VICTIMA

Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources

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.

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

Virtual Address

Miss

Miss

Miss

Miss
Address Translation Overhead

Mean: 137 cycles

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
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
Drawbacks of Software-Managed TLB

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

<table>
<thead>
<tr>
<th>Large Hardware TLBs</th>
<th>Large Software-Managed TLBs</th>
</tr>
</thead>
</table>

Both approaches come with major drawbacks
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

- **Reuse 0**: 0%
- **1-5**: 20%
- **5-10**: 40%
- **10-20**: 60%
- **>20**: 100%

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

Miss

Eviction

L2 Cache

PTW Cost Estimator

PTE Block

Transform

TLB Block

Low latency

L2 Cache

Eviction

Miss

Miss

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
Victima: Detailed Design

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

Virtual Address

Tag
23 bits

ASID
9 bits

Offset
3 bits

TLB Entry

Tag
23 bits

ASID
9 bits

PTEs
0 1 2 3 4 5 6 7
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

Page Table Walker

Miss

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

PTW Cost Estimator

Page Table Walker

L2 Cache

PTE Block

TLB Block

Transform
Address Translation in Victima (I)

MMU

L2 TLB

Page Table Walker

Miss

L2 Cache

TLB Block

Hit 😊
Address Translation in Victima (II)

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

Page Table Entry

Counters

Update Counters

Page Table Walker

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
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
Talk Outline

- Background & Motivation
- Opportunity: Leverage Caches
- Victim: Overview
- Victim: Detailed Design
- Evaluation Results
Sniper Multicore Simulator extended with:
- TLB Hierarchy with multiple page sizes
- Radix page table walker
- Page walk caches

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

https://github.com/CMU-SAFARI/Victima
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-TLB\(^1\)**: 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
Employing an 8MB L2 cache with Victima reduces PTWs by 63%
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: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Translation Resources

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

  Konstantinos Kanellopoulos¹  Hong Chul Nam¹  F. Nisa Bostanci³  Rahul Bera¹  Mohammad Sadrosadati¹  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.

  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).

  **Drawback of Large Hardware TLBs.** First, a larger TLB has
Victima is Open Source

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

Documentation is available

TLB Lookup Model
Modifications to the TLB lookup function to implement Victim.

TLB Allocation Model
Modifications to the TLB allocation function to implement Victim.

```cpp
bool hit = m_cache.accessSingleLineTLB(address, Cache::LOAD, NULL, 0, now, true);

bool l2tlb_hit = true;
if (m_next_level) // is there a second level TLB?
{
    where.next = m_next_level->lookup(address, now, false, 2 /* no allocation */, model_count, lock_signal);
    if (where.next != TLB::MISS)
        l2tlb_hit = false;
} else if (victim_enabled) // No one at L2 TLB
{
    // L2 TLB Miss => check the cache hierarchy to see if TLB entry is cached
    UInt32 set;
    IntPtr tag;
    IntPtr cache_address = address >> (page_size - 3);
    Cache* l1cache = m_manager->getCache(MemComponent::component_t::L1_CACHE);
    Cache* l2cache = m_manager->getCache(MemComponent::component_t::L2_CACHE);
    Cache* nuca = m_manager->getNucaCache()->getCache();

    CacheBlockInfo* cb_l1 = l1cache->peekSingleLine(cache_address);
    CacheBlockInfo* cb_l2 = l2cache->peekSingleLine(cache_address);
    CacheBlockInfo* cb_nuca = nuca->nuca->peekSingleLine(cache_address);

    if (cb_l1 && cb_l2 && cb_nuca)
    {
        result = true;
        // Do something with the result...
    }
    else
    {
        // Handle the case where the TLB entry is not found...
    }
}
```
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 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
AddiKg Hardware-based L3 TLB

High access latency offsets the potential performance gains of hardware L3 TLB
More than 60% of TLB blocks experience reuse higher than 20
Employing the TLB-aware DRRIP leads to 1.8% higher performance compared to the conventional DDRIP.
Page Table Walk in X86-64

Four sequential memory accesses during a page table walk in x86-64

Up to 24 memory accesses in virtualized environments
Virtualized Environments

Two-level address translation

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

- **L2 TLB**
  - Host-Physical
  - Guest-Virtual

- **Nested TLB**
  - Host-Virtual
  - Guest-Virtual
Virtualized Environments

- MMU
- L2 TLB
- TLB Entry
- Nested TLB
- Nested TLB Entry
- L2 Cache
# 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
<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
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

---


Reduction in Host and Guest PTWs

Reduction of PTWs

- POM-TLB Guest PTW
- POM-TLB Host PTW
- Victima Guest PTW
- Victima Host PTW

0% - 100%