Rethinking Virtual Memory

Nastaran Hajinazar

Seminar in Computer Architecture
31 March 2022
The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

Nastaran Hajinazar*† Pratyush Patel* Minish Patel* Konstantinos Kanellopoulos* Saugata Ghose†
Rachata Ausavarungnirun○ Geraldo F. Oliveira* Jonathan Appavoo○ Vivek Seshadri▼ Onur Mutlu*†

*ETH Zürich †Simon Fraser University □University of Washington ‡Carnegie Mellon University
○King Mongkut’s University of Technology North Bangkok ○Boston University ▼Microsoft Research India

Published in ISCA 2020

Link to the paper
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
Outline

Motivation

VBI: Virtual Block Interface
Key Idea & Guiding Principles
Design Overview
Optimizations Enabled by VBI

Methodology

Results

Summary
Virtual Memory

Application

Virtual Memory
managed by the operating system

Hardware
Computing Systems Are Diversifying

Hardware

Application

Virtual Memory
managed by the operating system

Cannot adapt efficiently
Computing Systems Are Diversifying

Continually adapting the conventional virtual memory framework is challenging.
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

- Host
- Virtual Address Space
- Process 1
- Host OS
- VAS 1
- Host Page Tables
- Physical Memory
Challenge 2: Overheads in Virtual Machines

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

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

```
guest virtual  ─ to ─ host virtual
host virtual   ─ to ─ host physical
```

Diagram:
- **Guest** OS
  - Guest Virtual Address Space
  - Guest Page Tables
  - Virtualization layer
- **Host** OS
  - Host Virtual Address Space
  - Host Page Tables
  - Physical Memory
- Processes 1 and 2
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 heterogeneous 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**
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
VBI: Overview

Conventional Virtual Memory

- Virtual Address Space (VAS)
  - Page Tables managed by the OS
  - Physical Memory

VBI

- VBI Address Space
  - Memory Translation Layer in the memory controller
  - Physical Memory
Virtual Blocks

• Globally-visible *VBI address space*

![Diagram of Virtual Blocks and Memory Translation Layer](image)

- Processes
- Memory Translation Layer in the memory controller
- Physical Memory
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

- **VBI addresses** are *directly* used to access on-chip caches
  - No longer require address translation

- **Pros:**
  - Enables inherently virtual caches
    - no synonyms and homonyms
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

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

Processes

P_1 P_2 . . . P_n

VBI Address Space

Memory Translation Layer in the memory controller

Physical Memory
Decoupled Protection and Translation

Address mapping

Access permissions managed by OS

VBI

Memory Translation Layer in the memory controller

VBI Address Space

Processes

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
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
  - Inherently virtual caches
  - Delayed physical memory allocation
  - Eliminating 2D page walks in virtual machines
  - Early memory reservation mechanism
Example Optimizations

- Inherently virtual caches
- Delayed physical memory allocation
- Eliminating 2D page walks in virtual machines
Inherently Virtual Caches

In Conventional Virtual Memory

virtually-indexed physically-tagged (VIPT)

SAFARI
In Conventional Virtual Memory:

- Virtual Indexed Physically Tagged (VIPT) cache
- Translation Lookaside Buffer (TLB)
- Synonyms & Homonyms cache
- Permission check
- Page walk on miss

In VBI:

- No Synonyms & Homonyms cache
- Permission check on miss
- System-wide unique address

VBI reduces address translation overhead by enabling benefits akin to VIVT caches.
Example Optimizations

- Inherently virtual caches
- Delayed physical memory allocation
- Eliminating 2D page walks in virtual machines
Delayed Physical Memory Allocation

In VBI
- virtually-indexed
- physically-tagged (VIPT)

In Conventional Virtual Memory
- Delayed Physical Memory Allocation
- Core
  - Virtual address
  - TLB
    - Hit
      - Physical address
    - Miss
      - Physical Memory
        - Actual cache line
        - Miss page walk

SAFARI
Delayed Physical Memory Allocation

In Conventional Virtual Memory

- Virtually-indexed physically-tagged (VIPT)
- Physical Memory
- TLB
- Core
- Virtual address
- Hit/miss
- Physical address
- Miss page walk
- Actual cache line

In VBI

- VBI Cache
- Core
- VBI address
- Write back
- Allocates memory
- Physical Memory

SAFARI
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**
Example Optimizations

• Inherently virtual caches

• Inherently virtual caches

• Eliminating 2D page walks in virtual machines
Eliminating 2D Page Walks in Virtual Machines

Conventional virtual memory

guest virtual – to – host virtual

host virtual – to – host physical

Process running on a virtual machine (VM)

Guest OS

Guest Virtual Address Space

Guest Page Tables

---- virtualization layer ----

Host OS

Host Virtual Address Space

Host Page Tables

Physical Memory
Eliminating 2D Page Walks in Virtual Machines

Conventional virtual memory

Guest virtual – to – host virtual

host virtual – to – host physical
Guest OS and host OS interact once to attach Process 1 to its VBs.

MTL is the only component in the system that manages address mapping.
Guest OS and host OS interact once to attach Process 1 to its VBs.

By eliminating 2D page walks, VBI reduces address translation overhead in virtualized environments.
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](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
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

- astar
- bzip2
- GemsFDTD
- mcf
- milc
- namd
- sjeng
- bwaves-17
- deepsjeng-17
- lbm-17
- omnetpp-17
- img-dnn
- moses
- Graph500

Speedup normalized to Native

Virtual, Perfect TLB, VBI-Full

0.7x AVG
Use Case 1: Address Translation

Normalized to Native

1.9x

Normalized to Native

SAFARI
Use Case 1: Address Translation

![Graph showing speedup normalized to Native for various applications.]

- astar
- bzip2
- GemsFDTD
- mcf
- milc
- namd
- sjeng
- bwaves-17
- deepsjeng-17
- ibm-17
- omnetpp-17
- img-dnn
- moses
- Graph500

Speedup

Virtual
Perfect TLB
VBI-Full

Normalized to Native

2.4x

AVG
Use Case 1: Address Translation

Normalized to Native

Speedup

4.3x

Virtual
Perfect TLB
VBI-Full

ASTAR
bzip2
GemsFDTD
mcf
milc
namd
sieng
bwaves-17
deesjeng-17
ibm-17
omnetpp-17
img-dnn
moses
Graph500

Use Case 1: Address Translation

Normalized to Native

Speedup

4.3x

Virtual
Perfect TLB
VBI-Full

ASTAR
bzip2
GemsFDTD
mcf
milc
namd
sieng
bwaves-17
deesjeng-17
ibm-17
omnetpp-17
img-dnn
moses
Graph500
Use Case 1: Address Translation

<table>
<thead>
<tr>
<th>Application</th>
<th>Virtual Speedup</th>
<th>Perfect TLB Speedup</th>
<th>VBI-Full Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>0.0</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.0</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>GemsFDTD</td>
<td>0.0</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>mcf</td>
<td>13.3</td>
<td>13.3</td>
<td>13.3</td>
</tr>
<tr>
<td>milc</td>
<td>8.9</td>
<td>8.9</td>
<td>8.9</td>
</tr>
<tr>
<td>namd</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>bwaves-17</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>deepsjeng-17</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>lbm-17</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>omnetpp-17</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>img-dnn</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>moses</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Graph500</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Normalized to Native

49%
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.

**VBI PCM-DRAM vs. IDEAL**

- Normalized to Hotness-Unaware PCM–DRAM
- 33% Speedup

**Applications**:
- astart
- bzip2
- GemsfDTD
- hmmer
- mcf
- milc
- soplex
- sphinx3
- bwaves-17
- lbm-17
- omnetpp-17
- xalancbmk-17
- img-dnn
- moses
- Graph500

**Normative Data**

- VBI PCM-DRAM: Orange bars
- IDEAL: Black bars

**SAFARI**
## Outline

<table>
<thead>
<tr>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBI: Virtual Block Interface</td>
</tr>
<tr>
<td>Key Idea &amp; Guiding Principles</td>
</tr>
<tr>
<td>Design Overview</td>
</tr>
<tr>
<td>Optimizations Enabled by VBI</td>
</tr>
</tbody>
</table>

| Methodology |

| Results |

| Summary |
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
The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

Nastaran Hajinazar*† Pratyush Patel✉ Minesh Patel* Konstantinos Kanellopoulos* Saugata Ghose†
Rachata Ausavarungnirun○ Geraldo F. Oliveira* Jonathan Appavoo♦ Vivek Seshadri▼ Onur Mutlu*†

*ETH Zürich †Simon Fraser University ✉University of Washington ‡Carnegie Mellon University
○King Mongkut’s University of Technology North Bangkok ♦Boston University ▼Microsoft Research India

Published in ISCA 2020

Link to the paper
Rethinking Virtual Memory

Nastaran Hajinazar

Seminar in Computer Architecture
31 March 2022