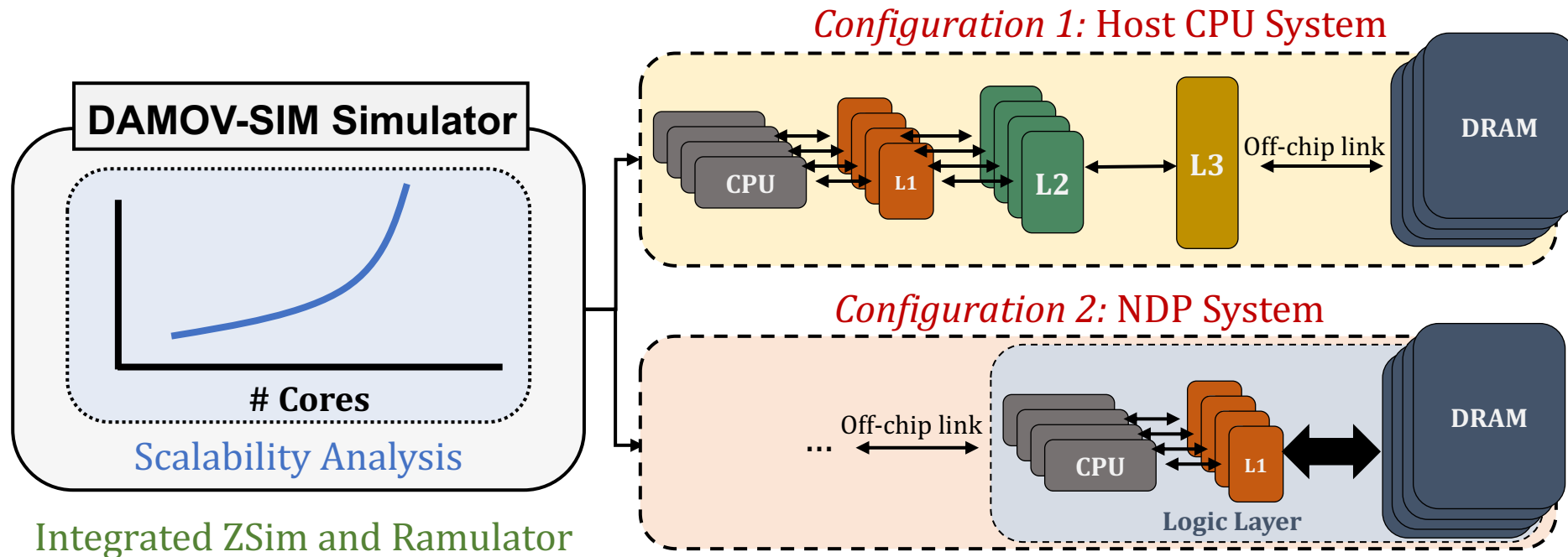
- LLC misses per one thousand instructions  
→ shows **memory intensity**

## Last-to-First Miss Ratio (LFMR)

- LLC misses per L1 misses  
→ shows if an application **benefits from L2/L3 caches**

# Step 3: Memory Bottleneck Classification (2/2)

- **Goal:** identify the specific sources of data movement bottlenecks



- **Scalability Analysis:**
  - 1, 4, 16, 64, and 256 out-of-order/in-order host and NDP CPU cores
  - 3D-stacked memory as main memory

# Outline

1. Data Movement Bottlenecks

2. Methodology Overview

**3. Application Profiling**

4. Locality-Based Clustering

5. Memory Bottleneck Analysis

6. Case Studies

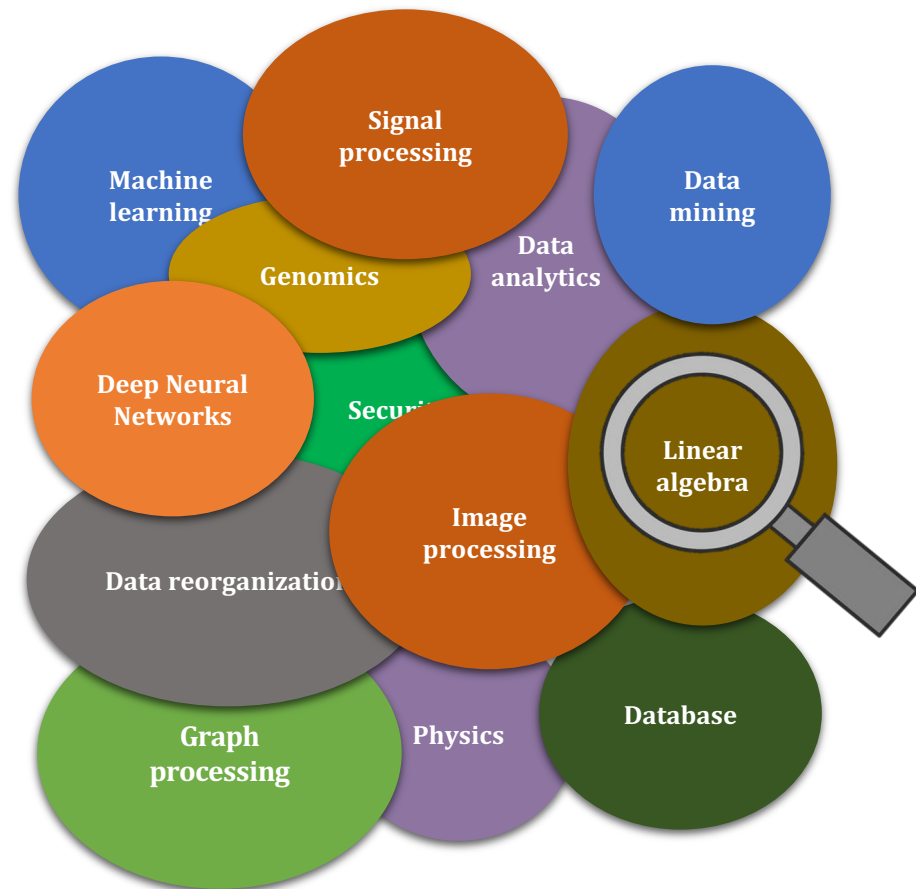
# Step 1: Application Profiling

- We analyze 345 applications from distinct domains:

- Graph Processing
- Deep Neural Networks
- Physics
- High-Performance Computing
- Genomics
- Machine Learning
- Databases
- Data Reorganization
- Image Processing
- Map-Reduce
- Benchmarking
- Linear Algebra

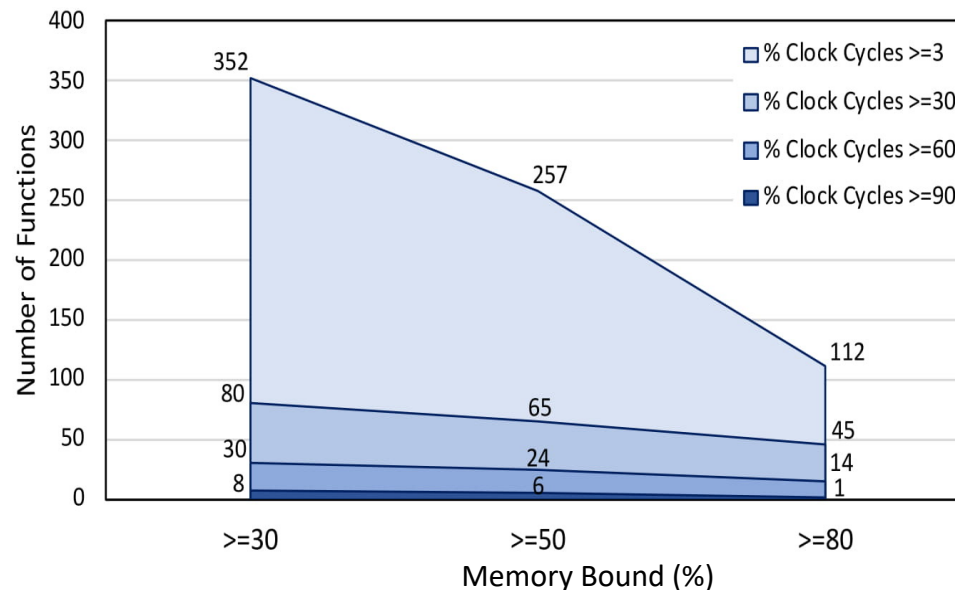
...

**SAFARI**



# Memory Bound Functions

- We analyze 345 applications from distinct domains
- **Selection criteria:** clock cycles > 3% and Memory Bound > 30%



- We find 144 functions from a total of 77K functions and select:
  - 44 functions → apply steps 2 and 3
  - 100 functions → validation

# Outline

1. Data Movement Bottlenecks

2. Methodology Overview

3. Application Profiling

**4. Locality-Based Clustering**

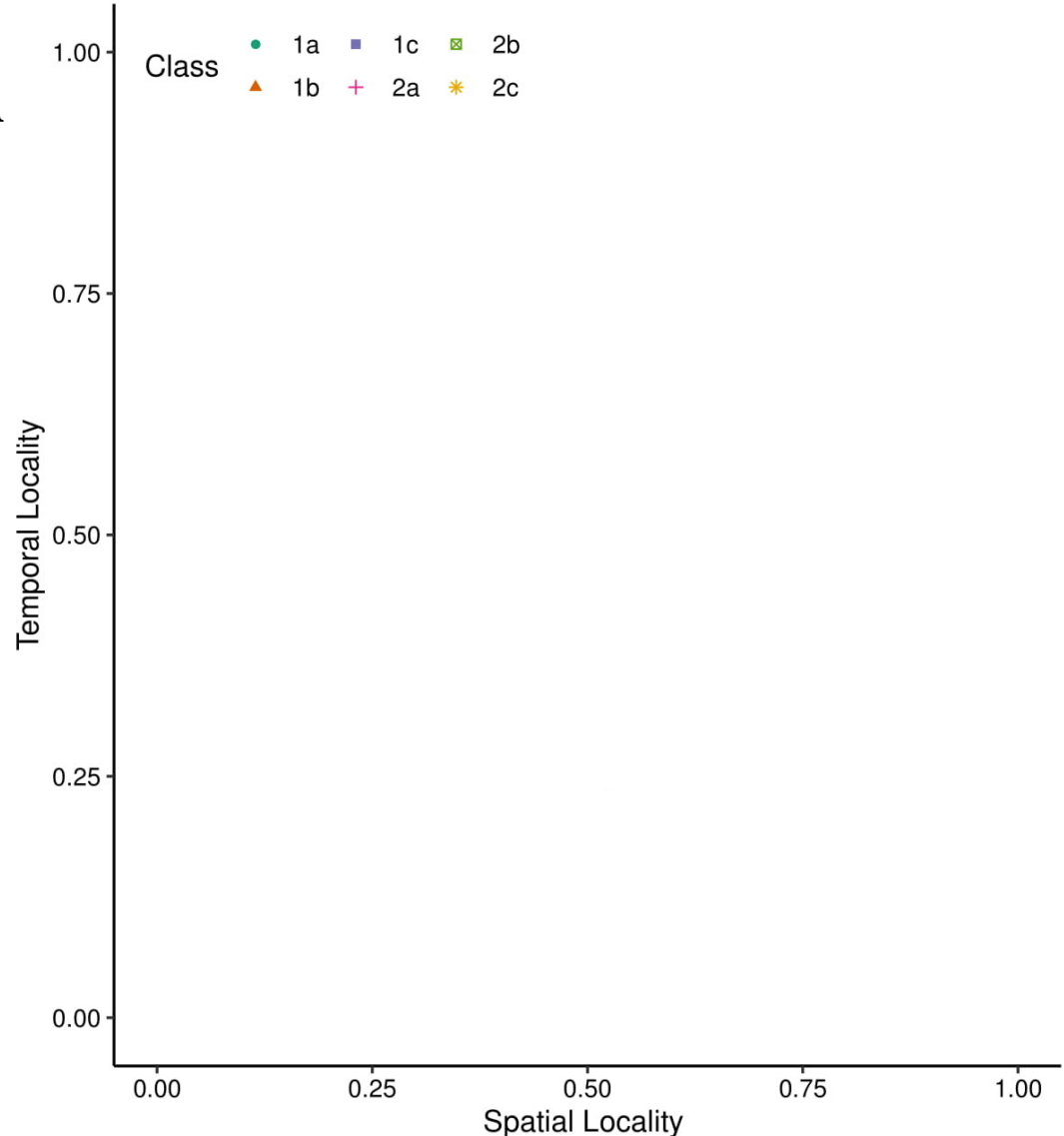
5. Memory Bottleneck Analysis

6. Case Studies

# Step 2: Locality-Based Clustering

We use K-means to cluster the applications across both **spatial and temporal locality**, forming two groups

1. Low locality applications (in orange)
2. High locality applications (in blue)



# Step 2: Locality-Based Clustering

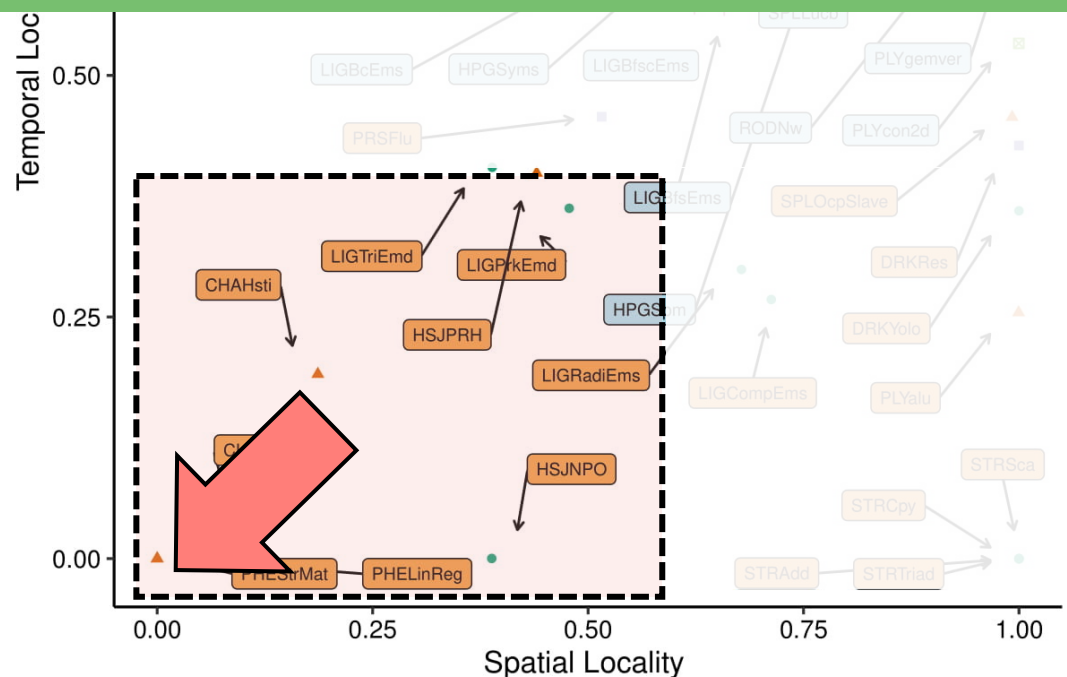
We use K-means to cluster the applications across both



The closer a function is to the **bottom-left corner**

→ less likely it is to **take advantage** of a deep cache hierarchy

2. High locality applications (in blue)



# Outline

1. Data Movement Bottlenecks

2. Methodology Overview

3. Application Profiling

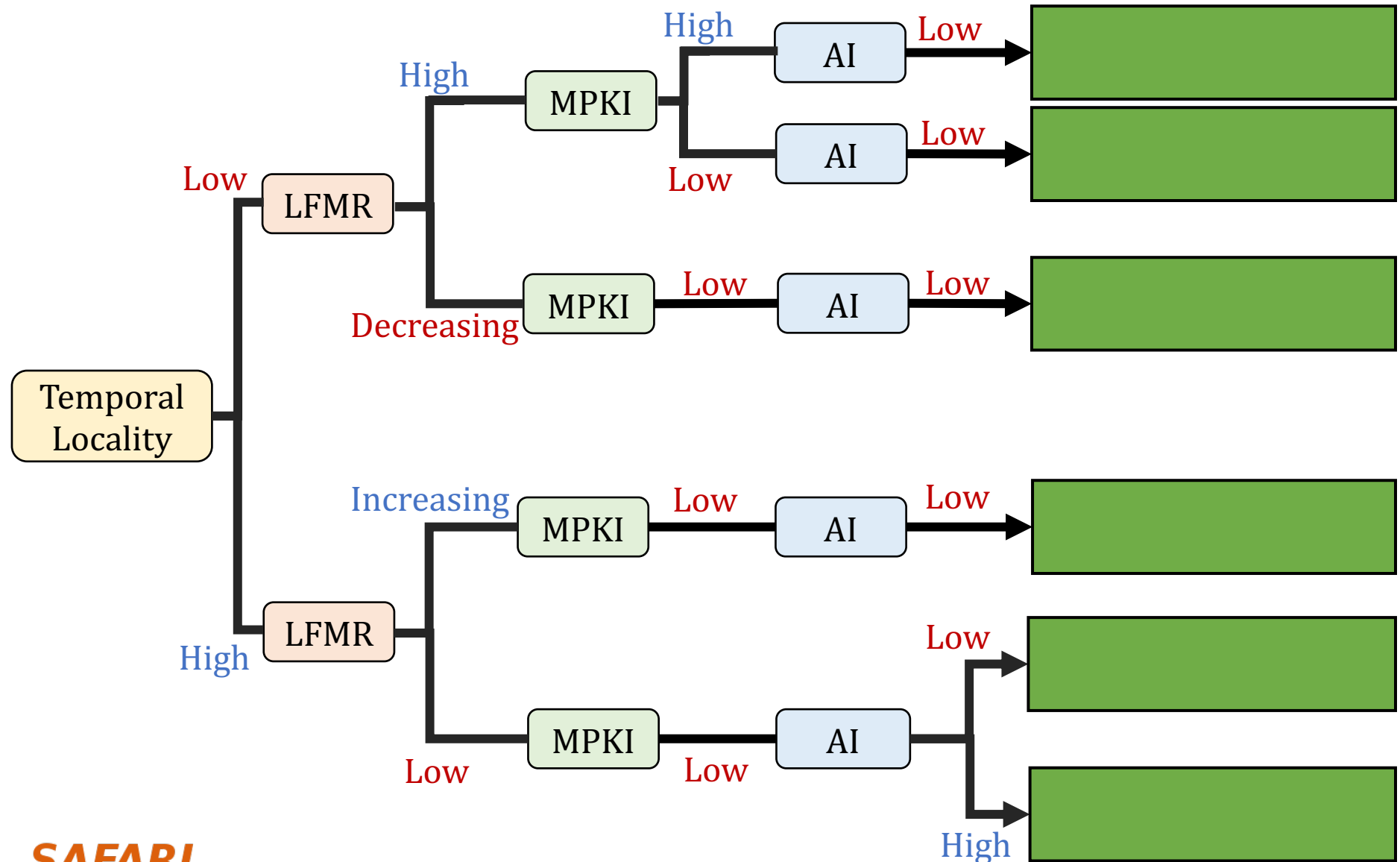
4. Locality-Based Clustering

**5. Memory Bottleneck Analysis**

6. Case Studies

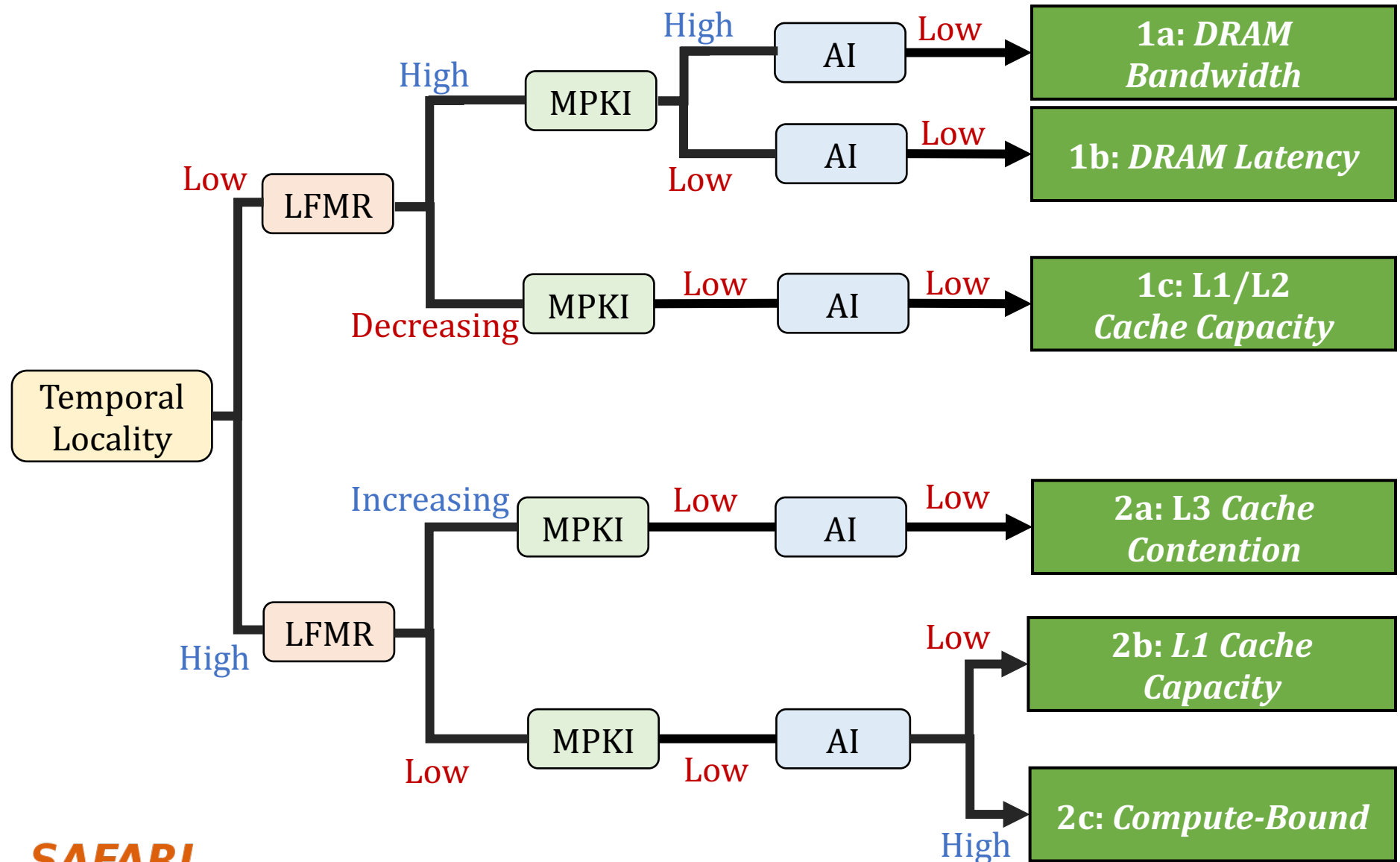
# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Step 3: Memory Bottleneck Analysis

**Six classes of  
data movement bottlenecks:**

each class  $\leftrightarrow$  data movement  
mitigation mechanism

## Memory Bottleneck Class

1a: *DRAM  
Bandwidth*

1b: *DRAM Latency*

1c: *L1/L2  
Cache Capacity*

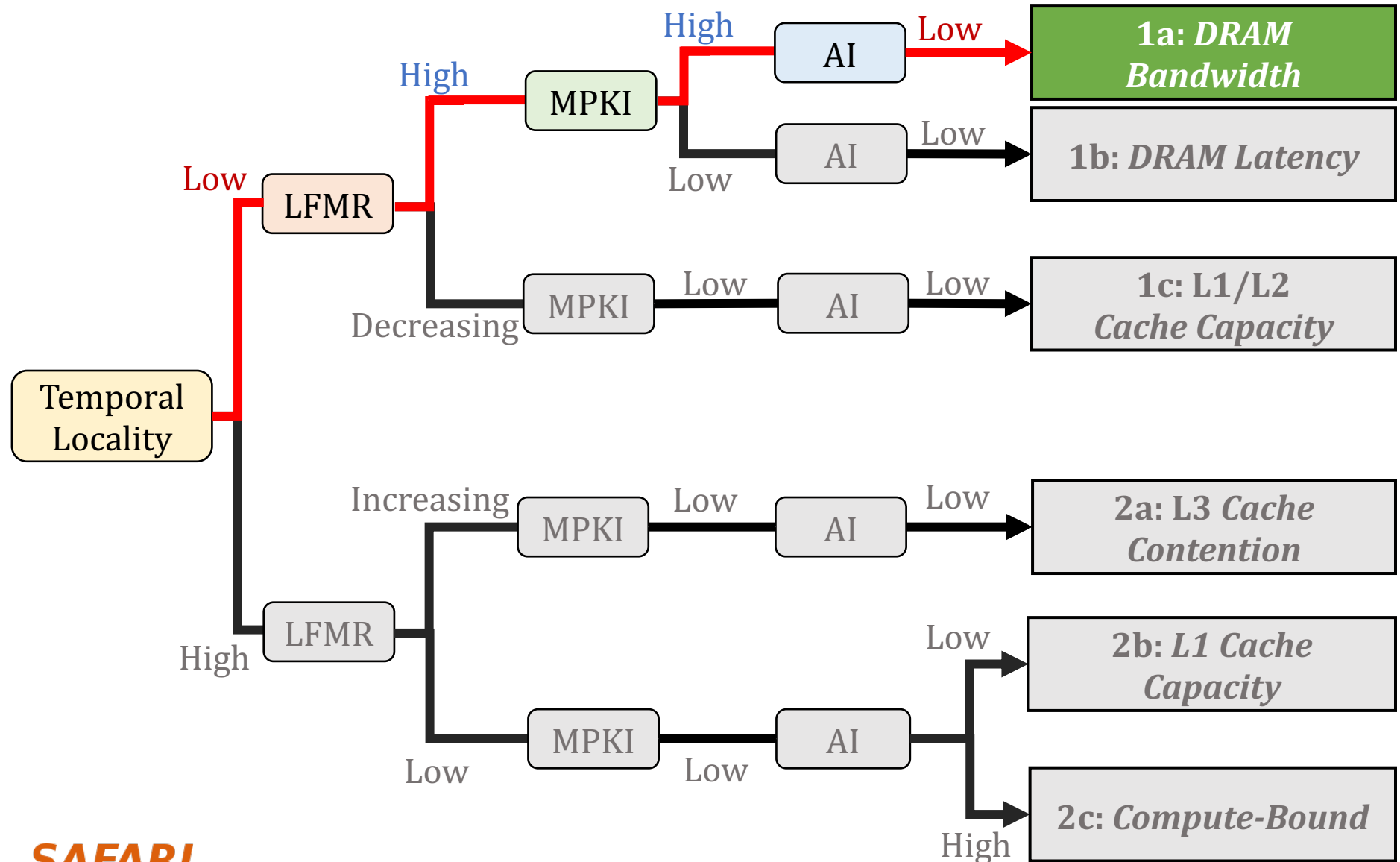
2a: *L3 Cache  
Contention*

2b: *L1 Cache  
Capacity*

2c: *Compute-Bound*

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 1a: DRAM Bandwidth Bound (1/2)

- High MPKI → **high memory pressure**
- Host scales well until **bandwidth saturates**
- NDP scales **without saturating** alongside attained bandwidth

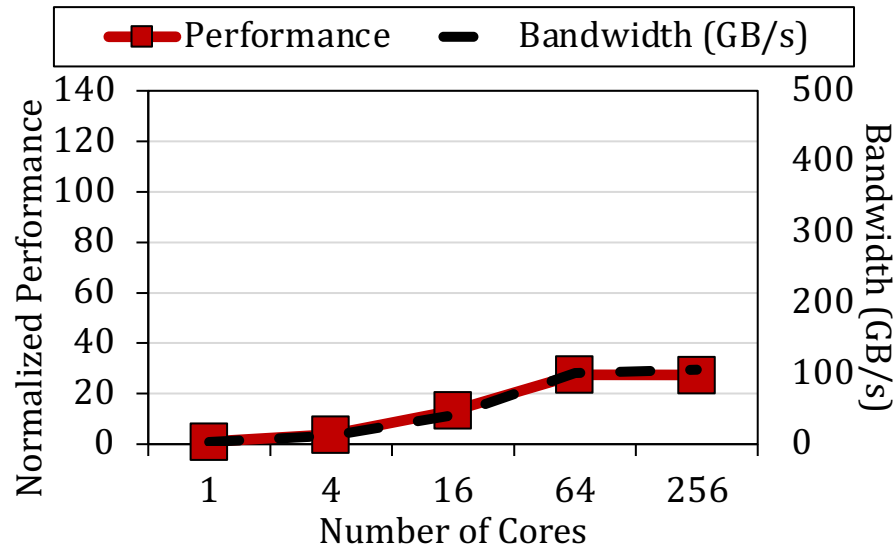
Temp. Loc: *low*

LFMR: *high*

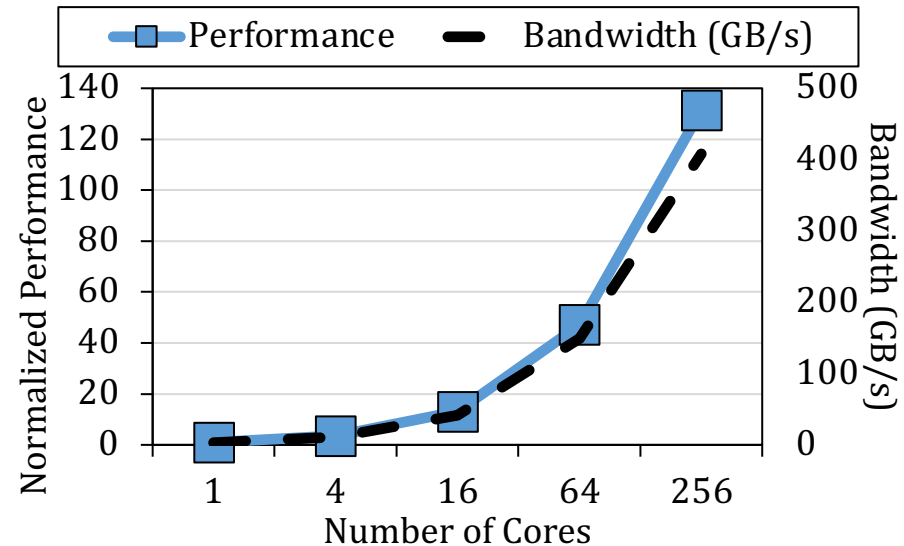
MPKI: *high*

AI: *low*

## Host



## NDP



## DRAM bandwidth bound applications:

NDP does better because of the **higher internal DRAM bandwidth**

# Class 1a: DRAM Bandwidth Bound (2/2)

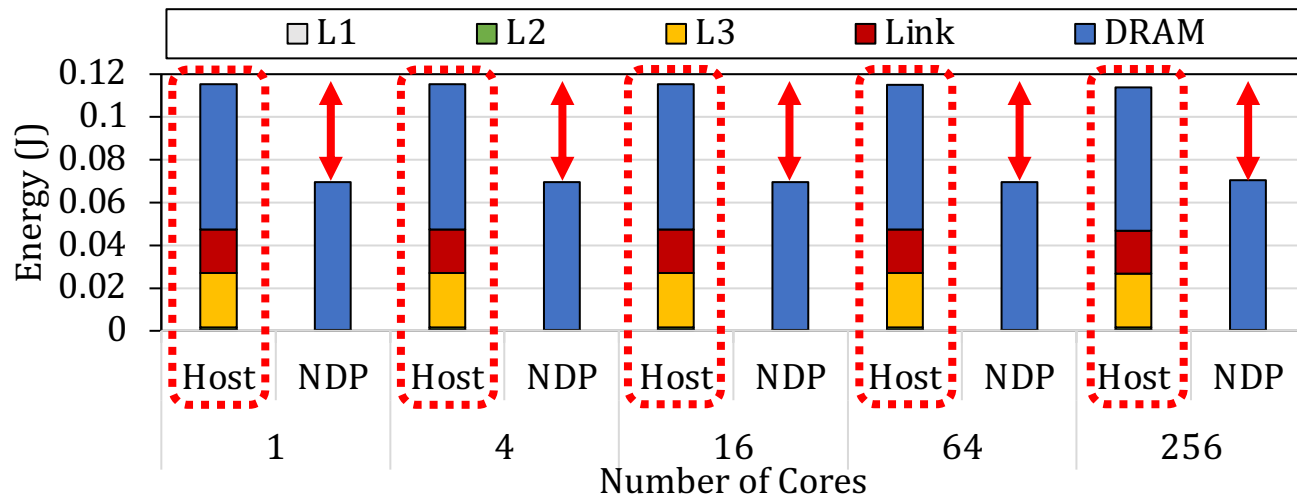
- High LFMR → **L2 and L3 caches are inefficient**
- Host's energy consumption is dominated by **cache look-ups and off-chip data transfers**
- NDP provides **large system energy reduction** since it does not access L2, L3, and off-chip links

Temp. Loc: *low*

LFMR: *high*

MPKI: *high*

AI: *low*

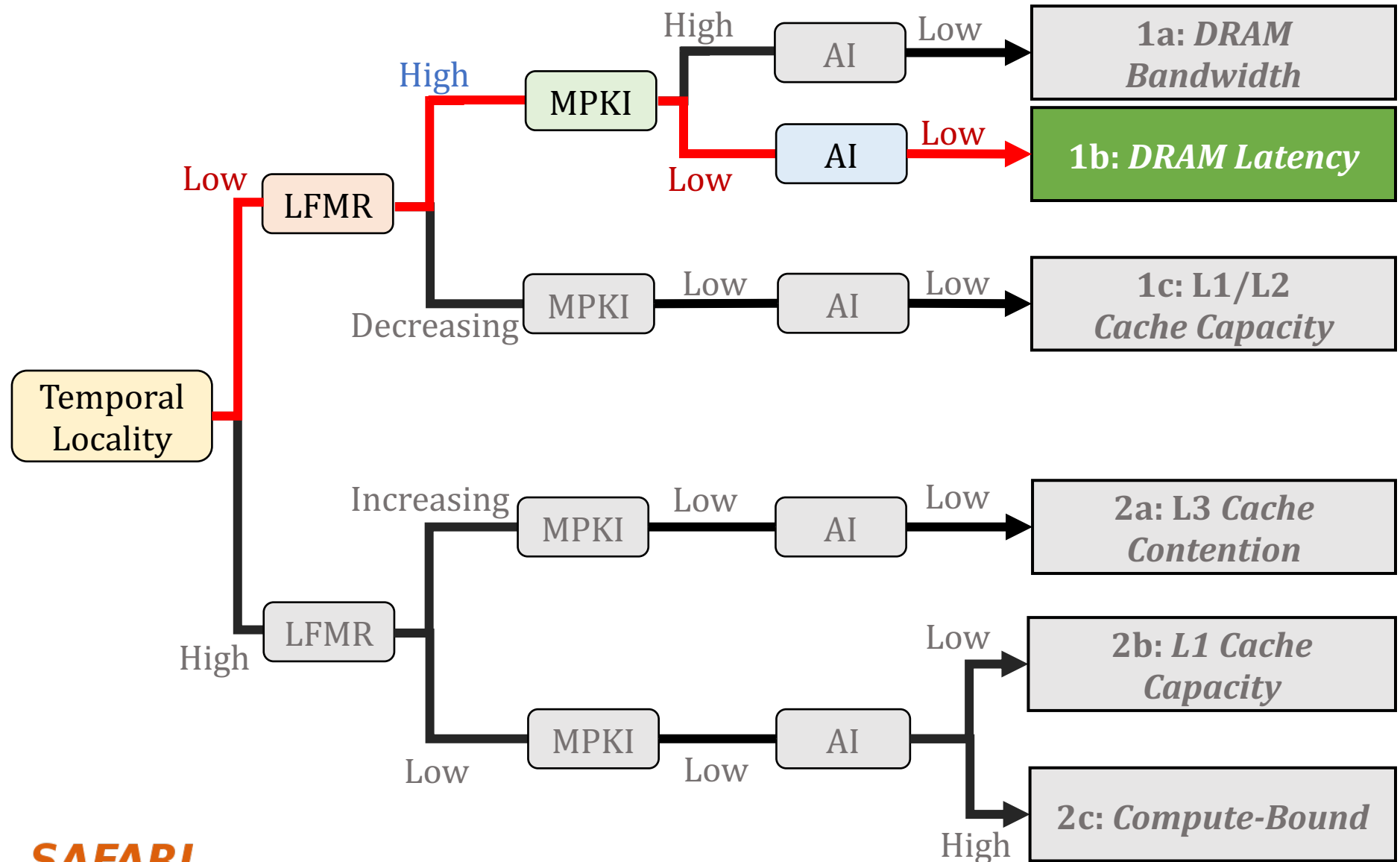


**DRAM bandwidth bound applications:**

NDP does better because it eliminates off-chip I/O traffic

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 1b: DRAM Latency Bound

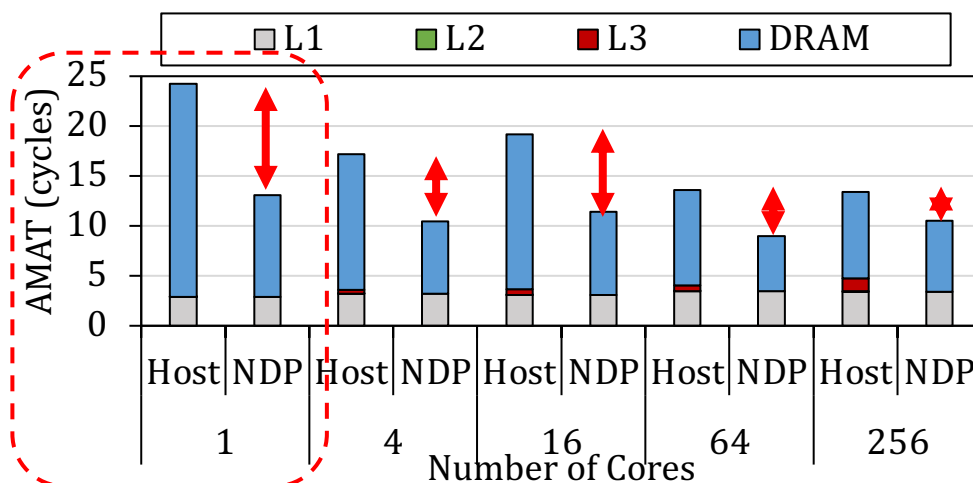
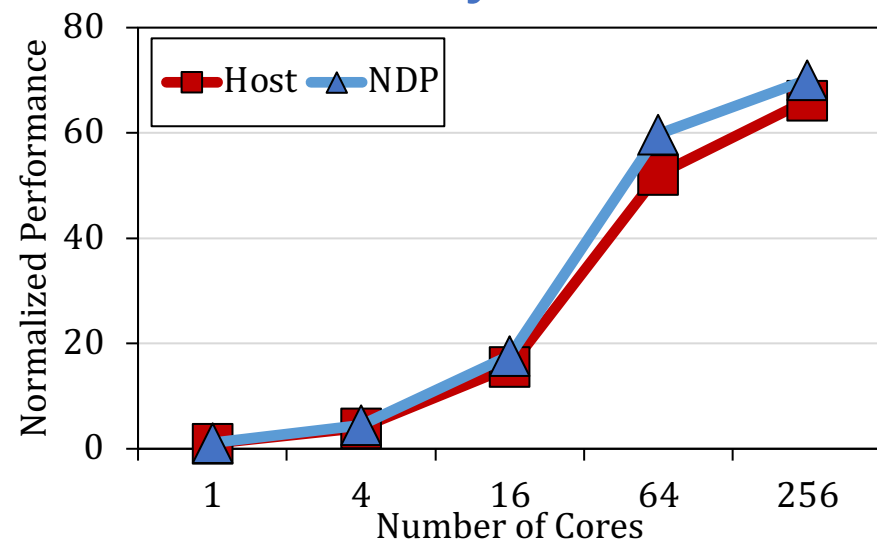
- High LFMR → L2 and L3 caches are inefficient
- Host scales well but NDP performance is always higher
- NDP performs better than host because of its **lower memory access latency**

Temp. Loc: *low*

LFMR: *high*

MPKI: *low*

AI: *low*

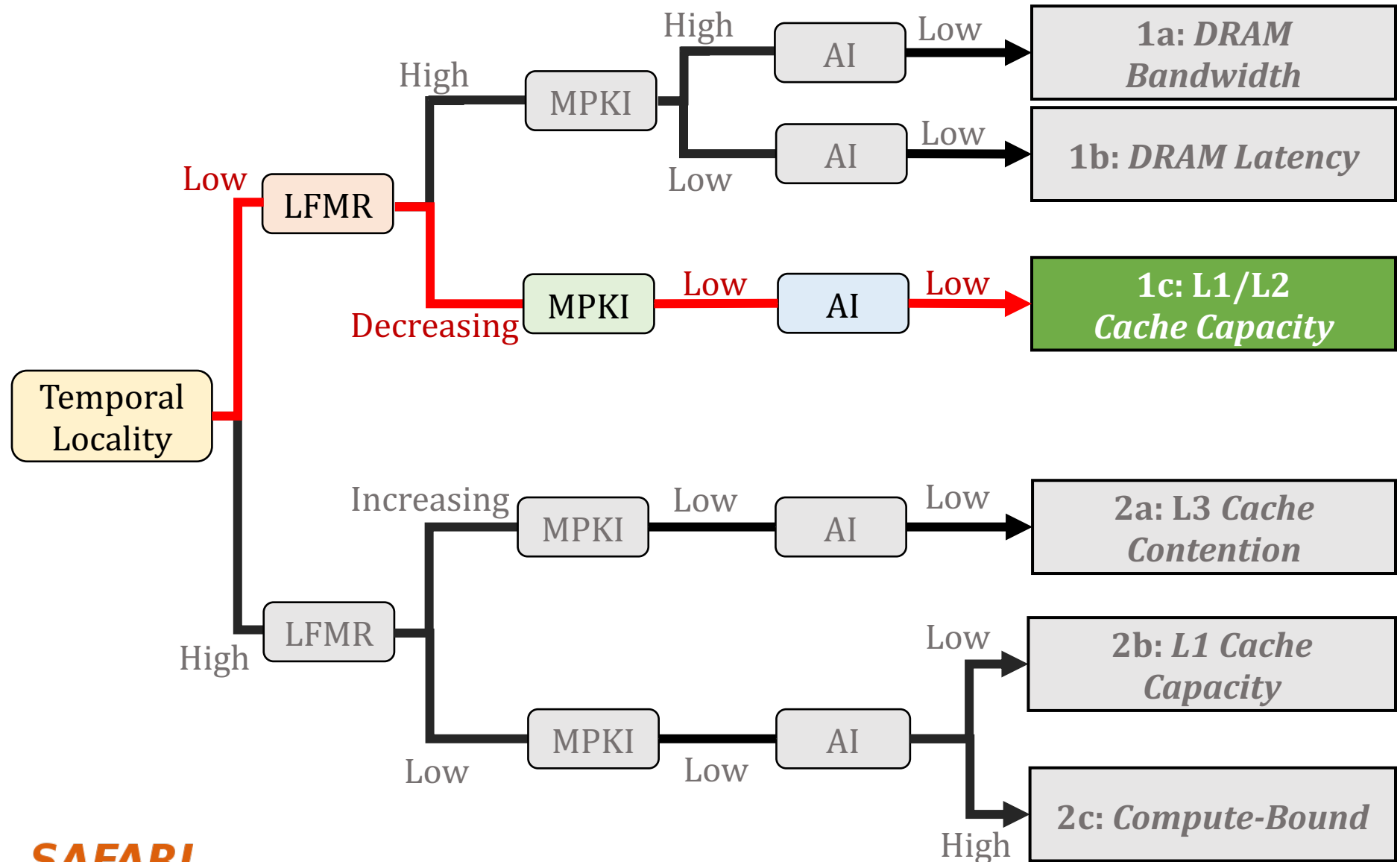


**DRAM latency bound applications:**

host performance is hurt by the **cache hierarchy** and **off-chip link**

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 1c: L1/L2 Cache Capacity

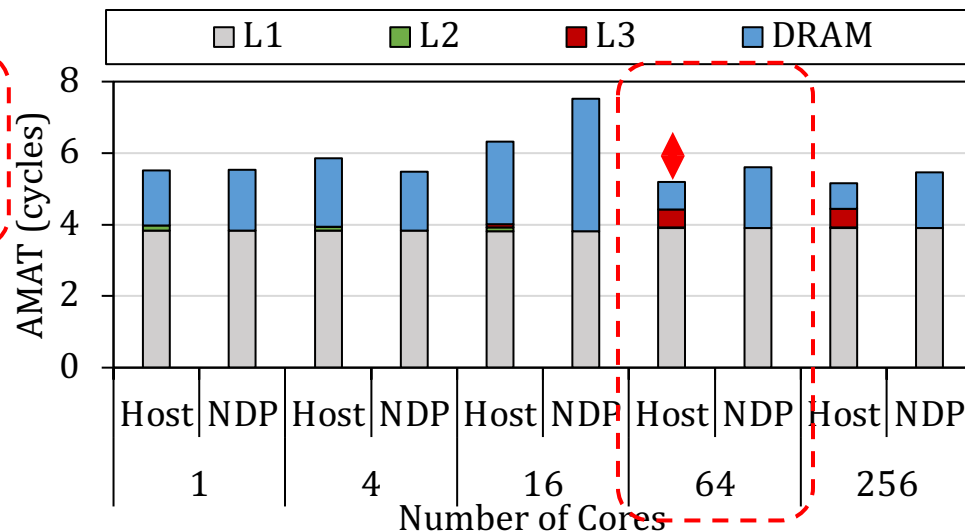
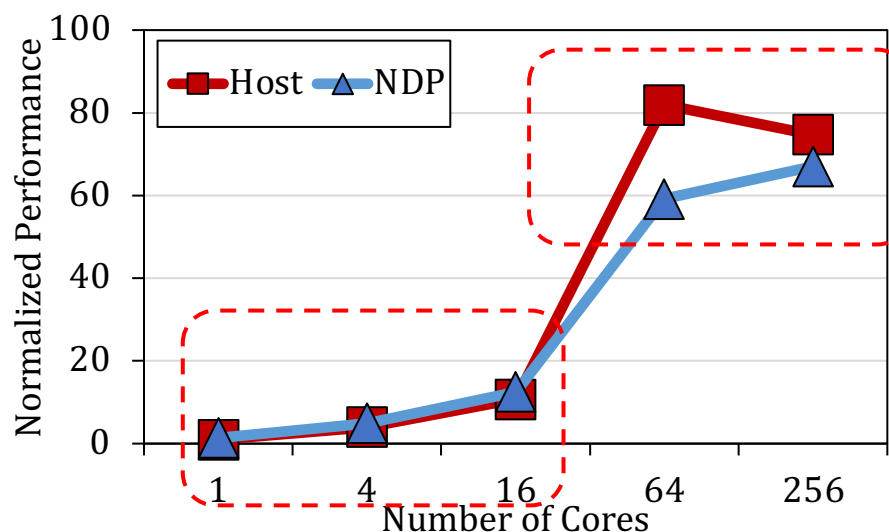
- Decreasing LFMR → L2/L3 caches turn efficient
- NDP scales better than the host at low core counts
- Host scales better than NDP at high core counts
- Host performs better than NDP at high core counts since it reduces memory access latency via data caching

Temp. Loc: *low*

LFMR: *decreasing*

MPKI: *low*

AI: *low*

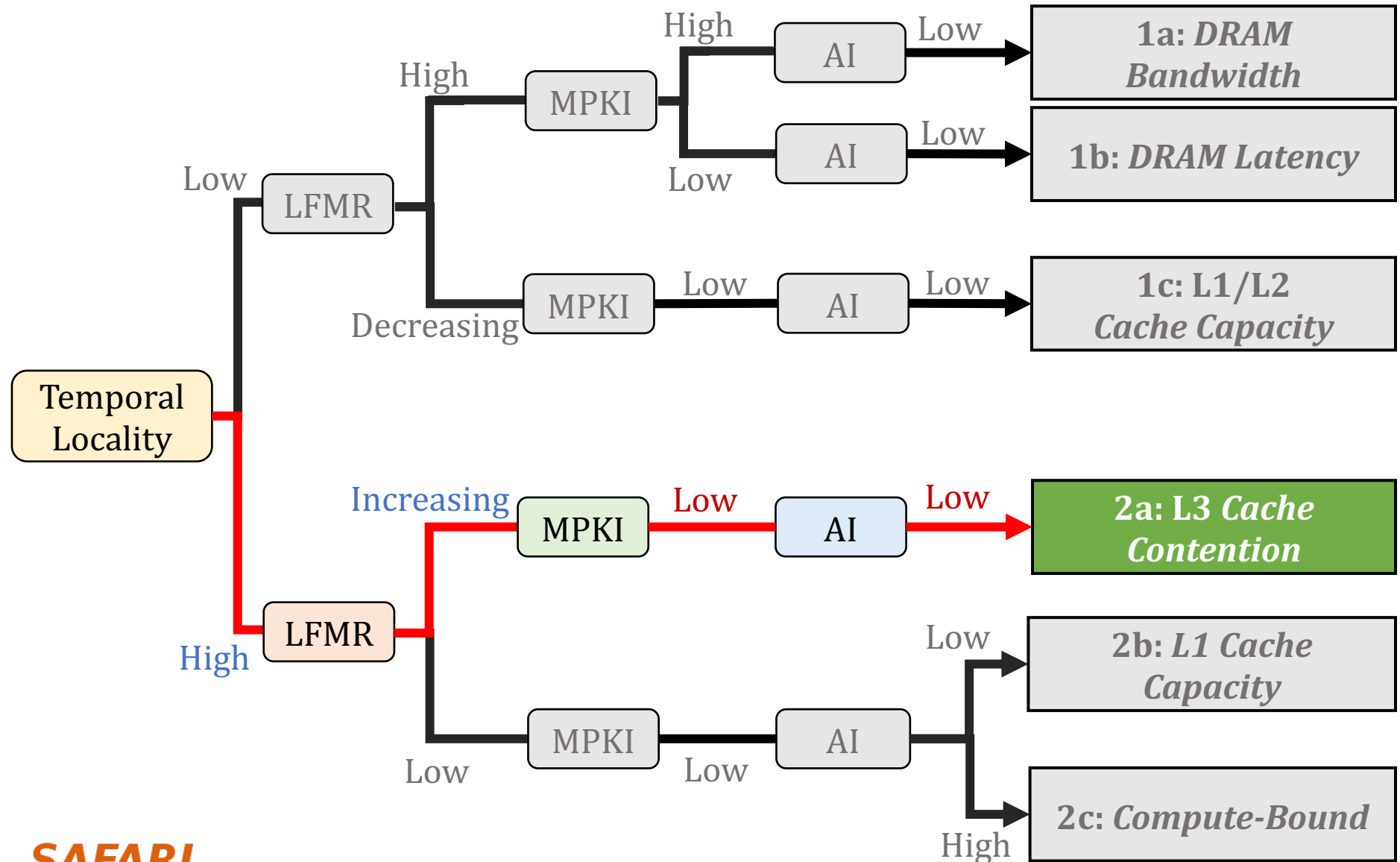


**L1/L2 cache capacity bottlenecked applications:**

NDP is higher performance when the aggregated cache size is small

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 2a: L3 Cache Contention

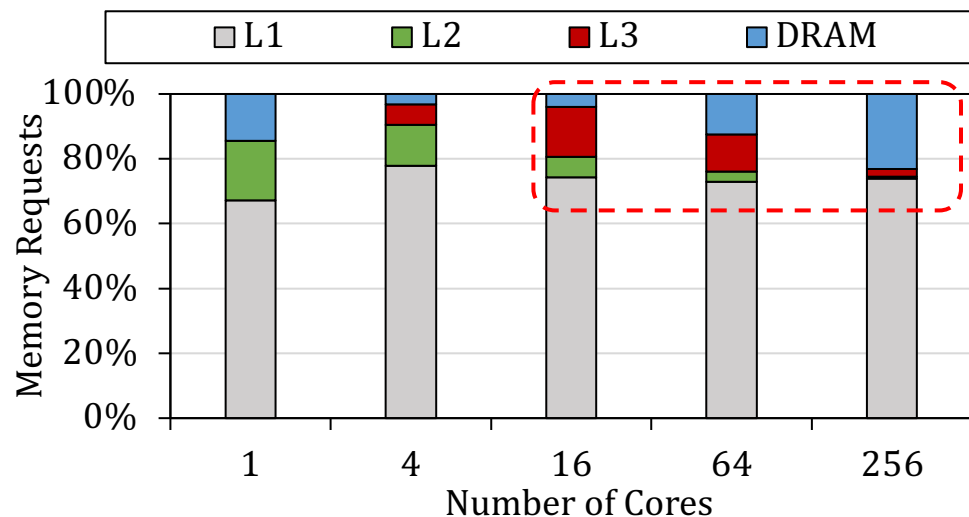
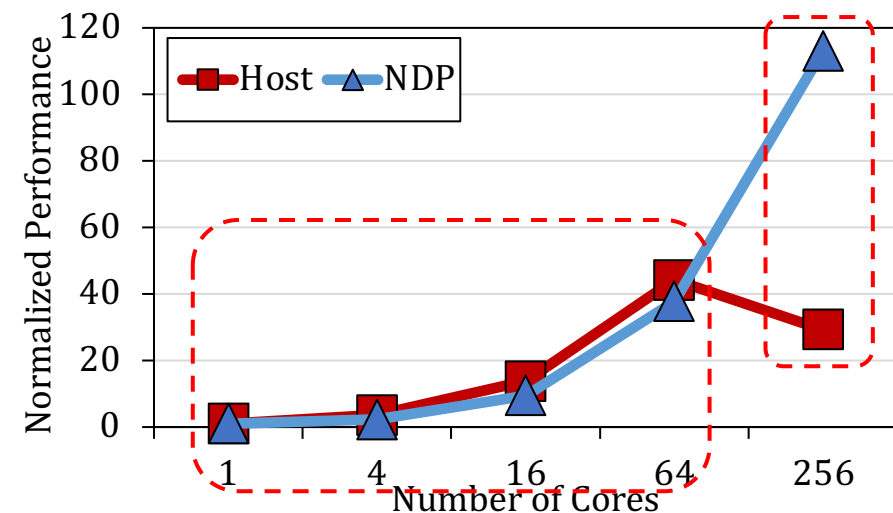
- Increasing LFMR → **L2/L3 caches turn inefficient**
- Host **scales better** than the NDP **at low core counts**
- NDP **scales better** than host **at high core counts**
- NDP performs better than host at high core counts since it **reduces memory access latency**

Temp. Loc: *high*

LFMR: *increasing*

MPKI: *low*

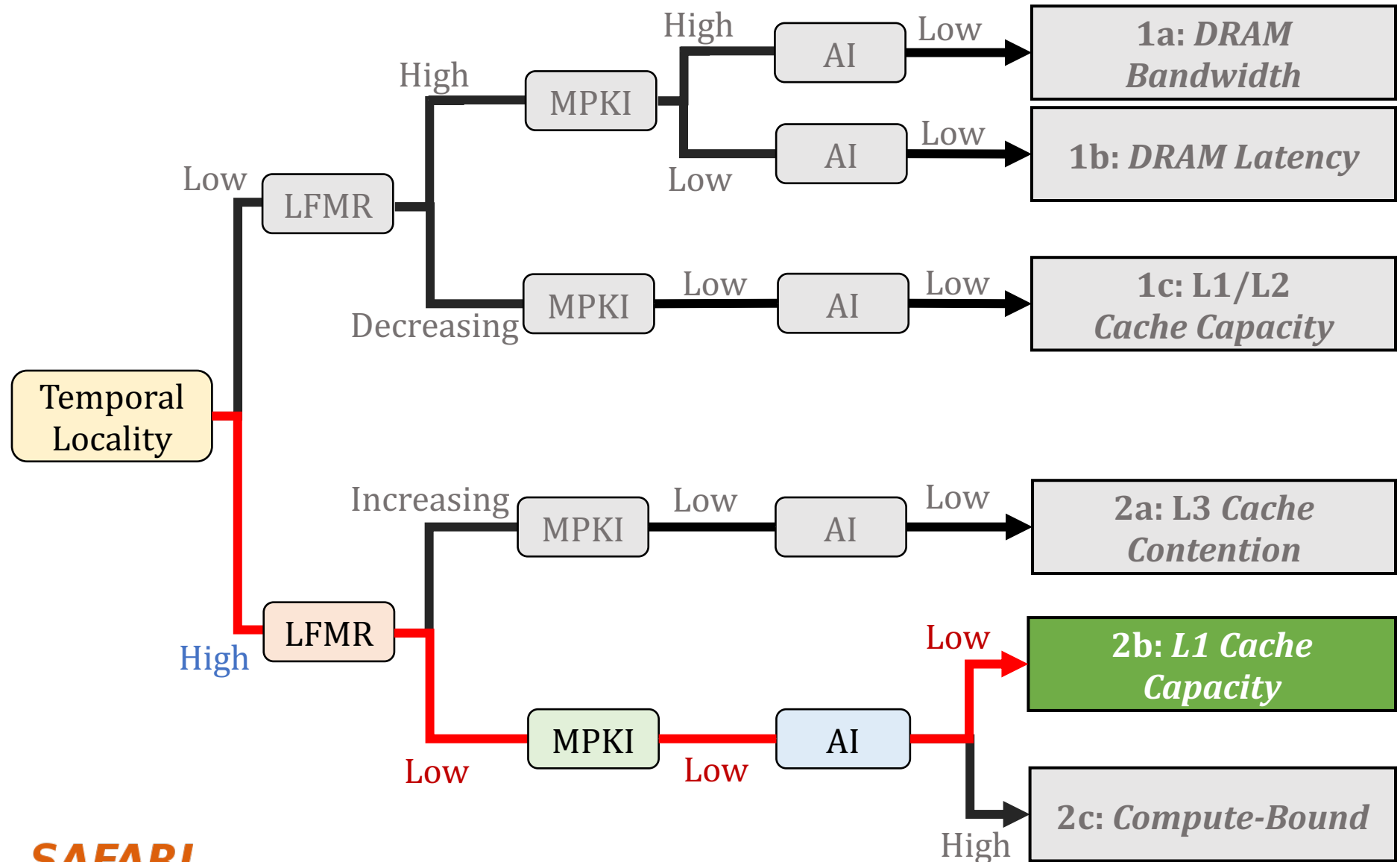
AI: *low*



**L3 cache contention bottlenecked applications:**  
at high core counts, applications turn into DRAM latency-bound

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 2b: L1 Cache Capacity

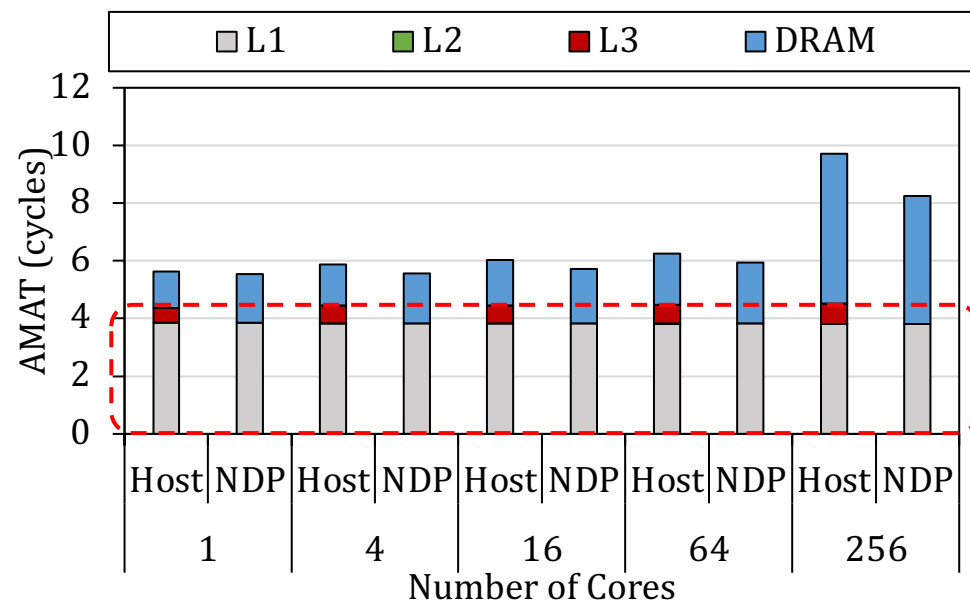
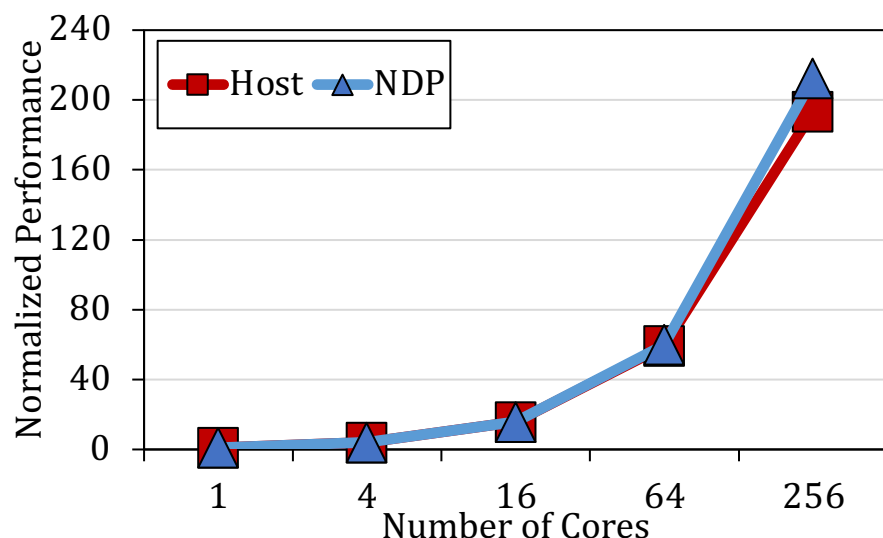
- Low LFMR, MPKI; high temporal locality  
→ efficient L2/L3 caches, low memory intensity
- Low AI → few operations per byte
- Host and NDP performance are similar  
→ L1 dominates average memory access time

Temp. Loc: *high*

LFMR: *low*

MPKI: *low*

AI: *low*

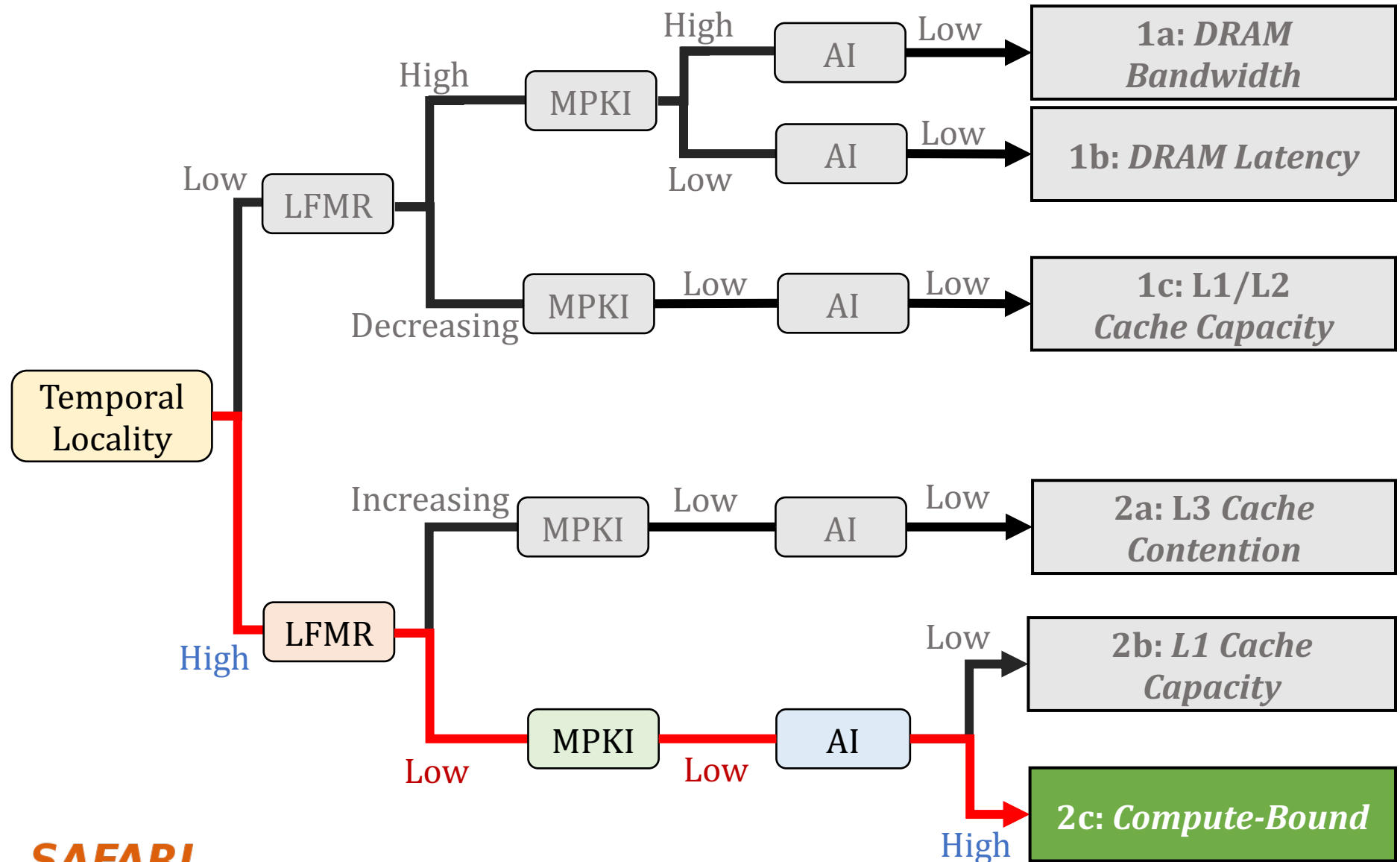


**L1 cache capacity bottlenecked applications:**

NDP can be used to **reduce** the host overall **SRAM** area

# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



# Class 2c: Compute-Bound

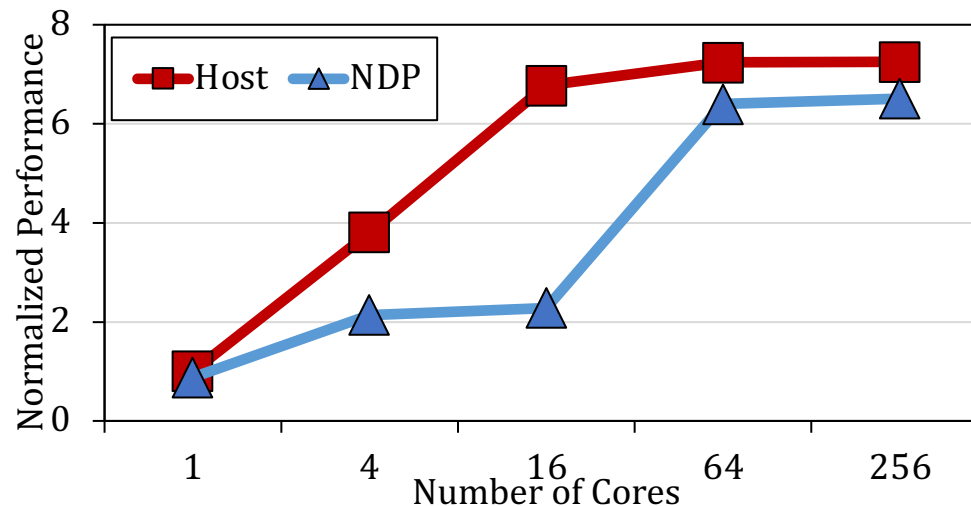
- Low LFMR, MPKI; high temporal locality  
→ efficient L2/L3 caches, low memory intensity
- High AI → many operations per byte
- Host performs better than NDP because computation dominates execution time

Temp. Loc: *high*

LFMR: *low*

MPKI: *low*

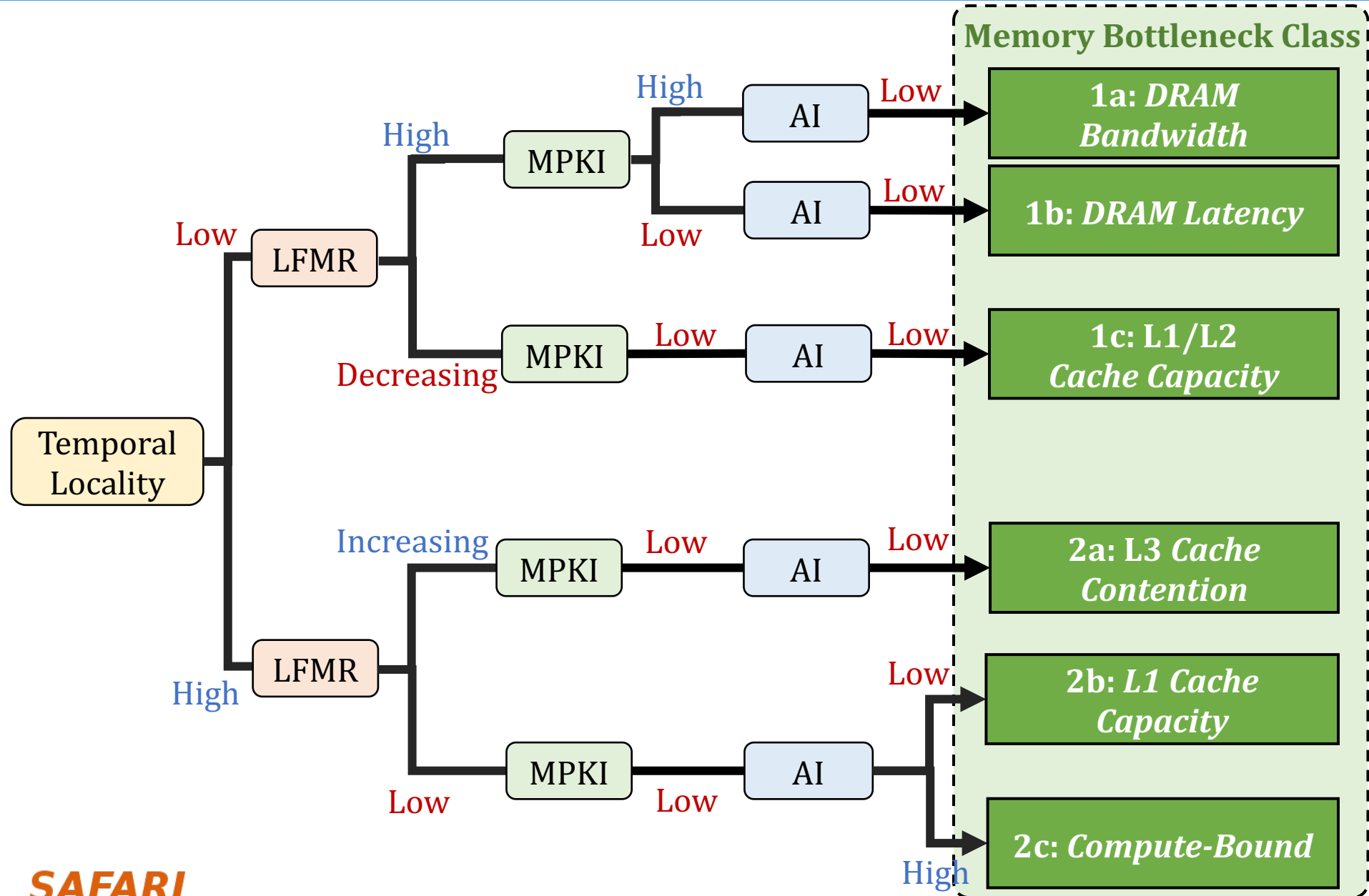
AI: *high*



**Compute-bound applications:**

benefit highly from cache hierarchy; NDP is *not* a good fit

# Step 3: Memory Bottleneck Analysis



# Step 3: Memory Bottleneck Analysis

## Memory Bottleneck Class



## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

GERALDO F. OLIVEIRA<sup>1</sup>, JUAN GÓMEZ-LUNA<sup>1</sup>, LOIS OROSA<sup>1</sup>, SAUGATA GHOSE<sup>2</sup>,  
NANDITA VIJAYKUMAR<sup>3</sup>, IVAN FERNANDEZ<sup>1,4</sup>, MOHAMMAD SADROSADATI<sup>1</sup>, and  
ONUR MUTLU<sup>1</sup>

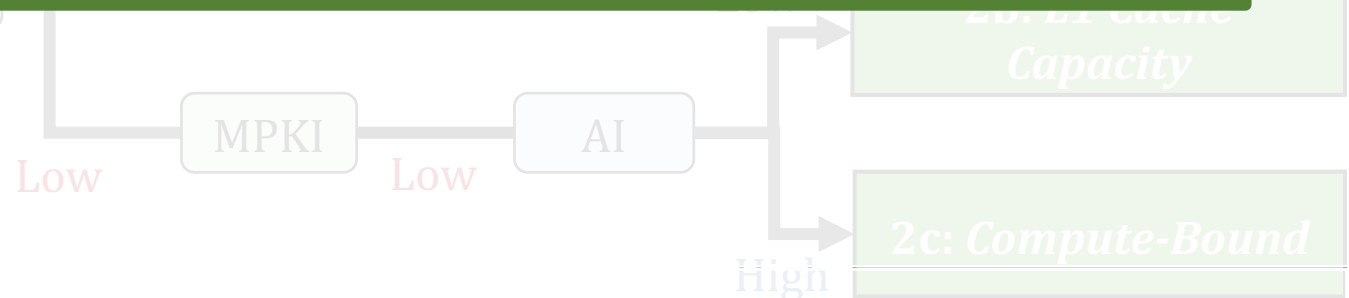
<sup>1</sup>ETH Zürich, Switzerland

<sup>2</sup>University of Illinois Urbana-Champaign, USA

<sup>3</sup>University of Toronto, Canada

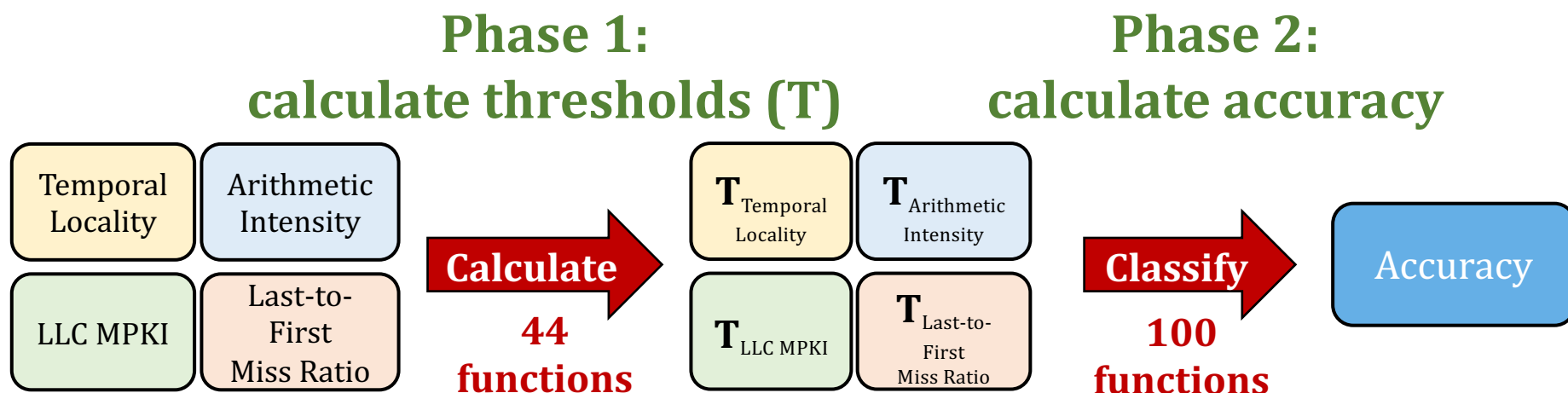
<sup>4</sup>University of Malaga, Spain

Corresponding author: Geraldo F. Oliveira (e-mail: geraldod@inf.ethz.ch).



# Methodology Validation

- **Goal:** evaluate the **accuracy** of our workload characterization methodically on a large set of functions
- Two-phase validation:



**High accuracy:**

our methodology accurately classifies 97% of functions into one of the six memory bottleneck classes

# More in the Paper

- Effect of the last-level cache size
  - Large L3 cache size (e.g., 512 MB) can **mitigate** some cache contention issues
- Summary of our workload characterization methodology
  - Including workload characterization **using in-order host/NDP cores**
- Limitations of our methodology
- Benchmark diversity

# More in the Paper

- Effect of the last-level cache size
  - Large L3 cache size (e.g., 512 MB) can mitigate some cache

## **DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks**

**GERALDO F. OLIVEIRA<sup>1</sup>, JUAN GÓMEZ-LUNA<sup>1</sup>, LOIS OROSA<sup>1</sup>, SAUGATA GHOSE<sup>2</sup>,  
NANDITA VIJAYKUMAR<sup>3</sup>, IVAN FERNANDEZ<sup>1,4</sup>, MOHAMMAD SADROSADATI<sup>1</sup>, and  
ONUR MUTLU<sup>1</sup>**

<sup>1</sup>ETH Zürich, Switzerland

<sup>2</sup>University of Illinois Urbana-Champaign, USA

<sup>3</sup>University of Toronto, Canada

<sup>4</sup>University of Malaga, Spain

Corresponding author: Geraldo F. Oliveira (e-mail: geraldod@inf.ethz.ch).

- Benchmark diversity

# Outline

1. Data Movement Bottlenecks

2. Methodology Overview

3. Application Profiling

4. Locality-Based Clustering

5. Memory Bottleneck Analysis

**6. Case Studies**

# Case Studies

- Many **open questions related to NDP** system designs<sup>8</sup>:
  - Interconnects
  - Data mapping and allocation
  - NDP core design (accelerators, general-purpose cores)
  - Offloading granularity
  - Programmability
  - Coherence
  - System integration
  - ...
- **Goal:** demonstrate how **DAMOV** is useful to study NDP system designs

[8] Mutlu+, "A Modern Primer on Processing in Memory," Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann, 2021

# Case Studies

**Load Balance and Inter-Vault Communication on NDP**

**NDP Accelerators and Our Methodology**

**Different Core Models on NDP Architectures**

**Fine-Grained NDP Offloading**

# Case Studies (1/4)

## Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults  
→ **increases the memory access latency for the NDP core**

## NDP Accelerators and Our Methodology

## Different Core Models on NDP Architectures

## Fine-Grained NDP Offloading

# Case Studies (2/4)

## Load Balance and Inter-Vault Communication on NDP

### NDP Accelerators and Our Methodology

NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c

→ **key observations hold for other NDP architectures**

### Different Core Models on NDP Architectures

### Fine-Grained NDP Offloading

# Case Studies (3/4)

**Load Balance and Inter-Vault Communication on NDP**

**NDP Accelerators and Our Methodology**

**Different Core Models on NDP Architectures**

using in-order cores limits performance of some applications  
→ **static instruction scheduling cannot exploit memory parallelism**

**Fine-Grained NDP Offloading**

# Case Studies (4/4)

**Load Balance and Inter-Vault Communication on NDP**

**NDP Accelerators and Our Methodology**

**Different Core Models on NDP Architectures**

**Fine-Grained NDP Offloading**

few basic blocks are responsible for most of LLC misses

**→ offloading such basic blocks to NDP are enough to improve performance**

# Case Studies

## Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults  
→ **increases the memory access latency for the NDP core**

## NDP Accelerators and Our Methodology

NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c  
→ **key observations hold for other NDP architectures**

## Different Core Models on NDP Architectures

using in-order cores limits performance of some applications  
→ **static instruction scheduling cannot exploit memory parallelism**

## Fine-Grained NDP Offloading

few basic blocks are responsible for most of LLC misses  
→ **offloading such basic blocks to NDP are enough to improve performance**

# Case Studies

## Load Balance and Inter-Vault Communication on NDP

### NDP Accelerators and Our Methodology

NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c

→ **key observations hold for other NDP architectures**

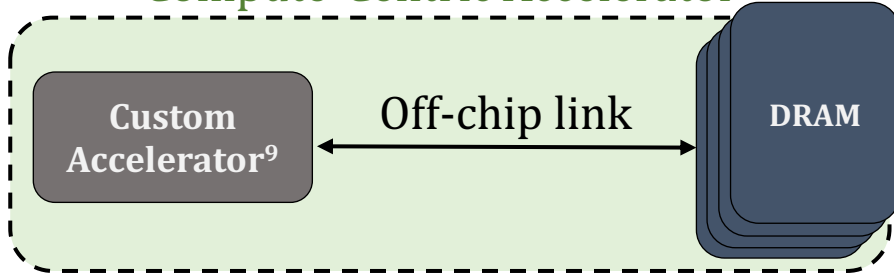
### Different Core Models on NDP Architectures

### Fine-Grained NDP Offloading

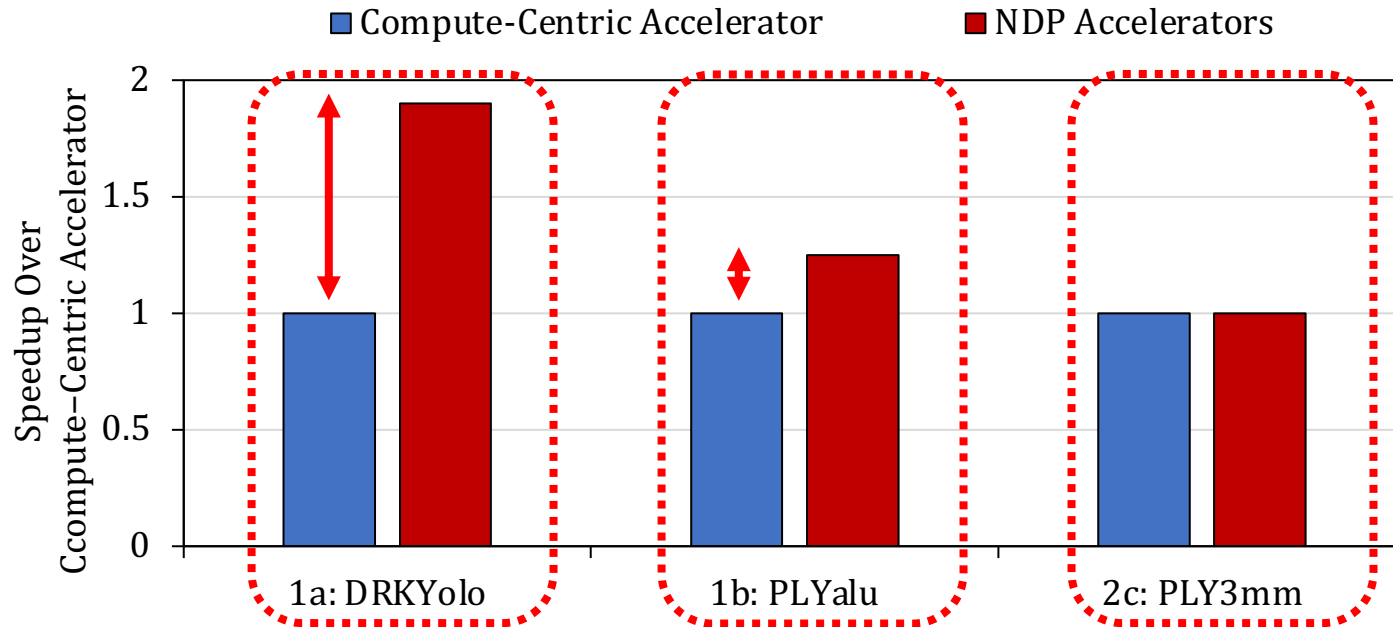
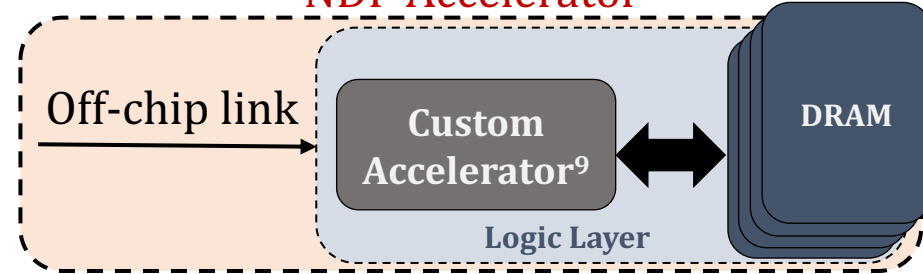
# NDP Accelerators and Our Methodology

- **Goal:** evaluate compute-centric versus NDP accelerators

Compute-Centric Accelerator



NDP Accelerator



[9] Shao+, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in ISCA, 2014

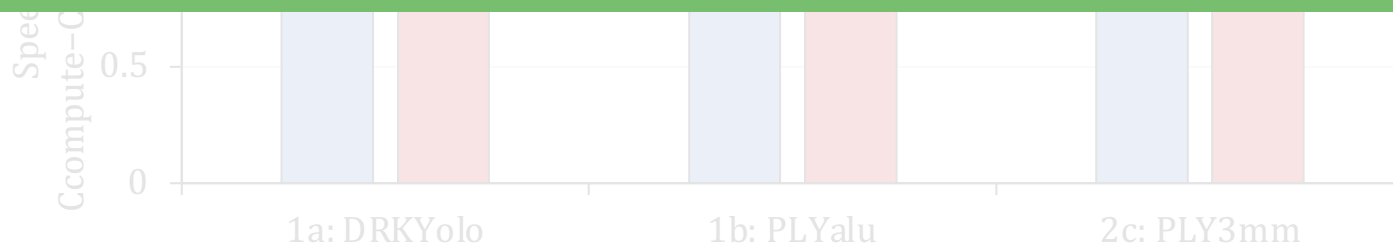
# NDP Accelerators and Our Methodology

- Goal: evaluate compute-centric versus NDP accelerators



**The performance of NDP accelerators are in line with the characteristics of the memory bottleneck classes:**

our memory bottleneck classification can be applied to study other types of system configurations



[9] Shao+, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in ISCA, 2014

# Case Studies

## Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults  
→ **increases the memory access latency for the NDP core**

## NDP Accelerators and Our Methodology

NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c  
→ **key observations hold for other NDP architectures**

## Different Core Models on NDP Architectures

using in-order cores limits performance of some applications  
→ **static instruction scheduling cannot exploit memory parallelism**

## Fine-Grained NDP Offloading

few basic blocks are responsible for most of LLC misses  
→ **offloading such basic blocks to NDP are enough to improve performance**

# Case Studies

## Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults

### **DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks**

**GERALDO F. OLIVEIRA<sup>1</sup>, JUAN GÓMEZ-LUNA<sup>1</sup>, LOIS OROSA<sup>1</sup>, SAUGATA GHOSE<sup>2</sup>,  
NANDITA VIJAYKUMAR<sup>3</sup>, IVAN FERNANDEZ<sup>1,4</sup>, MOHAMMAD SADROSADATI<sup>1</sup>, and  
ONUR MUTLU<sup>1</sup>**

<sup>1</sup>ETH Zürich, Switzerland

<sup>2</sup>University of Illinois Urbana-Champaign, USA

<sup>3</sup>University of Toronto, Canada

<sup>4</sup>University of Malaga, Spain

Corresponding author: Geraldo F. Oliveira (e-mail: geraldod@inf.ethz.ch).

## Fine-Grained NDP Offloading

few basic blocks are responsible for most of LLC misses

→ offloading such basic blocks to NDP are enough to improve performance

# DAMOV is Open-Source

- We open-source our benchmark suite and our toolchain

CMU-SAFARI / DAMOV

<> Code Issues Pull requests Actions Projects Security Insights Settings

main 1 branch 0 tags

Go to file

Add file

Code

About



DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing. Described by Oliveira et al. (preliminary version at <https://arxiv.org/pdf/2105.03725.pdf>)

Readme

Releases

No releases published  
[Create a new release](#)

Packages

No packages published  
[Publish your first package](#)

Languages



omutlu Update README.md

ce1b4ea 17 days ago 5 commits

simulator

Cleaning

19 days ago

README.md

Update README.md

17 days ago

get\_workloads.sh

DAMOV -- first commit

19 days ago

README.md

## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including [BWA](#), [Chai](#), [Darknet](#), [GASE](#), [Hardware Effects](#), [Hashjoin](#), [HPCC](#), [HPCG](#), [Ligra](#), [PARSEC](#), [Parboil](#), [PolyBench](#), [Phoenix](#), [Rodinia](#), [SPLASH-2](#), [STREAM](#).

DAMOV-SIM

DAMOV  
Benchmark

# DAMOV is Open-Source

- We open-source our benchmark suite and our toolchain

CMU-SAFARI / DAMOV

<> Code Issues Pull requests Actions Projects Security Insights Settings

main 1 branch 0 tags

Go to file

Add file

Code

About

DAMOV is a benchmark suite and a methodical framework targeting the

omutlu Update README.md

celb4ea 17 days ago 5 commits

## Get DAMOV at:

<https://github.com/CMU-SAFARI/DAMOV>

## DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.

### Releases

No releases published  
[Create a new release](#)

### Packages

No packages published  
[Publish your first package](#)

### Languages

# Conclusion

- **Problem**: Data movement is a major bottleneck in modern systems. However, it is **unclear** how to identify:
  - **different sources** of data movement bottlenecks
  - the **most suitable** mitigation technique (e.g., caching, prefetching, near-data processing) for a given data movement bottleneck
- **Goals**:
  1. Design a methodology to **identify** sources of data movement bottlenecks
  2. **Compare** compute- and memory-centric data movement mitigation techniques
- **Key Approach**: Perform a large-scale application characterization to identify **key metrics** that reveal the sources of data movement bottlenecks
- **Key Contributions**:
  - **Experimental characterization** of 77K functions across 345 applications
  - A **methodology** to characterize applications based on data movement bottlenecks and their relation with different data movement mitigation techniques
  - **DAMOV**: a **benchmark suite** with **144 functions** for data movement studies
  - **Four case-studies** to highlight DAMOV's applicability to open research problems

# DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

P&S Ramulator  
29.04.2022

**Geraldo F. Oliveira**

Juan Gómez-Luna   Lois Orosa   Saugata Ghose

Nandita Vijaykumar   Ivan Fernandez   Mohammad Sadrosadati

Onur Mutlu

**SAFARI**

**ETH** *Zürich*



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN



UNIVERSITY OF  
**TORONTO**



UNIVERSIDAD  
DE MÁLAGA

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

**Nastaran Hajinazar\***

**Geraldo F. Oliveira\***

Sven Gregorio

Joao Ferreira

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**SAFARI**



SIMON FRASER  
UNIVERSITY

**ETH** zürich



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# Executive Summary

- **Motivation**: Processing-using-Memory (PuM) architectures can effectively perform bulk bitwise computation
- **Problem**: Existing PuM architectures are not widely applicable
  - Support only a limited and specific set of operations
  - Lack the flexibility to support new operations
  - Require significant changes to the DRAM subarray
- **Goals**: Design a processing-using-DRAM framework that:
  - Efficiently implements complex operations
  - Provides the flexibility to support new desired operations
  - Minimally changes the DRAM architecture
- **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

# Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

6. Conclusion

# Outline

## 1. Processing-using-DRAM

## 2. Background

## 3. SIMDGRAM

Processing-using-DRAM Substrate  
Framework

## 4. System Integration

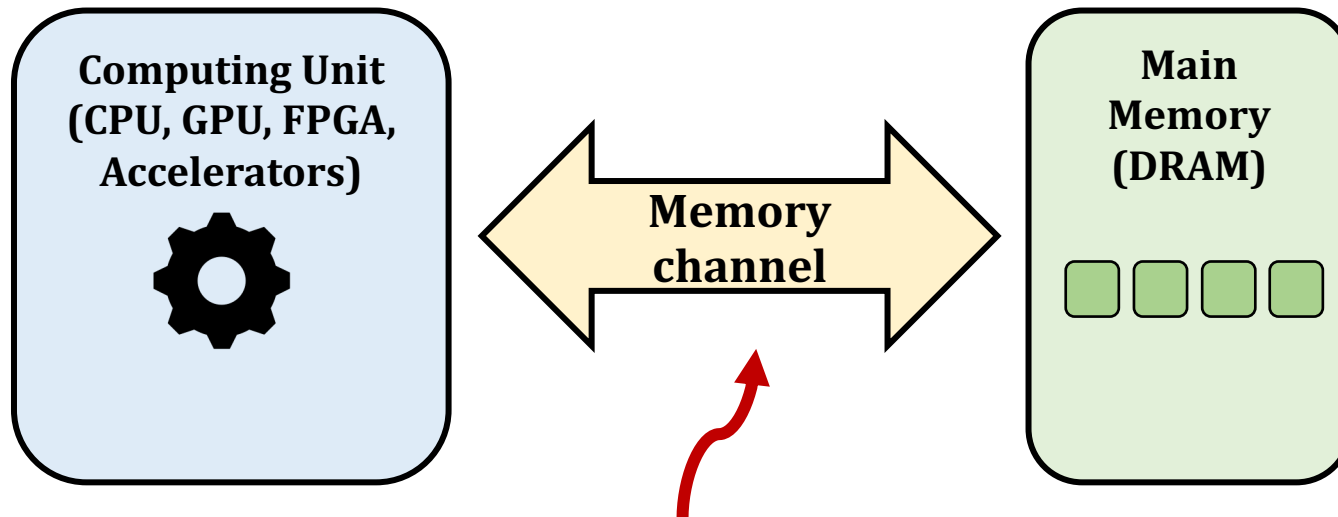
## 5. Evaluation

## 6. Conclusion

# Data Movement Bottleneck

- Data movement is a major bottleneck

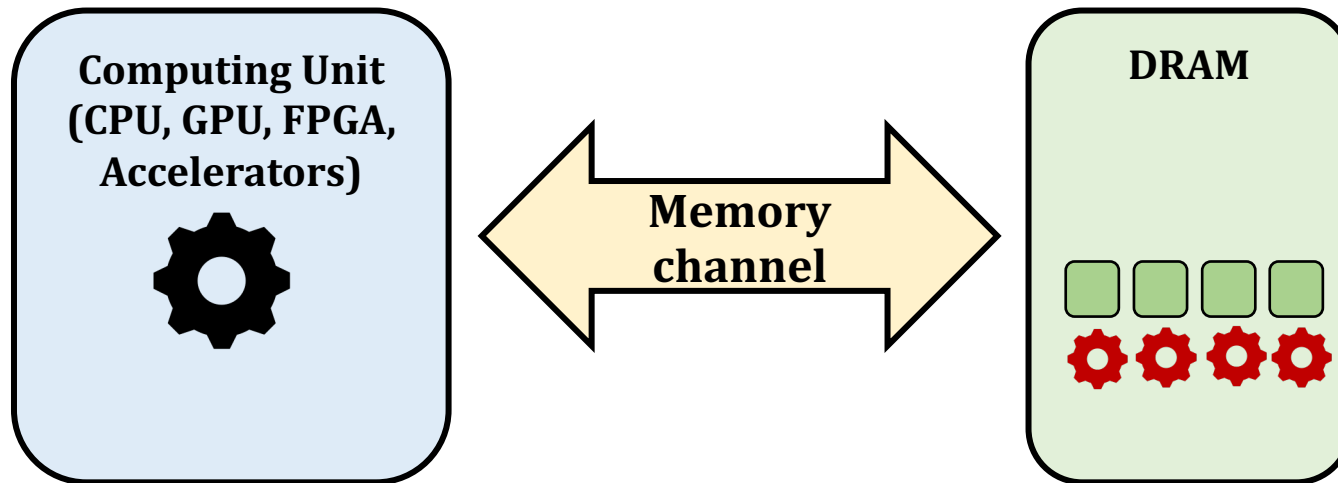
More than **60%** of the total system energy  
is spent on **data movement**<sup>1</sup>



**Bandwidth-limited and power-hungry memory channel**

# Processing-in-Memory (PIM)

- **Processing-in-Memory:** moves computation closer to where the data resides
  - **Reduces/eliminates** the need to move data between processor and DRAM



# Processing-using-Memory (PuM)

- **PuM**: Exploits analog operation principles of the memory circuitry to perform computation
  - 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)

# Outline

1. Processing-using-DRAM

2. Background

3. SIMDGRAM

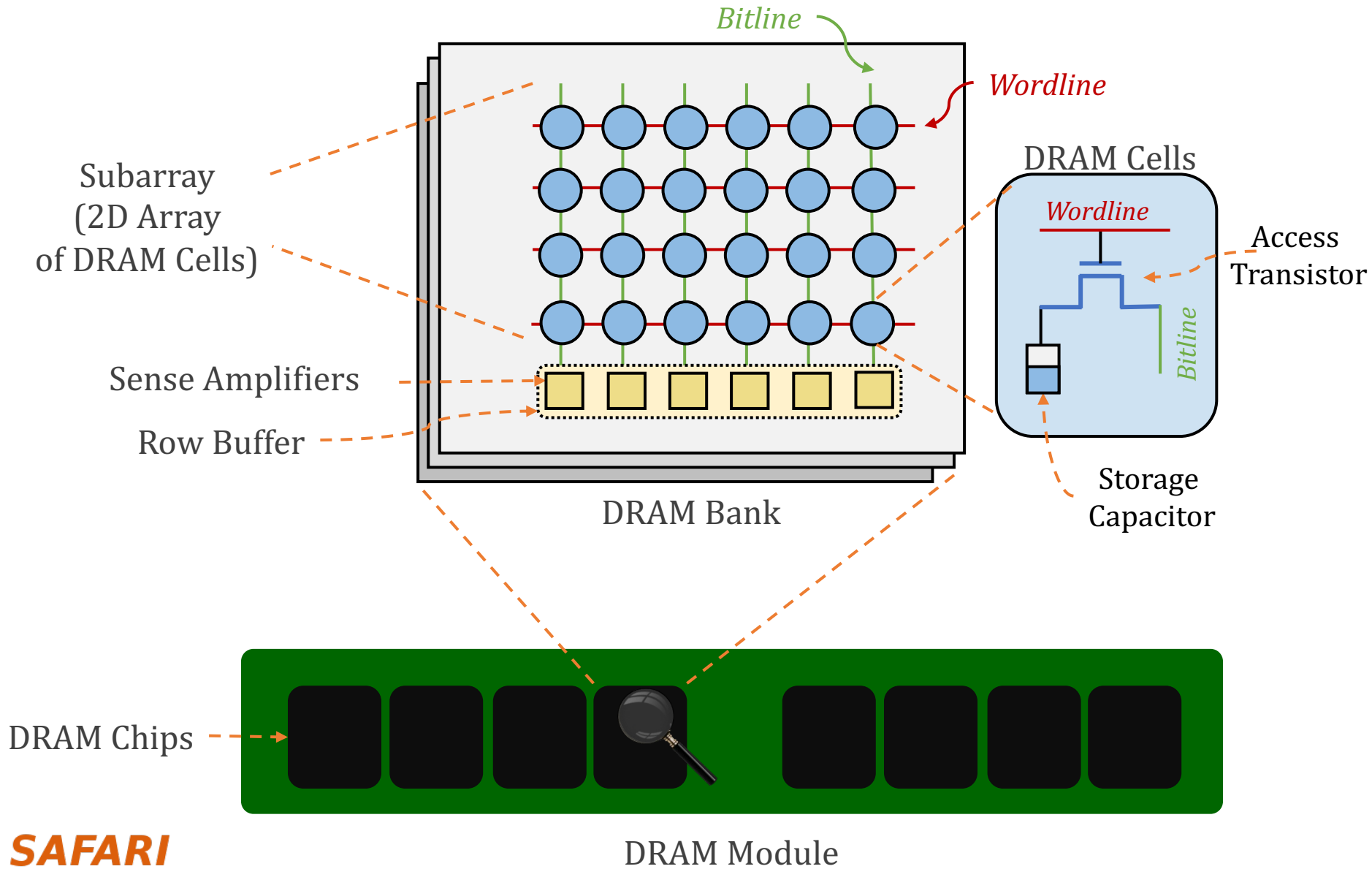
Processing-using-DRAM Substrate  
Framework

4. System Integration

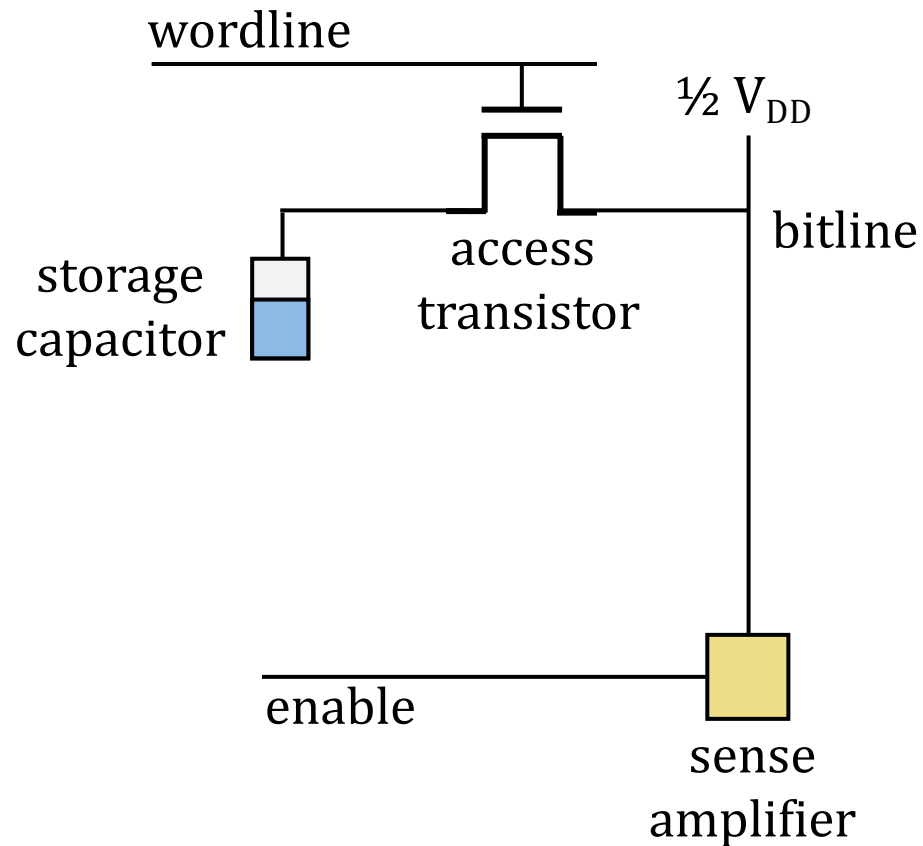
5. Evaluation

6. Conclusion

# Inside a DRAM Chip



# DRAM Cell Operation

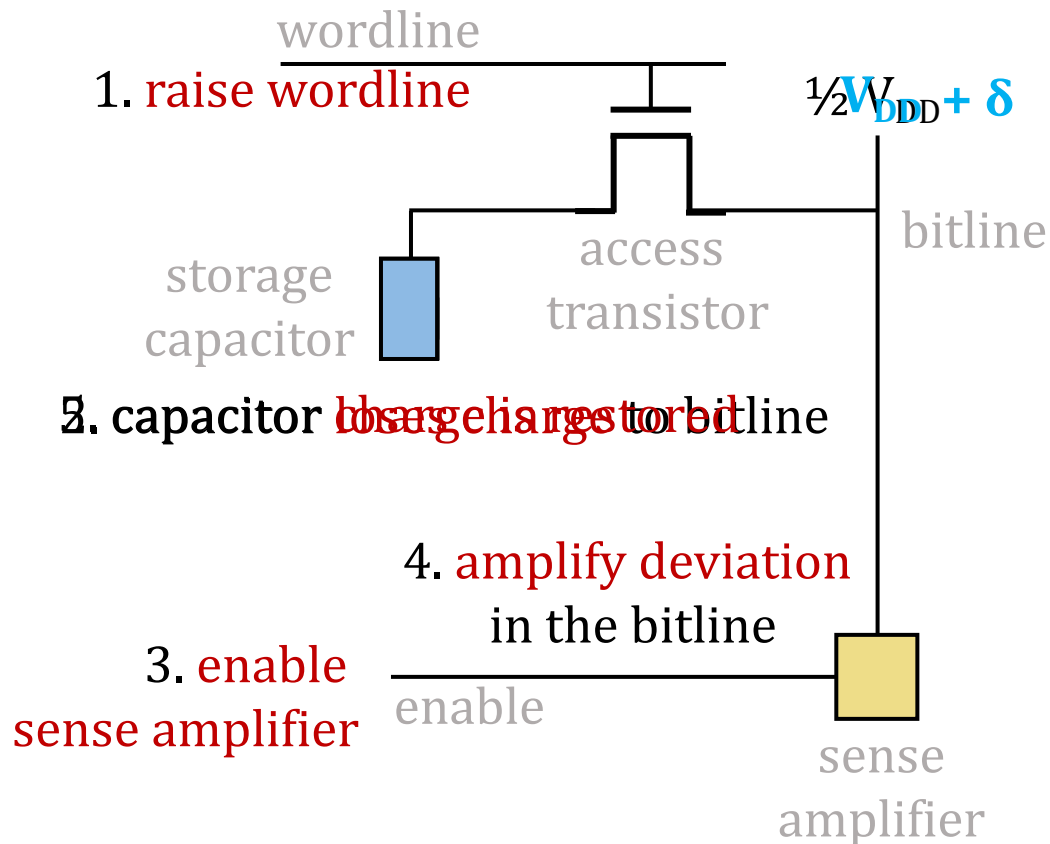


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

# DRAM Cell Operation (1/3)

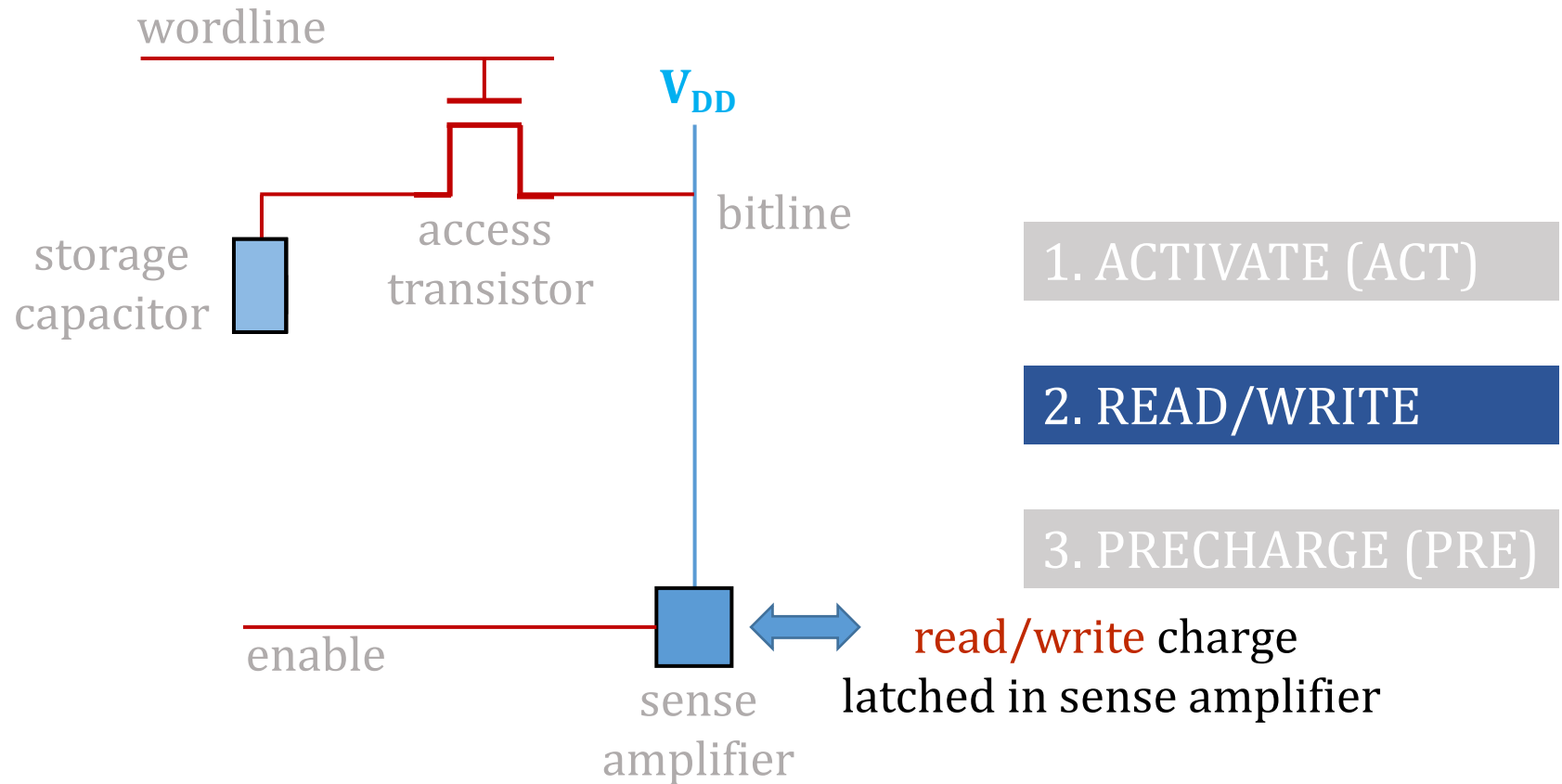


1. ACTIVATE (ACT)

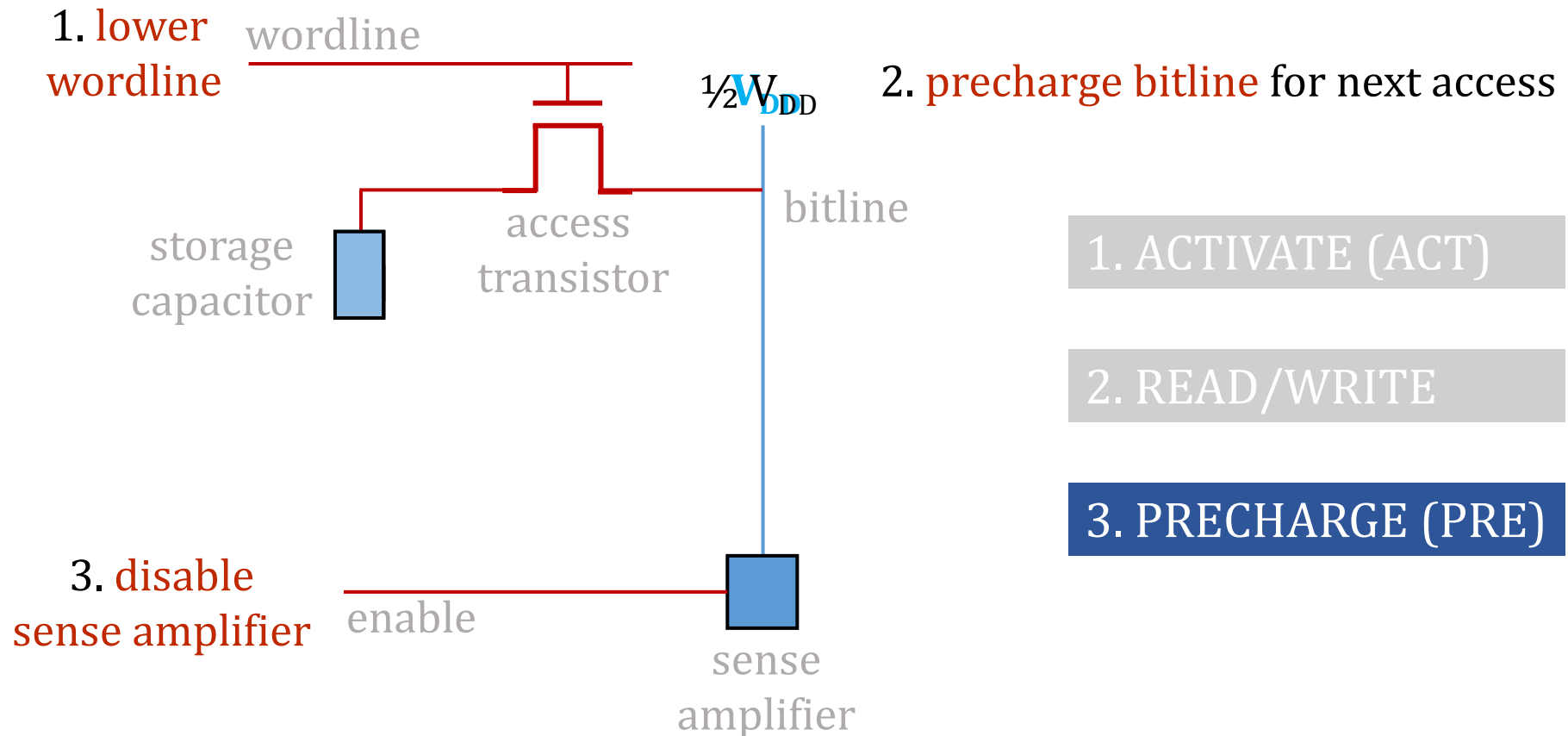
2. READ/WRITE

3. PRECHARGE (PRE)

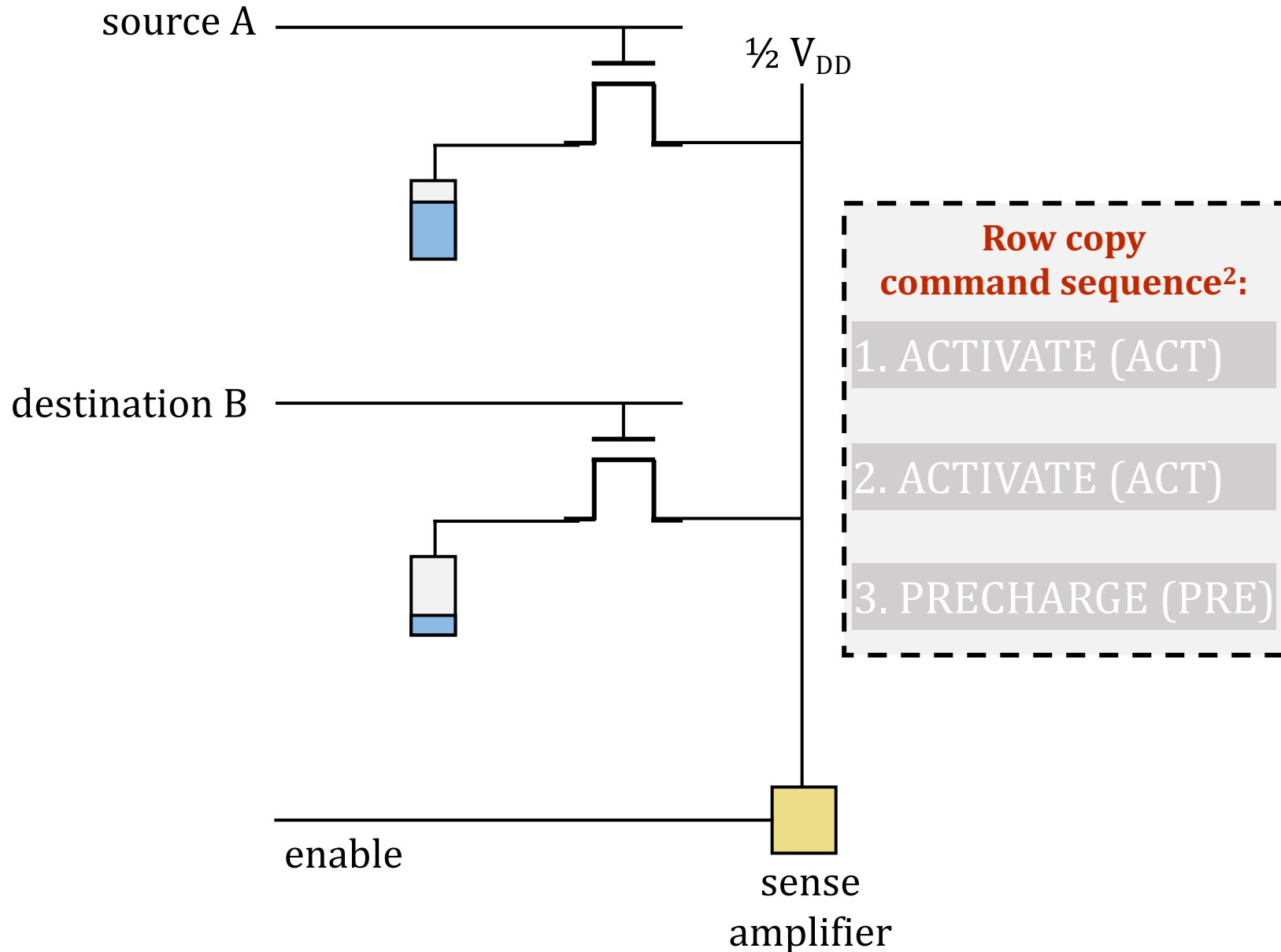
# DRAM Cell Operation (2/3)



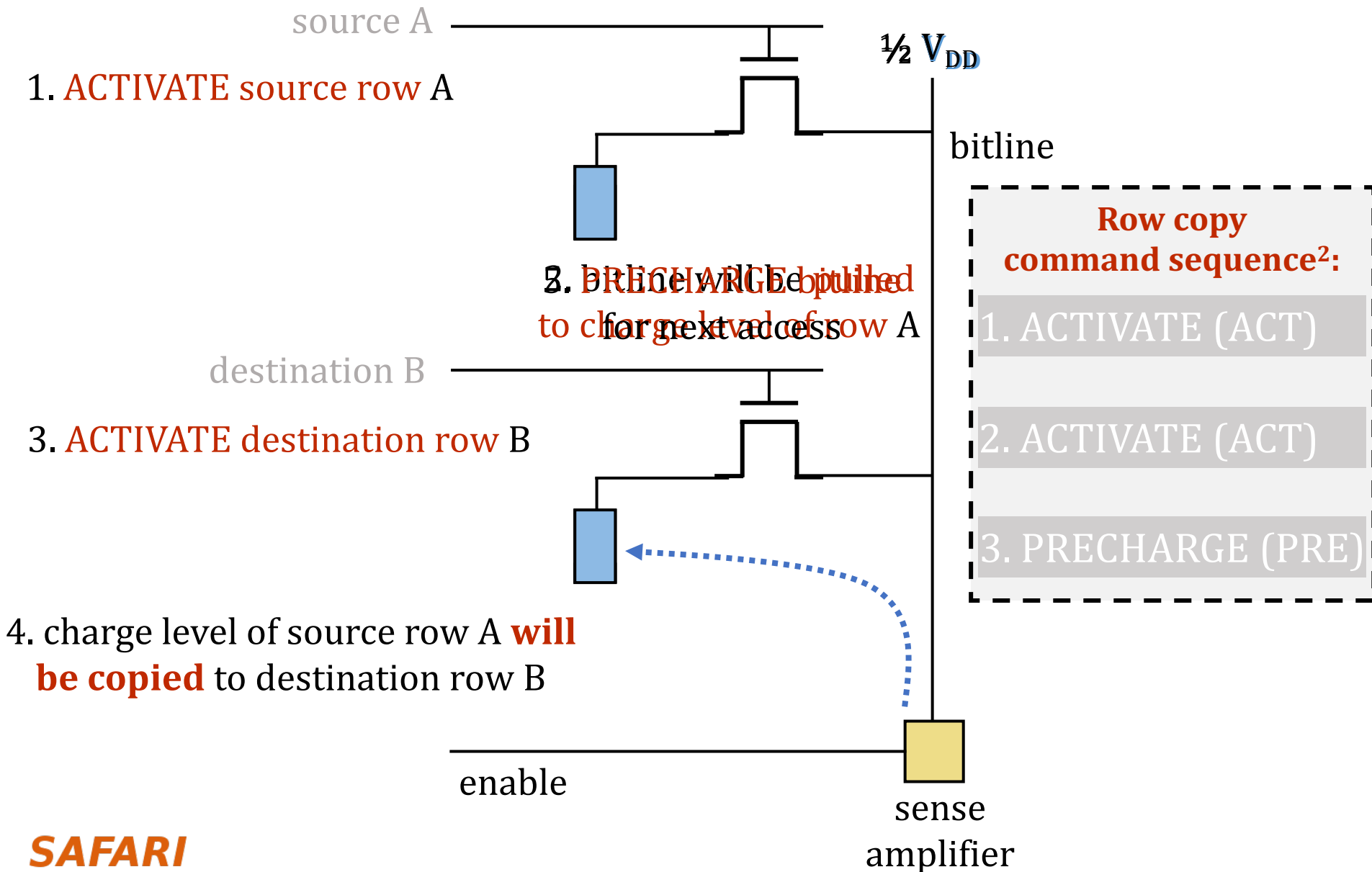
# DRAM Cell Operation (3/3)



# RowClone: In-DRAM Row Copy (1/2)

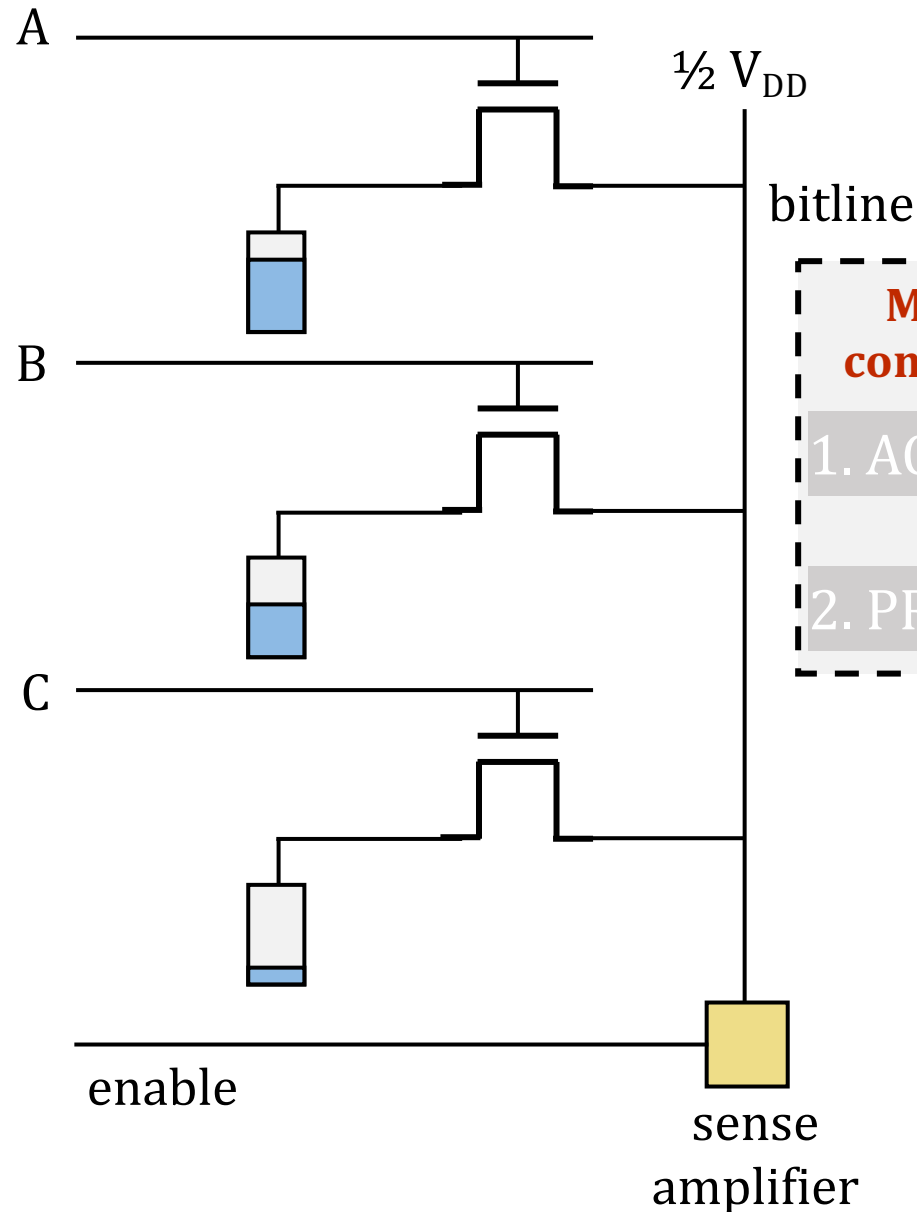


# RowClone: In-DRAM Row Copy (2/2)



<sup>2</sup> V. Seshadri et al., "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization", MICRO, 2013

# Triple-Row Activation: Majority Function



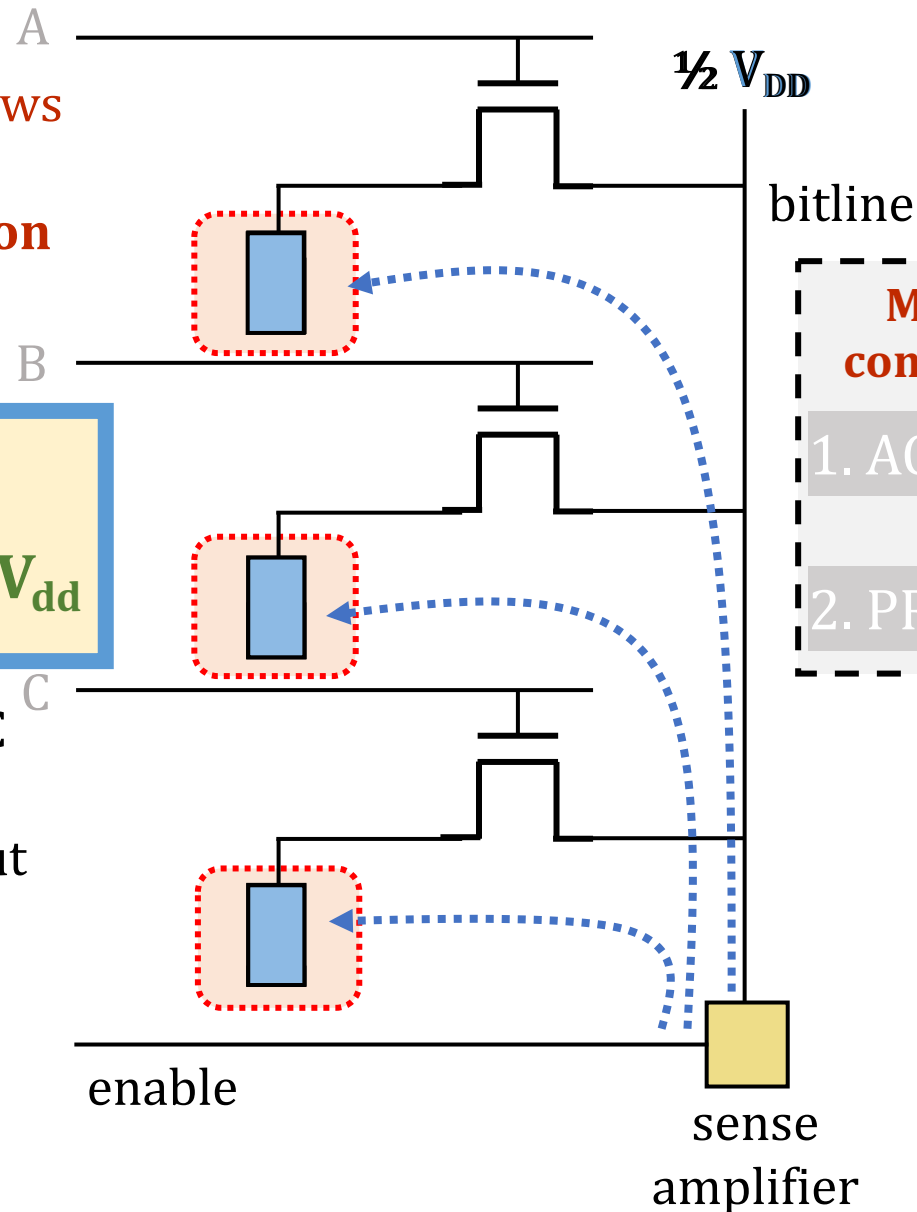
# Triple-Row Activation: Majority Function

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

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

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

4. **PRECHARGE** bitline for next access

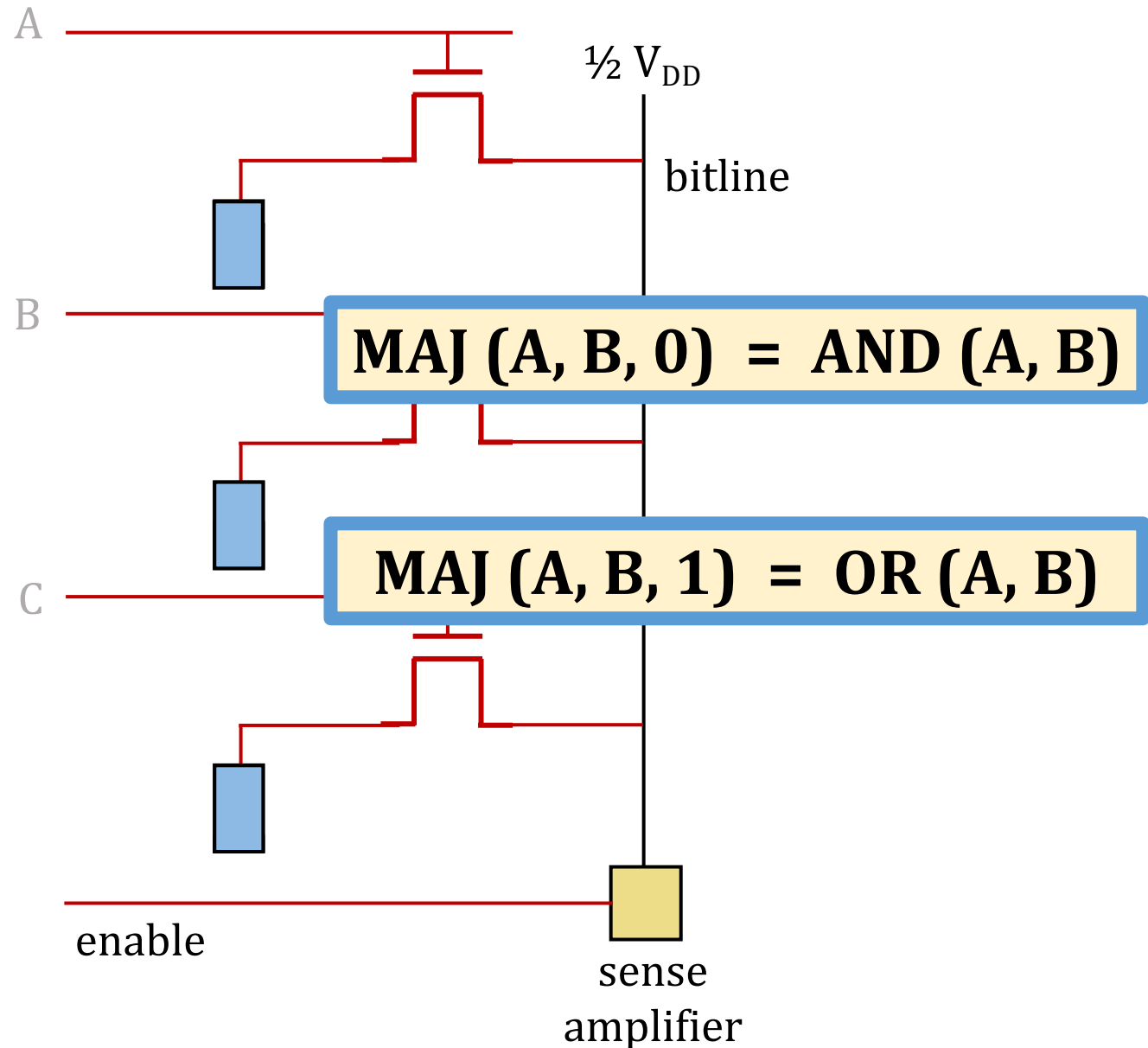


**Majority function command sequence<sup>3</sup>:**

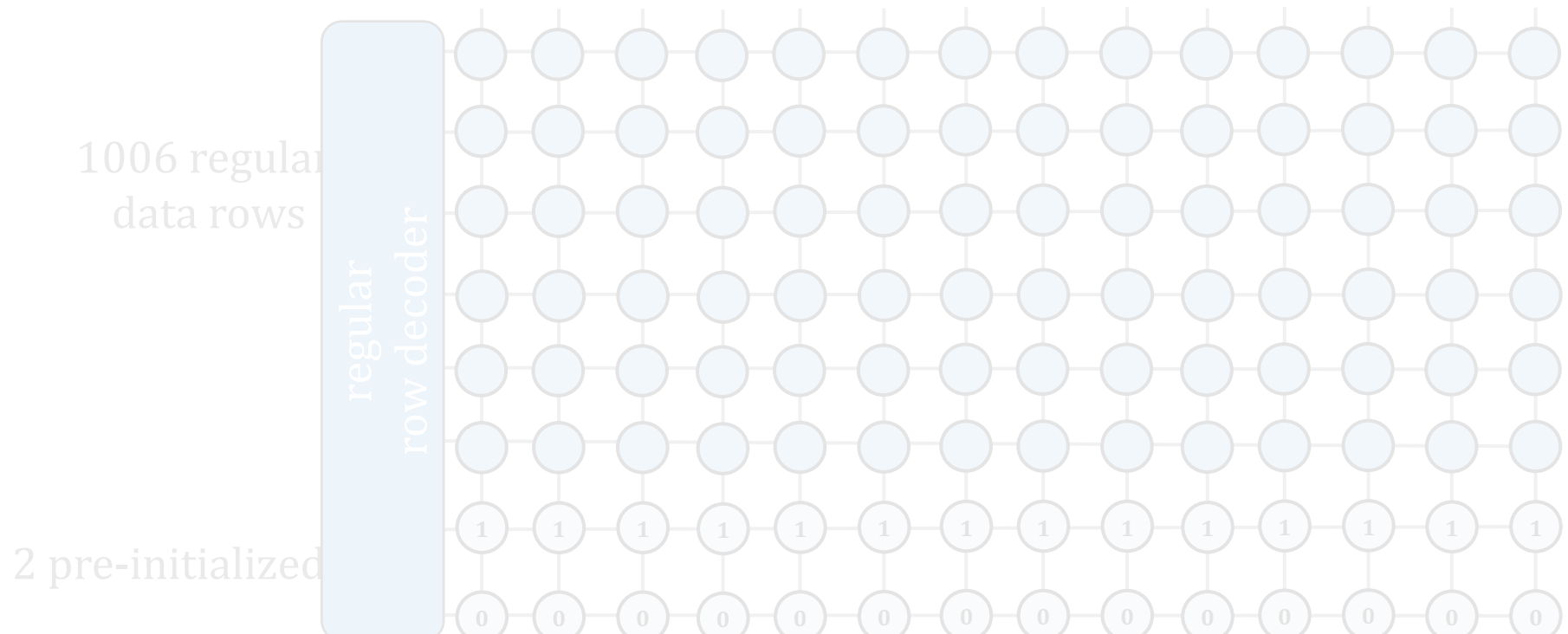
1. **ACTIVATE (ACT)**

2. **PRECHARGE (PRE)**

# Ambit: In-DRAM Bulk Bitwise AND/OR



# Ambit: Subarray Organization



**Less than 1% of overhead  
in existing DRAM chips**

# 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**

- Costly (e.g., area, power)

# 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

# Outline

1. Processing-using-DRAM

2. Background

**3. SIMD RAM**

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

6. Conclusion

# 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

# Outline

1. Processing-using-DRAM

2. Background

**3. SIMD RAM**

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

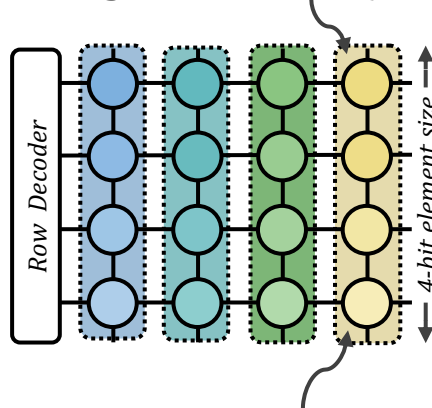
6. Conclusion

# SIMDRAM: PuM Substrate

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

## (1) Vertical data layout

most significant bit (MSB)



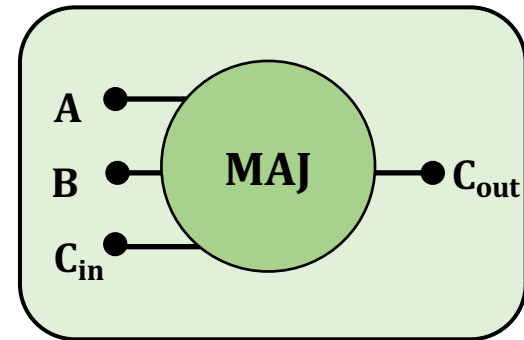
least significant bit (LSB)

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

# Outline

1. Processing-using-DRAM

2. Background

**3. SIMD RAM**

Processing-using-DRAM Substrate  
Framework

4. System Integration

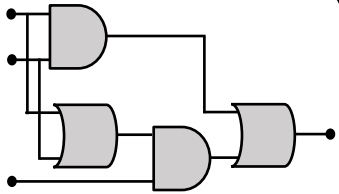
5. Evaluation

6. Conclusion

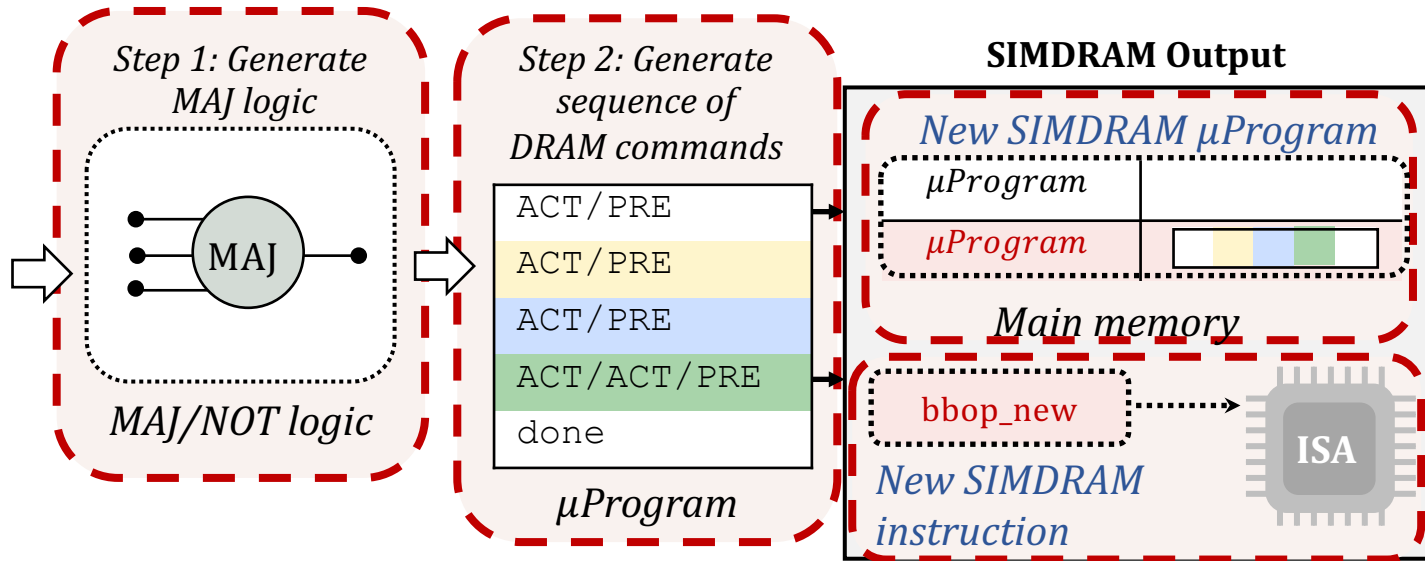
# SIMDRAM Framework

## User Input

*Desired operation*



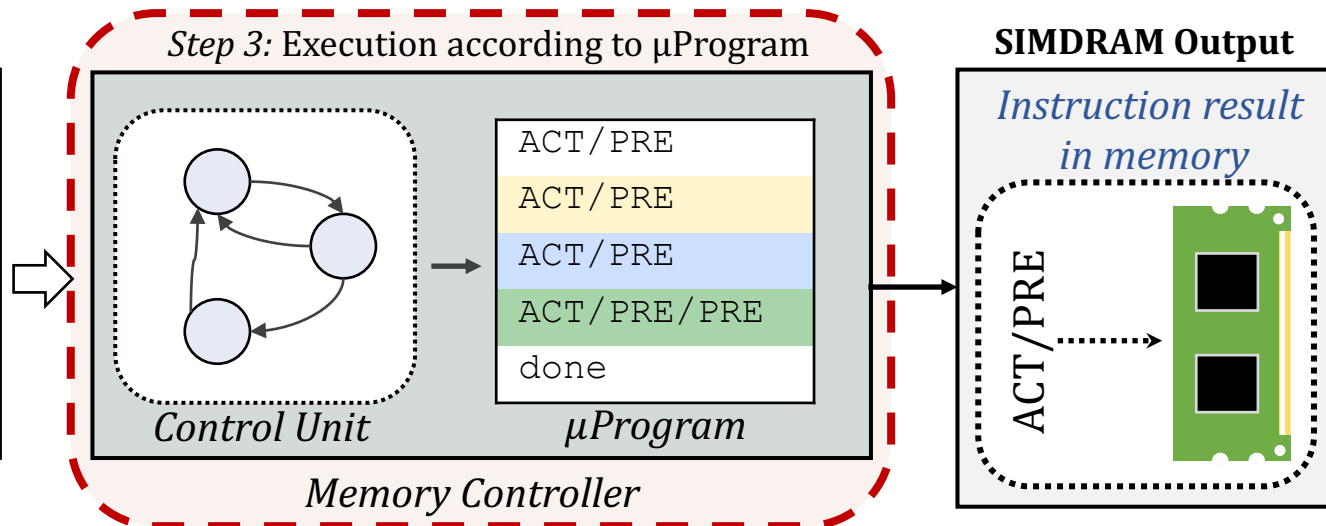
*AND/OR/NOT logic*



## User Input

*SIMDRAM-enabled application*

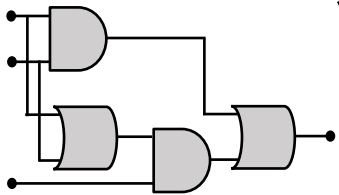
```
foo () {  
    bbop_new  
}
```



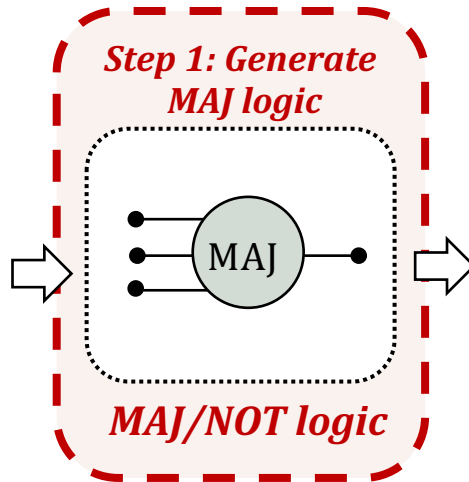
# SIMDRAM Framework: Step 1

## User Input

*Desired operation*



*AND/OR/NOT logic*



*Step 2: Generate sequence of DRAM commands*



## SIMDRAM Output

*New SIMD RAM  $\mu$ Program*

*$\mu$ Program*

*Main memory*

*bbop\_new*

*New SIMD RAM instruction*

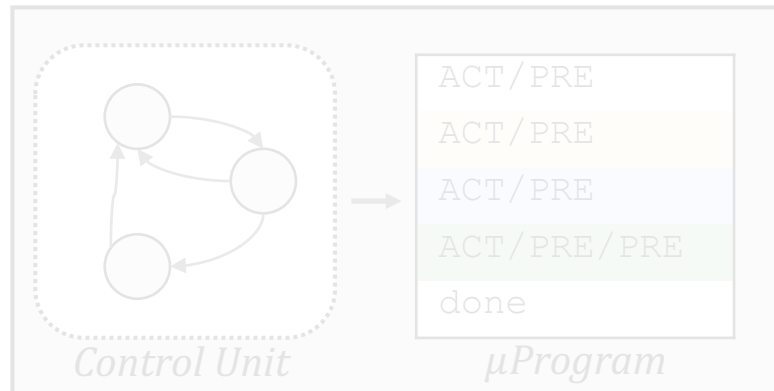
*ISA*

## User Input

*SIMDRAM-enabled application*

```
foo () {  
  bbop_new  
}
```

*Step 3: Execution according to  $\mu$ Program*

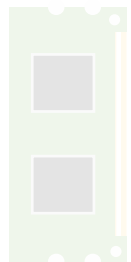


*Memory Controller*

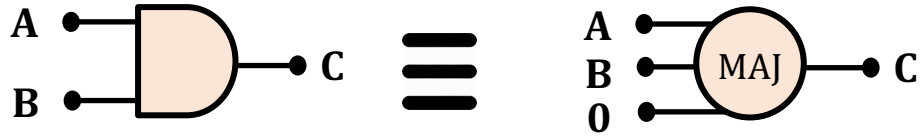
## SIMDRAM Output

*Instruction result in memory*

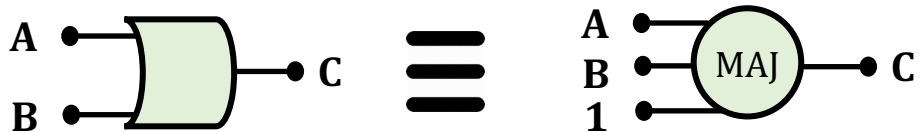
*ACT/PRE*



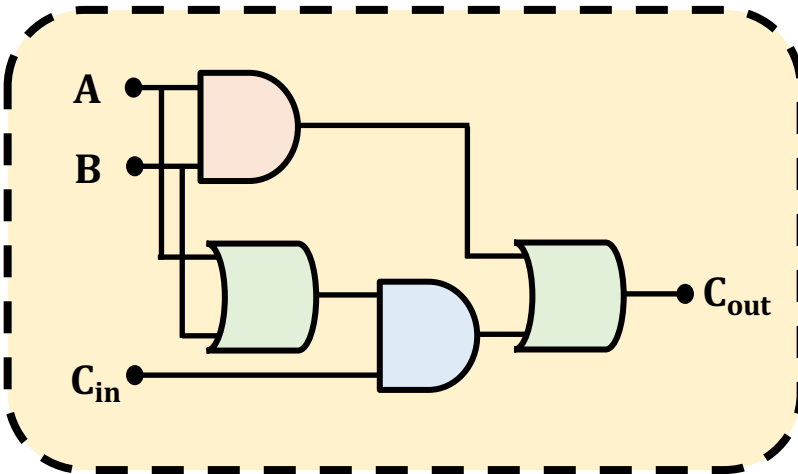
# Step 1: Naïve MAJ/NOT Implementation



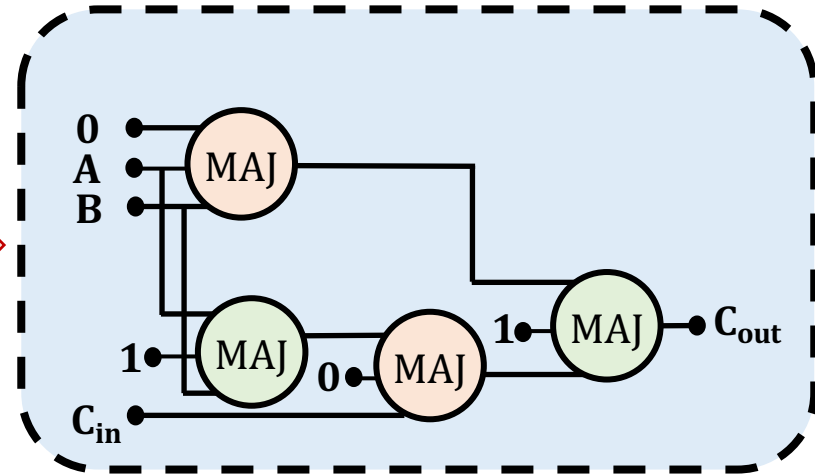
output is "1" only when  $A = B = "1"$



output is "0" only when  $A = B = "0"$

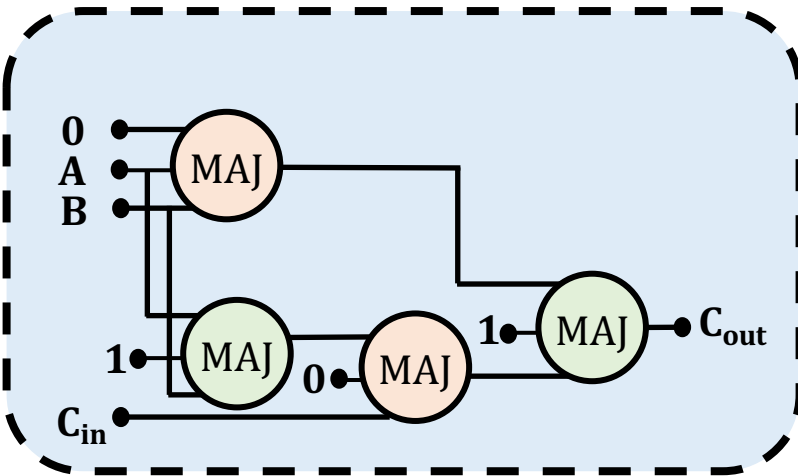


Part 1



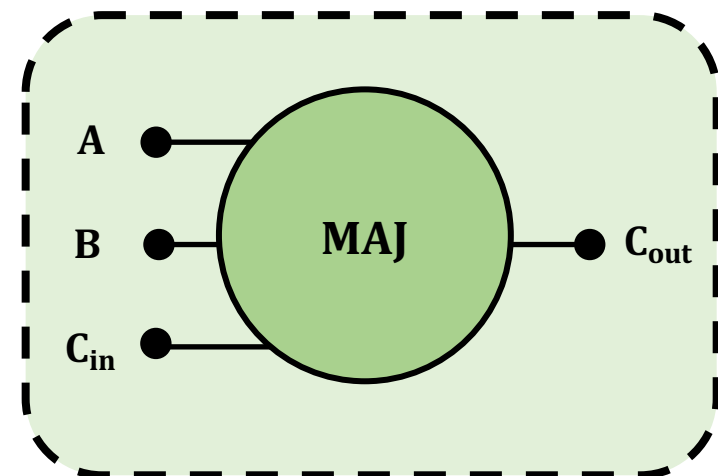
**Naïvely** converting **AND/OR/NOT-implementation** to **MAJ/NOT-implementation** leads to an **unoptimized circuit**

# Step 1: Efficient MAJ/NOT Implementation



Greedy  
optimization  
algorithm<sup>4</sup>

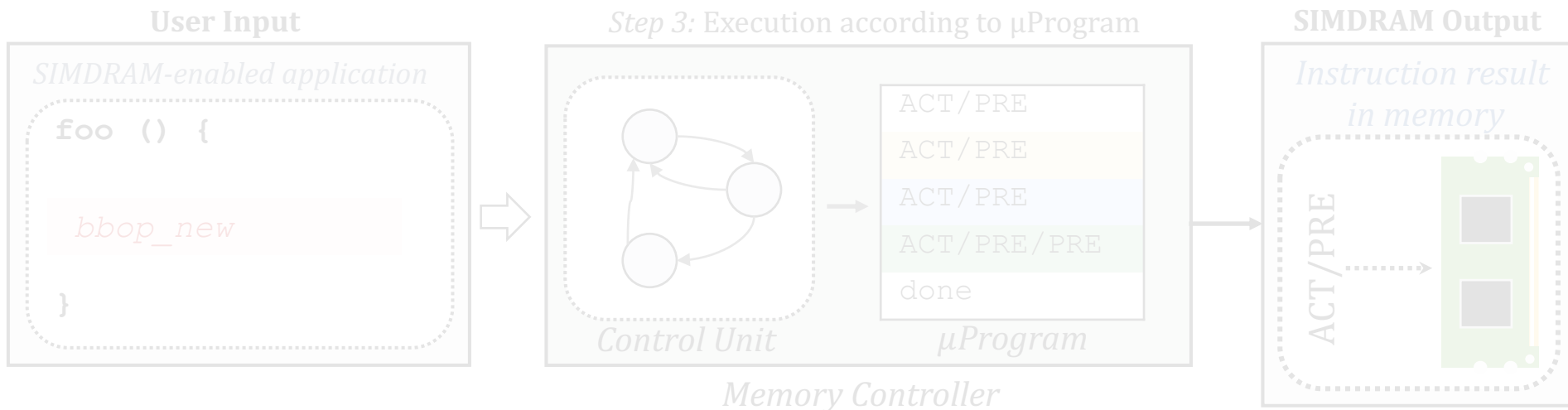
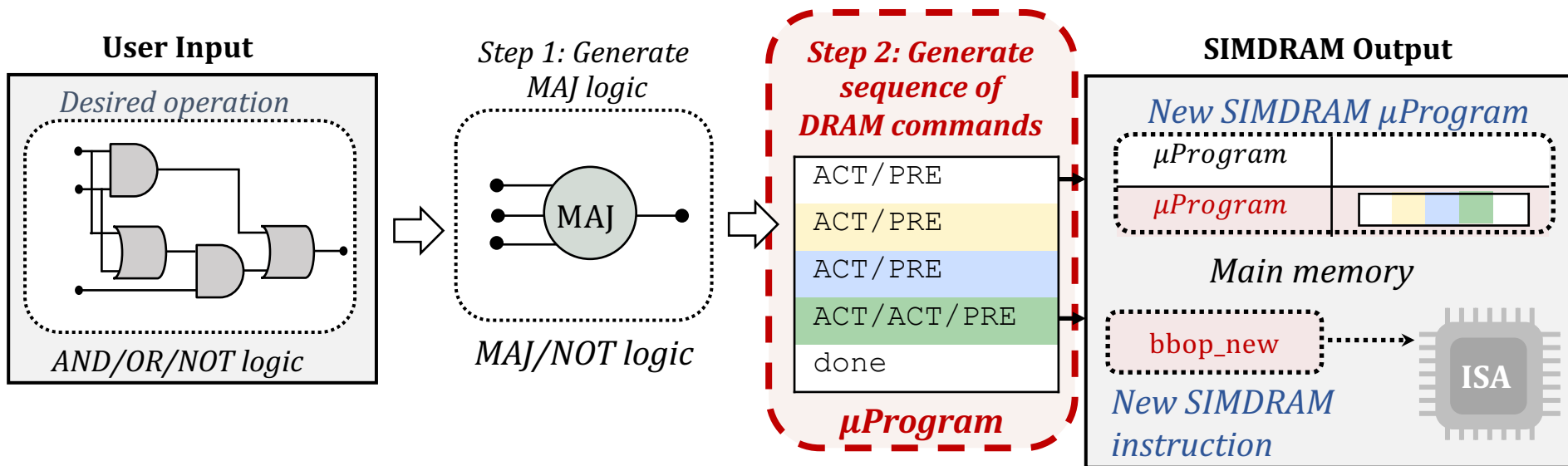
Part 2



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

<sup>4</sup> 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: $\mu$ Program Generation

- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate  $\mu$ Program

## Step 2: $\mu$ Program Generation

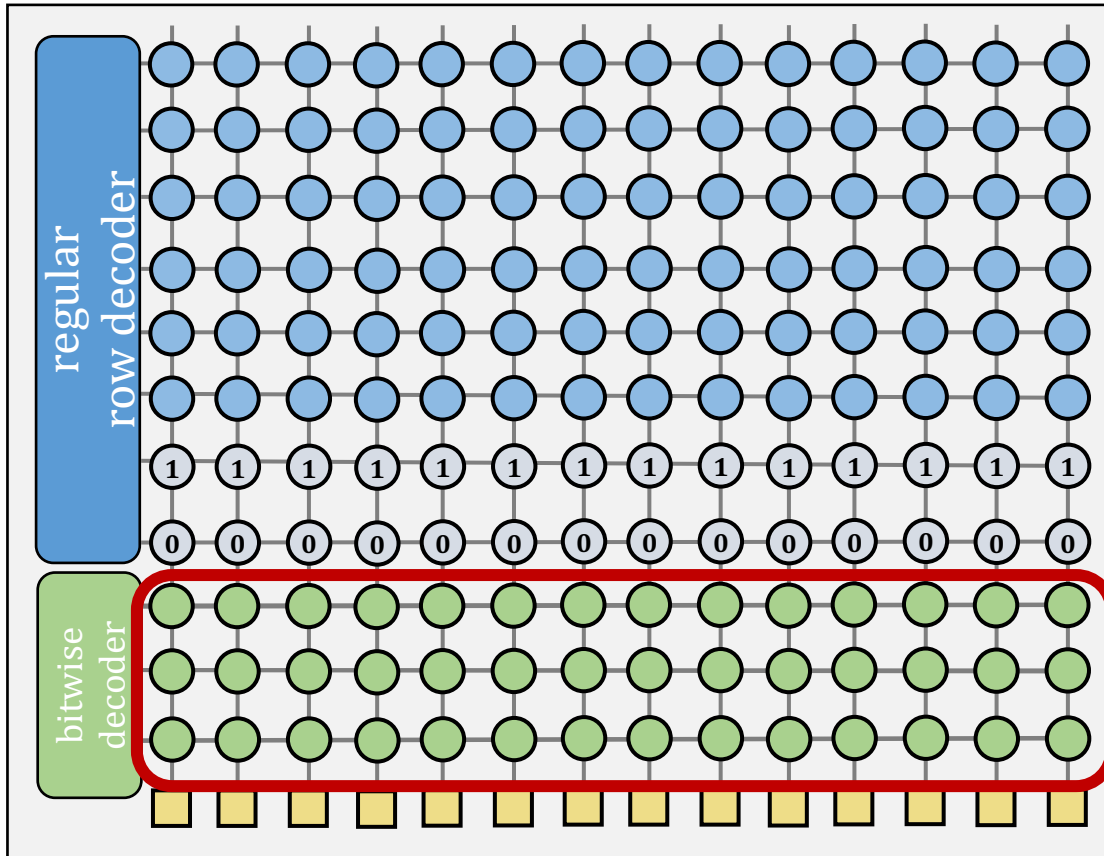
- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate  $\mu$ Program

# Task 1: Allocating DRAM Rows to Operands

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



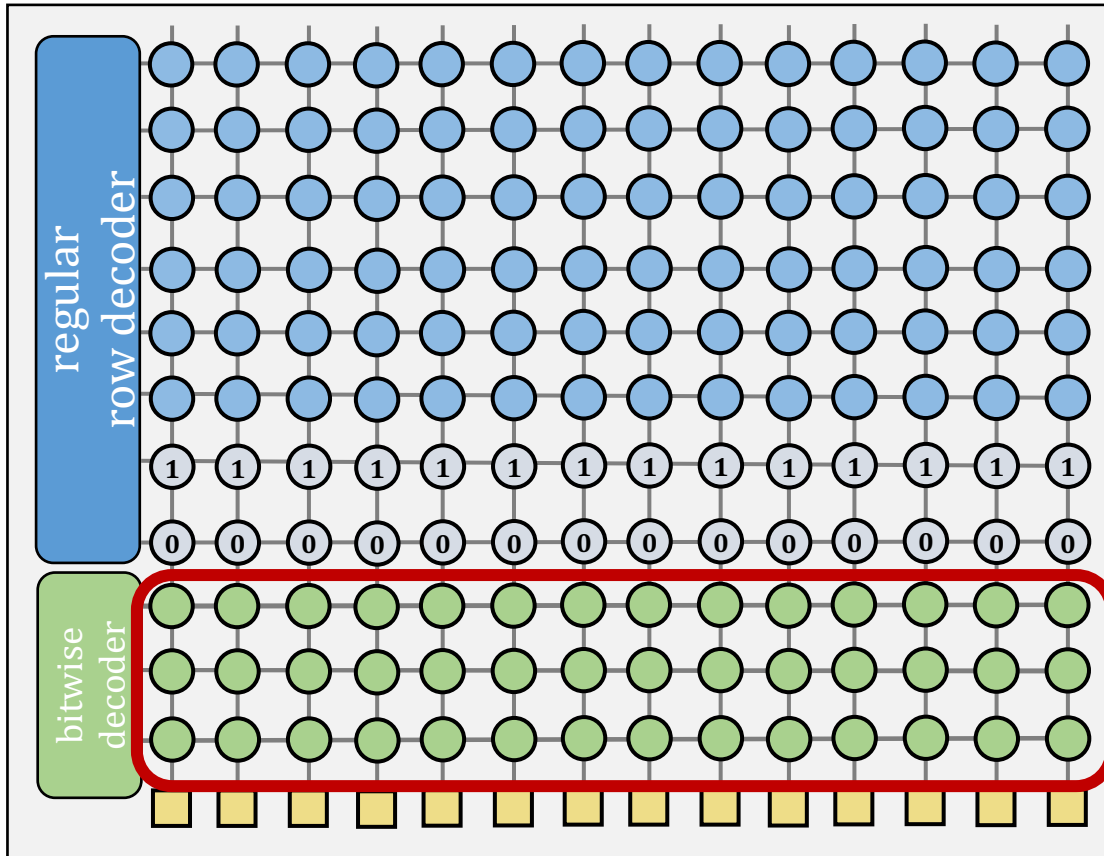
**Constraint 1:**  
**Limited** number of rows  
reserved for computation

**Compute  
rows**

**subarray organization**

# Task 1: Allocating DRAM Rows to Operands

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



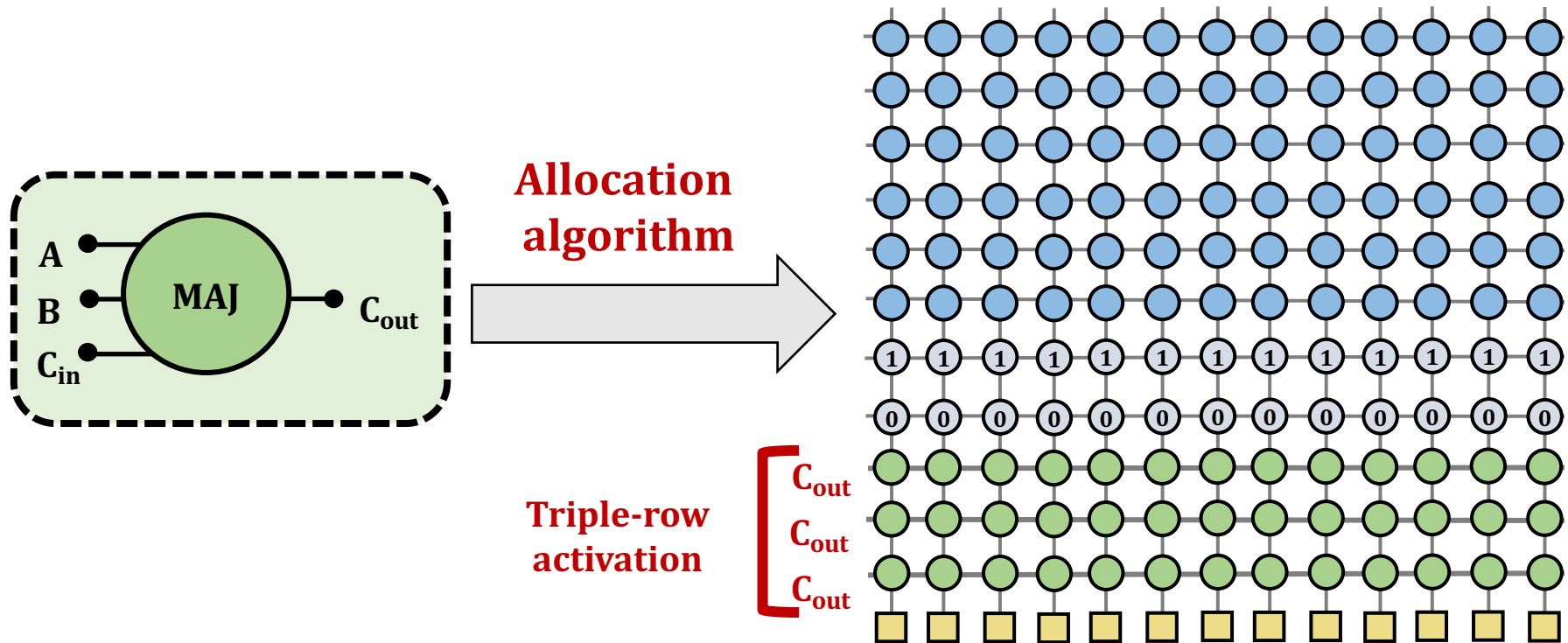
**Constraint 2:**  
**Destructive** behavior  
of triple-row activation

**Overwritten  
with MAJ output**

**subarray organization**

# 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**



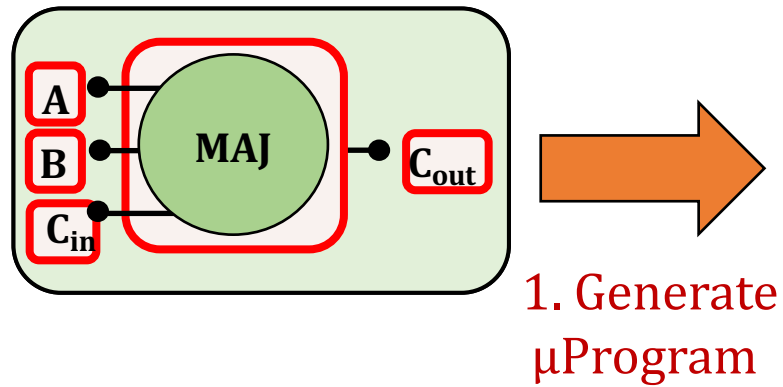
## Step 2: $\mu$ Program Generation

- **$\mu$ Program:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMD RAM uses to execute SIMD RAM operation in DRAM
- **Goal of Step 2:** To generate the  $\mu$ Program that executes the desired SIMD RAM operation in DRAM

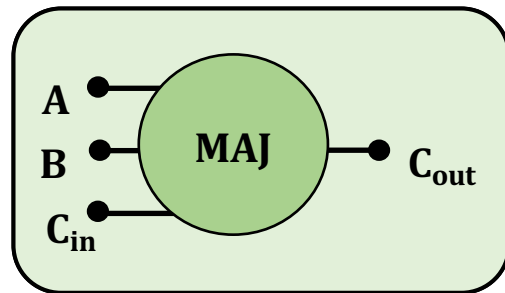
Task 1: Allocate DRAM rows to the operands

Task 2: Generate  $\mu$ Program

# Task 2: Generate an initial $\mu$ Program



# Task 2: Optimize the $\mu$ Program



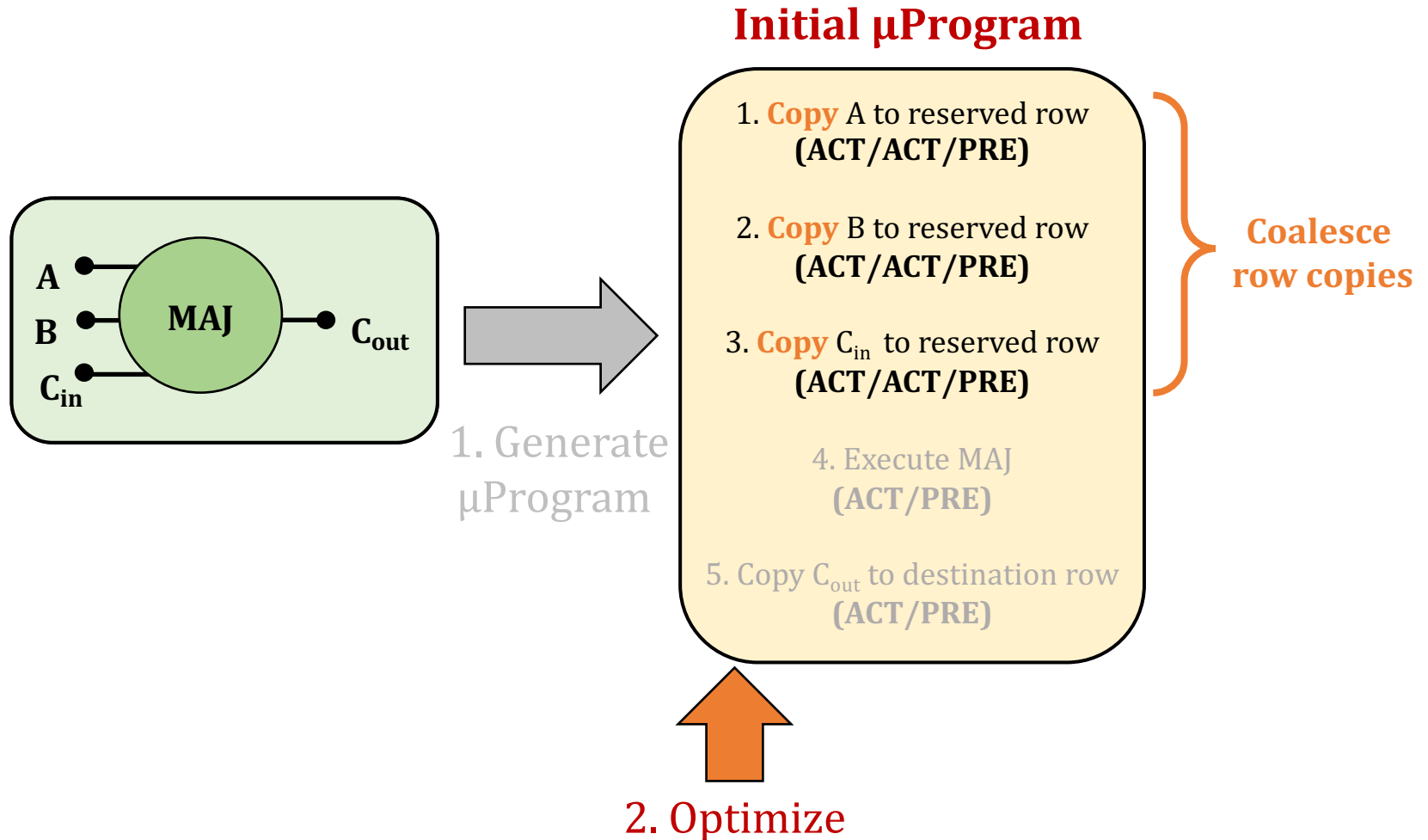
1. Generate  $\mu$ Program

## Initial $\mu$ Program

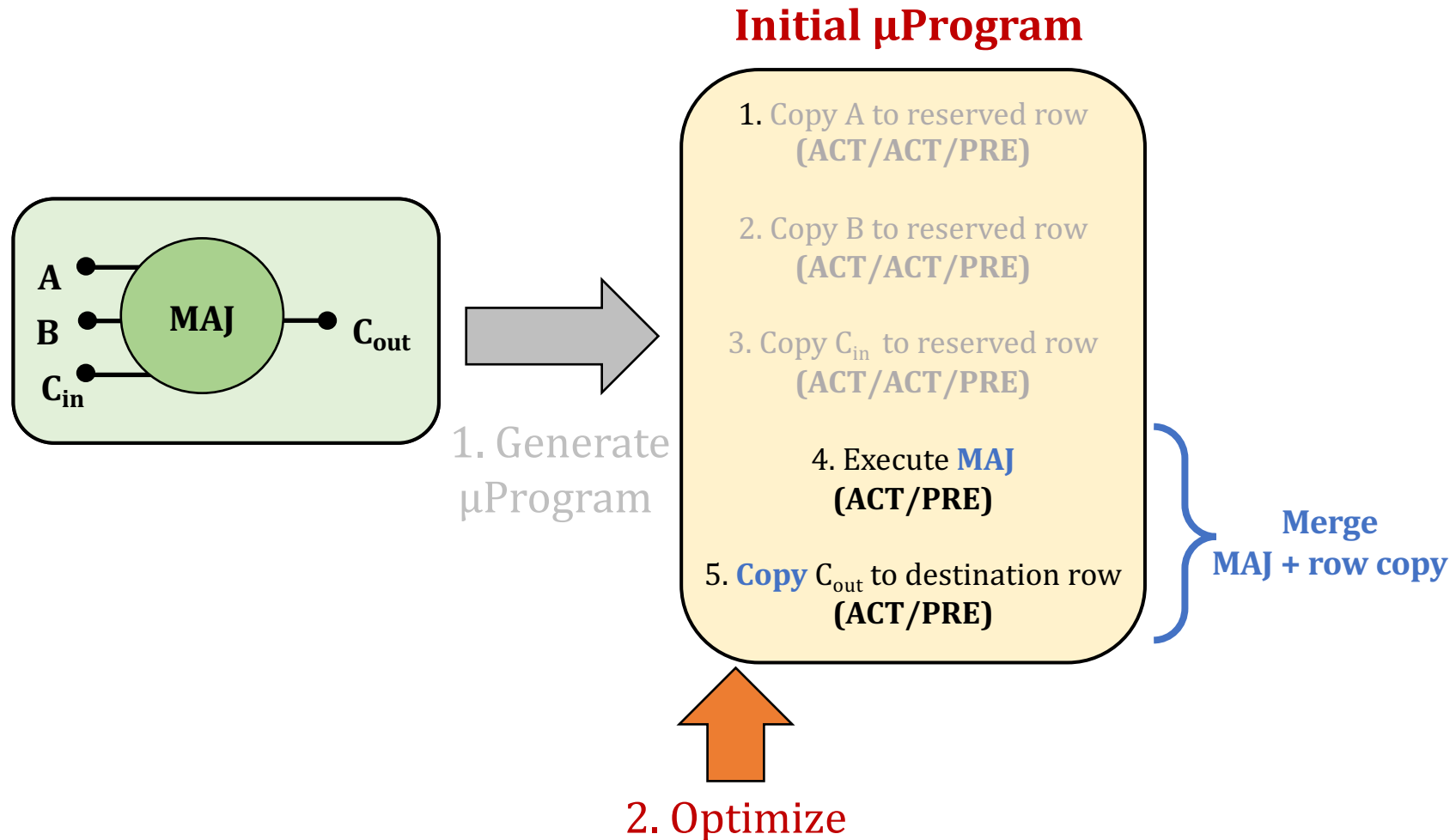
1. Copy A to reserved row  
(ACT/ACT/PRE)
2. Copy B to reserved row  
(ACT/ACT/PRE)
3. Copy C<sub>in</sub> to reserved row  
(ACT/ACT/PRE)
4. Execute MAJ  
(ACT/PRE)
5. Copy C<sub>out</sub> to destination row  
(ACT/PRE)

2. Optimize

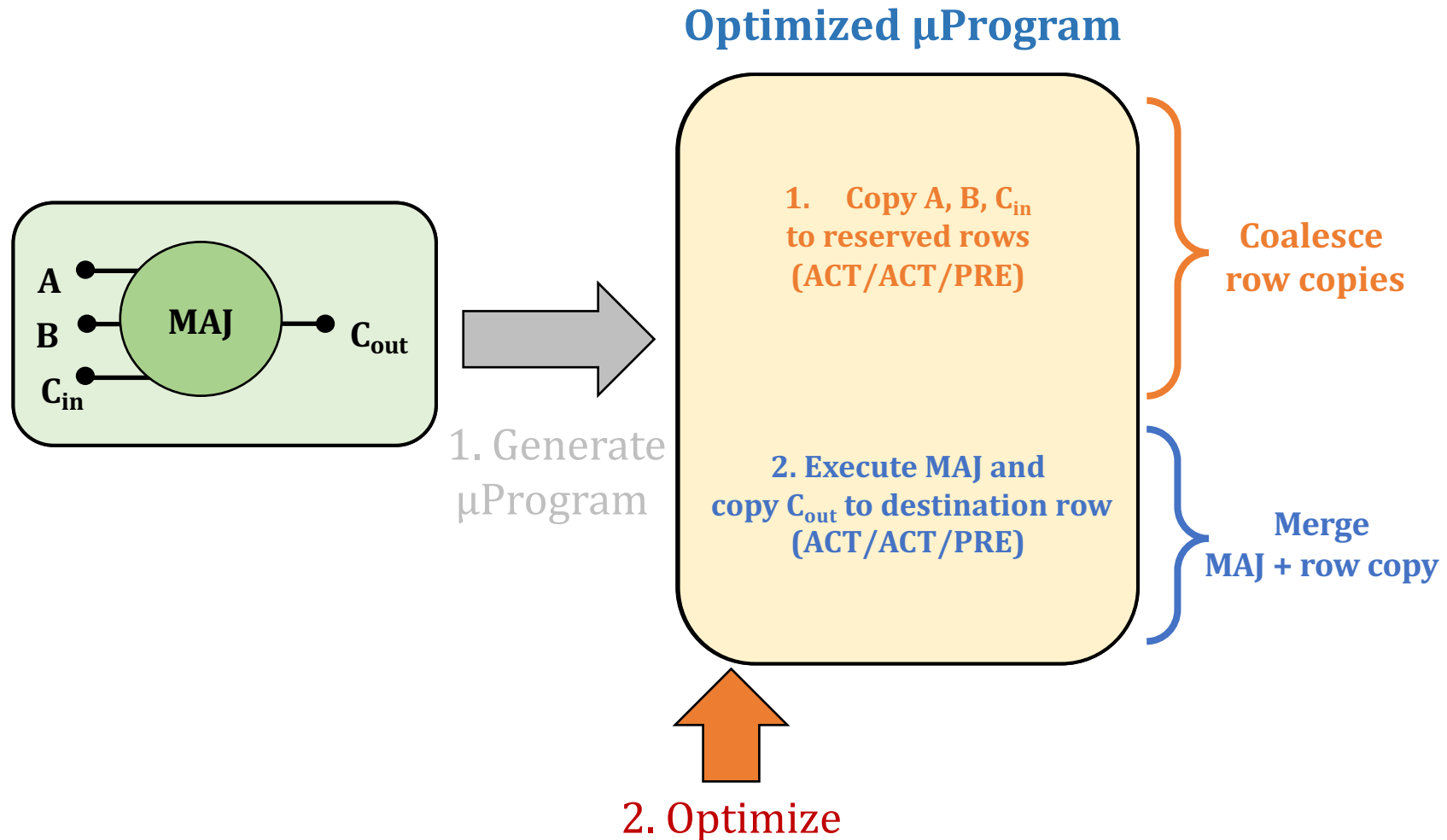
# Task 2: Optimize the $\mu$ Program



# Task 2: Optimize the $\mu$ Program

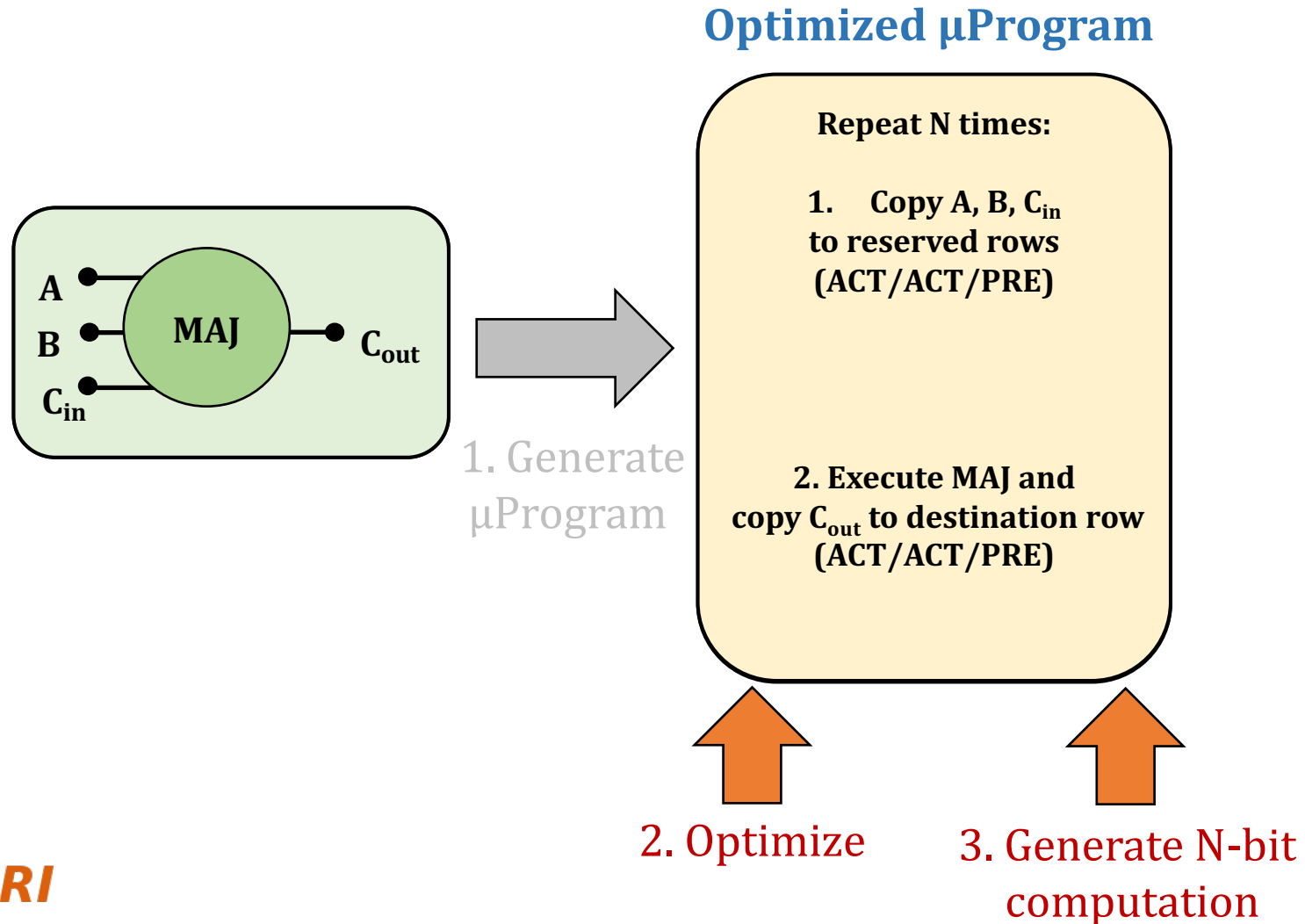


# Task 2: Optimize the $\mu$ Program



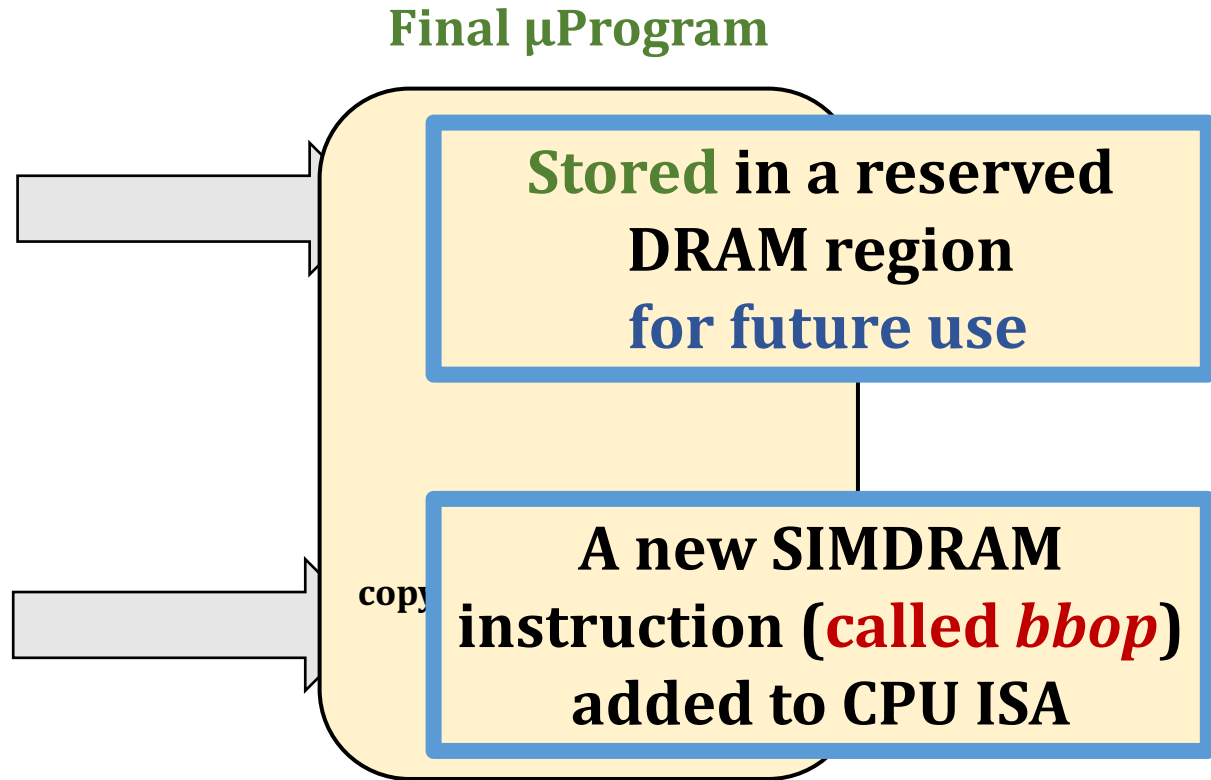
# Task 2: Generate N-bit Computation

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



# Task 2: Generate $\mu$ Program

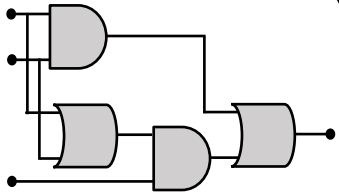
- **Final  $\mu$ Program** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion



# SIMDRAM Framework: Step 3

## User Input

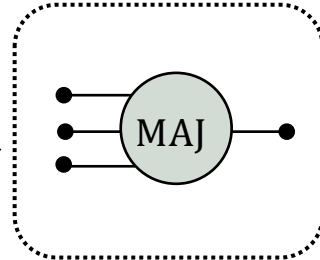
*Desired operation*



*AND/OR/NOT logic*



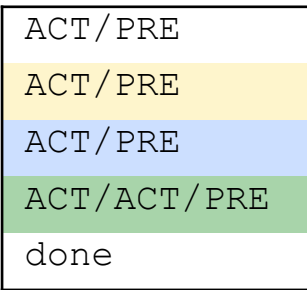
*Step 1: Generate  
MAJ logic*



*MAJ/NOT logic*



*Step 2: Generate  
sequence of  
DRAM commands*



*μProgram*



## SIMDRAM Output

*New SIMD RAM μProgram*

*μProgram*

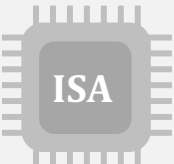
*μProgram*



*Main memory*

*bbop\_new*

*New SIMD RAM  
instruction*



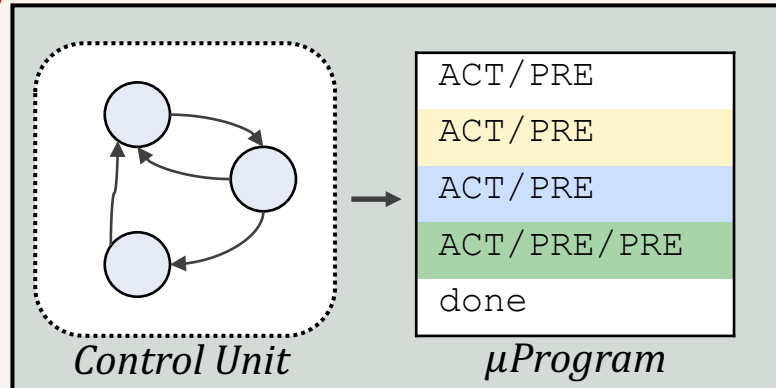
## User Input

*SIMDRAM-enabled application*

```
foo () {  
  bbop_new  
}
```



*Step 3: Execution according to μProgram*



*Control Unit*

*μProgram*

*Memory Controller*



## SIMDRAM Output

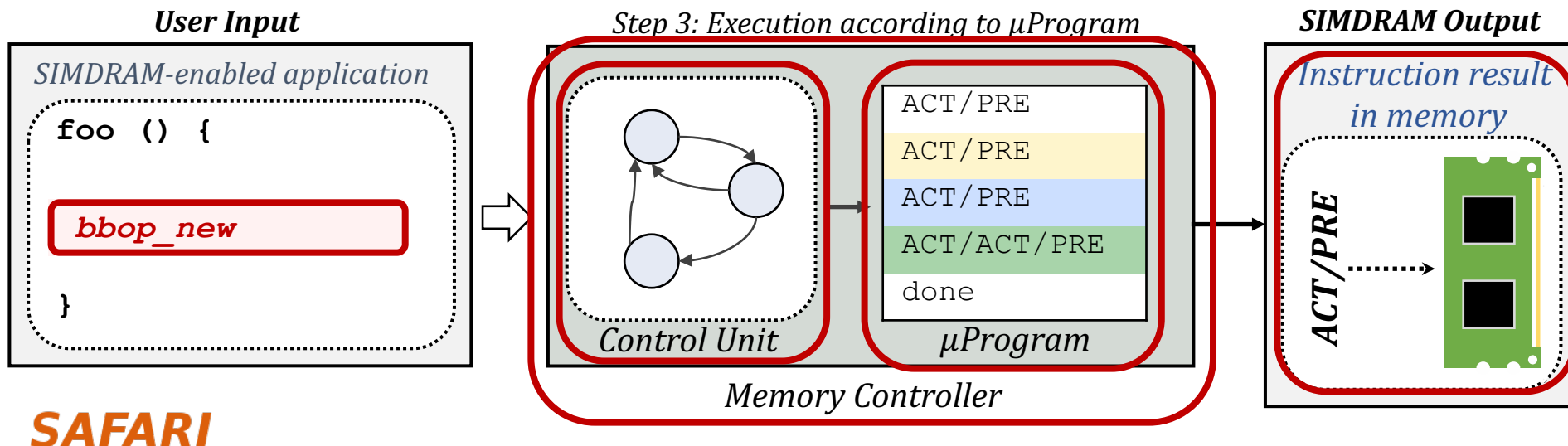
*Instruction result  
in memory*

*ACT/PRE*



# Step 3: $\mu$ Program Execution

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



# Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

6. Conclusion

# 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

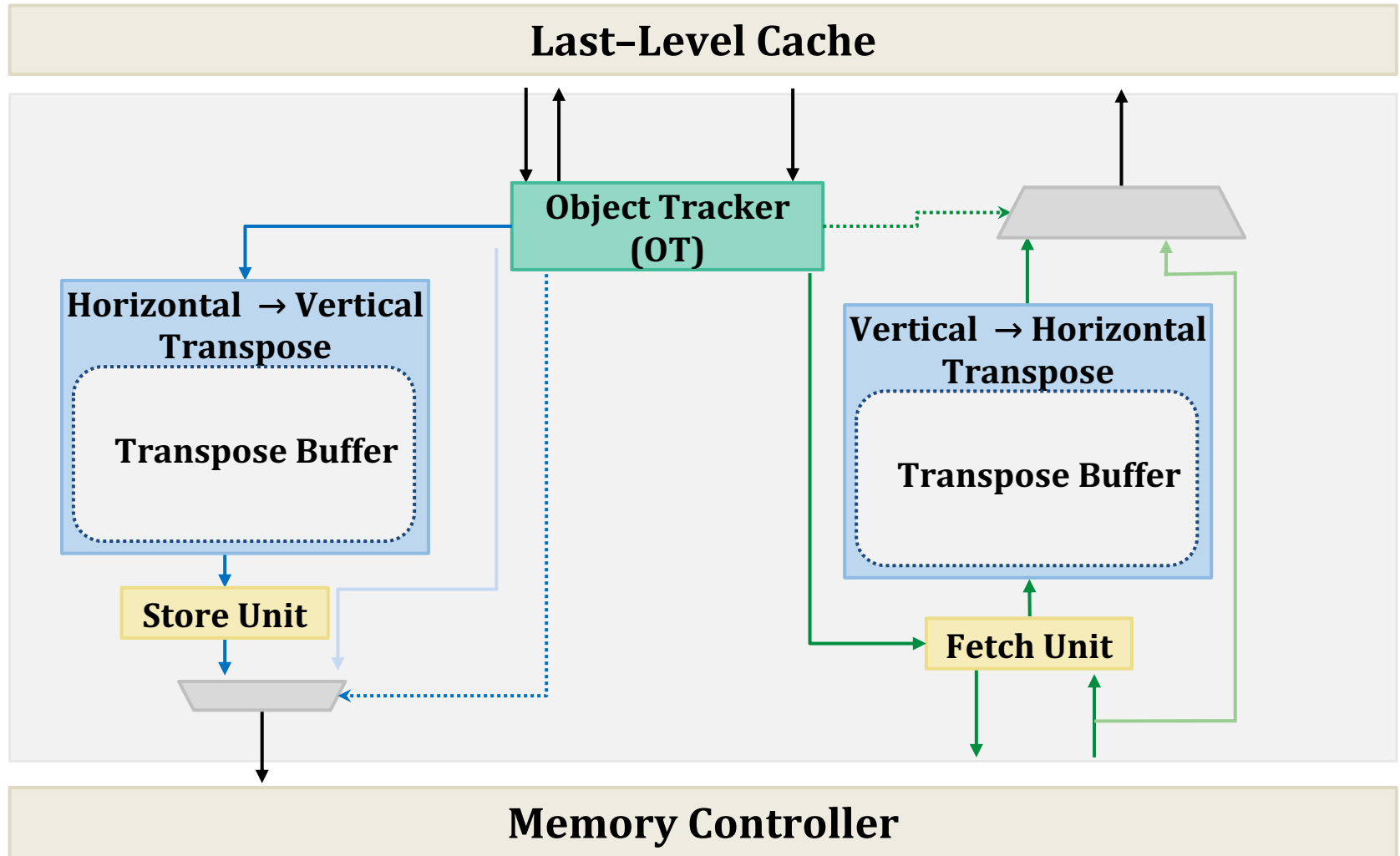
- SIMD RAM operates on **vertically-laid-out** data
- Other system components expect data to be laid out **horizontally**



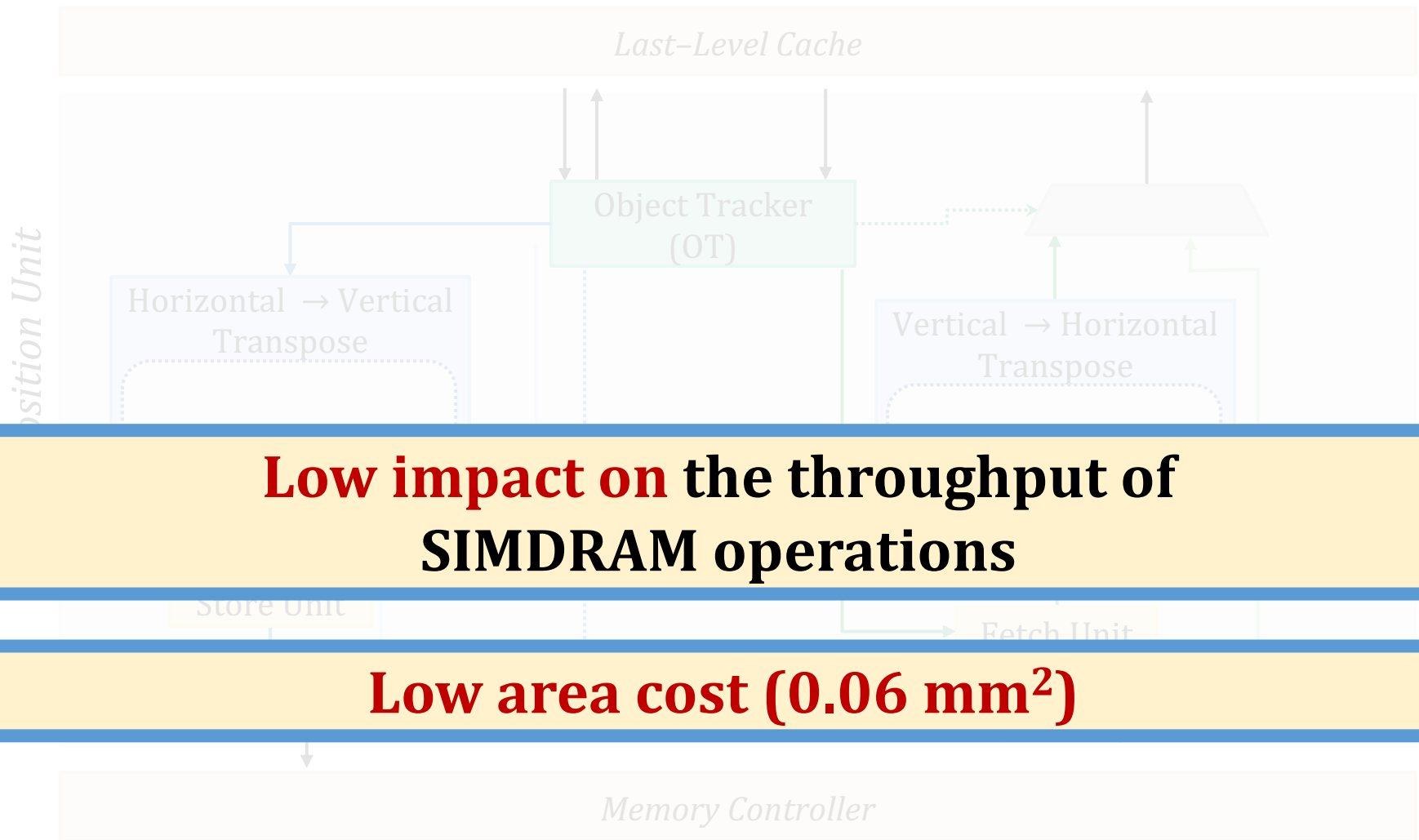
**Challenging** to share data between SIMD RAM and CPU

# Transposition Unit

Transposition Unit



# Efficiently Transposing Data



# 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

# Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
------	------------

# Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	<code>bbop_trsp_init address, size, n</code>

# Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	<code>bbop_trsp_init address, size, n</code>
1-Input Operation	<code>bbop_op dst, src, size, n</code>

---

# Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	<code>bbop_trsp_init address, size, n</code>
1-Input Operation	<code>bbop_op dst, src, size, n</code>
2-Input Operation	<code>bbop_op dst, src_1, src_2, size, n</code>

# Programming Interface

- Four new SIMD RAM ISA extensions

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

# Code Using SIMD RAM 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 SIMD RAM 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 SIMD RAM 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 SIMD RAM 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 SIMD RAM 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 SIMD RAM 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 SIMD RAM 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

## **SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM**

\*Nastaran Hajinazar<sup>1,2</sup>

Nika Mansouri Ghiasi<sup>1</sup>

<sup>1</sup>ETH Zürich

\*Geraldo F. Oliveira<sup>1</sup>

Minesh Patel<sup>1</sup>

Juan Gómez-Luna<sup>1</sup>

<sup>2</sup>Simon Fraser University

Sven Gregorio<sup>1</sup>

Mohammed Alser<sup>1</sup>

Onur Mutlu<sup>1</sup>

<sup>3</sup>University of Illinois at Urbana-Champaign

João Dinis Ferreira<sup>1</sup>

Saugata Ghose<sup>3</sup>

coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework

# Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

6. Conclusion

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

# Methodology: Workloads

## Evaluated:

- 16 complex in-DRAM operations:

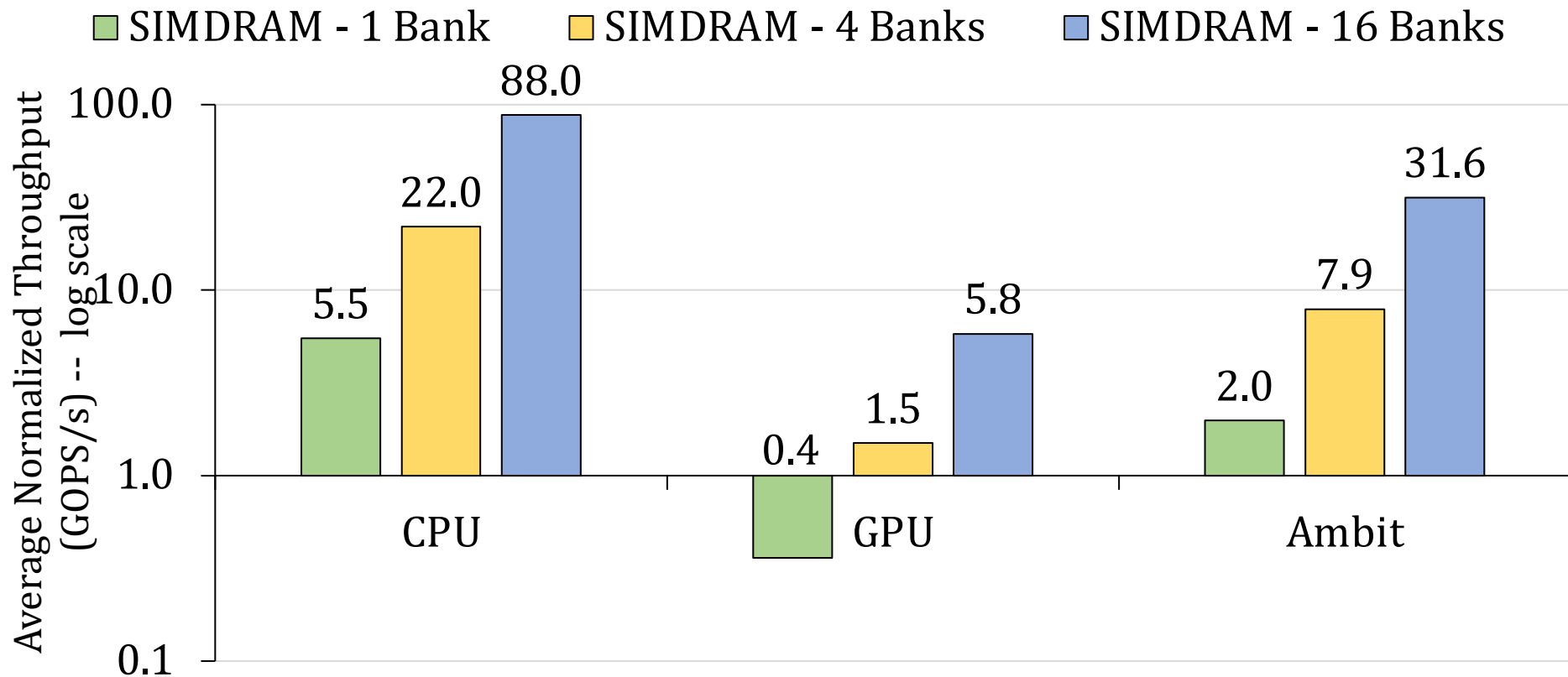
- Absolute
- Addition/Subtraction
- BitCount
- Equality/ Greater/Greater Equal
- Predication
- ReLU
- AND-/OR-/XOR-Reduction
- Division/Multiplication

- 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

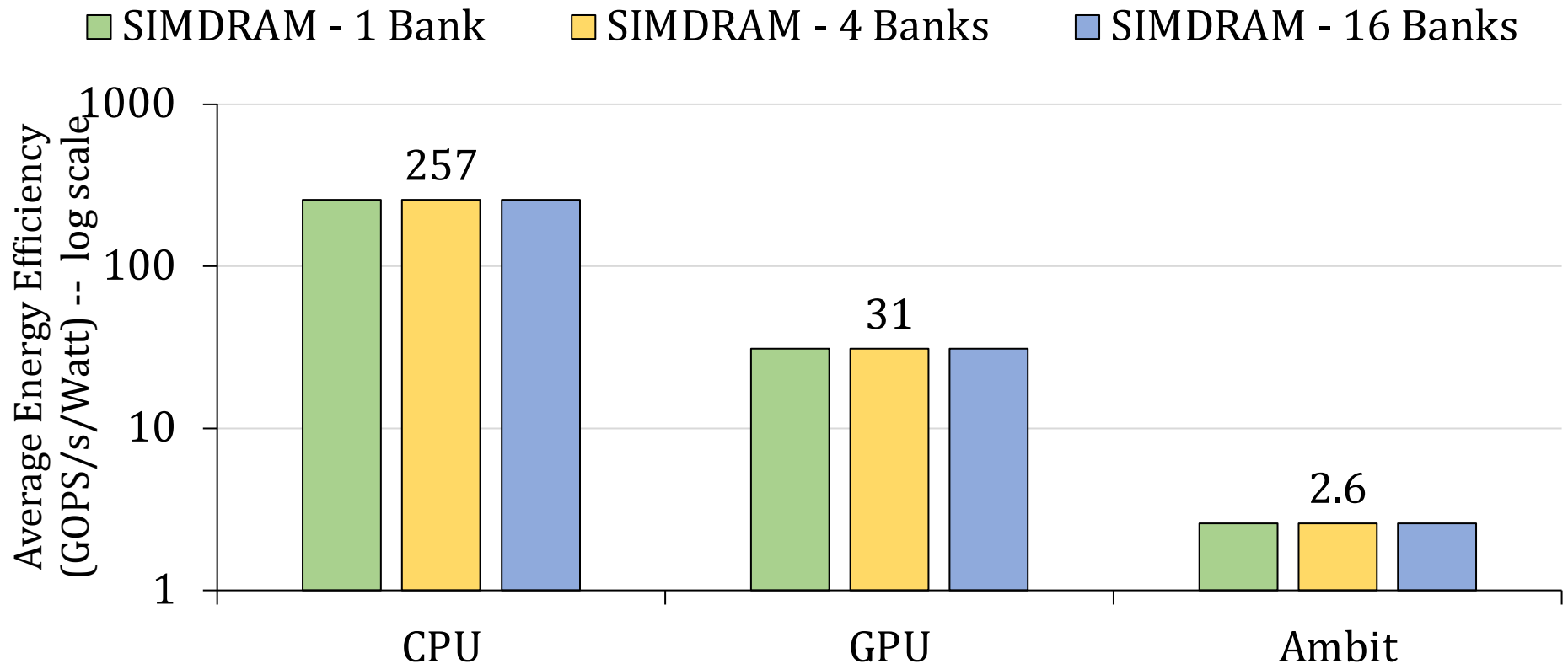
Average normalized throughput across all 16 SIMD RAM operations



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

# Energy Analysis

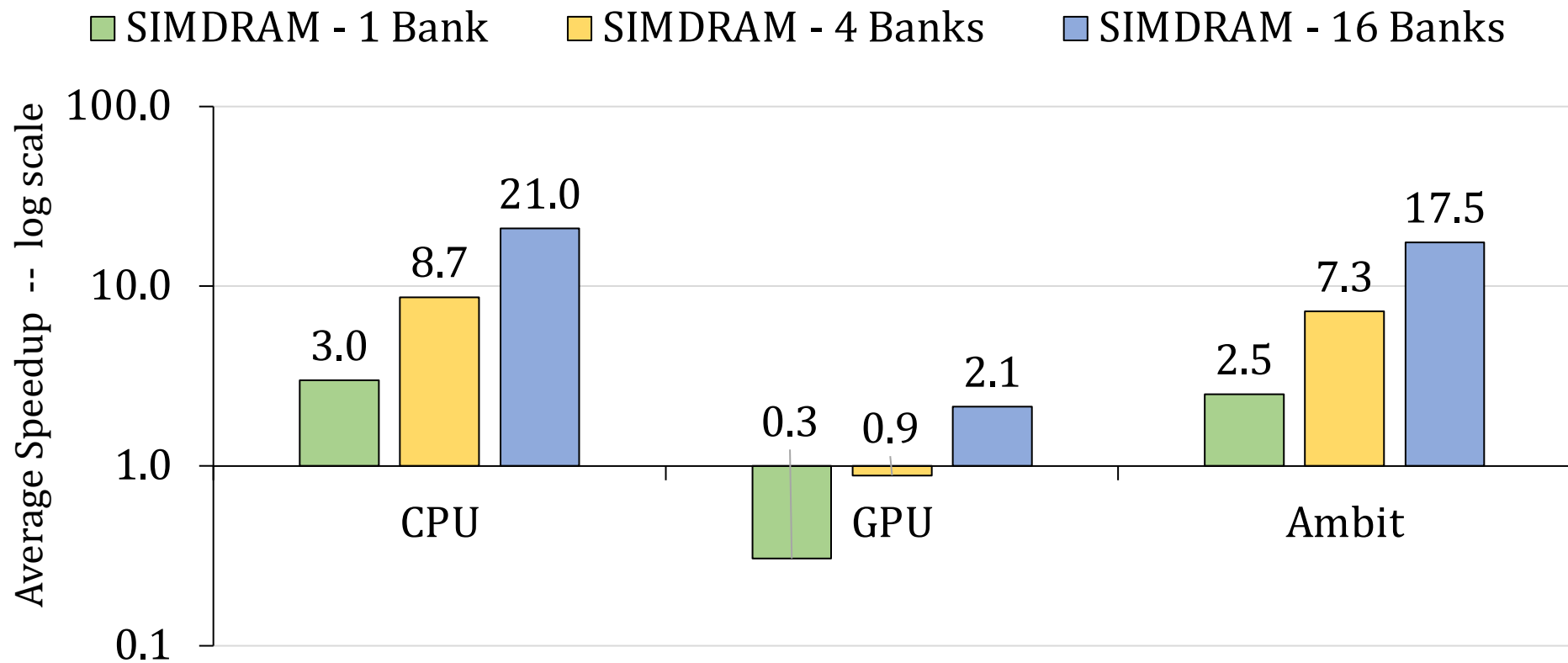
Average normalized energy efficiency across all 16 SIMD RAM operations



**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

# More in the Paper

- Evaluation:

- Reliability

## **SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM**

\*Nastaran Hajinazar<sup>1,2</sup>

Nika Mansouri Ghiasi<sup>1</sup>

\*Geraldo F. Oliveira<sup>1</sup>

Minesh Patel<sup>1</sup>

Juan Gómez-Luna<sup>1</sup>

Sven Gregorio<sup>1</sup>

Mohammed Alser<sup>1</sup>

Onur Mutlu<sup>1</sup>

João Dinis Ferreira<sup>1</sup>

Saugata Ghose<sup>3</sup>

<sup>1</sup>ETH Zürich

<sup>2</sup>Simon Fraser University

<sup>3</sup>University of Illinois at Urbana–Champaign

- Comparison to in-cache computing

# Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate  
Framework

4. System Integration

5. Evaluation

6. Conclusion

# 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

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

*P&S Processing-in-Memory*  
Spring 2022  
2 June 2022



**Nastaran Hajinazar\***

**Geraldo F. Oliveira\***

Sven Gregorio

Joao Ferreira

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**SAFARI**



SIMON FRASER  
UNIVERSITY

**ETH** zürich



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN