Artifacts Available V1.1

Artifacts Evaluated Functional V1.1

Artifacts Evaluated Reusable V1.1

Results Reproduced V1.1

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

SAFARI Research Group
safari.ethz.ch

ETH zürich

HUAWEI

NTNU

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)

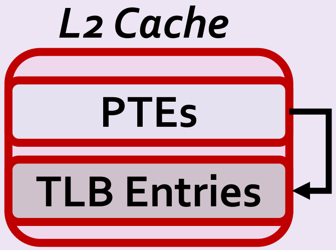
Motivation: Increasing the translation reach (i.e., memory covered by the TLBs) reduces PTWs. However, employing large TLBs leads to increased area, power and latency overheads.

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

Talk Outline

Background & Motivation

Opportunity: Leverage Caches

Victima: Overview

Victima: Detailed Design

Evaluation Results

Talk Outline

Background & Motivation

Opportunity: Leverage Caches

Victima: Overview

Victima: Detailed Design

Evaluation Results

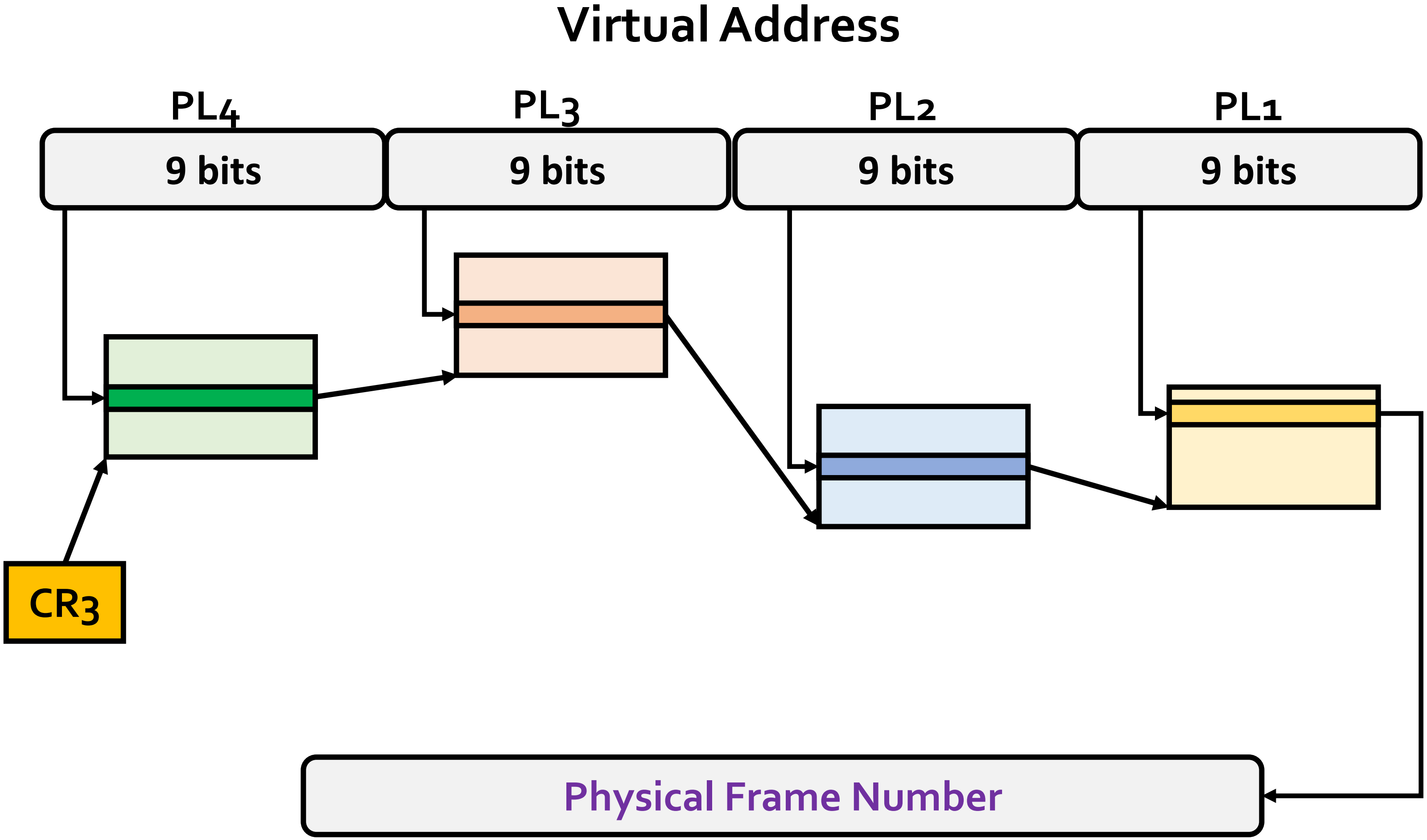
Virtual Memory Basics

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

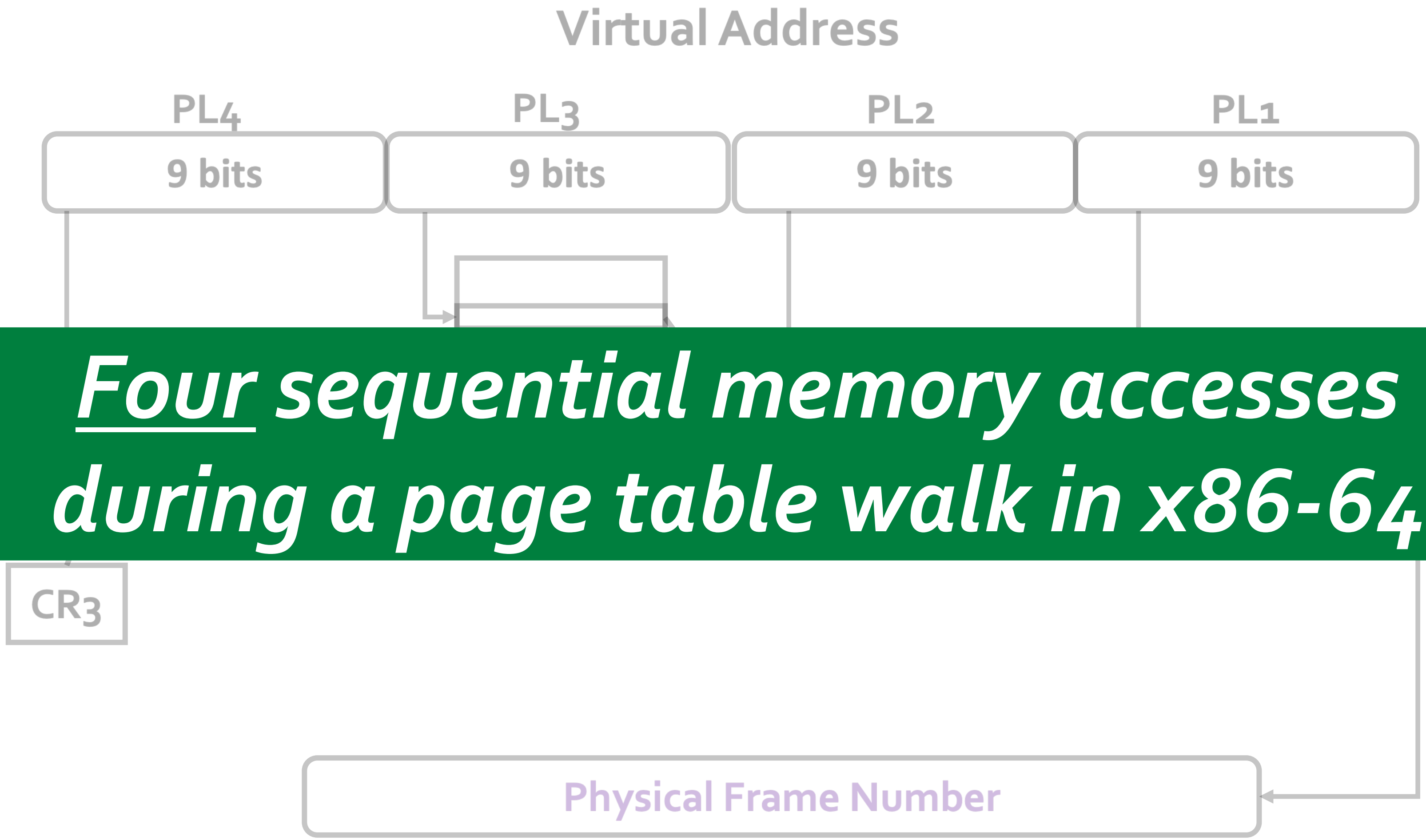
- The x86-64 PT is organized as a **4/5-level radix tree**

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

Page Table Walk in x86-64

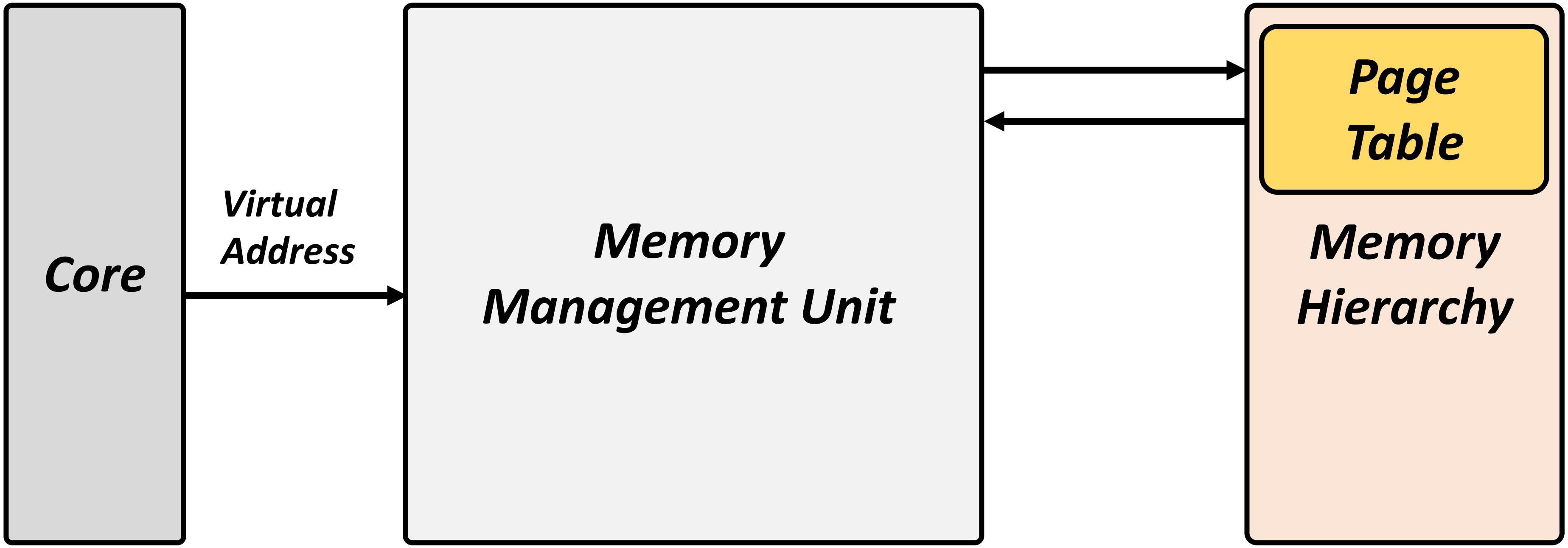


Page Table Walk in x86-64

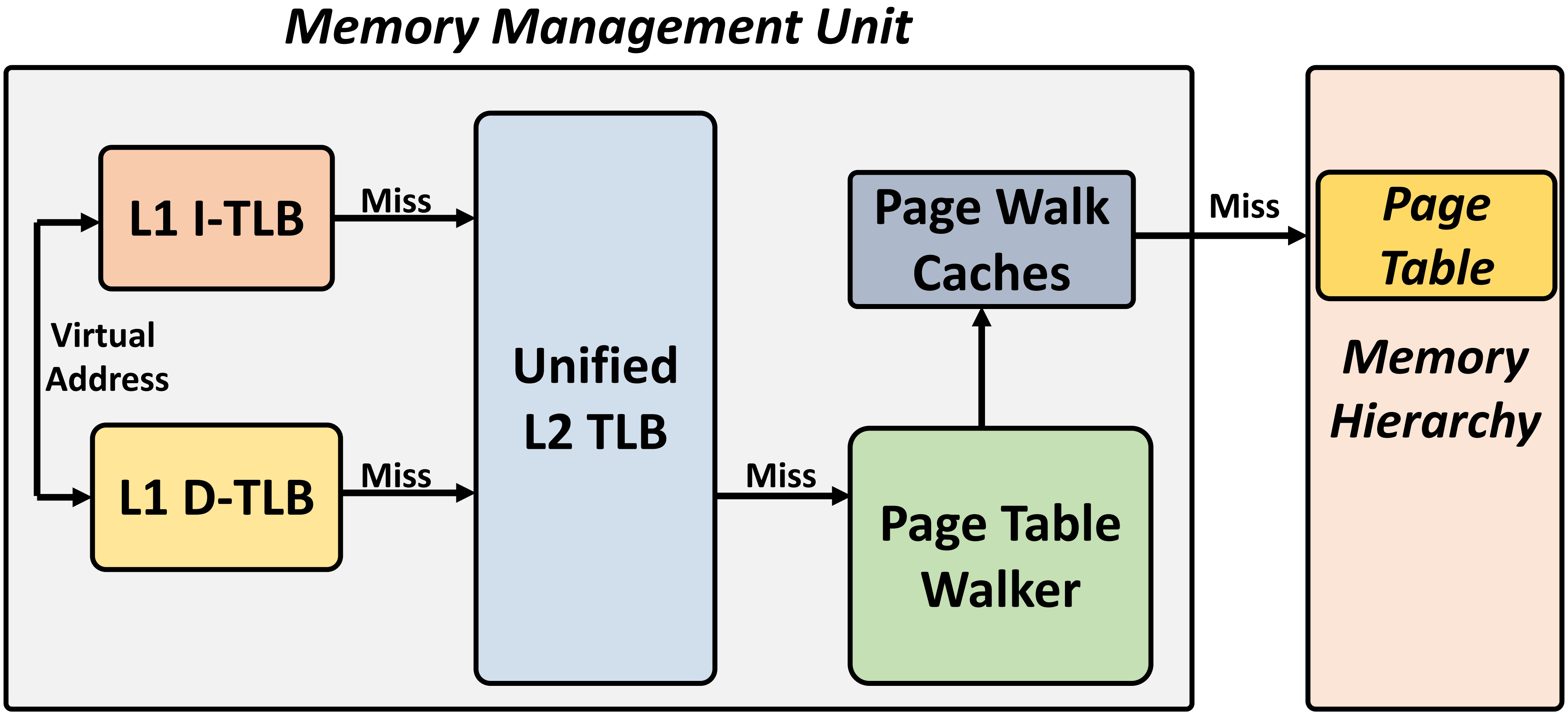


Four sequential memory accesses during a page table walk in x86-64

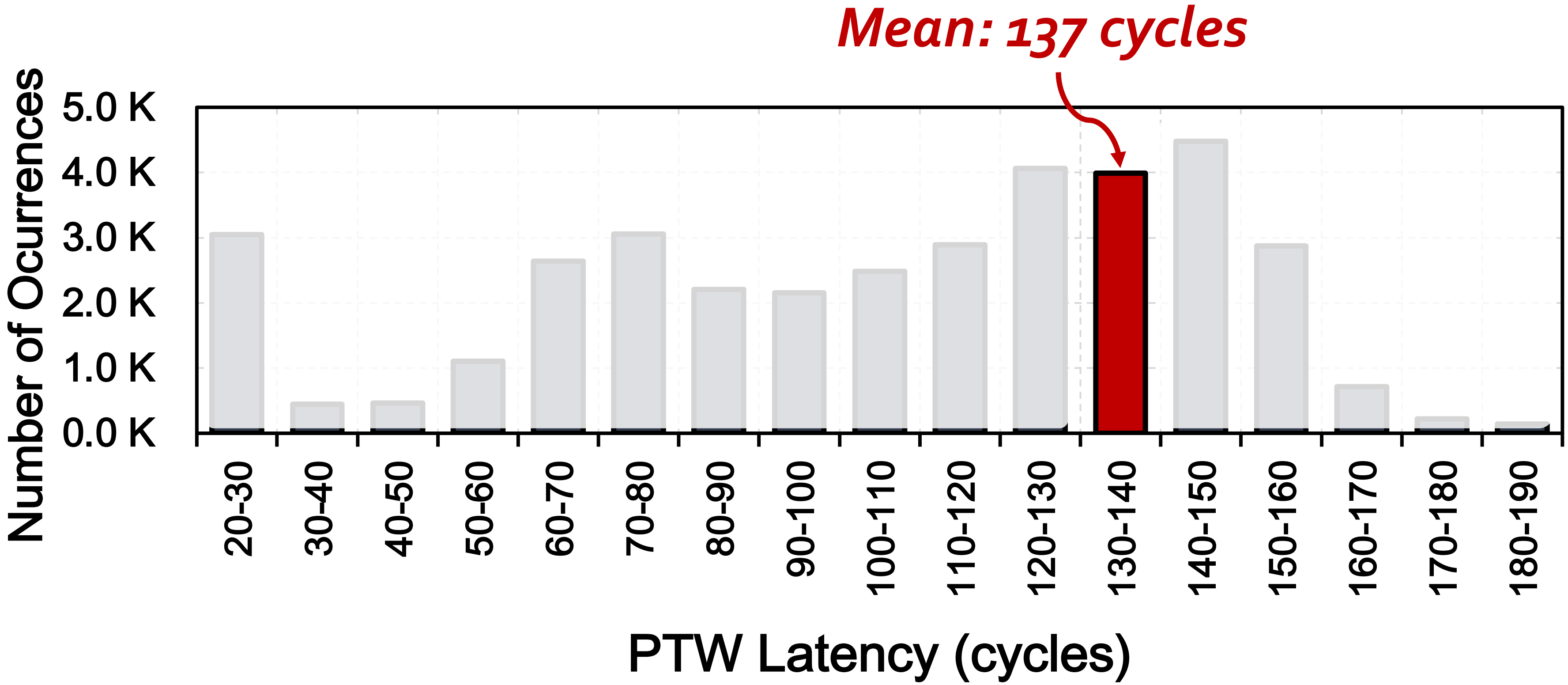
Address Translation Flow (I)



Address Translation Flow (II)

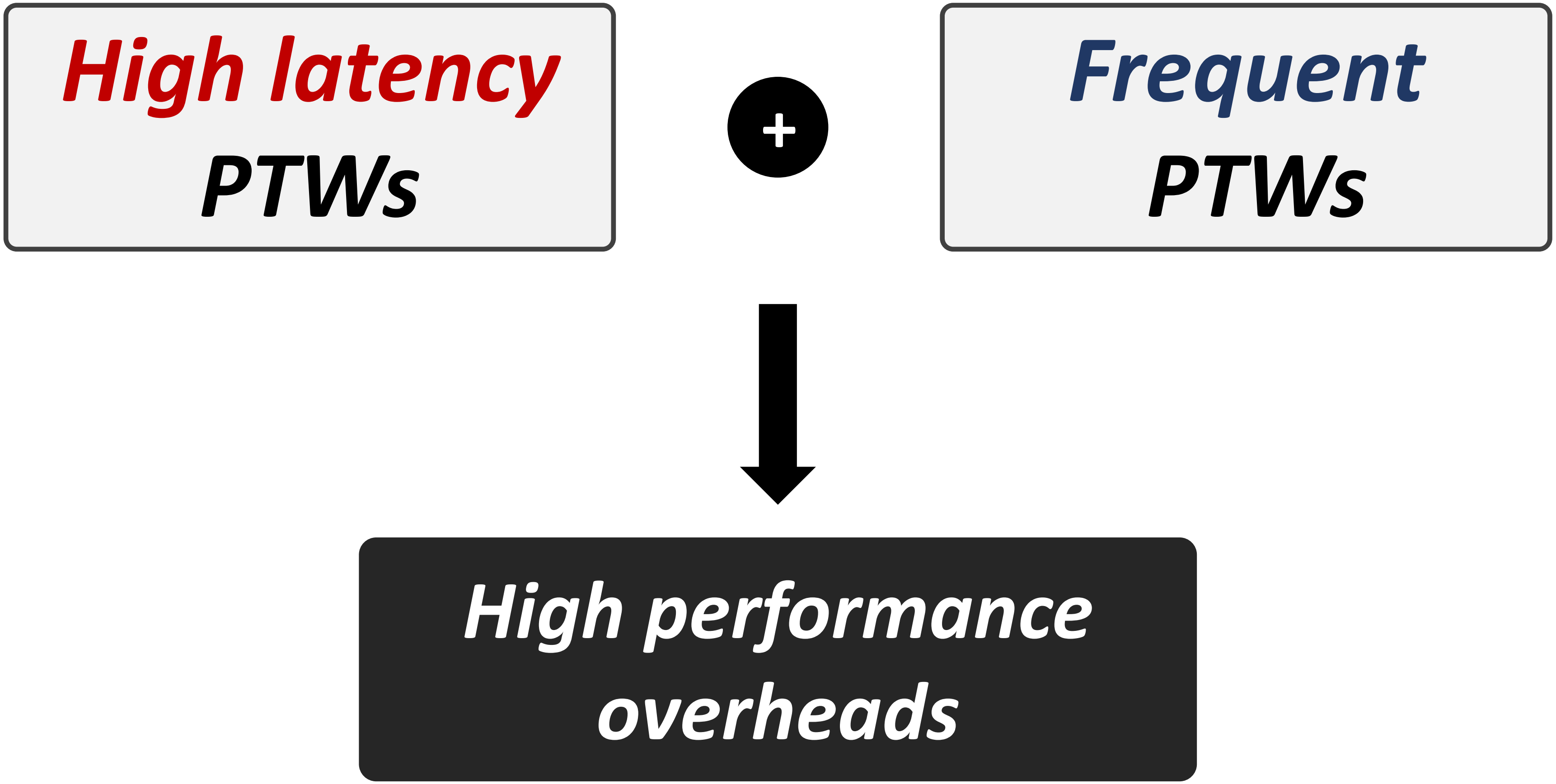


Address Translation Overhead



Core spends 137 cycles on average to perform a PTW

Address Translation Overhead



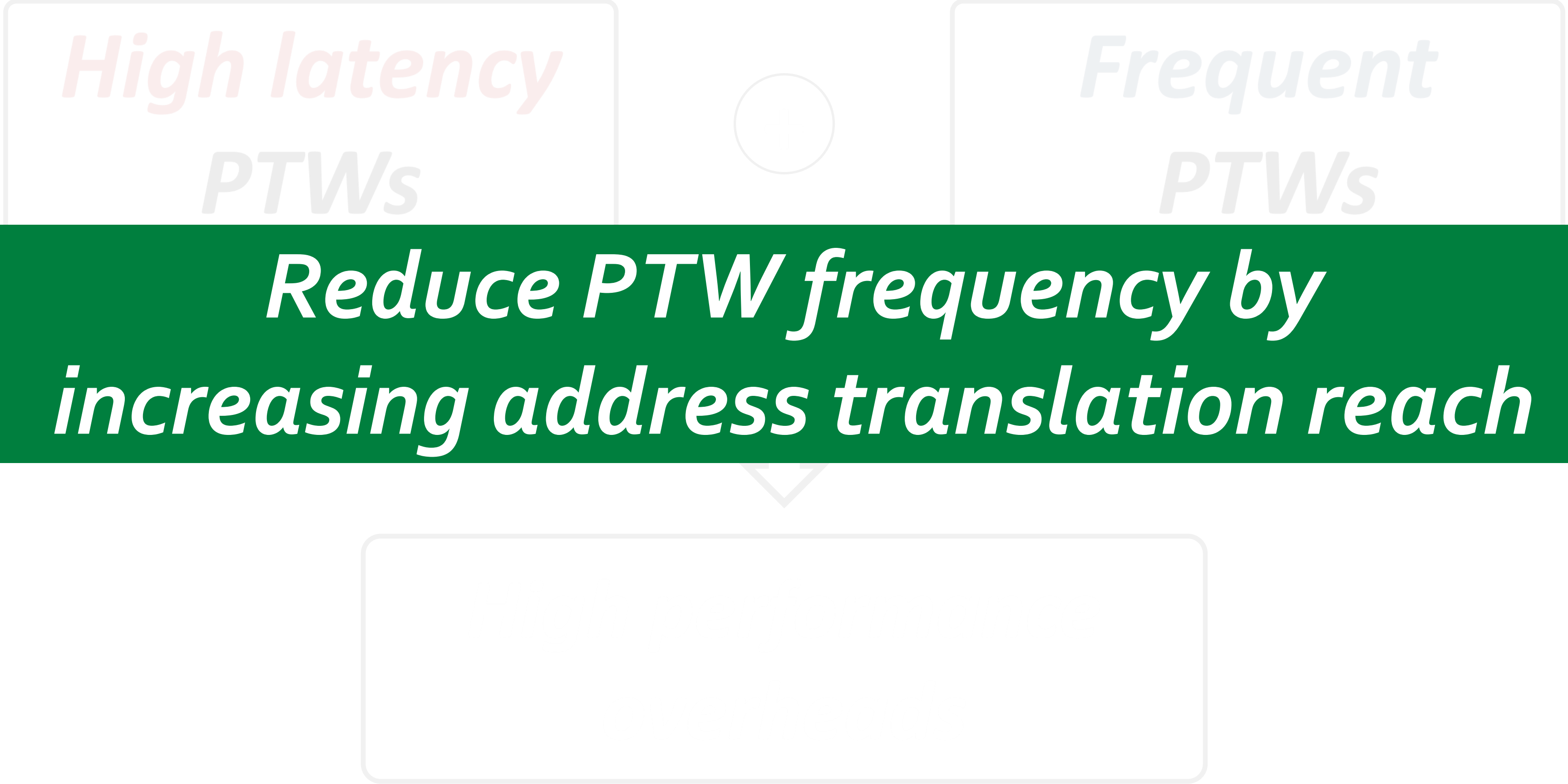
High latency *PTWs*

+

Frequent *PTWs*

High performance overheads

Potential Solution

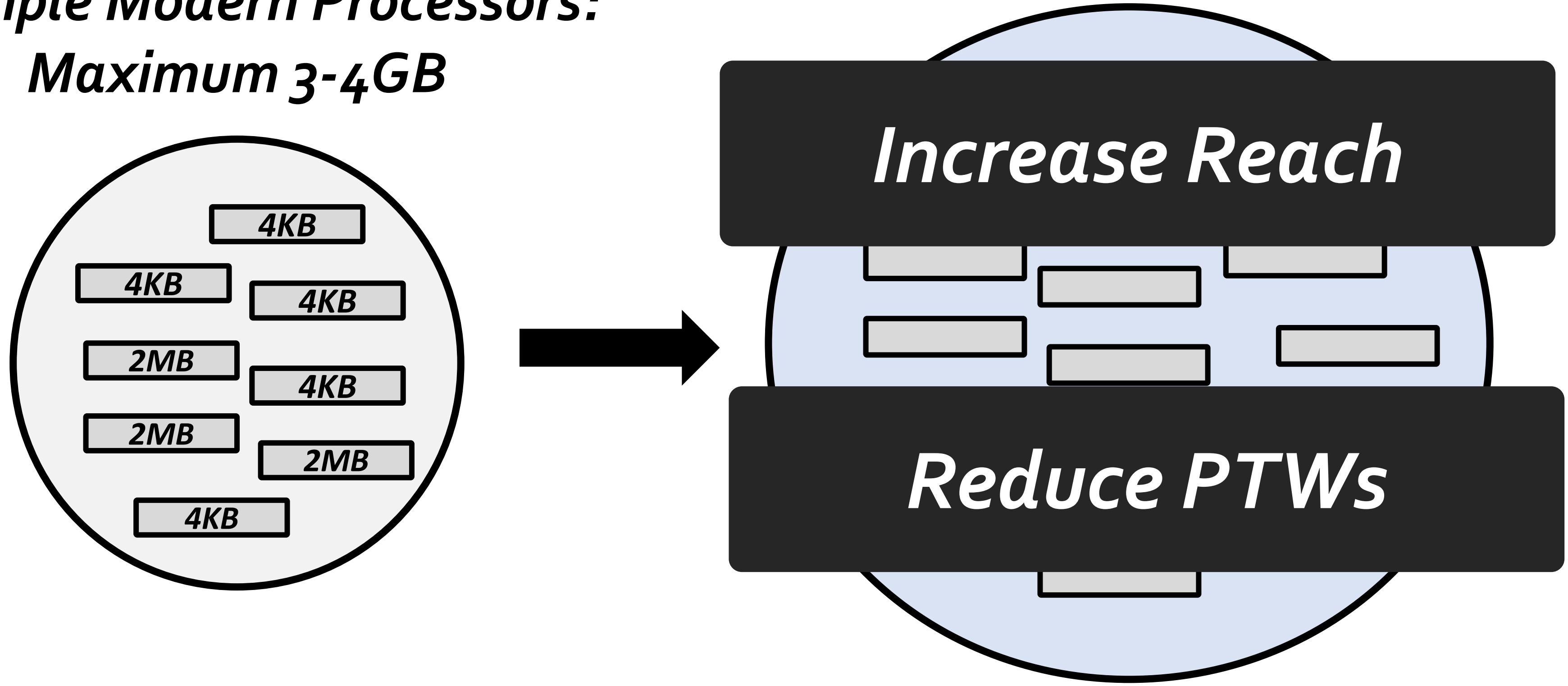


Reduce PTW frequency by increasing address translation reach

Address Translation Reach: Definition

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

Example Modern Processors:
Maximum 3-4GB



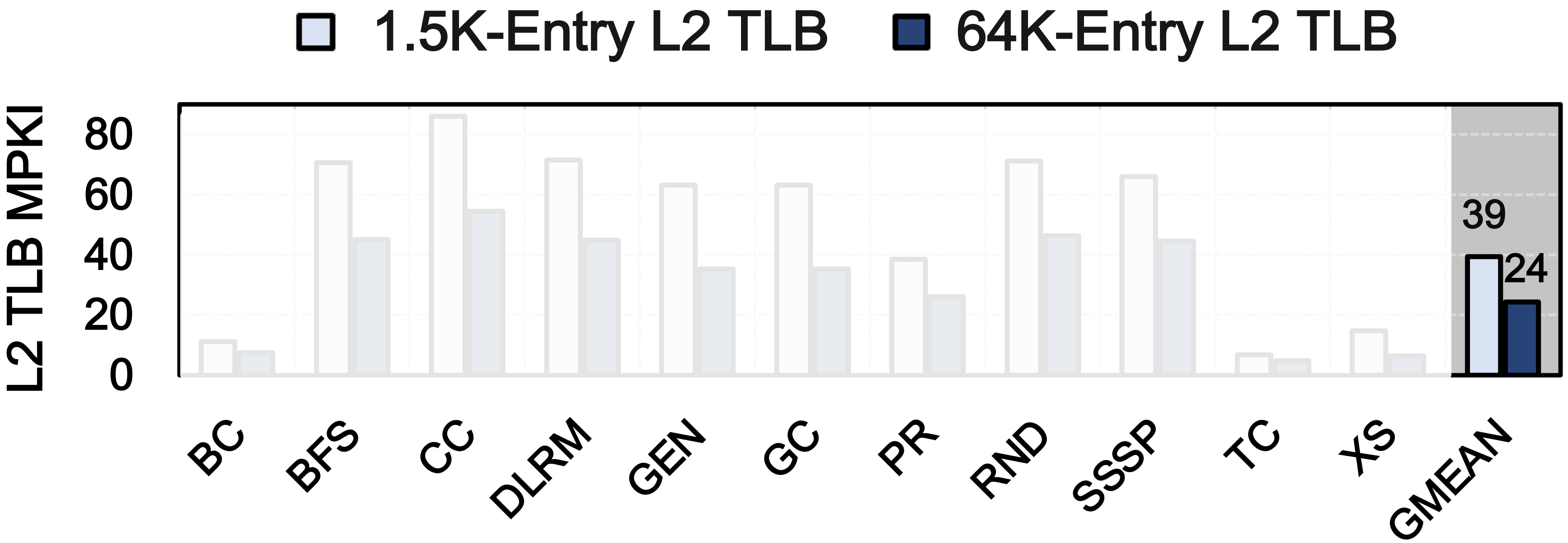
Increase Reach

Reduce PTWs

Increasing Address Translation Reach

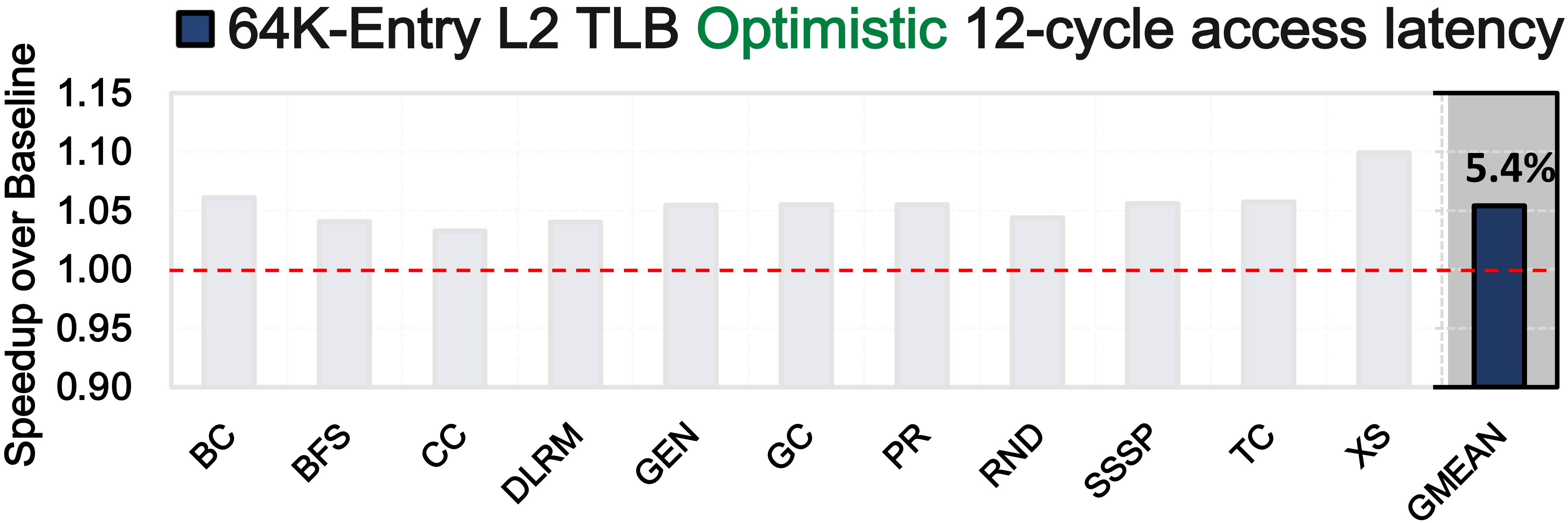
Large Hardware TLBs

Scaling Hardware L2 TLB (I)



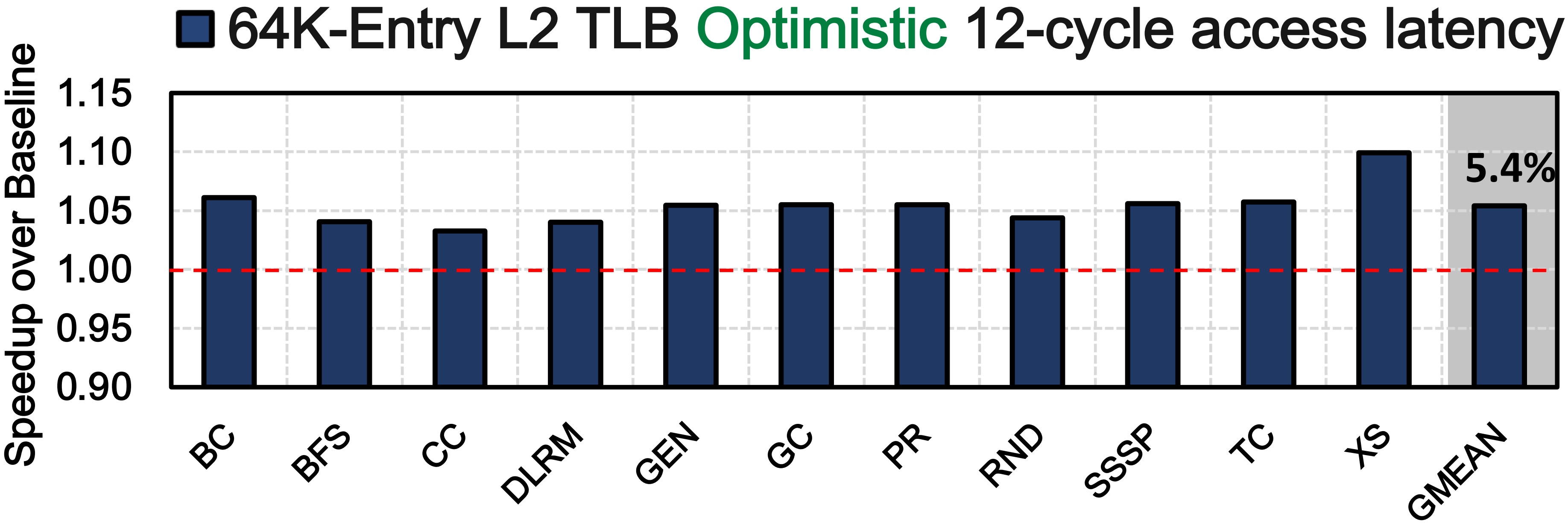
Employing a 64K-entry L2 TLB reduces MPKI from 39 to 24

Scaling Hardware L2 TLB (II)



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

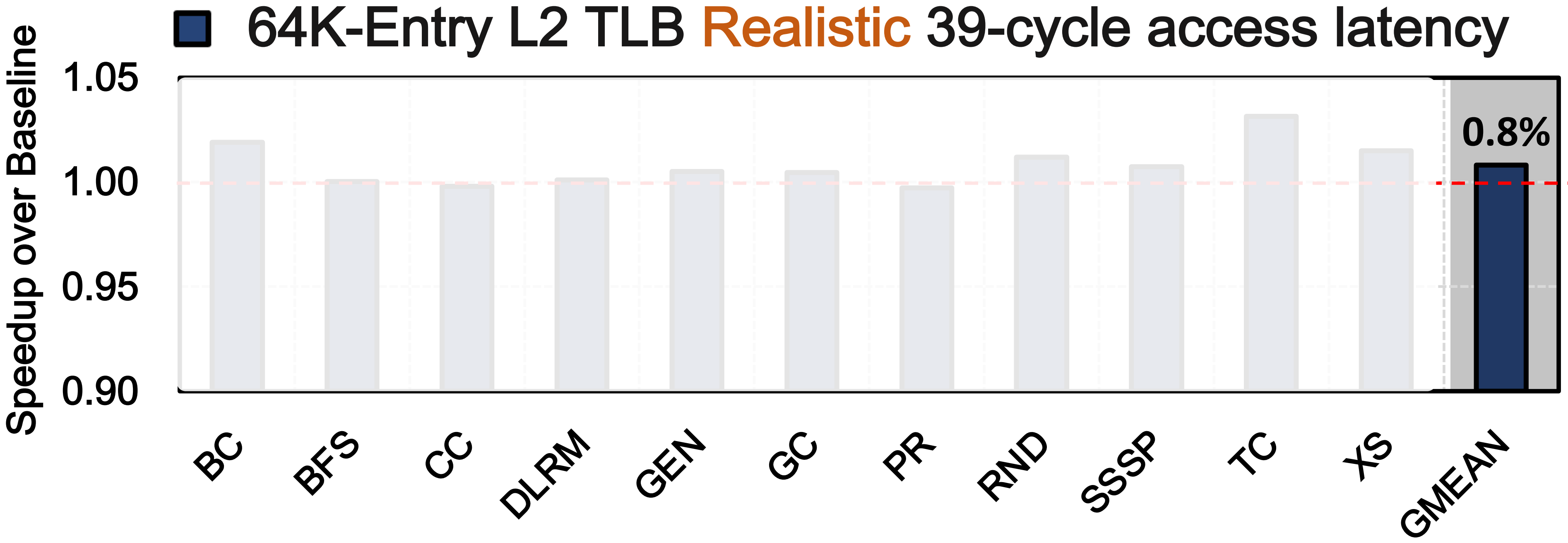
Scaling Hardware L2 TLB (II)



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

Benefits come for free?

Scaling Hardware L2 TLB (III)



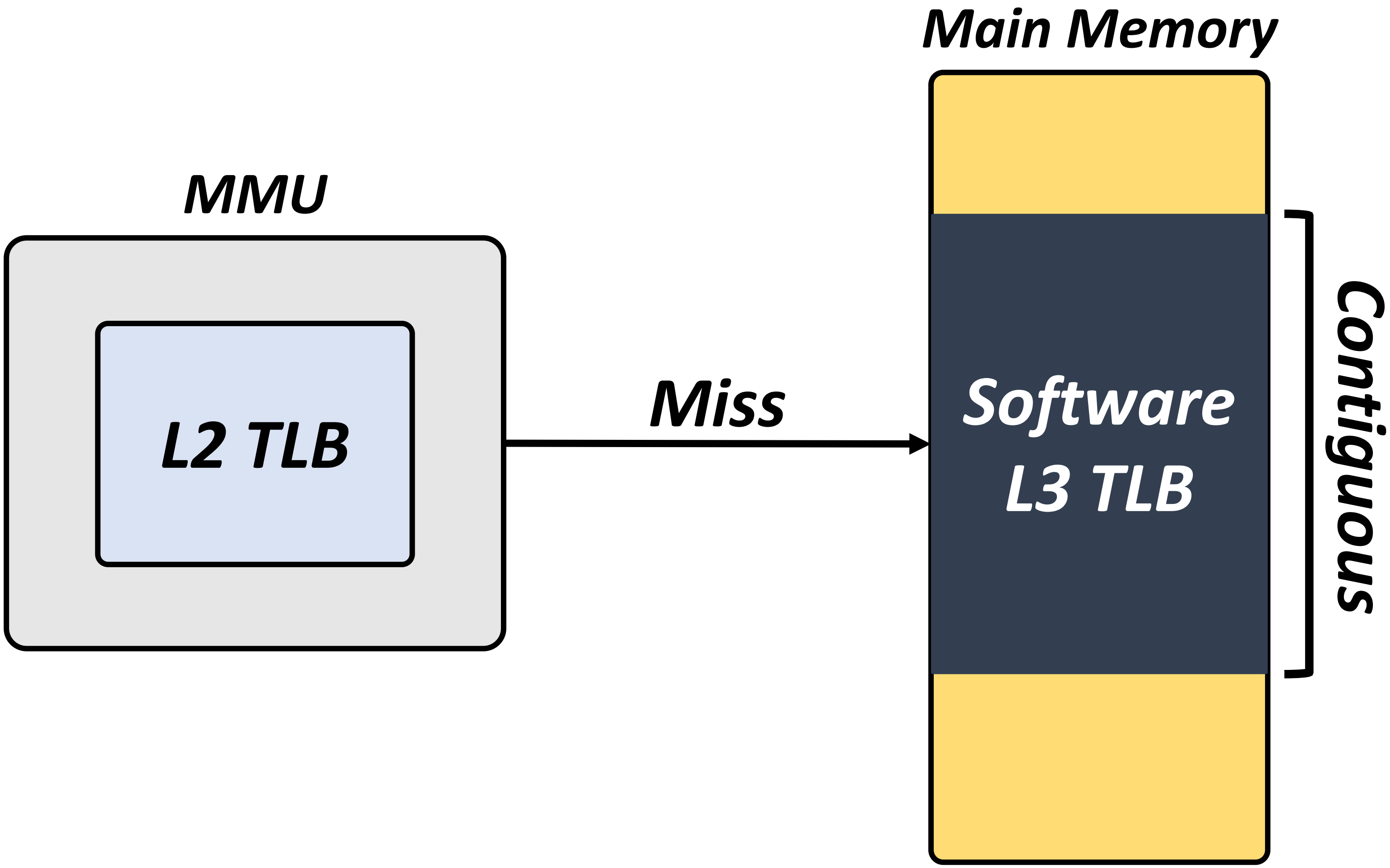
64K-entry L2 TLB with realistic access latency provides only 0.8% speedup over baseline

Increasing Translation Reach

Large Hardware TLBs

Large Software-Managed TLBs

Large Software-Managed L3 TLB



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

Victima: Overview

Victima: Detailed Design

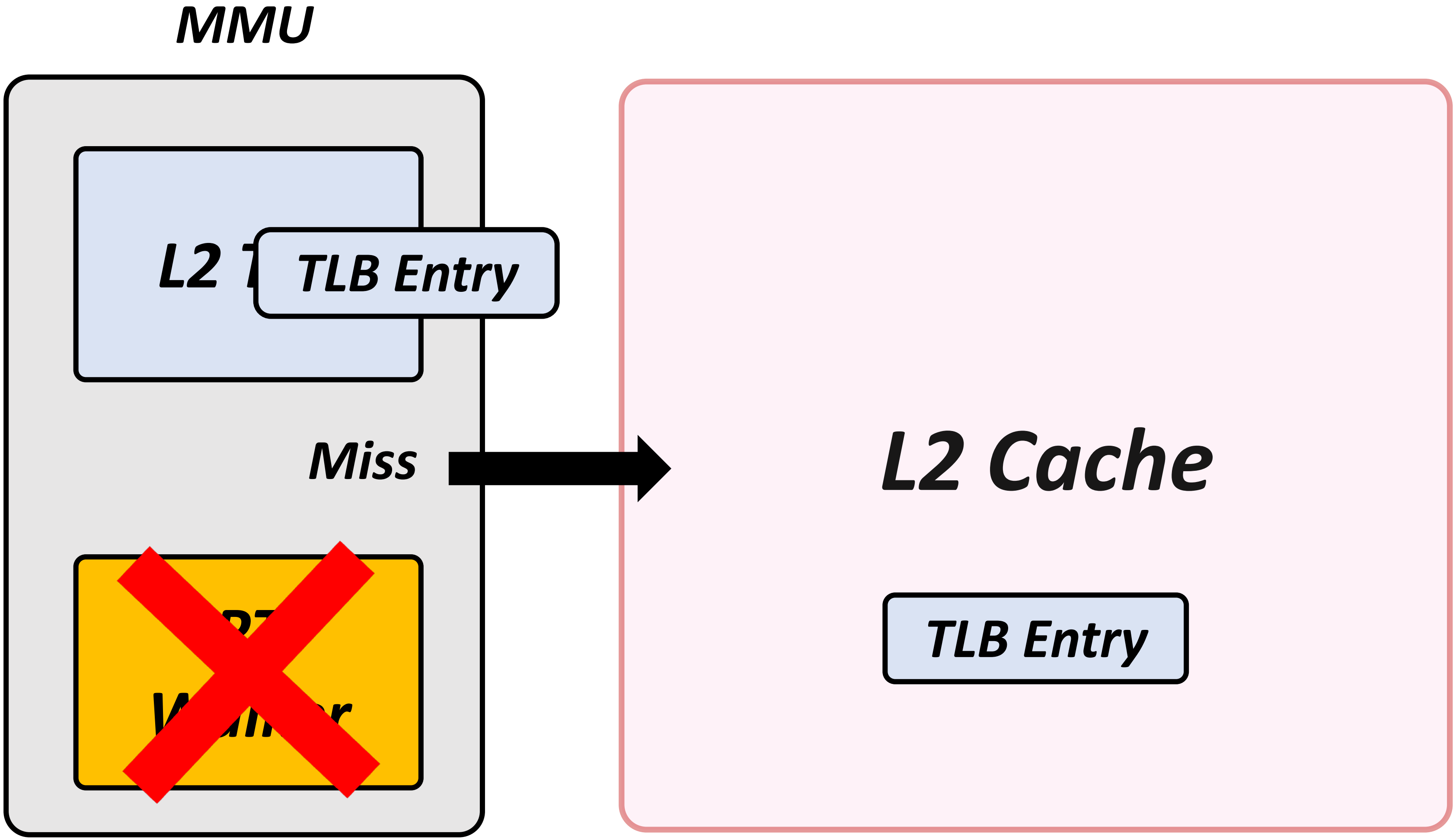
Evaluation Results

SAFARI

Opportunity: Leverage Caches

Store TLB entries in hardware caches

Leverage Cache Hierarchy



Where is the Benefit?

L2 TLB

1.5K entries

12-cycle latency

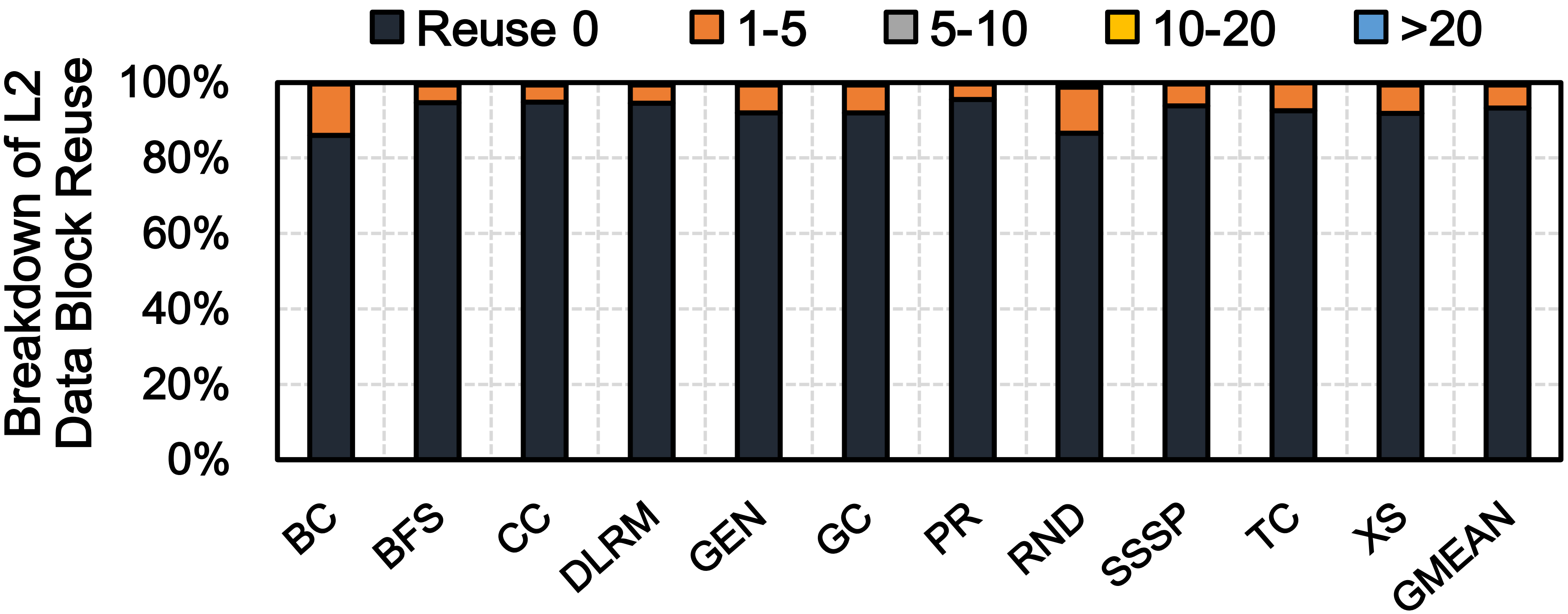
2MB L2 Cache

Fits 36x more TLB entries

Low latency (e.g., 16 cycles)

PTW takes 137 cycles on average

Interference with Program Data?



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

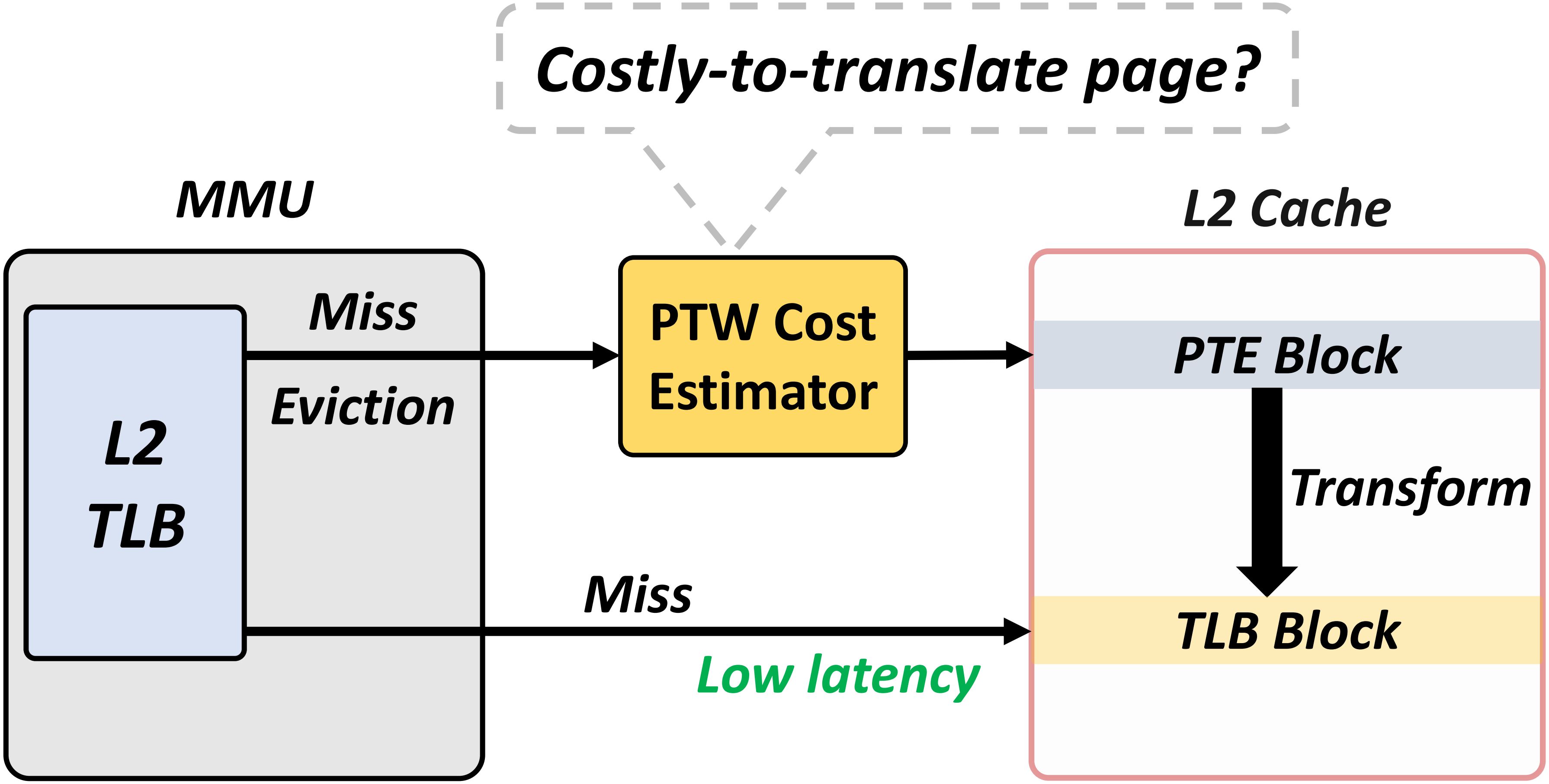
Repurpose L2 cache blocks

to store clusters of TLB entries

Low-latency *and* ***high-capacity*** *component*

to back up the L2 TLB

Victima: Overview



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

Victima: Overview

Victima: 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

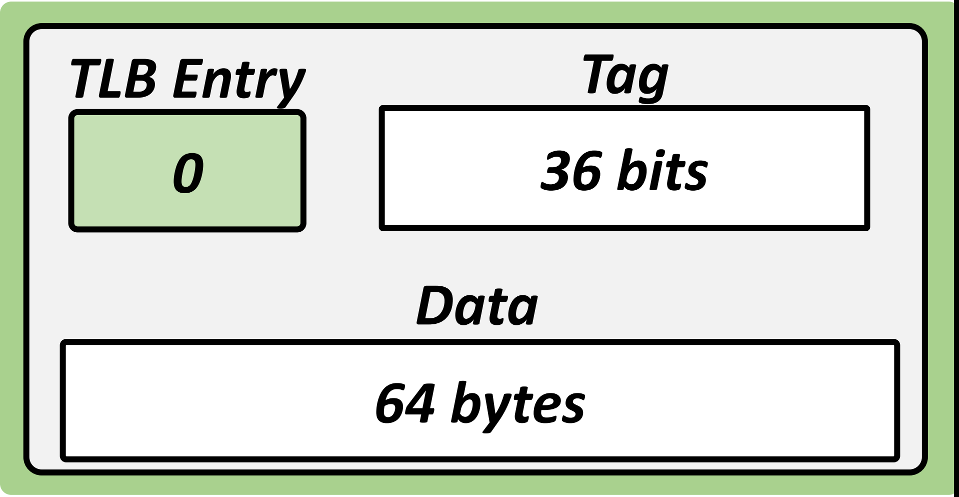
1MB

16-way associative

Set index *10 bits*

Data Blocks vs. TLB Blocks in Caches

Data Block



TLB Entry

0

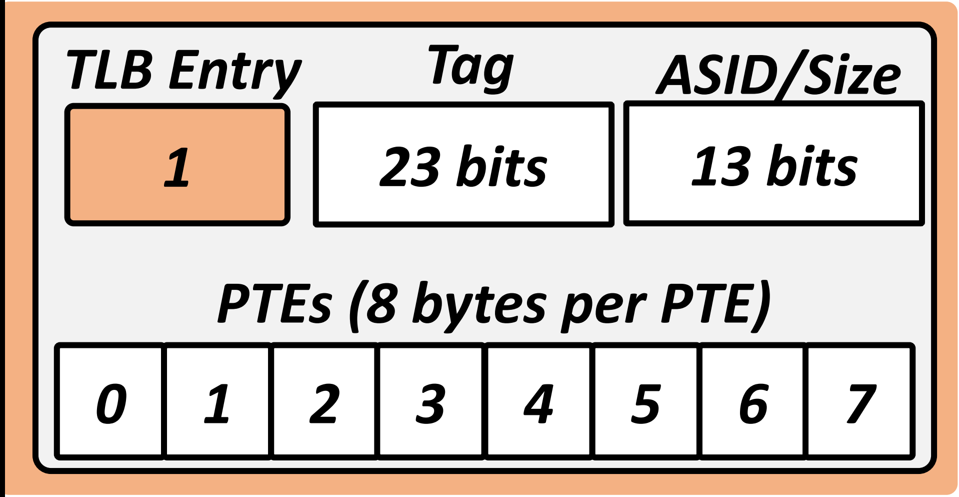
Tag

36 bits

Data

64 bytes

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

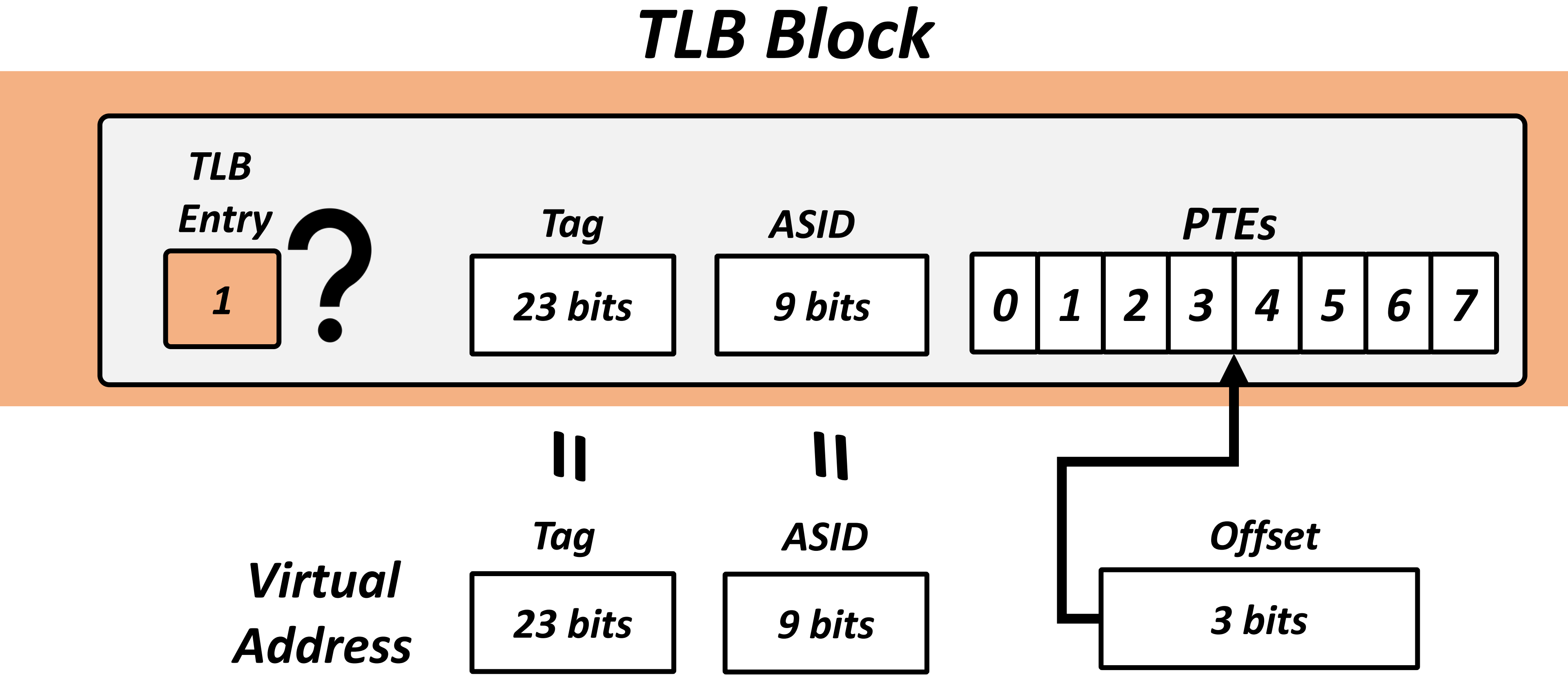
52-bit Physical Address

Tag	Set index	Offset
36 bits	10 bits	6 bits

36-bit Virtual Page Number (4KB)

Tag	Set index	Offset
23 bits	10 bits	3 bits

Tag Matching for TLB Block



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

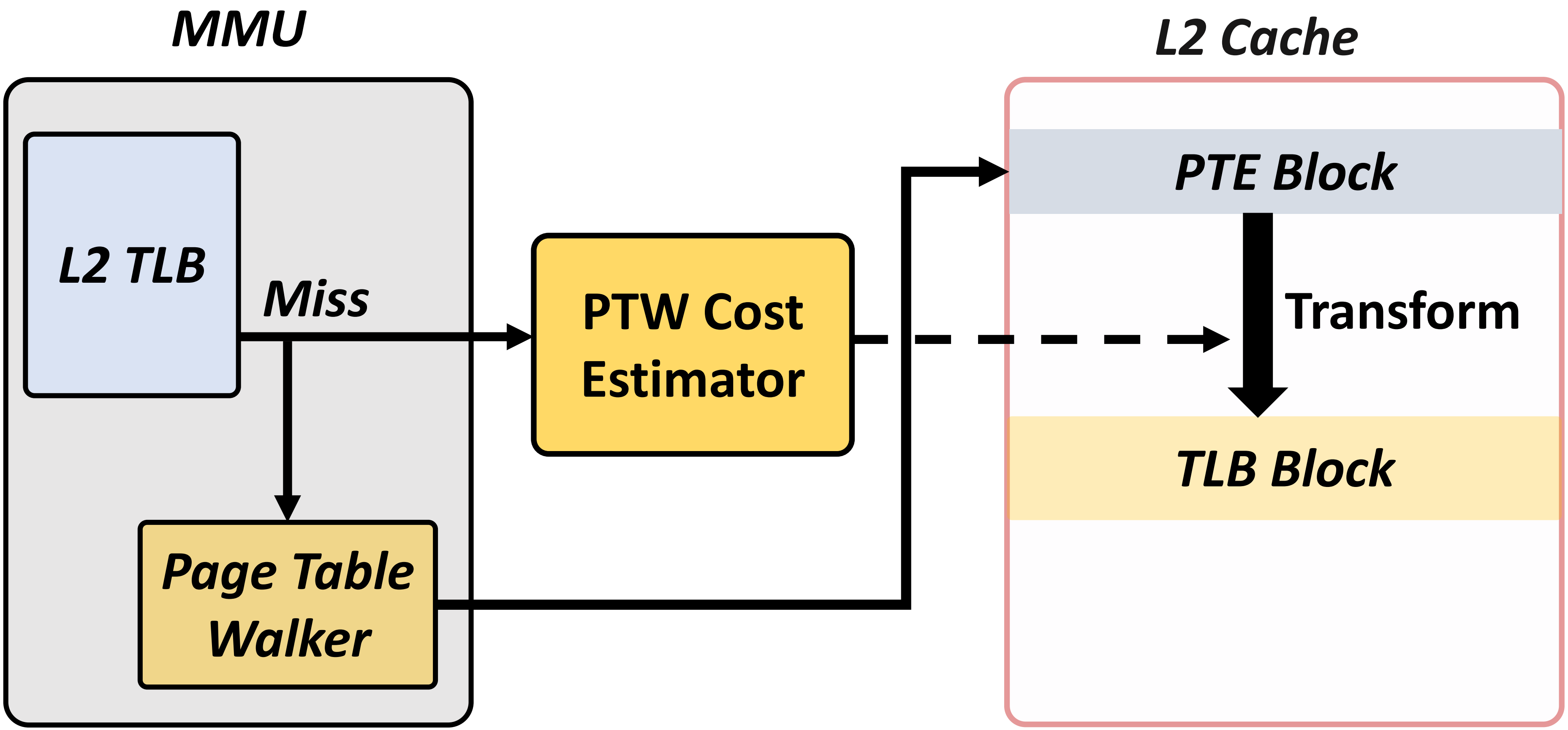
2 On L2 TLB Eviction

Allocation of TLB Entries in L2 Cache

1. On L2 TLB Miss

2. On L2 TLB Eviction

Allocating TLB Blocks – L2 TLB Miss

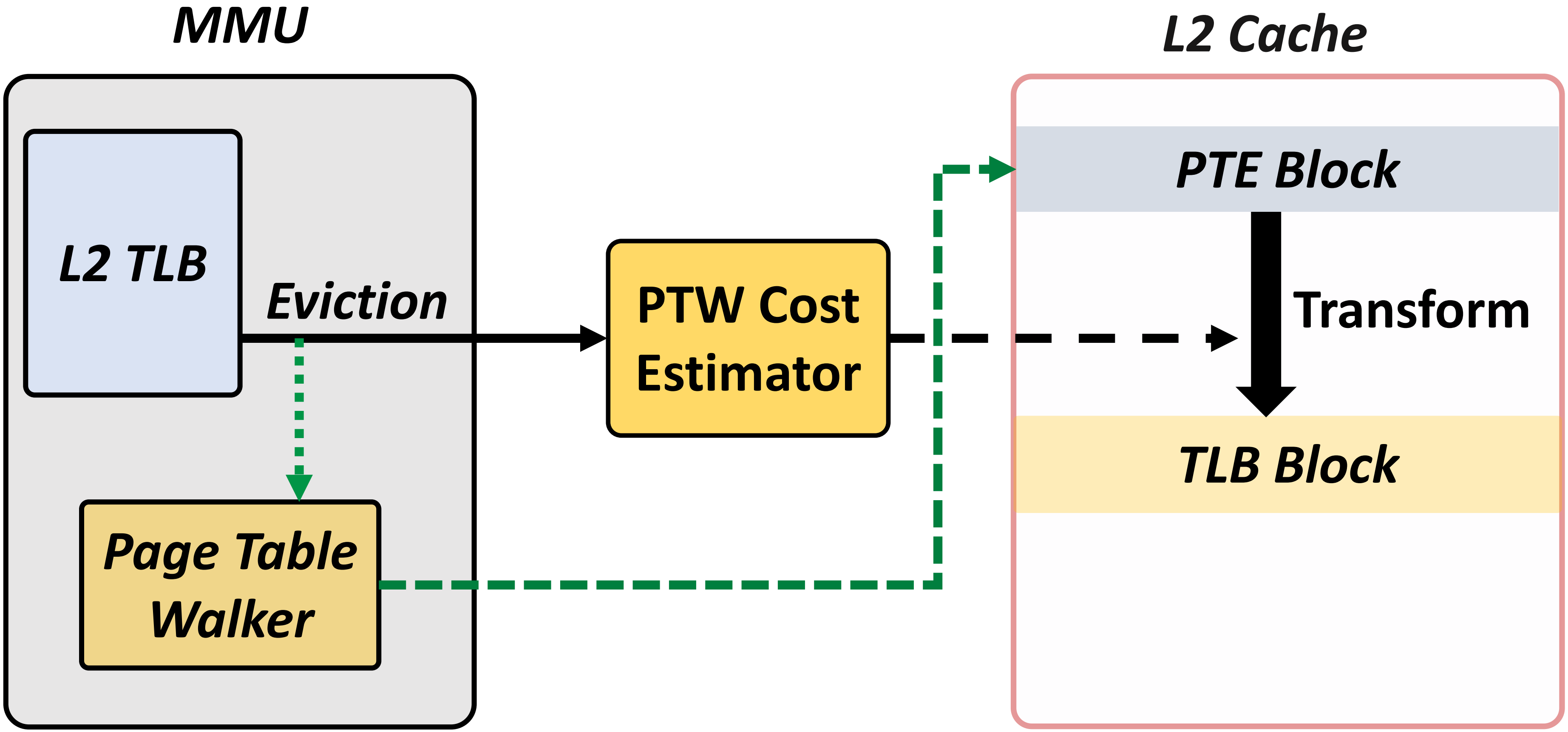


Allocation of TLB Entries in L2 Cache

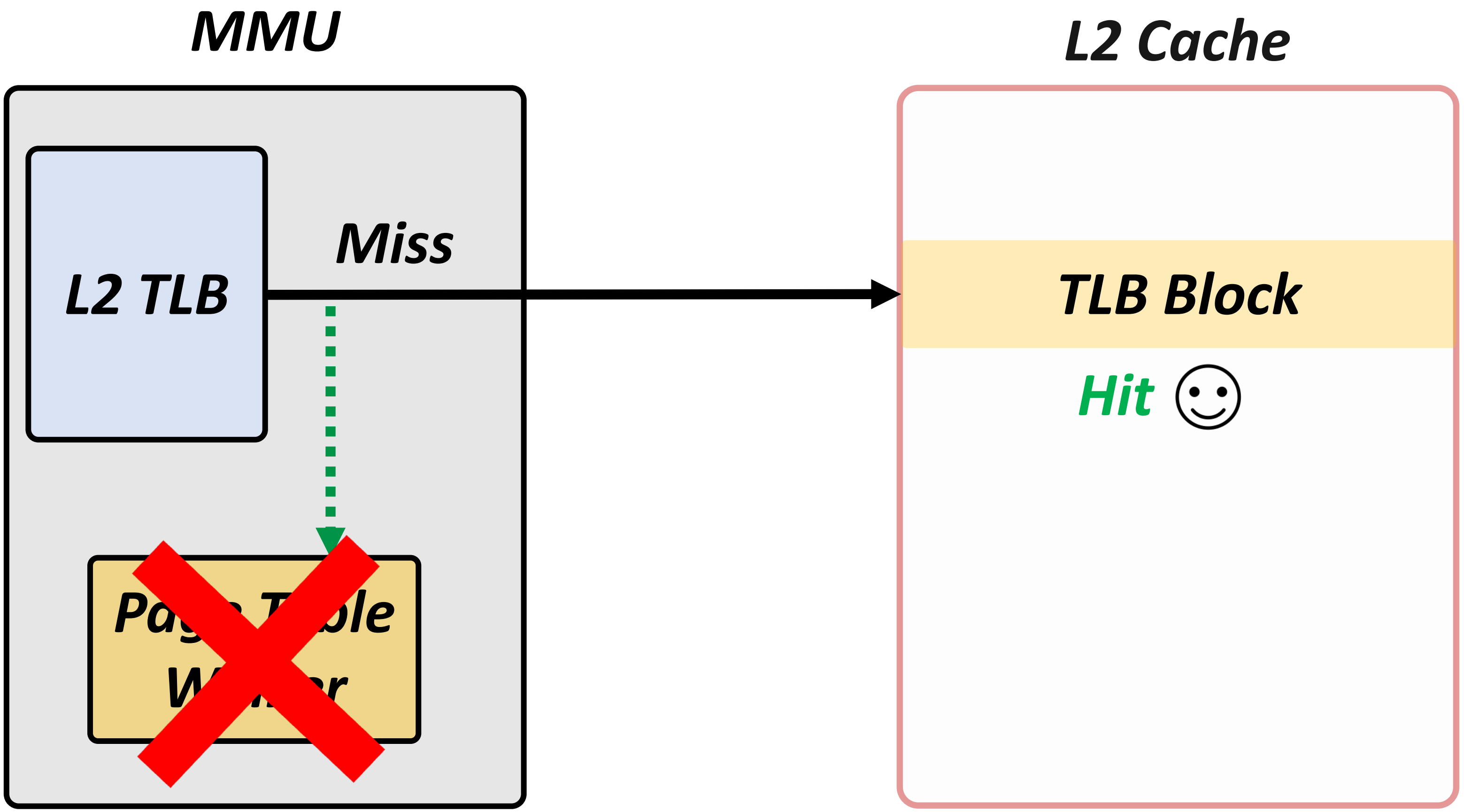
1 On L2 TLB Miss

2 On L2 TLB Eviction

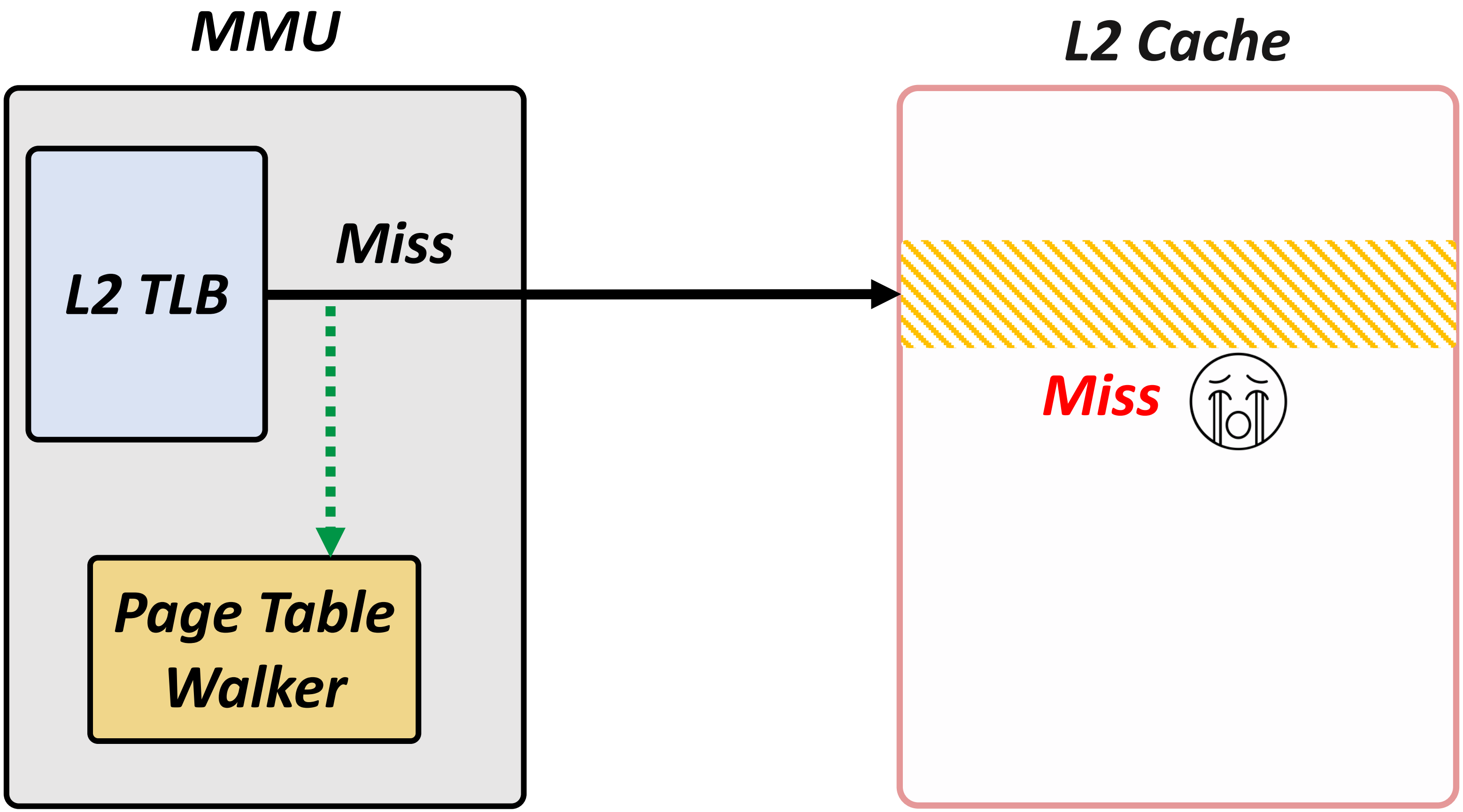
Allocating TLB Blocks – L2 TLB Eviction



Address Translation in Victima (I)



Address Translation in Victima (II)



Victima: Detailed Design

1 *L2 Cache Modifications*

2 *Allocation of TLB Blocks in L2 Cache*

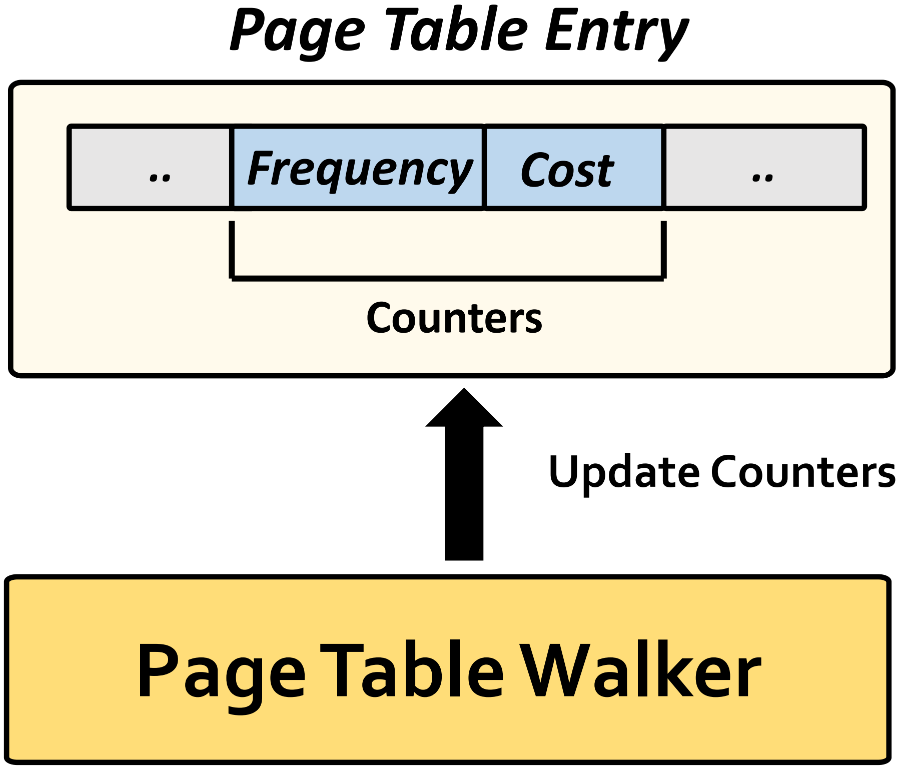
3 ***Page Table Walk Cost Predictor***

PTW Cost Predictor: Objective

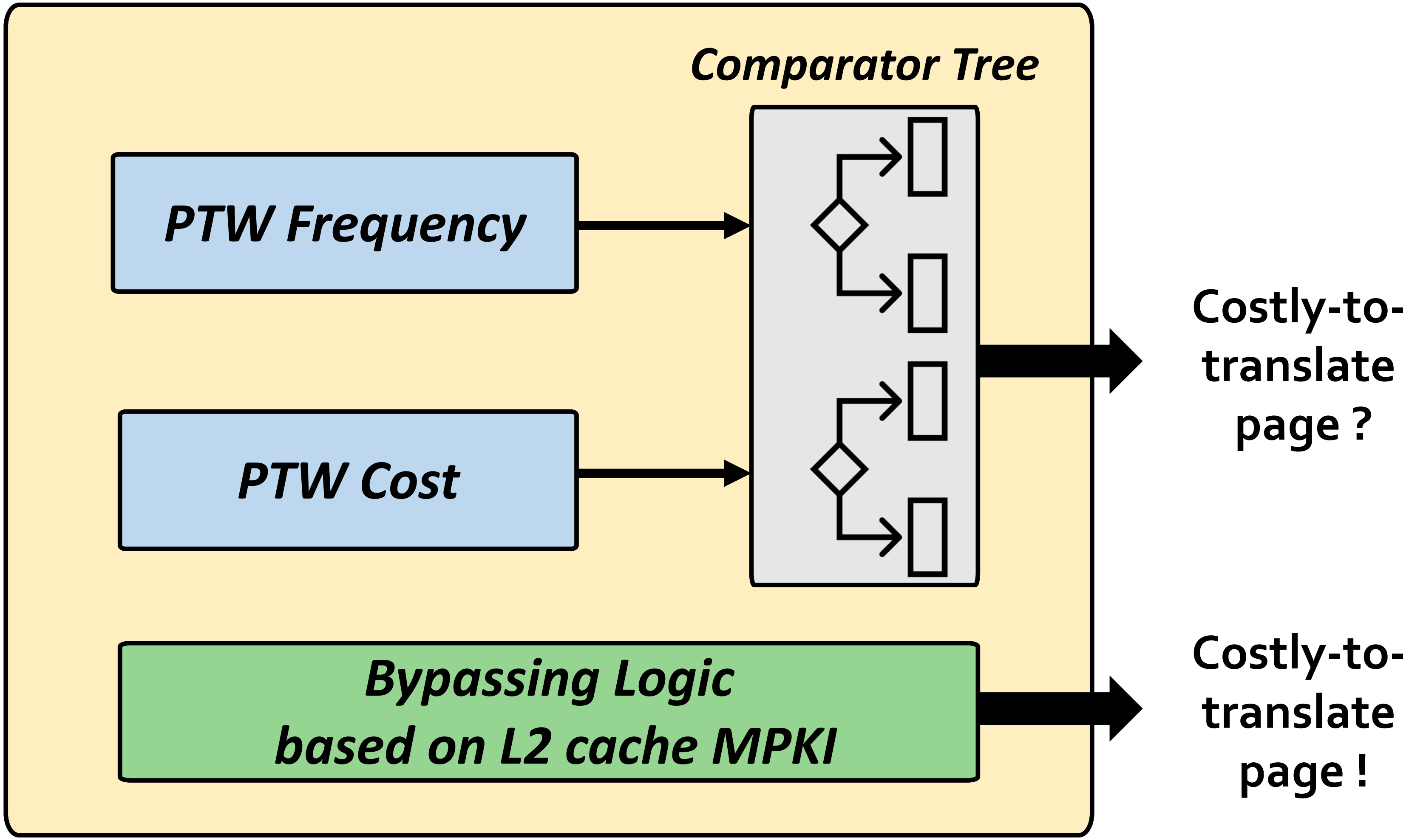
Predict which pages are costly-to-translate

Insert only those TLB blocks in L2 cache

Tracking Costly-to-Translate Pages



PTW Cost Predictor (PTW-CP)



PTW-CP Details in the Paper

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

Victima: Overview

Victima: 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

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)

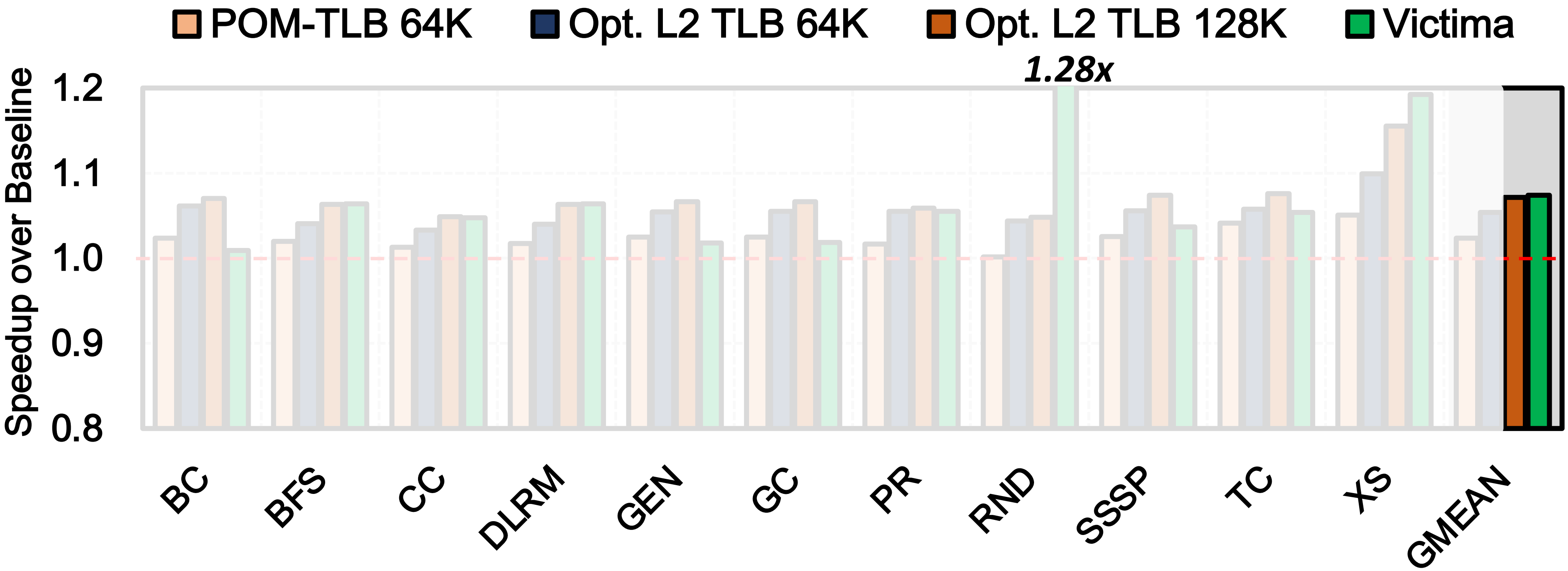
- **Optimistic L2 TLB-128K**: System with 128K-entry L2 TLB (optimistic 12-cycle access latency)

- **POM-TLB[1]**: System with 64K-entry software-managed L3 TLB

- **Victima**

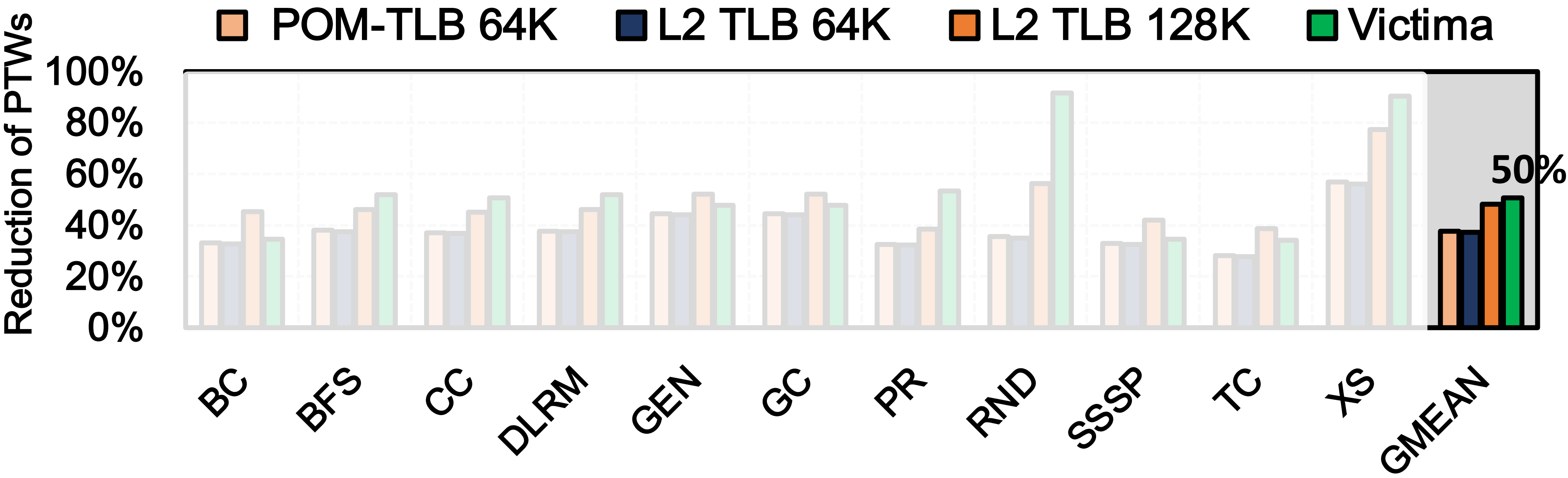
[1] Ryoo et al. "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB" ISCA 2017

Performance Speedup



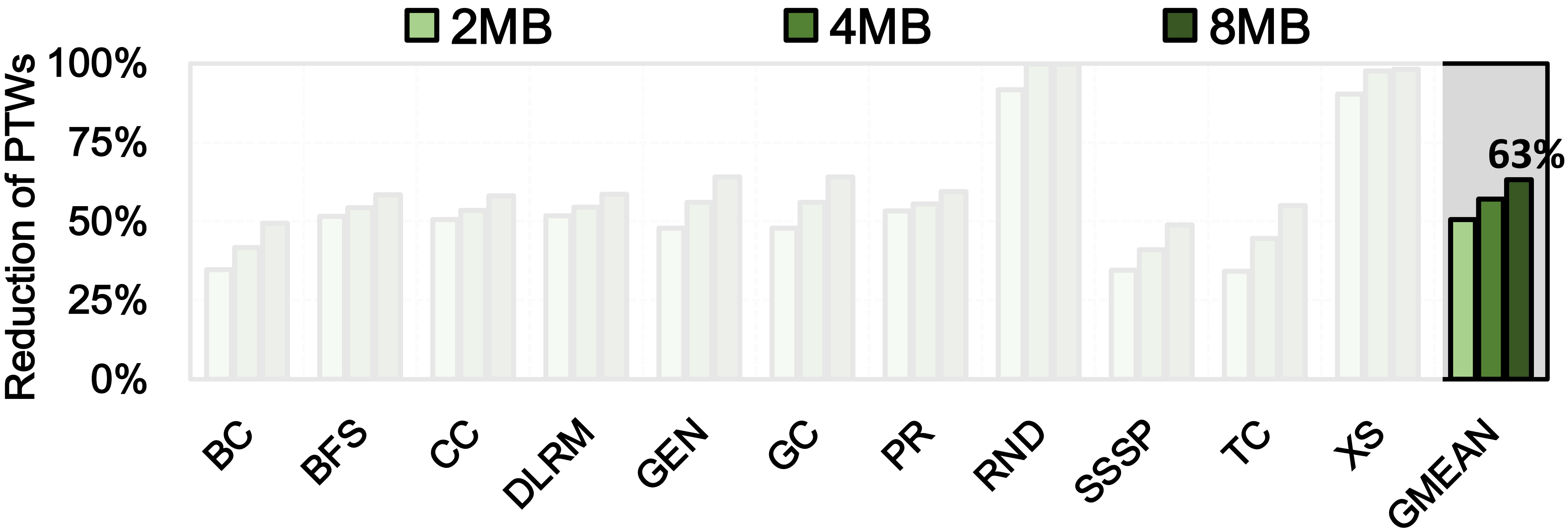
Victima achieves similar performance to the optimistically fast 128K-entry L2 TLB

Reduction of Page Table Walks



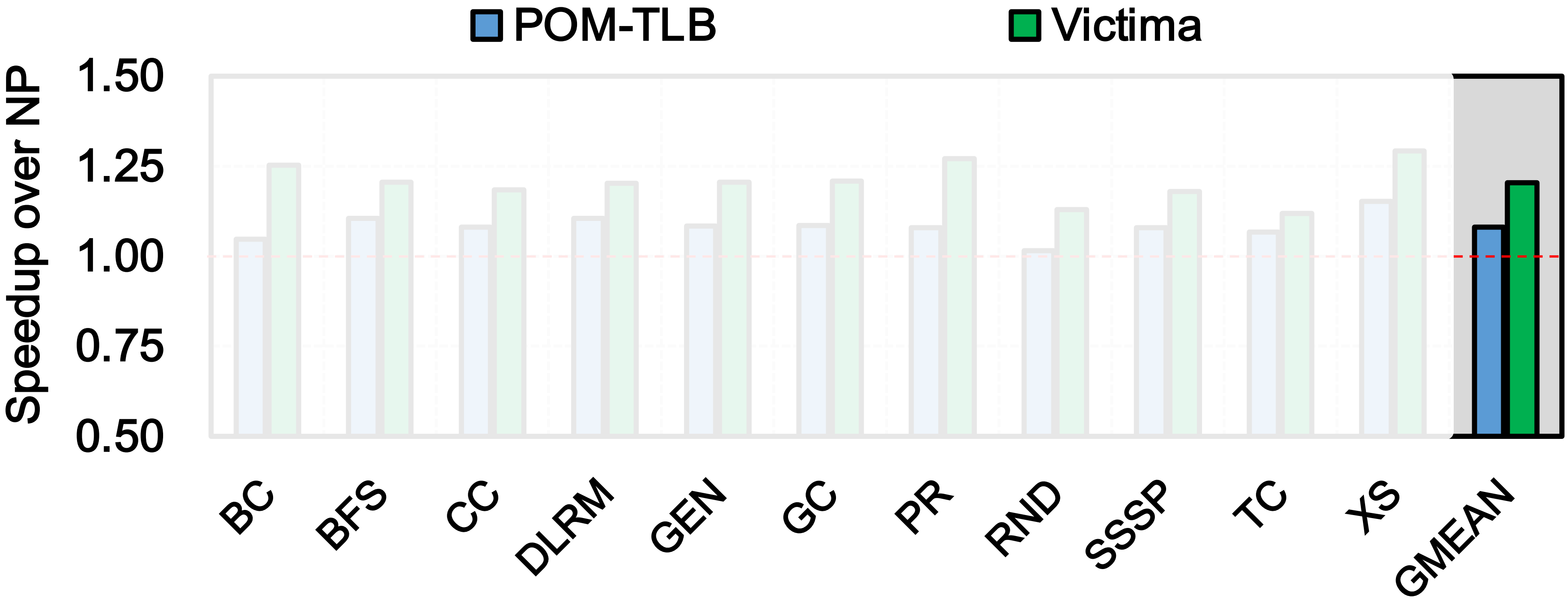
Victima reduces PTWs by 50% on average compared to the baseline

Effect of L2 Cache Size on Victima



Employing an 8MB L2 cache with Victima reduces PTWs by 63%

Performance in Virtualized Environments



Victima outperforms 64K-entry software-managed TLB by 12%

Area & Power Overhead

- Area and power overhead evalution 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

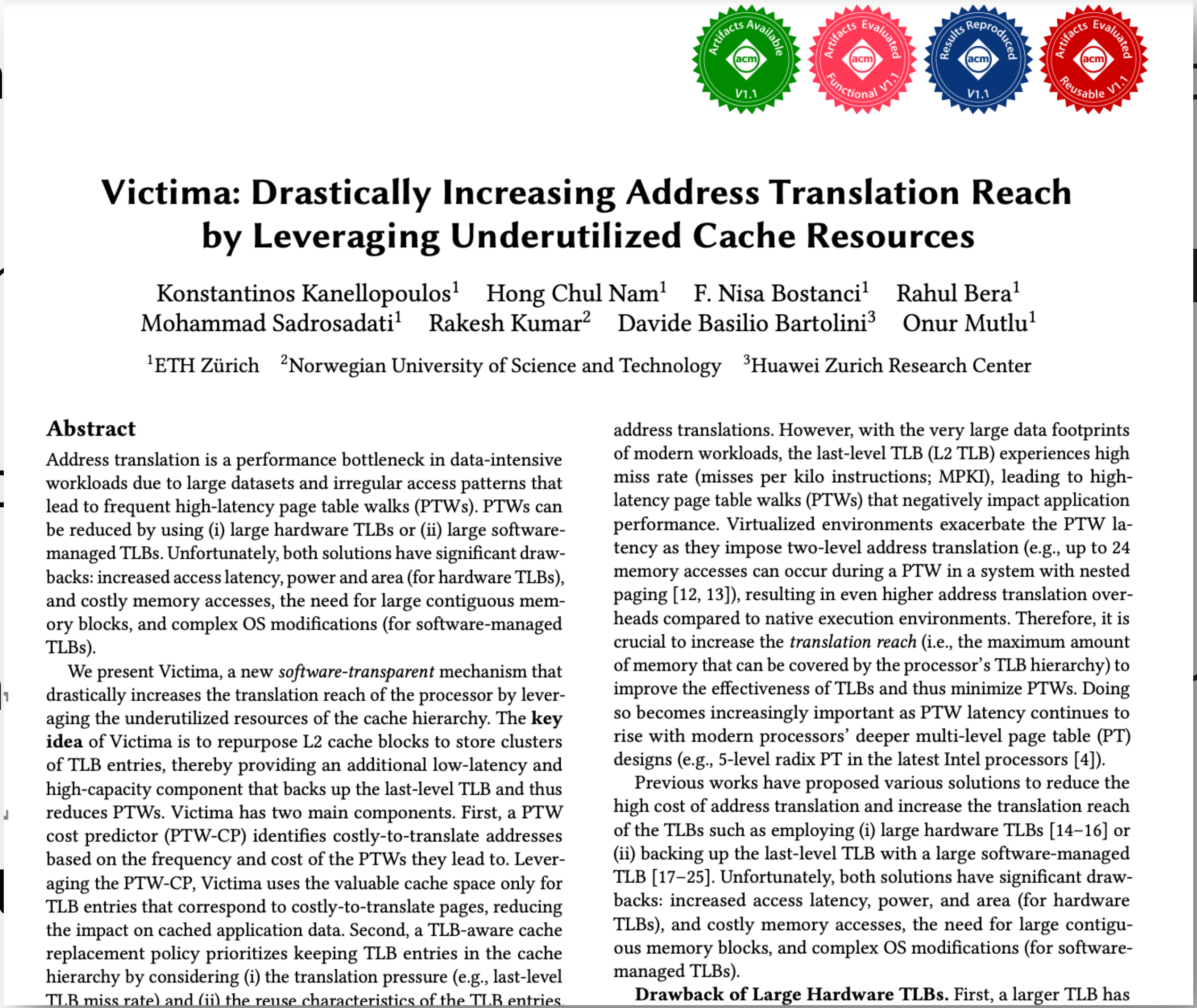
- Implementation details of PTW cost estimator

- Translation reach provided by Victima

https://arxiv.org/abs/2310.04158

More in the paper

- Victima ts
- Mainter lowns
- TLB-Blc
- Implem
- Translat



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

Konstantinos Kanellopoulos[1] Hong Chul Nam[1] F. Nisa Bostanci[1] Rahul Bera[1]
Mohammad Sadrosadati[1] Rakesh Kumar[2] Davide Basilio Bartolini[3] Onur Mutlu[1]

[1]ETH Zürich [2]Norwegian University of Science and Technology [3]Huawei 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]).

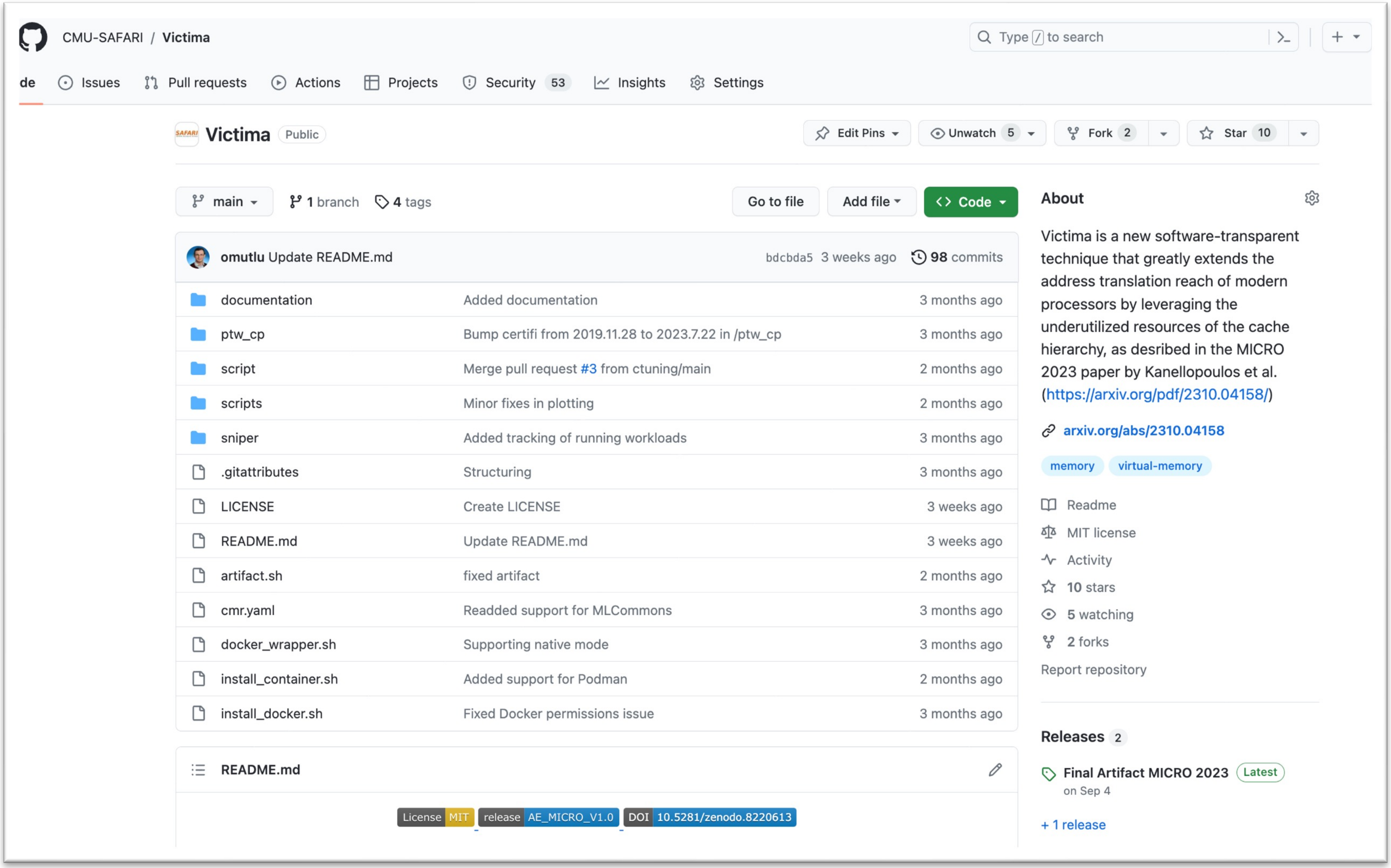
Previous works have proposed various solutions to reduce the high cost of address translation and increase the translation reach of the TLBs such as employing (i) large hardware TLBs [14–16] or (ii) backing up the last-level TLB with a large software-managed TLB [17–25]. Unfortunately, both solutions 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

https://arxiv.org/abs/2310.04158

Victima is Open Source

Artifacts Available acm V1.1 | Artifacts Evaluated acm Functional V1.1 | Artifacts Evaluated acm Reusable V1.1 | Results Reproduced acm V1.1



CMU-SAFARI / Victima

Type / to search

Victima Public

omutlu Update README.md bdcbda5 3 weeks ago 98 commits

documentation	Added documentation	3 months ago
ptw_cp	Bump certifi from 2019.11.28 to 2023.7.22 in /ptw_cp	3 months ago
script	Merge pull request #3 from ctuning/main	2 months ago
scripts	Minor fixes in plotting	2 months ago
sniper	Added tracking of running workloads	3 months ago
.gitattributes	Structuring	3 months ago
LICENSE	Create LICENSE	3 weeks ago
README.md	Update README.md	3 weeks ago
artifact.sh	fixed artifact	2 months ago
cmr.yaml	Readded support for MLCommons	3 months ago
docker_wrapper.sh	Supporting native mode	3 months ago
install_container.sh	Added support for Podman	2 months ago
install_docker.sh	Fixed Docker permissions issue	3 months ago

README.md

License MIT | release AE_MICRO_V1.0 | DOI 10.5281/zenodo.8220613

About

Victima is a new software-transparent technique that greatly extends the address translation reach of modern processors by leveraging the underutilized resources of the cache hierarchy, as desribed in the MICRO 2023 paper by Kanellopoulos et al. (https://arxiv.org/pdf/2310.04158/)

arxiv.org/abs/2310.04158

memory virtual-memory

Releases 2

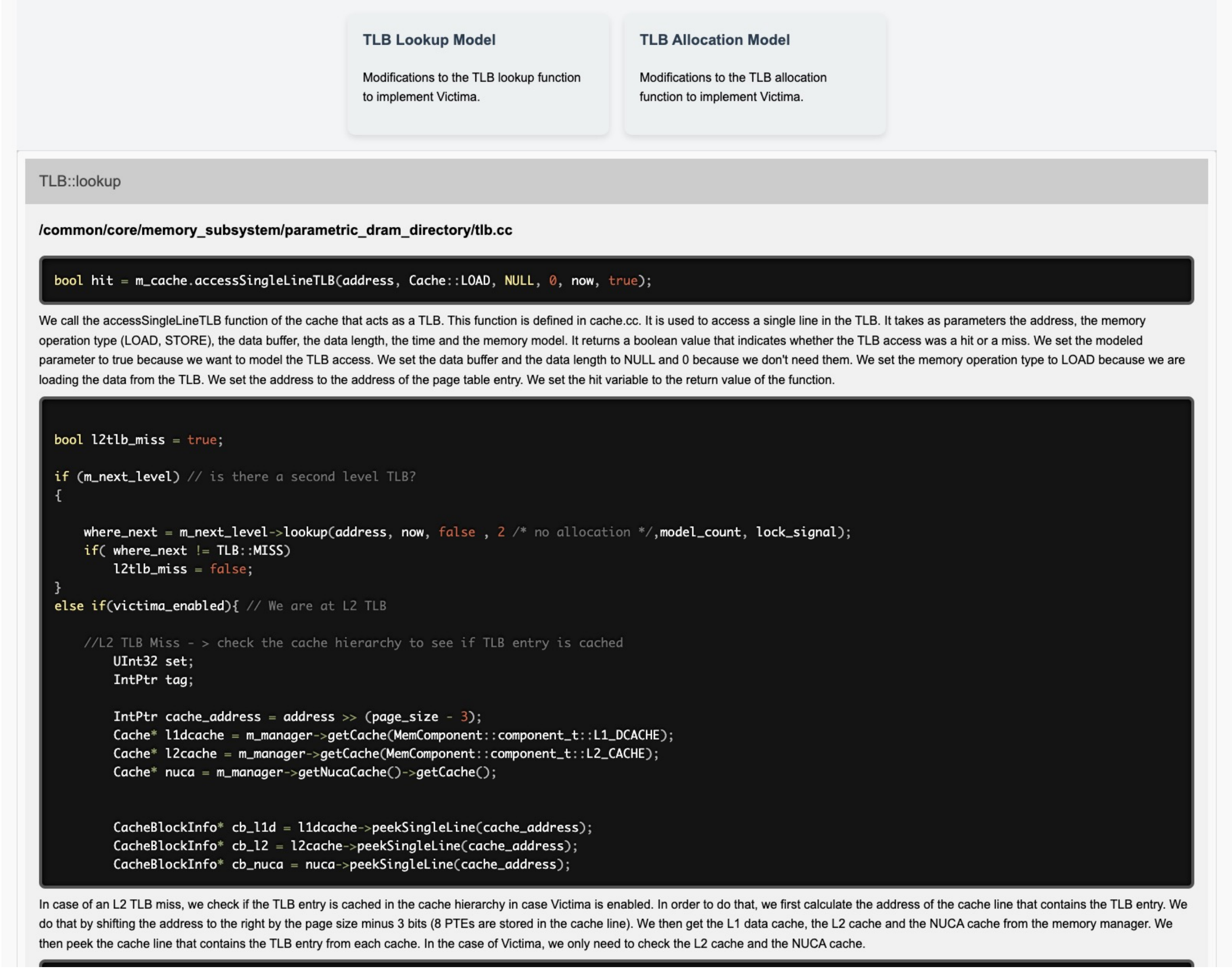
Final Artifact MICRO 2023 Latest
on Sep 4

+ 1 release

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

Victima is Open Source

Documentation is available



TLB Lookup Model

Modifications to the TLB lookup function to implement Victima.

TLB Allocation Model

Modifications to the TLB allocation function to implement Victima.

TLB::lookup

/common/core/memory_subsystem/parametric_dram_directory/tlb.cc

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

We call the accessSingleLineTLB function of the cache that acts as a TLB. This function is defined in cache.cc. It is used to access a single line in the TLB. It takes as parameters the address, the memory operation type (LOAD, STORE), the data buffer, the data length, the time and the memory model. It returns a boolean value that indicates whether the TLB access was a hit or a miss. We set the modeled parameter to true because we want to model the TLB access. We set the data buffer and the data length to NULL and 0 because we don't need them. We set the memory operation type to LOAD because we are loading the data from the TLB. We set the address to the address of the page table entry. We set the hit variable to the return value of the function.

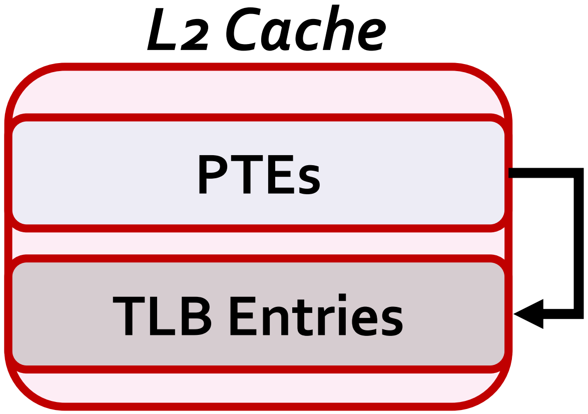
```
bool l2tlb_miss = true;

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_miss = false;
}
else if(victima_enabled){ // We are 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* l1dcache = m_manager->getCache(MemComponent::component_t::L1_DCACHE);
        Cache* l2cache = m_manager->getCache(MemComponent::component_t::L2_CACHE);
        Cache* nuca = m_manager->getNucaCache()->getCache();


        CacheBlockInfo* cb_l1d = l1dcache->peekSingleLine(cache_address);
        CacheBlockInfo* cb_l2 = l2cache->peekSingleLine(cache_address);
        CacheBlockInfo* cb_nuca = nuca->peekSingleLine(cache_address);
```

In case of an L2 TLB miss, we check if the TLB entry is cached in the cache hierarchy in case Victima 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 (8 PTEs are stored in the cache line). We then get the L1 data cache, the L2 cache and the NUCA cache from the memory manager. We then peek the cache line that contains the TLB entry from each cache. In the case of Victima, we only need to check the L2 cache and the NUCA cache.

Conclusion

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

Arxiv

Artifacts Available acm V1.1

Artifacts Evaluated acm Functional V1.1

Artifacts Evaluated acm Reusable V1.1

Results Reproduced acm V1.1

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

SAFARI
SAFARI Research Group
safari.ethz.ch

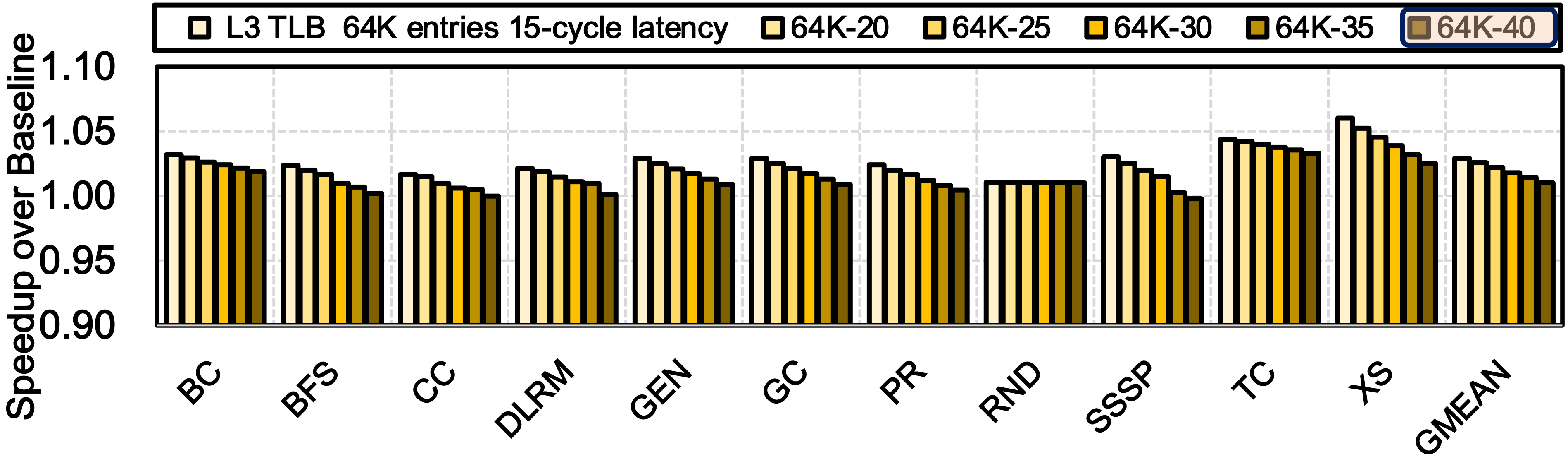
ETH zürich

HUAWEI

NTNU

Adding Hardware-based L3 TLB

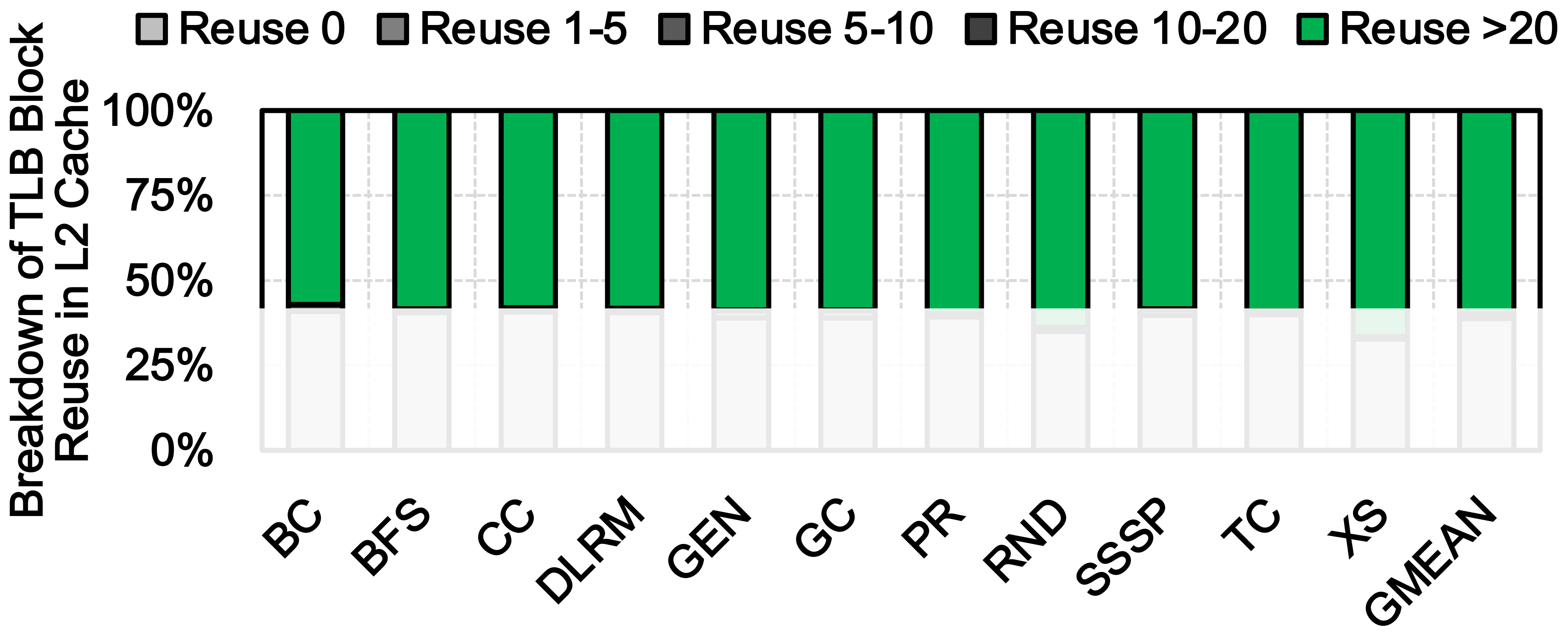
Cacti7.0 Estimation



High access latency offsets the potential performance gains of hardware L3 TLB

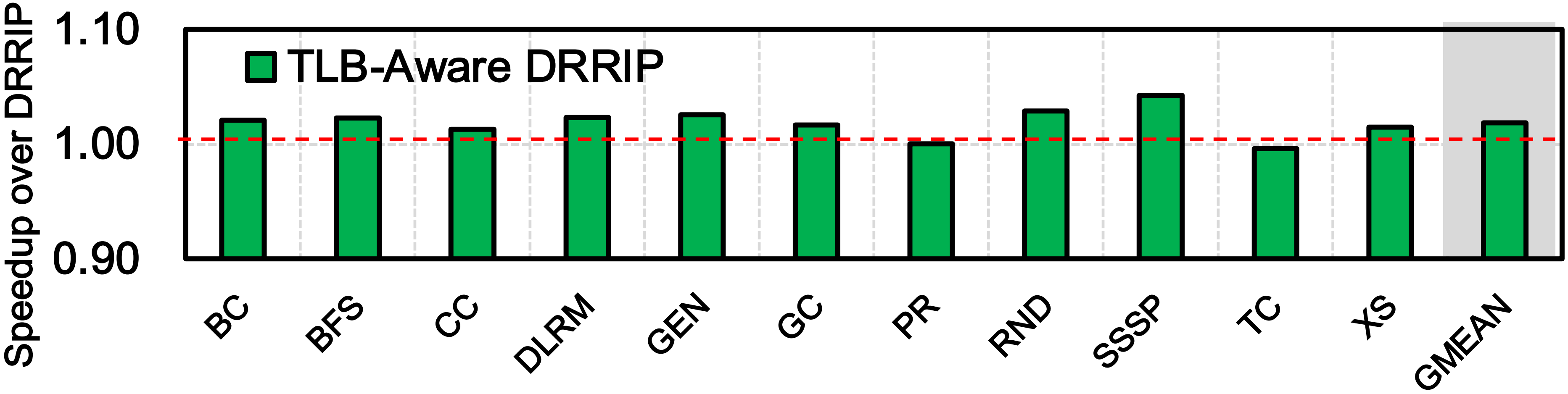
SAFARI

Reuse of TLB Blocks in L2 Cache



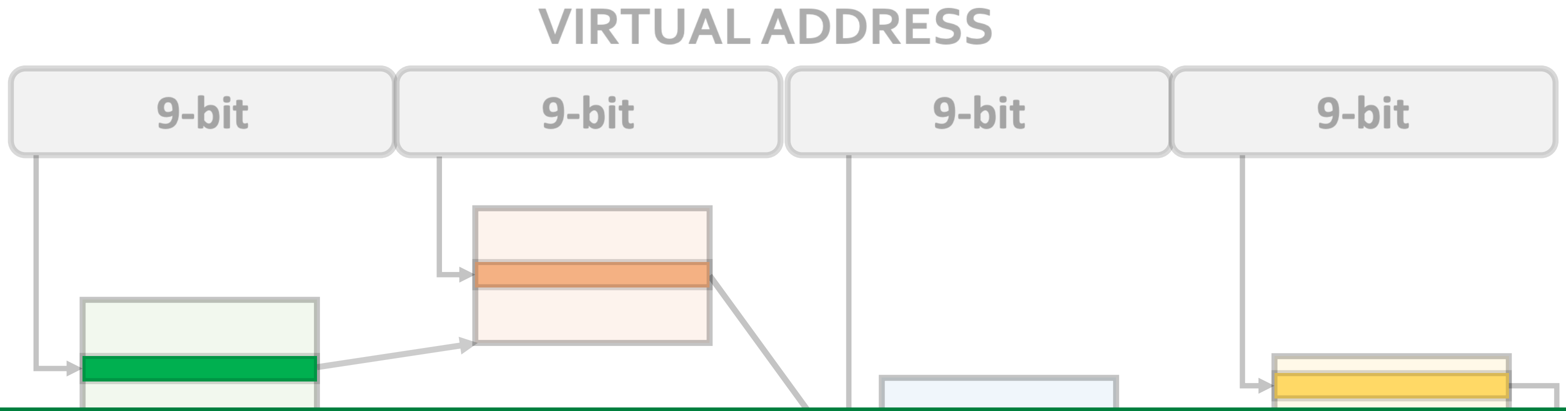
More than 60% of TLB blocks experience reuse higher than 20

Sensitivity to L2 Cache Replacement Policy



Employing the TLB-aware DRRIP leads to 1.8% higher performance compared to the conventional DDRIP

Page Table Walk in X86-64

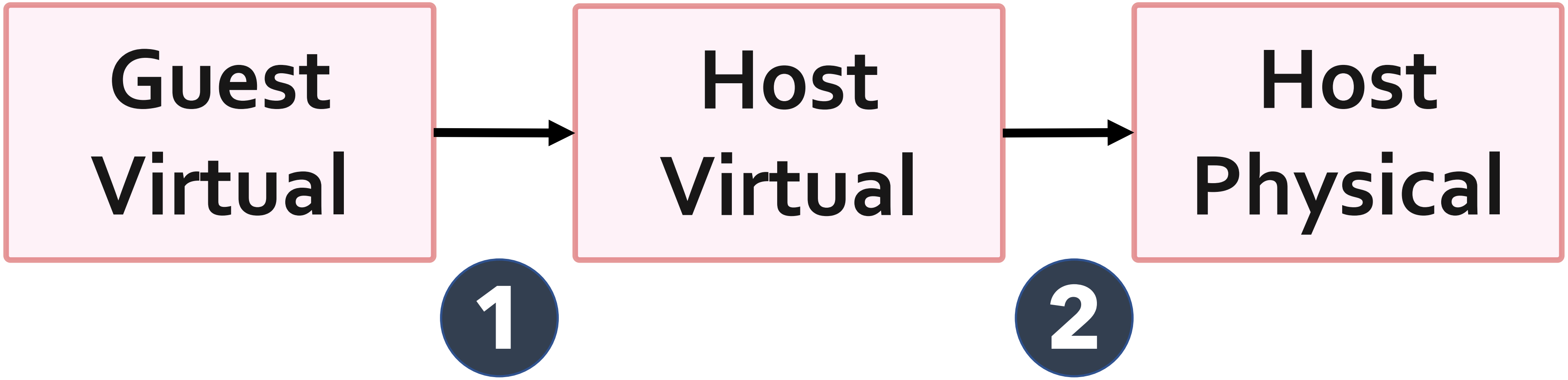


__Four__ sequential memory accesses during a page table walk in x86-64

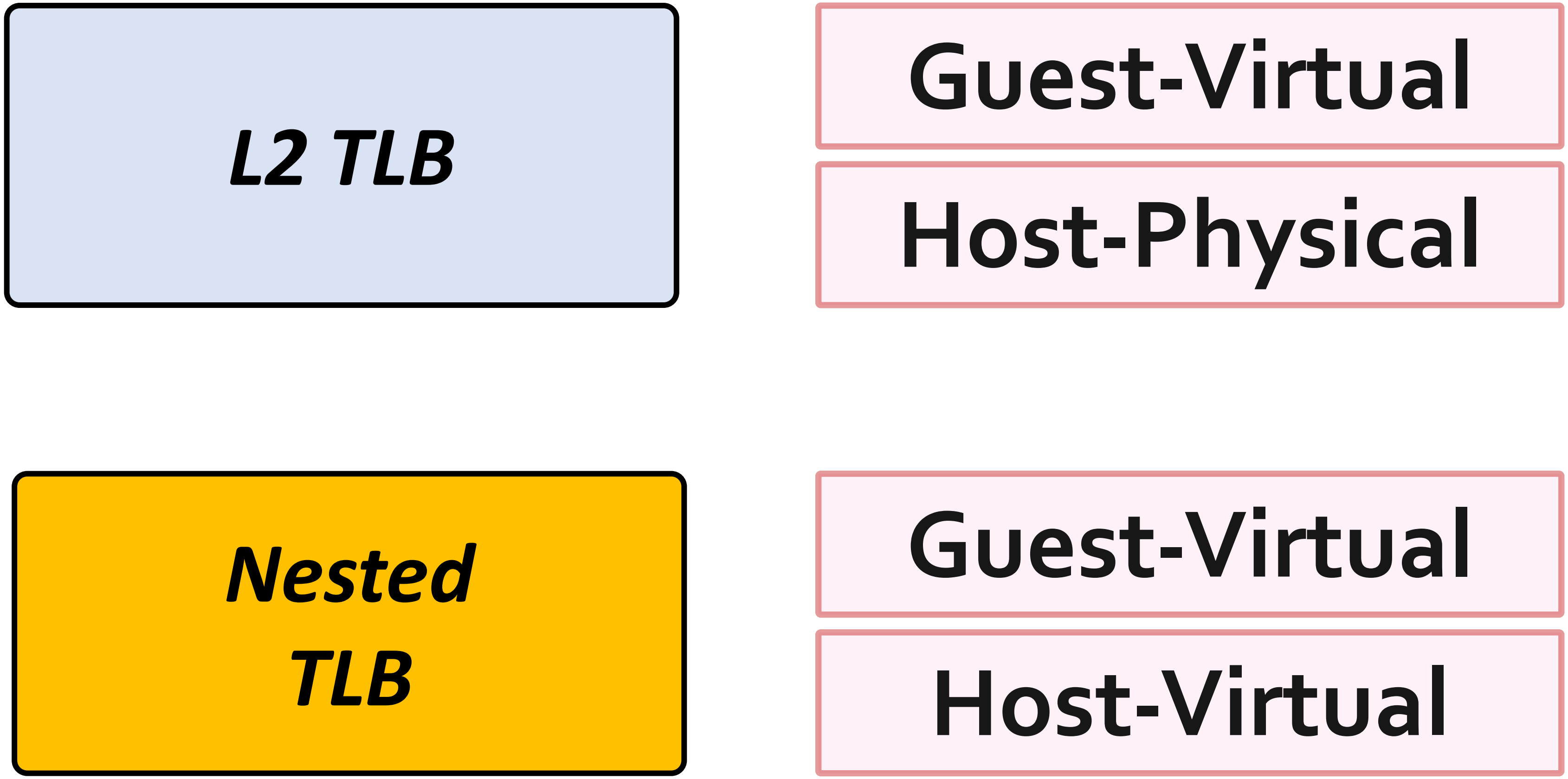
Up to 24 memory accesses in virtualized environments

Virtualized Environments

Two-level address translation



Virtualized Environments



L2 TLB

Guest-Virtual

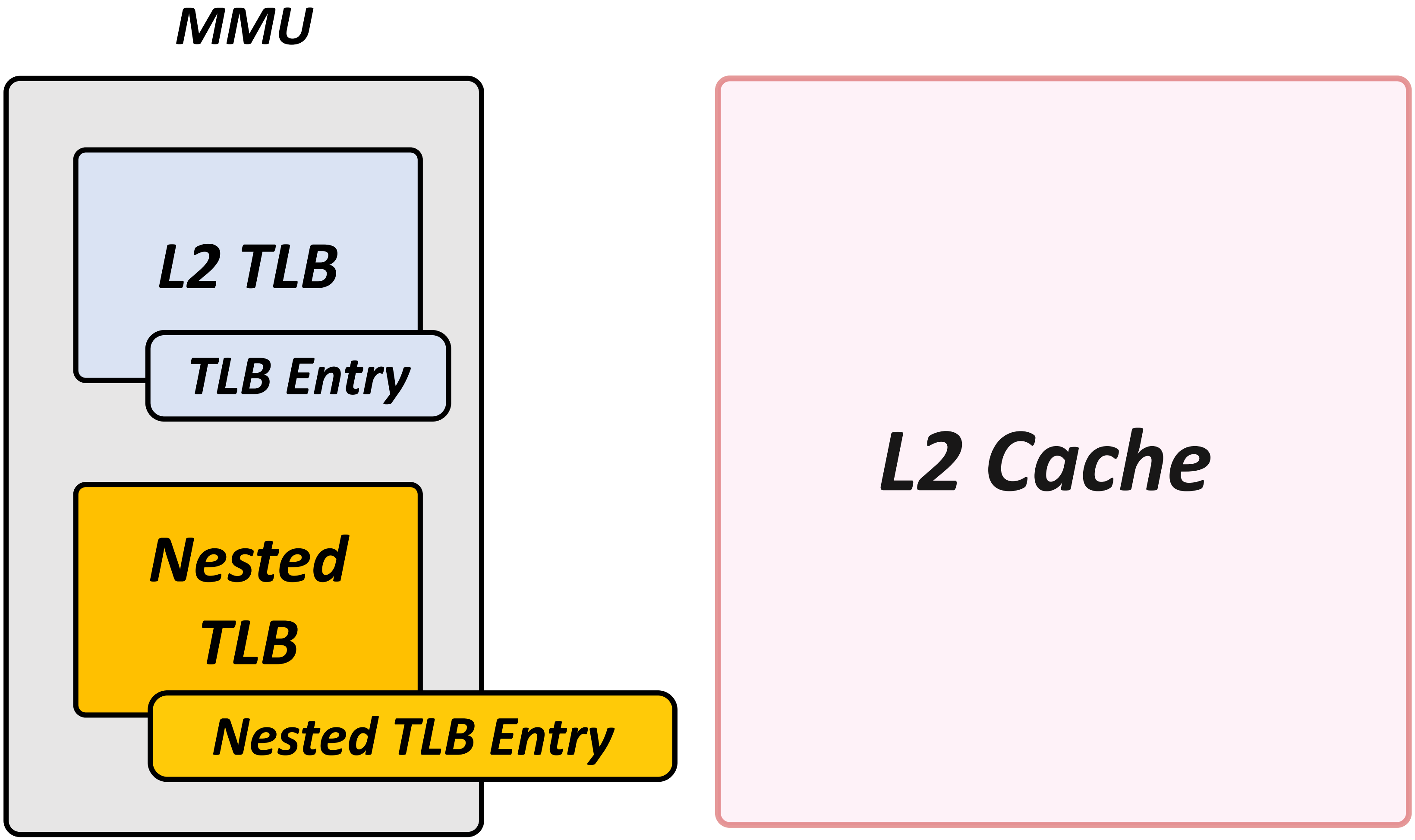
Host-Physical

Nested TLB

Guest-Virtual

Host-Virtual

Virtualized Environments



PTW-CP Feature Set

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

Feature engineering to find minimal set of useful features

PTW-CP Exploration

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

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

Configurations in Virtualized Environments

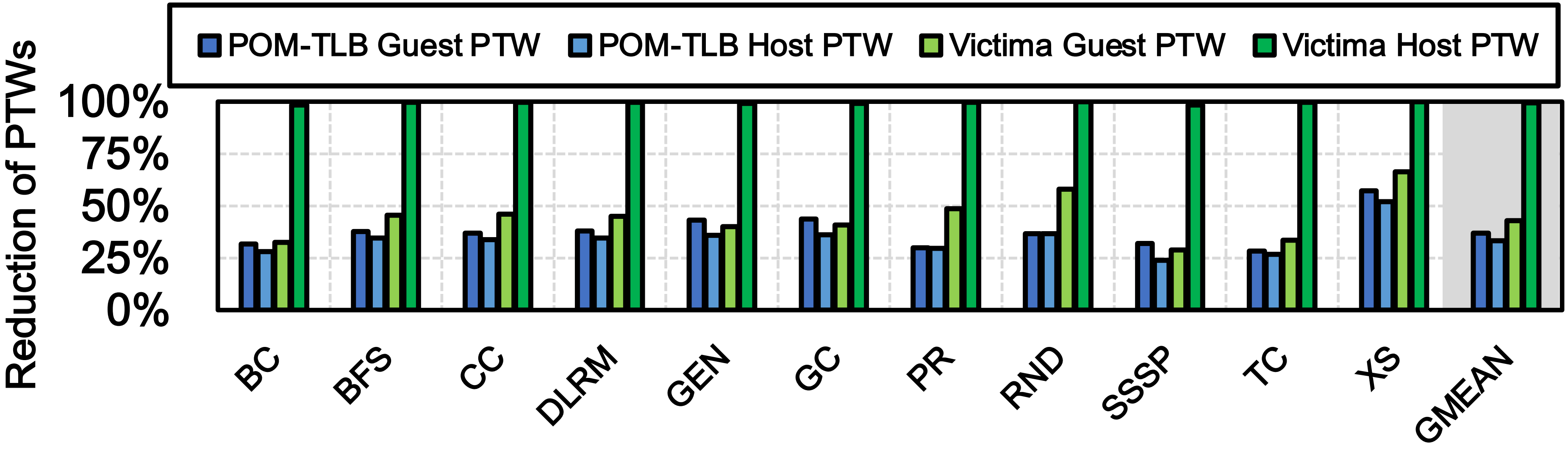
- **Nested Paging[1]:** Baseline system that performs Nested PTWs
- **POM-TLB[2]**: System with 64K-entry software-managed L3 TLB
- **Ideal Shadow Paging[3]**: System that employs an ideal version of Shadow Paging
- **Victima**: Caching both TLB and Nested TLB entries in the L2 cache

[1] Bhargava et al. "Accelerating two-dimensional page walks for virtualized systems" ASPLOS 2008

[2] Ryoo et al. "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB" ISCA 2017

[3] "Agile paging: Exceeding the best of Nested and Shadow Paging" ISCA 2016

Reduction in Host and Guest PTWs



SAFARI