# *The Virtual Block Interface:*
## *A Flexible Alternative to the Conventional Virtual Memory Framework*

*ISCA, 2020*

Nastaran Hajinazar  Pratyush Patel  Minesh Patel
Konstantinos Kanellopoulos  Saugata Ghose
Rachata Ausavarungnirun  Geraldo F. Oliveira  Jonathan Appavoo
Vivek Seshadri  Onur Mutlu

**Presented by Stefan Scholbe**

Most slides by Nastaran Hajinazar

SAFARI  ETH Zürich  SFU Simon Fraser University  University of Washington
Carnegie Mellon  KMUTNB  Microsoft  Boston University

# Executive Summary

- **Motivation**: Modern computing systems continue to **diversify** with respect to system architecture, memory technologies, and applications' memory needs
- **Problem**: Continually adapting the conventional virtual memory framework to each possible system configuration is challenging
  - Results in performance loss or requires non-trivial workarounds
- **Goal**: Design an alternative virtual memory framework that
  (1) Efficiently supports a wide variety of new system configurations
  (2) Provides the key features and eliminates the key inefficiencies of the conventional virtual memory framework
- **Virtual Block Interface (VBI)**:
  Delegates memory management to dedicated hardware in the memory controller
  - Efficiently adapts to diverse system configurations
  - Reduces overheads and complexities associated with conventional virtual memory
  - Enables many optimizations (e.g., low-overhead page walks in virtual machines, virtual caches)
- **Evaluation**: Two example use cases
  1. **VBI** significantly improves performance for both native execution (**2.4x**) and virtual machine environments (**4.3x**)
  2. **VBI** significantly improves heterogeneous memory architecture effectiveness

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Outline

**Motivation**

**VBI: Virtual Block Interface**

- Key Idea & Guiding Principles
- Design Overview
- Optimizations Enabled by VBI

**Methodology**

**Results**

**Summary**

**Review**

**Discussion**

# Virtual Memory

**Application**

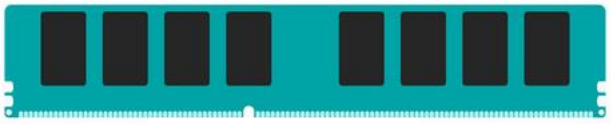
---

## Virtual Memory

**managed by the operating system**

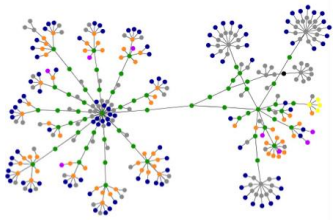
---

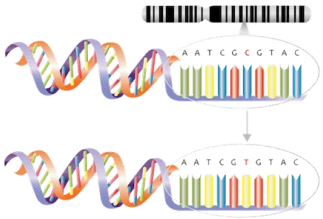
**Hardware**



# Computing Systems Are Diversifying

## Application





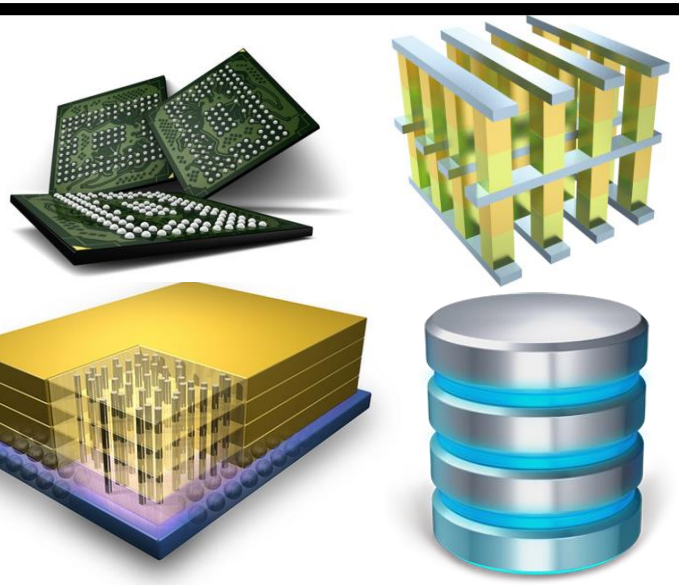


**Cannot adapt efficiently**

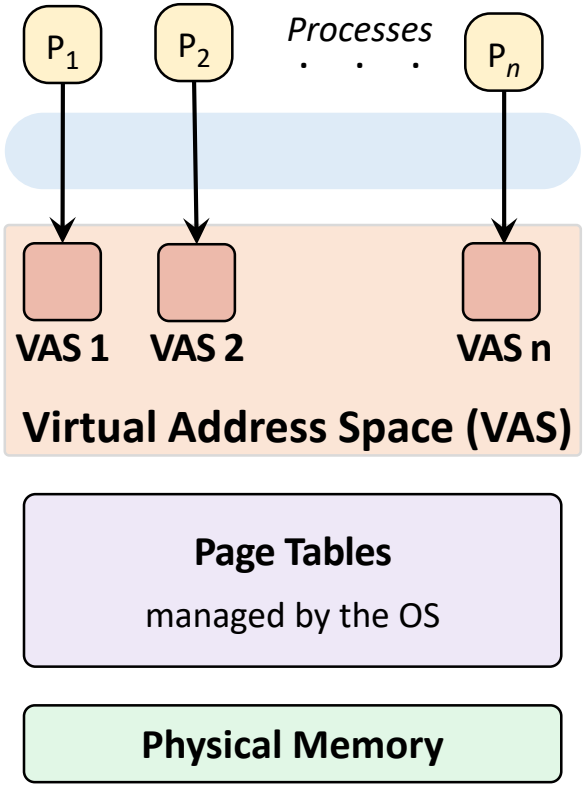
## Virtual Memory

**managed by the operating system**

## Hardware



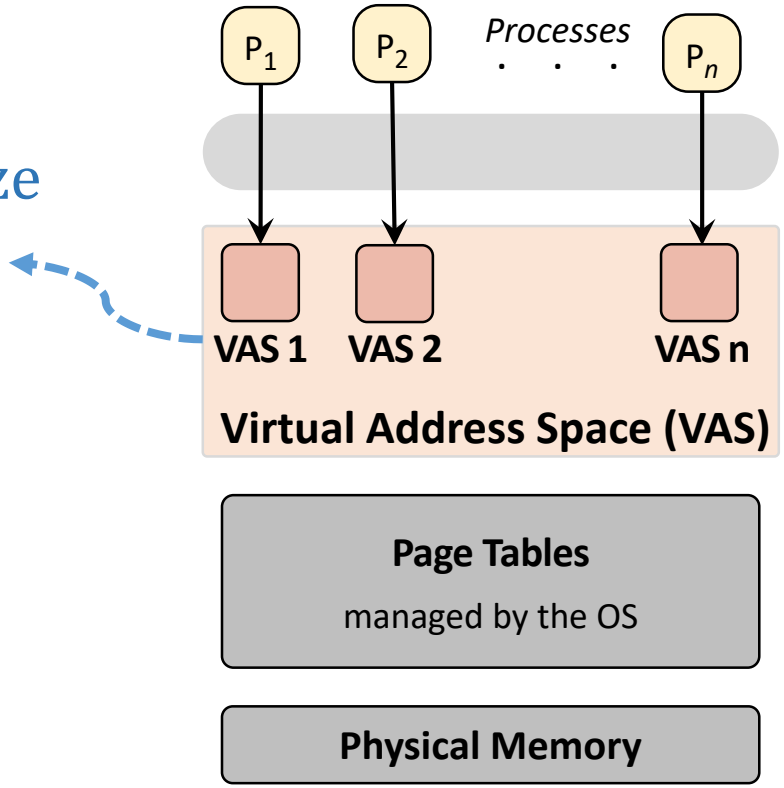
# Conventional Virtual Memory Framework



# Conventional Virtual Memory Framework

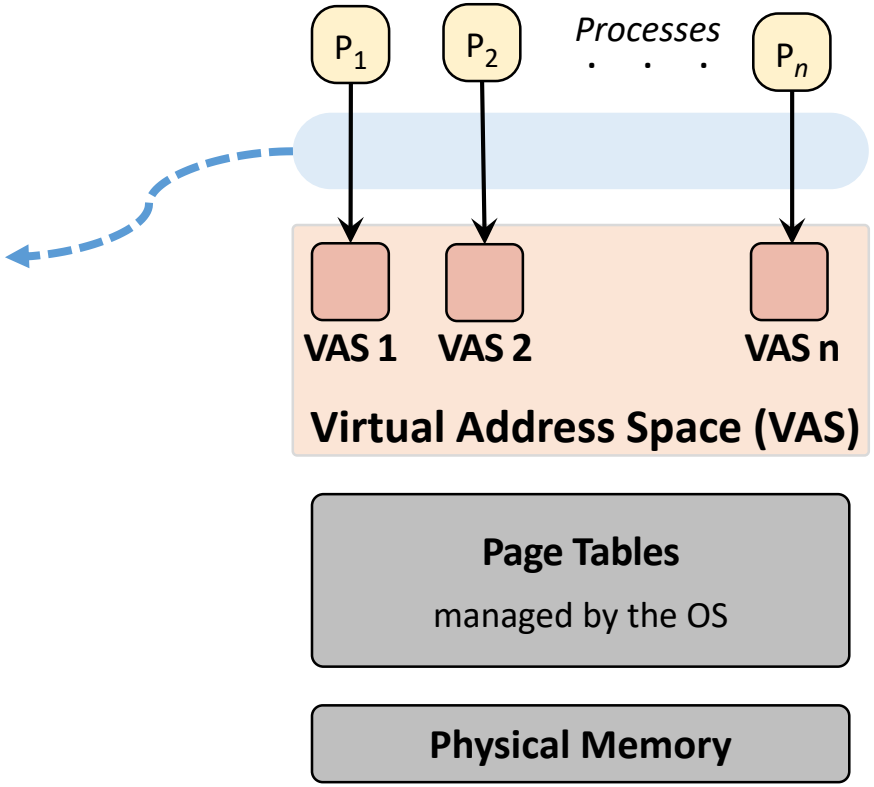
each process is mapped to a fixed-size virtual address space

*e.g., 256 TB in Intel x86-64*



# Conventional Virtual Memory Framework

one-to-one mapping
managed by the OS

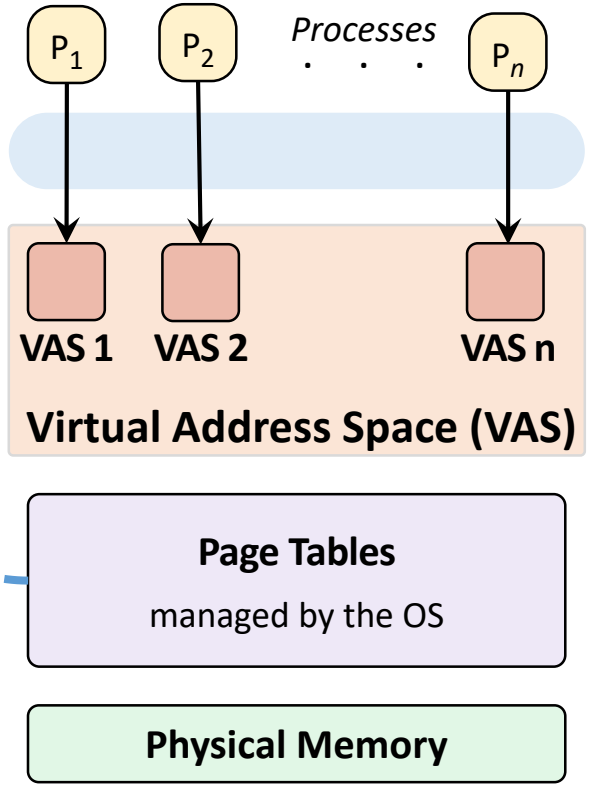


# Conventional Virtual Memory Framework

per-process page tables to
map each VAS to physical memory

*managed by the OS*

*read by hardware*

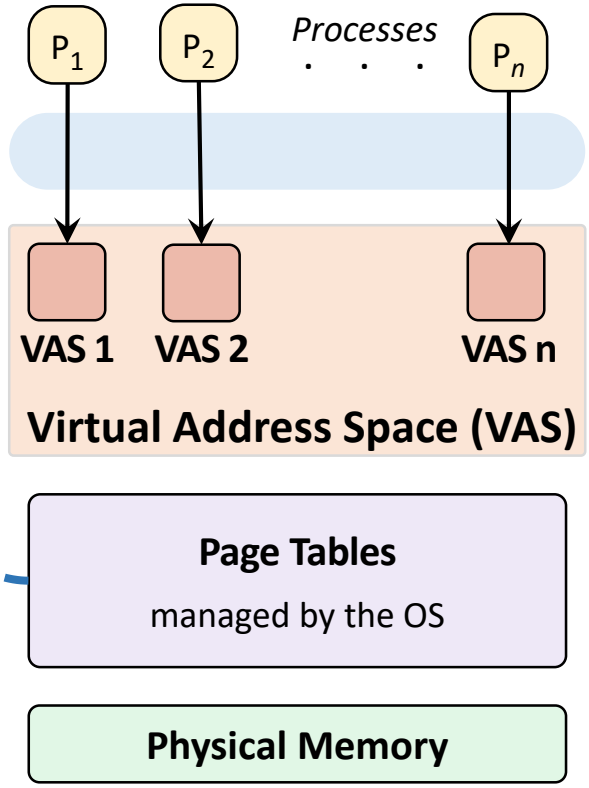


# Challenges

- **Three examples** of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
  - Requiring a rigid page table structure
  - High address translation overhead in virtual machines
  - Inefficient heterogeneous memory management

# Challenge 1: Rigid Page Table Structures

- **Flexibly customized** page tables can reduce the address translation overhead
  - Customized to the application's memory behavior
    - e.g., larger granularities for more densely allocated memory regions



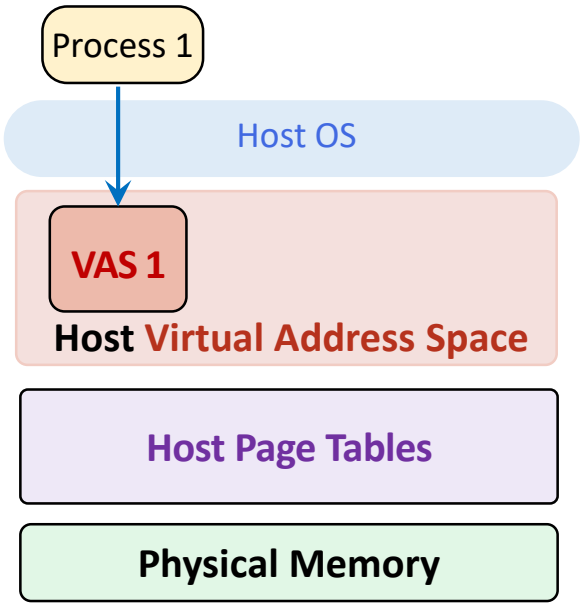

accessed by both **OS** and **hardware**

- **Con:**
  - Requires a rigid page table structure
    - e.g., fixed-granularity 4-level page table in Intel x86

# Challenges

- **Three examples** of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
  - Requiring a rigid page table structure
  - High address translation overhead in virtual machines
  - Inefficient heterogeneous memory management

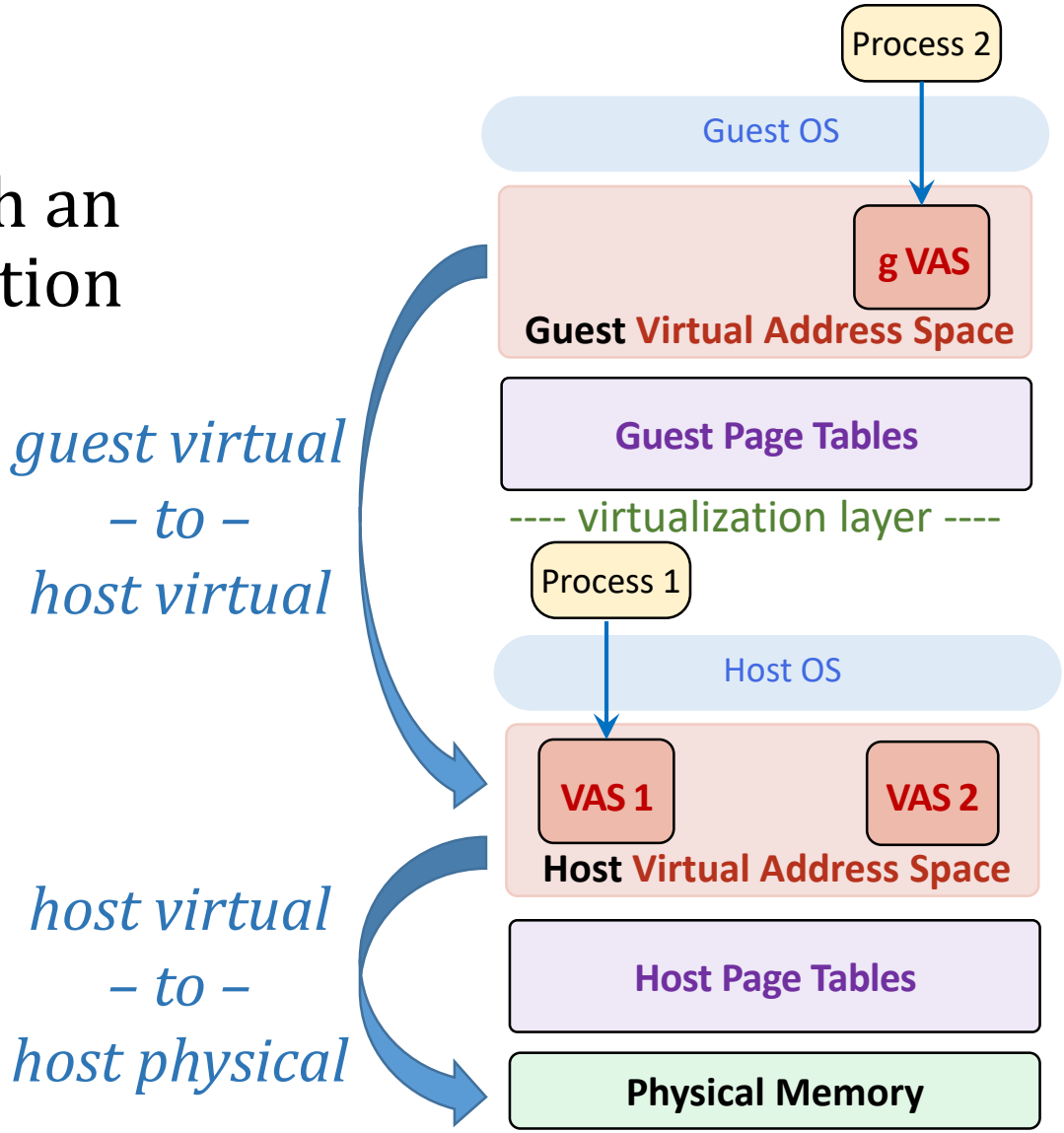
# Challenge 2: Overheads in Virtual Machines



# Challenge 2: Overheads in Virtual Machines

- In virtual machines, processes go through an extra level of indirection

- **Con:**
  - 2D page table walks

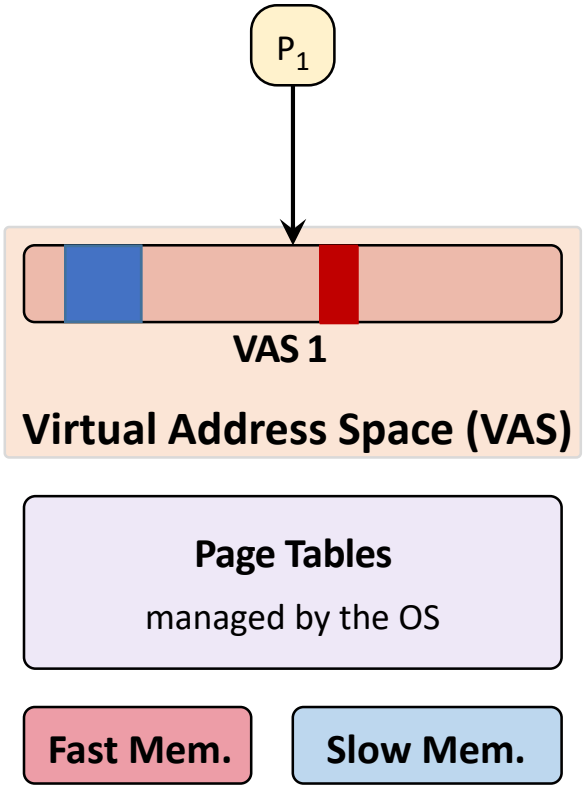


# Challenges

- **Three examples** of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
  - Requiring a rigid page table structure
  - High address translation overhead in virtual machines
  - Inefficient heterogeneous memory management

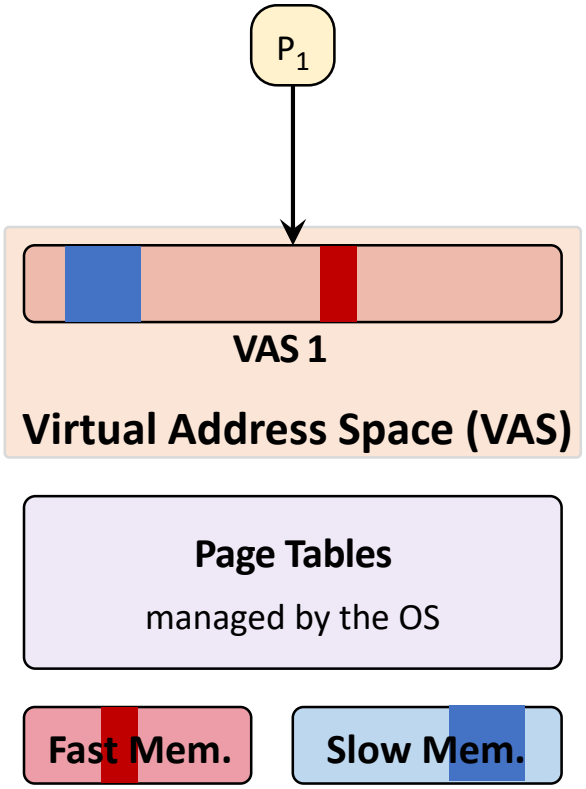
# Challenge 3: Managing Heterogeneous Memory

- Enhancing performance with **heterogenous memories** requires:
  - Data mapping



# Challenge 3: Managing Heterogeneous Memory

- Enhancing performance with **heterogenous memories** requires:
  - Data mapping
  - Data migration




- **Con:**
  - OS has low visibility into runtime memory behavior
    - Timely reaction to the changes is challenging

# Prior Works

- Optimizations that **alleviate the overheads** of the conventional virtual memory framework

## Shortcomings:

- Based on **specific** system or workload characteristics
  - Are applicable to only limited problems or applications
- Require **specialized** and **not necessarily compatible** changes to both the OS and hardware
  - Implementing all in a system is a daunting prospect

# Prior Works

- Optimizations that **alleviate the overheads** of the conventional virtual memory framework

**Shortcomings:**

- Based on **specific** system or workload characteristics
  - Are applicable to only limited problems or applications

**We need a holistic solution that efficiently supports increasingly diverse system configurations**

# Our Goal

**Design an alternative virtual memory framework that**

- Efficiently and flexibly supports increasingly diverse system configurations
- Provides the key features of conventional virtual memory framework while eliminating its key inefficiencies

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Virtual Block Interface (VBI)

**VBI** is an alternative virtual memory framework

**Key idea:**

**Delegate** physical memory management to dedicated hardware in the **memory controller**

# VBI: Guiding Principles

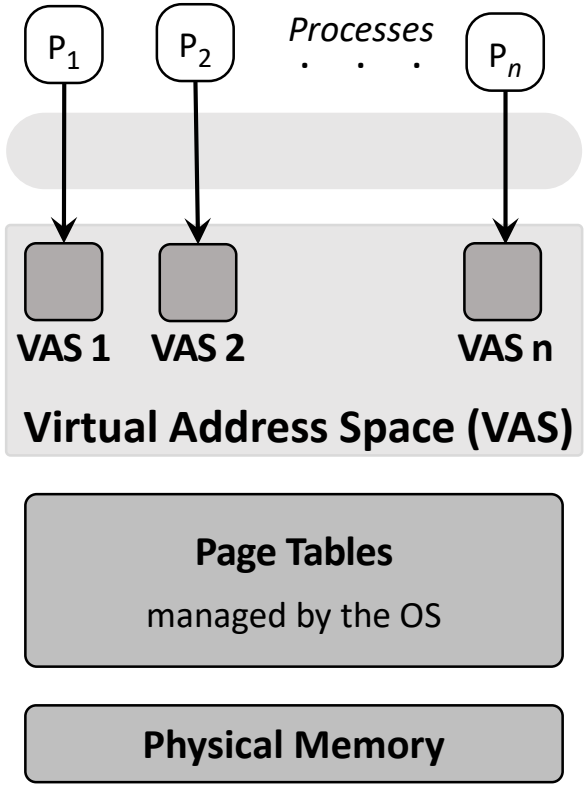
- **Size virtual address spaces appropriately for processes**
  - Mitigates translation overheads of unnecessarily large address spaces
- **Decouple address translation from access protection**
  - Defers address translation until necessary to access memory
  - Enables the flexibility of managing translation and protection using separate structures
- **Communicate data semantics to the hardware**
  - Enables intelligent resource management

**Addresses the rigidness and lack of information in current frameworks, to reduce large overheads**

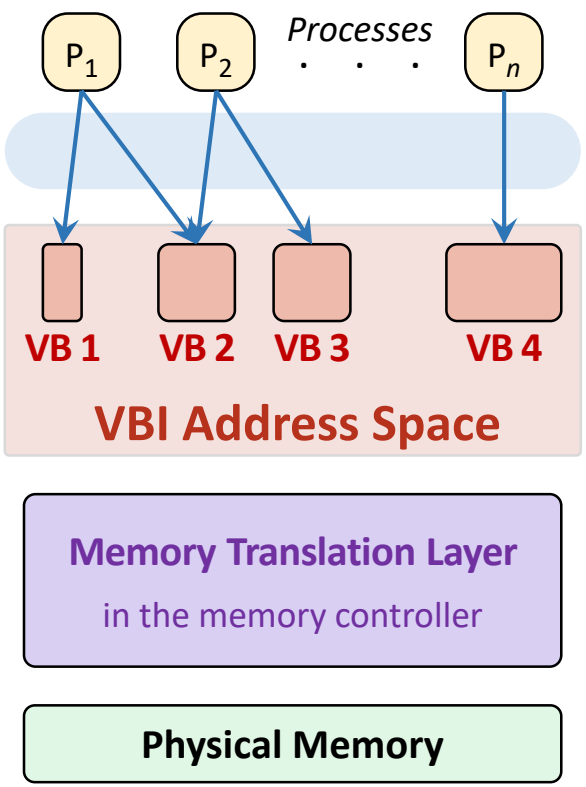
# Outline

- Motivation
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- Methodology
- Results
- Summary
- Review
- Discussion

# VBI: Overview



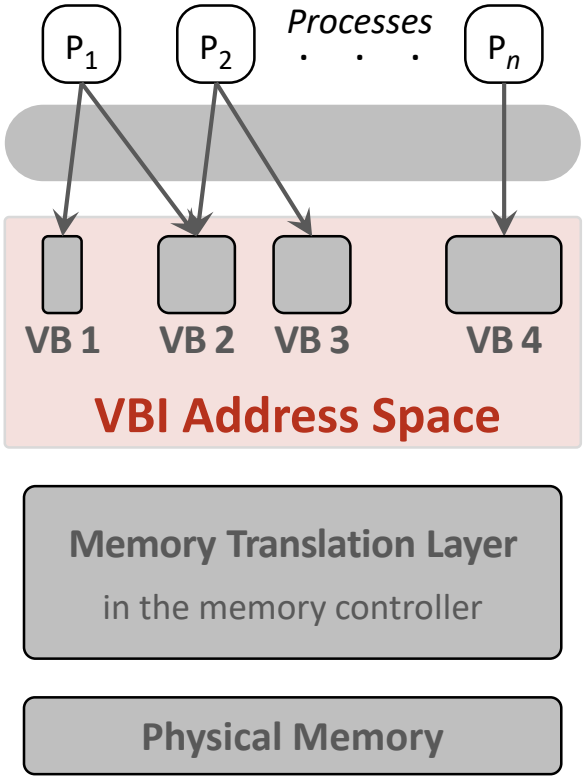

Conventional Virtual Memory




VBI

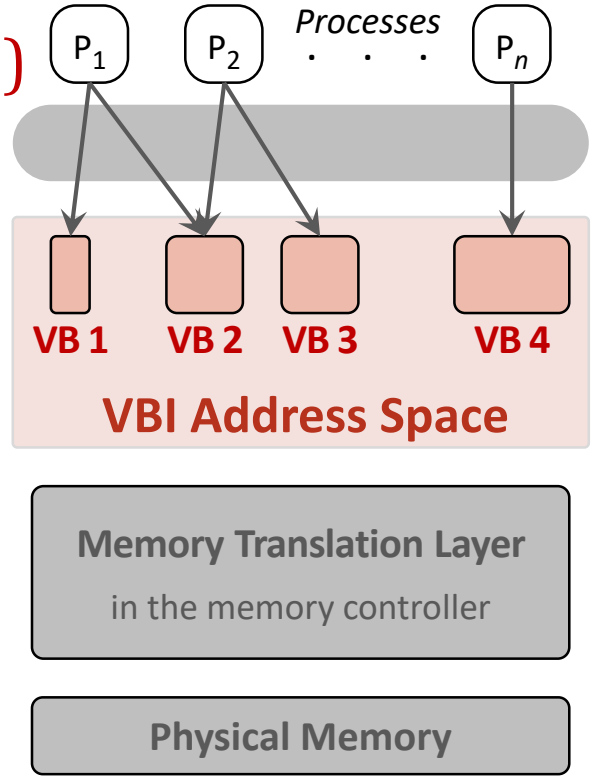
# Virtual Blocks

- **Globally-visible *VBI address space***



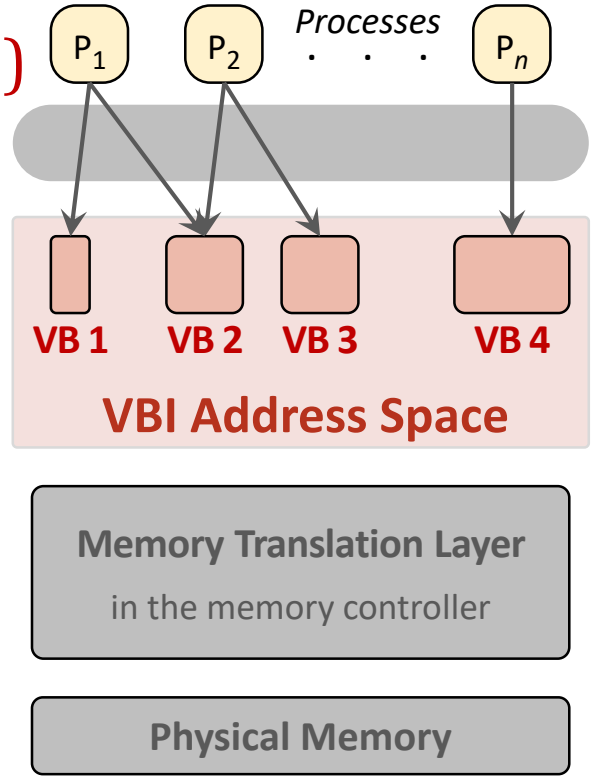
# Virtual Blocks

- **Globally-visible *VBI address space***
  - Consists of a set of ***virtual blocks*** (VBs) of different sizes
    - Example size classes: 4 KB, 128 KB, 4 MB, 128 MB, 4 GB, 128 GB, 4 TB, 128 TB



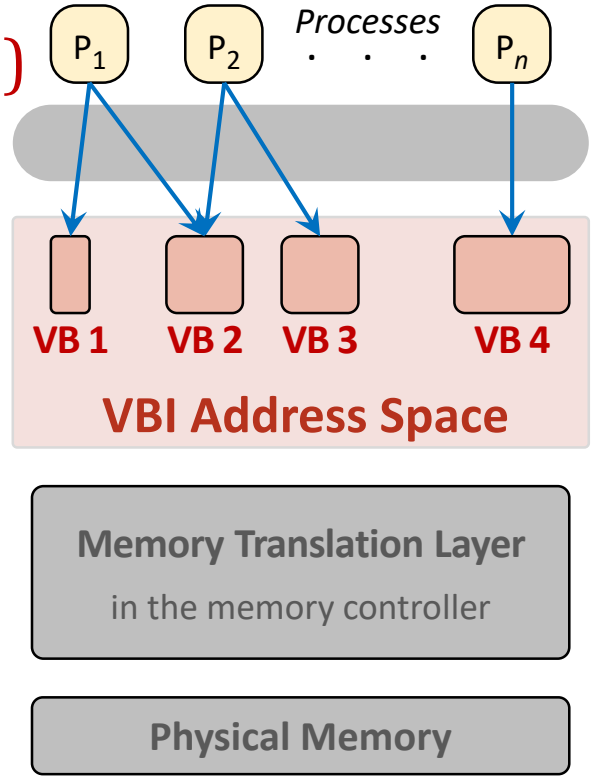
# Virtual Blocks

- **Globally-visible *VBI address space***
  - Consists of a set of ***virtual blocks*** (VBs) of different sizes
    - Example size classes: 4 KB, 128 KB, 4 MB, 128 MB, 4 GB, 128 GB, 4 TB, 128 TB
- All VBs are visible to all processes



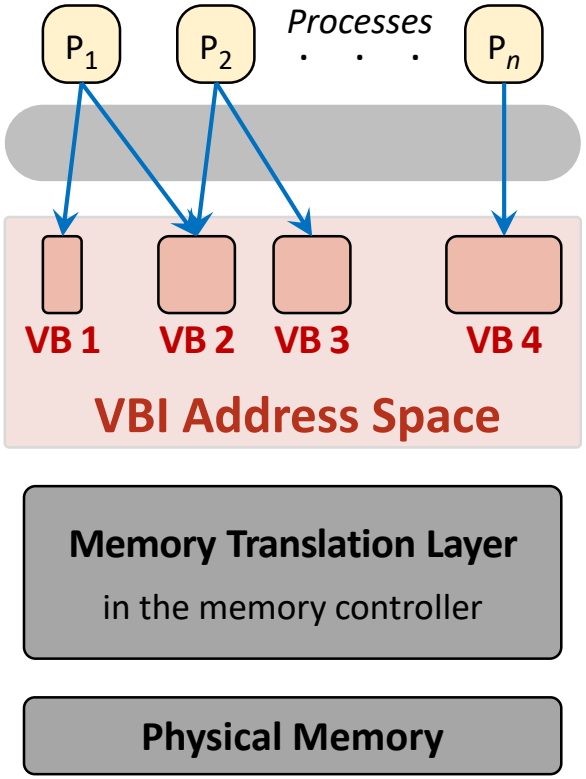
# Virtual Blocks

- **Globally-visible *VBI address space***
  - Consists of a set of ***virtual blocks*** (VBs) of different sizes
    - Example size classes: 4 KB, 128 KB, 4 MB, 128 MB, 4 GB, 128 GB, 4 TB, 128 TB
- All VBs are visible to all processes
- Processes map each **semantically meaningful unit of information** to a separate VB
  - e.g., a data structure, a shared library



# Inherently Virtual Caches

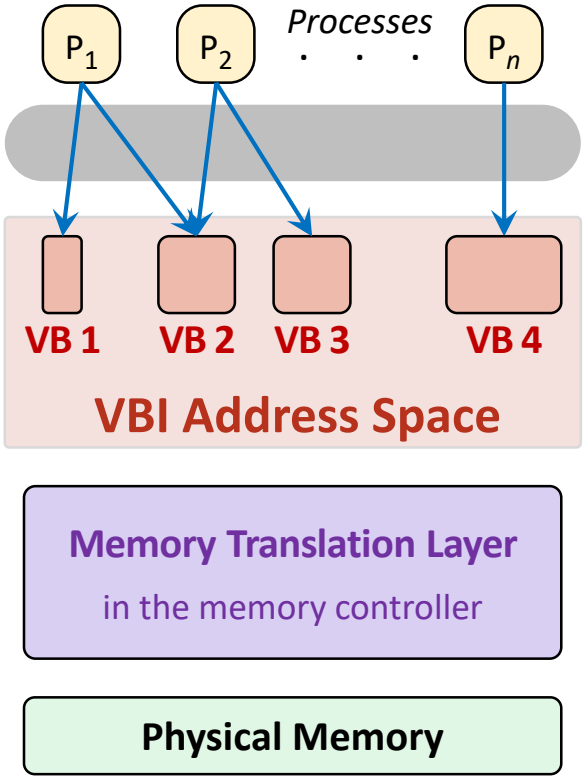
- VBI address space provides ***system-wide*** **unique VBI addresses** which are also visible to caches
- **VBI addresses** are **directly** used to access on-chip caches
  - No longer require address translation
- **Pros:**
  - Enables inherently virtual caches
    - no synonyms and homonyms




synonym = different addresses for the same data
homonym = same address for different data

# Hardware-Managed Memory

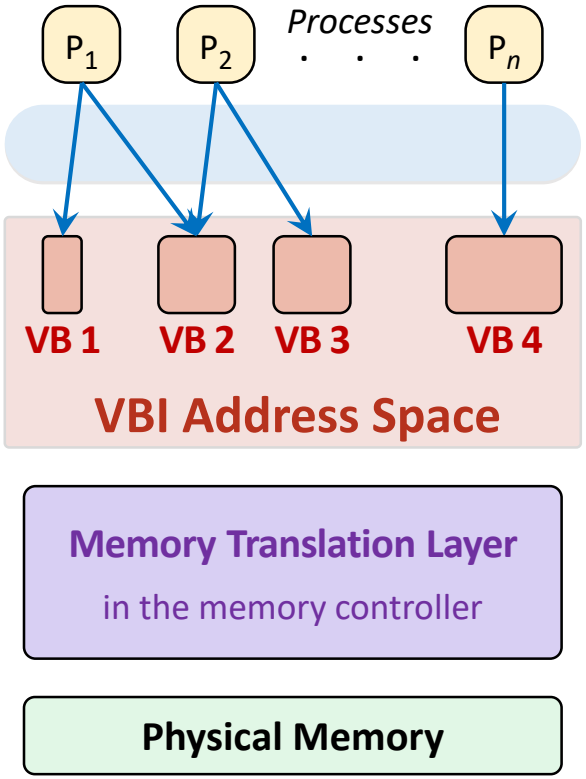
- Memory management is **delegated** to the **Memory Translation Layer** (MTL) in the memory controller
  - Address translation
  - Physical memory allocation




- **Pros:** Many benefits, including
  - Physical memory is allocated only when the location needs to be written to memory
  - No need for 2D page walks in virtual machines
  - Enabling flexible translation structures

# OS-Managed Access Permissions

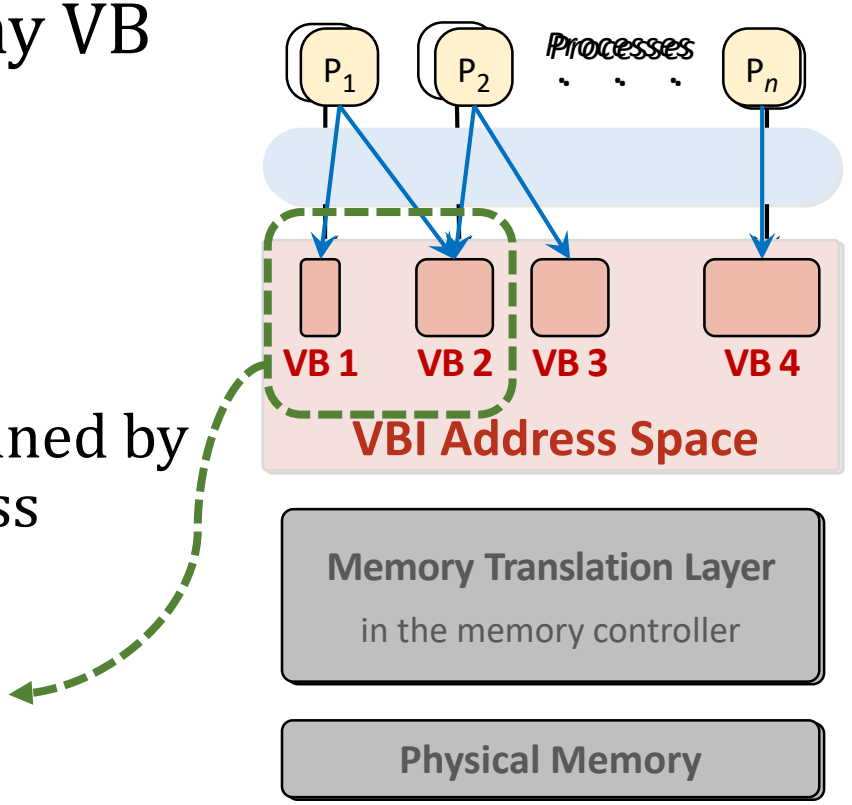
- OS controls which processes access which VBs
- Each process has its own permissions (read/write/execute) when ***attaching*** to a VB
- OS maintains a list of VBs attached to each process
  - Stored in a per-process table
  - Used during permission checks



# Process Address Space in VBI

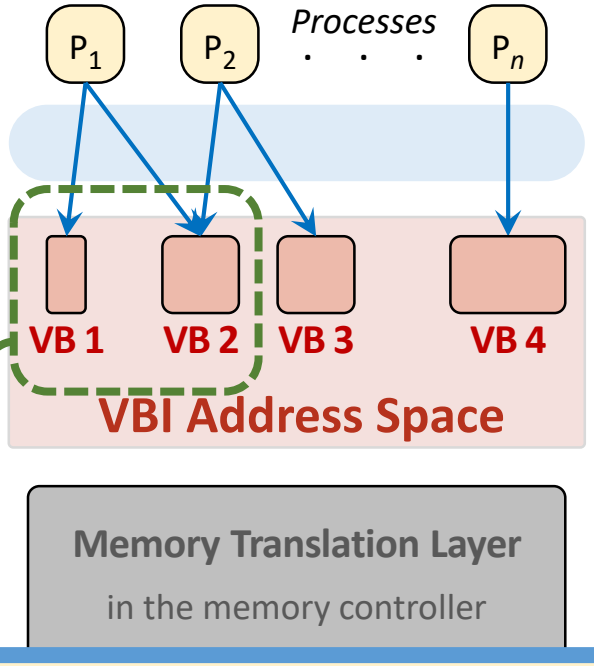
- Any process can attach to any VB
- A process' VBs define its **address space**
  - Address space size is determined by the actual needs of the process

**the address space of process $P_1$**



# Process Address Space in VBI

- Any process can attach to any VB
- A process' VBs define its **address space**
  - Address space size is determined by the actual needs of the process




the address space of

**First guiding principle:**

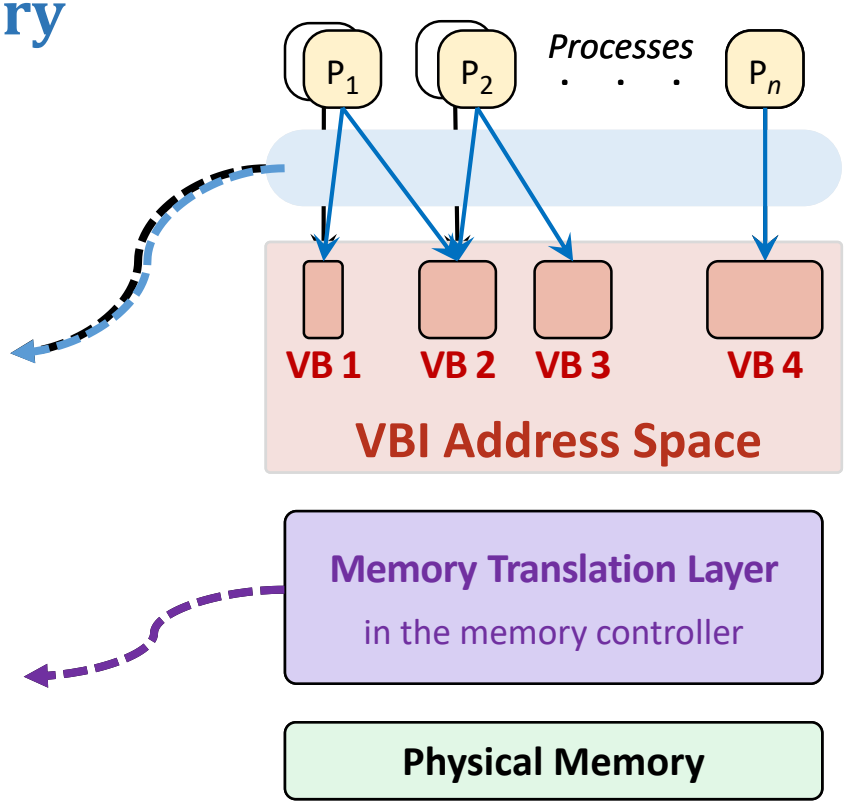
**Appropriately-sized virtual address spaces**

# Decoupled Protection and Translation

Conventional Virtual memory

Access permissions managed by OS

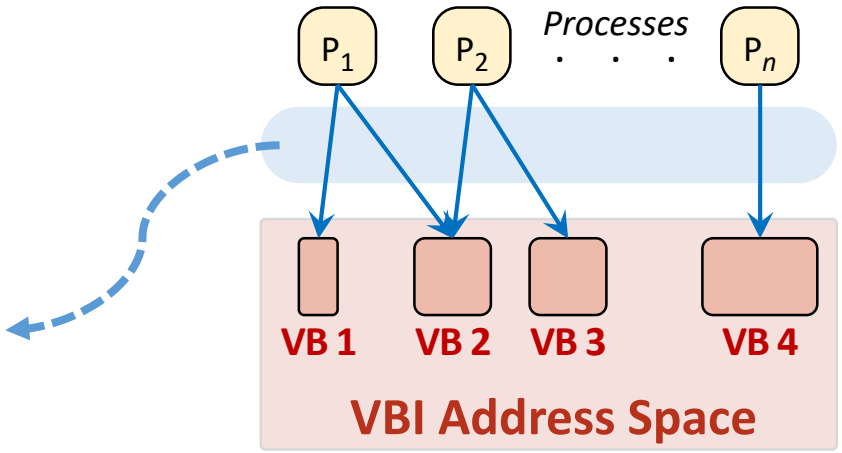
Address mapping managed by the MTL



# Decoupled Protection and Translation

**VBI**

**Access permissions managed by OS**




**Address mapping**

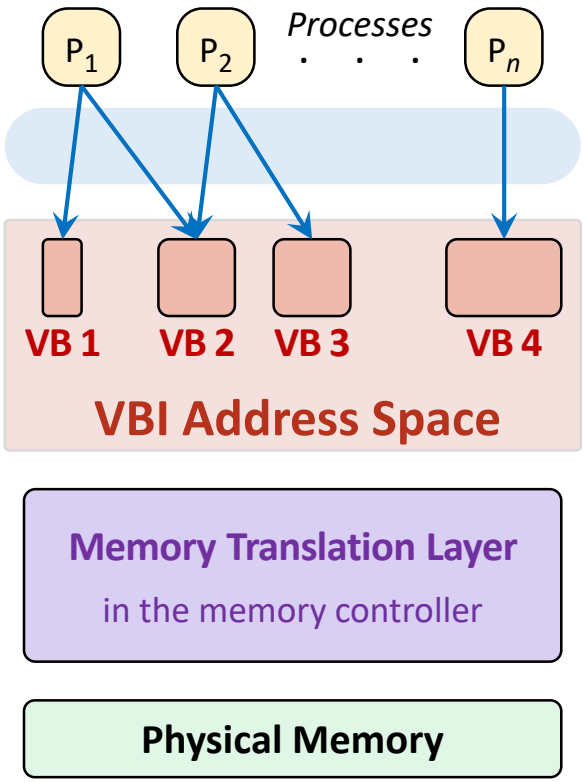
**Memory Translation Layer**
in the memory controller

**Second guiding principle:**

**Decoupling address translation from access protection**

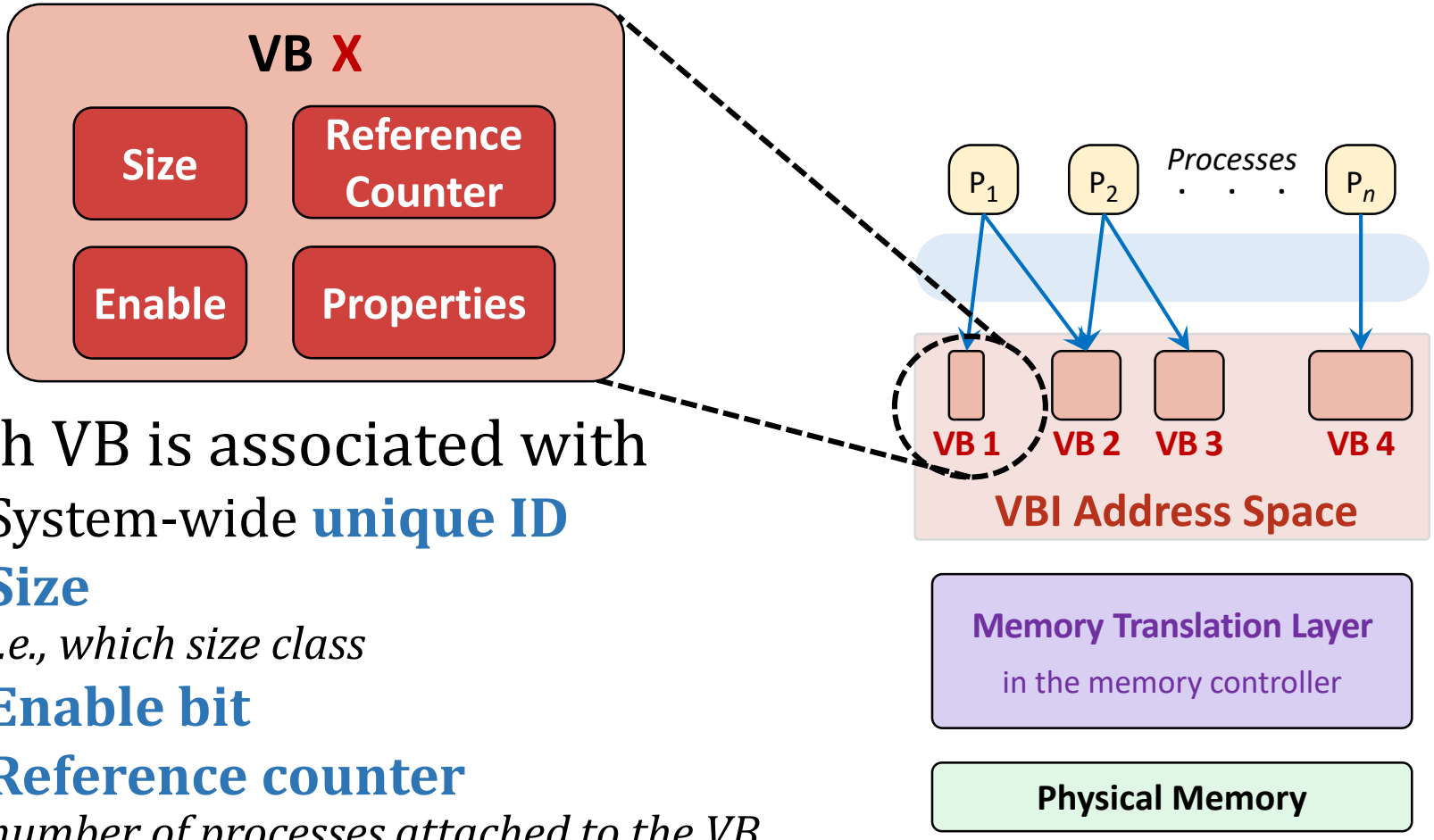
# Address Translation Structures in VBI

- Translation structures are **not shared** with the OS
  - Separate structures for translation and permission information
  - Allows flexible translation structures
  - Per-VB translation structure tuned to the VB's characteristics
    *e.g., single-level tables for small VBs*



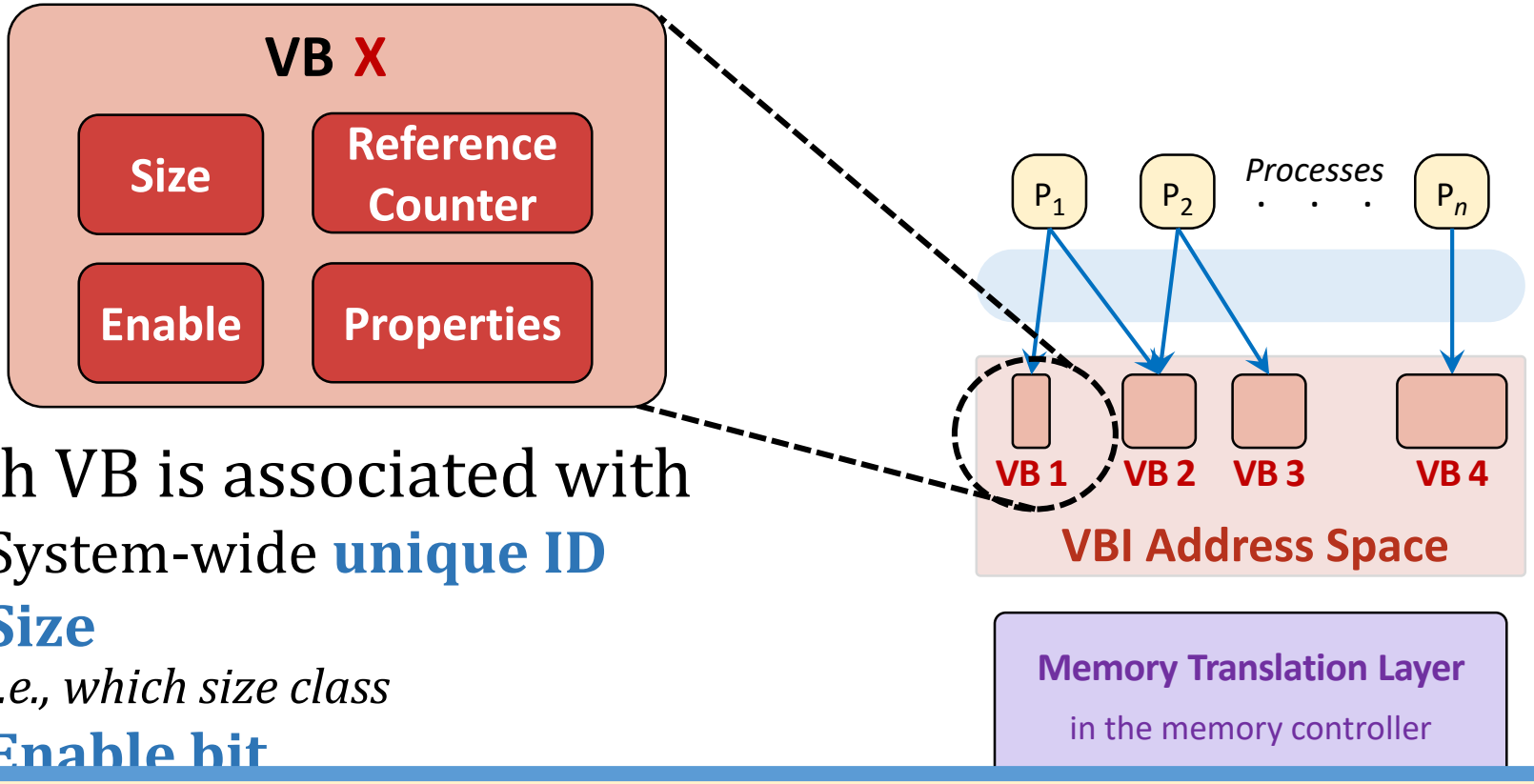

- **Pros:**
  - Lowers overheads and allows for customization

# VB Information




- Each VB is associated with
  - System-wide **unique ID**
  - **Size**
    *i.e., which size class*
  - **Enable bit**
  - **Reference counter**
    *number of processes attached to the VB*
  - **Properties bit vector**
    *semantic information about VB contents,*
    *e.g., access pattern, latency sensitive vs. bandwidth sensitive*

# VB Information



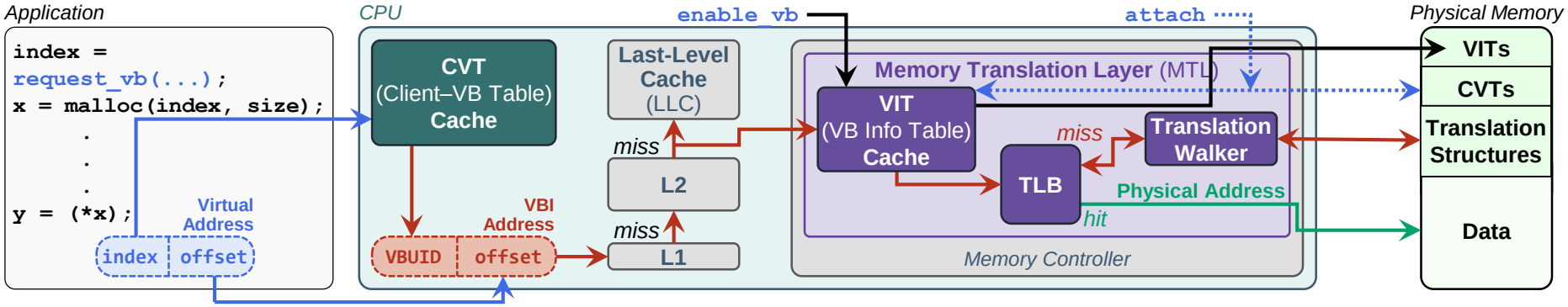

- Each VB is associated with
  - System-wide **unique ID**
  - **Size**
    *i.e., which size class*
  - **Enable bit**

**Third guiding principle:**

**Communicating data semantics to the hardware**

# Implementing VBI

- Please refer to our paper
  - Detailed reference implementation and microarchitecture



# Outline

- Motivation
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- Methodology
- Results
- Summary
- Review
- Discussion

# Optimizations Naturally Enabled by VBI

- **Many optimizations not easily attainable before**

- **Examples:**
  - Appropriately sized process address space
  - Flexible address translation structures
  - Communicating data semantics to the hardware

  *Achieved through guiding principles*

  - Delayed physical memory allocation
  - Eliminating 2D page walks in virtual machines

  *Covered next*

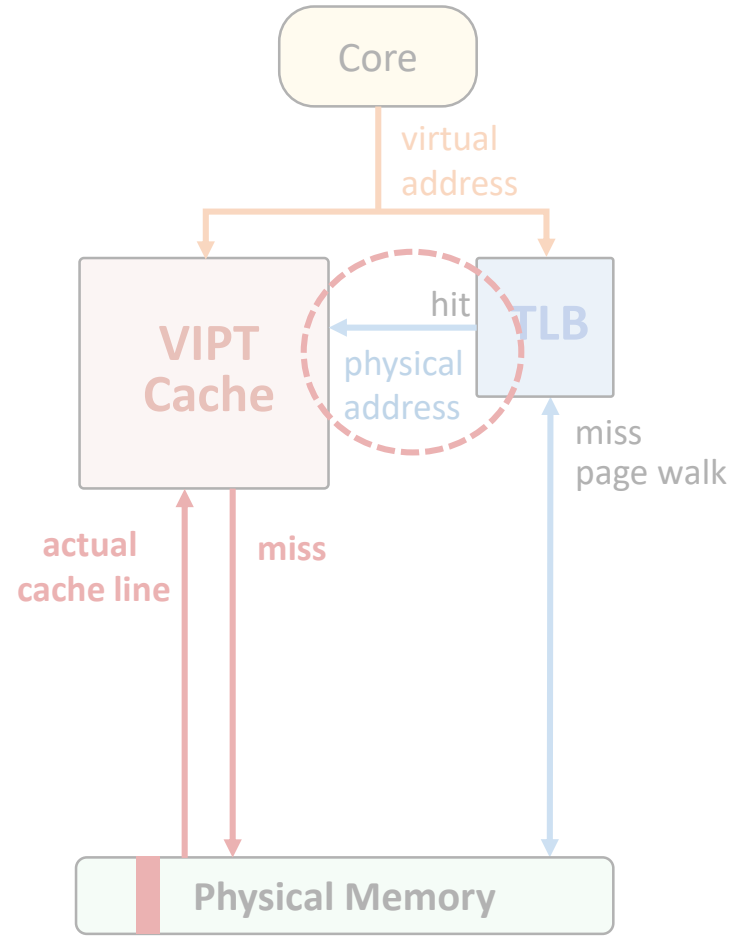
  - Inherently virtual caches
  - Early memory reservation mechanism

  *Covered in our paper*

# Example Optimizations

- Delayed physical memory allocation
- Eliminating 2D page walks in virtual machines

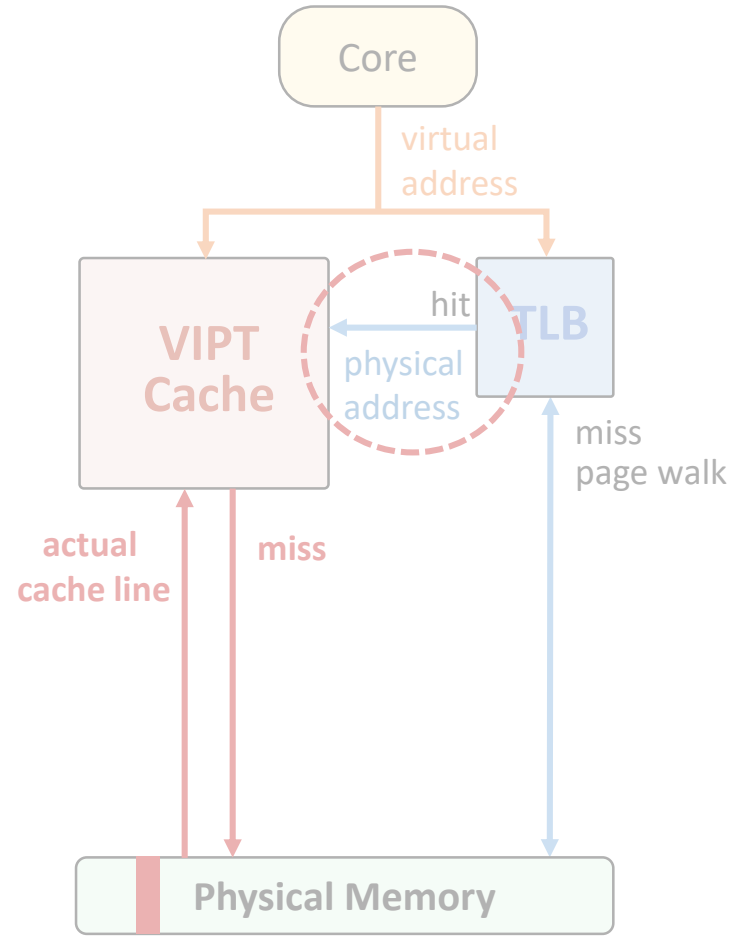
# Delayed Physical Memory Allocation




virtually-indexed physically-tagged (VIPT)

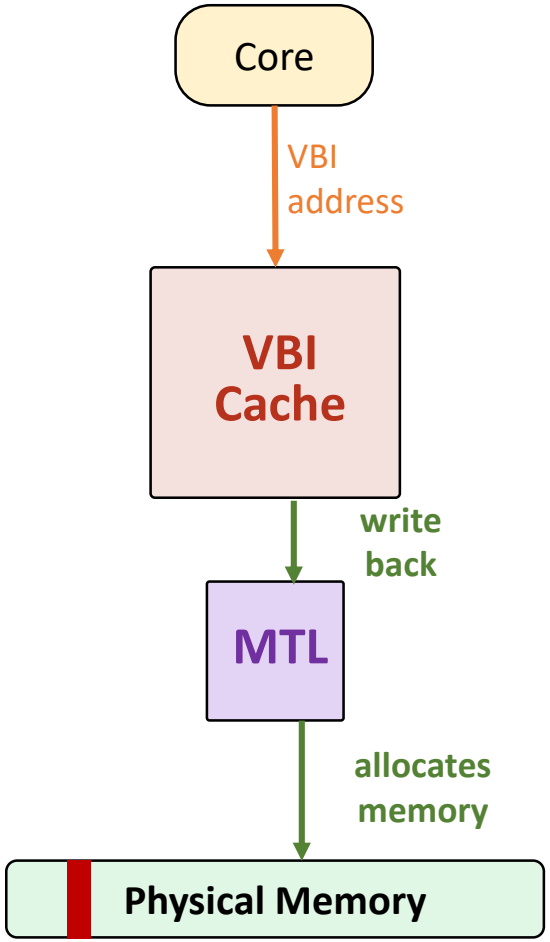
In Conventional Virtual Memory

# Delayed Physical Memory Allocation




virtually-indexed physically-tagged (VIPT)

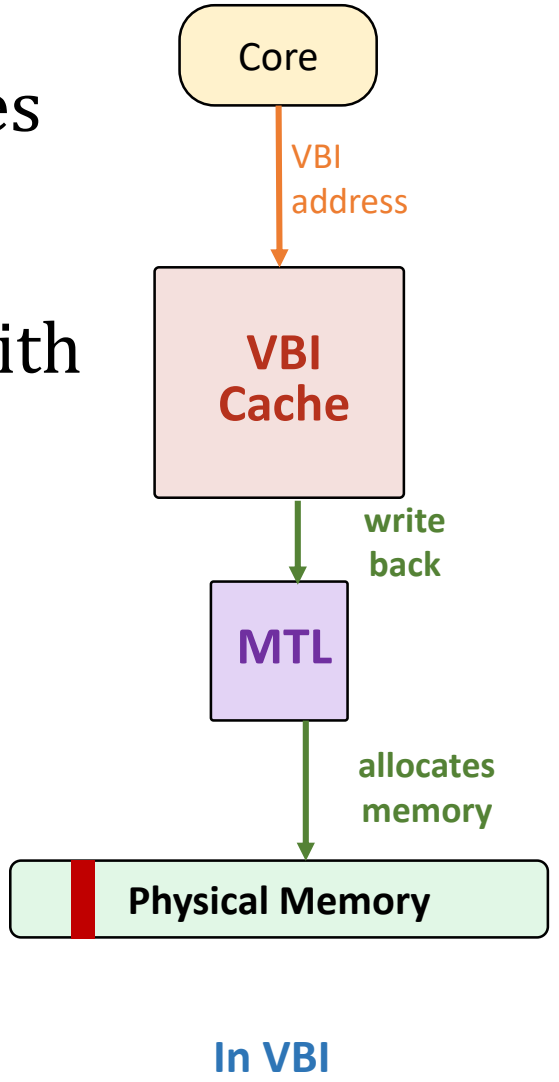
In Conventional Virtual Memory




In VBI

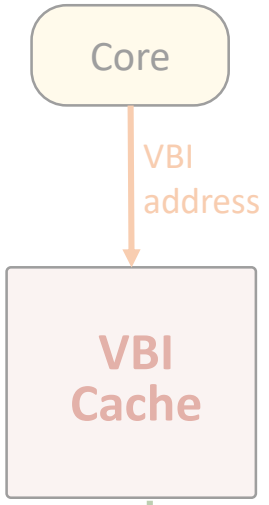
# Delayed Physical Memory Allocation

- No address translation for accesses to regions with no allocation
- No memory accesses to regions with no allocation yet
- No memory allocation for VBs that never leave the cache during their lifetime



# Delayed Physical Memory Allocation

- No address translation for accesses to regions with no allocation
- No memory access to regions with no allocation yet
-




**VBI reduces address translation overhead, improves overall performance, and reduces memory consumption**

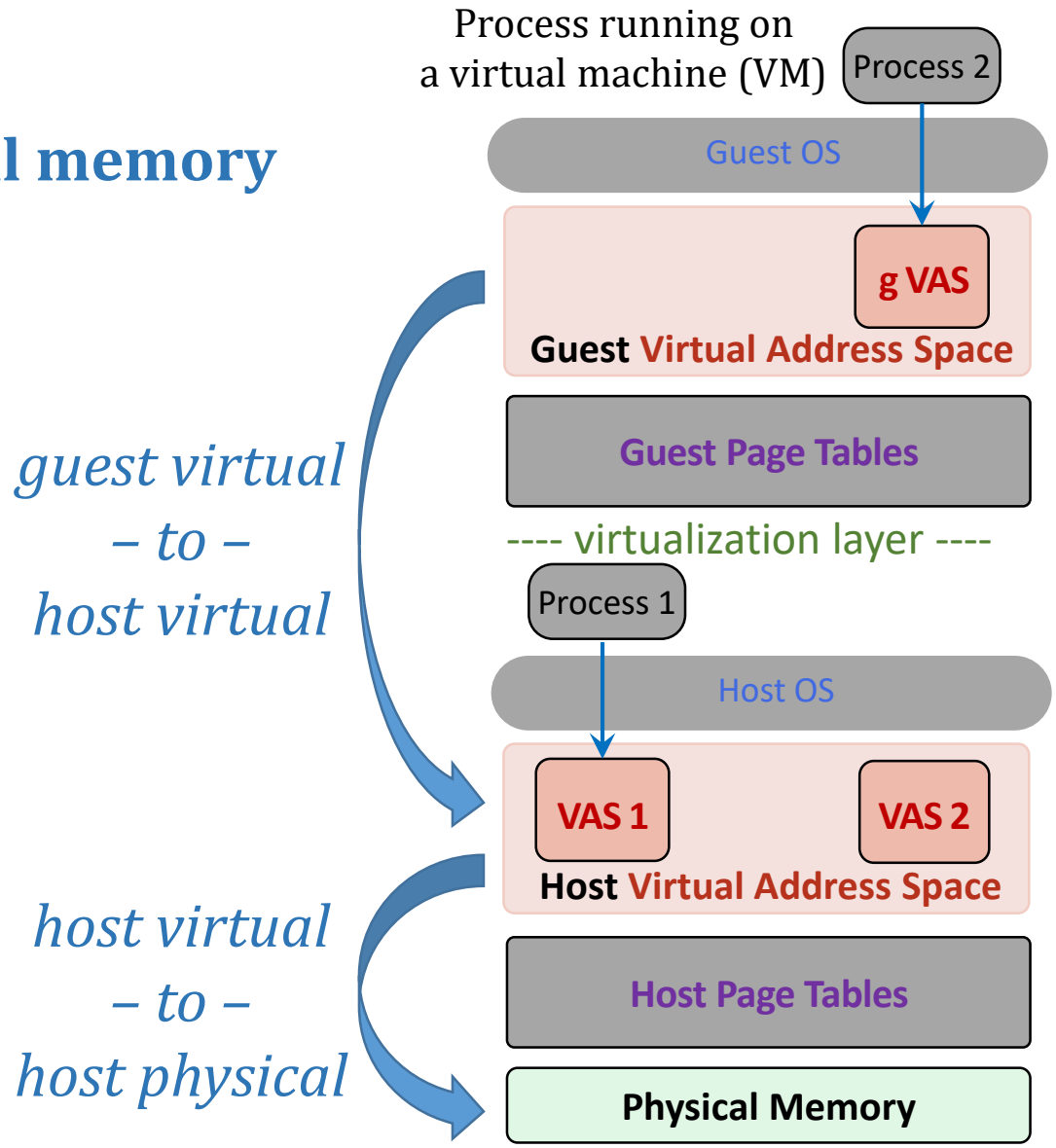
in VBI

# Example Optimizations

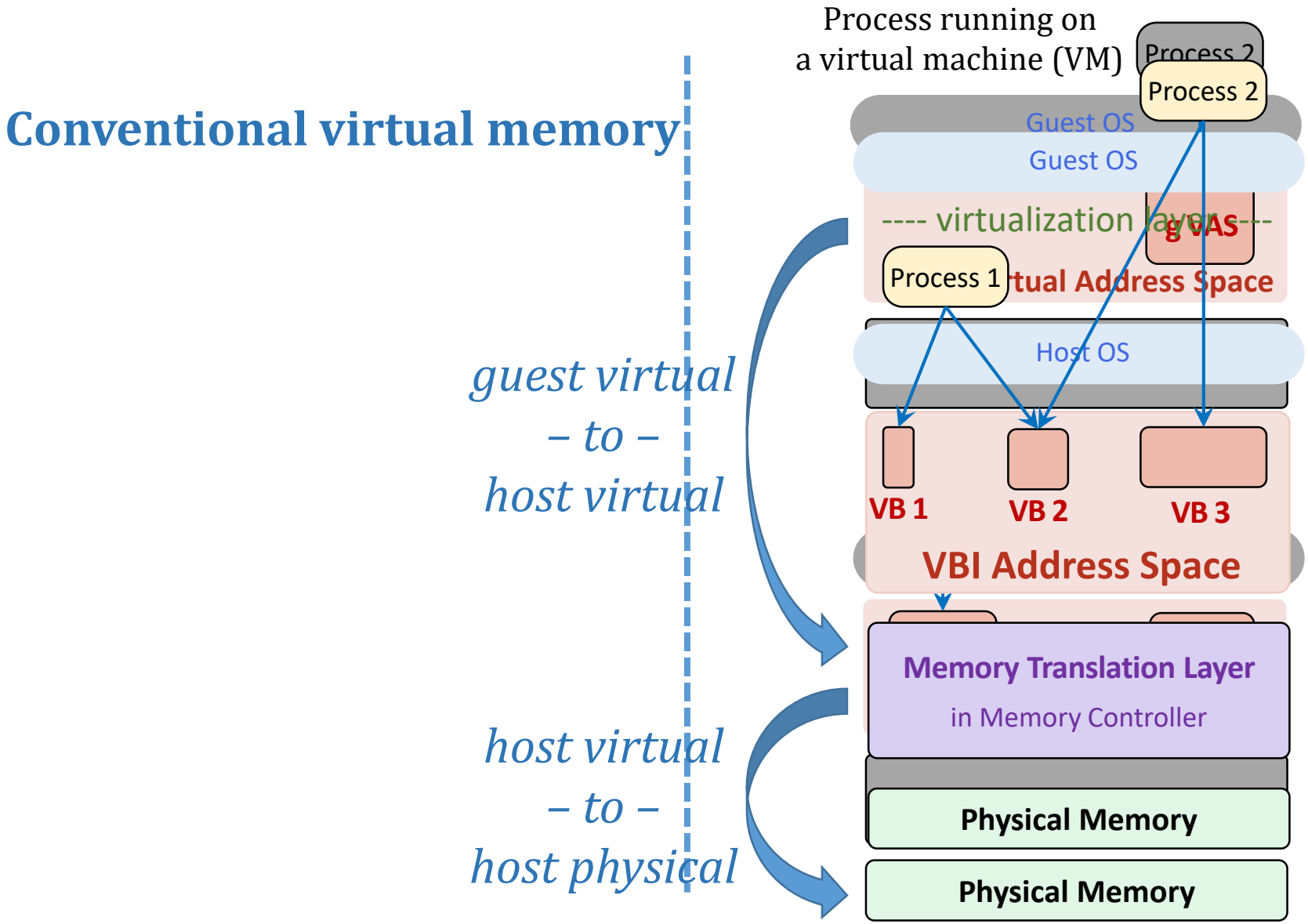
- Inherently virtual caches
- Eliminating 2D page walks in virtual machines

# Eliminating 2D Page Walks in Virtual Machines

## Conventional virtual memory



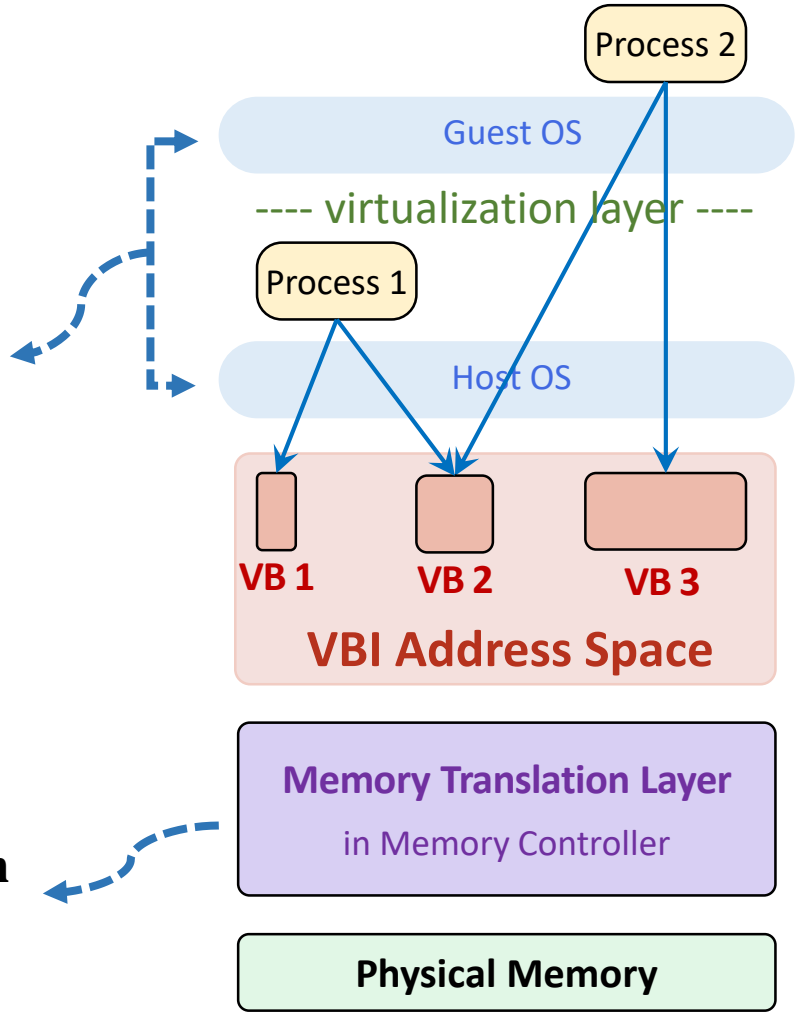
# Eliminating 2D Page Walks in Virtual Machines



# Eliminating 2D Page Walks in Virtual Machines

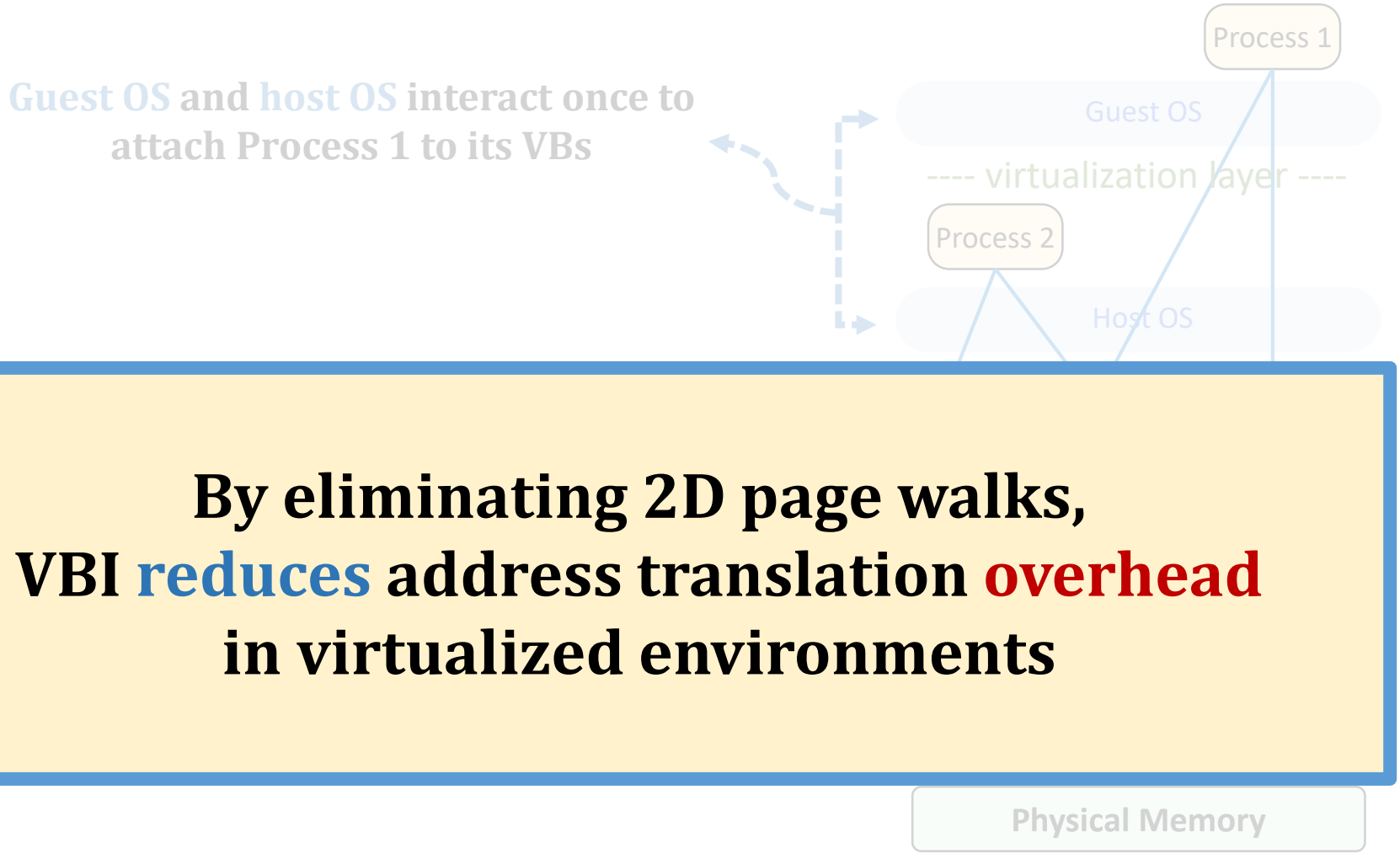
## VBI

Guest OS and host OS interact once to attach Process 2 to its VBs




MTL is the only component in the system that manages address mapping

# Eliminating 2D Page Walks in Virtual Machines




**By eliminating 2D page walks,
VBI reduces address translation overhead
in virtualized environments**

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Methodology

- **Simulator:** heavily-modified version of Ramulator
  - Models virtual memory components (e.g., TLBs, page tables)
  - Available at https://github.com/CMU-SAFARI/Ramulator-VBI
- **Workloads:** SPECspeed 2017, SPEC CPU 2006, TailBench, Graph 500
- **System parameters:**
  - Core: 4-wide issue, OOO, 128-entry ROB
  - L1 Cache: 32 KB, 8-way associative, 4 cycles
  - L2 Cache: 256 KB, 8-way associative, 8 cycles
  - L3 Cache: 8 MB (2 MB per-core), 16-way associative, 31 cycles
  - L1 DTLB: - 4 KB pages: 64-entry, fully associative
    - 2 MB pages: 32-entry, fully associative
  - L2 DTLB: 4 KB and 2 MB pages: 512-entry, 4-way associative
  - Page Walk Cache: 32-entry, fully associative
  - DRAM: DDR3-1600, 1 channel, 1 rank/channel, 8 banks/rank
  - PCM: PCM-800, 1 channel, 1 rank/channel, 8 banks/rank

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

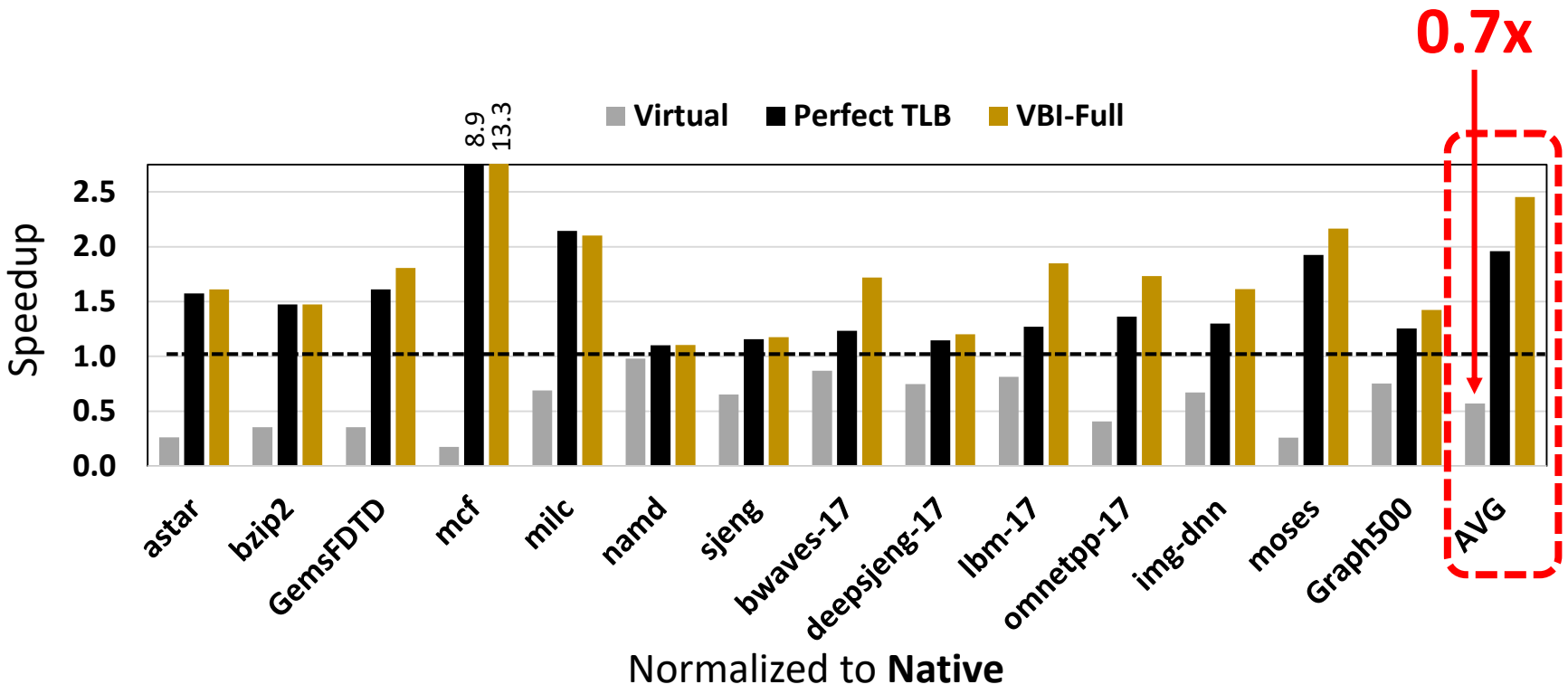
# Use Case 1: Address Translation

- The impact of VBI on reducing the address translation overhead in both native execution and virtual machines

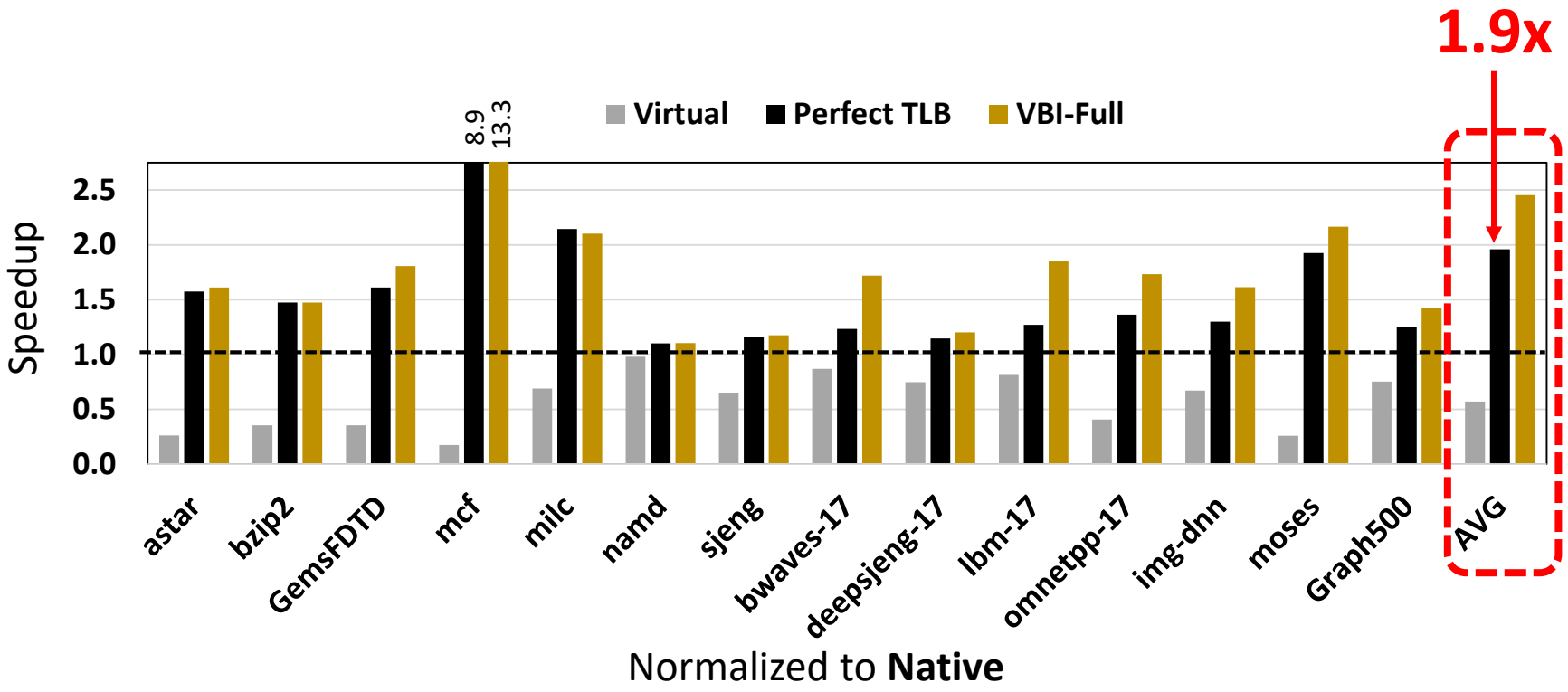
- **Evaluated systems:**
  - **Three baselines:**
    - **Native:** applications run natively on an x86-64 system
    - **Virtual:** applications run inside a virtual machine (accelerated using 2D page walk cache [Bhargava+, ASPLOS'08])
    - **Perfect TLB:** an unrealistic version of Native with no translation overhead
  - **One VBI configuration:**
    - **VBI-Full:** VBI with all the optimizations that it enables

- See our paper for results on more system configurations

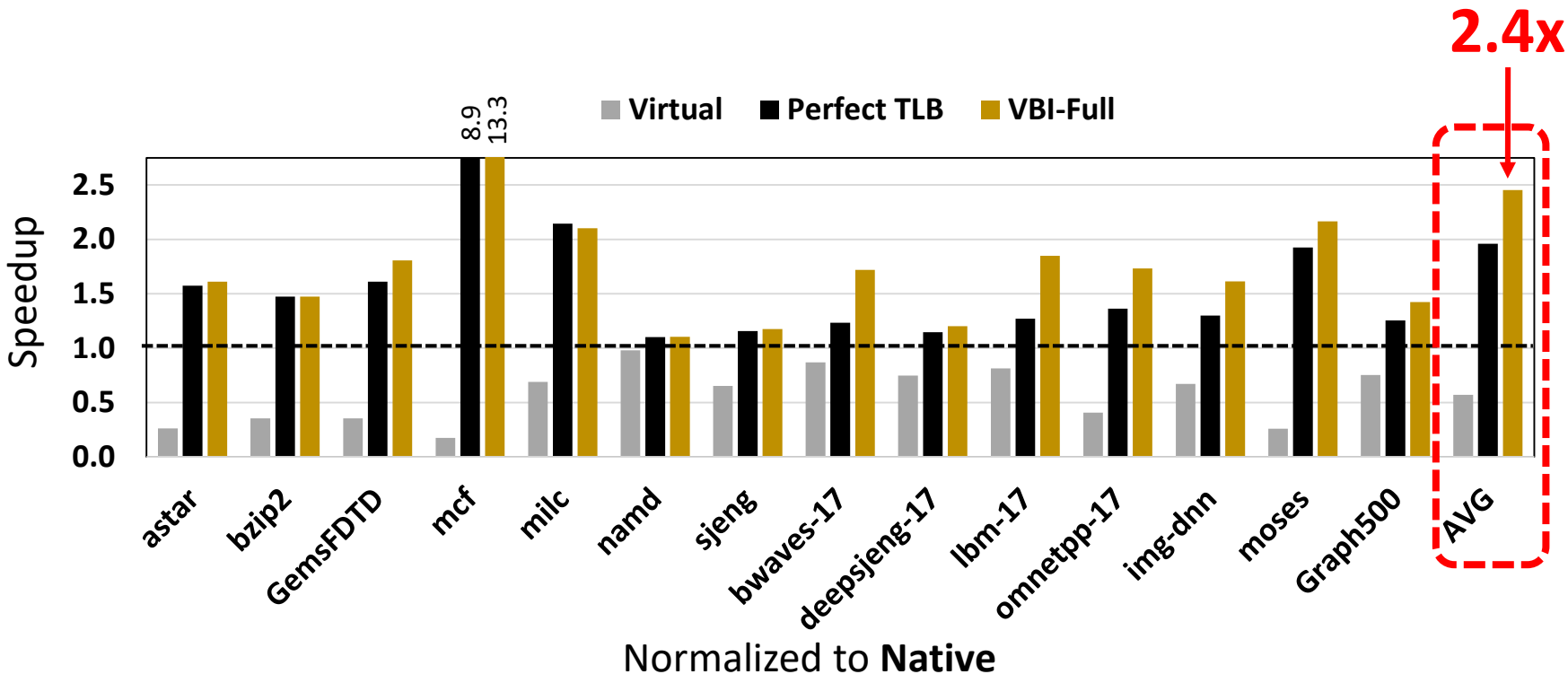
# Use Case 1: Address Translation



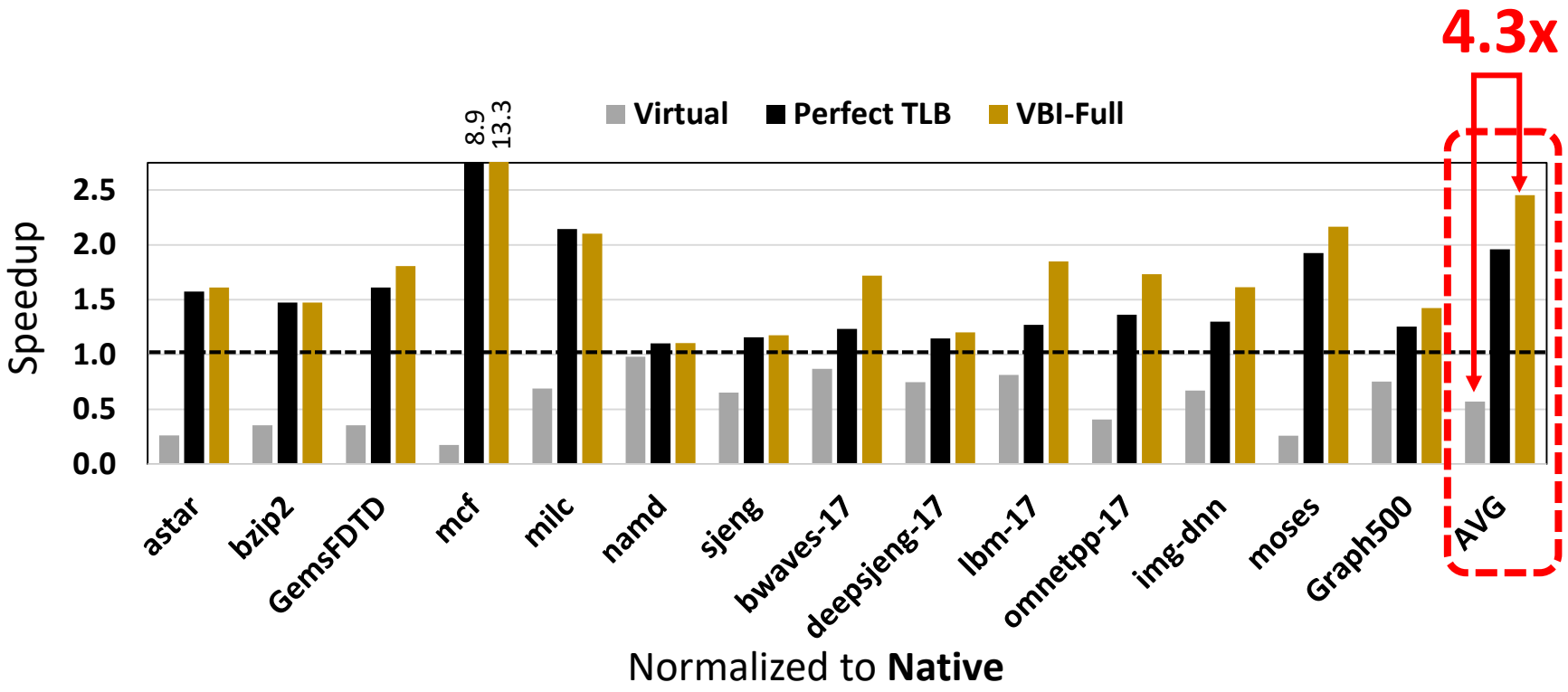
# Use Case 1: Address Translation



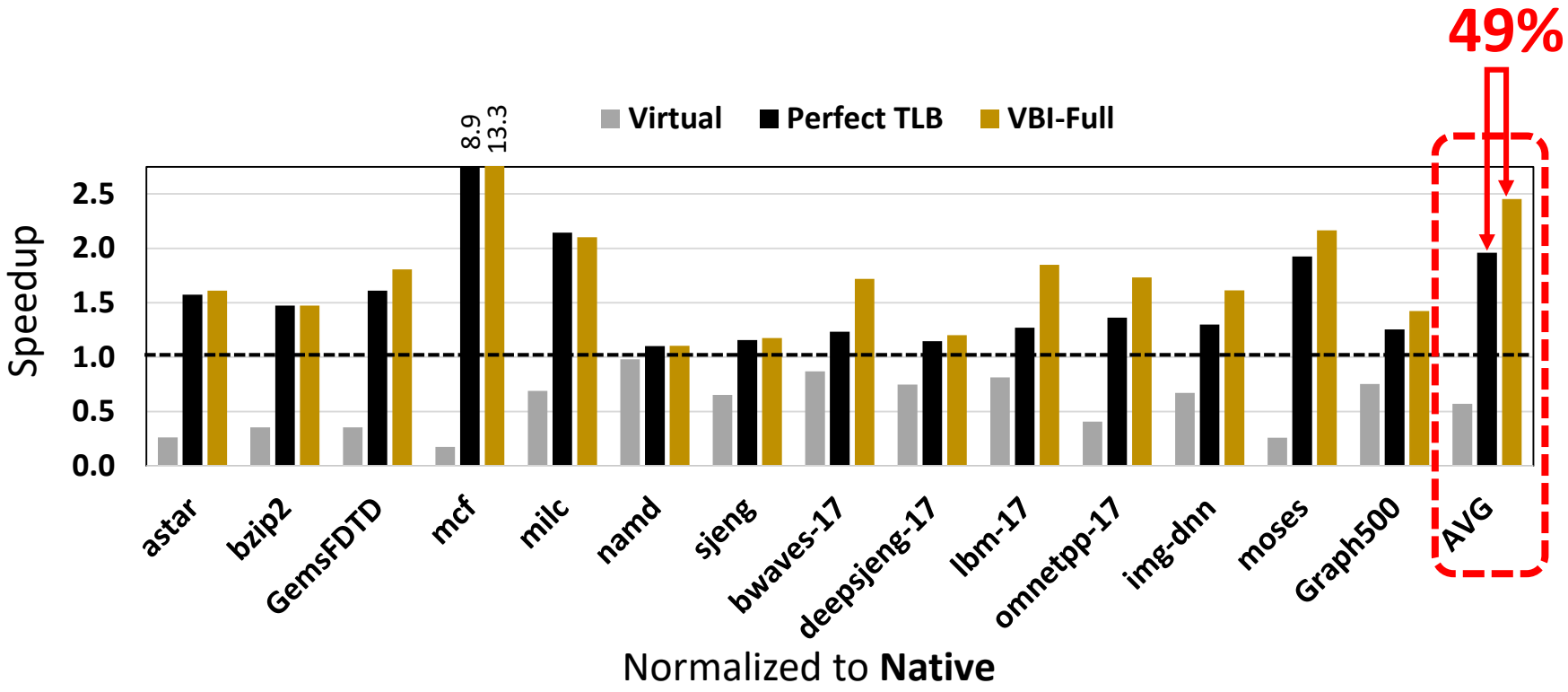
# Use Case 1: Address Translation



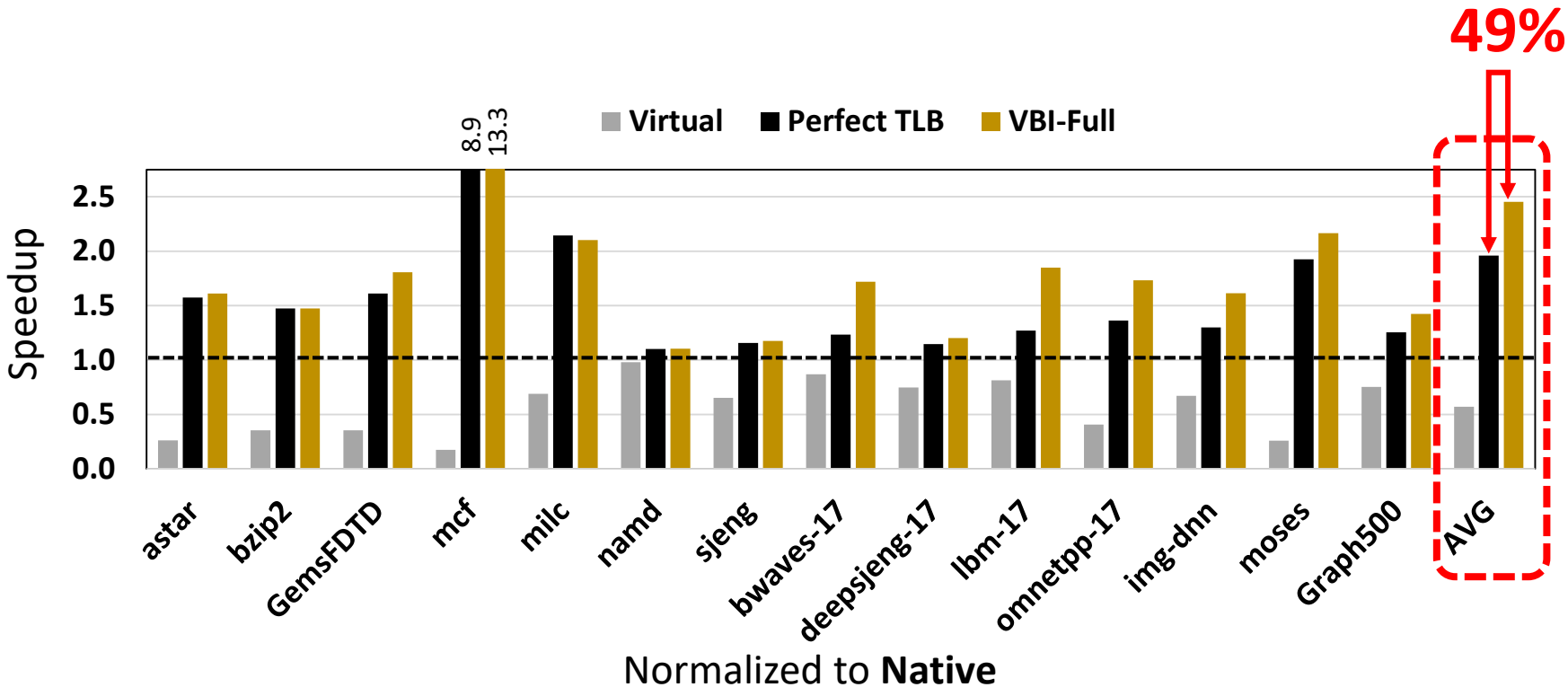
# Use Case 1: Address Translation



# Use Case 1: Address Translation



# Use Case 1: Address Translation

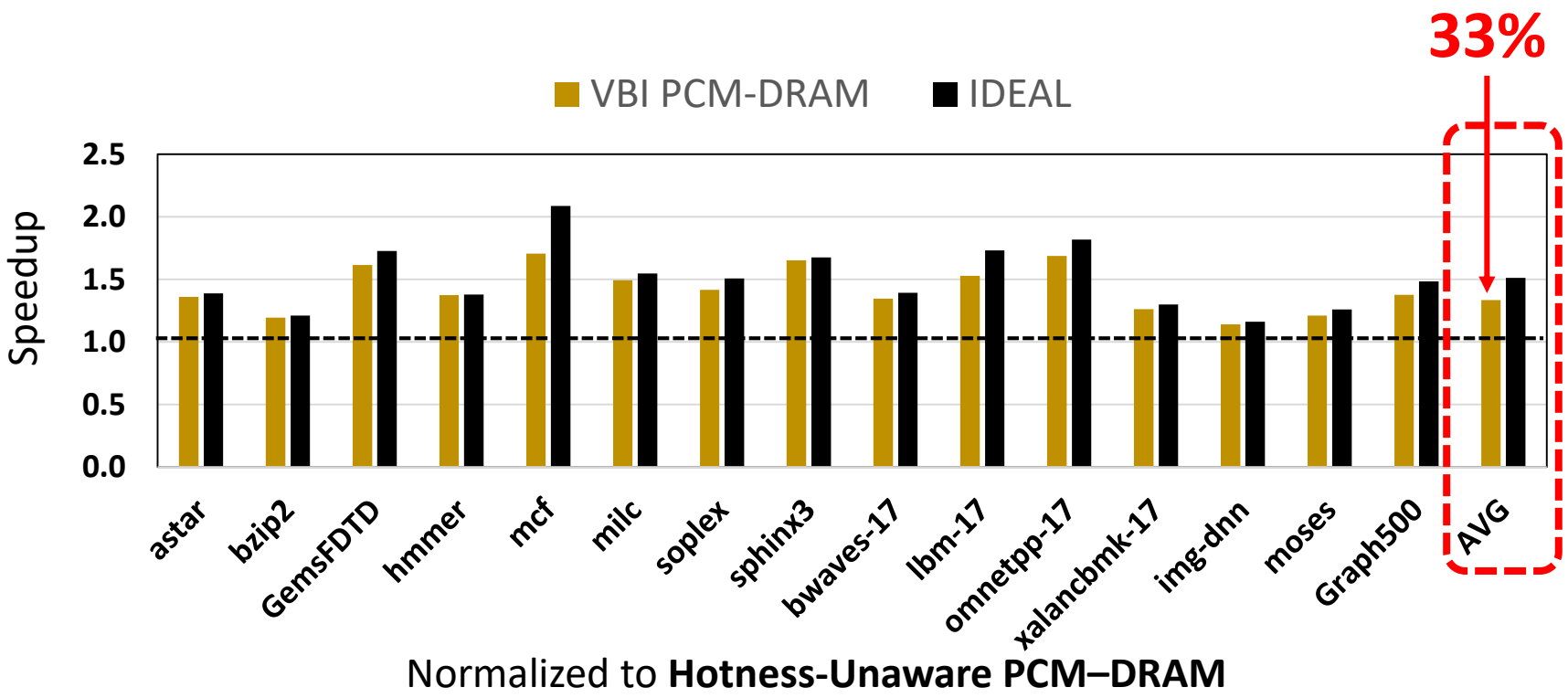



**VBI significantly improves performance in both native execution and virtual machines**

# Use Case 2: Memory Heterogeneity

- The benefits of VBI in harnessing the full potential of heterogeneous memory architectures
  - Hybrid PCM–DRAM memory architecture
- **Evaluated systems:**
  - **Two baselines:**
    - **Hotness-Unaware PCM–DRAM:** unaware of the data hotness
    - **IDEAL:** always maps frequently-accessed data to DRAM
  - **One VBI configuration:**
    - **VBI PCM–DRAM:** VBI maps and migrates frequently-accessed VBs to the DRAM

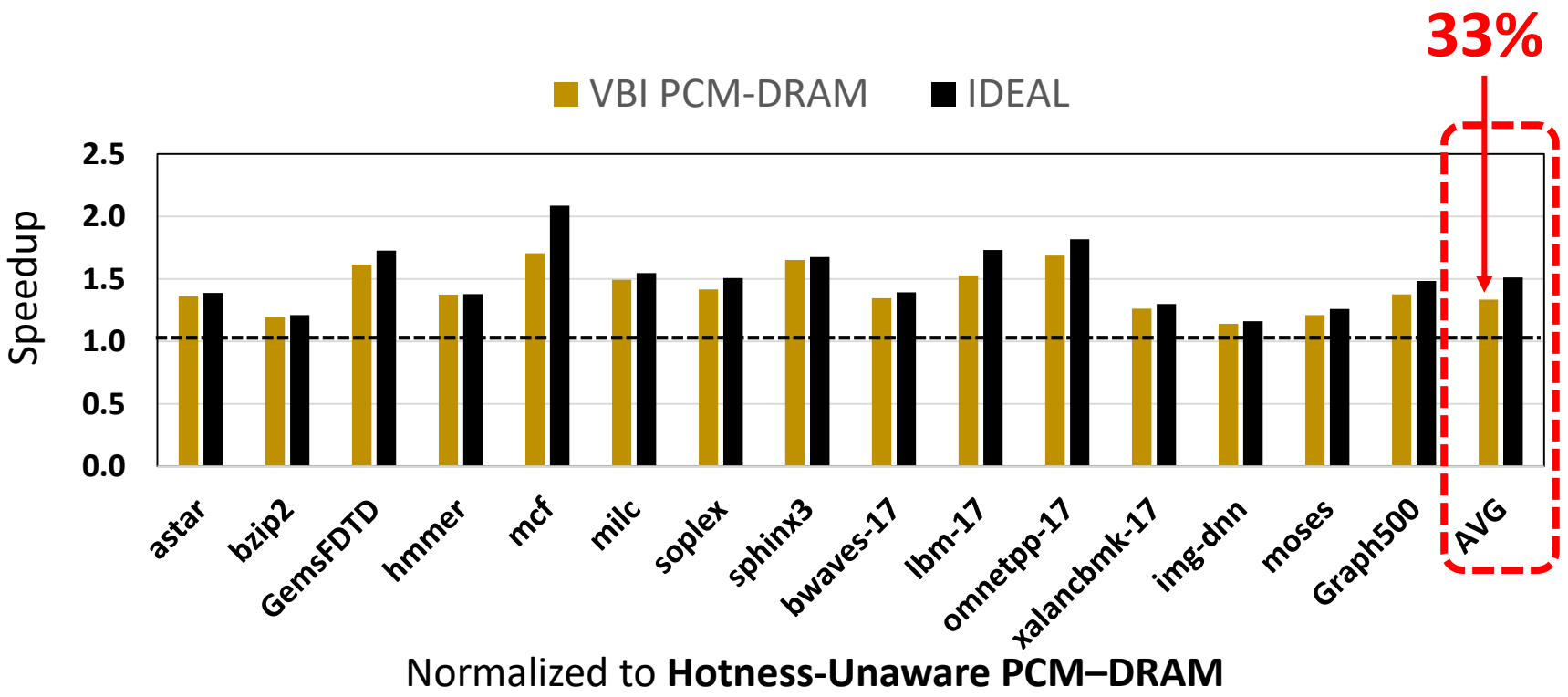
# Use Case 2: Memory Heterogeneity




**More in our paper:**

- Similar performance improvement for Tiered-Latency-DRAM [Lee+, HPCA'13]

# Use Case 2: Memory Heterogeneity




**VBI enables efficient data mapping and data migration for heterogeneous memory systems**

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Summary

- **Virtual Block Interface (VBI):** A new virtual memory framework
  - Addresses the challenges in adapting conventional virtual memory to increasingly diverse system configurations and workloads
- **Key Idea:** Delegate physical memory management to dedicated hardware in the memory controller
- **Benefits:** Not easily attainable in conventional virtual memory (e.g., inherently virtual caches , delaying physical memory allocation, and avoiding 2D page walks in virtual machines)
- **Evaluation:**
  - VBI significantly improves performance in both native execution and virtual machines
  - Increases the effectiveness of managing heterogeneous memory architectures
- **Conclusion:** VBI is a promising new virtual memory framework
  - Can enable several important optimizations
  - Increases design flexibility for virtual memory
  - A new direction for future work in novel virtual memory frameworks

# Questions

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Strengths

- Novel and very efficient idea
  - **General-purpose**: solves many problems at once
  - Application/OS/hardware developers all profit
- The authors give an implementation proposal
  - Stimulates to think creatively!
- Offer proposals/replacements for many OS functionalities
  - C-o-W, MM-I/O, Swapping …

**VBI should be the future!**

# Remember RowClone?

- In-memory copy/bulk zeroing
- **Goal:** Reduce cache pollution, latency, bandwidth and energy waste
- Extremely fast within subarray (FPM)
  - Requires data to be mapped in same subarray
  - *Maximize by making OS subarray-aware*

Thanks to VBI this has become easier!

→ OS does no longer manage physical memory

# Weaknesses

- Paper is very dense
  - I struggled with it
    - What is *really* important? How do you present it?
- High initial investment
  - VBI drastically changes all existing software/OS/hardware designs
    - But don't fall into the rat hole!
- Coordination + standardization
  - How to guarantee extensibility of the property bit vector?
    - What if future technologies could profit of more detailed software-provided hints? Where to draw the line?
  - Could lead to inconsistencies across implementations
    - Think about Intel vs. AMD

# Outline

- **Motivation**
- **VBI: Virtual Block Interface**
  - Key Idea & Guiding Principles
  - Design Overview
  - Optimizations Enabled by VBI
- **Methodology**
- **Results**
- **Summary**
- **Review**
- **Discussion**

# Discussion

- Ask me about the implementation proposal …
  - Anything you are interested in!
    - Changes to you as a software/OS/hardware developer
- What do you think about VBI?
  - Do you like/dislike it? Why?
  - Do you think modularity is the right step? (vs. integration)
    - Any ideas where this also should be used?
  - Can even embedded systems profit of VBI?
  - Would you want to achieve compatibility to existing OS?
    - Can you think of an “easy fix”?

# *The Virtual Block Interface:*
## *A Flexible Alternative to the Conventional Virtual Memory Framework*

*ISCA, 2020*

Nastaran Hajinazar  Pratyush Patel  Minesh Patel
Konstantinos Kanellopoulos  Saugata Ghose
Rachata Ausavarungnirun  Geraldo F. Oliveira  Jonathan Appavoo
Vivek Seshadri  Onur Mutlu

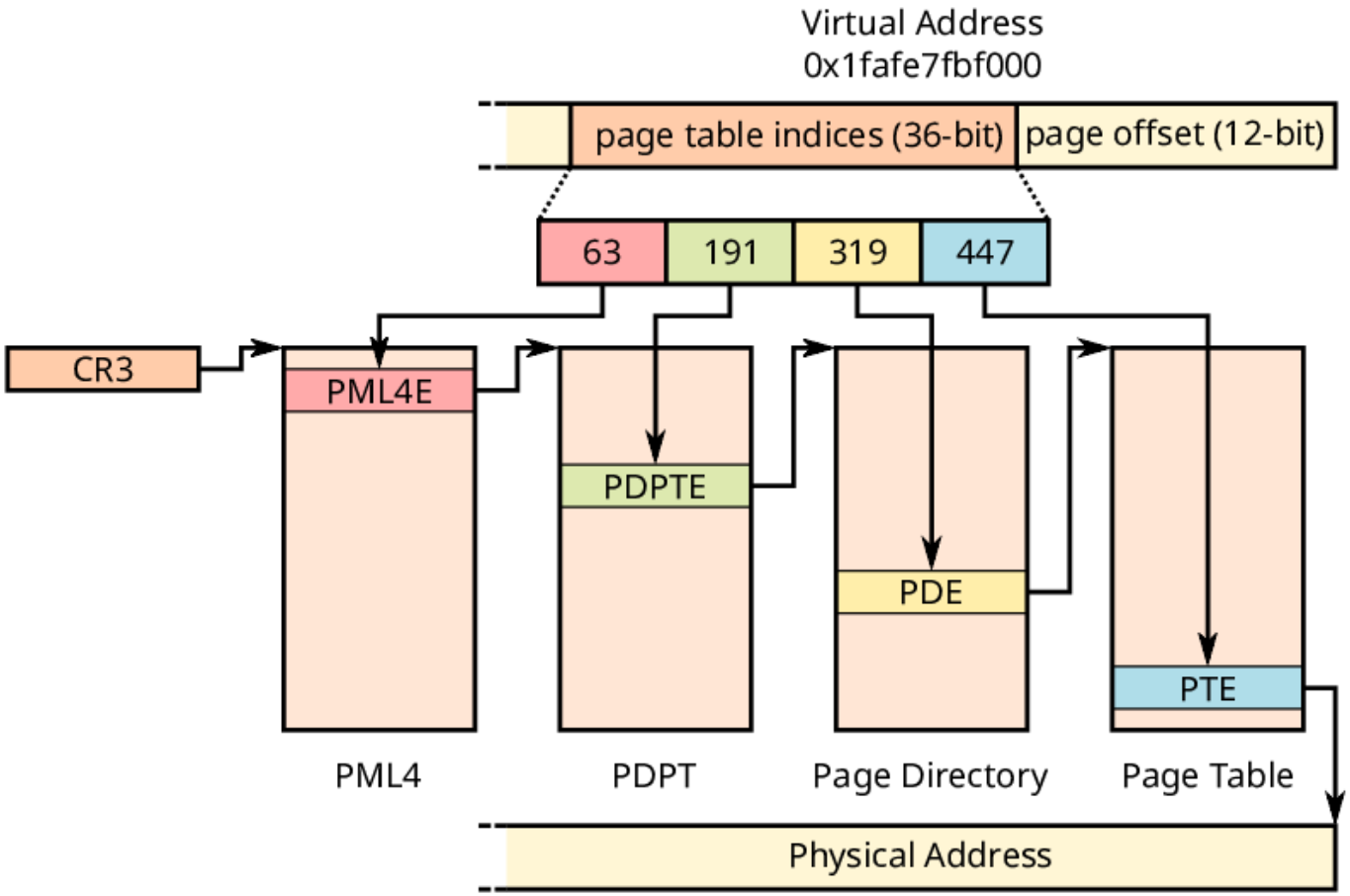
**Presented by Stefan Scholbe**

Most slides by Nastaran Hajinazar

SAFARI  ETH Zürich  SFU Simon Fraser University  University of Washington
Carnegie Mellon  KMUTNB  Microsoft  Boston University

# Backup Slides

# 2D Page Table Walks



# Malloc in VBI

```
index_t request_vb(size_t expectedSize, enum_t hints);

void* malloc(size_t size, index_t vb);
```

# Addressing a VB

- Global **VBI address** is split up

# VB Management

- Stored globally in **VB Info Tables (VIT)** (per size class)
- Reside in a reserved region of PM
- Implicitly managed by OS

| VBID | Enabled | Flags + Hints | RefCount | TransStructType | TransStructPtr |
|---|---|---|---|---|---|
| 0 | Yes | Kernel, **Latency** | 3 | Single-level | 0xAA00BB00… |
| 1 | No | - | 0 | - | - |
| 2 | Yes | User, **Bandwidth** Compressible | 5 | Direct-mapped | 0xDEADBEEF… |

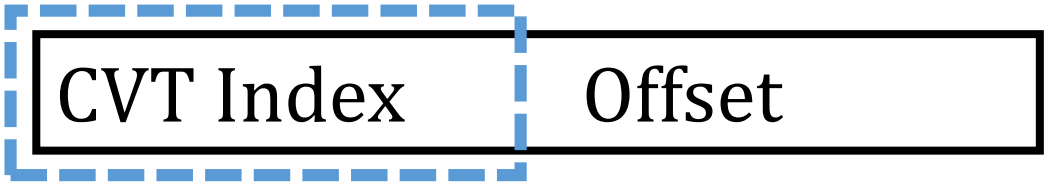
# Client VB-Table (CVT)

- **Per memory client**
- Stores the attached VBs
- Implicitly managed by OS

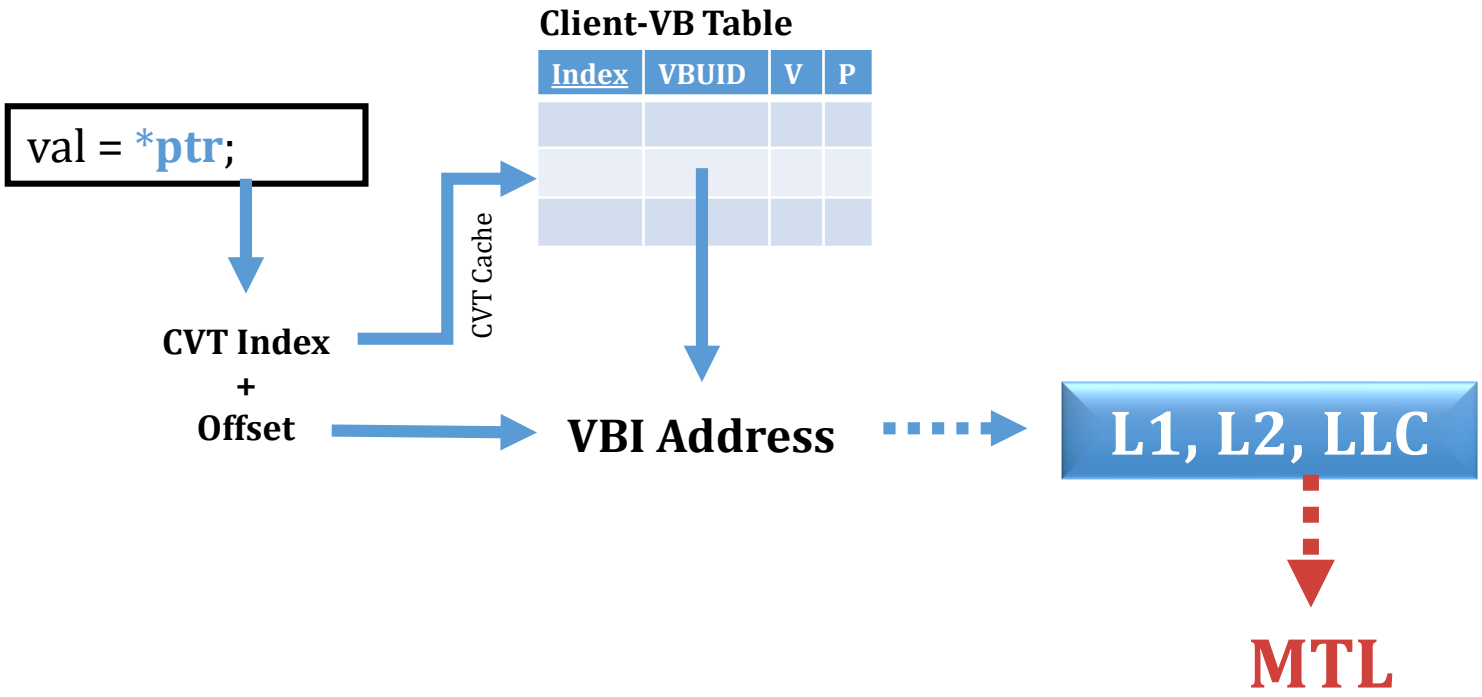
| Index | VBUID | Valid | Permissions |
|---|---|---|---|
| 0 | 3 | Yes | RW |
| 1 | 12 | Yes | RX |
| 2 | 8 | No | - |

# Addressing a VB now

- **Virtual address** is split up
- Provide another level of indirection and allows easier relocation



# Addressing a VB now (CPU)



# Addressing a VB now (MTL)