Εικονική Μνήμη & Μετάφραση Διευθύνσεων

Πηγές/Βιβλιογραφία

- Krste Asanovic, “Address Translation”, CS 152/252 Computer Architecture and Engineering, EECS Berkeley, 2020
  - https://inst.eecs.berkeley.edu/~cs152/sp20/lectures/L08-AddressTranslation.pdf
  - https://inst.eecs.berkeley.edu/~cs152/sp20/lectures/L09-VirtualMemory.pdf
- “Computer Architecture: A Quantitative Approach”, J. L. Hennessy, D. A. Patterson, Morgan Kaufmann Publishers, INC. 6th Edition, 2017.
- “Appendix L: Advanced Concepts on Address Translation”, Abhishek Bhattacharjee
  - www.cs.yale.edu/homes/abhishek/abhishek-appendix-l.pdf

VM features track historical uses

- Bare machine, only physical addresses
  - One program owned entire machine
- Batch-style multiprogramming
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (supports *swapping* entire programs but not demand-paged virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived
- Time sharing
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory

Virtual Memory Use Today

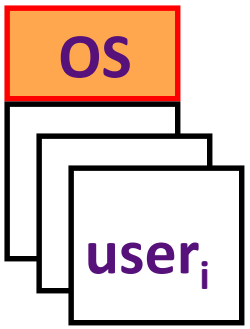
- Servers/desktops/laptops/smartphones have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Most embedded processors and DSPs provide physical addressing only
  - Can’t afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement precise or restartable exceptions for exposed architectures

Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

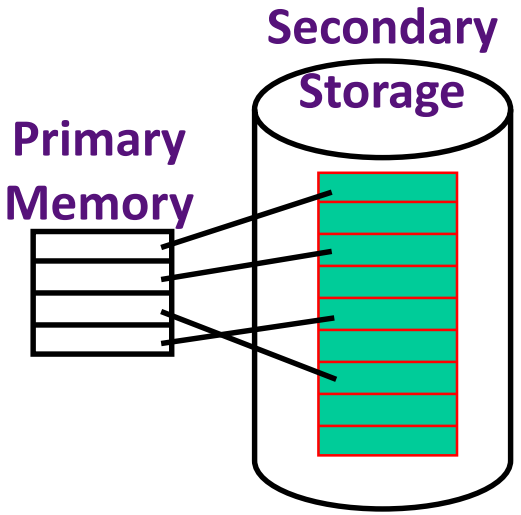
several users, each with their private address space and one or more shared address spaces



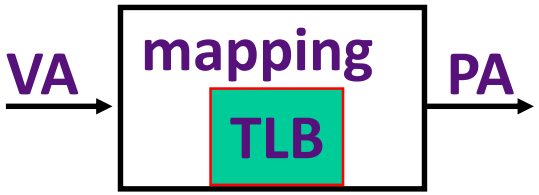
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations



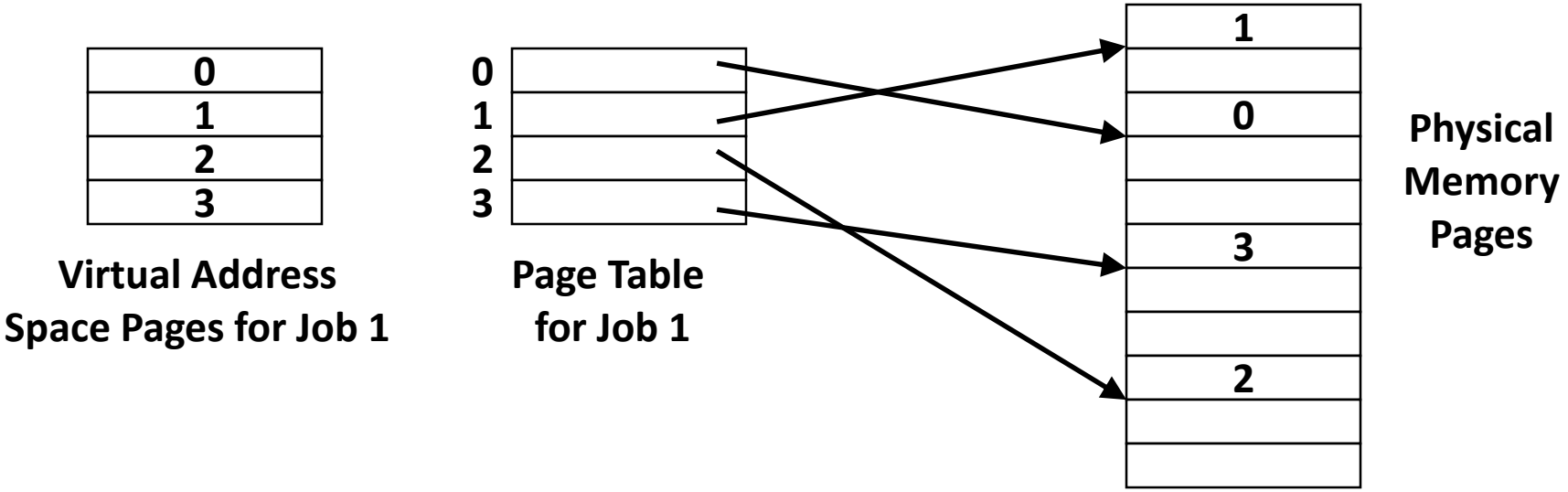
The price is address translation on each memory reference



Paged Memory Systems

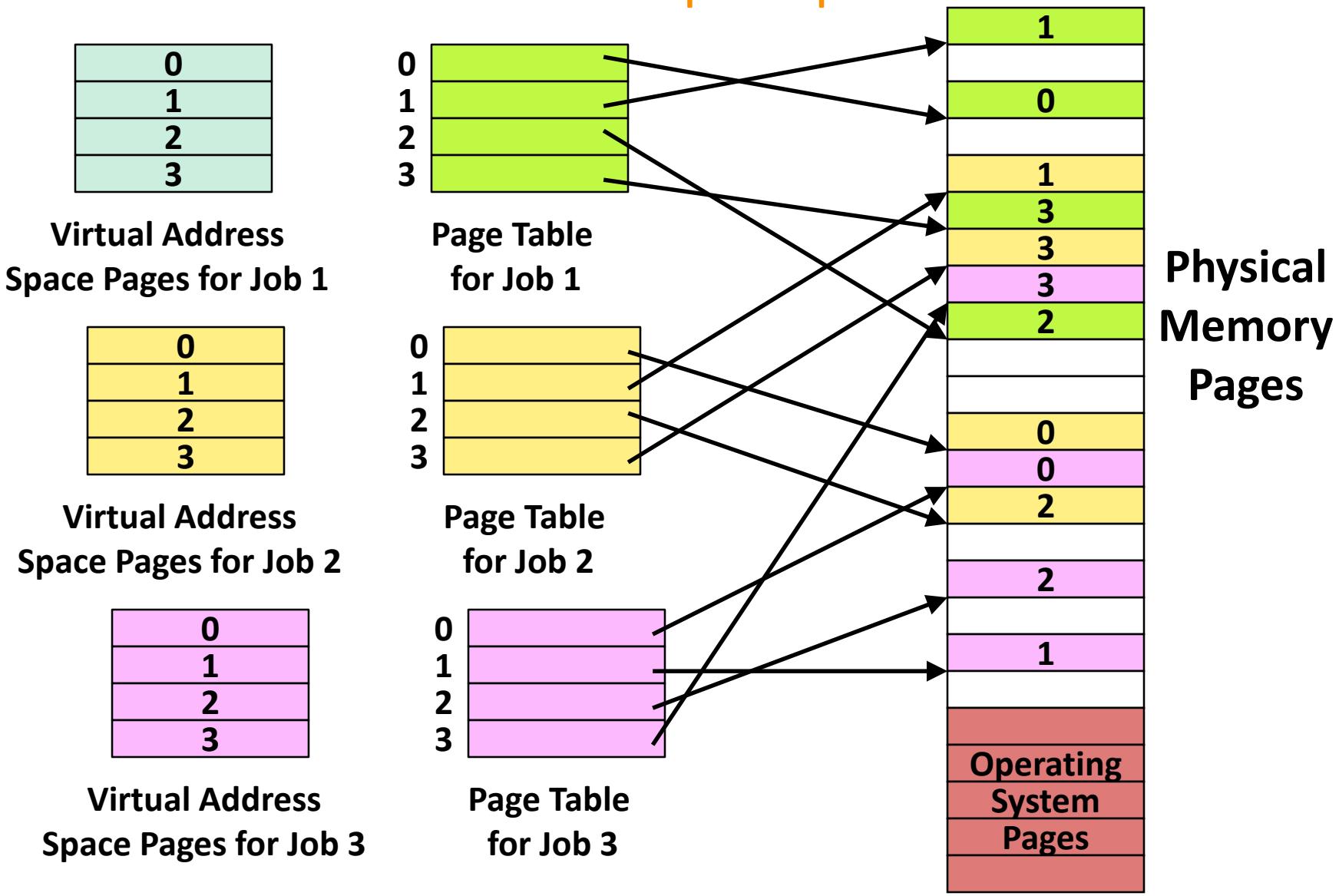
Program-generated (*virtual* or *logical*) address split into:

- Page Table contains physical address of start of each fixed-sized page in virtual address space

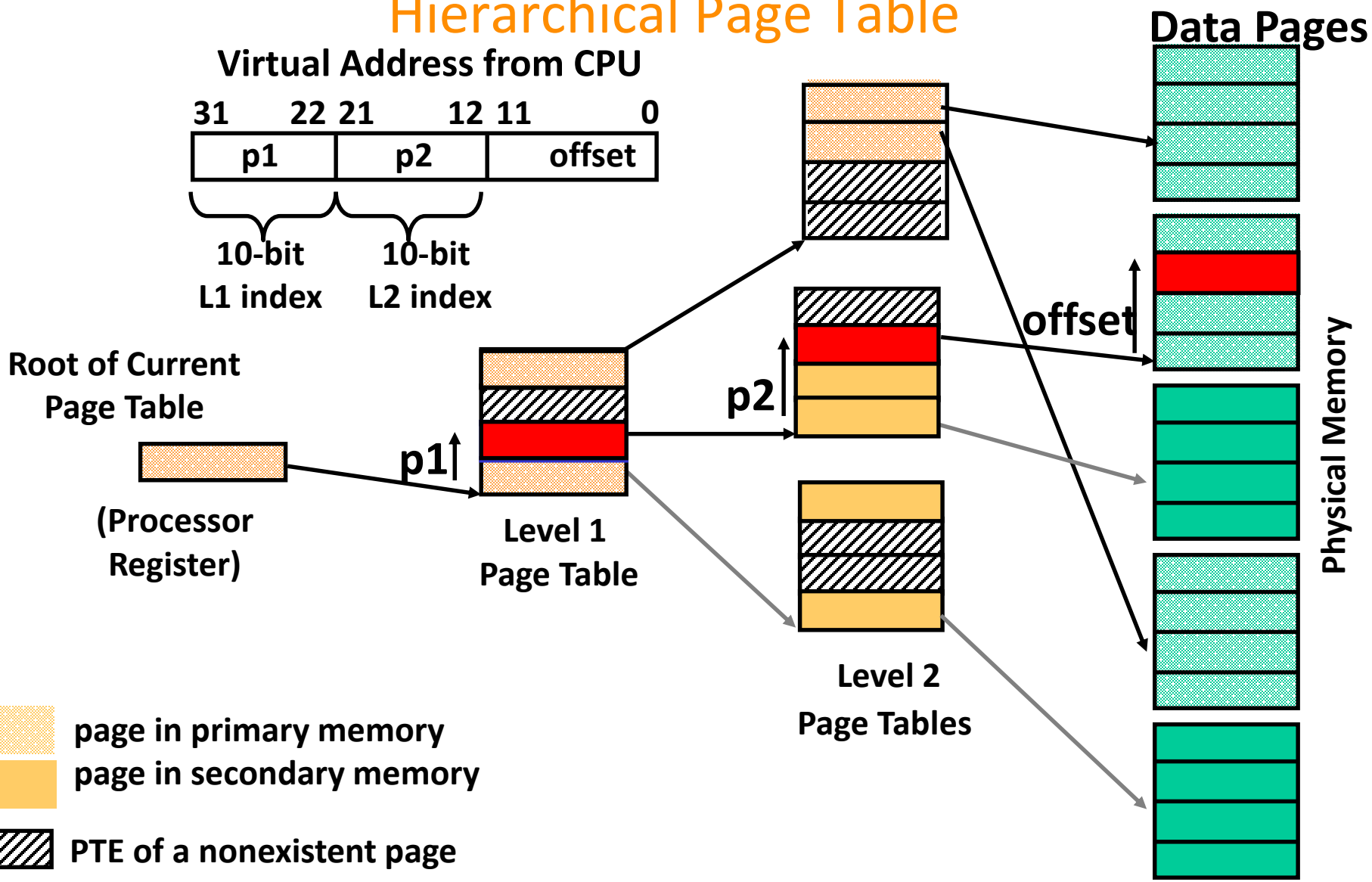


- Paging makes it possible to store a large contiguous virtual memory space using non-contiguous physical memory pages

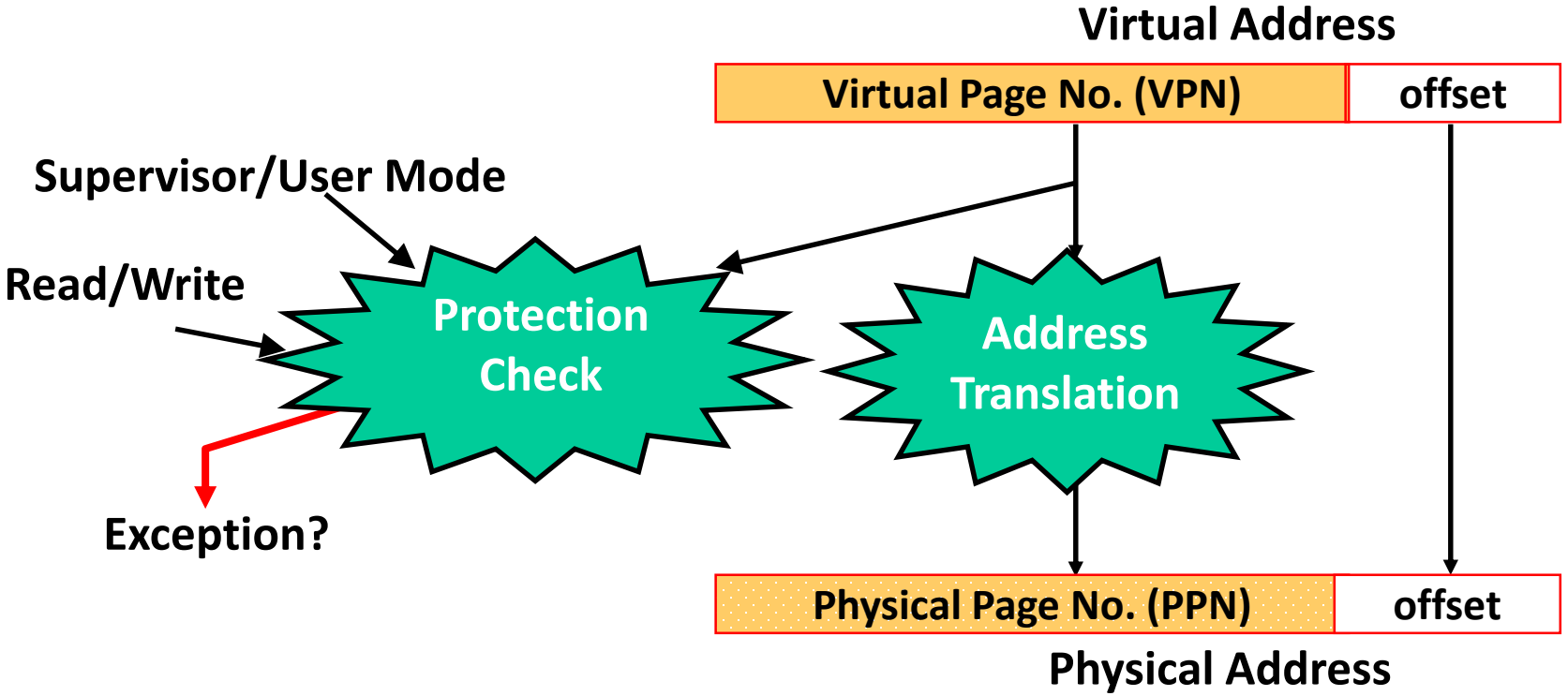
Private Address Space per User



Hierarchical Page Table



Address Translation & Protection



Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient

Page-Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    - Pseudo-LRU replacement policy, implemented in software
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

Translation-Lookaside Buffers (TLB)

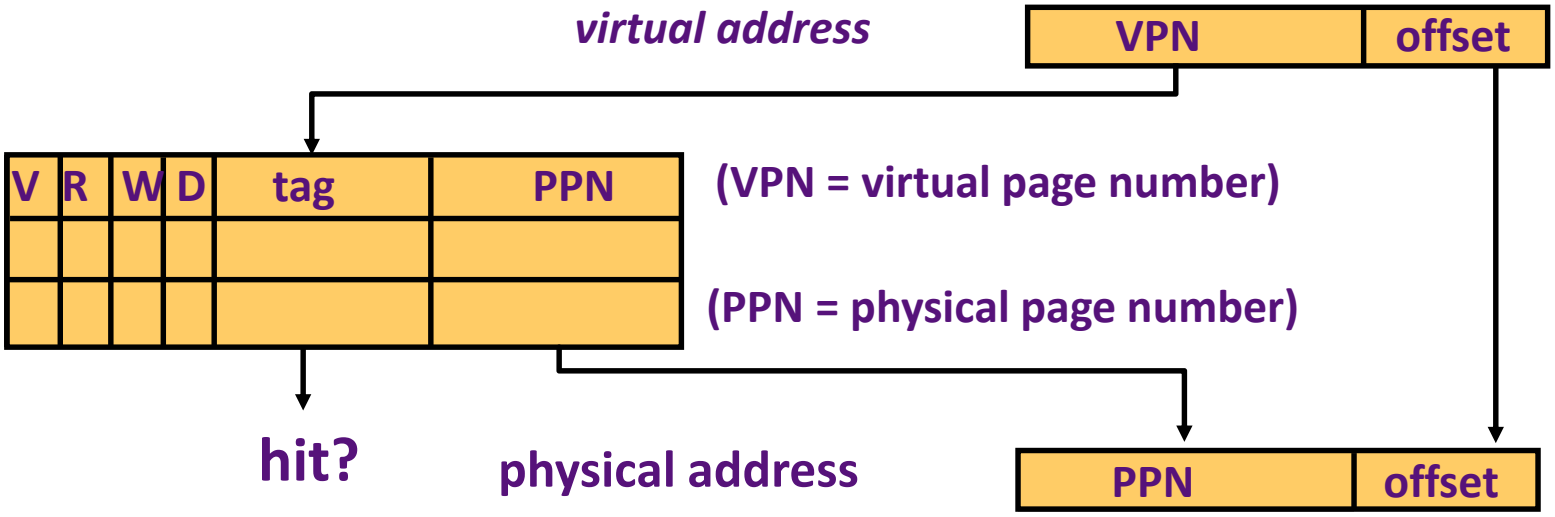
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single-Cycle Translation*

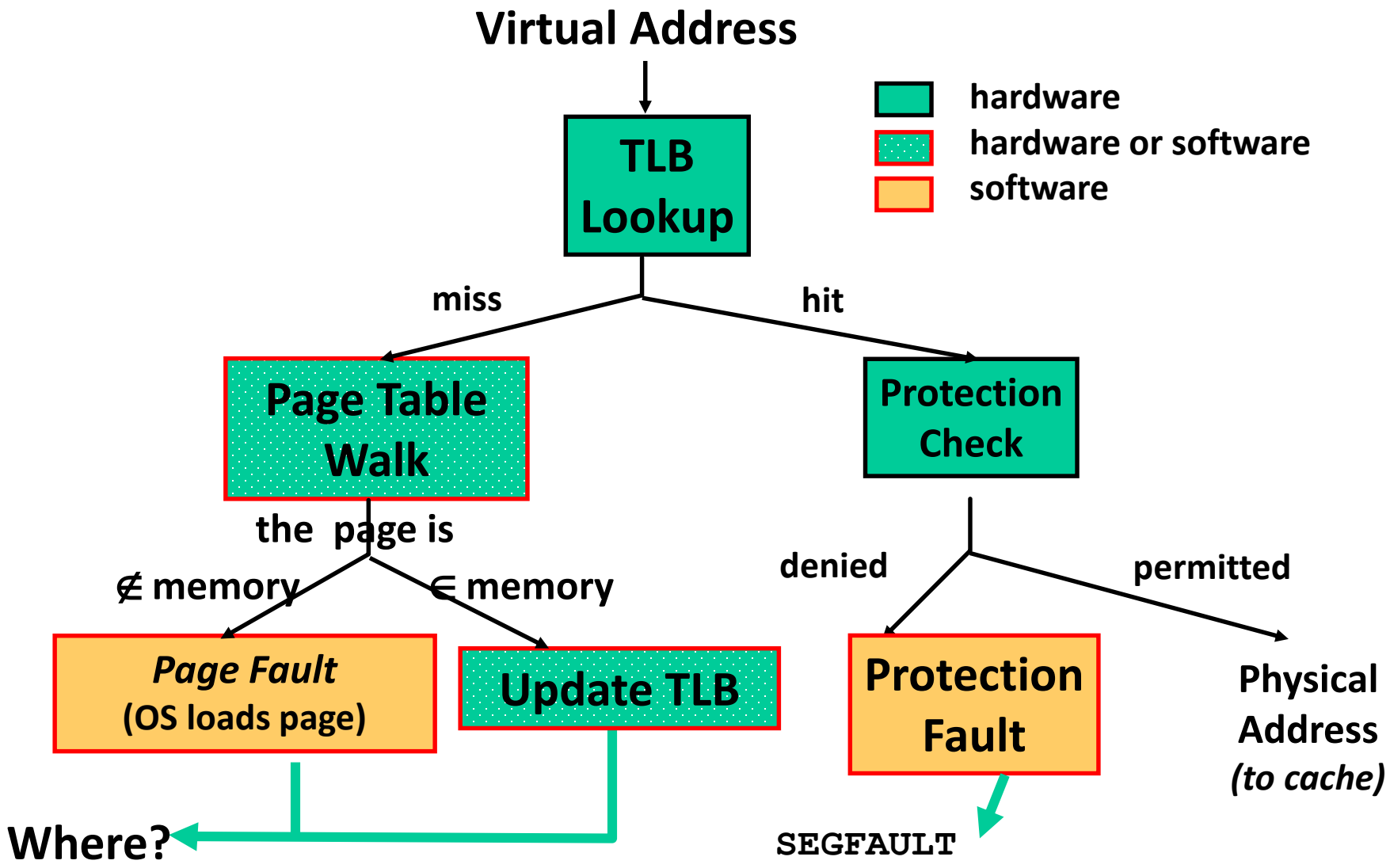
TLB miss ⇒ *Page-Table Walk to refill*



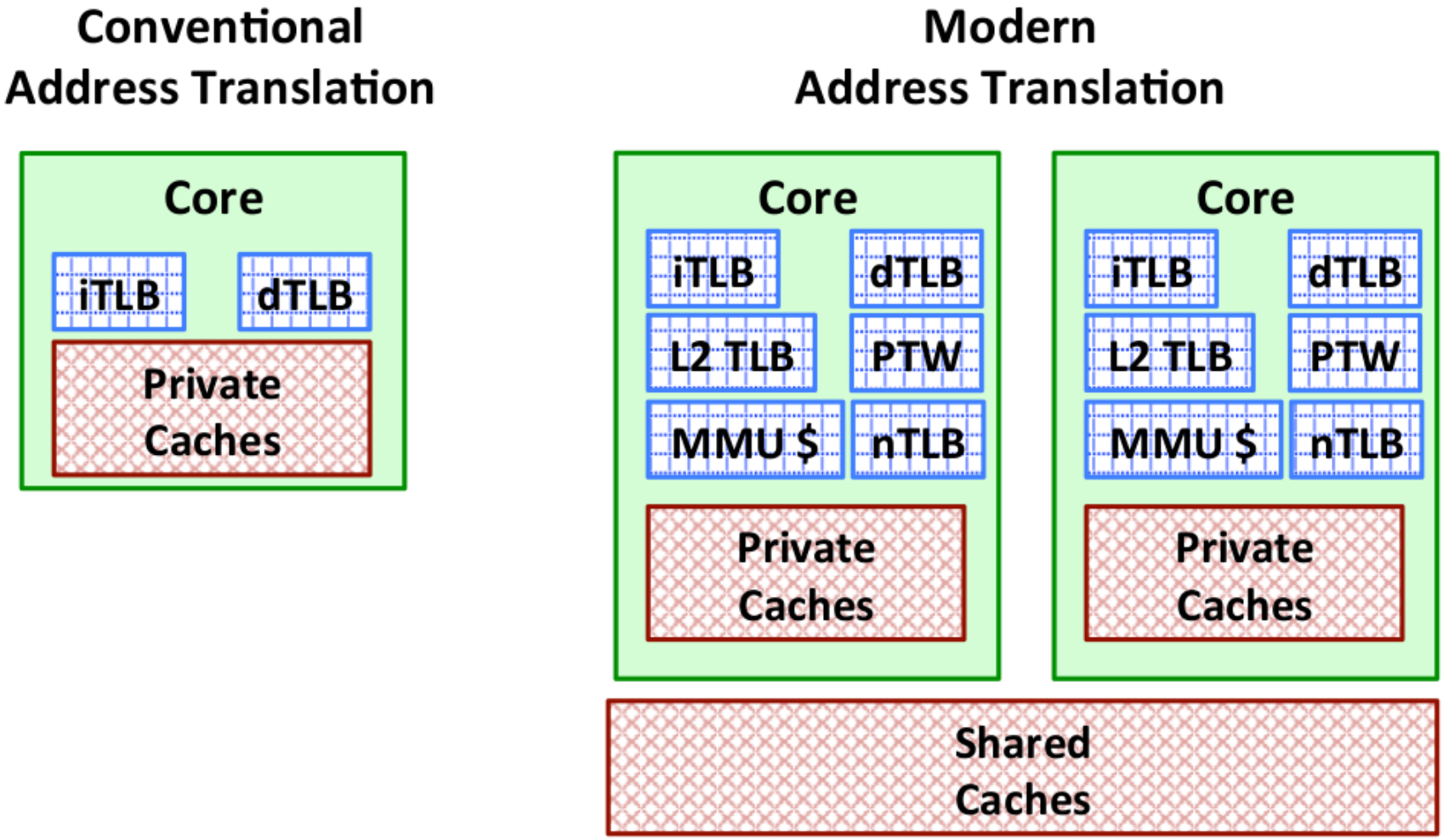
Handling a TLB Miss

- Software (*MIPS, Alpha*)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged “untranslated” addressing mode used for walk.
  - Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.
- Hardware (*SPARC v8, x86, PowerPC, RISC-V*)
  - A memory management unit (MMU) walks the page tables and reloads the TLB.
  - If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page Fault exception for the original instruction.

Address Translation: *putting it all together*



Address Translation on Modern Chips



Address Translation on Modern Chips

- **Separate L1 TLBs**
  - for instructions and data
- **Unified L2 TLBs**
  - cache translations for instructions and data
- **Hardware page table walkers (PTWs)**
  - handle TLB misses without invoking the OS
- **Memory management unit (MMU) caches**
  - accelerate TLB misses
- **Nested TLBs**
  - Support for running virtual machines

Address Translation on Modern Chips

L1 dTLBs

4KB pages: 64-entry, 4-way set-associative L1 TLB;
1 cycle access;
9 cycle miss penalty.

2MB pages: 32-entry, 4-way set-associative L1 TLB;
1 cycle access;
9 cycle miss penalty.

1GB pages: 4-entry, fully-associative L1 TLB;
1 cycle access;
9 cycle miss penalty.

L1 iTLBs

4KB pages: 128-entry, 8-way set-associative L1 TLB;
1 cycle access;
9 cycle miss penalty.

2MB pages: 16-entry, fully-associative L1 TLB;
1 cycle access;
9 cycle miss penalty.

PTWs

Supports 4 concurrent TLB misses.

Unified L2 TLBs

4KB/2MB pages: 1536-entry, 12-way set associative L2 TLB;
14 cycle access.

1GB pages: 16-entry, 4-way set- associative L2 TLB;
1 cycle access;
9 cycle miss penalty.

Others

MMU $: 32-entry, fully-associative;
1 cycle access.

nTLB: 16-entry, fully-associative; 1 cycle access (from microbenchmark).

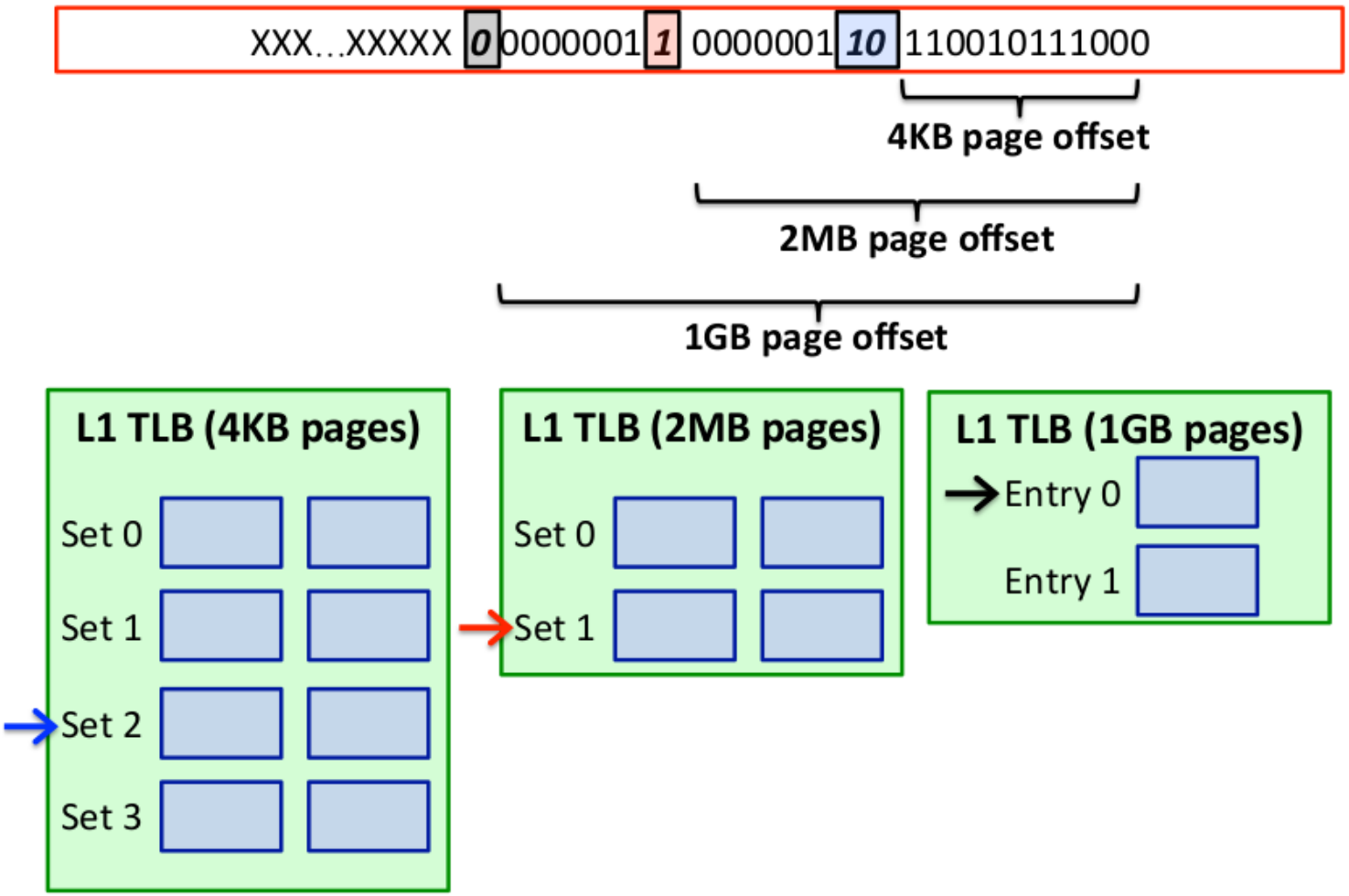
L1 TLBs – Separate Instruction and Data TLBs

- Modern superscalar out-of-order pipelines can require several concurrent instruction and data virtual-to-physical translations per cycle
  - Implementing separate iTLBs and dTLBs reduces the chances of pipeline hazards due to contention at the TLB from limited port count
- Instructions and data exhibit different locality of reference or reuse attributes
- Different policies among the TLB resources when supporting simultaneous multithreading (or hyperthreading) in hardware
  - Statically partitioned iTLBs for different simultaneous hardware threads
  - Dynamically partitioned dTLBs

L1 TLBs – Support for Different Page Sizes

- L1 TLBs must be fast and energy-efficient
  - TLBs reside on the critical L1 datapath of pipelines.
  - Need to meet timing constraints, with lookup and miss handling characteristics that are amenable to speed.
  - Hence, L1 TLBs are usually set-associative structures.
- Larger page sizes (commonly employed by most OSes today) enable greater effective TLB capacity
  - A single TLB entry can provide address translation for a larger (e.g., 2MB or 1GB) part of memory.
- Challenge: Supporting multiple page sizes also complicates the design of set-associative TLBs.
  - Why? Because different page sizes require a different number of page offset bits and the page size is not known at lookup time

L1 TLBs – Support for Different Page Sizes



L1 TLBs – Support for Different Page Sizes

- Problem: the page size is unknown at access time
  - And page size affects indexing
- Solution: Separate L1 TLBs for different page sizes
  - Intel Skylake → separate L1 TLBs for 4KB, 2MB, and 1GB pages
  - Translations are inserted into the ”right” TLB on misses
  - A translation can be placed in only one of the split L1 TLBs
  - On a memory reference, all L1 TLBs are looked up in parallel
- Alternative solution: Use fully-associative L1 TLB
  - One hardware structure
  - But expensive in terms of area and power

L2 TLBs

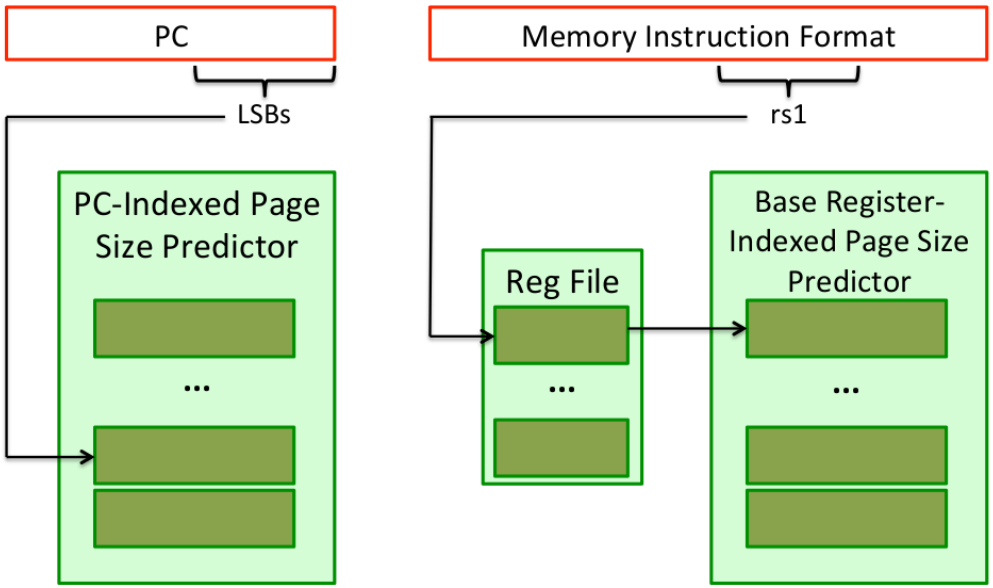
- Significantly larger than L1 TLBs
- Several important attributes to consider:
  - Access time
    - Longer access time compared to L1 TLBs due to larger size
  - Hit rate
    - Need be as high possible to counterbalance the higher access times of unified L2 TLBs
  - Multiple page size support
    - Alternative options arise now
  - Inclusive, mostly-inclusive, or exclusive designs
    - Implementation choices and tradeoffs similar to caches.

L2 TLBs – Support for Different Page Sizes

- Hash-rehashing
  - The L2 TLB is first probed (or hashed) assuming a particular page size
  - On a miss, the TLB is again probed (or rehashed) using another page size
  - This process continues with rehashed lookups for any remaining page sizes
- Skewing
  - Change the notion of set in a TLB
  - the ways of a set no longer share the same index bits and each way of a set uses its own index function
  - To support multiple page sizes, a translation maps to a subset of TLB ways depending not just on its address but also its page size.

L2 TLBs – Support for Different Page Sizes
Hash - rehashing

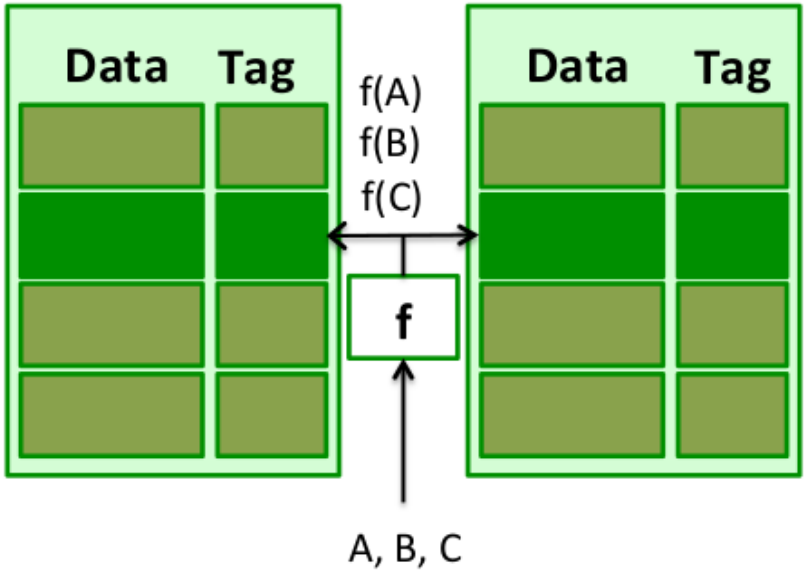
- Relative simplicity of implementation
- But now the hit time varies
- Improve hit time
  - Page size prediction (e.g., using the Program Counter or the memory instruction format)
  - Parallel lookup
  - Parallel page table walks



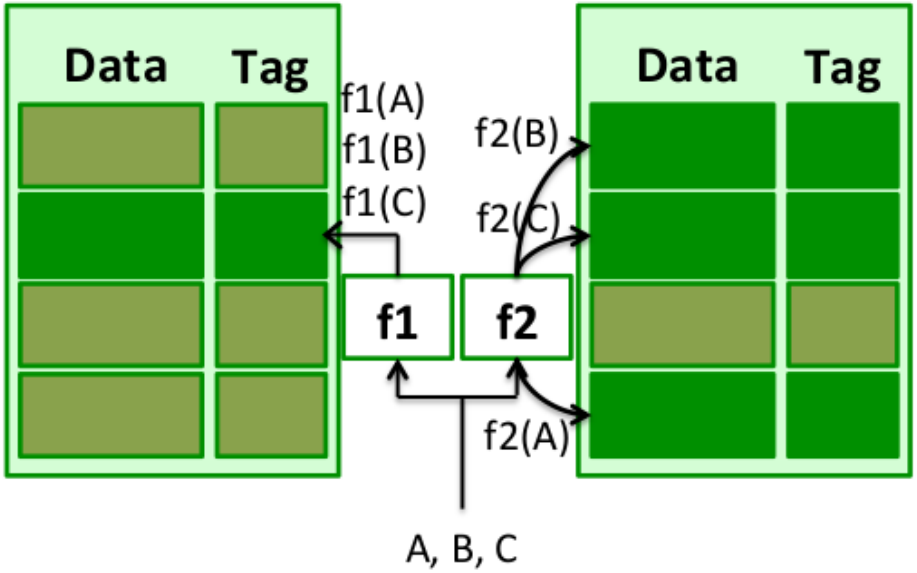
L2 TLBs – Support for Different Page Sizes Skewing

- Skew-associative TLBs can support multiple page sizes concurrently without the complications of multi-latency hit times or slow identification of TLB misses
- Require multiple hash functions, which can be complex to implement.
- Require additional area from time stamps needed for replacement policies.
- The more page sizes there are to accommodate, the lower the effective associativity per page size.

L2 TLBs – Support for Different Page Sizes
Skewing



Conventional indexing



Skew indexing

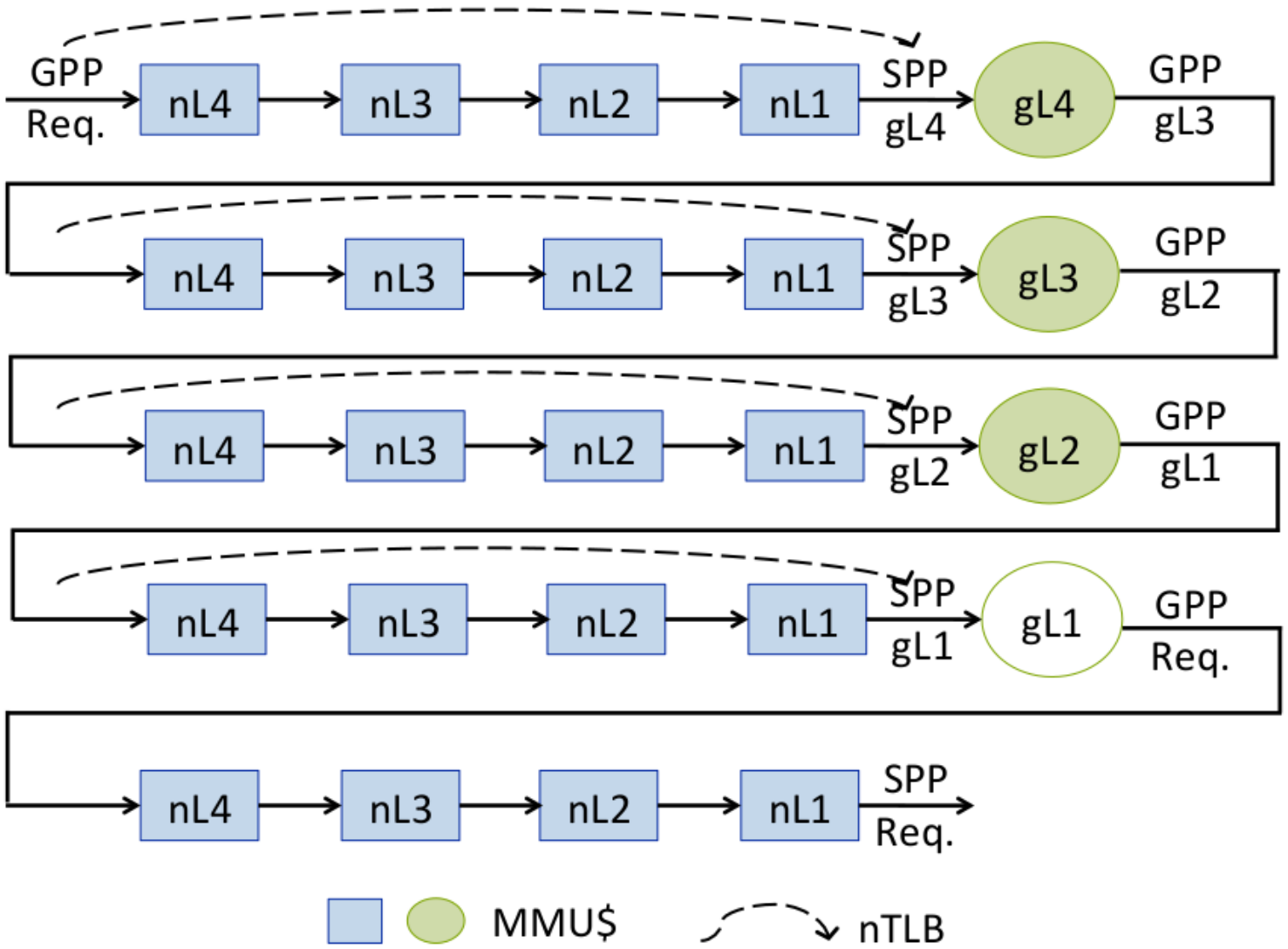
Page Table Walks

- When TLB misses occur, the page table must be searched or ”walked” to locate the desired translation.
- Can be handled either in hardware or software (OS)
- Software managed TLB
  - Flexible: Allow the OS to organize and manage page tables in a flexible manner
  - Slow: require pipelines to be context switched and OS code to be invoked for all TLB misses
- Hardware managed TLB
  - High performance
    - Overlapping TLB misses with useful work
    - Concurrently handling multiple misses
  - Reduced flexibility

2D Page Table Walks in Virtualized Execution

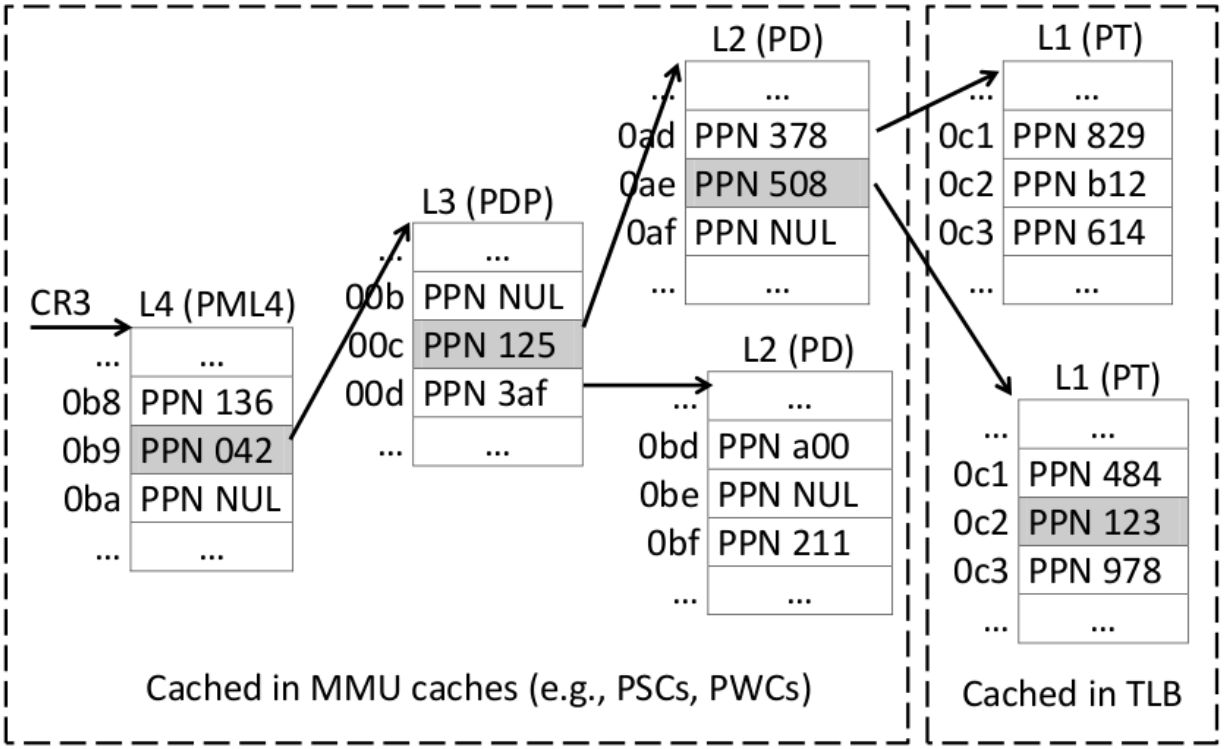
- Two-dimensional page table walks for virtualized systems
  - First, a guest virtual address must be converted to a guest physical address.
  - This guest physical address must then be converted to a system physical address.
- Many modern architectures maintain two levels of page tables to enable this two-step translation process.
  - The first one, the guest page table, translates the guest virtual pages to guest physical pages and is maintained by the guest OS.
  - The second one, the nested page table, translates the guest physical pages to the system physical pages, and is maintained by the hypervisor.
- The 2D page table walk requires more memory references than in native execution (24 vs 4 memory references in x86-64). For this reason, address translation performance is particularly problematic in virtualized environments.

2D Page Table Walks in Virtualized Execution



Memory Management Unit Caches

- Page table walks are lengthy because they require multiple sequential memory references.
- The leaf level of the page table is cached in the TLB
- The upper levels are cached in the MMU caches.

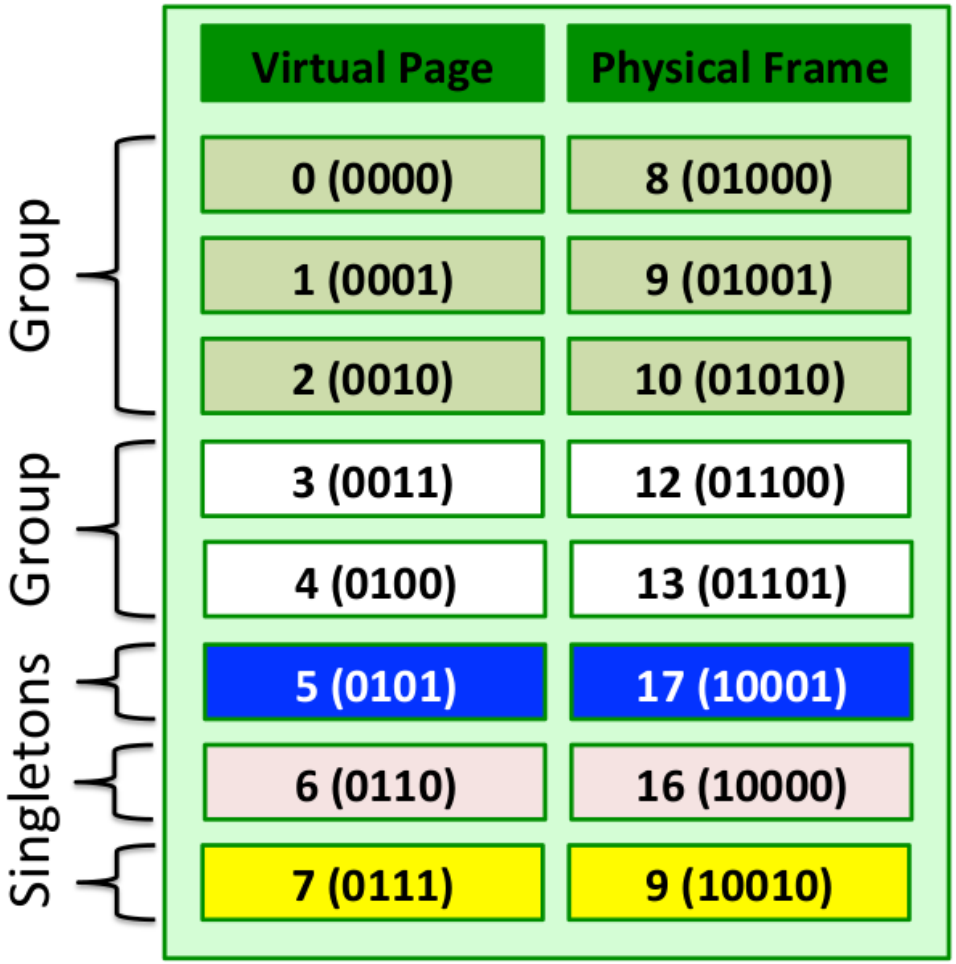


Translation Contiguity

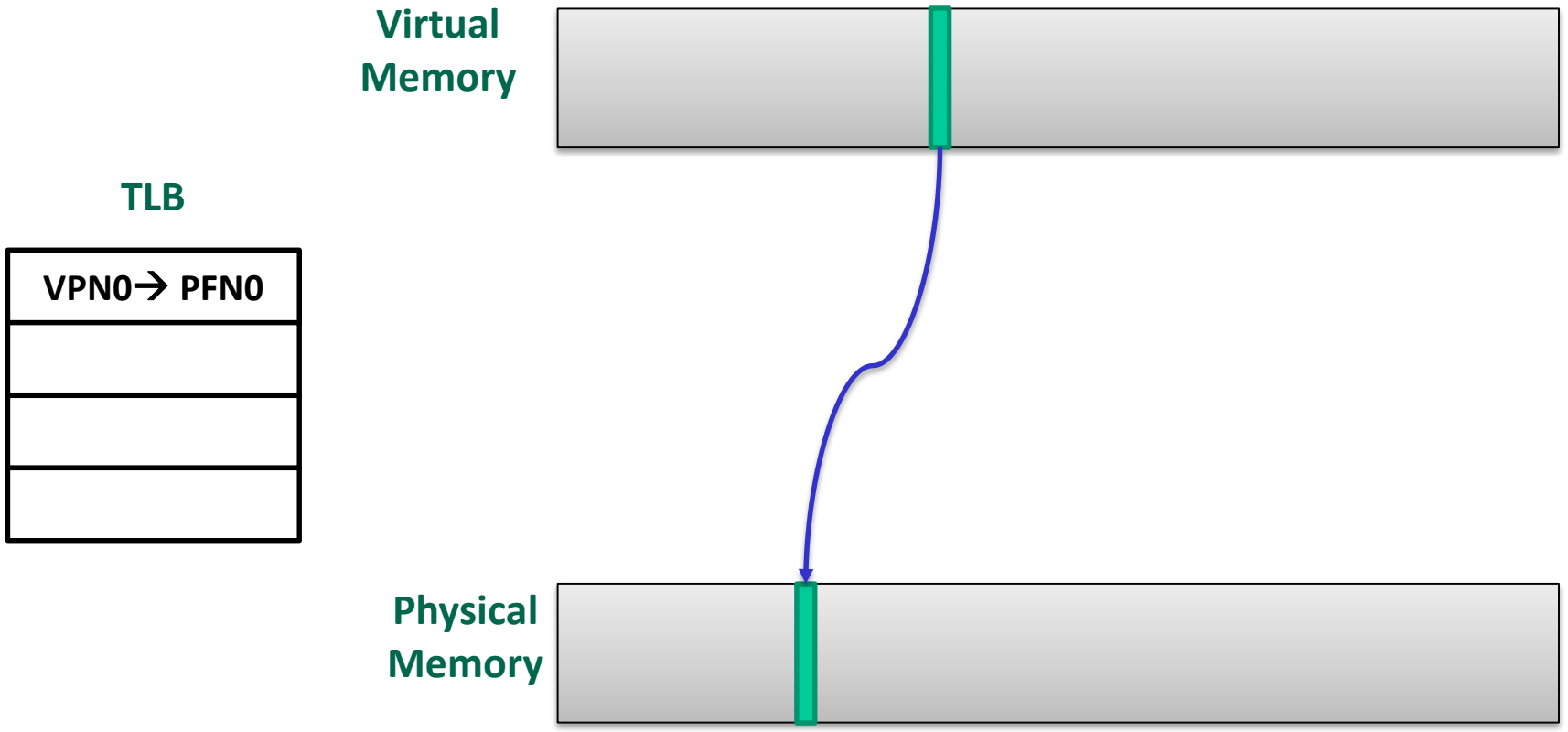
- The OS allocates adjacent virtual pages to adjacent physical pages
  - Random behavior (intermediate contiguity)
  - Explicit support (contiguity beyond page size limit)
- It is then possible to propose hardware that stores groups of adjacent page table entries in a single TLB entry to reduce miss rates and increase performance.
- Why not just using larger page size?
  - Memory management becomes challenging for the OS
  - Memory fragmentation
  - Alignment restrictions

Translation Contiguity

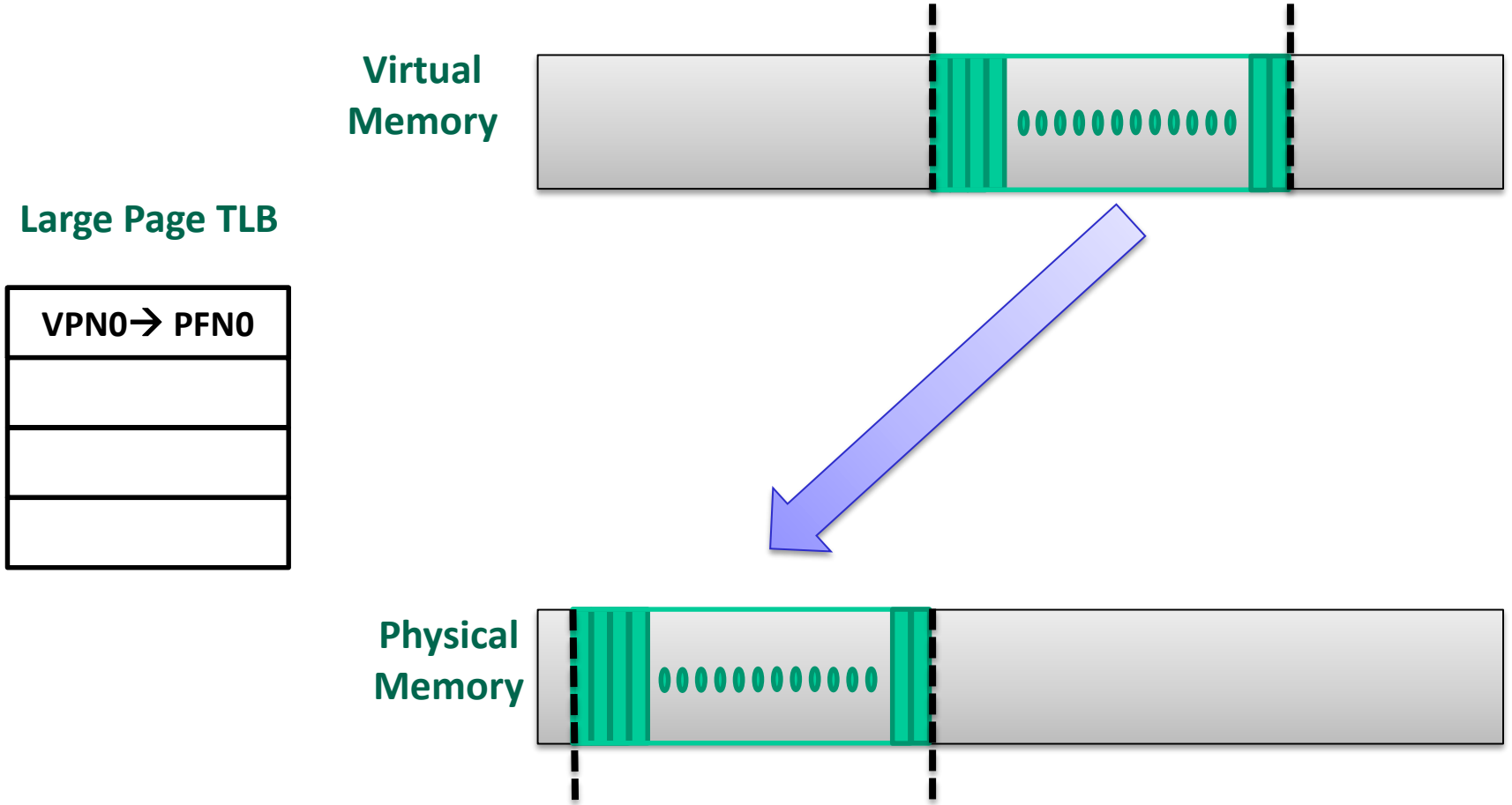
- TLB Coalescing
- Direct Segments
- Range Translations



Page-based Translation

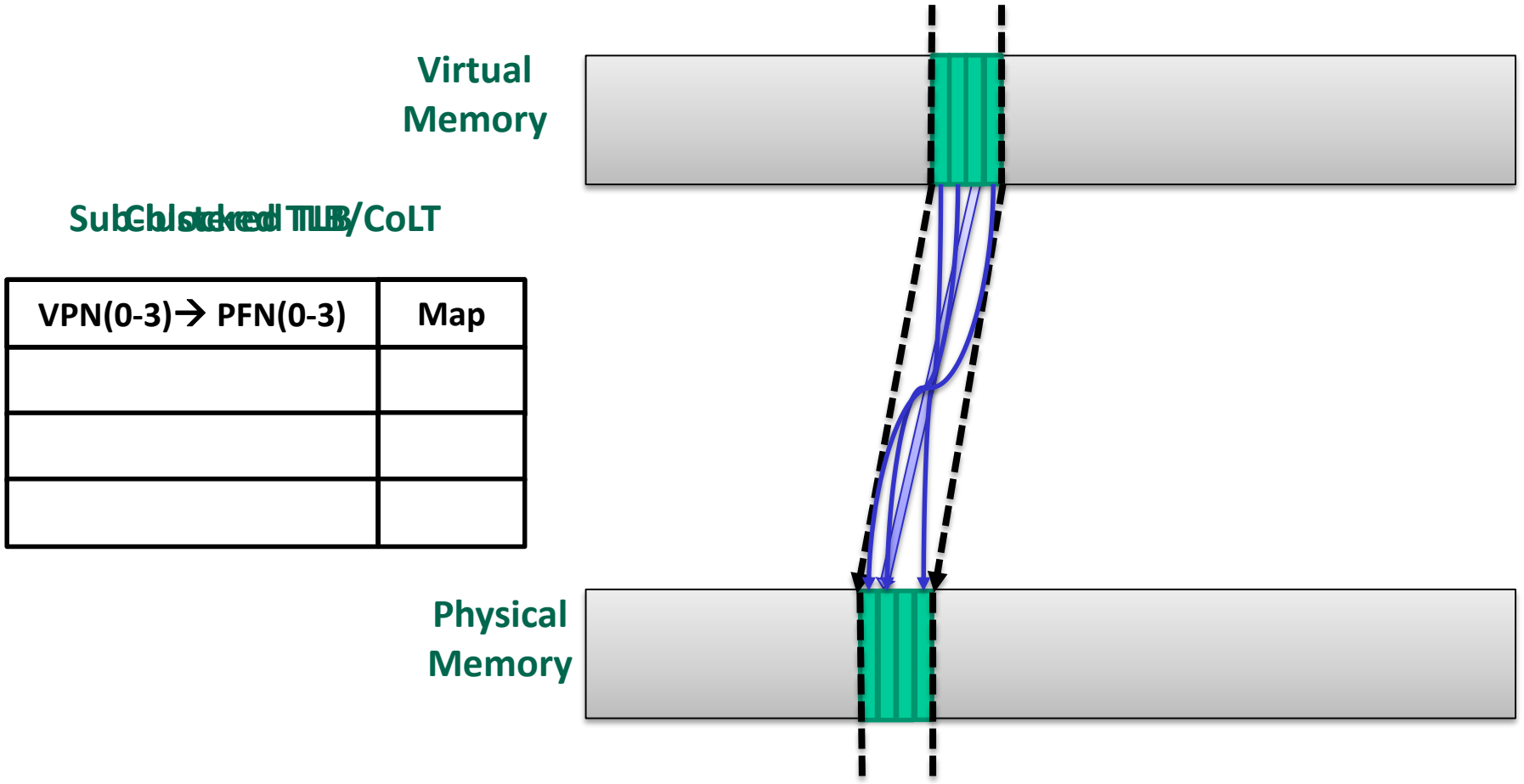


Large Pages



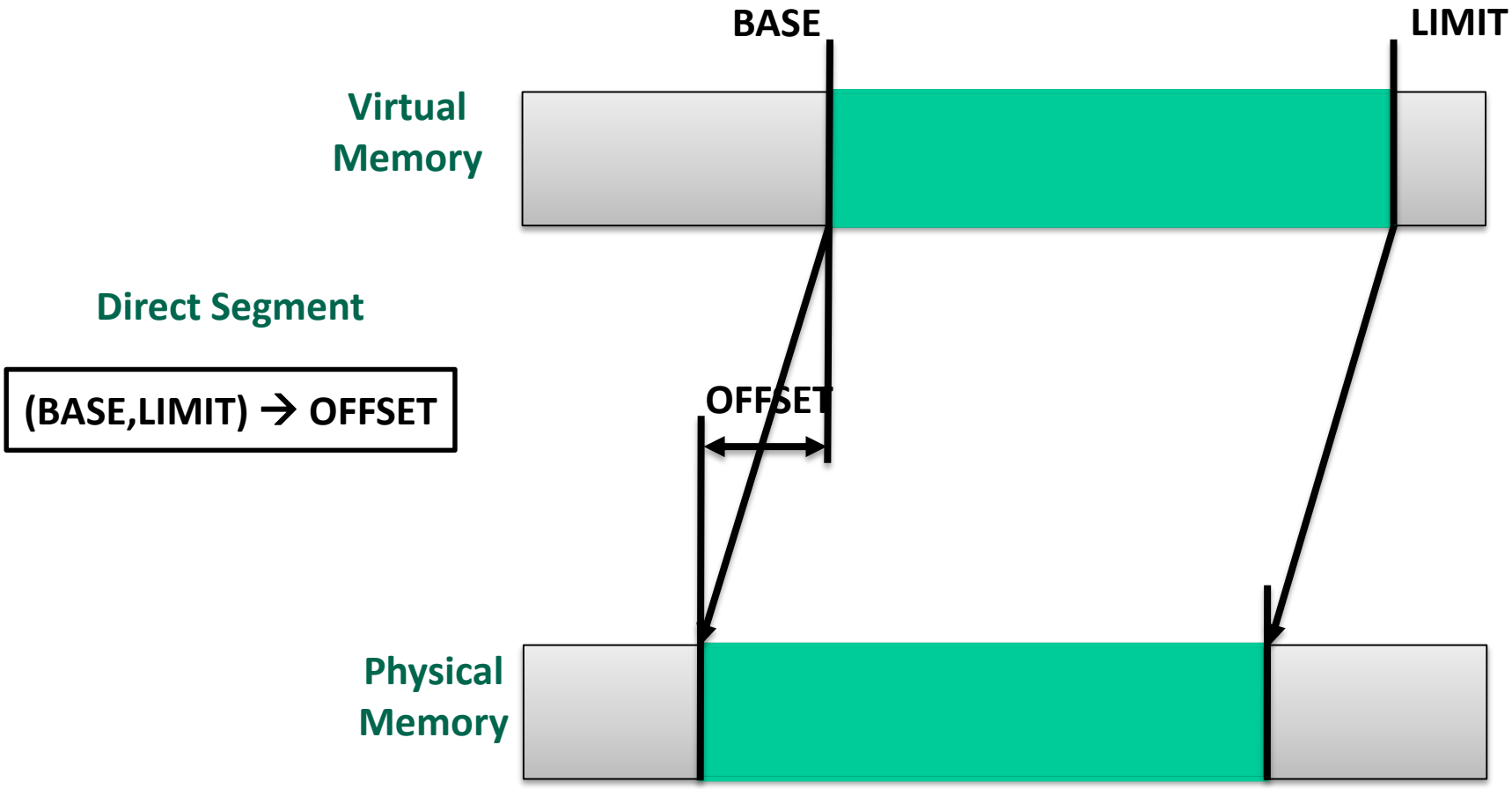
[Transparent Huge Pages and libhugetlbfs]

TLB Coalescing



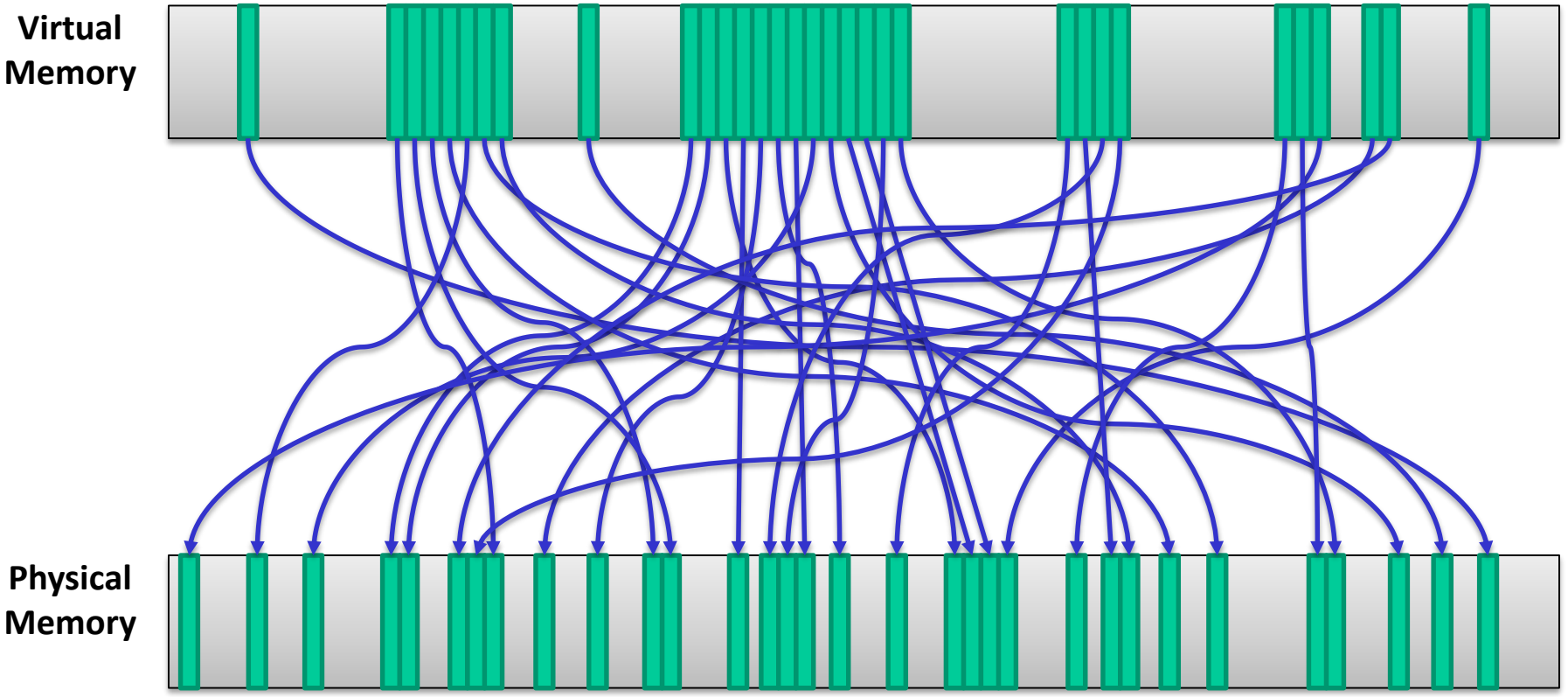
[ASPLOS'94, MICRO'12 and HPCA'14]

Direct Segments

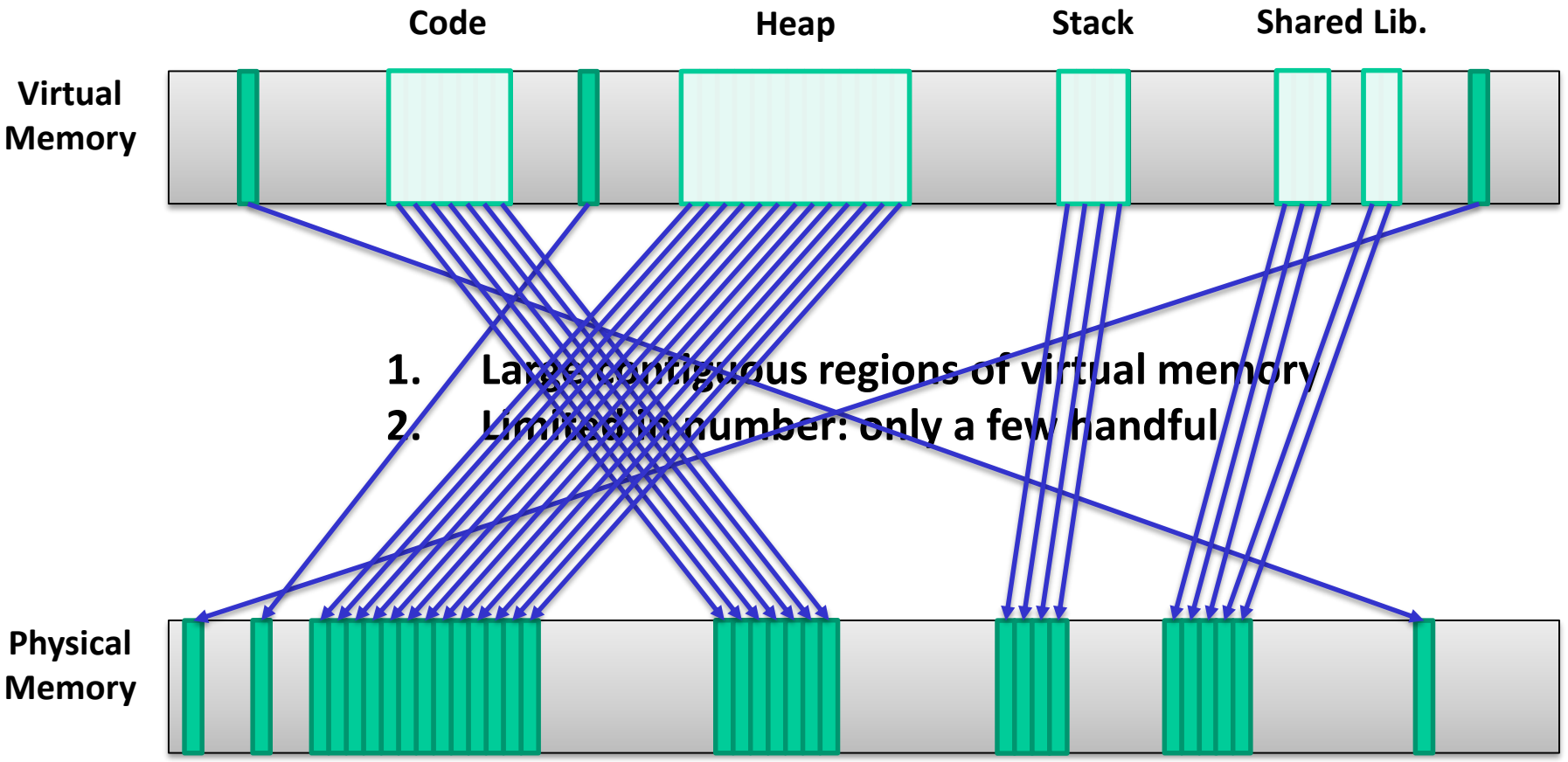


[ISCA'13 and MICRO'14]

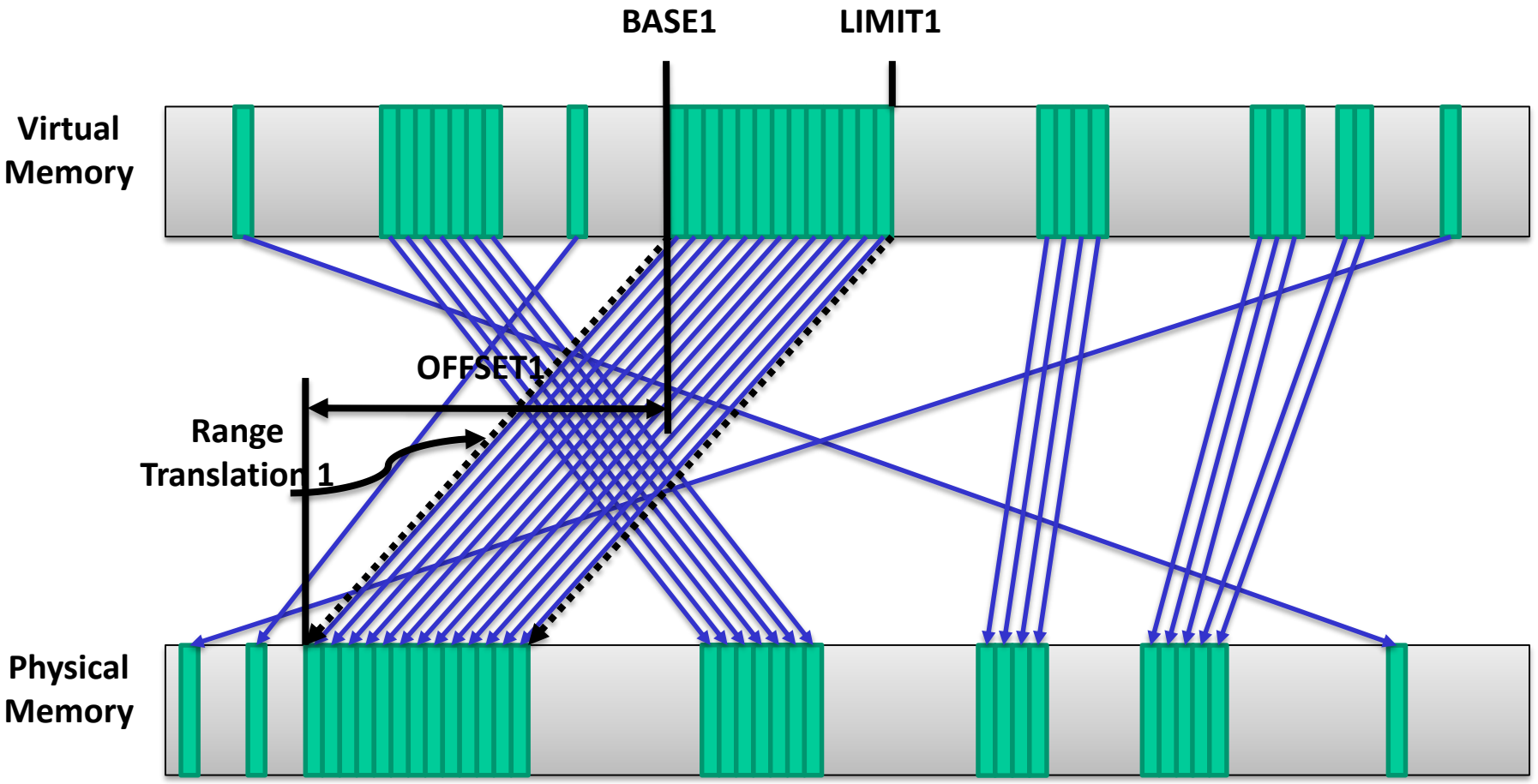
Key Observation



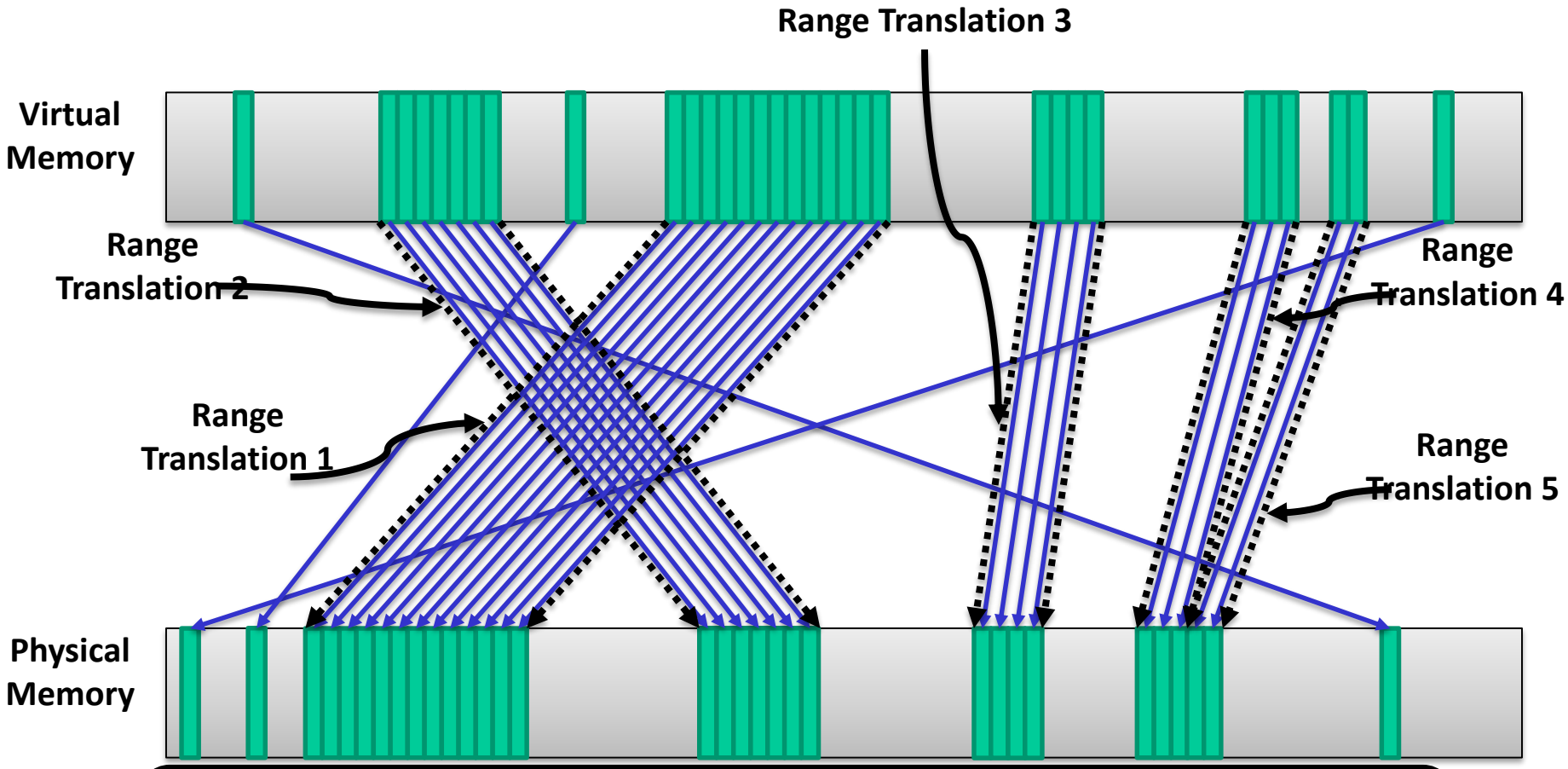
Key Observation



Compact Representation: Range Translation

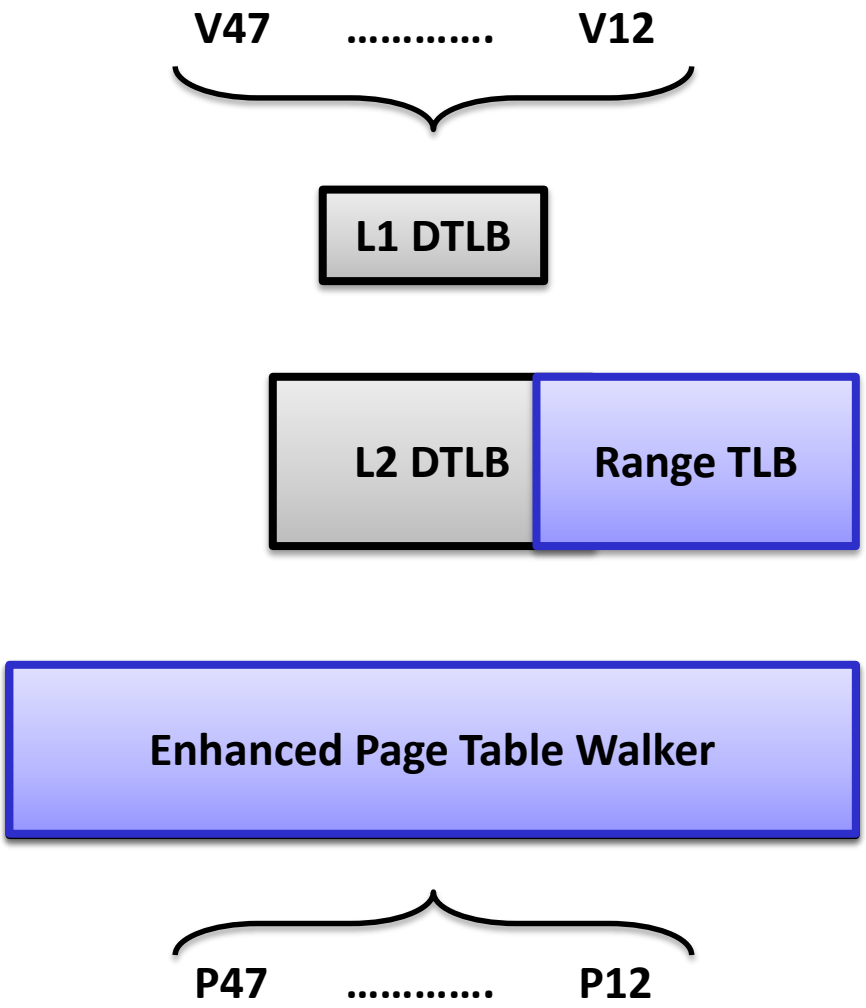


Redundant Memory Mappings [ISCA'15, TopPicks'16]



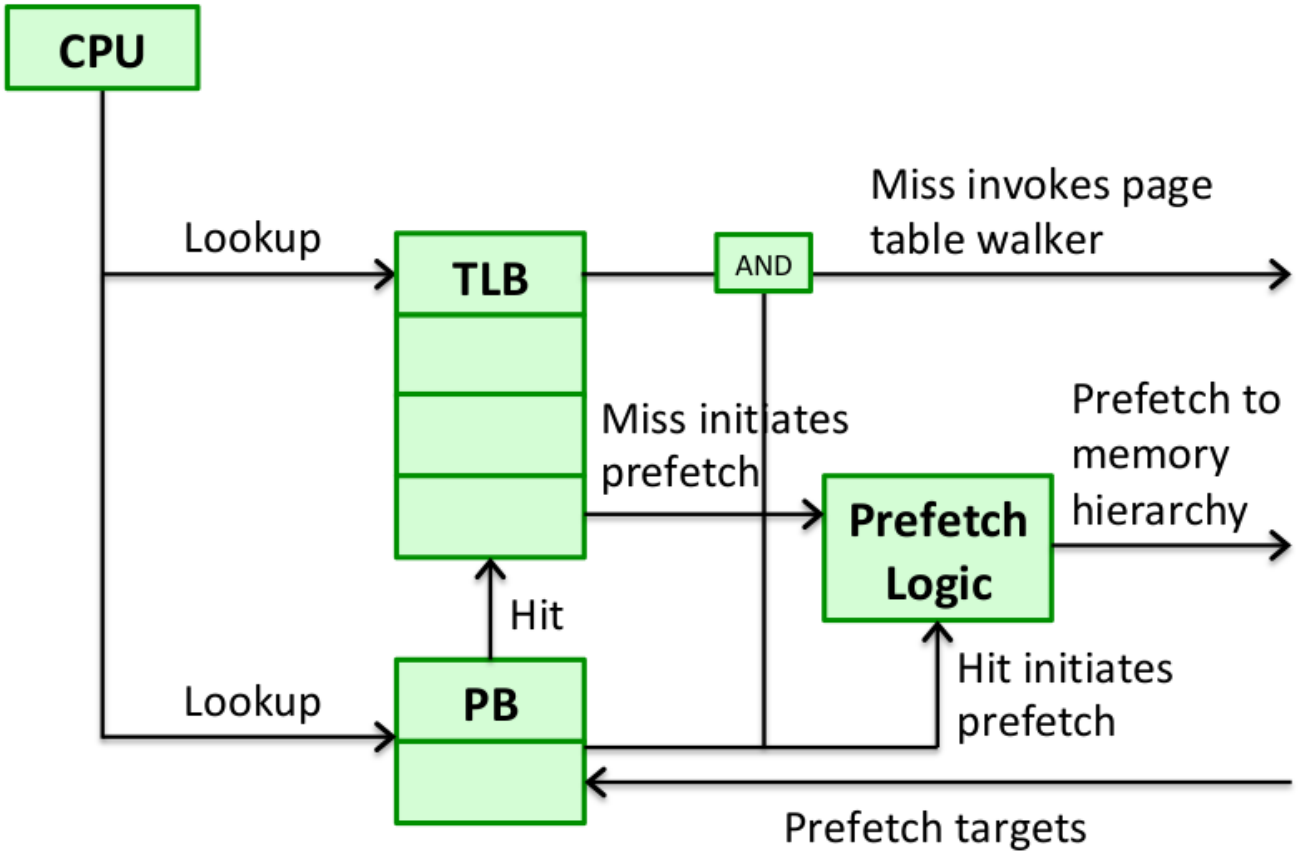
Map most of process's virtual address space redundantly with modest number of range translations in addition to page mappings

Redundant Memory Mappings [ISCA'15, TopPicks'16]



TLB Prefetching

- All prefetched TLB entries are allocated in a prefetch buffer
  - Avoid pollution in the TLB

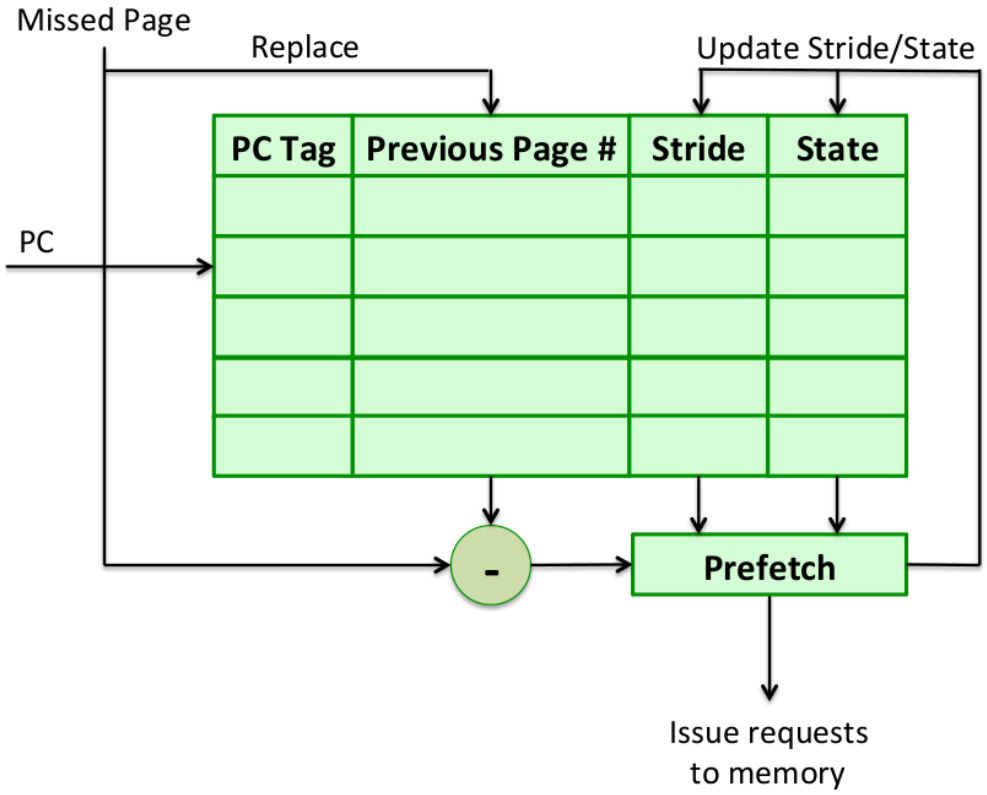


TLB Prefetching

- Cache-line prefetching
  - Cache lines can usually hold multiple page table entries.
  - For example, x86-64 architectures use 8-byte page table entries.
  - A cache lines of 64-128-bytes can hold 8-16 page table entries for consecutive virtual pages.
- Sequential prefetching
  - the prefetch target is the page table entry of the virtual page that is +/-1 away from the virtual page number of the current access, whether it is a TLB hit or miss

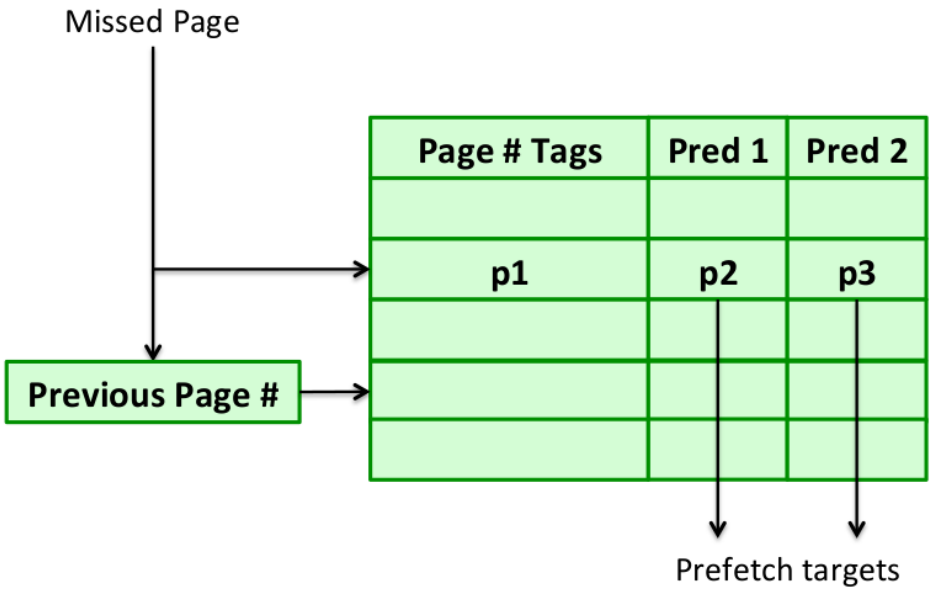
TLB Prefetching

- Arbitrary-stride prefetching
  - Prefetch target is +/-N away from the virtual page number of the current access
  - Dynamically identify the stride N



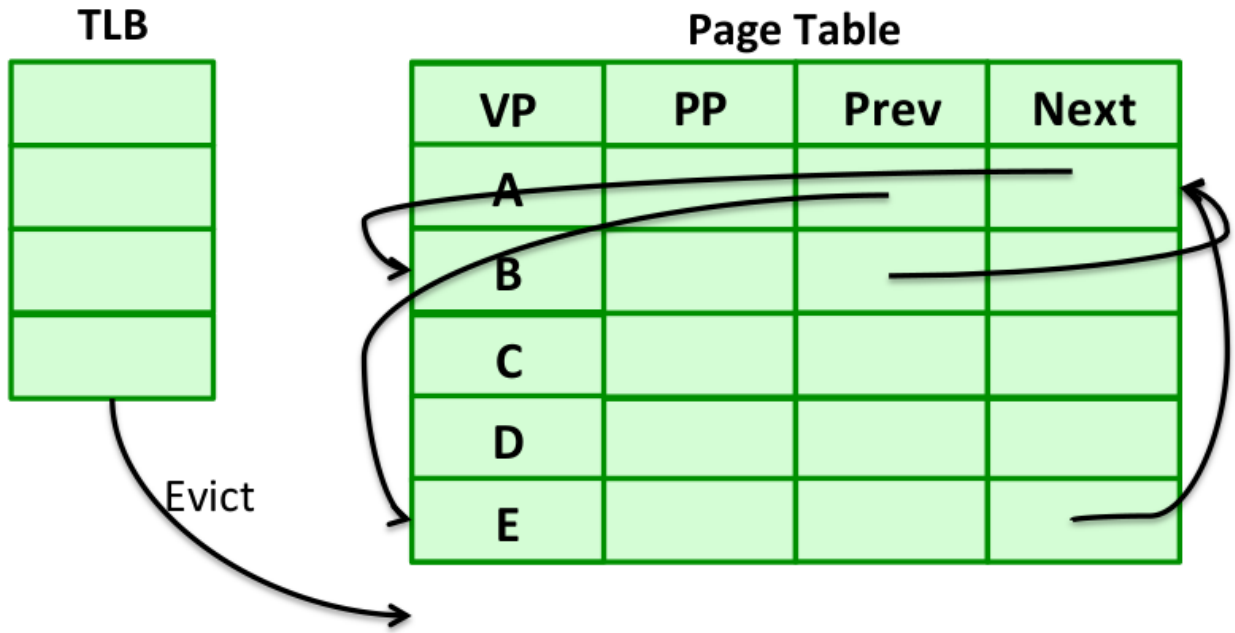
TLB Prefetching

- Markov prefetching
  - More complex temporal patterns exist instead. In such cases, one may expect certain chains of virtual pages to be accessed in order, with no fixed stride between successive virtual pages
  - A hardware table that represents a Markov state transition diagram, with states denoting the referenced virtual page, and transition arcs denoting the probability with which the next page table entry is accessed



TLB Prefetching

- Recency-based prefetching
  - Targeted particularly at TLBs
  - Virtual pages that were referenced close together in time in the past will be referenced close together in time in the future
  - Requires modifications in the page table organization

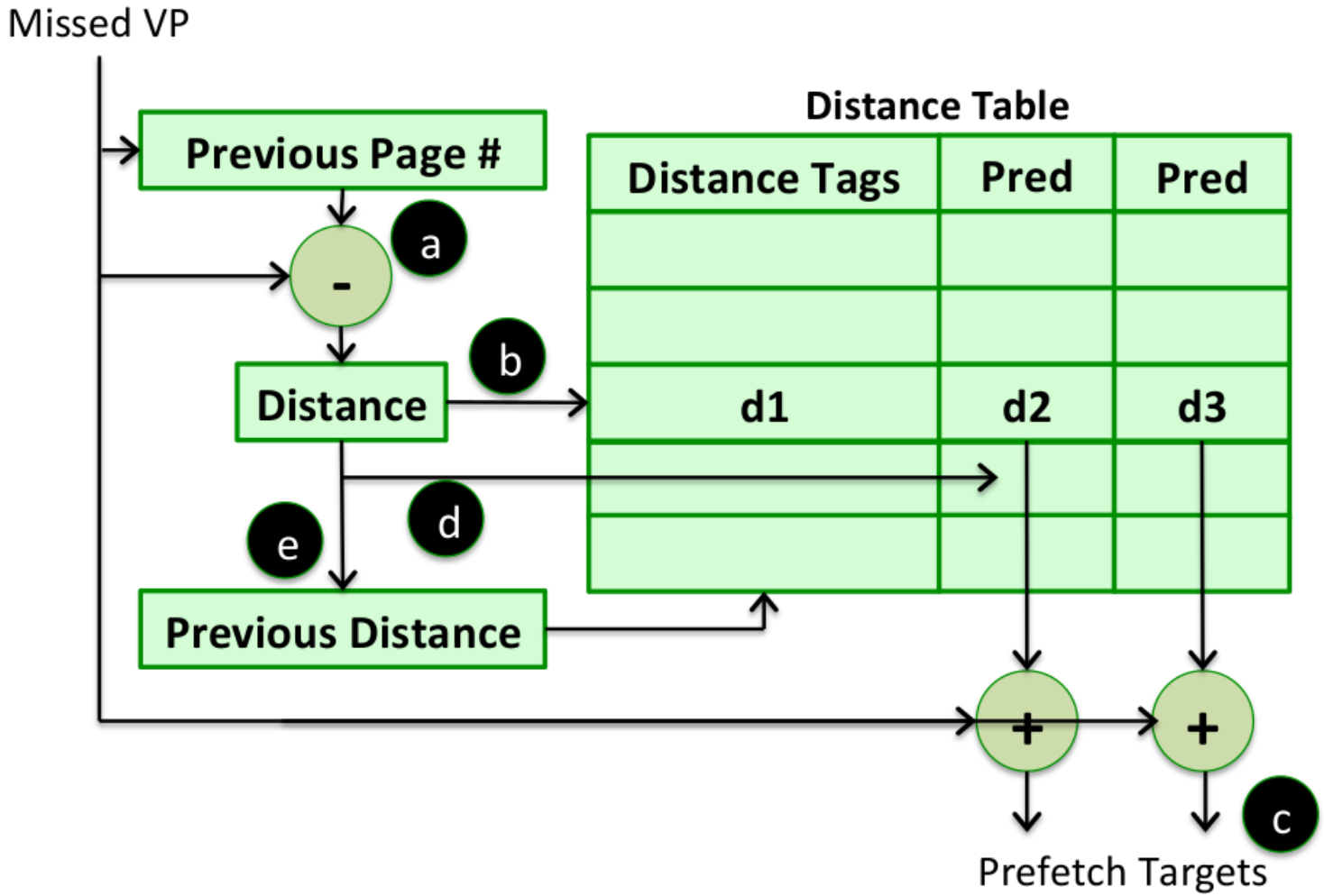


TLB Prefetching

- Distance prefetching
  - Get the best of all these worlds
  - Identify most of the patterns that Markov and recency-based prefetching discover (and maybe other extra patterns) without the storage overheads, while also capturing stride patterns
  - Keeps track of differences (or ”distances”) between successive virtual page numbers
    - For instance, consider a memory access stream to virtual pages 1, 2, 4, 5, 7, and 8.
    - Distance-based prefetching uses hardware that can track the fact that if a distance of 1 is followed by a predicted distance of 2, then we need only a two-entry table for this predictor to capture the entire stream of virtual pages.

TLB Prefetching

- Distance prefetching



Other Translation Techniques & Optimizations

- TLB speculation
- Virtualization
- Energy-efficient address translation
- Translation coherence
- Accelerators
- Memory protection
- Shared address translation structures
- Software optimizations