Address Translation

HW3 Code Review on Monday!
HW4 released next week!

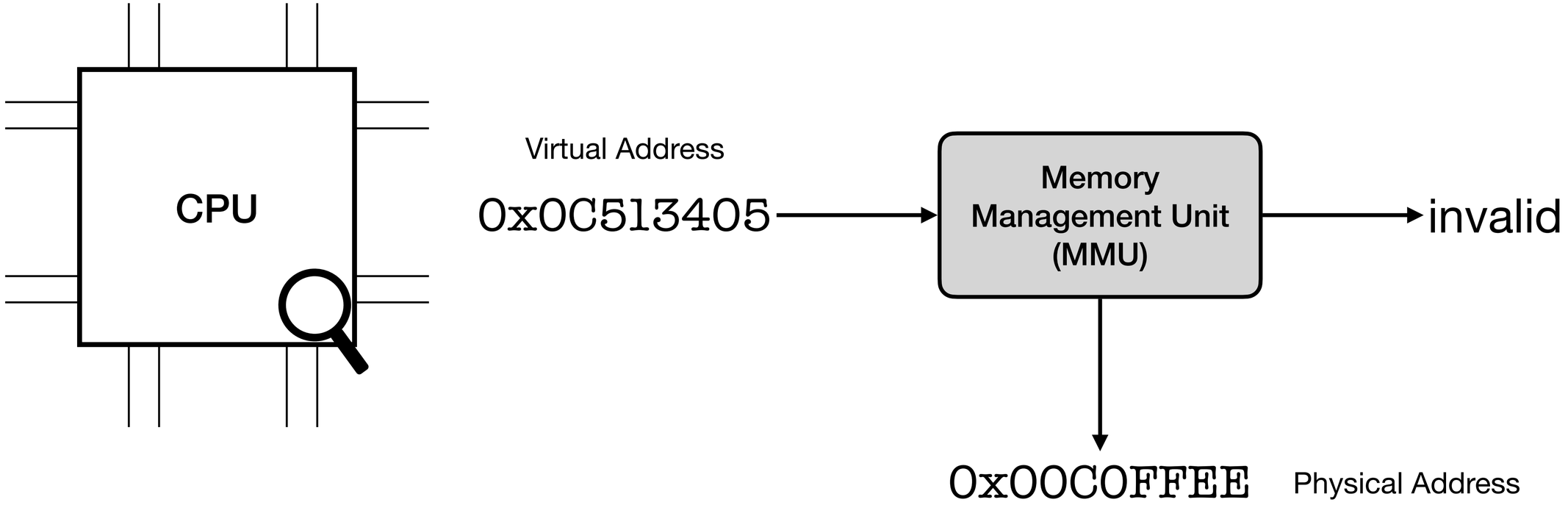
Outline

- Virtual address translation review
- TLB considerations and challenges
- Modern approaches towards address translation

Review: The State of Physical Memory

- From CS105, we know that data is stored in memory at a *page*-granularity (e.g., 4kB chunks of data) ➡️ data may be program data, program instructions, shared libraries, memory-mapped files, etc…
- To ensure that applications cannot reach data that is not mapped to its process, the operating system and hardware jointly need to implement *address translation* so that only certain addresses are reachable by certain applications
- To ensure that memory addresses are all reachable and used efficiently, the operating system needs to implement efficient *memory allocation* so that memory remains efficiently reachable with minimal fragmentation

Review: Address Translation

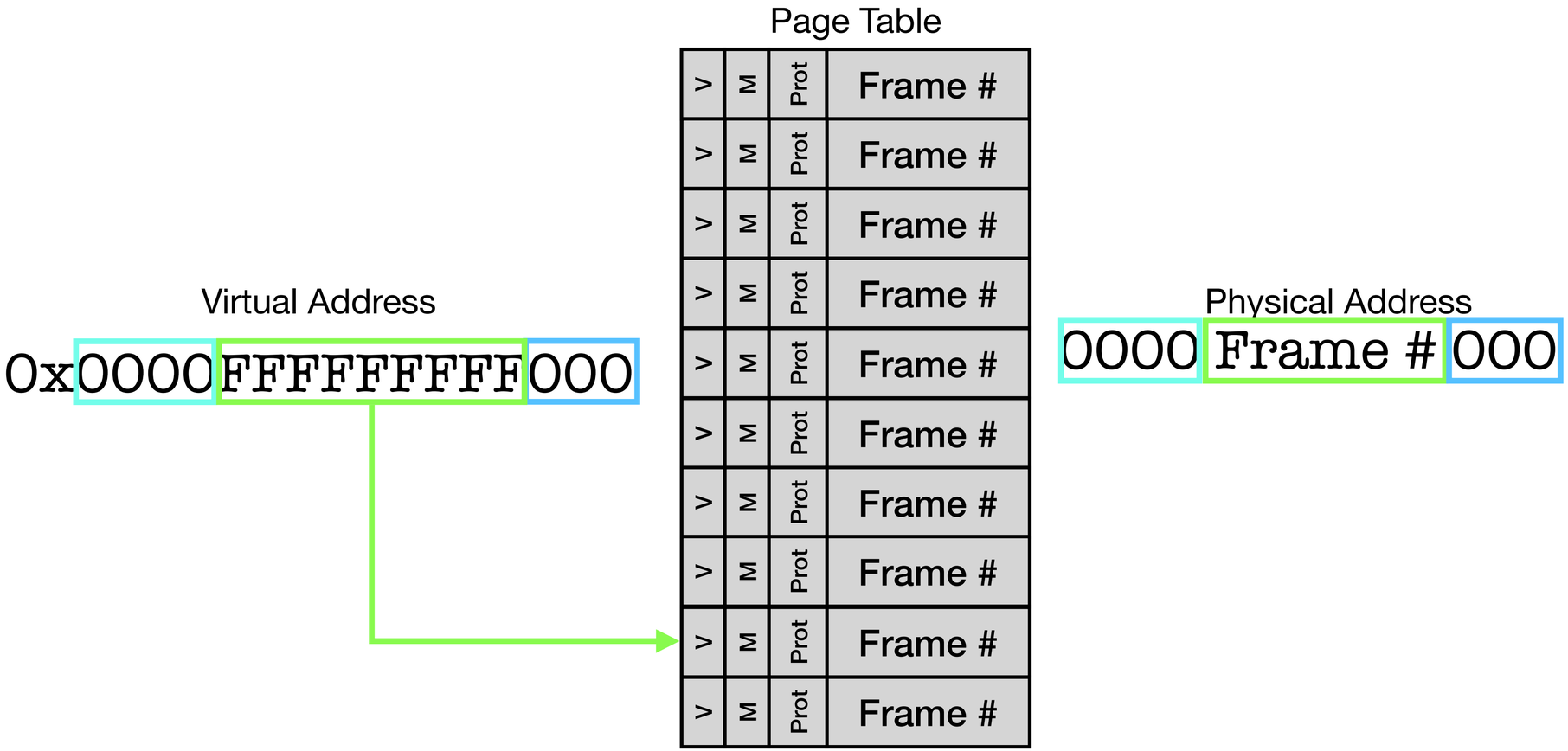


Review: Address Translation

In x86_64, addresses are 48 bits long so the 16 most significant bits are unused

If we use page sizes of 4KiB, then the 12 least significant bits describe the offset within the page

The remaining bits in the virtual address are used to derive the physical address



Review: Address Translation

Valid: indicates whether or not the PTE contains a valid Frame #

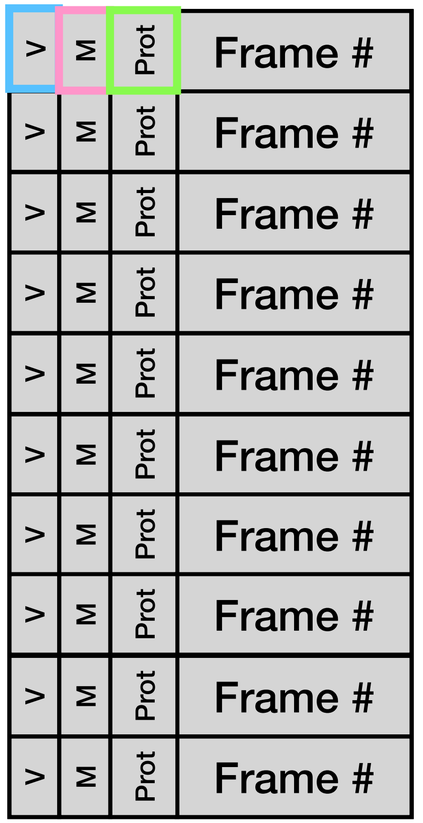
If not, page fault!

Modified: indicates whether or not the page has been written to

Useful to implement CoW!

Protection Bits: indicates whether or not the page is readable, writable, executable

Allows MMU to enforce permissions!



Single-Level Page Tables

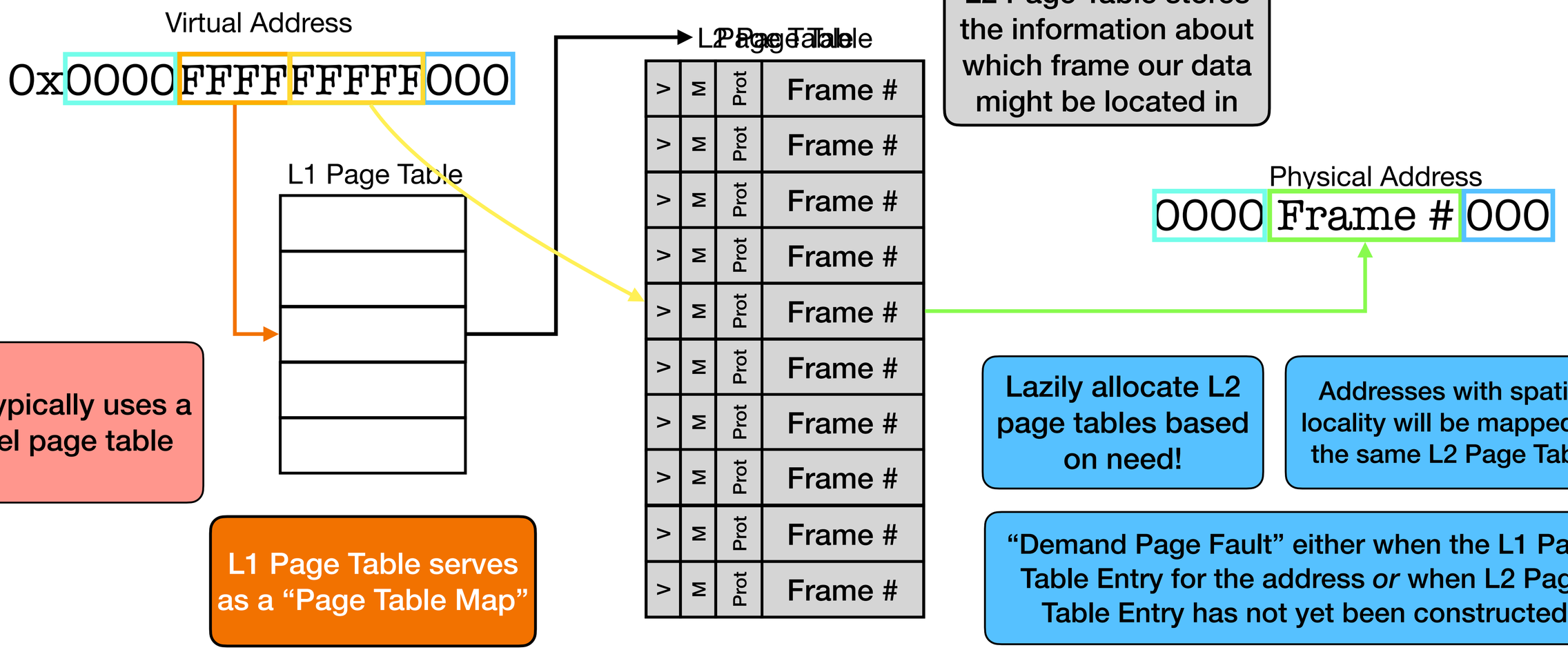
- To implement the mapping of virtual addresses to physical addresses, our operating system needs to define the size of the regions by which it will allocate “chunks” of physical memory ➡️ most modern systems do this at a *page* granularity where pages are defined to be 4KiB
- The least significant bits of a virtual address (from 0 to the page size) describe the offset of the byte to lookup within a page, so these remain the same across the virtual and physical addresses
- The remaining bits are used to provide some context of how to map into the *page table*, which is a metadata structure maintained by the operating system used to implement the mapping from virtual-to-physical addresses and describe the state of the frame (page in physical memory) that is being mapped to

Chat with your neighbor(s)!

Suppose we wanted to implement virtualization by leveraging a single-level page table. What are the drawbacks of this approach? You may want to think about the initialization of a virtual address space.

We need to allocate space for a page table entry for every possible page in the virtual address space!

Multi-Level Page Tables



Multi-Level Page Tables

- To avoid having to pre-allocate a page table entry for every page in the virtual address space, we can create levels of indirection between different page table levels
- The L1 Page Table (Page Table Map) is used to point to the appropriate L2 Page Table for this address ➡️ the upper-most “translation bits” are used to index into the L1 Page Table, whose data indicates the index of the appropriate L2 Page Table
- The result of this design decision is that the L*n* page tables are much smaller than the single-level page table equivalent ➡️ only when the whole of the address space is mapped will the page table utilize that much space

Chat with your neighbor(s)!

Describe the worst case translation of a virtual to physical address using an *n*-level page table. What would the operating system need to do at each step?

L1 Page Table entry is invalid (page fault), create the L2 Page Table

L2 Page Table entry is invalid (page fault), create the L3 Page Table

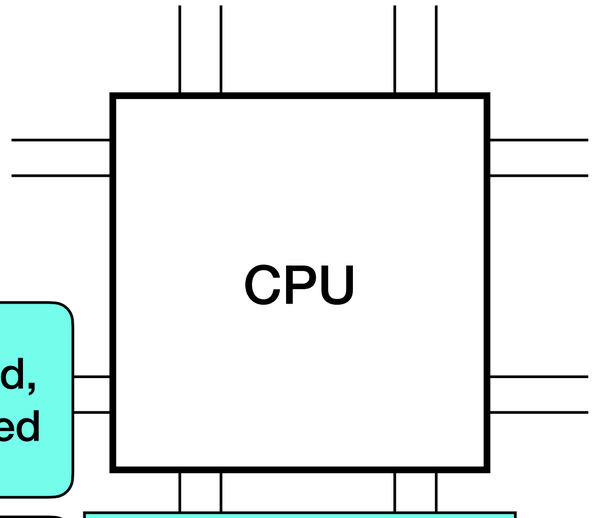
...

L*n* Page Table Entry is invalid (page fault), create the mapping to an available frame

Translation Lookaside Buffer (TLB)

- In practice, to accelerate the “page table walk” procedure (e.g., traversing the levels of the page table tree), the memory management unit (MMU) keeps a small cache to help accelerate the lookup procedure ➡️ this cache is called the *translation look aside buffer* (TLB)
- Page Table Entries are mapped to cache lines in the TLB using the memory addresses of the PTE to produce the tag/index, just as would happen in a traditional cache
- On TLB miss, the page table walk is performed using the version of the PTE in memory (just like a normal cache miss would occur)
- Page Table walks require several lookups, so several TLB entries are used for a single page table walk ➡️ inefficient use of space!

TLBs in Modern Processors



Virtual Address

0x0C513405

Memory Management Unit (MMU)

Virtually indexed, physically tagged

Can start lookup for data before translation finishes

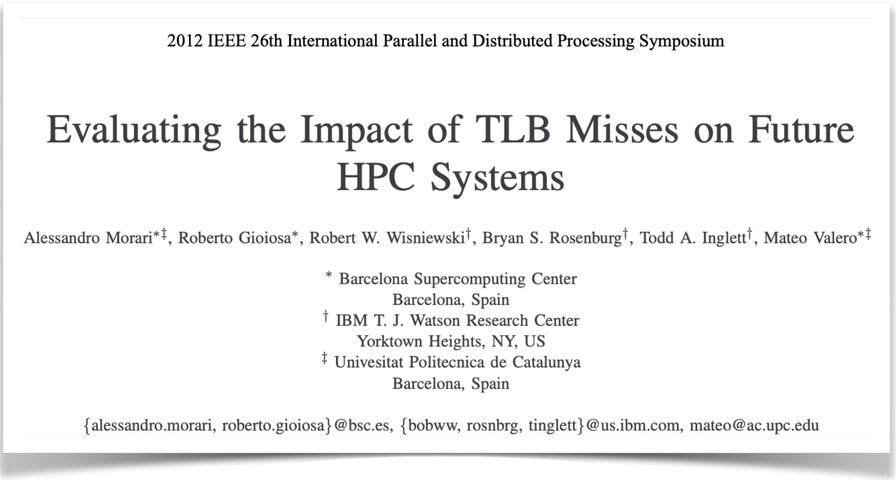
L1 Cache

Instruction Translation Lookaside Buffer (iTLB)

Data Translation Lookaside Buffer (dTLB)

L2 Translation Lookaside Buffer (L2TLB)

Translation Lookaside Buffer (TLB)



2012 IEEE 26th International Parallel and Distributed Processing Symposium

Evaluating the Impact of TLB Misses on Future HPC Systems

Alessandro Morari*‡, Roberto Gioiosa*, Robert W. Wisniewski†, Bryan S. Rosenburg†, Todd A. Inglett†, Mateo Valero*‡

* Barcelona Supercomputing Center
Barcelona, Spain
† IBM T. J. Watson Research Center
Yorktown Heights, NY, US
‡ Univesitat Politecnica de Catalunya
Barcelona, Spain

{alessandro.morari, roberto.gioiosa}@bsc.es, {bobww, rosnbrg, tinglett}@us.ibm.com, mateo@ac.upc.edu

Apps	Cores	
amg	128	202,718.24
amg	256	217,190.96
amg	512	229,597.23
amg	1024	232,221.82
amg	2048	243,566.54
amg	4096	NA
irs	125	NA
irs	216	NA
irs	512	NA
irs	1000	NA
irs	1728	NA
irs	4096	NA
lammps	128	1,017,630.05
lammps	256	1,061,163.23
lammps	512	1,021,721.25
lammps	1024	1,085,015.14
lammps	2048	1,040,772.45
lammps	4096	NA
umt	128	NA
umt	256	NA
umt	512	NA
umt	1024	NA
umt	2048	NA
gtc	128	NA
gtc	256	NA
gtc	512	NA
gtc	1024	NA
gtc	2048	NA
gtc	4096	NA

TLB misses per second

By increasing the page size, we increase the number of offset bits

With fewer PTEs per walk, we improve utilization of TLB

Virtual Address

0x0000FFFFFFFFF000

Concern: TLB misses will become dominant term as applications grow in memory used…

Using Huge Pages to Improve TLB Efficiency

- Long page table walks lead to inefficient utilization of the TLB ➡️ several PTEs at the various levels of the page table each require their own line in the TLB
- Modern versions of Linux support 2MB and 1GB *huge pages* of contiguous physical memory ➡️ a huge page is translated in the same way as a normal page, but omits the lowest level of the page table walk in a 2MB huge page and avoids the two lowest levels of the page table walk with a 1GB huge page
- This means that entries in the various levels of the page table will serve different roles depending on whether they map to a huge page or to an entry at a lower level page table

Chat with your neighbor(s)!

Seeing as huge pages alleviate pressure on the TLB, certain OS developers and researchers have proposed allowing every page to be mapped as a huge page. What would be a limitation of this approach? What types of programs would trigger this limitation?

Programs that use small, non-contiguous data would suffer bad *fragmentation*!

Summary

- We use the page table to translate virtual pages/page numbers into physical frame numbers to construct a physical address
- Single-level page tables incur huge memory overhead in main memory, so using a multi-level page table allows for lazy allocation of the page table on demand page faults
- To accelerate the page table walk of a multi-level page table, the MMU hardware uses a hierarchy of TLBs to make lookups as fast as possible
- Efficient use of the TLBs is an issue that modern OS and architecture developers/researchers are actively working to improve

SoTA Approaches to Huge Pages

Mosaic Pages: Big TLB Reach with Small Pages

Krishnan Gosakan*
Rutgers University
USA
krishnan.gosakan@gmail.com

Jaehyun Han*
The University of North Carolina at Chapel Hill
USA
jaehyun@cs.unc.edu

William Kuszmaul
Massachusetts Institute of Technology
USA
kuszmaul@mit.edu

Ibrahim N. Mubarek
Carnegie Mellon University
USA
imubarek@andrew.cmu.edu

Nirjhar Mukherjee
Carnegie Mellon University
USA
nirjhar@cmu.edu

Karthik Sriram
Yale University
USA
karthik.sriram@yale.edu

Guido Tagliavini
Rutgers University
USA
guido.tag@rutgers.edu

Evan West
Stony Brook University
USA
etwest@cs.stonybrook.edu

Michael A. Bender
Stony Brook University
USA
bender@cs.stonybrook.edu

Abhishek Bhattacharjee
Yale University
USA
abhishek@cs.yale.edu

Alex Conway
VMware Research
USA
me@ajhconway.com

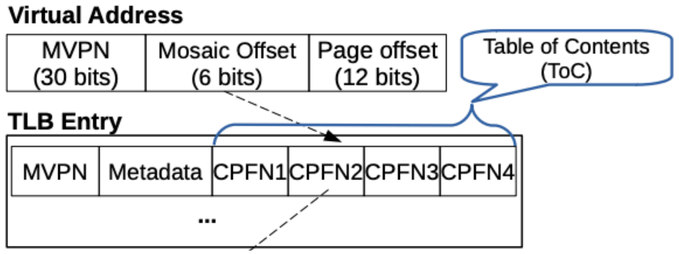
Martin Farach-Colton
Rutgers University
USA
martin@farach-colton.com

Jayneel Gandhi
Meta
USA
jayneel@meta.com

Rob Johnson
VMware Research
USA
robj@vmware.com

Sudarsun Kannan
Rutgers University
USA
sudarsun.kannan@rutgers.edu

Donald E. Porter
The University of North Carolina at Chapel Hill
USA
porter@cs.unc.edu



SoTA Approaches to Huge Pages

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

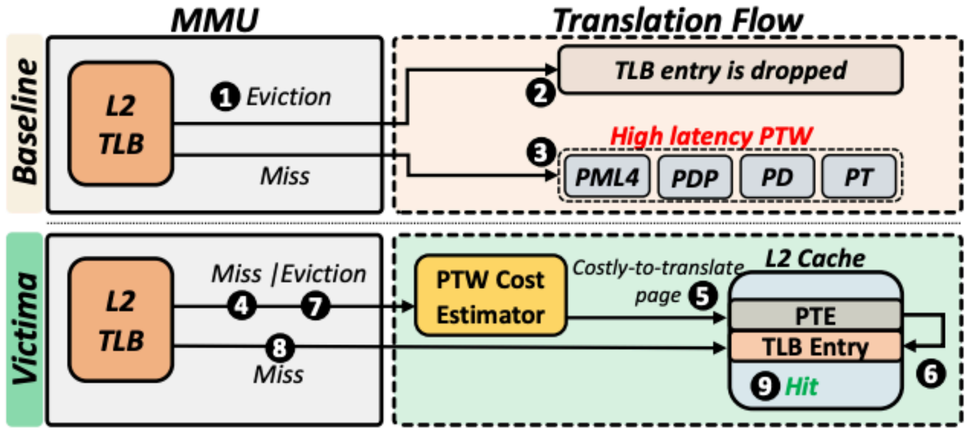


Figure 12: Address translation flow in a conventional baseline processor [51] and Victima.

SoTA Approaches to PTW Acceleration

Heliostat: Harnessing Ray Tracing Accelerators for Page Table Walks

Yuan Feng
University of California, Merced
Merced, California, USA
yfeng44@ucmerced.edu

Yuke Li
University of California, Merced
Merced, California, USA
yli304@ucmerced.edu

Jiwon Lee
Samsung Electronics
Seoul, Republic of Korea
jiwon24.lee@gmail.com

Won Woo Ro
Yonsei University
Seoul, Republic of Korea
wro@yonsei.ac.kr

Hyeran Jeon
University of California, Merced
Merced, California, USA
hjeon7@ucmerced.edu

Gaze into the Pattern: Characterizing Spatial Patterns with Internal Temporal Correlations for Hardware Prefetching

Zixiao Chen, Chentao Wu✉, Yunfei Gu, Ranhao Jia, Jie Li, and Minyi Guo
Shanghai Jiao Tong University
{chen_zx, wuct, gu.yunfei, ranhao.j}@sjtu.edu.cn, {lijie, guo-my}@cs.sjtu.edu.cn

2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)

Revelator: Rapid Data Fetching via System-Software-Guided Hash-based Speculative Address Translation

Konstantinos Kanellopoulos[1] Konstantinos Sgouras[1] Harsh Songara[1]
Andreas Kosmas Kakolyris[1] Vlad-Petru Nitu[1] Spiros Galanopoulos[1] Rahul Bera[1]
Konstantina Koliogeorgi[1] Rakesh Kumar[2] Onur Mutlu[1]
[1]ETH Zürich [2]NTNU

Marching Page Walks: Batching and Concurrent Page Table Walks for Enhancing GPU Throughput

Jiwon Lee*, Gun Ko*, Myung Kuk Yoon†, Ipoom Jeong‡*, Yunho Oh§, and Won Woo Ro*
*Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea
†Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea
‡Department of System Semiconductor Engineering, Yonsei University, Seoul, Korea
§School of Electrical Engineering, Korea University, Seoul, Korea
Emails: {jiwon.lee, gun.ko}@yonsei.ac.kr, myungkuk.yoon@ewha.ac.kr,
ipoom@yonsei.ac.kr, yunho_oh@korea.ac.kr, wro@yonsei.ac.kr