Today’s Lecture

**VICTIMA**
Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources

**UTOPIA**
Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings

Presented in MICRO 2023
Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings

Presented at MICRO 2023

Konstantinos Kanellopoulos
Rahul Bera, Kosta Stojiljkovic, Nisa Bostanci, Can Firtina,
Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar,
Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu
**Executive Summary**

**Problem:** Conventional virtual memory (VM) frameworks enable a virtual address to flexibly map to any physical address. This flexibility necessitates large translation structures leading to:

1. high translation latency and
2. large translation-induced interference in the memory hierarchy

**Motivation:** Restricting the address mapping leads to compact translation structures and reduces the overheads of address translation. Doing so across the entire memory has two major drawbacks:

1. Limits core VM functionalities (e.g., data sharing)
2. Increases swapping activity in the presence of free physical memory

**Key Idea:** Utopia is a new hybrid virtual-to-physical address mapping scheme that allows both flexible and restrictive hash-based address mappings to harmoniously co-exist in the system.

Utopia manages physical memory using two types of physical memory segments:

- **Restrictive Segment**
  - Modulo Hash Function
  - Limited VM features

- **Flexible Segment**
  - X86-64 Radix PT Table
  - Supports all VM features

**Key Results:** Outperforms (i) the state-of-the-art contiguity-aware translation scheme by 13%, and (ii) achieves 95% of the performance of an ideal perfect-TLB.

[https://github.com/CMU-SAFARI/Utopia](https://github.com/CMU-SAFARI/Utopia)
Talk Outline

- Virtual Memory Background
- Address Translation Overheads
- Utopia: Hybrid Address Mappings
- Utopia: Key Challenges
- Evaluation Results
Talk Outline

Virtual Memory Background

Address Translation Overheads

Utopia: Hybrid Address Mappings

Utopia: Key Challenges

Evaluation Results
Virtual Memory Basics

Virtual Memory (VM) is one of the **cornerstones** of most modern computing systems.

**Conventional VM designs** provide:

1. Application-transparent **memory management**
2. Data **sharing**
3. Process **isolation**
A core feature of virtual memory management is that the mapping between virtual-to-physical pages is fully-associative.

Perform Page Table Walk (PTW) to retrieve the mapping.
Page Table Walk in x86-64

Virtual Address

PL4 9 bits

PL3 9 bits

PL2 9 bits

PL1 9 bits

CR3

Physical Frame Number
Four sequential memory accesses during a page table walk in x86-64
Architectural Support for VM

- To send a memory request to the memory hierarchy, the processor needs to translate the virtual address to the corresponding physical.

- Resolving an address translation request requires accessing the Page Table.

- Modern processors employ a specialized hardware unit called Memory Management Unit (MMU) to accelerate address translation.
Address Translation Flow (I)

Core

Virtual Address

Memory Management Unit

Page Table

Memory Hierarchy

Handles address translation
Address Translation Flow (II)

Memory Management Unit

- L1 I-TLB
- L1 D-TLB
- Unified L2 TLB
- Page Walk Caches
- Page Table Walker
- Page Table
- Memory Hierarchy

Stores intermediate PT entries

Store recent PTEs

State machine that looks up the PT
Address Translation Flow (III)

Memory Management Unit

Page Table Entry

1  PPN  Metadata

PA

Cache Hierarchy
Address Translation Flow (III)

Memory Management Unit

Page Table Entry

0  Swap  Metadata

Physical Address Space

DRAM

Swap Space

Swap in

Page Fault

PA
Address Translation Flow (III)

Translation Latency

Core

Virtual Address

Hit in TLBs

Memory Management Unit

Page Table

Memory Hierarchy
Translation Latency

Core → TLB Miss → Cache/DRAM Access → Page Table

- Address Translation Flow (III)
- Memory Hierarchy
- Virtual Address
- Translation Latency
- Core
- Memory Management Unit
Address Translation Flow (IV)

Core

Physical Address

Memory Hierarchy
Talk Outline

- Virtual Memory Background
- Address Translation Overheads
- Utopia: Hybrid Address Mappings
- Utopia: Key Challenges
- Evaluation Results
Data-Intensive Workloads

- Generative AI
- Graph Analytics
- Bioinformatics

High address translation overheads
Address Translation Overhead

High-latency page table walks

Frequent page table walks
86 cycles on average to access the state-of-the-art hash-based PT
Frequent PTWs

Even the largest 64K-entry L2 TLB experiences 24 MPKI on average
Address Translation Overhead

High latency PTWs + Frequent PTWs

High performance overheads

High interference in memory hierarchy
High Performance Overhead

Completely avoiding address translation leads on average to 31% higher performance

High Interference in Main Memory

![Normalized Row Buffer Conflicts](image)

**State-of-the-art Page Table**

**Perfect TLB**

- Completely avoiding address translation leads to 30% fewer DRAM row buffer conflicts

Idea: Restricting VA-to-PA Mapping

Restrict the VA-to-PA mapping to perform fast address translation$^{1,2}$

Virtual Address Space

- ..
- 00101
- 00110
- 00111
- ..

Physical Address Space

- 00000
- 00000
- 00010
- ..
- ..

[1] Picorel et al. “Near-Memory Address Translation” PACT 2017
Idea: Restricting VA-to-PA Mapping

Restrict the VA-to-PA mapping to perform fast address translation$^{1,2}$

Virtual Address Space

| .. | 00101 | 00110 | 00111 | .. |

Physical Address Space

| .. | Hash Mask 11100 | .. | 00010 | .. |

Drawbacks of Restricting VA-to-PA Mapping

Empowering a restrictive mapping across the entire memory comes with two key drawbacks:

1. Limits core **VM functionalities** such as data sharing

2. Increases **swapping activity** since the system cannot map virtual pages to free physical pages
Sole use of restrictive mapping leads to 2.37x higher swapping activity over the baseline
Our Goal

Design a virtual-to-physical address mapping scheme that:

• Provides fast and efficient translation using a restrictive hash-based address mapping

• Enjoys the benefits of the conventional fully-flexible address mapping
Utopia: Key Idea

We propose **Utopia**, a new virtual-to-physical mapping scheme that enables both:

- **Restrictive Mapping**
- **Flexible Mapping**

Harmoniously co-exist in the system
Manage physical memory using two types of physical memory segments:

- **Restrictive Segments**
- **Flexible Segments**
Utopia: Key Idea (I)

Restrictive Segment (RestSeg)

- Fast address translation
- Limited VM functionalities

Flexible Segments

Hash function

Virtual Page Number

Page  Page  Page  Page
Utopia: Key Idea (II)

Flexible Segment (FlexSeg)

- Supports all conventional VM features
- High-latency address translation

Restrictive Segments
RestSeg Properties

**Structural Properties**

*Address Translation for Data in RestSeg*
RestSeg Properties

*Structural Properties*

RestSeg is organized in a *set-associative* manner similar to how hardware caches operate.
Example: 2-way associative RestSeg with 2 sets

Set-associative design offers high flexibility
Multiple RestSegs in the System

RestSeg #1
4KB Page 4KB Page 4KB Page 4KB Page

RestSeg #2
2MB Page 2MB Page

Backward compatible with large page mechanisms
RestSeg Properties

Structural Properties

Address Translation for Data in RestSeg
Restrictive Segment Walk (RSW)

RestSeg

Page

Way 0

Way 1

Set 0

Way 0

Way 1

Set 1

Virtual Page
How can we find out the physical location of the virtual page?
RSW Operations

1. Tag Matching
2. Set Filtering
RestSeg: Tag Matching

- **Tag matching** requires comparing the tags of all ways with the tag of the virtual page.
- **Tag Array (TAR):** Array that stores the tag of each entry.

Do we always have to do tag matching?
RestSeg: Set Filtering

- **Set Filtering**: quickly discover if a set in the RestSeg is empty or not and filter tag mismatches

- **Set Filter (SF)**: Array of counters that keep track of the number of pages inside each set

![Diagram](image-url)
Address Translation in Utopia

System employs:
- 2 RestSegs, one for 4KB and one for 2MB pages
- 1 FlexSeg

```
L1 TLB Miss
```

```
L2 TLB Access
```

```
RestSeg Walks
```

```
Page Table Walk
```

Talk Outline

- Virtual Memory Background
- Address Translation Overheads
- Utopia: Hybrid Address Mappings
- Utopia: Key Challenges
- Evaluation Results
Utopia: Key Challenges

1. Which data should be placed in the RestSeg?

2. How to maintain RestSegs in the system

3. How to integrate Utopia in the address translation pipeline
Utopia: Key Challenges

1. Which data should be placed in the RestSegs?

2. How to maintain RestSegs in the system

3. How to integrate Utopia in the address translation pipeline
Page Placement in RestSeg

Our **goal** is to place costly-to-translate pages into a RestSeg.

We propose **two techniques** to perform **data placement in Utopia**:

- **Page-Fault-based Allocation Policy**
- **PTW-Tracking-based Migration Policy**
Page-Fault-Based Allocation Policy

The OS attempts to allocate the page directly in the RestSeg

- **Page Fault**
  - **Occupancy < Threshold**
    - Yes: Allocate in RestSeg
    - No: Allocate in FlexSeg
What about costly-to-translate pages that cannot be directly allocated in a RestSeg?
PTW-Tracking-Based Migration Policy

Use unused bits of each PTE as a counter that tracks the number and cost (i.e., DRAM accesses) of PTWs.

- Physical Frame #
- Metadata
- Counters

Unused bits > Threshold

RestSeg

Migrate regardless of occupancy
Utopia: Three Key Challenges

1. Which data should be placed in the RestSegs?

2. How to maintain RestSegs in the system

3. How to integrate Utopia in the address translation pipeline
OS Support for Utopia

OS supports Utopia in three ways by handling:

1. **Allocations** in a RestSeg
2. **Replacements** in a RestSeg
3. **Migrations** to/from a RestSeg
OS: Allocation in RestSeg

There is a free way! Store data there.
OS: Allocation in RestSeg

Discover if there is a free way in the set
OS: Replacement in RestSeg

VPN

Hash Function

Global Tag Array

No way is free! Evict a page!
OS: Migration to/from RestSeg

DMA engine is responsible for migrating data between RestSegs and FlexSegs.
Utopia: 3 Key Challenges

• Which data should be placed in the RestSegs?

• How to maintain RestSegs in the system

• How to integrate Utopia in the address translation pipeline
Architectural Support for Utopia

New **RestSeg walker** enables address translation for pages residing in a RestSeg:

- **Specialized hardware circuitry to perform the tag matching and the set filtering**
  - Avoid expensive software-based accesses to translation structures

- **Small caches for the the Tag Array and Set Filter**
  - Accesses to Permissions Filter and Tag Array may exhibit **high spatial and temporal locality**

- **Minor modifications** in the address translation pipeline:
  - RSW occurs in parallel with L2 TLB access
RestSeg Walker in MMU

**RestSeg Walker**

- **FSM**: Generates TAR and SF addresses based on VA
- **TAR Cache**: SRAM structures store recent TAR/SF entries
- **SF Cache**: Memory Hierarchy
  - **TAR**
  - **SF**
RestSeg Walker in MMU (II)

Parallel Ops

L1 TLB Miss

L2 Unified TLB

FSM

TAR Cache

SF Cache

Wait for RSW result before starting a FlexSeg Walk

Mis

Miss

SAFARI
Page resides inside a RestSeg
Address Translation Flow (1)

RestSeg Walker

Parallel Ops

L1 TLB Miss

FSM

TAR Cache

SF Cache

Data is in RestSeg

L2 Unified TLB

Miss

Stall

Abort FlexSeg Walk
Address Translation Flow (2)

2 Page resides inside a FlexSeg

FlexSeg

RestSeg

Physical Memory
Address Translation Flow (2)

RestSeg Walker

Parallel Ops

L1 TLB Miss

L2 Unified TLB

Mis

TAR Cache

SF Cache

Data is not in RestSeg

Start FlexSeg Walk
Address Translation Flow (3)

3 TLB hit

FlexSeg | RestSeg

Physical Memory
Address Translation Flow (3)

Parallel Ops

L1 TLB

Hit

Miss

Abort RSW

L2 Unified TLB

Hit

RestSeg Walker

FSM

TAR Cache

SF Cache
Area & Power Overhead

- Area and power overhead evaluation using McPat
- Comparison to a high-end Intel Raptor Lake

Utopia incurs 0.64% area and 0.72% power overhead per core
Talk Outline

Virtual Memory Background

Address Translation Overheads

Utopia: Hybrid Address Mappings

Utopia: Key Challenges

Evaluation Results
Evaluation Methodology

- Sniper Multicore Simulator extended with:
  - Page table walker & page walk caches
  - Buddy allocator
  - Migration Latency

Our poster at MICRO 2023 SRC introduces a new open-source simulation framework for VM research

https://github.com/CMU-SAFARI/Virtuoso
Evaluation Methodology

• Sniper Multicore Simulator extended with:
  • Page table walker & page walk caches
  • Buddy allocator
  • Migration Latency

• Workloads: **Executed for 500M instructions**
  
  • **GraphBIG**: PR, BFS, BC, GC, CC
  • **HPCC**: Randacc
  • **XSBench**: Particle Simulation with 15K grid
  • **DLRM**: SLS-like
  • **GenomicsBench**: k-mer count
Evaluated Mechanisms

- Baseline with **Radix PT** and **Transparent Huge Pages** enabled
- **POM-TLB**¹: State-of-the-art large software-managed TLB
- **ECH**²: State-of-the-art hash-based page table
- **RMM**³: Contiguity-aware address translation
- **Utopia**: 512MB RestSegs (one for 4KB and one for 2MB pages)
- **P-TLB**: Perfect L1 TLB (translation requests always hit in L1 TLB)

---

Utopia outperforms the second-best performing scheme (RMM) by 13% and Radix by 24%
Performance Results

Utopia outperforms the second-best performing scheme (RMM) by 13% and Radix by 24%

Utopia’s performance is within 95% of P-TLB
Translation Latency

Utopia reduces average translation latency by 63% over Radix and 14% over RMM
Main Memory Interference

Utopia reduces row buffer conflicts by 20% over Radix and 9% less than P-TLB
More Results and Details in the Paper

- Sensitivity across different hash functions
- Sensitivity to parallel/serial TLB access and RSW
- Sensitivity to RestSeg size
- Reuse-level distribution of 4KB pages
- Effect of migration to memory requests
- TAR & SF cache hit rate
- Overhead across different context-switch quanta

https://arxiv.org/abs/2211.12205
More Results and Details in the Paper

- Sensitivity across different hash functions
- Sensitivity to parallel/serial TLB access and RSW
- Sensitivity to RestSeg size
- Reuse-level distribution of 4KB pages
- Effect of migration to memory requests
- TAR & SF cache hit rate
- Overhead across different context-switch quanta

Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings

Konstantinos Kanellopoulos1 Rahul Bera1 Kosta Stojiljkovic1 Nisa Bostanci1 Can Firtina1 Rachata Ausavarungnirun2 Rakesh Kumar3 Nastaran Hajinazar4 Mohammad Sadrosadati1 Nandita Vijaykumar5 Onur Mutlu1

1ETH Zürich 2King Mongkut’s University of Technology North Bangkok 3Norwegian University of Science and Technology 4Intel Labs 5University of Toronto

Abstract

Conventional virtual memory (VM) frameworks enable a virtual address to flexibly map to any physical address. This flexibility necessitates large data structures to store virtual-to-physical mappings, which leads to high address translation latency and large translation-induced interference in the memory hierarchy, especially in data-intensive workloads. On the other hand, restricting the address mapping so that a virtual address can only map to a specific set of physical addresses can significantly reduce address translation overheads by making use of compact and efficient translation structures. However, restricting the address mapping flexibility across the entire main memory severely limits data sharing across different processes and increases data accesses to the swap space of the storage device even in the presence of free memory.

We propose Utopia, a new hybrid virtual-to-physical address mapping scheme that allows both flexible and restrictive hash-based address mapping schemes to harmoniously co-exist in the system. The key idea of Utopia is to manage physical memory using two types of physical memory segments: restrictive segments and flexi-

1 Introduction

Virtual memory (VM) serves as a foundational element in most computing systems, simplifying the programming model by offering an abstraction layer over physical memory [2–24]. In the presence of VM, the operating system (OS) maps each virtual address to its corresponding physical memory address to facilitate application-transparent memory management, process isolation, and memory protection. The virtual-to-physical mapping scheme in conventional VM frameworks allows a virtual address to flexibly map to any physical address. This flexibility enables key VM functionalities, such as (i) data sharing between processes while maintaining process isolation and (ii) avoiding frequent swapping (i.e., avoiding storing data in the swap space of the storage device in the presence of free main memory space). However, a flexible mapping scheme requires mapping metadata for every virtual address and its corresponding physical address, which is stored in the page table (PT). As shown in multiple prior works [25–35], data-intensive workloads do not efficiently use translation-dedicated hardware structures and the processor performs frequent PT accesses, i.e., a process called

https://arxiv.org/abs/2211.12205
Conclusion

We propose **Utopia**, a new virtual-to-physical mapping scheme that enables both:

- **Restrictive Mapping**
- **Flexible Mapping**

Harmoniously co-exist in the system

Utopia achieves (i) **13% higher performance** than the state-of-the-art **contiguity-aware translation scheme** and (ii) **95% of the performance** of an ideal **perfect-TLB**

**Utopia is open source**

[https://github.com/CMU-SAFARI/Utopia](https://github.com/CMU-SAFARI/Utopia)
Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings

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

Konstantinos Kanellopoulos
Rahul Bera, Kosta Stojiljkovic, Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu

SAFARI
ETH Zürich

Intel
University of Toronto
NTNU
VICTIMA
Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources

Presented at MICRO 2023

Konstantinos Kanellopoulos
Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini and Onur Mutlu
Executive Summary

**Problem:** Address translation is a major **performance bottleneck** in data-intensive workloads. Large datasets and irregular memory access patterns lead to **frequent L2 TLB misses** (e.g., 20-50 MPKI) and **frequent high-latency** (e.g., 100-150 cycles) page table walks (PTW).

**Motivation:** Increasing the translation reach (i.e., memory covered by the TLBs) reduces PTWs. However, employing large TLBs leads to increased area, power and latency overheads.

**Opportunity:** Increase the translation reach of the TLB hierarchy by storing the existing TLB entries within the **existing cache hierarchy**.

**Victima:** **New software-transparent scheme** that drastically increases the address translation reach of the processor’s TLB hierarchy by leveraging the underutilized cache resources.

**Key Idea:**
- Transform L2 cache blocks that store PTEs into blocks that store TLB entries.

**Key Benefits:**
- **Efficient** in native/virtualized environments
- **Fully transparent** to application/OS software
- **Compatible** with huge page schemes

**Key Results:** Victima (i) outperforms by **5.1%** a state-of-the-art large TLB design and (ii) achieves **similar performance** to an optimistically fast 128K-entry L2 TLB.

*SAFARI*

[https://github.com/CMU-SAFARI/Victima](https://github.com/CMU-SAFARI/Victima)
Talk Outline

Background & Motivation

Opportunity: Leverage Caches

Victima: Overview

Victima: Detailed Design

Evaluation Results
Virtual Memory Basics

• The **Page Table (PT)** stores all virtual-to-physical address mappings

• The x86-64 PT is organized as a **4/5-level radix tree**

• To access the PT, the system performs a **Page Table Walk (PTW)**
Page Table Walk in x86-64

Virtual Address

- PL4: 9 bits
- PL3: 9 bits
- PL2: 9 bits
- PL1: 9 bits

CR3

Physical Frame Number
Page Table Walk in x86-64

Four sequential memory accesses during a page table walk in x86-64
Address Translation Flow (I)

Core → Virtual Address → Memory Management Unit → Page Table → Memory Hierarchy
Address Translation Flow (II)

Memory Management Unit

L1 I-TLB

Virtual Address

L1 D-TLB

Unified L2 TLB

Page Walk Caches

Page Table Walker

Page Table

Memory Hierarchy

Miss

Miss

Miss

Miss

Virtual Address
Core spends 137 cycles on average to perform a PTW
Address Translation Overhead

**High latency PTWs** + **Frequent PTWs** → **High performance overheads**
Potential Solution

Reduce PTW frequency by increasing address translation reach

High latency PTWs + Frequent PTWs

High performance overheads
Address Translation Reach: Definition

Amount of VA-to-PA mappings stored by the processor’s TLB hierarchy

Example Modern Processors: Maximum 3-4GB

Increase Reach
Reduce PTWs
Increasing Address Translation Reach

Large **Hardware** TLBs
Scaling Hardware L2 TLB (I)

Employing a 64K-entry L2 TLB reduces MPKI from 39 to 24
Scaling Hardware L2 TLB (II)

64K-entry L2 TLB with optimistic access latency provides 5.4% speedup over baseline
Scaling Hardware L2 TLB (II)

64K-entry L2 TLB with optimistic access latency provides 5.4% speedup over baseline.

Benefits come for free?
64K-entry L2 TLB with realistic access latency provides only 0.8% speedup over baseline.
Increasing Translation Reach

Large Hardware TLBs

Large Software-Managed TLBs
Large Software-Managed L3 TLB

MMU

L2 TLB

Main Memory

Software L3 TLB

Contiguous

Miss
Drawbacks of Software-Managed TLB

1. High Latency
2. Contiguous Physical Allocations
3. OS Modifications
Increasing Translation Reach

| Large Hardware TLBs | Large Software-Managed TLBs |

Both approaches come with major drawbacks
Talk Outline

- Background & Motivation
- Opportunity: Leverage Caches
- Victim: Overview
- Victim: Detailed Design
- Evaluation Results
Opportunity: Leverage Caches

Store TLB entries in hardware caches
Leverage Cache Hierarchy
Where is the Benefit?

**L2 TLB**
- 1.5K entries
- 12-cycle latency

**2MB L2 Cache**
- Fits 36x more TLB entries
- Low latency (e.g., 16 cycles)

*PTW takes 137 cycles on average*
Interference with Program Data?

Breakdown of L2 Data Block Reuse

L2 cache is heavily underutilized
Talk Outline

- Background & Motivation
- Opportunity: Leverage Caches
- Victima: Overview
- Victima: Detailed Design
- Evaluation Results
Our Goal

Leverage cache resources to store TLB entries

Drastically increase the address translation reach of the processor
Victima: Key Idea

Repurpose L2 cache blocks to store clusters of TLB entries

Low-latency and high-capacity component to back up the L2 TLB
Victima: Overview

Costly-to-translate page?

MMU

L2 Cache

L2
TLB

Miss
Eviction

PTW Cost Estimator

PTE Block

TLB Block

Transform

Miss

Low latency

SAFARI
Victima Benefits

+ **Drastic increase** in address translation reach

+ **Fully transparent** to application/OS software

+ **No need** for contiguous physical allocations

+ **Compatible** with huge pages
Talk Outline

- Background & Motivation
- Opportunity: Leverage Caches
- Victimia: Overview
- Victimia: Detailed Design
- Evaluation Results
1. L2 Cache Modifications

2. Allocation of TLB Entries in L2 Cache

3. Page Table Walk Cost Predictor
Victima: L2 Cache Modifications

1. Access TLB blocks using virtual address

2. Perform tag matching for TLB blocks
Example: Cache Configuration

L2 Cache

1MB
16-way associative

Set index  10 bits
Data Blocks vs. TLB Blocks in Caches

**Data Block**
- **TLB Entry**: 0
- **Tag**: 36 bits
- **Data**: 64 bytes

**52-bit Physical Address**
- **Tag**: 36 bits
- **Set index**: 10 bits
- **Offset**: 6 bits

**TLB Block**
- **TLB Entry**: 1
- **Tag**: 23 bits
- **ASID/Size**: 13 bits

**PTEs (8 bytes per PTE)**
- **0**: 1 2 3 4 5 6 7

**36-bit Virtual Page Number (4KB)**
- **Tag**: 23 bits
- **Set index**: 10 bits
- **Offset**: 3 bits
Tag Matching for TLB Block

**TLB Block**

- **TLB Entry**: 1
- **Tag**: 23 bits
- **ASID**: 9 bits
- **PTEs**: 0 1 2 3 4 5 6 7

**Virtual Address**

- **Tag**: 23 bits
- **ASID**: 9 bits
- **Offset**: 3 bits
Victima: L2 Cache Modifications

1. L2 Cache Modifications

2. Allocation of TLB Entries in L2 Cache

3. Page Table Walk Cost Predictor
Allocation of TLB Entries in L2 Cache

1. On L2 TLB Miss

2. On L2 TLB Eviction
Allocation of TLB Entries in L2 Cache

1. On L2 TLB Miss

2. On L2 TLB Eviction
Allocating TLB Blocks – L2 TLB Miss

- MMU
  - L2 TLB
  - Miss
  - Page Table Walker
- PTW Cost Estimator
- L2 Cache
  - PTE Block
  - Transform
  - TLB Block
Allocation of TLB Entries in L2 Cache

1. On L2 TLB Miss

2. On L2 TLB Eviction
Allocating TLB Blocks – L2 TLB Eviction

MMU

L2 TLB

Eviction

Page Table Walker

PTW Cost Estimator

L2 Cache

PTE Block

Transform

TLB Block
Address Translation in Victima (I)

1. L2 TLB
2. MMU
3. TLB Block
4. L2 Cache
5. Page Table Walker
6. Hit ☺️
7. Miss

SAFARI
Address Translation in Victim (II)

- MMU
  - L2 TLB
    - Page Table Walker
  - Miss
- L2 Cache
  - Miss

Safari
Victima: Detailed Design

1. L2 Cache Modifications

2. Allocation of TLB Blocks in L2 Cache

3. Page Table Walk Cost Predictor
PTW Cost Predictor: Objective

Predict which pages are costly-to-translate
Insert only those TLB blocks in L2 cache
Tracking Costly-to-Translate Pages
PTW Cost Predictor (PTW-CP)

- PTW Frequency
- PTW Cost
- Bypassing Logic based on L2 cache MPKI

Comparator Tree

Costly-to-translate page?

Costly-to-translate page!
Feature engineering to find minimal set of useful features

2-feature comparator predicts costly-to-translate pages with 82% accuracy
## PTW-CP Feature Set

<table>
<thead>
<tr>
<th>Feature (per PTE)</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size</td>
<td>1</td>
<td>The size of the page (4KB or 2MB)</td>
</tr>
<tr>
<td>Page Table Walk Cost</td>
<td>3</td>
<td>DRAM accesses during a PTW</td>
</tr>
<tr>
<td>Page Table Walk Frequency</td>
<td>3</td>
<td>The number of PTWs</td>
</tr>
<tr>
<td>LLPWC Hits</td>
<td>5</td>
<td>The number of third-level PWC hits</td>
</tr>
<tr>
<td>L1 TLB Misses</td>
<td>5</td>
<td>The number of L1 TLB misses</td>
</tr>
<tr>
<td>L2 TLB Misses</td>
<td>5</td>
<td>The number of L2 TLB hits</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>5</td>
<td>The number of L2 cache hits</td>
</tr>
<tr>
<td>L1 TLB Evictions</td>
<td>5</td>
<td>The number of L1 TLB evictions</td>
</tr>
<tr>
<td>L2 TLB Evictions</td>
<td>6</td>
<td>The number of L2 TLB evictions</td>
</tr>
<tr>
<td>Accesses</td>
<td>6</td>
<td>The number of accesses to the page</td>
</tr>
</tbody>
</table>

**Feature engineering to find minimal set of useful features**

SAFARI
PTW-CP Exploration

<table>
<thead>
<tr>
<th>Feature</th>
<th>NN-10</th>
<th>NN-5</th>
<th>NN-2</th>
<th>Comparator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Size</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>Size of Hidden Layers</td>
<td>16</td>
<td>64</td>
<td>4</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of Neurons</td>
<td>737</td>
<td>8769</td>
<td>97</td>
<td>N/A</td>
</tr>
<tr>
<td>Size (B)</td>
<td>5896</td>
<td>70152</td>
<td>776</td>
<td>24</td>
</tr>
<tr>
<td>Recall</td>
<td>0.9334</td>
<td>0.9244</td>
<td>0.8962</td>
<td>0.8961</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.9213</td>
<td>0.9172</td>
<td>0.8290</td>
<td>0.8290</td>
</tr>
<tr>
<td>Precision</td>
<td>0.8768</td>
<td>0.8747</td>
<td>0.7333</td>
<td>0.7334</td>
</tr>
<tr>
<td>F1-score</td>
<td>0.9042</td>
<td>0.8989</td>
<td>0.8066</td>
<td>0.8066</td>
</tr>
</tbody>
</table>

2-feature comparator predicts costly-to-translate pages with 82% accuracy
Virtualized Environments

Two-level address translation

1. Guest Virtual
2. Host Virtual
3. Host Physical
Virtualized Environments

- L2 TLB
- Nested TLB
- Guest-Virtual
- Host-Physical
- Guest-Virtual
- Host-Virtual
Virtualized Environments

- MMU
  - L2 TLB
    - TLB Entry
  - Nested TLB
    - Nested TLB Entry
  - L2 Cache
Talk Outline

- Background & Motivation
- Opportunity: Leverage Caches
- Victim: Overview
- Victim: Detailed Design
- Evaluation Results
Evaluation Methodology

**Sniper Multicore Simulator** extended with:
- TLB Hierarchy with multiple page sizes
- Radix page table walker
- Page walk caches

[https://github.com/CMU-SAFARI/Victima](https://github.com/CMU-SAFARI/Victima)

**Workloads:** Executed for 500M instructions
- **GraphBIG:** PR, BFS, BC, GC, CC
- **HPCC:** Randacc
- **XSBench:** Particle Simulation
- **DLRM:** Sparse-length sum
- **GenomicsBench:** k-mer counting
Configurations – Native Execution

• **Radix**: Baseline system with 1.5K-entry L2 TLB and Transparent Huge pages enabled

• **Optimistic L2 TLB-64K**: System with 64K-entry L2 TLB (optimistic 12-cycle access latency)

• **Optimistic L2 TLB-128K**: System with 128K-entry L2 TLB (optimistic 12-cycle access latency)

• **POM-TLB¹**: System with 64K-entry software-managed L3 TLB

• **Victima**

---

Victima achieves similar performance to the optimistically fast 128K-entry L2 TLB.
Victima reduces PTWs by 50% on average compared to the baseline.
Effect of L2 Cache Size on Victima

Employing an 8MB L2 cache with Victima reduces PTWs by 63%
Configurations in Virtualized Environments

• **Nested Paging**¹: Baseline system that performs Nested PTWs

• **POM-TLB**²: System with 64K-entry software-managed L3 TLB

• **Ideal Shadow Paging**³: System that employs an ideal version of Shadow Paging

• **Victima**: Caching both TLB and Nested TLB entries in the L2 cache

---

Performance in Virtualized Environments

Victima outperforms 64K-entry software-managed TLB by 12%
Area & Power Overhead

• Area and power overhead evaluation using McPAT
• Comparison to a high-end Intel Raptor Lake

Victima incurs 0.04% area and 0.08% power overheads
More in the paper

• Victima integration in virtualized environments

• Maintenance operations to handle TLB shootdowns

• TLB-Block-aware replacement policy

• Implementation details of PTW cost estimator

• Translation reach provided by Victima

https://arxiv.org/abs/2310.04158
More in the paper

- Victima integration in virtualized environments
- Maintenance operations to handle TLB shootdowns
- TLB-Block-aware replacement policy
- Implementation details of PTW cost estimator
- Translation reach provided by Victima

Victima: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Translation Resources

Konstantinos Kanellopoulos¹ Hong Chul Nam¹ F. Nisa Bostanci¹ Rahul Bera¹ Mohammad Sadrosadati² Rakesh Kumar² Davide Basilio Bartolini³ Onur Mutlu¹

¹ETH Zürich ²Norwegian University of Science and Technology ³Huawei Zurich Research Center

Abstract

Address translation is a performance bottleneck in data-intensive workloads due to large datasets and irregular access patterns that lead to frequent high-latency page table walks (PTWs). PTWs can be reduced by using (i) large hardware TLBs or (ii) large software-managed TLBs. Unfortunately, both solutions have significant drawbacks: increased access latency, power and area (for hardware TLBs), and costly memory accesses, the need for large contiguous memory blocks, and complex OS modifications (for software-managed TLBs).

We present Victima, a new software-transparent mechanism that drastically increases the translation reach of the processor by leveraging the underutilized resources of the cache hierarchy. The key idea of Victima is to repurpose L2 cache blocks to store clusters of TLB entries, thereby providing an additional low-latency and high-capacity component that backs up the last-level TLB and thus reduces PTWs. Victima has two main components. First, a PTW cost predictor (PTW-CP) identifies costly-to-translate addresses based on the frequency and cost of the PTWs they lead to. Leveraging the PTW-CP, Victima uses the valuable cache space only for TLB entries that correspond to costly-to-translate pages, reducing the impact on cached application data. Second, a TLB-aware cache replacement policy prioritizes keeping TLB entries in the cache hierarchy by considering (i) the translation pressure (e.g., last-level TLB miss rate) and (ii) the reuse characteristics of the TLB entries.

Drawback of Large Hardware TLBs. First, a larger TLB has address translations. However, with the very large data footprints of modern workloads, the last-level TLB (L2 TLB) experiences high miss rate (misses per kilo instructions; MPKI), leading to high-latency page table walks (PTWs) that negatively impact application performance. Virtualized environments exacerbate the PTW latency as they impose two-level address translation (e.g., up to 24 memory accesses can occur during a PTW in a system with nested paging [12, 13]), resulting in even higher address translation overheads compared to native execution environments. Therefore, it is crucial to increase the translation reach (i.e., the maximum amount of memory that can be covered by the processor’s TLB hierarchy) to improve the effectiveness of TLBs and thus minimize PTWs. Doing so becomes increasingly important as PTW latency continues to rise with modern processors’ deeper multi-level page table (PT) designs (e.g., 5-level radix PT in the latest Intel processors [4]).

Previous works have proposed various solutions to reduce the high cost of address translation and increase the translation reach of the TLBs such as employing (i) large hardware TLBs [14–16] or (ii) backing up the last-level TLB with a large software-managed TLB [17–25]. Unfortunately, both solutions have significant drawbacks: increased access latency, power, and area (for hardware TLBs), and costly memory accesses, the need for large contiguous memory blocks, and complex OS modifications (for software-managed TLBs).

https://arxiv.org/abs/2310.04158
Victima is Open Source

https://github.com/CMU-SAFARI/Victima
Victima is Open Source

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

Distinguished Artifact Award
Victima is Open Source

Documentation is available

In case of an L2 TLB miss, we check if the TLB entry is cached in the cache hierarchy. If the victim is enabled, in order to do that, we first calculate the address of the cache line that contains the TLB entry. We do that by shifting the address to the right by the page size minus 3 bits (if PTEs are stored in the cache line). We then get the L1 data cache, the L2 cache and the MUCA cache from the memory manager. We then use the cache line that contains the TLB entry from each cache. In the case of Victims, we only need to access the L2 cache and the MUCA cache.
We present Victima, a new software-transparent scheme that drastically increases the translation reach of the processor’s TLB hierarchy by leveraging the underutilized cache resources.

Key idea: Transform L2 cache blocks that store PTEs into blocks that store TLB entries.

Key Results: Victima (i) outperforms by 5.1% a state-of-the-art software-managed TLB and (ii) achieves similar performance to an optimistically fast 128K-entry L2 TLB design without the associated area and power overheads.

https://github.com/CMU-SAFARI/Victima
VICTIMA

Drastically Increasing Translation Reach by Leveraging Underutilized Cache Resources

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

Konstantinos Kanellopoulos
Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini and Onur Mutlu