ETH zürich

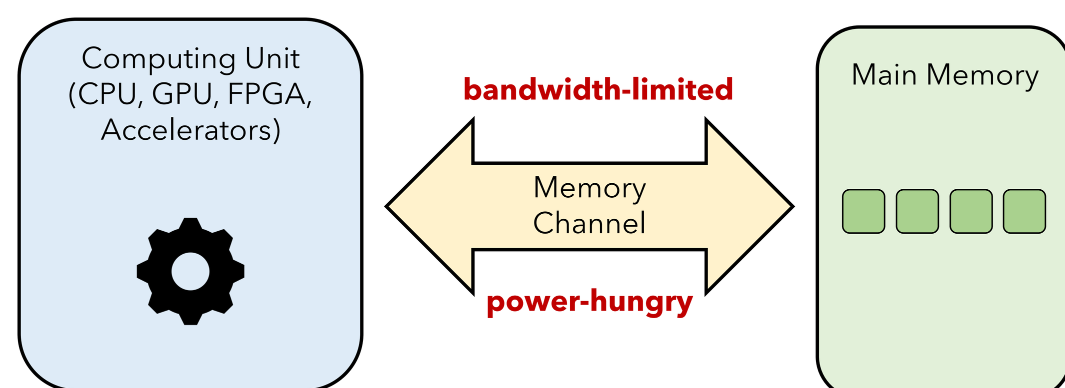
SAFARI

PUMA: Efficient and Low-Cost Memory Allocation and Alignment Support for Processing-Using-Memory Architectures

Geraldo F. Oliveira (*Ph.D. Candidate*) Onur Mutlu (*Ph.D. Advisor*)

Data Movement Bottleneck

Data movement is a major bottleneck in modern computer architectures

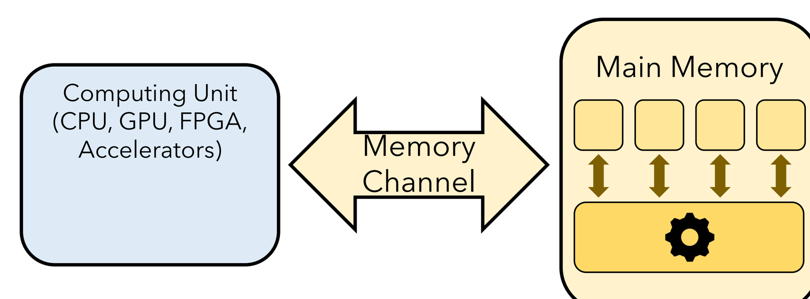


Over 60% of the total system energy is spent on data movement

Processing-in-Memory: Overview & Landscape

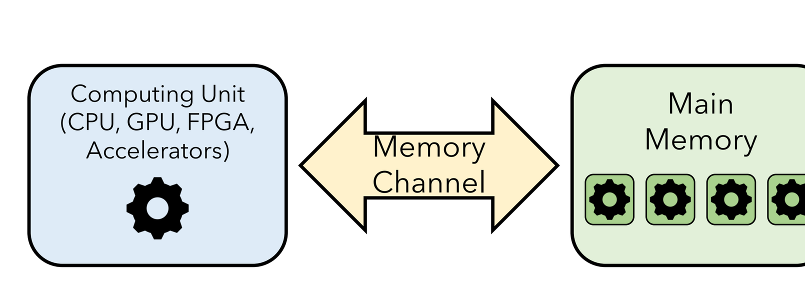
Processing-in-Memory: move computation to where the data resides

1. Processing-near-Memory



Processing-*near*-Memory leverages additional logic for computation

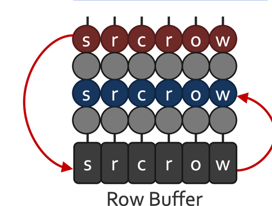
2. Processing-using-Memory



Processing-using-Memory uses memory's operational principles for computation

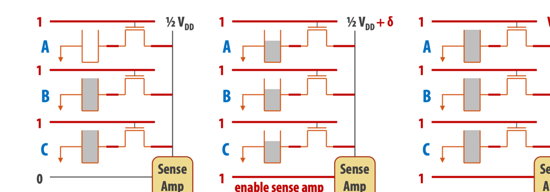
Processing-using-DRAM (PUD) alleviates data movement bottlenecks

RowClone



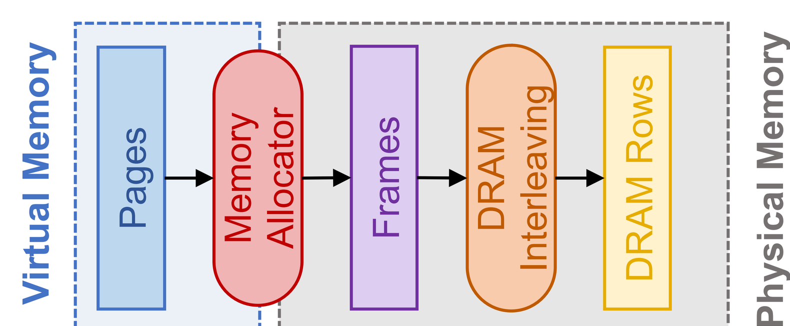
Ambit

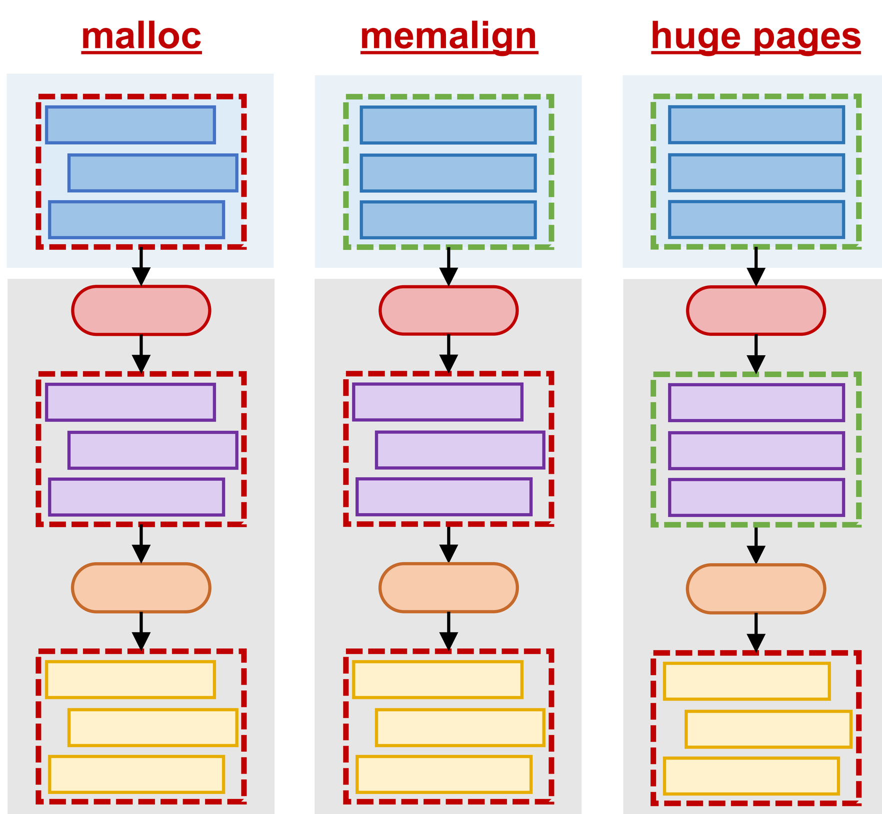
Ambit AND/OR – Triple-Row Activation (TRA)



The Memory Allocation Issue

Traditional memory allocators cannot take full advantage of PUD techniques since they do *not* satisfy the memory allocation requirements of PUD substrates



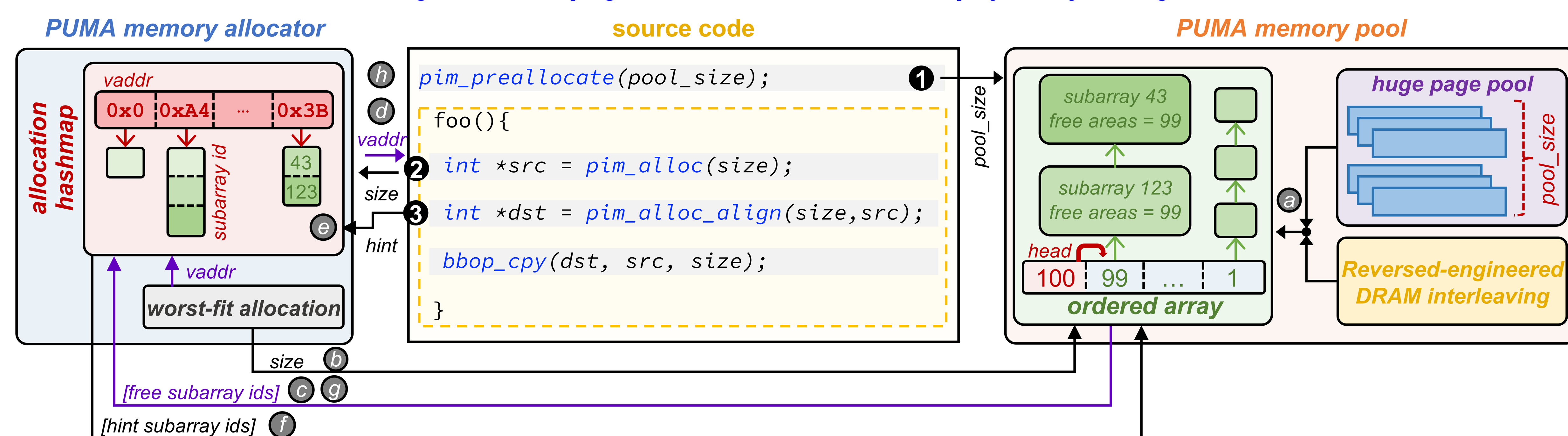


Our Goal

To provide a flexible memory allocation mechanism that (*i*) allows programmers to have control over physical memory allocation and (*ii*) enables PUD execution from the operating system viewpoint.

PUMA: Efficient Memory Allocation & Alignment

Key Idea:
use the internal DRAM mapping information, together with huge pages to *split huge pages into finer-grained allocation units* that are (*i*) aligned to the page address and size and (*ii*) *physically contiguous*.



PUMA: Overview & Execution Steps

1 Pre-Allocation of Operands

- The first step in PUMA is to indicate the **number of huge pages** that are available for PUD allocations using the **pim_preallocate API**.
- We left it up to the user to **provide the number of huge pages used for PUD operations** since they are scarce in the system.

2 First Operand Allocation

- PUMA uses the **worst-fit allocation scheme** to manage the allocation of memory regions in the huge page pool.
- For **the first PUD memory allocation**, using the **pim_alloc API**, PUMA scans the ordered array of free memory regions to select the subarray with the **largest amount of memory regions available**.

3 Aligned Operand Allocation

- For **aligned operand allocation,** we implement the **pim_alloc_align API**, which takes a **hint pointer** as input.
- For each memory region, PUMA **identifies its source subarray address** and tries to allocate another memory region **at the same subarray** for the new allocation.

Implementation Considerations

- The DRAM interleaving scheme can be obtained by **reverse engineering the bit locations** of memory addresses.
- The **memory controller** informs PUMA of the DRAM interleaving scheme via an **open firmware device tree**.
- The **pim_alloc_align API fails** to allocate data in case the hint address is **not found**.

Evaluation

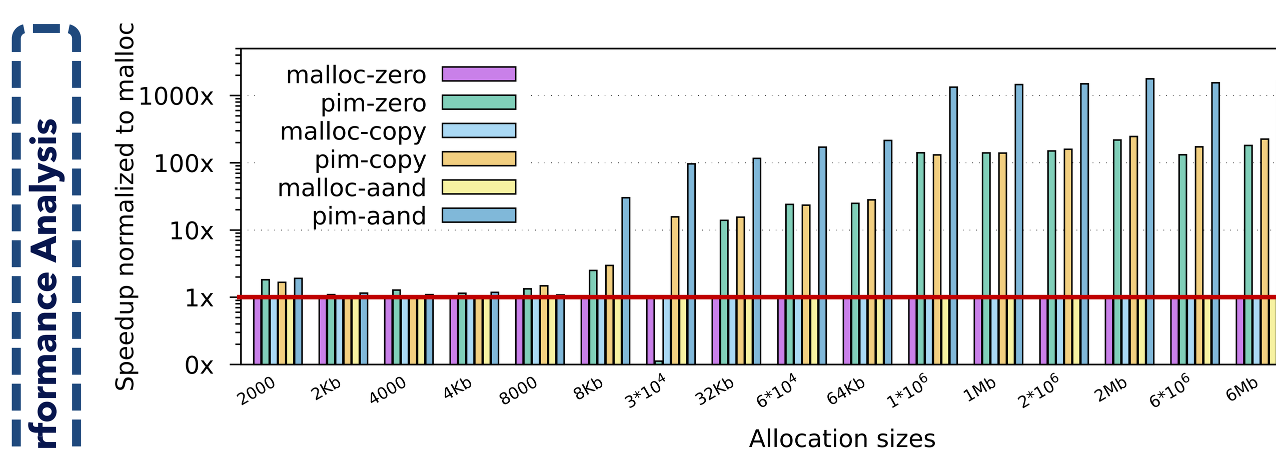
Experimental Setup

- **QEMU-based implementation**
  - **Baseline Host CPU:** We emulate a RISC-V machine running Fedora 33 with v5.9.0 Linux Kernel for **malloc()**.
  - **PIM:** We emulate the implementation of a PUD system capable of executing row copy operations (as in RowClone) and Boolean AND/OR/NOT operations (as in Ambit) for **PUMA**.

Workloads

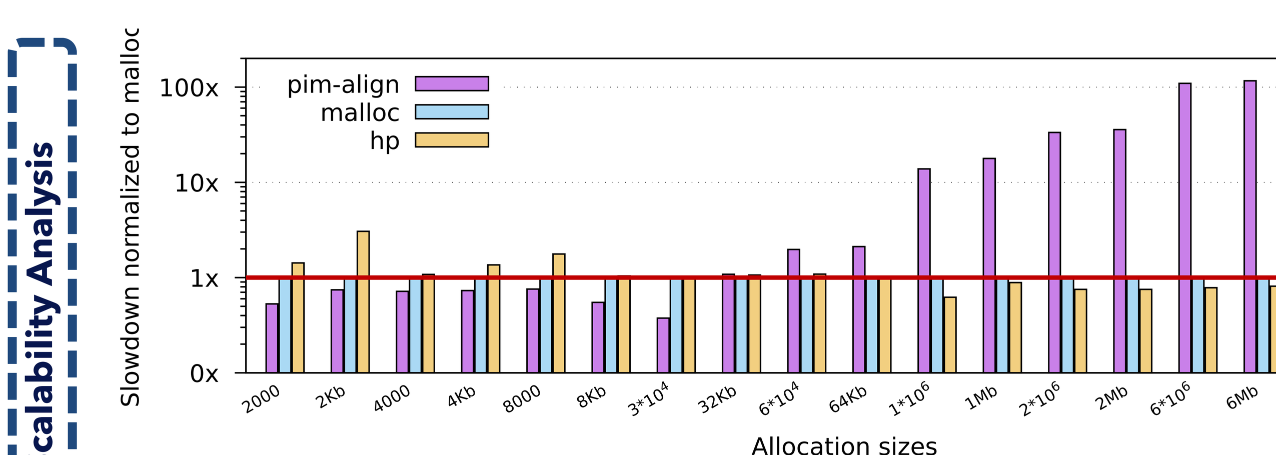
- **3 micro-benchmarks**
  - initialize an array with zeros **(*-zero)**
  - copy data from one array to another **(*-copy)**
  - perform vector bitwise AND operations C[i] = A[i] AND B[i] using Ambit **(*-aand)**
- **Allocation sizes**
  - For each micro-benchmark, we vary the allocation sizes from 2000 bits to 6 Mb.

Performance Analysis



PUMA significantly outperforms the baseline memory allocators for all micro-benchmarks and allocation sizes

Scalability Analysis



PUMA can lead to performance penalties for large allocation sizes

Conclusion

Background: Processing-in-Memory (PiM) alleviates the performance and energy bottlenecks caused by data

- *Processing-using-DRAM* (PUD) architectures can accelerate bulk data copy and bitwise operations at low cost

Problem: PUD architectures impose a restrictive data layout and alignment for computation

- Operands must **reside in the same DRAM subarray** and be **aligned to the boundaries** of a DRAM row

Goal: To provide a **flexible memory allocation mechanism** that allows programmers to have control over **physical memory allocation**, enabling PUD execution

PUMA: *A software-only new **lazy data allocation routine** for PUD architectures that*

- uses a pool of **huge pages** and **reverse-engineered DRAM mapping information** to allocate PUD operands correctly;
- exposes three new allocation APIs to the user, and **automatically manages data alignment** within DRAM

Key Results: Our evaluation shows that PUMA

- ***outperforms*** malloc-based memory allocators
- ***scales* well** depending on the data allocation sizes