Appendix L: Advanced Concepts on Address Translation

Abhishek Bhattacharjee
Department of Computer Science, Rutgers University

September 11, 2018

Locality is a principle of nature. Caching works because our brains organize information by localities.
– Peter Denning, pioneer of virtual memory and program locality analysis.

1 Introduction

In previous chapters, we discussed the concepts of virtual memory and address translation. We also discussed how modern computer architectures support these abstractions. This appendix presents a detailed treatment of hardware support for address translation.

Address translation hardware is designed to maximize system performance and minimize power/energy consumption. However, these attributes have to be balanced with the need to support a programmable virtual memory interface for software developers [4, 17, 20, 30, 31]. After all, virtual memory was originally conceived to make programming easy. It achieves this by allowing programmers to reason about how their data structures map to a flat and linear virtual address space, eschewing the complexity of the physical address space which is made up of an assortment of memory and storage devices. This separation of virtual and physical memory addresses has become so vital to the success of computing that we tend not to even think of the existence of the virtual memory abstraction when writing code today. And yet, imagine what would happen in its absence. Programmers would have to carefully reason about the capacities of device memory and storage in order to write correct code. Code written for one system would have to be refactored to port over to another system with different memory/storage configurations. Multiprogramming and security would be compromised because program would be able to overwrite the memory image of another.

For all these reasons, processor vendors and software developers are willing to pay an area/performance/energy ”tax” to enjoy the benefits of virtual memory. This ”tax” is realized in the form of dedicated address translation hardware structures like TLB caches (and other hardware) that we discuss in this appendix. Our goal is to build intuition on how best to realize this hardware in as high-performance but also area-efficient a way as possible. A recurring theme in this appendix is our focus on real-world hardware available on commercial microprocessors today.

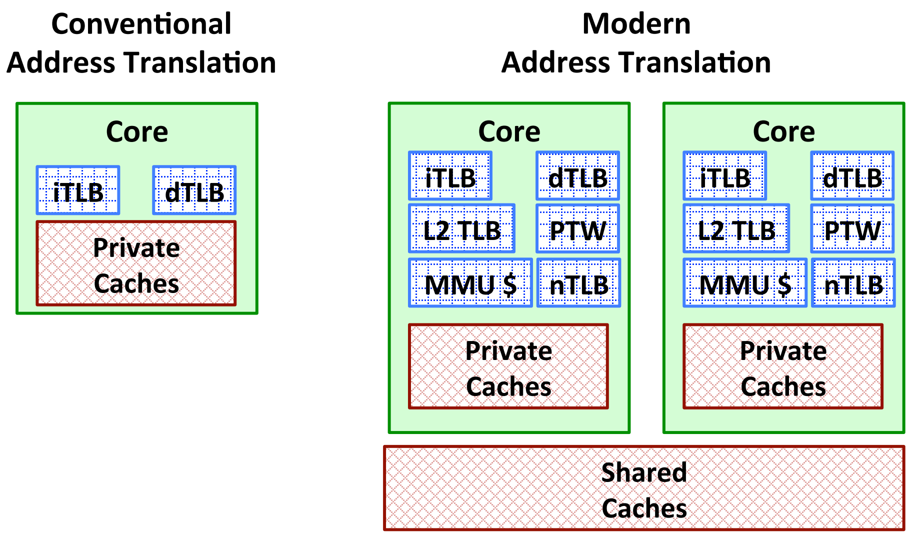


Figure 1: The evolution of address translation hardware from the conventional case (used circa '90s) to what is commercially available today. Address translation structures are shown in patterned blue boxes. Although private and shared caches are not dedicated address translation structures, they do cache entries from the page table.

2 Address Translation on Modern Chips

Figure 1 illustrates the evolution of address translation hardware over several processor generations. We refer to address translation hardware historically seen on uniprocessors in the '90s as "conventional" hardware, and contrast this against the more "modern" multi-core hardware used today.

Figure 1 shows that in the conventional case (similar to today), vendors used to implement separate TLBs for instructions and data; i.e., iTLBs and dTLBs. Additionally, hardware caches, made up of an assortment of L1, L2, etc., caches for data and instructions, were also used to store frequently-used entries from the page table. Unlike TLBs, in which entries store information about a single virtual-to-physical page translation, cache lines store information about multiple virtual-to-physical page translations. This is because cache lines are typically bigger than page table entries. For example, x86-64 systems with 64-byte cache lines typically store translations for 8 adjacent virtual pages since each page table entry is 8 bytes. An important question in address translation design is the following – who handles TLB misses when they occur? The traditional approach would interrupt the operating system (OS) upon a TLB miss, at which point the OS would run a lightweight interrupt handler to "walk" the page table, find the desired page table entry, and install it in the iTLB or dTLB, depending on whether the memory reference was to an instruction or data [42, 44].

Modern address translation hardware has come a long way since these early designs. Figure 1 shows that today, we can expect to see more sophisticated address translation hardware. Key components are:

- **Separate L1 TLBs for instructions and data.** Although not shown, these structures tend to be further split into separate L1 TLBs for different page sizes. As we discuss in subsequent sections, modern OSes support multiple page sizes (e.g., x86-64 systems support 4KB, 2MB, and 1GB pages) but it is difficult to build a set-associative L1 TLB that can concurrently support translations for multiple page sizes [29, 61, 62, 78, 84, 85].

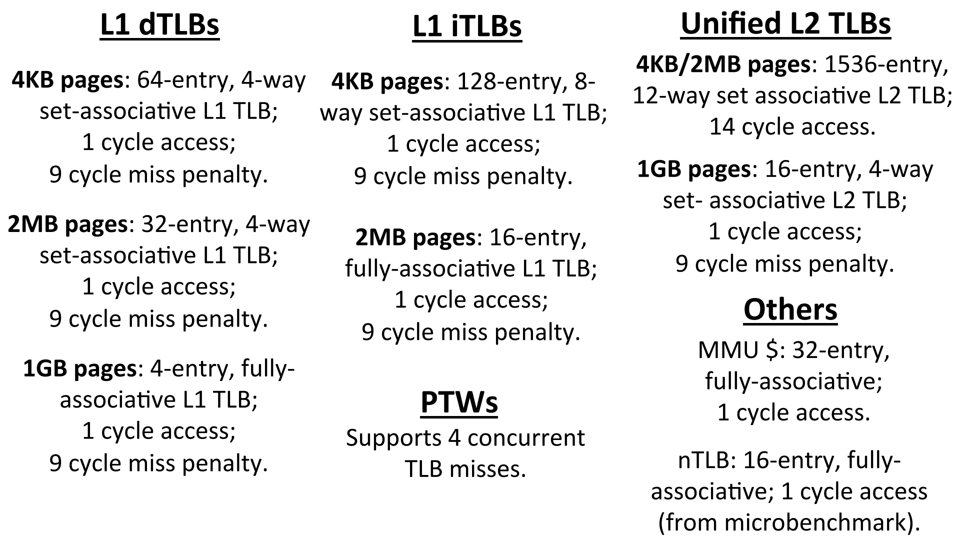
- **Unified L2 TLBs that can cache translations for instructions and data [21, 53].** Modern processors occasionally support a limited set of page sizes concurrently in this L2 TLB (unlike in L1 TLBs, where it is challenging to support multiple page sizes concurrently in a single structure) [29]. For example, modern Intel and AMD x86-64 systems implement L2 TLBs that can simultaneously support 4KB and 2MB pages, but not 1GB pages. Although L2 TLBs are unified for data and instructions, they still remain *private* to individual cores.

- **Hardware page table walkers (PTWs) that are used to handle TLB misses *without* invoking the OS.** In the traditional approach without hardware PTWs, the processor was expected to context switch to the OS on every TLB miss, interrupting user-level execution, flushing the pipeline, and polluting caches [42, 44]. In contrast, modern hardware PTWs obviate the need for all these expensive mechanisms. In fact, hardware PTWs also enable two more optimizations to hide TLB miss latencies: (1) they can overlap page table walks with independent instructions executing on the processor (as per out-of-order execution strategies like scoreboarding or Tomasulo's algorithm); and (2) they can be designed to service multiple TLB misses concurrently [11, 64, 66–68].

- **Memory management unit (MMU) caches that are used to accelerate TLB misses [10, 16].** MMU caches are a relatively new addition to the address translation hardware stack, and are targeted at x86- and ARM-style page tables. On these systems, page tables are implemented as *forward-mapped multi-level radix trees* rather than as linear page tables. MMU caches store page translations from non-leaf entries of these page tables are used to accelerate page table walks.

- **Nested TLBs are specialized TLB structures aimed at supporting cloud environments running virtual machines [15, 93].** Since these environments suffer from particularly high address translation overhead, processor vendors build dedicated TLBs to support their complex page table structures. Virtualization is a topic that requires a detailed treatment in its own right and is hence out of the scope of this appendix – we point interested readers to relevant papers on this topic [15, 20, 93].

- **Hardware caches caches page table entries.** Some architectures like SPARC use special software structures called *Translation Storage Buffers (TSBs)* to aid page table entry caching in the processor caches. We describe TSBs in subsequent sections [21–23].



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).

Figure 2: Parameters of the addess translation hardware available on Intel's Skylake chips as of 2018. While the parameters for most of these hardware structures are public, the nested TLB sizes are inferred based on experiments we ran using microbenchmarks designed to identify each hardware structure's capacity and associativity.

While the particular configuration parameters of these parts of the address translation stack can vary across architectures, we provide some insight on their relative sizes and latencies for Intel's Skylake chip. Figure 2 quantifies the capacities, organizations, and access latencies of the various system TLBs, hardware page table walkers, MMU caches, and nested TLBs. The remainder of this appendix is dedicated to performing a deep dive on each of these hardware structures.

3 L1 TLBs

Modern processors implement a group of split L1 TLBs for instructions and data, separated by page size.

3.1 Separate I and D TLBs

There are several reasons that processor vendors opt to employ separate L1 TLBs for instructions and data. First, 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 (and other similar constraints). Second, instructions and data exhibit different locality of reference or reuse attributes. For example, it is well known that programs generally have much smaller portions of their memory footprint dedicated to instructions than to data. At the same time, instruction references are more "critical" to performance; i.e., while data references can generally be overlapped by independent streams of instructions because of out-of-order capabilities, instruction references are often on the critical path of pipeline execution. Therefore, instruction TLB misses can have a particularly pernicious impact on performance. For these reasons, it is difficult – although not impossible – to design a single TLB structure with the appropriate replacement/allocation policy to manage the needs of instruction and data references. Consequently, processors vendors opt for separate iTLBs and dTLBs [20].

One implication of split L1 TLBs is that vendors can use different policies among the TLB resources when supporting simultaneous multithreading (or hyperthreading) in hardware. Because of the criticality of instruction references to overall performance, processor vendors typically implement statically partitioned iTLBs for different simultaneous hardware threads [40, 41]. Consider, for example, Intel's Skylake architecture, where the 128-entry L1 iTLBs are statically partitioned across the two threads when two-way hyperthreading is enabled. In this case, each thread is guaranteed 64 instruction TLB entries, preventing fairness problems that would arise in a dynamically partitioned scheme if one thread had a bigger instruction footprint than the other. In contrast, dTLBs are typically partitioned dynamically among simultaneous hardware threads or hyperthreads [40, 41]. The rationale is that different threads may have different memory footprints dedicated to their heaps/stacks, and adapting a common pool of TLB resources dynamically among the competing threads may make better overall use of the structure. For this reason, Intel's Skylake architecture dynamically partitions the L1 dTLB among hyperthreads.

3.2 Separate L1 TLBs for Different Page Sizes

L1 TLBs must be fast and energy-efficient. Performance is desirable as TLBs reside on the critical L1 datapath of pipelines. This means that they are usually built in a simple manner that meets timing constraints, with lookup and miss handling characteristics that are amenable to speed. Additionally, energy-efficiency is desirable as TLBs can consume a considerable – as much as 15% –of processor energy [13, 18, 29].

Consequently, L1 TLBs are usually realized with set-associative RAM or CAM structures rather than fully-associative ones. However, set-associativity poses correctness challenges with respect to large pages or superpages. Large pages are commonly employed by most OSes today. The motivation for large pages is that they enable greater effective TLB capacity. For example, a single TLB entry on x86-64 systems can support a 4KB page but with large page support, the same entry can be used to support a 2MB or 1GB large page instead. Large pages can dramatically reduce the frequency of TLB misses [29, 68, 84, 85].

However, supporting multiple page sizes also complicates the design of set-associative TLBs. The reason is that different page sizes require a different number of page offset bits. For example, on x86-64 systems, 4KB baseline pages require 12 page offset bits, 2MB large pages require 21 page offset bits, while 1GB large pages require 30 page offset bits. To reduce conflict misses in set-associative TLBs, we wish to use the least significant bits of the virtual page number. This means that for a TLB with 8 sets, for example, we would want to use bits 14-12 for 4KB pages and bits 23-21 for 2MB pages as the TLB index. To do this, we need to know the page offset bits (i.e., the

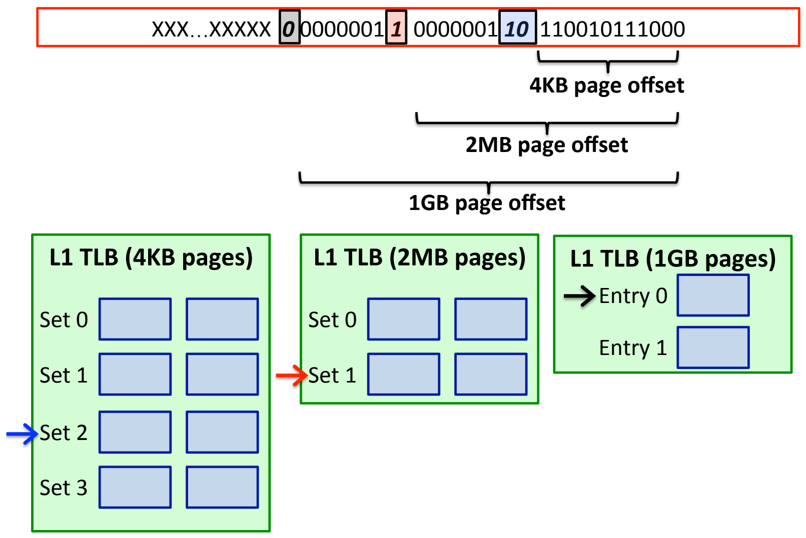


Figure 3: Parallel lookups of set-associative split TLBs for all page sizes. The bits in blue (bits 12 and 13) are used to select among the four sets in the TLB for 4KB pages; the bit in red (bit 21) is used to select between the two sets in the TLB for 2MB pages; the bit in black (bit 30) is used to select between the two entries in the direct-mapped TLB for 1GB pages.

page size) of the desired translation at *lookup time*. This presents a chicken-and-egg problem since the page size is usually known *after* address translation [29, 61, 78].

To side-step this issue, processor vendors implement separate L1 TLBs for different page sizes (in contrast, L2 TLBs are typically architected with hardware support to accomodate multiple page sizes, as we discuss in subsequent sections). As detailed in Figure 2, Intel Skylake chips maintain separate L1 TLBs for 4KB, 2MB, and 1GB pages. Translations are inserted into the ”right” TLB on misses. In other words, TLB misses trigger page table walks. After the translation is found, hardware in the MMU identifies the page size of the desired translation. Subsequently, the translation is entered in the appropriate L1 TLB. For example, if the translation is to a 4KB page, it is filled into the 4KB page L1 TLB. 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. Figure 3 shows how this proceeds for an x86-64 system with 4KB, 2MB, and 1GB pages. The hardware is designed to extract bits from the virtual address (shown in the red box at the top) that are more significant than the various page offset bits. In other words, bits 13 and 12 (which are more significant than the 12 bits used to determine addresses within 4KB ranges) are used as an index into the 4KB page TLB. In tandem, bit 21 (which is more significant than the 21 bits used to determine addresses within 2MB ranges) is used to select between the two sets in the 2MB page TLB. Similar logic is used to look up the 1GB page TLB in parallel.

The lookup operation in Figure 3 has two possible outcomes. One is that the desired translation is absent from all L1 TLBs. This prompts a lookup into the L2 TLB hierarchy, which we further describe in Section 4. The other possibility is that the desired translation is cached in the L1 TLBs. If this is the case, the translation will be cached in only *one* of these split TLBs. Hence, a lookup can hit in only one L1 TLB.

Q&A 1

Q1. Suppose that a CPU is executing a program where the bulk of the data accesses (or the ”working set” of the program) are to a memory region that is 10GB in size. Moreover, suppose that the CPU runs the Linux OS, which can generate multiple page sizes. Let us perform some back of the envelope calculations for how the TLBs should be sized.

(Q1a) If the OS can generate only 4KB pages for the working set of our program, how should we size our split L1 TLBs?

(Q1b) Suppose the OS can generate a mix of 4KB and 2MB pages for the working set of our program. Furthermore, suppose that 80% of the working set is with 2MB pages, while the remainder is with 4KB pages. How should we size split L1 TLBs?

(Q1c) Suppose the OS can only generate 1GB pages. How should we size split L1 TLBs?

A1. The general approach to sizing TLBs is to consider the size of a *typical* program's memory footprint, or its working set size. In this exercise, we sidestep the question of a typical program, and focus on the one program whose details are provided.

(A1a) If the OS generates only 4KB pages, the working set can be covered by 10GB $\div$ 4KB or 2621440 pages. This would require (an infeasibly large) 2621440-entry TLB.

(A1b) Based on the distribution of pages, 80% of 10GB or 8GB is covered using 2MB pages. This means that we need 8GB $\div$ 2MB or 1024 entries in our 2MB page L1 TLB. We also need 2GB $\div$ 4KB or 524288 entries in our 4KB page L1 TLB.

(A1c) Finally, if the OS generates 1GB pages, we can just use a 10-entry 1GB page L1 TLB to cover the working set of 10GB.

Note that split TLBs do not have to be sized equivalently. In fact, processor vendors usually devote the least area to 1GB page TLBs, more area to 2MB page TLBs, and the most area to 4KB page TLBs. The reason is that TLBs for the smallest page size require commensurately more entries to cover the same size of memory as TLBs for a larger page size. Processor vendors typically size TLBs for the worst case – i.e., 4KB page TLBs are sized assuming that the OS will be unable to generate 2MB or 1GB pages for typical workloads the system is expected to run. This is why, as shown in Figure 2, TLB sizes (especially L2 TLBs, which we subsequently discuss) have now become fairly large in terms of the number of entries.

Q&A 2

Q2. Suppose we use 64-entry, 4-way set-associative L1 TLBs for 4KB pages, 2MB pages, and 1GB pages. Would all the TLBs require the same area to implement? Assume an x86-64 architecture for the processor on which these TLBs are implemented.

A2. Entries in TLBs for larger pages require fewer bits to implement. This is because larger pages have longer page offsets than baseline pages. Therefore each TLB entry's tag and data fields can have a reduced bit-width. Specifically, our 64-entry, 4-way TLBs implement 16 sets and therefore require 4 index bits. Consider TLBs for 4KB pages – virtual page numbers require 52 bits of tag space and 52 bits of data in the TLB entries. Since the index requires 4 bits, this means that each entry in the 4KB page TLB has 48-bit tags and 52-bit data fields. In contrast, 2MB pages have 21-bit page offsets, meaning that virtual and physical page numbers are 43 bits each. Therefore, each 2MB page TLB requires 39-bit tags (43 bits - 4 bits for the index) and 43-bit data fields. In other words, each 2MB page TLB entry requires 18 fewer tag/data bits than each 4KB page TLB entry. Similarly, each 1GB page TLB entry requires 18 fewer tag/data bits than each 2MB page TLB entry.

---

4 L2 TLBs

Section 3 discussed the L1 TLB layer in modern microprocessors. We now shift our attention to the unified L2 TLB. L2 TLBs usually maintain both instructions and data, but remain private to each core (although recent studies have begun exploring the benefits of L2 TLBs shared among cores [21, 23, 83]). In assessing how to build unified L2 TLBs, there are several important attributes to consider:

Access time: This is an important attribute and is equal to the sum of the traversal time of the interconnect between the L1 and L2 TLBs and the access time of the RAM/CAM array used to physically realize the L2 TLB [14, 21]. In general, the bigger the L2 TLB, the longer the interconnect traversal and array access times. For example, Intel's Skylake systems have 8-10 cycle latencies in looking up the L2 TLB. Like conventional caches, TLBs can be banked to partly reduce RAM/CAM access latency.

Hit rate: Hit rates have to be as high possible to counterbalance the higher access times of unified L2 TLBs. This is where L2 TLBs shine. Because they are significantly bigger than L1 TLBs, they typically enjoy high hit rates [14, 21]. Although little is publicly known about TLB replacement policies, academic studies using microbenchmarks that glean properties of commercial TLB hierarchies suggest that L2 TLBs use intelligent replacement policies akin to cache replacement strategies (e.g., approxmiations of LRU, etc.) [64, 66]. This is in contrast to L1 TLBs, where simpler replacement policies like random eviction are often used for ease of implementation [66, 76].

Multiple page size support: An important aspect of unified L2 TLBs is the question of supporting multiple page sizes. For reasons outlined in previous sections, supporting multiple page sizes in a single set-associative structure can be challenging. Modern L2 TLBs tend to support a limited set of multiple page sizes – e.g., x86-64 architectures have supported 4KB and 2MB pages concurrently in L2 TLBs since the advent of Intel Skylake's Haswell/Broadwell architectures and AMD's Zen architecture. Additionally, these architectures maintain a (smaller) separate 1GB page TLB. In Section 5, we discuss how L2 TLBs support multiple page sizes concurrently.

Inclusive, mostly-inclusive, or exclusive designs: When designing multi-level TLBs, an important question is whether they should be architected such that L2 TLBs are inclusive of the L1 TLBs. Implementation choices and tradeoffs are similar to caches in this regard. In general, there exist three options.

The first option – and generally the most commonly-used – is to implement the TLBs as *mostly inclusive* [21, 93]. In this scenario, while it is likely that L1 TLB entries are also present in the L2 TLB, there is no guarantee. Operationally, when the L1 and L2 TLBs miss and a page table walk occurs, the desired translation entry is filled into both L1 and L2 TLBs. When the miss is to an instruction translation, it is filled into the L1 iTLB and the unified L2 TLB. When the miss is to a data translation, it is filled into the L1 dTLB and the unified L2 TLB. However, after these TLBs are filled, translation eviction from each TLB occurs independently because each structure maintains it's own replacement policy. That is, it is possible for a translation to be evicted from the L2 TLB while remaining resident in the L1 TLB. This approach is similar to mostly-inclusive designs for caches [21, 43].

The second option is to use *strict inclusive*. Strictly inclusive architectures are similar to mostly inclusive ones, with one difference – L1-L2 inclusion is guaranteed. In other words, if a translation is absent from the L2 TLB, it is guaranteed to also be absent from the L1 TLB. This means that if a translation is evicted from the L2 TLB, a back-invalidation message is relayed to the L1 TLB. This message checks to see if the translation evicted from the L2 TLB is present in the L1 TLB. If it is, it is invalidated from the L1 TLB to maintain the inclusion property. TLB back-invalidation operations is similar to back-invalidations that are used for strictly inclusive caches [43].

The third option is to use an *exclusive* policy. In this scenario, L1 and L2 TLBs are designed to avoid any type of redundancy. When a desired translation is absent from the L1 and L2 TLBs, triggering a page table walk, the entry is filled only in the L1 TLB (and not the L2 TLB). Only when the translation is evicted from the L1 TLB does it get filled into the L2 TLB, mirroring the design of exclusive caches [43, 45].

Choosing which of these approaches to use depends on various factors. If the L2 TLB is area- or power-constrained, an exclusive architecture may be desirable. This is because maintaining multiple copies of the same translation in the L1 and L2 TLBs may adversely affect performance when TLB entries are scarce. Again, this is similar in spirit to situations where exclusive cache hierarchies are useful. If, on the other hard, interconnect bandwidth between the L1 and L2 TLBs is limited, a mostly inclusive approach without the overheads of back-invalidation messaging may be appropriate. However, if coherence overheads are a concern, a strictly inclusive organization may be more appropriate, for reasons detailed below.

Implications of inclusive/exclusive hierarchies on translation coherence: Although a detailed treatment of translation coherence is out of the scope of this appendix, we briefly introduce the problem in this section. Like private caches, private TLBs must also be kept coherent. On a multi-core or multi-processor system, a single core running the OS may change a translation entry; e.g., the physical page assigned to a virtual page may change, page permissions (read/write/execute) may change, etc. These changes must be reflected in all private TLBs [5, 51, 58, 59, 65, 74, 89, 93]. For example, suppose that on a four-core system, core 0 modifies a translation. Initially, the translation shows that virtual page 2 points to physical frame 10. Now suppose that the OS modifies the translation so that virtual page 2 points to physical frame 12. Not only must this change be reflected in core 0's TLB, but also in cores 1-3's private L1 and L2 TLBs too. This means that invalidation messages for the TLB entries corresponding to virtual page 2 must be relayed system-wide. While TLB coherence messages are likely far fewer than cache coherence messages, they can occur frequently enough that one may wish to design TLB hierarchies that minimize them. In this scenario, a strictly inclusive setup can reduce coherence messages as the L2 TLB can act as a coherence filter for the L1 TLB (as with L1 and L2 caches [82]). In other words, coherence transactions can first look up the L2 TLB. If the translation is found in the L2 TLB, it may also exist in the L1 TLB, so a coherence message must also be relayed to the L1 TLB. However, if the translation is absent in the L2 TLB, the property of strict inclusion tells us that the translation *must* also be absent from the L1 TLB. Therefore, no coherence message needs to be relayed to the L1 TLB in this case, saving interconnect bandwidth versus a mostly inclusive design, where a coherence message would still need to be relayed to the L1 TLB.

Q&A 3

Q3. Consider a strictly-inclusive TLB hierarchy where a back-invalidation message is sent from the L2 TLB to L1 TLB, whenever a translation is evicted from the former. The L1 TLB is a 4-entry, 2-way set-associative structure, while the L2 TLB is an 8-entry, 2-way set-associative structure. Consider translations *A*-*H*, where *A* corresponds to virtual page 0, *B* to virtual page 1, *C* to virtual page 2, and so on. Furthermore, suppose that the L1 TLB has the following contents – translations *A* and *C* are resident in set 0, while *B* and *D* are resident in set 1. Now suppose that the L2 TLB has the following contents – translations *A* and *E* are in set 0, *B* and *F* are in set 1, *C* and *G* are in set 2, and *D* and *H* are in set 3. Suppose that the CPU makes a memory request for translation *J* (corresponding to virtual page 9). How do the L1 and L2 TLB contents change?

A3. TLBs are looked up and filled using conventional modulo set-indexing, which is why *A*, which corresponds to virtual page 0, maps to set 0 in both TLBs; why *B*, which corresponds to virtual page 1, maps to set 1 in both TLBs; why *C*, which corresponds to virtual page 2, maps to set 0 in the L1 TLB (which has only two sets) but set 2 in the L2 TLB (which has four sets). If J is requested, it must be filled into the L1 and L2 TLBs, and some contents must be evicted. First, consider how J is placed in the L2 TLB. Since its virtual page number is 9, it maps to set 1. This means that either *B* or *F* must be evicted from the L2 TLB. Suppose that *B* is evicted from the L2 TLB. Because of strict inclusion, *B* must also be evicted from the L1 TLB too, so a back-invalidation message is relayed. Next, *J* must be filled in to the L1 TLB, where it also maps to set 1. Since the entry that previously held *B* is now empty, it becomes the natural location in the L1 TLB to fill *J* in.

Now consider the situation where instead of *B*, *F* had been evicted from the L2 TLB to make room for *J*. If this had been the case, a back-invalidation message would have been sent to set 1 in the L1 TLB. In this case, because *F* is not resident in the L1

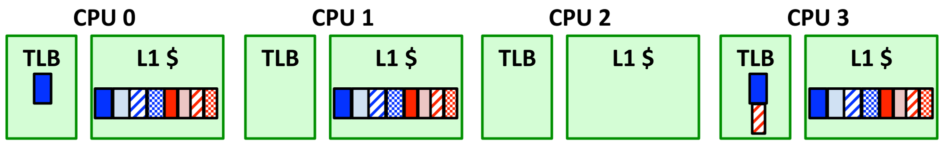


Figure 4: Translation coherence requires maintaining coherence among page table entries within the private caches, as well as the TLBs (and other address translation structures).

TLB, the back-invalidation would not have an impact on the L1 TLB. In our example, *B* or *D* need to be evicted from the L1 TLB, and can be done silently without informing the L2 TLB, which must have these translations due to strict inclusion.

Q&A 4

Q4. Consider Figure 4, which shows the private L1 caches and L1 TLBs of a four-core system. Page table entries are shown in the solid/striped blue and red boxes. A cache line of eight page table entries is filled into CPU 0 and CPU 3, because they accessed at least one page table entry within the cache line (via their page table walker or the OS) . CPU 0 has accessed the blue page table entry in the past, while CPU 3 has also accessed the blue and striped red page table entries. Consequently, these page table entries are in their TLBs. Meanwhile, CPU 1 has likely only accessed a page table entry within the cache by executing the OS rather than the page table walker; therefore, the cache line is present in its L1 cache but no page table entry from it is resident in the TLB. Finally, CPU 2 has not accessed any page table entry within this cache line. In this exercise, we will analyze translation coherence and how the addition of per-core L2 TLBs with strict inclusion can help reduce probes (due to coherence activity) of the L1 TLBs.

(Q4a.) Suppose CPU 1 runs the OS, which chooses to update the physical frame number assigned to the virtual page in the solid blue page table entry. What steps are necessary in order to maintain coherence?

(Q4b.) What if CPU 0 performs the action described in (Q4a.), rather than CPU 1?

(Q4c.) How would the actions in (Q4a.) and (Q4b.) change if all CPUs maintained private L2 TLBs that were organized as being strictly inclusive of the L1 TLBs?

(Q4d.) What happens if CPU 1 executes a store to the red striped page table entry?

(A4a.) Translation coherence is made up of two separate sets of coherence activities – coherence of the page table entries within the TLBs, and coherence of the page table entries within the L1 caches. Consequently, if CPU 1 updates the solid blue page table entry, any copies of the page table entry in the TLBs and caches of CPU 0, 2, and 3 must also be updated.

TLB coherence is maintained in two ways. In x86-64-style architectures, CPU 1 invokes OS routines on the other CPUs via *inter-processor interrupts*, which prompt a context switch of CPU 0, 2, and 3 to run OS code [93]. This OS code then executes a privileged *invlpg* instruction which invalidates each core's TLB's copy of the translation. In the example in Figure 4, CPU 0 therefore invalidates its copy of the solid blue TLB-resident page table entry, CPU 2 does not find an entry to invalidate, while CPU 3 invalidates its TLB-resident solid blue page table entry. Alternately, in ARM-style architectures, CPU 1 executes a *tlbi* instruction, which leads to a broadcast message to all other cores to invalidate their TLB-resident copies of the translation. In either case, TLB entries corresponding to the solid blue entry are invalidated using this approach.

Coherence of the page table entries in L1 caches, meanwhile, are maintained using the standard cache coherence protocol (e.g., MESI, MOESI, etc.). This means that when CPU 1 has to update its copy of the solid blue page table entry, it needs to first gain exclusive access to the cache line. To achieve this, CPU 1 sends invalidate messages to CPU 0, 2, and 3 first. CPU 0 and 3 may therefore suffer from false sharing, where all the page table entries in the cache line are invalidated, even though CPU 1 only updated the solid blue page table entry.

(A4b.) If CPU 0 updates the solid blue page table entry instead, there is only a minor difference versus the activities outlined above. Before launching inter-processor interrupts to CPU 1, 2, and 3, CPU 0 would have to invalidate its own local TLB's copy of the blue page table entry.

(A4c.) If all the CPUs maintained private L2 TLBs that were strictly inclusive, the L2 TLBs could act as a filter for the L1 TLBs. In particular, CPU 2's TLB does not cache the solid blue page table entry. If a probe of CPU 2's L2 TLB finds that the blue page table entry is absent, there is no need to look up the L1 TLB. It is possible that the (bigger) L2 TLB may actually have the blue page table entry, in which case the L1 TLB would still have to be probed. However, by strict inclusion, at least a fraction of the coherence traffic on the L1 TLB may be avoided.

(A4d.) In the previous examples, we assumed that the CPU ran the OS, which modified a page table entry. However, hardware page table walkers (which are used to search the page table on TLB misses and are described further in Section 6) can also generate coherence traffic by updating status bits in page table entries. In particular, page table entries maintain dirty bits which have to be set whenever a CPU performs a write to the corresponding page (see Section 6 for details). Therefore, when CPU 1 executes a store to the red striped page table entry, it has to generate cache coherence traffic to invalidate the cache lines of CPU 1 and CPU 3 before writing the dirty bit. However, since dirty bits are not maintained in TLBs (only the virtual-to-physical translation is), there is no need for TLB coherence activity.

---

5 Multiple Page Size Support

In recent years, processor vendors have begun integrating support for concurrent caching of translations for different page sizes in the L2 TLB. These approaches tolerate a higher access time or additional implementation complexity to concurrently support multiple page sizes. While these approaches could technically also be applied to L1

TLBs, L1 access time requirements make such approaches infeasible.

Hash-rehashing: With this approach, the L2 TLB is first probed (or *hashed*) assuming a particular page size, usually the baseline page size [29, 61]. 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 until there is a TLB hit, or all possible lookups have been exhausted because there are no remaining page sizes.

A benefit of hash-rehashing is its relative simplicity of implementation. For this reason, although not confirmed, it is likely that it is the implementation technique that has been used to support both 4KB and 2MB pages in Intel Skylake/Broadwell's L2 TLBs. Having said that, hash-rehashing also suffers from some important problems. The most obvious one is that there are now multiple hit times, depending on the hash or rehash lookup that achieves a hit. This means that some hits can be slower than others. This also means that it takes longer to identify TLB misses (i.e., only after all the hash/rehash lookups are performed can one establish the occurence of a TLB miss). There are several ways of addressing these problems:

① Page size prediction: One way of reducing TLB hit time is to use a hardware structure to predict the page size of a translation [61]. One could then perform the hash (or the first TLB probe) for the predicted page size. Accurate prediction allows faster TLB hits. A key question is how the predictor structure should be designed. Such a structure can be realized as a direct-mapped table that can be indexed using program counter bits and/or virtual address bits. Figure 5 shows both approaches. The predictor shown on the left is indexed using the lower-order bits of the program counter of the memory reference. By using the program counter as the index, the predictor can be looked up early in the pipeline (technically as early as the instruction fetch stage). This enables predictor access latency to be taken off the critical path of the memory reference. The downside is that the page sizes are learned separately for different instructions, even if they refer to the same page. Consider, for example, a scenario where a program processes different fields of a data structure via different instructions. These instructions can map to different predictor entries, leading to predictor state duplication, longer learning time, and more predictor aliasing.

Consequently, using a portion of the virtual address bits can present a more efficient indexing scheme for the page size predictor. With this approach, shown on the right in Figure 5, the page size predictor must still be accessed early in the pipeline. Consequently, rather than extracting bits from the virtual address, bits are extracted from the encoded memory instruction. In particular, the region of the instruction that corresponds to the base register, *rs* may be used. This is used to probe the register file, from where the register value is used to look up the predictor. The intuition is that the final virtual address is unlikely to be significantly different from the base register.

Regardless of the choice of page size predictor, an important point is that both approaches consume area and power. Naturally, it is important for a designer to weigh up these overheads versus the benefits.

② Parallel lookup: A simpler alternative to using a page size predictor is to simply perform multiple parallel lookups of the TLB structure, with one lookup per supported page size. The benefits of this approach is that it obviates the need for predictors, it

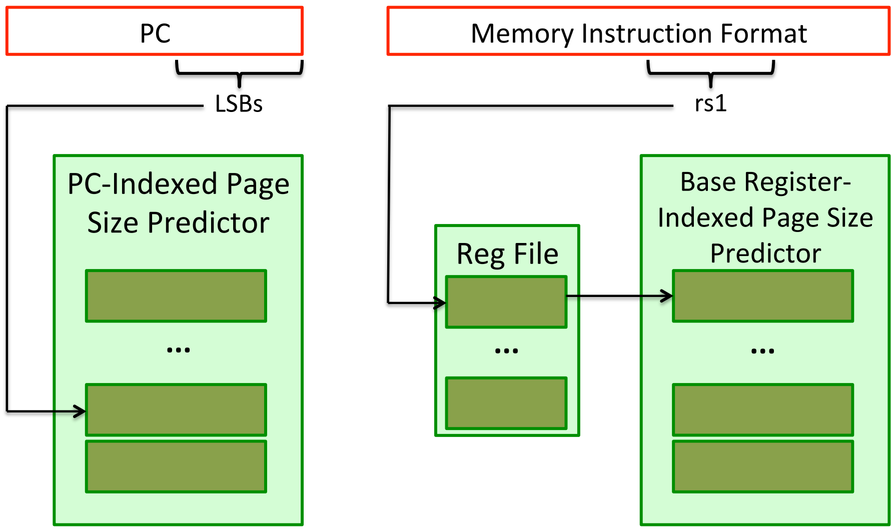

PC
LSBs
PC-Indexed Page Size Predictor
...
Memory Instruction Format
rs1
Reg File
...
Base Register-Indexed Page Size Predictor
...

Figure 5: The figure on the left shows a page size predictor based on the program counter value of the memory instruction probing the TLB. An entry in the predictor is chosen by using a set of least significant bits (LSBs) from the program counter address. The figure on the right shows a predictor based on the base register value of a memory instruction.

makes sure that all TLB hits are quick (and have the same latency), and it permits TLB misses to be identified with the latency of one probe. The downsides are the energy overheads of performing multiple lookups, most of which are spurious since a translation can only be cached with one page size in the TLB structure.

③ Parallel page table walks: Finally, another approach that can (partly) mitigate the increased cost of identifying a TLB miss is that of performing page table walks in parallel with the rehash probes. Consider the case where an initial probe of the TLB results in a miss. At this point, in the conventional case, a new address based on the next page size would be rehashed and used to probe the TLB. To improve performance, it is possible to simultaneously launch a speculative page table walk so that if the rehashes also miss in the TLB, the page table walk can at least partly overlap the latency of the rehashes. While the notion of speculative page table walks does introduce some complexities – e.g., page table entries hold status bits that, among other things, record an ”access” to a page, and these can be erroneously updated by speculative walks – modern architectures already maintain support for such scenarios since speculative page table walks can also occur in other ways (see Section 6).

Skewing: Skewed TLBs are inspired by work on skew-associative caches [79]. The basic idea is to change the notion of set in a TLB. Traditionally, the bits extracted from a virtual page uniquely determine a set – in other words, all the ways in a TLB set are reachable via the same set of index bits because TLB lookup uses a single hash function. Skewed TLBs use an alternate idea – the ways of a set no longer share the

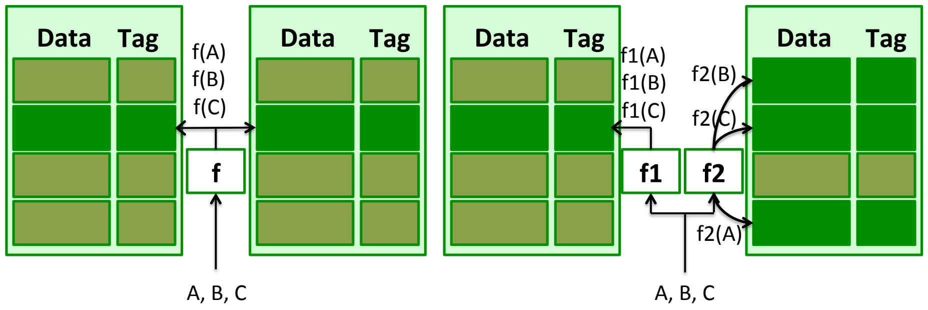


Figure 6: The figure on the left shows a TLB with conventional indexing. A hash function, *f* is applied to all addresses. In this example, addresses *A*, *B*, and *C* hash to the same TLB set, which is made up of two ways, shown in the two separate data/tag array boxes. The figure on the right shows the same TLB with skew indexing. Since there are two ways per set, two hash functions *f1* and *f2* are used. The two hash functions can map to different ways in each set.

same index bits and each way of a set uses its own index function. Figure 6 illustrates this concept. The figure on the left shows a conventional set-associative TLB, while the one on the right shows a skew-associative TLB. In both cases, we show a two-way set-associative TLB with four sets. Each horizontal slice of the data/tag arrays represents a set, each with two ways.

The figure on the left shows that addresses *A*, *B*, and *C* hash to the same set. When using conventional indexing with the hash function *f*, this means that these addresses can be allocated in either of the two ways assigned to that set. In our example, this means that one of the three translations must be evicted as there are only two ways in each set. Contrast this with the figure on the right, which shows an example of a skewed-associative TLB. Since there are two ways per set, a skew TLB uses two hash functions *f1* and *f2*. In this case, *A*, *B*, and *C* hash to the same way in the set on the left via hash function *f1*. However, the translations for these three addresses map to different ways in the set on the right because of hash function *f2*. The benefit of this approach is that the same TLB structure can now accommodate *A*, *B*, and *C* simultaneously compared to the traditional set-associative approach.

To support multiple page sizes, the standard skew-associative structures need to be modestly changed in the following way – a translation maps to a subset of TLB ways depending not just on its address but also its page size. Specifically, the hash functions are designed such that a virtual address can only reside in a way if it has a certain page size. This also means that each page size has an effective associativity in the TLB.

To explain the lookup process, consider an example where we have a 512-entry 8-way skew-associative TLB. Furthermore, suppose that the TLB is designed for a SPARC architecture with 8KB, 64KB, 512KB, and 4MB pages. Choosing among the 64 sets in the TLB requires the use of 6 index bits from the virtual page number. One possible way of engineering the hash functions could be to use bits 23-21 with the following rules: (a) map to ways 0 and 4 if part of a 8KB page; (b) map to ways 1 and 5 if part of a 64KB page; (c) map to ways 2 and 6 if part of a 512KB page; and

(d) map to ways 3 and 7 if part of a 4MB page. The number of supported page sizes is statically designed into in the hash indexing functions and they all have the same effective associativity (two in our example). When an entry needs to be allocated, and hence an entry needs to be evicted, the effective ways of that page size are searched for an eviction candidate. In our prior example, if virtual address A belongs to an 8KB page, then only ways 0 and 4 are searched. Since the potential victims reside in different sets, LRU is infeasible. Hence, skew associative TLBs rely on time stamps placed with each entry, with the entry with the oldest timestamp becoming the primary eviction candidate [61, 78].

While skew-associative TLBs can support multiple page sizes concurrently without the complications of multi-latency hit times or slow idenfication of TLB misses, they do suffer from key drawbacks. The first is that they require multiple hash functions, which can be complex to implement. The second is that they require additional area from time stamps needed for replacement policies. And finally, the more page sizes there are to accommodate, the lower the effective associativity per page size.

Q&A 5

Q5. Consider a CPU with separate L1 TLBs for for 4KB pages, 2MB pages, and 1GB pages. Suppose further that the L2 TLB can support 4KB and 2MB pages, but not 1GB pages. Is there any way that lookups for 4KB pages can cause changes in the contents of the L1 TLB for 2MB pages, and vice versa? Can lookups for 4KB pages cause changes in the contents of the L1 TLB for 1GB pages, and vice versa?

A5. Even though it may seem impossible at first blush, it is possible for lookups for 4KB pages to affect the L1 TLB for 2MB pages *if the L2 TLB is strictly inclusive with the L1 TLB*. In other words, suppose that a lookup for a 4KB page prompts TLB misses at the L1 level. Suppose further that there is a miss at the L2 TLB, prompting a page table walk. Once the translation is retrieved, it is possible that it may evict an existing 2MB page entry in the L2 TLB. If this happens, because the L2 TLB is strictly inclusive of the L1 TLBs, a back-invalidation message will have to be relayed to the L1 TLB for 2MB pages. Consequently, an entry in the L1 TLB for 2MB pages may be invalidated. A similar sequence of events is possible for lookups for 2MB pages. Again, it may be that a 4KB page translation needs to be evicted from the L2 TLB to make room for a 2MB page translation, prompting a back-invalidation of a translation in the L1 TLB for 4KB pages. In contrast, in our hypothetical TLB hierarchy, it is not possible for 4KB page lookups to affect 1GB pages, because the L2 TLB does not concurrently cache translations for 1GB pages with those for 4KB and 2MB pages.

---

6 Page Table Walks

Thus far, we have focused on the L1 and L2 TLBs available per-core on modern multi-core chips. However, even with highly effective TLBs, misses are unavoidable. When TLB misses occur, the page table must be searched or ”walked” to locate the desired translation. These page table walks (also called translation walks) can occur either in software or in hardware. Computing systems have, over decades of design, evolved to embrace one technique over the others. For example, early Alpha systems generally-used hardware-managed translation walks [20]. This evolved into software translation walks on SPARC and ARM architectures. More recently, processor vendors have again adopted hardware translation walkers due to their superior performance.

Software approaches: The key idea with software-managed translation walks is that TLB misses force a context switch to the OS, which is tasked with walking the page table. Processors rely on several implementation techniques to force this context switch. On some older embedded x86-64 processors, a TLB miss would invoke microcode for a trap instruction, which in turn would force the processor to switch to privileged mode. Other techniques involved constructing the TLB datapath to be able to directly signal the core to save state and switch to privileged mode. Regardless of the technique used to invoke the OS, several steps take place in response. First, the pipeline is drained and architectural state is saved. Next, OS code is invoked to, among other things, consult a trap table to vector into the OS routine designed to handle the TLB miss. As the OS code is invoked, instruction and data caches suffer misses in bringing in cache lines requested by the OS (because the OS had not been running in the recent past). Finally, the core executes a OS TLB miss handler with knowledge of the page table organization. The handler searches the page table for the desired page table entry or translation. This process of searching or walking the page table is typically expensive, with multiple sequential memory references. The precise number of memory references is dependent on the page table organization. Consider, for example, an x86-64-style page table, which is organized as a forward-mapped multi-level data structure. Figure 7 illustrates an x86-64 page table. Current x86-64 page tables usually consist of four levels and there is even burgeoning support for five-level page tables.

Figure 7 shows that page table walks are expensive because they require multiple memory references (four in the figure). To represent page tables in a space-efficient manner, x86-64 page tables use multiple levels to represent a program's address space. Each level is itself stored in a page of memory; hence we refer to each level as a page table page. Since baseline pages in x86-64 architectures are 4KB, and because page table entries are 8 bytes, this means that each page table page maintains 512 entries.

To walk the page table, the virtual address is first separated into a page offset and the virtual page number. Then, the virtual page number is separated into four 9-bit indices, each of which is used to index into one of the four page table pages comprising the different levels. In our example, we assume that the virtual address can be decomposed in a hexadecimal sequence of (0b9, 00c, 0ae, 0c2, 016). This means that in the first step, a page table walk requires that we concatenate the physical address of the first entry in the root page table (which we call *L4* and is known as as *Page Map Level 4* or *PML4* in x86-64 terminology) with the first 9-bit index of the address or *0x0b9*. A lookup at this location (highlighted in the L4 table) reveals the physical page number (or *PPN*) of the next level of the page table (*L3*). We concatenate this physical page number (*0x042*) with the next 9-bit index, *00c*, to look up the L3 page table page. We carry on with this process until we reach the leaf page table page or *L1*, which finally tells us the physical page number of the virtual address we wanted to translate.

Naturally, traversing this multi-level page table is expensive, especially because it

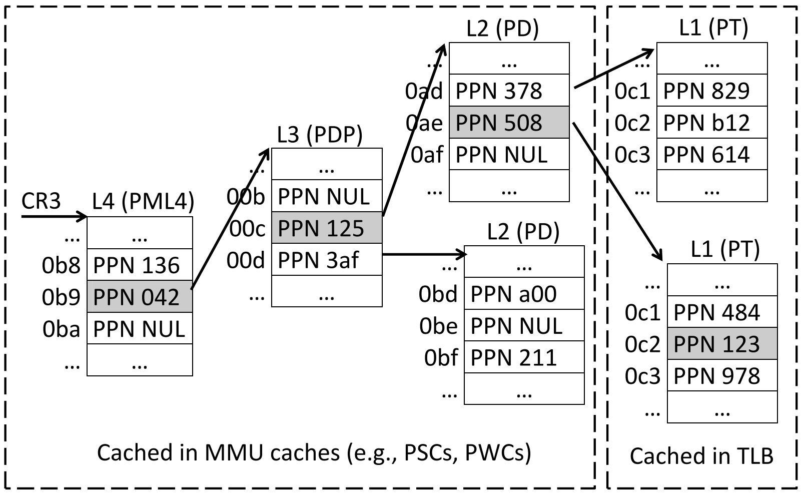


Figure 7: Example x86-64 page table, with four page table levels. The *CR3* register stores the physical address of the root level of the page table. Each page table level stores the physical page number of the next page table page, where *PPN* stands for *physical page number*. The highlighted path through the page table is for a virtual address that is made up of the following hexadecimal values concatenated together (0b9, 00c, 0ae, 0c2, 016). TLBs cache entries from the leaf level or the *L1* level of the page table. Because the page table walk can be expensive, dedicated memory management unit (MMU) caches, which we discuss in Section 7, cache entries from the root page table page (*L4*) as well as the *L3* and *L2* levels.

requires several memory references, all of which must occur sequentially. Therefore, if walks are performed by the interrupt handler in the OS, it is a long process that can easily require tens to hundreds of clock cycles in practice [18, 20]. Finally, when the desired translation is identified, it is filled in the hardware TLBs (both L1 and L2 if the TLBs are mostly- or strictly-inclusive, only the L1 TLB if the TLBs are exclusive). At the end of this process, the core context switches back in the user application – which has to warm up its instruction and data caches again – at which point the memory instruction that originally missed in the TLB is *replayed*.

It is important to note, from Figure 7, that processor vendors use various techniques to try to accelerate page table walks. In fact, L1 and L2 TLBs are used to cache page table entries from the leaf or *L1* levels. But in addition, as we will discuss in Section 7, more advanced address translation implementations also maintain dedicated Memory Management Unit (MMU) caches for the *L4*, *L3*, and *L2* page table levels. While MMU caches can technically be used in tandem with software-managed translation walks, they are typically only found in systems with hardware-managed translation walks. Therefore, we defer a discussion of these structures to subsequent sections.

Translation storage buffers: As software-managed translation walking has matured, several optimizations have been proposed to accelerate their operation. One popular technique has been the use of Translation Storage Buffers (TSBs), originally developed by SPARC architects [11, 68]. TSBs are software caches (usually direct-mapped) and store frequently-used portions of the page table. They are accessed early in the OS handler to seek out the desired translation fast (hopefully with a single memory reference) before falling back on the page table walk. In other words, the OS uses the lower-order bits from the virtual address that suffered the original TLB miss to look up the desired entry in the TSB. If the translation is found in the TSB, the OS handler can return without suffering the four memory references of the page table walk. Otherwise, the page table is walked. When the desired translation is located, it is inserted into both the software TSB and the hardware TLB structures.

An important question with TSB design is how it should be sized. Since it is a software structure, it can be sized dynamically and grown to arbitrarily large structures [11]. However, excessively large TSBs can pollute the processor caches and ultimately decrease performance. Therefore, most real-world TSBs are sized to a fixed capacity; i.e., translations in the TSB may have to be evicted to make room for new ones.

Managing status bits: Our discussions have focused on the virtual-to-physical page translations in the processor page tables. However, page tables (and TLBs) also maintain information about the permissions associated with a virtual page (i.e., whether it is readable, writable, and/or executable), attributes of the page (i.e., whether it is a kernel or user-level page, whether the page is globally shared across multiple address spaces), and information about how the page has been manipulated (i.e., access bits, which track page hotness, and dirty bits, which track whether a page has been written to). When the OS walks the page table, it must interact with these status bits appropriately.

Consequently, after the OS performs a translation walk, it checks the protection bits of the translation. If the access is permitted as per the read/write/execute permissions, only then is the physical page number returned. If the access is not permissible, the OS is vectored to a new interrupt handler which takes care of this ”protection fault”.

Page table entries also maintain access and dirty bits. These are used by the OS to track whether a page is in the working set of a program, and whether it needs to be written back to memory on eviction because its copy in main memory is stale, respectively. Access bits are typically set by the TLB miss handler when a translation is loaded into the TLB after a page table walk. Depending on the page replacement policy implemented by the OS, it may opt to manipulate these access bits in the future. For example, with *LRU with second chance policies*, the OS replacement policy periodically clears access bits to see whether their corresponding pages are accessed again the near future. This permits the OS to maintain relatively up-to-date information about page hotness. Similarly, the dirty bit can be set by the OS when there is a TLB miss to the store instruction. When the replacement policy evicts a page which has its store bit set, the OS ensures that the page is written back to secondary storage.

Pros and cons: The benefit of software-managed translation walks is that they allow the OS to organize and manage page tables in a flexible manner. Because the OS's TLB handler walks the page table, it can be maintained either as a multi-level page table (like the one showed in our x86-64 example), or in other ways (like inverted page tables or hashed page tables [20]). This permits the OS to manipulate page table organization in a manner that best meets the system's requirements in terms of size of the page table, time taken to search it, and even whether it can be used to store more complex attributes corresponding to page hotness than just a single access bit [2].

The disadvantage of software-managed translation walks is that require pipelines to be context switched and OS code to be invoked for all TLB misses. Modern high-

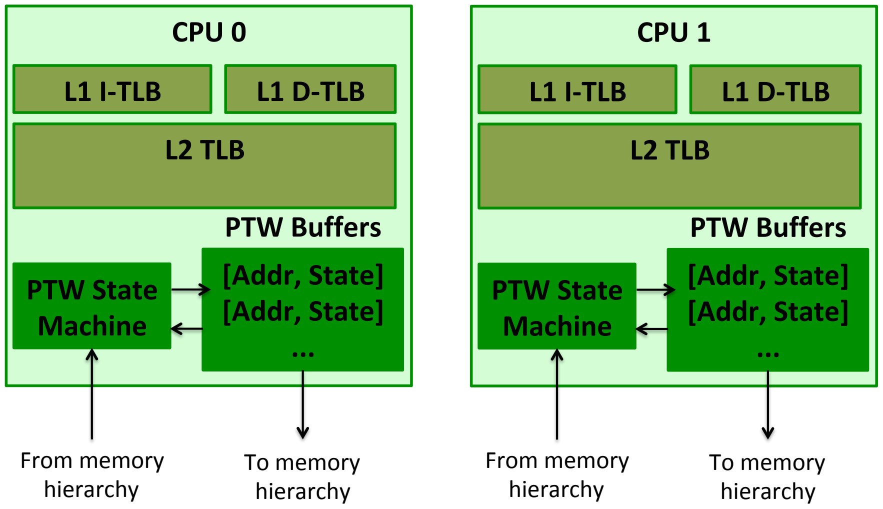


Figure 8: Hardware page table walkers (PTWs) integrated per-core perform hardware translation walks. The hardware page table walker is made up of a state machine (*PTW State Machine*) and buffers that store the status of outstanding TLB misses (*PTW Buffers*) which operate similarly to miss-status holding registers (MSHRs) for caches. These two hardware blocks in tandem launch references to the memory hierarchy to search for the page table, for both instruction and data translations that miss in the TLBs. PTWs can be used in tandem with any number of TLBs and not just the two-level TLB hierarchy shown.

performance processors maintain sizeable architectural state that needs to be saved on these context switches, use deep pipelines that need many cycles to drain and fill, and use many architectural predictors that are polluted by these context switches. Recent studies have shown that even in the absence of cache/predictor pollution effects, processors like Intel's Haswell/Broadwell like take in the range of 700 clock cycle to accomplish a context switch [93]. Consequently, most processor vendors have replaced software translation walks with hardware translation walks.

Hardware approaches: The goal of hardware translation walking is to improve performance by integrating per-core hardware units that understand the page table's organization and can walk it without context switching the main processor to OS code. To enable this, most processor vendors build hardware page table walkers (or PTWs) per core, as shown in Figure 8. Hardware PTWs are made up of two components. The first component is a state machine that is built to understand the organization of the page table supported by the architecture. The second component is the PTW buffer, a set of registers that maintain information about each outstanding TLB miss. PTW Buffers are similar to miss-status holding registers that track outstanding cache misses [20].

To understand the operation of the hardware PTW, consider the x86-64 page table example in Figure 7 again. Suppose further that there has been a TLB miss for the translation shown in the shaded path in the page table. At this point, rather than invoking the OS and executing a TLB miss handler, the core can activate the hardware

PTW. The hardware PTW usually has access to a register that maintains information about the physical address of the root of the page table. In our example in Figure 7, this corresponds to the *CR3 register*, which stores the physical address of the L4 page table level. The PTW state machine concantenates the contents of the *CR3* with the first 9-bit index from the virtual address (0x0b9 in our example). This information is used to update the state field of the appropriate outstanding miss recorded in the PTW buffer – in this case, the state field records the fact that the PTW is about to launch a memory reference for the information in the desired L4 page table. The reference is launched to the memory hierarchy, where it probes the L1/L2/LLC caches, and then potentially main memory. When the information stored in the L4 level is returned (the physical page number 0x42 in our example), it is relayed to the PTW state machine. The state machine concatenates this returned address with the 9-bit L3 index from the virtual address (0x00c in our example), updates the outstanding miss buffer to indicate that a upcoming memory request for this L3 page table level, and launches the memory request. This process continues until the entire page table walk completes and the desired translation is extracted from the page table. At this point, the translation is inserted into the TLBs, and the memory reference that prompted the original TLB miss is replayed [18, 19]. In this manner, the page table is walked with no intervention from software. While conceptually easy to understand, there are important aspects of hardware PTW design complexity. We now discuss some of these complexities:

Microcode injection: Correct address translation often requires the use of microcode instructions that are not fetched as ISA-level instructions from the user or kernel levels. In particular, TLB misses invoke the hardware PTWs, which in turn invoke microcode operations that perform the loads from the page table levels. These microcode operations are pushed into the execution core of modern out-of-order processors. These microcode operations can, however, be not just load operations to walk the page table levels, but also store operations that modify status bits, as we next discuss.

Managing status bits: When using purely hardware translation walks, hardware PTWs are tasked with updating the access and dirty bits of page table entries. These updates are considered to be ”sticky” from the point of view of the walker – i.e., only the OS is capable of clearing a bit when it is set. Furthermore, PTWs use microcode instructions to atomically update these bits. From a memory consistency standpoint, these updates must be guaranteed to be committed before the load/store with which they are associated have committed [54, 55, 73].

Concurrent page table walks across cores: Since each core maintains separate hardware PTWs, they are permitted to simultaneously walk page tables. If the applications running on different cores are from different address spaces, the PTWs would walk their distinct page tables. However, even in the case where multiple cores run different threads from the same address space, multiple PTWs are permitted to access the same page table to satisfy TLB misses on different cores. Multiple threads may update access and dirty bits simultaneously so it is possible for multiple cores to update access/dirty bits in PTEs that fall in the same cache line. The cache coherence protocol is in charge of arbitrating the order in which these updates occur and are made visible to other cores.

```
1.   L.D      F6, 34(R1)    // Effective address of 0x100012
2.   L.D      F2, 45(R2)    // Effective address of 0x100123
3.   L.D      F4, 31(R3)    // Effective address of 0x010012
4.   L.D      F8, 22(R4)    // Effective address of 0x020111
5.   ADD.D    F12, F6, F2
6.   ADD.D    F13, F4, F8
7.   ST.D     F12, 34(R1)
```

Figure 9: Suppose that the first load suffers a TLB miss. With software translation walks, the CPU would have to context switch to the OS and run it to walk the page table. Only after this would the remaining instructions be executed. With a hardware translation walk, however, the CPU could continue executing independent instructions in parallel with the hardware PTW servicing the TLB miss. Since the second load instruction has an effective address mapping to the same virtual page number (assuming 4KB pages), this would not be an independent instruction. However, load instructions 3 and 4, as well as the add (instruction 6) are independent and can be executed while the TLB miss is being serviced.

Pros and cons: The benefit of hardware PTWs is the performance that they offer over software alternatives. By obviating the need for context switches on TLB misses, several hundreds of clock cycles for a TLB miss can typically be converted into tens of clock cycles. The downside of hardware PTWs are the additional area and power needs of PTW state machine and buffers and their reduced flexibility versus using TLB miss handlers in software. The first problem is not severe – recent studies show that hardware PTWs do not account for more than 5-7% of the area of typical L2 TLBs on modern chips, and add significantly less than 1% additional area to each core [14, 69, 70]. The second problem, however, is a bigger concern. For hardware PTWs to work effectively, their state machine must be designed in a manner that is aware of the particular page table organization. In our x86-64 example in Figure 7, the PTW state machine was aware that the page table was organized as a four-level structure with 512 page table entries each. This means that page table formats are fixed at design time and cannot be changed even if we discover that at runtime, there may be other page table formats that could be more efficient. Although OS-managed walks can support this flexibility, hardware translation walks achieve much higher performance because of the elimination of OS context switches, and also two other reasons:

① Overlapping TLB misses with useful work: If hardware PTWs are integrated on CPUs that support out-of-order execution, it is possible to hide at least part of the TLB miss latency with the execution of other streams of independent instructions. Consider, for example, the snippet of code shown in Figure 9. Suppose that instruction 1, which performs a load from memory, suffers a TLB miss. For architectures with software translation walks, the OS is expected to execute a TLB miss handler to locate the desired translation. While this handler executes on the CPU, no other instructions from the user-level program can be executed. However, with hardware translation walks, this restriction can be lifted. In the example in Figure 9, the third, fourth and sixth instructions can be identified as being independent by the underlying out-of-order mechanism (e.g., scoreboarding, Tomasulo's algorithm, etc.) and can be issued and executed out of order. Note that the load in instruction 2, however, will not be executed until the TLB miss from the first instruction is serviced because its effective address is to the same virtual page (0x100) as the first instruction. Since the fifth and seventh instructions are dependent on the second and first instruction, these are also not executed until the TLB miss is handled. Nevertheless, hardware PTWs permit at least part of the TLB miss to be overlapped by independent instructions, mitigating the performance impact of page table walks.

② Concurrently handling multiple misses: One of the implicit requirements to overlap independent instructions with TLB misses is that hardware PTWs enable TLBs with support for hits under misses. That is, even though the first load suffered a TLB miss, the third and fourth loads were permitted to look up the TLB (and enjoy hits) because they correspond to different virtual page numbers. However, a key question remains – when the hardware PTW is engaged in a page table walk, what if a parallel TLB lookup also suffers a miss? Can hardware PTWs handle multiple concurrent TLB misses?

The first generations of hardware PTWs were designed to satisfy only one outstanding TLB miss. There were many reasons for this – lower hardware complexity (the PTW buffers in Figure 8 need only a single entry), simpler microcode control (support for concurrent TLB misses requires microcode for two page table walks to be injected into the pipeline simultaneously), and simply the fact that TLB misses were rare enough (although expensive when they occurred) that multiple TLB misses were rarely encountered. However, more recent processor designs have begun to incorporate PTWs with support for multiple misses for several reasons. First, architects continue to exploit more ILP per CPU by accommodating progressively larger reorder buffers; consequently, the chances that a TLB miss may occur while a page table walk is in progress continue to rise. Second, modern TLB hierarchies are generally expected to service several hardware threads via simultaneous multi-threading or hyperthreading. The greater the number of hardware threads, the higher the likelihood that several threads may encounter TLB misses concurrently. Third, emerging hardware accelerators like GPUs execute hundreds to thousands of threads concurrently and often encounter concurrent TLB misses.

The bottomline is that hardware PTWs currently have support for multiple outstanding misses. As shown in Figure 8, the PTW buffers maintain multiple entries, one for each outstanding miss. This is the technique used, for example, in Intel's Haswell/Broadwell/Skylake chips which can support 2-4 outstanding TLB misses, or GPUs, which can support several tens of misses [28, 29, 69, 70, 72, 87].

Hybrid approaches: Although rare, hybrid page table walkers that combine the best of hardware and software page table walks are employed by some architectures. Consider, for example, the way Linux supports ARM V7 architectures. Linux maintains two page tables – a software managed one, and a hardware managed one that the MMU also has access to. On TLB misses, a hardware PTW can walk the hardware-managed page table to populate the TLB. However, hardware PTWs cannot set dirty or access bits. Instead, the dirty bit is emulated by granting hardware write permission if and only if the page is marked writable and dirty in the software page table. In other words, a write to a clean page causes permission faults. This forces the OS to run a handler that marks the page table entry as dirty. After the hardware table is updated with new permissions and book keeping is done, the user mode program is started at the replayed instruction. In general, the benefit of a hybrid approach is that TLB misses can be handled quickly (in hardware), but the OS can maintain access/dirty information in a manner that better tracks the hotness of pages than traditional hardware-only PTWs.

Q&A 6

Q6. Although a detailed treatment of virtualization is out of the scope of this appendix, we provide a gentle introduction through the following exercise. Consider a virtualized system where user-level programs make memory references in a virtual machine running a guest operating system, which sits atop a hypervisor. In this model of execution, two levels of address translation are required. 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. Assuming that the page table walker has access to both page tables, how many memory references are expected for a page table walk in this virtualizated environment? You may assume an x86-64 architecture.

A6. Figure 10 shows a two-dimensional page table walks for virtualized systems. A guest virtual page (GVP), which is not shown, is translated to guest physical page (GPP). This is done by concatenating the L4 index bits from the GVP with a special CR3 register, the guest CR3, which maintains the system physical page (SPP) of the root of the guest page table. This generates a GPP (which we call GPP Req in our example), which must be converted to an SPP. This SPP is used to look up the L4 guest page table level (gL4). The four-level nested page table (nL4-nL1) is used to generate this SPP. Then, the SPP is combined with the GVP's L3 index bits to look up the gL4, which provides the GPP of the gL3. Again, the nested page tables must then be looked up, and so on. This process continues until the GPP of the L1 is known. This then can be used to look up the nested page table one last time to determine the final SPP. Overall, this two-dimensional page table walk requires 24 memory references, which is a significant factor over the native execution case. For this reason, address translation performance is particularly problematic in virtualized environments. To partly mitigate this overhead, processor vendors include dedicated caches, MMU caches and nested TLBs, to short-circuit some of these memory references. While nested TLBs are out of the scope of this appendix, we discuss MMU caches in the next section.

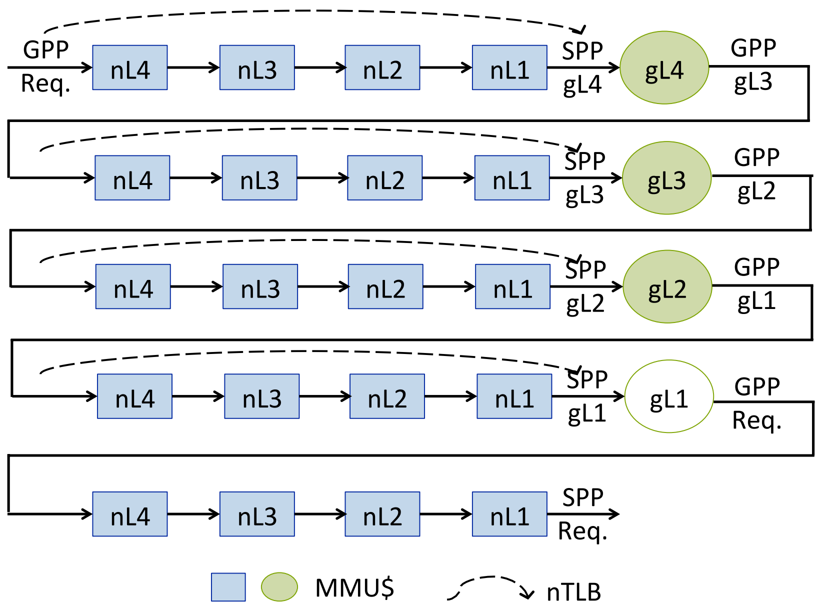


Figure 10: Two-dimensional page table walks for virtualized systems. Nested page tables are represented by boxes and guest page tables are represented by circles. Each page table's levels from 4 to 1 are shown. TLBs cache translations from the requested guest virtual page (GVP) to the requested system physical page (SPP).

7 Memory Management Unit Caches

Although hardware PTWs can mitigate the detrimental performance impact of TLB misses, the quest for ever-increasing performance has led processor vendors to explore other techniques to reduce page table walk latencies. One means of achieving this – which many commercial processors use – is through the use of *Memory Management Unit (MMU) caches* or *prefix caches* [10, 16]. MMU caches were originally built based on the observation that page table walks are lengthy because they require multiple sequential memory references. For example, x86-64 architectures require four sequential memory references to traverse all four levels of the page table. MMU caches are used to short-circuit several (ideally three) of these memory references, as detailed below.

While multi-level page tables require multiple accesses to translate a virtual address, accesses to the first few levels enjoy significant temporal locality. This is natural – after all, each page table entry in the root level of an x86-64 page table maps 512GB contiguous chunks of the virtual address space in contrast to the leaf level of the page table, which maps only 4KB chunks of the virtual address space. In general, translation walks for two consecutive pages in the virtual address space are likely to use the page table entries in the L4, L3, and L2 page table levels since the 9-bit indices used to select the entries are extracted from high-order bits of the virtual address, which change less frequently.

When the MMU accesses the page table table, it does so through the memory hierarchy. Technically, the temporal locality enjoyed by the entries in the upper levels of the page table can be exploited by the processor caches. However, these processor caches must also cache user-level data, and other kernel-level data structures. Consequently, page table entries are often only found in the LLC, which can take several tens of cycles of access. It is this problem that MMU caches solve. MMU caches are low-latency caches for the upper level page table entries. As Figure 7 shows, while the leaf level of the page table is cached in the TLB, the upper levels are cached in the MMU caches.

MMU caches may be designed in several ways. One way is to design them similar to conventional data caches, where each entry stores a page table entry and is tagged with the physical address of the corresponding location in the memory-resident page table. Examples include AMD's page walk cache [15]. Alternately, Intel uses an approach where MMU caches are indexed by parts of the virtual address – these structures are known as paging structure caches. Finally, research proposals have also suggested an alternate format, known as a translation path cache [10]. For these tagging schemes, elements from different levels of the page table can be mixed in a single cache (a unified cache), or placed into separate caches (a split cache). We detail these options next.

Page walk caches: Consider standard instruction and data caches. The hardware PTW generates a physical address based on the page table level that is to be accessed and the appropriate 9-bit index from the virtual address. This is the address that is used to look up the memory hierarchy. Page walk caches use the same approach. Each entry in a page walk cache is tagged with the physical address in the page table. These tags are the size of the physical page number plus the size of a page table index. (As previously described, L1 entries of the x86-64 page table are not cached in the page walk cache as they are cached in TLBs instead.)

① Unified page walk caches: These structures are high-speed read-only caches that store page table entries from all the upper page table levels together. Entries from the different levels of the page table are generally mixed together – and usually treated with similar priorities as per the replacement policy – and indexed via their physical address. The diagram on the left in Figure 11 shows an example unified page walk cache after the completion of the page table walk in Figure 7 for a virtual address with decomposed 9-bit indices (0b9, 00c, 0ae, 0c2, 016). Tags are made up of a combination of the base physical address (*Base*) and the 9-bit index that corresponds to the L4, L3, or L2 entry (*Index*). Data – or the *Base* physical address of the next page table level – is stored in the *Next* field.

Consider the cached state shown in Figure 11 and suppose that the MMU subsequently tries to translate virtual address (0b9, 00c, 0ae, 0c3, 103). The hardware PTW will initiate a lookup for the page table entry stored in location 0b9 in the L4 page table (which is located in physical page number 613, as indicated by the *CR3* register). This entry is present in the page walk cache (as indicated by the third entry with a *Base* of 613 and *Index* of 0b9), with a *Next* value of 042. Because of this page walk cache hit, the PTW does not need to launch this memory reference to the cache hierarchy, improving performance greatly (MMU caches are generally single-cycle access). In the next step, the hardware PTW looks up the unified page walk cache for a *Base* of

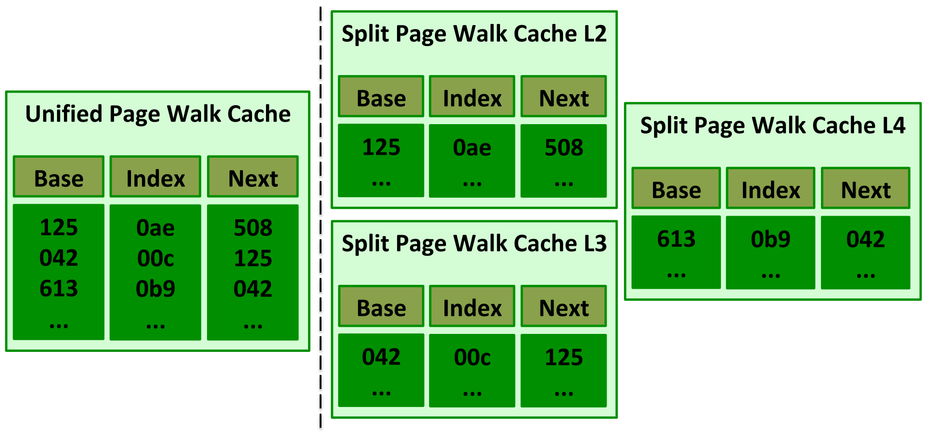


Figure 11: On the left, we show an example of a unified page walk cache, where entries from the L4, L3, and L2 levels are mixed. Tags are made up of a combination of the *Base* and *Index* fields, and we show the state of the page walk cache after the walk in Figure 7. The figure on the right shows the state of split page walk caches – where there are separate structures for L4, L3, and L2 entries – after the same walk.

042 and *Index* of 00c (from the 9-bit L3 index found in the virtual address). Again, this results in a page walk cache hit, obviating the need for a lookup of the memory hierarchy. This process continues until the address of the L1 page table level is found (508), concatenated with the L1 index (0c2), and followed with a memory reference made up of this concatenated address.

Without a page table cache, all four of these accesses to page table entries would have required a memory reference, each of which may hit in the processor caches or have to go to main memory. Instead, with a page walk cache, MMU cache lookups for the first three page table levels hit, and only the L1 lookup requires a memory reference.

② Split page walk caches: The main problem with unified page walk caches is that they complicate replacement policies – since L4, L3, and L2 page table entries can freely intermingle in the cache, and since they map 512GB, 1GB, and 2MB chunks of the virtual address space respectively, their locality properties can differ widely. In this situation, replacement policies can be complicated to implement since traditional cache LRU policies are based on the idea that each line maps an equal amount of the address space.

Consequently, an alternate design for page table caches separates the page table entries from different levels into different hardware structures. We show this split approach on the right in Figure 11. In this design, each individual entry contains the same tag and data as it would in the unified page table cache. The main difference is that each page table level has a private cache, and entries from different levels do not compete for a common pool of slots. This fixes the interference problem among translations for different levels and also makes it easier to build replacement algorithms.

Paging structure caches: Page walk caches tag their entries by physical address; but

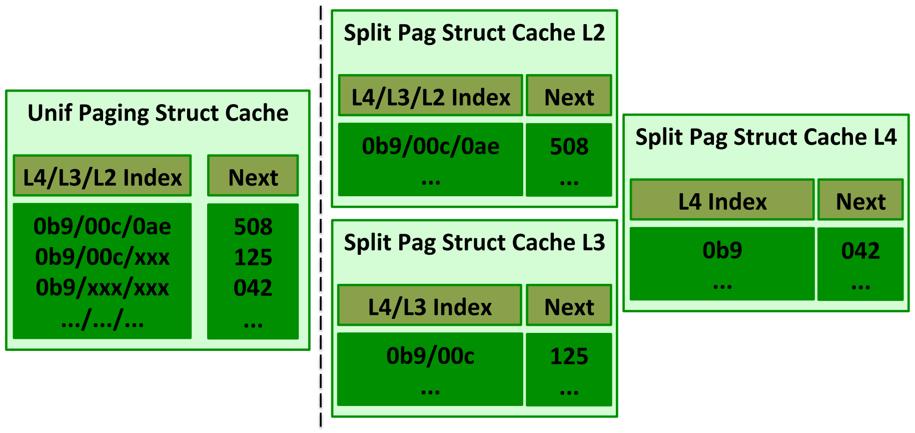


Figure 12: On the left, we show an example of a unified paging structure cache, where entries from the L4, L3, and L2 levels are mixed. Tags are made up of a combination of the 9-bit L4, L3, and L2 indices (or prefixes) from the virtual address. Some of the lower-order bits of this tag are made up of don't care values (*XXX*) depending on whether the corresponding entry represents an L4, L3, or L2 translation. we show the state of the page walk cache after the walk in Figure 7. The figure on the right shows the state of split page walk caches – where there are separate structures for L4, L3, and L2 entries – after the same walk.

an alternative is to tag entries by the indices in the virtual address. This is the approach taken when designing paging structure caches. With this approach, L4 entries are tagged with 9-bit L4 indices, L3 entries are tagged with 9-bit L4 indices and 9-bit L3 indices, while L2 entries are tagged with 9-bit L4 indices, 9-bit L3 indices, and 9-bit L2 indices. With this approach, data from one entry is not needed to look up the entry at the subsequent lower page table level. In other words, lookups for L4, L3, and L2 can be concurrently performed. In the end, the MMU selects the entry which has the longest matching prefix with the virtual address because it allows the hardware PTW to skip over most levels.

① Unified paging structure caches: The diagram on the left in Figure 12 shows a unified paging structure cache, where entries from the L4, L3, and L2 levels are intermingled. We show the state of the cache after the page table walk for (0b9, 00c, 0ae, 0c2, 016) shown in Figure 7. Now suppose that the MMU has to translate the virtual address (0b9, 00c, 0dd, 0c3, 929). At this point, the hardware PTW will attempt to fullfill a longest virtual address prefix match in the paging structure cache, so that it can minimize the number of lookups of the structure. This means that the hardware PTW begins by trying to match L4, L3, and L2 indices. This leads to matches for the L4 and L3 indices – the entries for 0b9/00c/xxx and 0b9/xxx/xxx both match (while 0b9/00c/0ae does not, because the desired L2 index is 0dd). Of these two matches, the first represents the longest prefix. Therefore, the hardware PTW extracts the data, (125), and concatenates the L2 index with it (0dd) to generate the desired physical address of the page table entry within the L2 page table page. This is then sent to the memory hierarchy, followed by a reference for the L1 page table page. In this example,

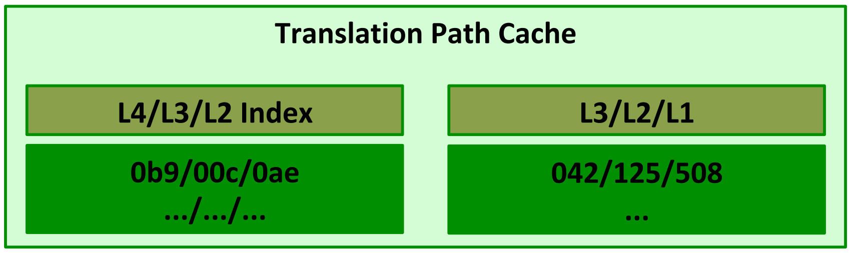


Figure 13: An example of the contents of a translation path cache after the page walk in Figure 7. Tags are made up of three 9 bit indices and all three 40-bit physical page numbers are stored for all three page table levels.

the paging structure cache permits skipping over two lookups in the page table.

② Split paging structure caches: The diagram on the right in Figure 12 shows a split paging structure cache design. This operates similarly to the unified paging structure cache, but like split page walk caches, eliminates the problem of interference among L4, L3, and L2 entries (due to differing locality properties). Furthermore, splitting the structures makes it easier to look up all the paging structure caches in parallel.

Translation path caches: In the paging structure caches in Figure 12, the tags for the three entries for a single page table walk path have the same content. The L4 and L3 entries maintain several *don't care* bits, but the partial translations they do hold have the same content. Consequently, researchers have proposed translation path caches, structures where this kind of area waste is mitigated by storing all three physical page numbers in a single entry [10, 16]. In such caches, a single entry represents an entire path, including all intermediate entries, rather than individual entries within a walk.

Figure 13 shows the contents of a translation path cache after the page table walk in Figure 7. All the data from the walk is stored in a single translation path cache entry. If the MMU starts to translate virtual address (0b9, 00c, 0ae, 0c3, 929), the L1 entry is discovered. In particular, the hardware PTW finds the entry in the cache with the tag (0b9, 00c, 0ae) and reads the physical page number 508 of the L1 page location. Later, if the PTW has to translate virtual address (0b9, 00c, 0de, 0fe, 829), this address shares a partial path (0b9, 00c) with the previously inserted entry. Therefore, the translation path cache provides the physical address of the L2 entry.

Design comparison: Having detailed various types of MMU caches, we now compare their design attributes.

① Indexing: The indexing scheme of the MMU cache impacts its overall operation and efficiency. The schemes we considered can tag MMU cache entries by physical or virtual addresses. For example, page walk caches use physical addresses to tag page table entries, and are essentially like versions of conventional processor data caches but dedicated to upper-level page table entries. This makes them easy to design, which is why they have been adopted in commercial processors by vendors like AMD [15]. However, they have an important drawback – they require multiple lookups for translations in top-down order. Specifically, the result of the L4 entry search is required before the L3 entry search can begin, because the L4 entry gives the physical page number of the L3 page table page, which is needed to generate the physical address of the L3 page table entry. Similarly, the L2 search is dependent on the result of the L3 search. The need for top-down searches presents several problems. First, the page walk cache must be looked up multiple times for a walk, accruing access latency and energy, even if all the levels of the walk are resident. Paging structure caches, on the other hard, use parts of the virtual address (the indices) as tags for the entries. This allows them to be searched in any order (L4 first, L2 first, or in parallel), making them efficient.

② Coverage: The coverage of an address translation cache is the amount of the virtual address space that can be represented by the finite number of entries it supports. For example, a 128-entry TLB for 4KB pages has a coverage of 128$\times$4KB or 512KB. Unfortunately, characterizing the coverage of an MMU cache is not as straightforward. For example, consider a scenario where a hardware PTW lookup of a paging structure cache hits on the L3 entry, but not the L2 entry. In this case, the translation is accelerated by the MMU cache, but still requires a memory reference to fetch the L2 page table entry – should this lookup be counted as being covered or not?

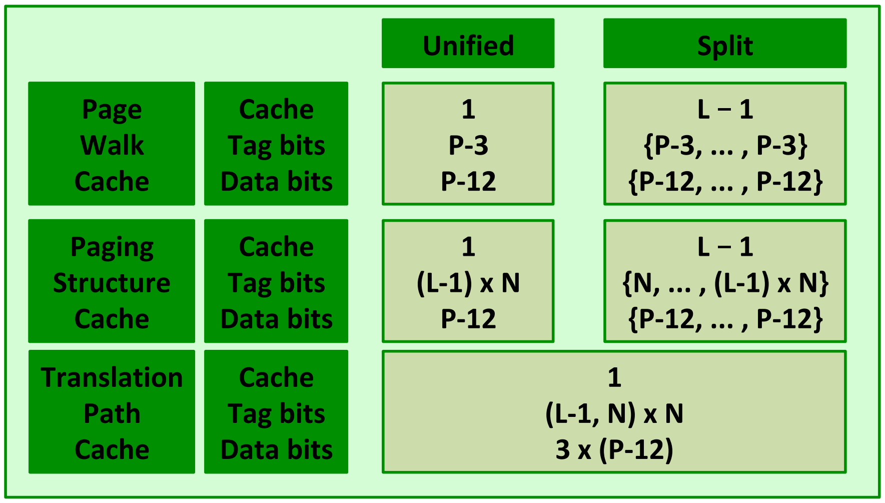
Prior work has taken the strict position that MMU coverage is only said to occur when no memory references are made to fetch L4, L3, and L2 page table entries [10, 16]. In general, when equally sized, paging structure caches are able to cover a larger portion of the address space than page walk caches. This is because paging structure caches generally make more efficient use of their entries than page walk caches. For a page walk cache to provide coverage according to its strict definition, it must simultaneously hold L4, L3, and L2 page table entries. In contrast, a single paging structure cache entry can provide the full path through L4, L3, and L2 levels.

③ Complexity: MMU caches can be built as either fully-associative or set-associative caches using CAMs and SRAM. The different variants have differing tag and data widths, leading to differing implementation complexity. Figure 14 shows the complexity of all variants of MMU caches that we have considered. We separate analysis for page walk caches and paging structure caches, showing the number of caches that they need, the number of bits per tag in each entry of the cache, and the number of bits per data field in each entry of the cache. Furthermore, the analysis is parameterized by the number of levels in the page table, L, the number of bits in a physical address, P, and the number of offset bits in a page table index for a particular level, N.

For modern x86-64 architectures, L is 4, P is 52, and N is 9. This means that paging structure caches and translation path caches require significantly smaller tags than page walk caches (27 bits versus 40 bits). This means that generally, paging structure and translation path caches are smaller and more energy-efficient.

8 MMU Integration: Putting Everything Together

We have presented several microarchitectural structures that are involved in address translation, from two-level TLB hierarchies and page table walkers to many types of MMU caches. In addition, these structures have important interactions with other mi-



		Unified	Split
Page Walk Cache	Cache	1	L − 1
	Tag bits	P-3	{P-3, … , P-3}
	Data bits	P-12	{P-12, … , P-12}
Paging Structure Cache	Cache	1	L − 1
	Tag bits	(L-1) x N	{N, … , (L-1) x N}
	Data bits	P-12	{P-12, … , P-12}
Translation Path Cache	Cache	1	
	Tag bits	(L-1, N) x N	
	Data bits	3 x (P-12)	

Figure 14: Number of caches, tag bits per entry, and data bits per entry for page walk caches and paging structure caches. We separate results for unified and split configurations. Note that translation path caches are neither unified nor split. All complexity analysis is parameterized by the number of levels of translation, L, the number of bits in a physical address, P, and the number of offset bits in a page table index for a particular level, N. As an example, in x86-64 processors, L is 4, P is 52, and N is 9.

croarchitectural components like load-store queues and processor caches. In sum, these components interact with one another in a manner that creates significant indirections, and can thus be complex to understand. We devote this subsection to detailing the interactions among all these microarchitectural components. Naturally, the nature of these interactions is determined by the specific microarchitectural components – i.e., physically-tagged versus virtually-tagged caches, hardware versus software translation walks, etc. To simplify our discussion, we assume an MMU that integrates with a virtually-indexed, physically-tagged L1 cache, and the use of hardware PTWs.

① Load/store queue to TLB: When CPUs generate memory references, they are inserted into load/store queues, which are the first microarchitectural structures that interact with address translation hardware [54, 55]. While little has been publicly disclosed about load/store queue-TLB interactions in commercial microprocessors, we present a plausible overview of these interactions based on information gleaned from published work and patents [54].

One of the hallmarks of high-performance processors is the need for memory prediction and disambiguation [26]. The prediction stage anticipates dynamic same-physical address dependencies between stores and loads to try to preemptively prevent correctness problems. The disambiguation stage later ensures that all predictions were correct. This pairing ensures that synonyms – situations where multiple virtual addresses map to the same physical address – can be detected while keeping the TLB off the forwarding critical path.

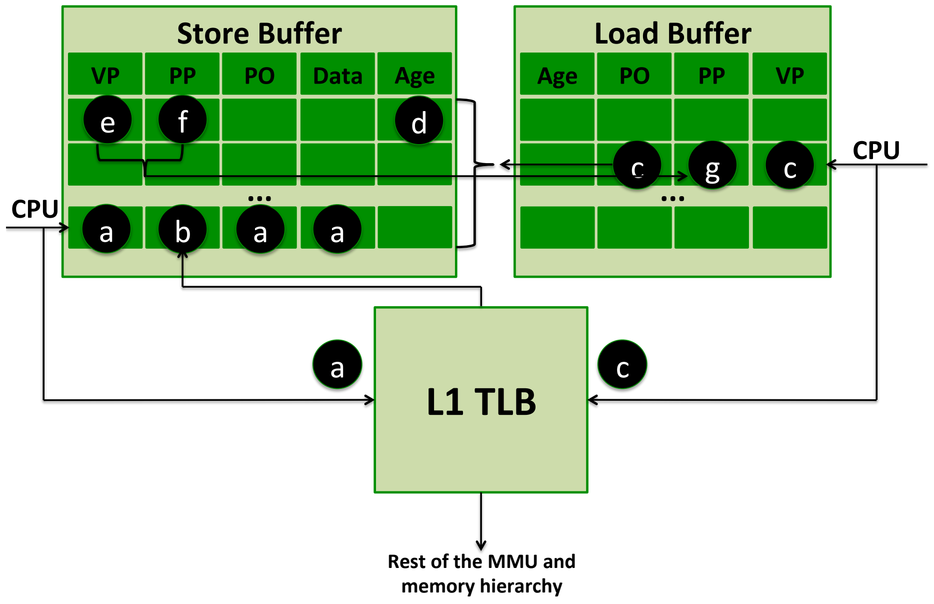


Figure 15: Interactions between the store buffer, load buffer, and TLB hierarchy. The bubbles represent various steps in the memory dependency prediction and disambiguation process. Both load and store buffers maintain a separate field for the virtual page (*VP*), physical page (*PP*), and page offset bits (*PO*), which are the same in the virtual and physical address spaces and hence do not need to be repeated. Additionally, store buffers maintain a data field, while load buffers do not. Finally, both buffers maintain age fields to track the program order in which the loads and stores were inserted in the buffers.

One plausible mechanism is shown in Figure 15. We separately show the L1 TLB (the remainder of the address translation and memory hierarchy are present but not shown), the load buffer, and the store buffer. Both load and store buffers maintain separate fields for the virtual page number (*VP*) and the physical page number (*PP*). Furthermore, they maintain a single field for the page offset, *PO*, since it is equivalent in virtual and physical address spaces. While the size of the page offset can vary depending on whether it pertains to base pages or superpages, the load and store buffers need just enough bits for the smallest page size, which is 4KB in x86-64 systems. Therefore, x86-64 load/store buffers require 12 bits in the page offset field.

In Figure 15, we show how stores and loads operate. All stores write their virtual address and data into the store buffer in parallel with searching the TLB (shown in ⓐ). This means that the virtual page, page offset, and data fields in the relevant store buffer entry are populated. The physical page number is later written into the store buffer entry when the TLB returns the data (shown in ⓑ).[1]

Figure 15 also shows load operation. Loads write their virtual page number and page offset bits in parallel with TLB access (shown in ⓒ). Then, each load compares these page offset bits against the page offset bits of all older stores in the store buffer

[1]We discuss the events constituting a TLB miss in subsequent subsections.

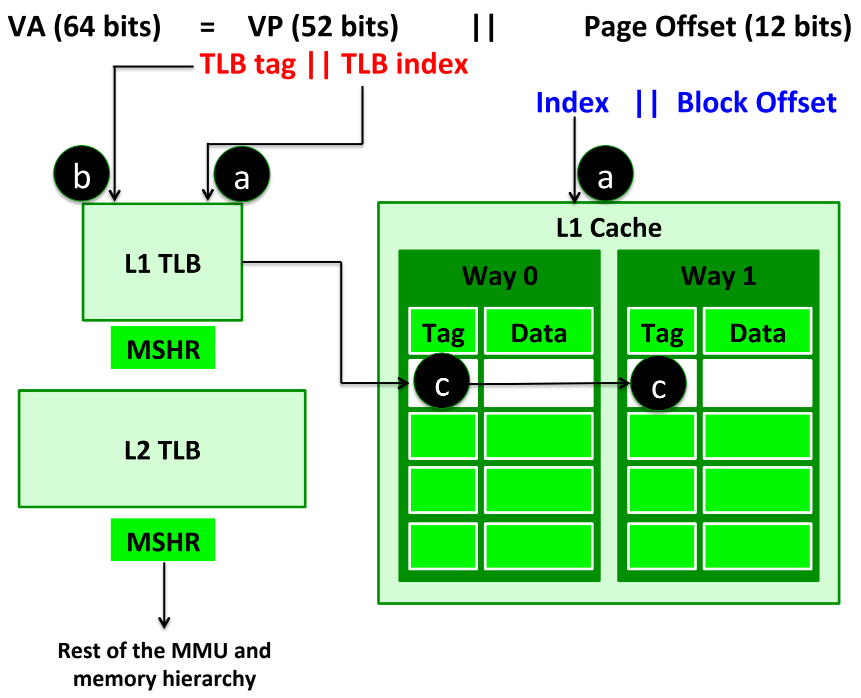


Figure 16: Virtually-indexed, physically-tagged cache operation. A virtual address is split into a virtual page number, *VP*, and page offset. The virtual page number is split into TLB index and tag, while the page offset houses the block offset and (part of) the cache index bits. The TLB lookup proceeds in parallel with identification of the cache set. We show an example of an L1 TLB hit, where the L1 cache lookup completes fast. Furthermore, our L1 cache assumes two ways per set, where each set is a horizontal slice of the the L1 cache.

(shown in ⓓ). If there is no match among the existing entries, this clearly means that none of the existing stores are to the same address as the load instruction. If an older store does exist but its address is not generated yet (so the page offset field is blank), we predict there to be no dependencies between the store and load.

If there is a match in page offset bits, the load then compares higher-order bits. If the load's virtual page number matches the store's virtual page number (shown in ⓔ), the store forwards its value to the load (shown in ⓖ). If there is no match, the load compares its physical page number against the store's physical page number, stalling if either instruction's TLB access has not yet returned a translation. If the physical page bits match, the store will forward its data to the load. Otherwise, the load will have determined that no dependency exists between the load and that particular store.

Finally, load/store queues must also reflect disambiguation; stores must determine whether their speculations were legal. Therefore, before each store commits, it checks the load buffer to see if any younger loads matching the same physical address have speculatively executed before it. If so, it squashes and replays them [54].

② TLB hierarchy to cache hierarchy: The load/store queue's interactions with the

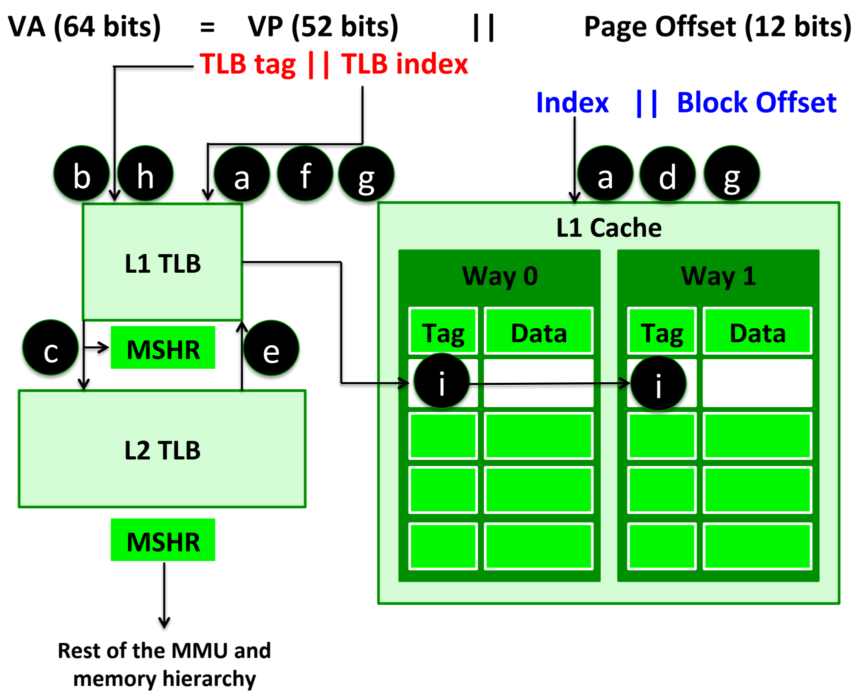


Figure 17: Similar setup as Figure 16, except that the L1 TLB initially misses. When this happens, the L1 cache lookup is aborted, freeing up its ports for lookups from other instructions that are non-dependent on this TLB miss. The L2 TLB is then looked up. If the translation is found in the L2 TLB, it is filled into the L1 TLB, and the memory reference that initially missed in the TLB is replayed. This time, the cache lookup completes.

TLB present just the first step in the life of a memory reference. We now detail the following steps which involve interactions between the TLB and the L1 caches. Without loss of generality, we focus on virtually-indexed physically-tagged (VIPT) caches, as shown in Figure 16. Consider the case where VIPT caches are used on an x86-64 architecture. The 64-bit virtual address is split into a virtual page number (*VP*) and a page offset. The basic idea is that to mitigate the access latency of the TLB, the TLB lookup can be performed in parallel with part of the L1 cache lookup. To do this, Figure 16 shows that the virtual page number can be split up into a portion that is used as the TLB index to select the desired TLB set (ⓐ). In parallel, the L1 cache can be probed to complete set selection (also ⓐ). With an effective VIPT organization, the tag match in the TLB (ⓑ) completes by the time the tags in the selected set in the L1 cache are ready to be compared (ⓒ).

Thus far, we have assumed that a situation where the L1 TLB lookup results in a hit. Figure 17 shows the case where the L1 TLB lookup results in a miss. When this happens, an entry in the L1 TLB's miss status holding register (MSHR) is allocated to record the virtual address of the memory reference (ⓒ). In the next step (ⓓ), the cache access is aborted so as to free up the cache port for use by other instructions that are independent of the TLB miss and can proceed without being blocked due to a structural hazard. Suppose, further, that the desired translation is found in the L2 TLB. At this point, it is filled into the L1 TLB (in ⓔ), after which the memory instruction is replayed by the CPU (in ⓕ). The replay re-initiates cache and TLB set lookups (in ⓖ), TLB tag match (in ⓗ). This time, the translation is found in the L1 TLB, so a cache tag match can proceed (in ⓘ).

Naturally, our description presents only one possible implementation of the TLB-L1 cache interface. Furthermore, our design presents two subtle points. The first is that the VIPT approach can only work if the bits used to select the cache index are guaranteed to be identical in physical and virtual address spaces. In x86-64 bit architectures, only the lowest 12 bits are guaranteed to be identical as they correspond to the smallest page size of 4KB. Therefore, even though x86-64 architectures support 2MB and 1GB pages – and hence, one might consider the idea of extracting cache index bits from the lowest 21 and 30 bits of the virtual address – we only use the lowest 12 bits for cache index selection because the page size is unknown until TLB lookup.

The second point is that using the lowest 12 bits of the virtual address to house the cache index bits implies a cap on the number of sets that a VIPT L1 cache can accommodate. For x86-64 bit architectures with 64-byte cache blocks, only a maximum of 6 bits remain for the cache index, implying a maximum of 64 cache sets. After 64 sets, L1 VIPT caches can be grown by increasing the number of ways in a set rather than the set count. This can present power and access time problems, although techniques like page coloring can partly mitigate this problem [24, 50, 62, 80, 95].

③ Page table walker to MMU caches and cache hierarchy: In our previous example in Figure 17, we assumed a situation where the lookup missed in the L1 TLB, which prompted the VIPT cache access to be aborted. We then assumed an L2 TLB hit. However, it is also possible that the L2 TLB lookup results in a miss. When this happens, an MSHR entry for the L2 TLB is allocated. This MSHR is the same as a PTW buffer in Figure 8 and maintains information about all the page table walks that have been initiated. Since the VIPT cache access was already aborted at the L1 TLB miss, nothing else needs to be done on the L1 cache side.

A hardware PTW interacts with the MMU caches and the processor's data caches. In the best case scenario, the lookups for the first three page table levels in the x86-64 architecture are hits, leaving the reference for the leaf page table entry for a lookup into the data caches and main memory. Note that references for instruction address translation are relayed to the data caches too (as opposed to the instruction caches), since page tables are treated as data by the processor.

Figure 18 shows the steps involved in a page table walk, assuming that we use a page walk cache as the MMU cache. Suppose that a CPU invokes the hardware PTW, which is responsible for generating physical addresses to the page table entries in the L4-L1 page table levels. For each of the L4-L2 levels, the PTW probes the MMU cache first to ascertain whether that particular page table entry can be quickly found. In the event that the desired page table entry is not in the page walk cache, a miss occurs. When this happens, the physical address of the page table entry in the L4-L2 levels is used to probe the memory hierarchy, made up of data caches and main memory. When the page table entry is found, it is filled into the MMU cache, which is not shown in

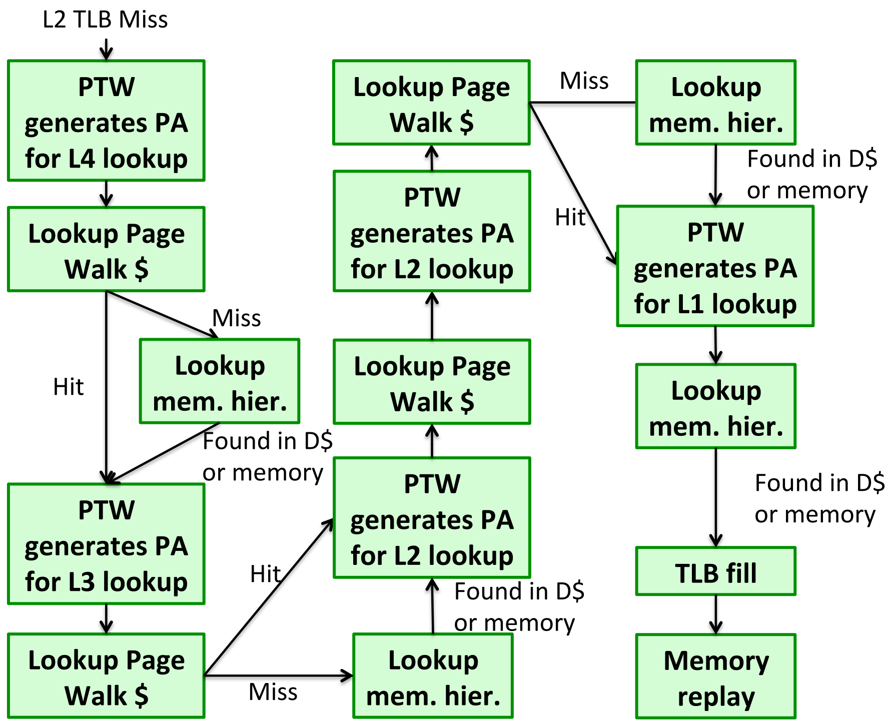


Figure 18: Steps in a page table walk, after the L2 TLB miss, for a page walk cache. The PTW generates physical addresses for the L4-L3 page table levels, which are looked up in the page walk cache. These lookups can results in hits or misses. When misses occur, the memory hierarchy consisting of data caches and main memory are used to identify the page table entry of that page table level. Finally, a lookup for the L1 page table level goes directly to the memory hierarchy. This is followed by a TLB fill and a memory replay.

the state diagram in Figure 18 because it occurs in parallel with address generation of the next page table level. Finally, when the PTW generates the physical address of the leaf page table level, it directly probes the memory hierarchy for it as this entry does not reside in the page walk cache. When the entry is located in the data cache or main memory, it is filled into the TLB. Finally, the CPU replays the memory reference, prompting a VIPT lookup of the L1 cache and TLB hierarchy again.

④ **Faults:** Figure 18 shows a scenario where the desired translation was identified and filled into the address translation hardware as appropriate. However, there may be situations for which this may not be immediately possible because of a need for OS intervention. All classes of these situations are generally known as faults.

The first class of faults correspond to situations where there is a memory protection problem. This may arise for many reasons. For example, it may be that a malicious program attempts to write another program's code segment. Pages maintaining code are marked read and execute only. When a program attempts to write part of a code page, a fault alerts the OS of this protection violation. The OS can then take a variety of measures to deal with this situation. As another example, memory protection support can be used to implement techniques like ”copy-on-write”, which are used by modern OSes to crate address spaces for child processes on system calls like ”fork”, or memory deduplication, where multiple copies of physical frames with identical data can be replaced with a single instance of the physical page [33, 68, 77, 86].

Consider, for example, the case of page deduplication, where two virtual pages map to separate physical frames with identical data. Suppose that the OS runs software that identifies this redundancy and decides to save memory. At this point, the OS changes the page table such that two virtual pages map to the same physical page. Furthermore, it write protects both pages (i.e., neither page has write permissions). At the same time, the OS maintains a separate data structure (for example, a virtual memory area tree in Linux [27]), to record the fact that the virtual pages sharing this physical page are in fact writeable, but that their physical page contents must be copied to a new location before a write can proceed. Now suppose that a program on the CPU issues a store instruction to this write-protected page. The desired translation is either in the TLB or is located by the hardware PTW. In either case, a protection fault ensues as the store instruction does not have permission to update the write-protected page. This prompts OS code for a protection fault handler to be invoked for execution on the CPU. This handler scans the virtual memory area and identifies that the target page is being write-protected because it is part of a copy-on-write operation. Consequently, the handler creates a copy of the physical frame in another physical frame location, updates the page table to reflect the fact that the virtual page of the offending store instruction should now point to this physical frame copy, marks this page writable, and then switches context back to the user process. At this point, the store instruction is replayed. Since it now has the appropriate write permissions, the store instruction continues without faulting.

Faults may also occur when a virtual page does not have a physical frame assigned. These kinds of faults come in two flavors – minor (or soft) page faults, and major (or hard) page faults. Minor page faults occur when the CPU accesses data in a page that is resident in memory, but is not marked in the page tables as having been allocated. When this happens, the OS page fault handler merely needs to update the page table with information about the fact that the page is loaded it memory and does not need to be read in from secondary storage. Minor page faults can occur for many reasons. For example, if a user is running a web browser, then the memory pages with the browser’s executable code can be shared across multiple users. When the second user starts the same web browser, minor page faults would be invoked – the binary would not be reloaded from secondary storage, but the physical frames of the code would be assigned to the virtual pages of the new user. Another example is the use of memory allocation calls like `malloc`. When programs call `malloc`, requesting multiple pages to be allocated, a range of contiguous virtual pages are usually reserved. However, physical frames are assigned only the first touch to each of these virtual pages. These first touches suffer minor page faults so that the OS page fault handler can update the page table accordingly [64, 66]. Finally, pages could have been removed from the working set of a process but not yet be written to disk or erased. For example, HP’s OpenVMS removes pages that do not need to be written to disk (because they are unchanged since they were last read from disk) and places them on a free list. However, these pages are not overwritten until the physical frame is reassigned, meaning it is still available if it is referenced by the original process before being allocated. Since these faults do not involve disk latency, they only suffer minor page faults.

Major page faults occur when a program accesses part of its image resident in secondary storage. When this occurs, the page needs to be loaded into memory. The page fault handler in the OS needs to find a free location: either a free physical frame in memory or a non-free physical frame. The latter might be in use by another process, in which case the OS writes out the data in that page (if it is dirty) and unmarks the page from the page table. This opens up space for the OS to read the desired data from secondary storage into this freed location. Major faults are more expensive than minor faults and add storage access latency to the interrupted program's execution.

⑤ **Memory replay:** Once the TLB miss (and potential page fault) are handled, the original instruction that suffered the TLB miss needs to be replayed. There are two possible scenarios to consider:

Post TLB miss: Suppose that the original TLB miss was serviced with a page table walk, but did not suffer a page fault. In this case (which is shown in Figure 18), the TLB is filled with the missing translation. So when the memory reference is replayed, this time an L1 TLB hit occurs, and the VIPT cache access can go ahead. (Note that the VIPT cache access may not be a hit, since the cache line may be further down the memory hierarchy.) In other words, the replay suffers from no address translation overheads of its own.

Post page fault: Suppose that the original TLB miss was serviced with a page table walk, but that this walk prompted a page fault. Once the page fault handler is executed and appropriate action is taken, one question is whether the translation of the original memory reference is filled into the TLB. Since this translation has to be updated by the OS before it can be filled into the TLB, most microarchitectures do not permit the page table walker to directly fill it into the TLB. This means that the memory replay suffers a secondary TLB miss, invoking another page table walk. However, this is typically not a performance problem – the replay's page table walk is usually fast because the translation has recently been updated by the OS, which means that it is likely in the faster (L1/L2) processor caches and can be quickly retrieved.

While the question of whether the memory replay suffers a TLB miss is important, let us reconsider the question of whether the cache access of the replay results in hits or misses for situations where no page fault occurred. To understand this, consider the diagram shown in Figure 19. We show a process address space, in terms of the radix tree page table layout of the four-level x86-64 architecture. The data that the page table points to effectively acts as a fifth level of this page table (*Data*). The number of entries in this fifth level depends on the size of the particular page and cache line. Figure 19 shows a scenario where we use 64-byte cache lines, and examples of both 4KB and 2MB pages. The 4KB page is comprised of 64 separate 64-byte cache lines, while the 2MB page can be realized with 327,680 separate 64-byte cache lines.

Figure 19 illustrates the relationship between locality of the page table levels and the data that they point to. In general, all memory references traverse this tree, requiring a translation first (the first four levels) and then a data access (the leaf level).

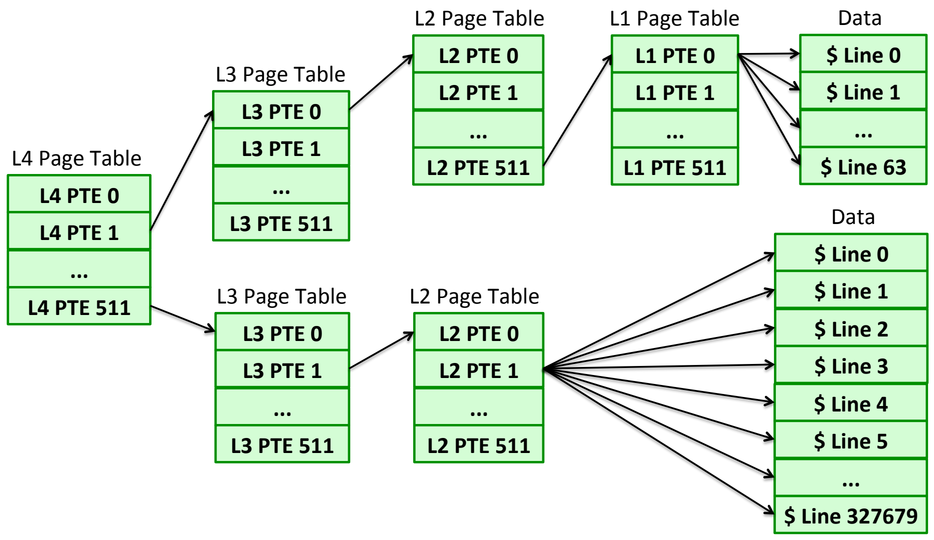


Figure 19: Overview of a process address space, relating the page table with the data that the page table points too, assuming an x86-64 architecture. We assume four-level page tables as usual, but data is the equivalent of adding a fifth level that the L1 page table level points to. We show a scenario where, assuming 64 byte cache lines, a 4KB page maintains 64 cache lines, and another scenario where a 2MB page maintains 327,680 cache lines.

When traversing the levels of the tree from the root to the leaves, locality of reference decreases. This is because the granularity of memory pointed to by an entry in each level decreases as we move towards the leaves. In other words, for any of the data cache lines we show in the 4KB page to have been touched, we would have had to also access the L1 PTE 0, increasing its locality of reference. This locality of reference has important properties for the memory replay after a TLB miss. If a memory reference misses in the TLB, it typically therefore also means that the data that is touched after the memory replay is cold, and hence suffers cache misses too.

Figure 20 shows an example of the access locality relationship between page tables and the data that they point to. We assume a page table walk that requires a main memory or DRAM lookup for the desired page table entry. Furthermore, since TLB misses occur less frequently than accesses to data, it is likely that the desired page table entry is not in an open row buffer within DRAM. Consequently, the row buffer miss is followed by a DRAM array lookup, where the translation ($V \rightarrow P$) is found, and then filled into the caches and TLB. Subsequently, the replay commences. Since the page table entry suffered from such poor locality that it required a DRAM array lookup however, data within any cache line sitting in the physical page frame pointed to by the translation has (usually) even poorer locality and requires a DRAM array lookup too.

It is important to note that there is one exception to the maxim that data pointed to by a page table entry has lower locality of reference than the page table entry. This

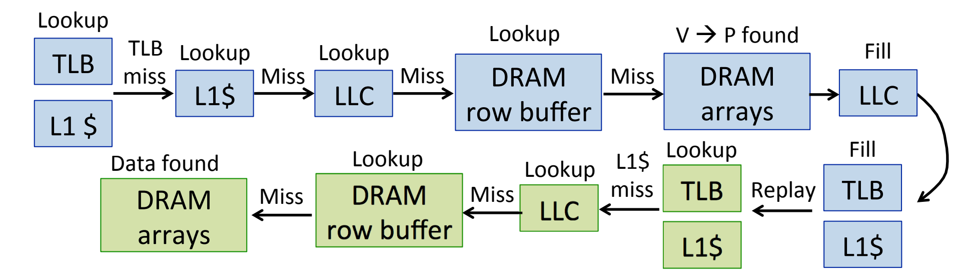


Figure 20: Page table walk and replay timing details. This example assumes a situation where the page table walk requires a lookup of main memory or DRAM. Furthermore, we assume a row buffer miss in DRAM, which requires a lookup of the DRAM array. Once the caches and TLBs are filled, replay commences. Because the access locality of the data pointed to by the page table is poorer than the page table itself, the replay also requires a DRAM lookup (which also misses in the row buffer).

exception occurs occasionally for virtual address synonyms – cases where two separate virtual pages map to the same physical page frame. In such a situation, accesses to the physical data via one of the virtual addresses may ensure that the data is in the upper levels of the cache hierarchy, even if the other synonym virtual page translation is cold. In such situations an access to the synonym virtual page may result in a TLB miss and a page table walk with memory references deep in the memory hierarchy, while the replay more quickly identifies the desired data.

9 Translation Prefetching

Like caches, TLBs and MMU caches are both amenable to prefetching approaches that can increase their hit rates. While there have been several research studies on TLB and MMU cache prefetching [16, 18, 20, 21, 23, 47, 76, 83], we focus on only some of these techniques and point interested readers to the original research papers for more details.

Cache-line prefetching: The simplest form of translation prefetching occurs in the processor's data caches and arises from the fact that cache lines can usually hold multiple page table entries. For example, x86-64 architectures use 8-byte page table entries. This means that cache lines of 64-128-bytes can hold 8-16 page table entries for consecutive virtual pages. In this context, consider a scenario where a CPU 0 performs a load to a memory address in virtual page 3 of a process's address space. Suppose further that this memory reference prompts a TLB miss, and that CPU 0 and CPU 1 share an L2 cache with 64-byte cache lines. On a TLB miss, a page table walk commences and assuming 8-byte page table entries, the page table walk brings in a cache line with translations for virtual page 0-7. This implicit prefetching is beneficial if CPU 0 or 1 later require page table walks for translations corresponding to virtual pages 0-2 and 4-7, assuming this cache line has not been evicted by then. Beyond implicit prefetching of page table entries, cache lines storing page table entries are also amenable to traditional cache prefetching techniques [6, 35, 56, 75]. In particular, our studies suggest that page table entries are prefetched using standard next-cache line prefetching techniques on Intel's Broadwell processor.

Sequential prefetching: We now discuss prefetching techniques for the TLB. Sequential prefetching is the simplest form of TLB prefetching strategy [23, 47]. All prefetched TLB entries are allocated in a *prefetch buffer* that is separate from the TLB. This ensures that prefetching cannot pollute the TLB. In general, TLB pollution occurs in two ways: (1) the algorithm may prefetch a TLB entry that is not used but still evict a resident TLB entry that is useful; and (2) the algorithm may prefetch a TLB entry that is useful, but evicts an existing TLB entry that will be useful sooner in time. Both these situations are eliminated with a prefetch buffer.

Several variations of stride prefetching are possible. In one variation, 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, whther it is a TLB hit or miss. We first probe the TLB to establish that the prefetch target is indeed absent from the TLB and if it is, the prefetch is launched. While this approach is good at prefetching aggressively, its downside is that it can potentially launch many prefetches/probes of the TLB. An excessive number of prefetches can create high memory bandwidth, and can actually end up degrading performance despite a higher TLB hit rate. In contrast, another approach might be to launch prefetches only on TLB misses. While this approach has been shown to significantly reduce the memory traffic problem by modulating prefetch rates, it can often miss opportunities to prefetch effectively.

In practice, systems that use sequential prefetching take a hybrid approach. In this approach, TLB prefetches can be triggered in two ways. The first is on TLB misses. The second trigger can be when the CPU accesses an item that that was previously prefetched. In this case, the prefetch algorithm builds confidence about its ability to prefetch correctly and can initiate a prefetch for the next adjacent translation. In implementations with prefetch buffers, this is readily implementable – a hit to an item in the prefetch buffer initiates a prefetch for the page table entry of the ± 1 virtual page.

In the design of simple stride prefetching, two design decisions worthy of further investigation are: Is there a need for a separate prefetch buffer for prefetched instructions or are there ways to prefetch directly into the TLB without excessively polluting it? What replacement policy should be used in the prefetch buffer (if it exists) or in the TLB with demand-based and prefetched TLB entries (if the prefetch buffer is subsumed entirely in the TLB)? These questions currently remain unanswered.

Arbitrary stride prefetching: The limitation of stride prefetching is that it captures only ± 1 patterns. Many real-world applications employ more sophisticated spatially-adjacent patterns, which is an observation that cache prefetchers have been designed to leverage in the past [35]. Arbitrary stride prefetching can similarly be used for TLBs [47]. Figure 21 shows the additional hardware that is necessary to perform arbitrary stride prefetching. Furthermore, Figure 22 shows how this table and other prefetching logic fits into the TLB datapath.

As with standard sequential prefetching, the CPU's lookups probe the TLB and prefetch buffer concurrently. When both the TLB and prefetch buffer miss, a page table walk is invoked. Although not shown, the translations identified via the page table

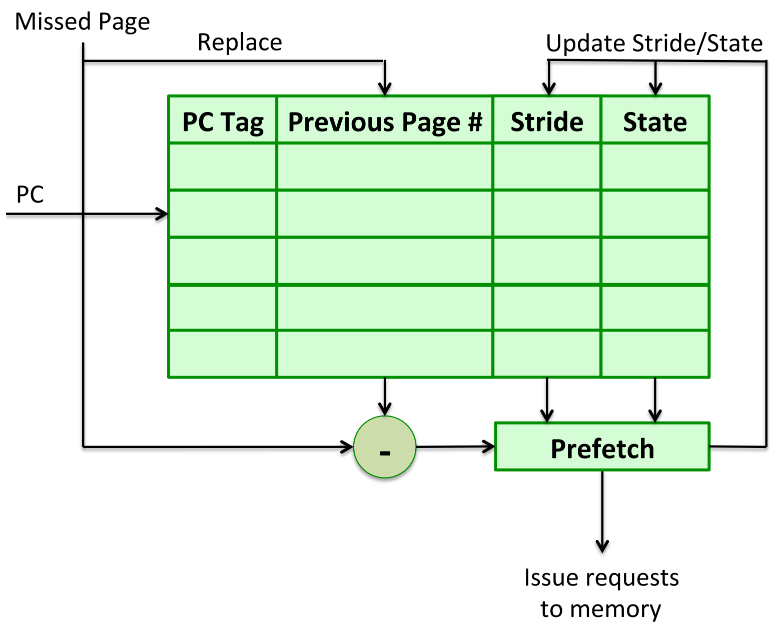


Figure 21: Adaptive sequential prefetching techniques are supported by maintaining a table which records previous strides from a single PC to the memory addresses it generates. The table is tagged with a PC, and maintains fields recording the previous virtual page number, and the previous stride that was encountered by this PC. The state is used to record whether this entry should be permitted to initiate prefetches. Strides should be seen consistently N times (where N is a design parameter) before prefetches are initiated.

walk are inserted into the TLB. Prefetches are invoked in two ways in our example – TLB misses, and prefetch buffer hits. Naturally, it may also be possible to invoke prefetches on TLB hits too, but for reasons already discussed, restricting prefetches to only TLB misses and prefetch buffer hits performs best [47]. The prefetch logic block responsible for generating prefetch targets is realized using the table in Figure 21. The table is indexed using the PC value. Each row maintains information about the last virtual page number that was accessed by the instruction at this PC (i.e., the *Previous Page Number* field), the stride encountered between the last two virtual pages requested by the instruction at this PC (i.e., the *Stride* field), and the state (i.e., the *State* field), which records how many times this stride has been seen. The *Previous Page* field is updated each time the PC touches this instruction, and the prefetch is initiated only when there is no change in the stride for more than N references by that instruction, where N is a design time parameter. In practice, it is frequently set to the value of 2 [47]. Such a safeguard is used to mitigate spurious changes in strides.

Markov prefetching: Thus far, the TLB prefetching schemes we have discussed detect regularity of accesses by detecting deltas between successive address references. However, is it possible that 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

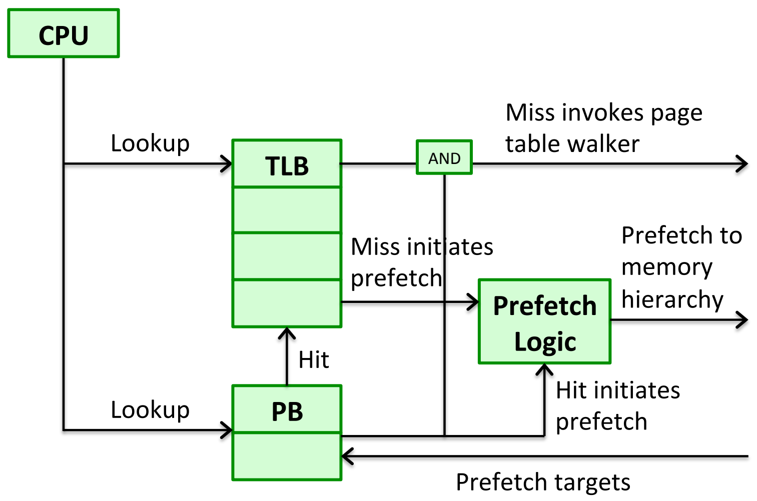


Figure 22: The prefetch logic is placed adjacent to the TLB and the prefetch buffer (PB). CPUs look up both TLB and PB concurrently. TLB misses and PB buffer misses/hits invoke the prefetch logic block, which is tasked with sending prefetch requests to the memory hierarchy. Prefetched items are then filled into the PB.

between successive virtual pages. Markov-based TLB prefetchers, like Markov-based cache prefetchers [35], aim to detect these patterns. The structure used in this case is 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. Figure 23 shows the main component in the prefetch logic, the Markov state table.

The prediction table is indexed using the virtual address that has suffered a TLB miss. Each row maintains N slots, where each slot contains a virtual page number that is expected to be missed upon soon in time after the virtual page that has currently suffered a miss. Therefore, these virtual pages become prefetch candidates.

Initially, all prefetch target slots are empty. When a TLB miss occurs, the table is looked up. Since the prefetch slots are empty at this point, no prefetches are initiated. The missing page is stored in the *Previous Page Number* register. Consequently, when the next TLB miss occurs, we use this *Previous Page Number* to identify the entry corresponding to the last TLB miss. In that location, we go to the first free slot and enter the virtual page number of the current miss, so that the table can record the association between the two virtual pages that caused successive misses.

In designing a Markov predictor, there are several potential design considerations. The number of prefetch slots dictates how many prefetches are initiated per TLB miss, and although we show a table that permits two prefetches, real-world systems often prefetch as many as four TLB entries [21, 47]. More prefetches can reduce TLB misses substantially, but also run the risk of generating too much memory traffic or increasing TLB pollution and prefetching inaccuracy. Another design choice is the replacement policy that is used to select where to enter a new virtual page number if all slot fields

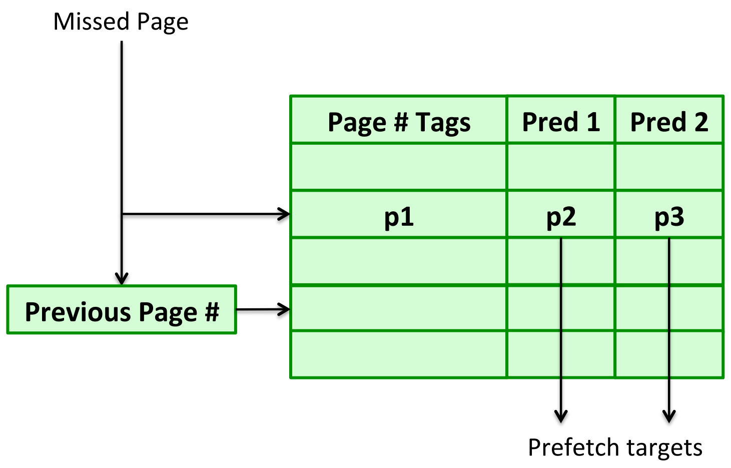


Figure 23: Markov predictors use a table to store relationships among states (virtual pages previously encountered by the program). Each entry maintains a virtual page previously seen, while the contents of the row maintain information about prefetch targets.

are full. While past work has considered using LRU policies to determine this [47], others are also possible and remain open to research exploration.

Recency-based prefetching: While the prefetching approaches considered thus far were originally conceived for caches before adaptation for TLBs, recency-based prefetching is targeted particularly at TLBs [76]. The key insight is that virtual pages that were referenced close together in time in the past will be referenced close together in time in the future. In a sense, this is similar to the insight behind Markov prefetching – however, the mechanism used to leverage this observation is orthogonal and can be used to identify the exact temporal relationship between many groups of pages. In particular, the mechanism used by recency-based prefetching is an *LRU stack* of virtual page numbers. We show this LRU stack, which is composed of both the hardware TLB and the software page table, in Figure 24.

In order to understand the LRU construction and operation better, let us temporarily assume that the TLB is fully-associative and uses LRU replacement. In this case, the top of the LRU stack is effectively realized by the TLB, with the most recently used translation at the top of the TLB's internal LRU stack. The bottom of the TLB's LRU list is housed by the least recently used translation, which is the next candidate for eviction. The remainder of the LRU stack is made up of the software page table entries. The page table is not inherently ordered with respect to the LRU stack. Rather spatially adjacent virtual page numbers have spatially adjacent translation locations in the page table. Therefore, to maintain information about the LRU ordering within the page table, Figure 24 shows that *previous* and *next* pointers are maintained to track the page table entries that were accessed previously and next in time, respectively. Note, further, that ”accesses” of the page table correspond to situations where there were TLB misses.

In the example in Figure 24, we can see that TLB misses occurred for *E*, *A*, and then *B*, in temporal order. Consequently, *E's* next pointer points to the physical address

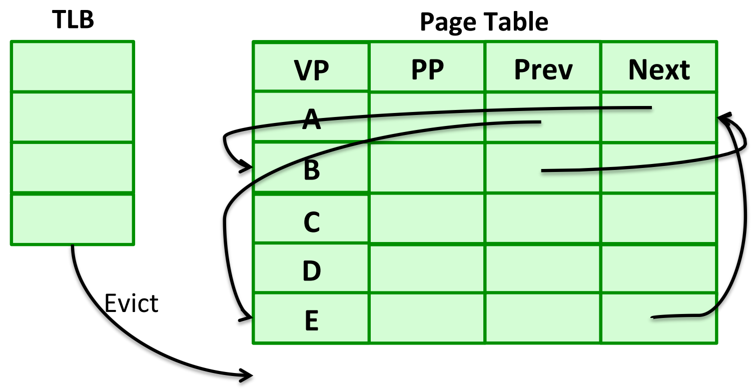


Figure 24: Recency-based prefetching builds an LRU stack of translations. The TLB acts as the top of the LRU stack (assuming a fully-associative TLB). The page table acts as the remainder of the LRU stack. Since the page table is linearly organized based on virtual page number, recency-based prefetching requires the addition of two pointers in the page table, *previous* and *next*, to create the LRU stack. A's previous pointer is E, meaning that E sits higher in the LRU stack, while its next pointer is B, meaning that B sits one step lower in the LRU stack. Evicted entries from the TLB are added to the top of the LRU stack in the page table. In our example, the page table entry corresponding to the evicted translation would have its next pointer updated to E, while its previous pointer would remain null to indicate that the previous pointer is in the TLB.

of translation *A*. *A's* next pointer is *B*, while its previous pointer is *E*. And *B's* previous pointer is *A*. Finally, we must consider what happens on TLB evictions. In this case, the evicted entry becomes the top of the page table's LRU stack. The previous top of the stack is assigned to the next pointer of this evicted translation, while the previous pointer is left null.

Now consider a situation where the CPU injects a memory reference within virtual page *A*, and suppose this is absent from the TLB. A TLB miss ensues, prompting a page table walk. Suppose that the page table entry is found. At this point, the previous and next pointers are scanned by the page table walker. Subsequently, the walker emits prefetch requests for translations *E* and *B*, the previous and next translations, respectively. The page table walker can readily emit these requests without requiring further page table walks because the pointers maintain the physical addresses of the page table entries. Furthermore, prefetcher aggressiveness can be increased, and it may be possible to use the next and previous pointers to identify more opportunities for prefetching too. For example, the page table walker could use the next pointer from *A* to identify the physical address for translation *B*, and could then emit another prefetch for the translation at *B's* next pointer. Naturally, the more aggressive the prefetching, the higher the potential performance rewards, but also worse the memory traffic and pollution. Finally, the overheads of maintaining previous and next pointers in the page table must also be weighed up. The higher space requirements of the page table may, in particular, may lead to more of the processor cache real-estate being expended on page table entries. Assessing the benefits of recency-based prefetching therefore requires a careful analysis of all these factors [47, 76].

Distance-based prefetching: All the prefetching options discussed thus far can be grouped into two categories – prefetchers that require little storage overheads, like sequential and adaptive stride prefetching, and those that require large tables to capture temporal relationships among virtual pages, like Markov and recency-based prefetching. The two categories also differ with respect to learning time – while the first category of prefetchers can learn reasonably quickly (depending on how many times the same stride needs to be observed for prefetches to be initiated), Markov and recency-based prefetchers can take longer because they require larger tables to be populated. The second category of prefetchers does present an advantage over the first – typically, they are more accurate.

Distance-based prefetching is a technique that tries to marry the best of these worlds. The goal is to 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. Distance-based prefetching is based on keeping track of differences (or ”distances”) between successive virtual page numbers. For instance, consider a 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. This is in contrast to a Markov predictor, which needs one entry for each of the six virtual pages in this example.

Figure 25 shows how distance-based prefetching may be implemented. Central to the hardware is a table that correlates patterns of distance values seen through prior memory accesses. Therefore, tags are for the current distance (i.e., the difference between the virtual page number currently prompting a TLB miss and the last virtual page number that suffered a TLB miss, calculated in ⓐ). The tags can be looked up to yield an entry that predicts the next few distances that the program is expected to experience (shown in ⓑ). In our example, the distance table tracks two predicted future distances (although a table could be built to capture more predicted distances). These distances are added to the current virtual page in ⓒ, and prefetch requests for these target virtual page numbers are sent to the memory hierarchy. This current calculated distance is placed in the predicted distance slot for the previous page number (shown in ⓓ). And finally, the previous distance is overwritten with the value of the current distance (shown in ⓔ). Overall, because of its relative modest hardware needs, distance-based prefetching has been shown to be an effective prefetching mechanism. Furthermore, while we have discussed distance-based approaches targeted at improving single-thread performance [47], similar schemes have been adapted to enable threads executing on a single CPU to prefetch TLB entries for threads running on other CPUs [23].

Summary: Figure 26 summarizes the prefetching strategies that we have studied, with a discussion of the approaches, their hardware/software overheads, their predictor table needs, and the additional number of memory references they inject into the memory hierarchy (excluding the references needed for the prefetches themselves). Beyond the parameters captured in the table, however, there are some other important aspects of prefetching that we briefly discuss below.

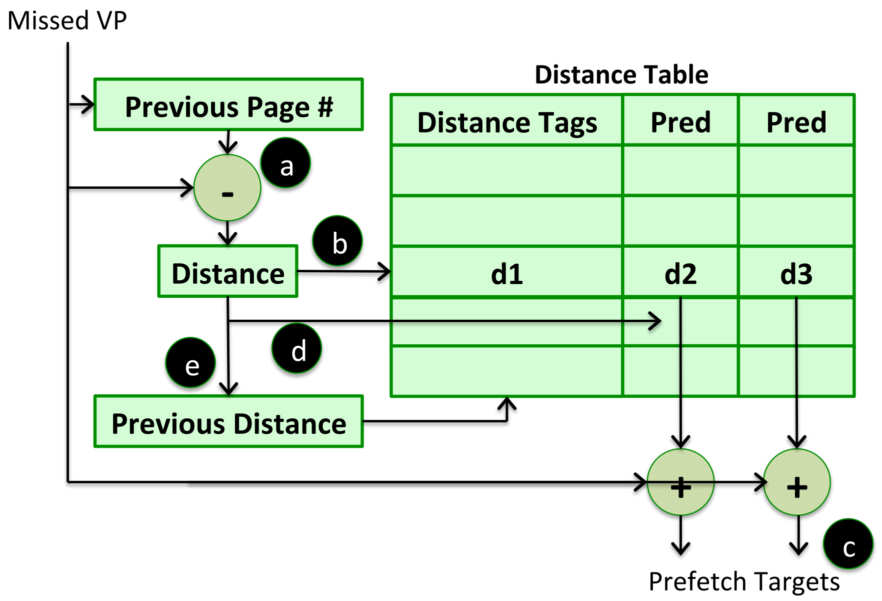
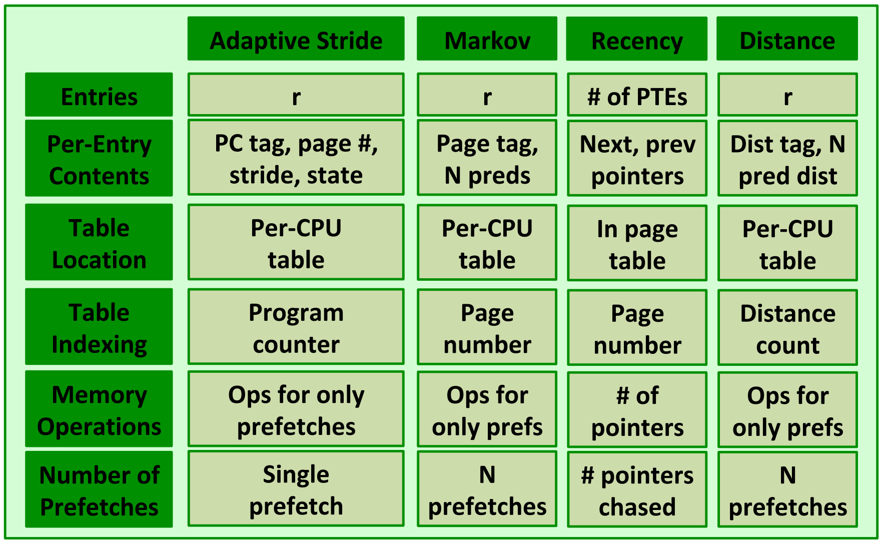


Figure 25: Hardware for distance-based prefetching. Tables are tagged with a distance identifier, which is calculated by taking the distance between the current virtual page missing in the TLB and the previous virtual page that missed in the TLB. Each distance table entry then indicates some number of next predicted distances learned from previous memory accesses. These distances are added to the requested virtual page number to generate prefetch targets, which are then sent to the memory hierarchy.

① Superpages: Our prefetching examples thus far have assumed baseline virtual pages but naturally, the prefetching algorithms have to support multiple page sizes. In general, there is nothing preventing a miss on a baseline virtual page number to prompt a prefetch of a translation for a superpage or vice versa. However, to prefetch translations of any page size, the prefetch buffer has to support for page sizes. For this reason, many proposed prefetch buffer designs are fully-associative rather than set-associative structures [23].

Beyond this, multiple page sizes can make techniques like recency-based prefetching particularly challenging because LRU stacks with entries corresponding to different sizes of the address space are complicated to maintain. For one thing, in a scenario where we add previous and next pointers to an x86-64 page table, the pointers may go to the physical address of different page table levels. For example, a translation for a 4KB page may maintain pointers to a 2MB page. In this case, pointers maintained in the L1 page table level may point to L3 page table levels. Therefore, setting the pointers up may require additional references to traverse the full tree from the root, in case a different page table level needs to be updated.

② Interaction with caches: Translation prefetches bring page table entries into the TLB subsystem, but should the prefetch also bring the cache line holding the desired page table entry into the processor's higher level caches (e.g., L1 caches)? On one hand, if



	Adaptive Stride	Markov	Recency	Distance
Entries	r	r	# of PTEs	r
Per-Entry Contents	PC tag, page #, stride, state	Page tag, N preds	Next, prev pointers	Dist tag, N pred dist
Table Location	Per-CPU table	Per-CPU table	In page table	Per-CPU table
Table Indexing	Program counter	Page number	Page number	Distance count
Memory Operations	Ops for only prefetches	Ops for only prefs	# of pointers	Ops for only prefs
Number of Prefetches	Single prefetch	N prefetches	# pointers chased	N prefetches

Figure 26: Table of prefetching techniques, with parameters, including the number of entries, contents of each entry, location of each table, and indexing mechanism for the table. Additionally, we show the number of memory references used by the prefetching approach (excluding the number of memory references for the actual prefetch), and the number of prefetches possible with the approach.

the prefetcher is accurate, it may be that the other page table entries within the same cache line may be desired in the near future, so bringing them into the caches may be beneficial. On the other hand, prefetches are inherently speculative operations – overly aggressive prefetching may needlessly pollute the processor caches, adversely impacting performance. Ultimately, the ”right” strategy is workload-dependent. In most real-world implementations, TLB prefetches do *not* bring page table entries into the caches. Instead, the assumption is that the prefetch engine in the TLB will do so well in identifying future translations, that they do not need the additional prefetching boost brought about by caching a line of page table entries in the L1 caches.

③ Interaction with status bits: Since a prefetch is a speculative operation, rather than a true demand from the CPU, they should ideally not influence the status bits of the prefetch targets. More specifically, page table walkers that inject prefetches should not be permitted to set the access bits in the prefetched translations (note that the dirty bit cannot be set because prefetches do not write any page). However, this can complicate the design of MMUs in architectures like x86-64, where the memory consistency model dictates that only translations whose access bits are set are permitted to be TLB-resident [54, 73]. Consequently, x86-64 vendors are typically willing to expend more hardware resources to create particularly accurate TLB prefetchers because all prefetches set access bits. This means that if a translation is prefetched but ultimately not used, its access bit remains set, and means that the OS’s replacement policy decisions, which rest on identifying hot pages via the access bits, are sub-optimal. Trying to ensure that

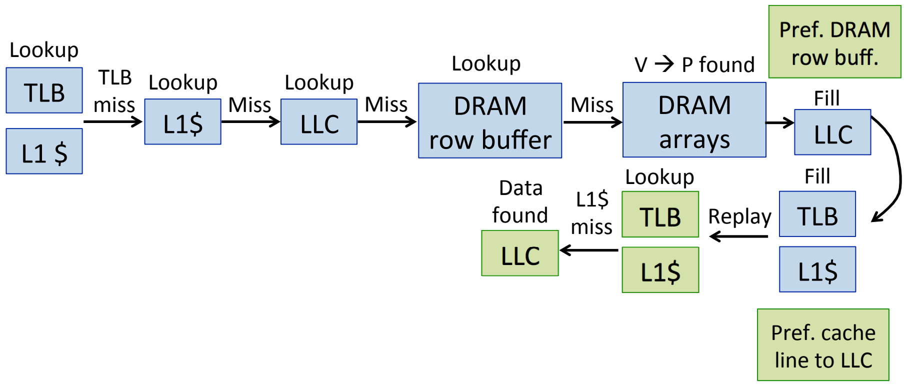


Figure 27: Translation-triggered prefetching is used to accelerate the instruction replay. The basic idea is that hardware support is added to the memory controller so that it can deduce the memory address that the replay will access after handling the TLB miss. This allows replay accesses to main memory to be converted to last-level cache (LLC) hits.

all prefetches are ultimately used is one way of avoiding this situation.

③ Interaction with page faults: Prefetches may prompt page faults when the prefetch target has an unmapped physical frame. The question is whether this kind of prefetch should invoke the OS and its page fault handler. Typically, prefetches are non-faulting – this means that if a prefetch target is found to have an unmapped physical page, the request is aborted without invoking the OS [20, 21, 23]. The main reason for this is that a prefetch is speculative and may be incorrect. Mistakenly invoking a page fault would be too high a performance penalty to pay. Rather than taking this risk, all faulting prefetches are usually aborted.

10 Replay Prefetching

Section 8 (and Figure 20 in particular) discussed the relationship between the hotness of a page table entry and the cache lines in the physical frame pointed to by that page table entry. The key observation is that a page table walk that needs to access deeper levels of the memory hierarchy like the last-level cache (LLC) or DRAM are typically followed by replays that also need an LLC or DRAM probe. Recent research studies have exploited this observation to prefetch the data ultimately required by the replay, so as to increase overall performance. Figure 27 shows how this *translation-triggered prefetching* approach works [18].

The high-level idea behind translation-triggered prefetching can be understood by comparing Figure 20 to Figure 27. The situation being tackled is one where a DRAM access is needed to find the desired the page table entry. This is followed by the replayed instruction also requiring DRAM access. The goal is to convert DRAM accesses for replays to LLC hits (or row buffer hits within DRAM). The original study adds hardware to the memory controller to identify DRAM page table accesses [18].

Combinational logic identifies the physical page stored in this translation. This is combined with information about the desired cache line – sent to the memory controller by the page table walker – to identify the post-page table memory address, before the memory replay. Figure 27 shows that this allows two optimizations. First, the 4-16KB row holding the data needed by the replay is prefetched from the DRAM array into the row buffer. Second, the cache line storing this data is prefetched into the LLC.

Ideally, translation-triggered prefetching allows replays to see LLC hits, eliminating: ① on-chip network traversal from the LLC to the memory controller; ② memory controller queueing delays; ③ DRAM row buffer lookup; ④ time to write back the contents of the DRAM row buffer; ⑤ time to activate the desired row in the DRAM array; ⑥ time to READ/WRITE the desired data from the DRAM array; and ⑦ cache fill activities. This can translate to a savings of 100-150+ cycles. Naturally, it is possible for the LLC line to be evicted before use, but studies suggest that this happens only rarely [18]. Furthermore, even when this occurs, DRAM row buffer hits may still occur, eliminating ④-⑦ .

Ultimately, translation-triggered prefetching leverages the fact that the prefetch can be overlapped with a window of time during which the translation is being filled into the caches and TLB (post-page table walk), and the replay is proceeding until the LLC lookup. This has two implications. First, this window of time is is usually long enough to perform row buffer and LLC prefetching. For example, Intel Haswell and Skylake processors usually take 120+ cycles for these events [18]. In contrast, prefetching from the DRAM array into the row buffer takes 60-100 cycles, while prefetching to the LLC adds another 20-30+ cycles. Second, even when prefetching time exceeds the length of this window, prefetching can be partially overlapped, boosting performance. In practice, studies find that in scenarios with partial overlap, LLC hits occur less often but DRAM row buffer hits remain prevalent [18].

To operate correctly, translation-triggered prefetching requires hardware enhancements to page table walkers and the on-chip memory controller (although no changes are needed to the OS or application). Figure 28 illustrates the desired changes. Consider a situation where the CPU makes a request for virtual address 0x01002. Assuming 4KB pages (although the basic mechanism also works with superpages), this virtual address can be separated into a virtual page number 0x1, and a page offset of 0x002. Suppose further, that a TLB lookup for virtual page number 0x1 results in a TLB miss, prompting a page table walk. A PTW walks the page table and ultimately discovers the desired page table entry. Suppose that during this walk, the page table entry had to be read from DRAM via the memory controller (MC). This enables the MC to trigger a prefetch for the data that the replay will ultimately need, which is likely to also be in DRAM. The question is, how can the prefetch target be constructed? Since prefetching is performed in the unit of cache lines, what is needed is the cache line number within the physical frame pointed to by the translation. In our example, we assume that virtual page 1 maps to physical page 3. This means that we are seeking cache line number 0x000 within physical frame number 0x3. The MC can extract the physical frame number easily since it has to read the page table entry for the page table walk anyway. What is now needed is to get access to the desired cache line number within this physical frame so that it can be concatenated with the frame number (0x3) extracted by the MC. In order to supply the MC this line number, the original translation-triggered

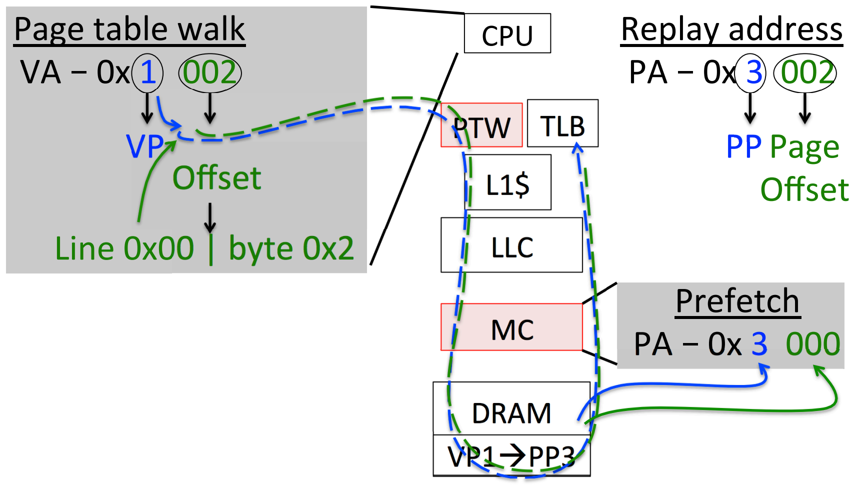


Figure 28: Translation-triggered prefetching requires changes to the page table walker (PTW) and the memory controller (MC). The PTW is adjusted to convey the desired cache line within the desired virtual page (which corresponds to the same cache line number within the physical frame). The MC concatenates this information with the physical frame number extracted from the page table entry to generate a prefetch target.

prefetching study proposes modification of the PTW to inject not just the virtual page number to the memory hierarchy for the page table walk, but also the line number 0x000. This addition of a few bits in the page table walk message permits the MC To perform a prefetch for the cache line that holds the data that the replay ultimately needs, *0x3002*.

Beyond the fact that translation-triggered prefetching is simple to implement, it differs from classical prefetching in another important way. Classical cache prefetching predicts future memory references and hence speculates on their target addresses, possibly incorrectly. Incorrect speculations waste energy, degrade performance, and needlessly pollute caches. Translation-triggered prefetching suffers from none of these problems because it is *non-speculative in terms of the target address*. The replay's memory address is *always* calculated correctly by the memory controller

11 Translation Contiguity

A particularly compelling approach to reducing TLB misses is to encourage the OS to allocate adjacent virtual pages to adjacent physical pages, a general technique we refer to as *translation contiguity*. 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. This is, in fact, the premise for large pages, which map hundreds of consecutive physical pages to hundreds of consecutive virtual pages. However, OSes often encounter several obstacles to generating the vast swaths of translation contiguity needed for large pages, particularly in real-world systems with long uptimes where

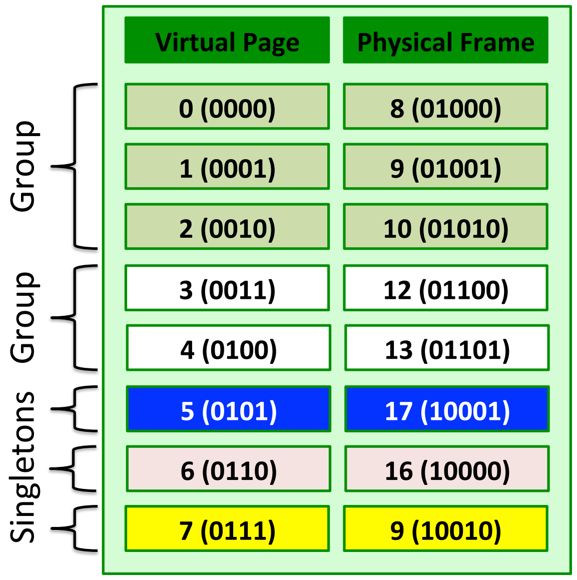


Figure 29: This page table shows examples of translation contiguity. Virtual pages 0-2 are contiguous because their physical frames are 8-10. Furthermore, the translations for virtual pages 3-4 are also contiguous. Both these *groups* of translation contiguity are intermediate, because they do not have the vast amounts of contiguity (i.e., 512+ contiguous 4KB translations) to create 2MB pages, but still represent 10s-100s of contiguous translations. In contrast, the translations for virtual pages 5, 6, and 7 are *singletons* because they are not contiguous in physical frames.

multiple workloads fragment the memory of a computer system [52, 57, 64, 68]. Therefore, we consider alternate ways to exploit more limited forms of translation contiguity.

OS-generated translation contiguity: While large pages require explicit OS intervention to ensure ample translation contiguity (ranging from 512 to thousands of contiguous pages), it is also possible for OSes to generate *intermediate* amounts of contiguity (in the range of tens to a few hundreds of pages). For example, Figure 29 shows a page table in which entries for virtual pages 0-2 use a spatially sequential group of physical frames. In other words, the translations for virtual pages 0-2 are indeed contiguous. Similarly, the translations for virtual pages 3-4 are also contiguous. While out of the scope of this appendix, several studies show that OSes allocate memory in this way – although they do not have to – because of their internal buddy allocators and memory compaction strategies, which create contiguous bundles of used and free space [52, 63, 64, 66, 68]. This intermediate contiguity is not big enough to form 2MB/1GB superpages, but may be exploited in hardware to improve TLB hit rates. The same cannot, however, be accomplished with the *singleton* page table entries (translations for virtual pages 5, 6, and 7 in Figure 29).

Coalescing: Figure 30 shows one way of exploiting translation contiguity by using a coalesced TLB [66] (these approaches have been implemented in commercial prod-

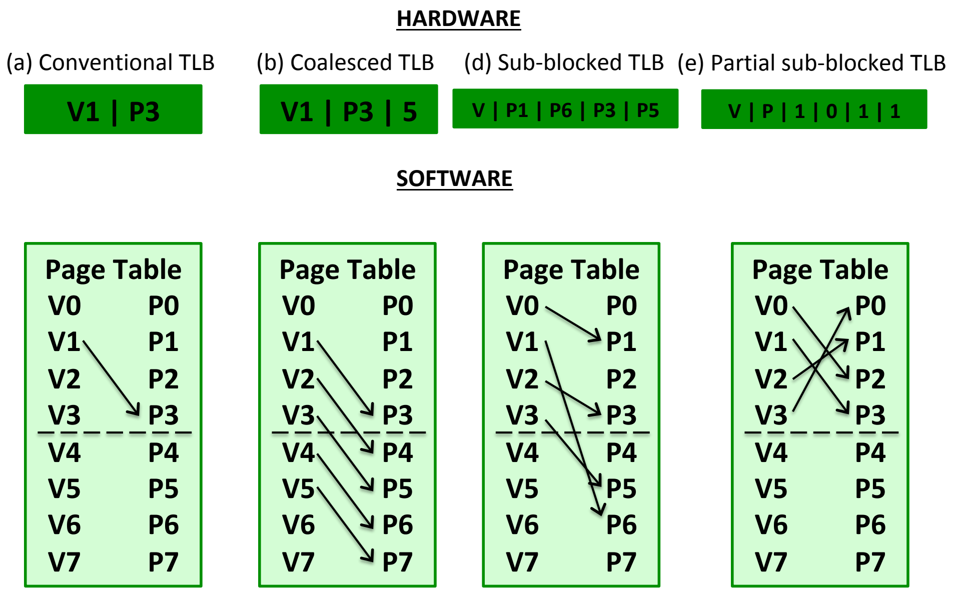


Figure 30: Combinations of page tables and the different techniques proposed to take advantage of spatial contiguity and clustering. We compare a TLB entry for the conventional case, a coalesced case [29, 66], complete sub-blocked case [85], and partial sub-blocked case [85].

ucts like AMD's Zen architecture). Consider Figure 30(a) and 30(b), which contrast the operation of a conventional TLB with a coalesced TLB. A conventional TLB entry corresponds to a single translation (virtual page V1 and physical page P3) but a coalesced TLB entry maps a group of contiguous, spatially-local translations (page table entries for virtual pages 1-5). Any arbitrary set of page table entries can be accommodated (e.g., five entries in Figure 30(b)) by recording only the base page table entry and the number of coalesced page table entries adjacent to the base. On look-up, the offset between the base virtual address stored in the tag is used to calculate the offset from the base physical page. Furthermore, no alignment restrictions are imposed. Hence, the base virtual page can be mapped to *any* physical frame. This is in contrast to, for example, 2MB or 1GB superpages, where virtual page and physical frame at the start of a superpage must be aligned to 2MB or 1GB address boundaries respectively. The only real requirement for coalesced TLBs to be effective is that each entry should be designed such that it only minimally requires more area than a conventional TLB entry.

Complete sub-blocking: The main benefit of coalesed TLBs is that they can harness intermediate contiguity that may still be present in system memory, even in situations where system fragmentation makes it difficult for the OS to form superpages. However, there may be situations where even intermediate amounts of spatial contiguity may be hard to generate. Sub-blocked TLBs can be helpful in such situations where contiguous spatial locality is scarcer [85]. Figure 30(c) shows that complete sub-blocked approaches are used for clusters of page table entries with contiguous virtual page numbers. For a sub-block factor N, this approach looks for B aligned virtual pages; this means that all virtual address bits apart from the bottom log2(B) bits are the same. Complete sub-blocked TLBs store all the page table entries corresponding to this group in one complete hardware entry. Figure 30(c) shows an example of this where virtual pages 0-3 all are aligned for a sub-block factor of 4. This means that their page table entries can be stored in one hardware TLB entry, as long as it maintains a field for each physical frame (e.g., P1, P6, P3, and P5). Unfortunately, the ability of complete sub-blocking to store any set of physical frames requires expensive hardware (multiple TLB entry fields for physical frames). Furthermore, unlike coalescing, which accommodates any length of contiguous page table entries, complete sub-blocking stores a contiguity count equal to the sub-block factor.

Partial sub-blocking: Talluri and Hill propose partial sub-blocking as a lower-overhead alternative to complete sub-blocking [85]. Figure 30 shows that partial sub-blocking searches for page table entries with an aligned group of virtual pages and an aligned group of physical pages. All page table entries that have virtual page numbers and physical frame numbers with the same offset from the start of the aligned package are coalescable into a single entry (i.e., in our example, page table entries for VPNs 0, 2, and 3). This approach permits "holes" in a group of page table entries when the physical page offset within the aligned packet is different from the virtual page offset. In our example, this corresponds to the case of virtual page 1. Partial sub-blocking achieves high reach using simpler hardware than complete sub-blocking by imposing alignment and offset restrictions on physical frames. Figure 30 shows that each entry maintains a bit vector to record the presence of the physical pages rather than all the bits of the physical frame number. Intuitively, partial sub-blocking goes beyond coalescing by exploiting contiguous spatial locality and situations where a cluster of spatially adjacent pages maps to another adjacent set of physical frames, but with any permutation of virtual to physical mappings within the cluster.

Integrating contiguity optimizations in the MMU: While the contiguity optimizations we have presented are understood easily conceptually, their integration into each CPU's MMU logic has several design variants that impact performance in many different ways. Covering all these variants in detail is beyond the scope of this appendix. However, we focus on the design of coalesced TLBs, as presented in the original research paper and believed to be similar to coalescing in AMD's Zen architecture [29, 66].

Lookup: Figure 31 illustrates the process of a TLB lookup. On the left, we show a page table, where the translations for V0-V3 are contiguous. On the right, we contrast the lookup of a traditional set-associative TLB with that of a coalesced TLB. We focus on a situation where a CPU probes the TLB for the translation for V1.

Conventional TLBs use the lower-order bits of the virtual page number to identify the TLB set to look up. In our example with a two-set TLB, this means that the lower-order bit becomes the index. This means that for the translation for V1, this must reside in set 1. However, coalesced TLBs operate differently since coalesced translations for adjacent virtual pages must map to the same set. Consider, for example, a coalesced

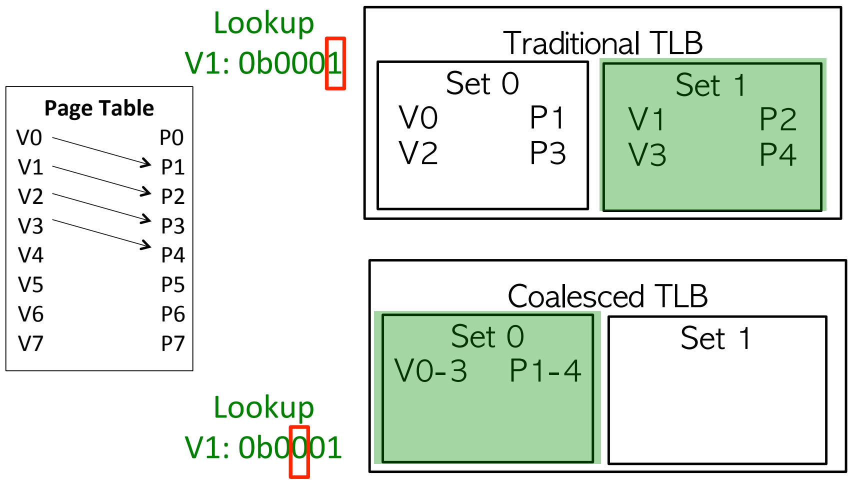


Figure 31: Coalesced TLB lookups for a set-associative TLB require index bits from more significant parts of the virtual address, but otherwise remaint the same as conventional TLB lookup.

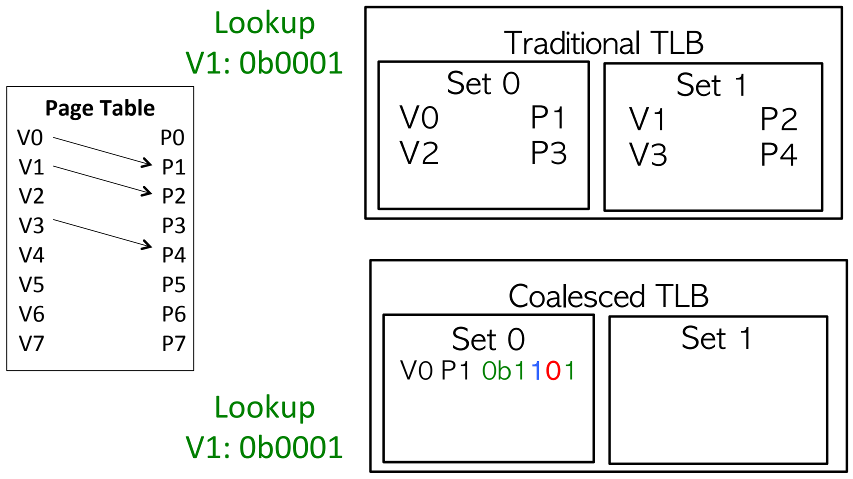


Figure 32: Bitmaps are used to encode the contiguous translations in a coalesced packet. The use of bitmaps also permits ”hotes” in contiguity among translations.

TLB that supports a maximum of four coalesced translations. In this case, shown in Figure 31, the translations for V0-V3 map to the same set, instead of striding across the two sets. To enable this, instead of using the lowest order bit as the index, coalesced TLBs use bit number 2 as the set index. Once the set index is known, the translations in the set are scanned. Figure 32 shows how this proceeds. Although only one translation is shown, all translations in the set are compared. Each translation maintains a bit vector that records which of the four possible coalescable translations are actually contiguous and are hence coalesced. Therefore, a lookup requires both the conventional

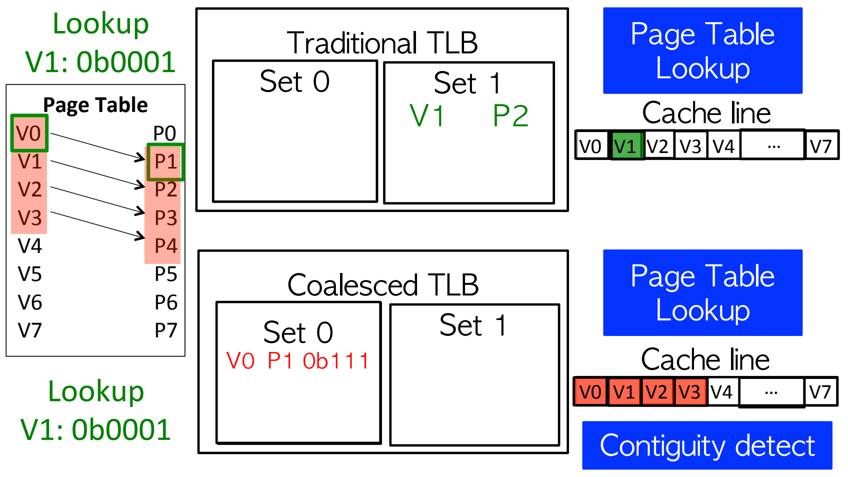


Figure 33: Coalescing is performed by combinational logic that is activated on page table walks. At that point, the logic inspects the cache line that stores the desired page table entry, and coalesces all contiguous translations within that cache line.

tag match using the virtual page bits, and also a lookup of the relevant bit in the bitmap. For a lookup of V1, this means that the bit shown in blue is looked up. If this is set, it means that the requested translation exists in this bundle. The physical page frame can then be calculated by adding the relevant offset to the physical page frame number stored (P1 + 1 in our example).

Figure 32 also shows that it is possible for multiple translations with the same virtual page tag to co-exist in the set. For example, if V0-V1 were contiguous individually (by pointing to P1-P3) and V2-V3 were also contiguous (by pointing to, for example, P5-P6), set 0 would contain two separate entries for each of these bundles of coalesced translations. The lookup distinguishes between the two coalesced bundles by looking up the relevant bit in the bitmap field. In the first entry, the bitmap would be 0b1100, while the second entry would be 0b0011. The bitmap acts as a way to distinguish among equivalent virtual page tags.

Figure 32 shows that the bitmap also enables an optimization that goes beyond strict contiguity in the page table. Specifically, the translation for V2 does not map to P3. However, because V3 maps to P4 – and this follows the contiguity pattern of the V0-V1 translations – it is possible to use a single entry to store all contiguous translations for V0, V1, and V3, provided that the bitmap entry for V2 (shown in red) is 0.

Finally, this coalesced TLB design requires the set indexing scheme to be statically changed. As long as the page tables demonstrate sufficient contiguity, this is a good design choice. However, if page tables see little contiguity, such a change in the set indexing scheme can potentially increase TLB misses. Therefore, past work considers options such as using parts of the TLB real-estate for coalescing, while leaving the rest as traditional TLB hardware [64]. Other approaches consider coalescing only at the L2 TLB, leaving L1 TLBs unchanged [66].

Fill: We now describe the mechanism used to identify contiguous translations. To ensure that TLB lookup latencies remains unchanged, coalescing is usually performed on the TLB fill path rather than the lookup [66]. Figure 33 shows TLB fill and coalescing. On lookup, suppose that there is a TLB miss for V1. This triggers a TLB miss on both traditional and coalesced TLBs. In both cases, the page table walker requests portions of the page table in the unit of caches lines. Consider an x86-64 system, where eight page table entries reside in a 64-byte cache line. With coalesced TLBs, combinational logic is added on the fill path to detect contiguous translations within the cache line. Once the contiguous translations are detected, they are coalesced and filled into the coalesced TLB. In practice, this can bound the number of translations that are coalescible to the number of page table entries within a cache line. To go beyond this, the page table walker could opt to read multiple adjacent cache lines storing page table entries.

In summary, the benefits of coalesced TLBs (and other approaches like sub-blocking) are twofold. They increase effective TLB capacity by caching information about multiple translations in a single entry, without greatly increasing the size of that entry. More subtly, TLB coalescing also performs a form of prefetching. For example, Figure 33, the coalesced TLB bundles translations for V0, V2, and V3 and fills them into the TLB, even though the original request for V1 only. In the near future, accesses to V0, V2, and V3, which would otherwise have been misses on conventional TLBs, become hits. This form of prefetching also provides performance benefits.

Q&A 7

Q7. In Section 5, we discussed the difficulties of designing a single set-associative L1 TLB structure that can concurrently cache translations for multiple page sizes. The key challenge was the fact that different page sizes have different page offset and virtual page bit-widths. At a high level, all the techniques we discussed address this issue by moving away from the idea that all translations, regardless of page size, should use the same set of virtual page bits as the index into the TLB. However, in this exercise, we ask whether there is a way we can design efficient set-associative L1 TLBs that can concurrently cache translations from multiple page sizes, assuming the following observation – superpages for consecutive virtual pages are often (though they do not have to be) allocated contiguously in physical frame numbers too.

A7. The idea of building set-associative L1 TLBs that can concurrently cache translations for multiple page sizes by leveraging superpage contiguity has been explored in recent work [29]. These TLBs are known as MIX TLBs. In order to understand their operation, consider the address space shown in Figure 34. We show virtual and physical addresses, with small page *A* and superpages *B-C*. Note that while we assume 64-bit systems, our examples show 32-bit addresses to save space. These addresses are shown in 4KB frame numbers (full addresses can be constructed by appending 0x000). Therefore, superpage *B* is located at virtual address 0x00400000 and physical address 0x00000000. Superpages *B* and *C* have 512 constituent 4KB frames, indicated by *B0-511* and *C0-511*.

Figure 35 illustrates the lookup and fill operation of MIX TLBs and contrasts them to split TLB lookup and fill. In ①, *B* is looked up but is determined to be absent as both split and MIX TLBs maintain only *A*. The hardware page table walker is invoked

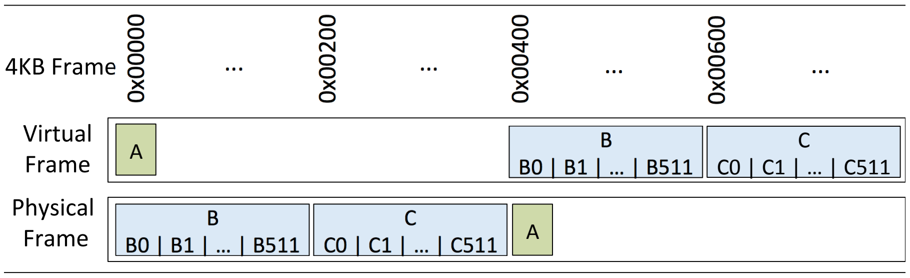


Figure 34: Example address space in an x86-64 architecture. We show 4KB frame numbers in hexadecimal. For example, translation *B* is for a 2MB page, made up of 4KB frame numbers *B0-B511*. 2MB translations *B-C* are contiguous.

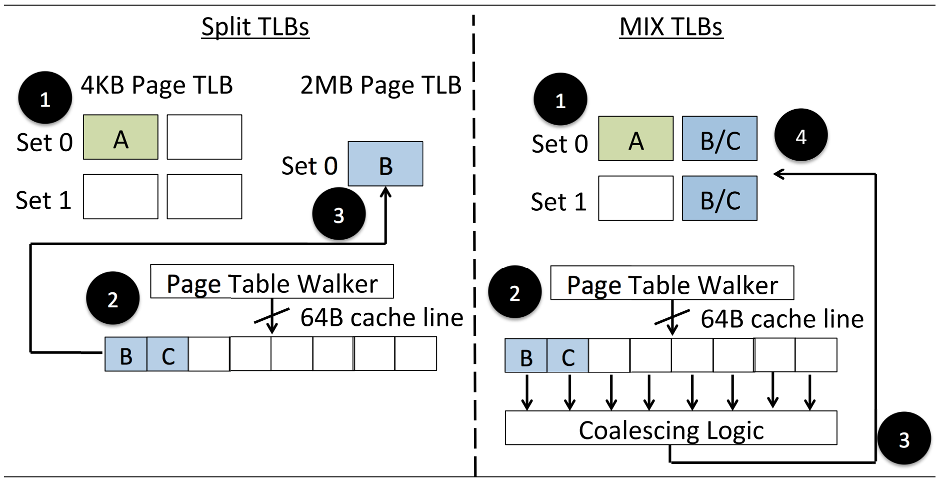


Figure 35: Conventional split TLBs verus MIX TLB lookup for superpage *B*.

in ② . The page table walker reads the page table in units of caches lines and so, eight translations (including *B* and *C*) are read in the cache line. Split TLBs then fill B into the superpage TLB ③ . Unfortunately, there remains no room for *C* despite 3 unused small page TLB entries.

In contrast, MIX TLBs cache translations of all page sizes. After a miss ① and a page table walk ②, *B* is filled in the correct set. Since MIX TLBs use index bits for small pages (in our 2-set TLB example, this corresponds to bit 12) on all translations, the index bits are picked from the superpage's page offset. Thus, superpages do not uniquely map to either set. Instead, *B* is *mirrored* or filled in both TLB sets.

Mirroring presents a problem, however. Whereas split TLBs maintain one copy of a superpage translation, MIX TLBs maintain several mirror copies. This reduces effective TLB capacity. However, MIX TLBs counteract this problem with the following observation – OSes frequently (though they don't have to) allocate superpages in adjacent virtual and physical addresses. For example, Figure 34 shows that *B* and

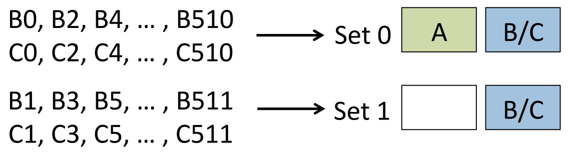


Figure 36: Though superpages *B* and *C* are maintained by multiple sets but on lookup, MIX TLB lookups probe the set corresponding to the 4KB region within the superpage that the request is to.

C are contiguous, not just in terms of their constituent 4KB frames (e.g., *B0-511* and *C0-511*) but also in terms of the full superpages themselves. MIX TLBs exploit this contiguity by using the following approach – when page table walkers read a cache line of translations ②, adjacent translations in the cache line are scanned to detect contiguous superpages. Just like standard coalesced TLBs, combinational coalescing logic on the TLB fill path can be used for this in step ③. In our example, *B* and *C* are contiguous and are hence coalesced and mirrored. Coalescing counteracts mirroring. If there are as many contiguous superpages as there are mirror copies (or close to as many), MIX TLBs coalesce them to achieve a net capacity corresponding to superpage reach, despite mirroring.

Beyond this initial description of MIX TLBs, Figure 36 shows that MIX TLB lookup remains largely unchanged from that of standard coalesced TLBs. While coalesced mirrors of superpages reside in multiple sets, lookups only probe one TLB set. In other words, virtual address bit 12 in our example determines whether the even- or odd-numbered 4KB regions within a superpage are accessed. Therefore accesses to *B0*, *B2*, etc., and *C0*, *C2*, etc., are routed to set 0. Naturally, this overview presents several important questions. We briefly address them below:

Why do MIX TLBs use the index bits corresponding to the small pages rather than large pages? The advantage of the latter is that each superpage maps uniquely to a set, eliminating the need for mirrors (e.g., *B* maps to set 0, and *C* maps to set 1). Unfortunately, this causes a different problem. Now, spatially adjacent small pages map to the same set. For example, if we use the index bits corresponding to a 2MB superpage (i.e., in our 2-set TLB example, bit 21), groups of 512 adjacent 4KB virtual pages map to the same set. Since real-world programs exhibit spatial locality, this elevates TLB conflicts unless associativity exceeds 512, which is far higher the 4-8 way associativity used for modern TLBs. One could envision coalescing small pages if the OS does allocate them contiguously in virtual and physical addresses. However, as we have discussed in this section, contiguity in small pages tend to range in the 10s to a few hundred at best. Prior work finds that using superpage index bits increases TLB misses by 4-8$\times$ on average, compared to using small page index bits [29].

How many mirrors can a superpage produce and how much contiguity is needed to offset effective TLB capacity lost due to mirrors? Consider a MIX TLB with M sets, and a superpage with N 4KB regions (N is 512 and 262144 for 2MB and 1GB superpages). Practical commercial L1 and L2 TLBs tend to have 16-128 sets. Therefore, today's systems have N > M; this means that a superpage has a mirror per set (or N mirrors). However, if future systems use N < M, M mirrors would be present instead. Ultimately, good MIX TLB hit rates rely on superpage contiguity. If the number of contiguous superpages is equal (or sufficiently near) the mirror count, MIX TLBs enjoy many hits. On modern 16-128 set TLBs, we desire (close to) 16-128 contiguous superpages. Prior work shows that real systems often do see this much superpage contiguity [29]

While this exercise presents many of the initial details of MIX TLBs, important design questions and microarchitectural details remain. We point interested readers to the original paper for more details [29] .

---

12 Conclusion

The goal of this appendix chapter was to provide an introduction to the complexity of the modern address translation stack. While several important concepts were covered, many other address translation techniques and optimizations remain to be explored in more detail. We point interested readers to studies that cover the following non-exhaustive list of topics:

- **Shared address translation structures:** While modern processors typically use per-core MMUs (with private per-core TLBs, MMU caches, PTWs, etc.), there is growing evidence that address translation hardware may be effectively shared among multiple system cores to improve performance [14, 21, 83].
- **TLB speculation:** Several recent studies [11, 68] propose techniques to further boost effective TLB hit rates by relying on speculative techniques that can ”guess” the physical frame number of a translation that misses in the TLBs. This permits execution of the CPU to continue in parallel with the page table walk.
- **Part-of-memory TLB optimizations:** While TLBs have traditionally been tightly coupled with CPUs, there is increasing evidence that there may also be value in building large TLBs as part of DRAM, particularly with the advent of high-bandwidth DRAM [75].
- **Direct segment, range optimizations, superpages:** An alternate set of techniques seek to dramatically reduce TLB requirements by relying on the OS (and possibly programmer hints) to generate vast amounts of contiguous translations that can reduce the number of necessary TLB entries [12, 36, 48]. Additionally, the research community is continuing to explore ways to improve superpage management [39, 52, 60, 68].
- **Software optimizations:** There has been recent work on managing page migration and allocation more efficiently, particularly in the context of heterogeneous memories and accelerators. We point readers to relevant papers [1, 2, 34, 38].

- **Energy-efficient address translation:** As the hardware resources dedicated to fast address translation continues to increase, its energy consumption is becoming a bigger problem. Recent studies explore ways in which to reduce address translation energy usage by better TLB design and management [13, 14, 49].
- **Translation coherence:** As the number of computation units with their own TLBs and MMU caches increases, the cost of maintaining coherence among them becomes higher. A range of new translation coherence mechanisms are being proposed in response [5, 9, 51, 58, 65, 73, 74, 89, 93].
- **Virtualization:** Modern processors maintain hardware support to accelerate the cost of TLB misses in virtualized environments, which can be a factor of 4$\times$ worse than native execution. We point readers to these papers to learn more [3, 15, 29, 37, 68, 93].
- **Accelerators:** With the advent of hardware accelerators, there is a large body of emerging work on extending address translation support for GPUs [7, 8, 29, 39, 69, 70, 72, 81, 87, 94], near-memory accelerators [71], and more [33].
- **Memory protection:** While most of this appendix focused on virtual-to-physical page translations, the virtual memory stack also offers memory protection. This is an area which has seen a much recent work of late. We point interested to several emerging research directions on page-granularity protection, object-granularity protection, capabilities, and much more [25, 32, 46, 88, 90–92].

References

[1] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[2] Neha Agarwal and Thomas Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *International Symposium on Computer Architecture*, 2012.

[4] Hana Alam, Tianhao Zheng, Mattan Erez, and Yoav Etsion. Do It Yourself Virtual Memory Translation. *International Symposium on Computer Architecture*, 2017.

[5] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. *USENIX Annual Technical Conference*, 2017.

[6] Eishi Arima and Hiroshi Nakamura. Page Table Walk Aware Cache Management for Efficient Big Data Processing. *Workshop on Big data benchmarks, Performance Optimization, and Emerging hardware (BPOE)*, 2017.

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. *International Symposium on Microarchitecture*, 2017.

[8] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher Rossbach, and Onur Mutlu. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[9] Amro Award, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel Loh. Avoiding TLB Shootdowns through Self-invalidating TLB Entries. *International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *International Symposium on Computer Architecture*, 2010.

[11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *International Symposium on Computer Architecture*, 2011.

[12] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark Hill, and Mike Swift. Efficient Virtual Memory for Big Memory Servers. *International Symposium on Computer Architecture*, 2013.

[13] Arkaprava Basu, Mark Hill, and Michael Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. *International Symposium on Computer Architecture*, 2012.

[14] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. Scalable Distributed Shared Last-Level TLBs Using Low-Latency Interconnects. *International Symposium on Microarchitecure*, 2018.

[15] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[16] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *International Symposium on Microarchitecture*, 2013.

[17] Abhishek Bhattacharjee. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE MICRO*, 2017.

[18] Abhishek Bhattacharjee. Translation-triggered prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[19] Abhishek Bhattacharjee. Breaking the Address Translation Wall by Accelerating Memory Replays. *IEEE Micro Top Picks*, 2018.

[20] Abhishek Bhattacharjee and Daniel Lustig. Architectural and Operating System Support for Virtual Memory. *Synthesis Lectures*, 2017.

[21] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. *International Symposium on High Performance Computer Architecture*, 2011.

[22] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[23] Abhishek Bhattacharjee and Margaret Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. *ASPLOS*, 2010.

[24] Edouard Bugnion, Jennifer Anderson, Todd Mowry, Mendel Rosenblum, and Monica Lam. Compiler-Directed Page Coloring for Multiprocessors. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[25] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N. M. Watson. CHERI-JNI: Sinking the Java security model into the C. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[26] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proc. of the 25th Intl. Symp. on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.

[27] Austin Clements, Frans Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. *ASPLOS*, 2012.

[28] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. Supporting Address Translation for Accelerator-Centric Architectures. *HPCA*, 2017.

[29] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[30] Peter Denning. Virtual Memory. *Computing Surveys*, 2(3), 1970.

[31] Peter Denning and Stuart Schwartz. Properties of the Working-Set Model. *Communications of the ACM*, 15(3), 1972.

[32] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan Smith, Thomas Knight, Benjamin Pierce, and Andre DeHon. Architectural Support for Software-Defined Metadata Processings. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[33] Xiaowan Dong, Sandhya Dwarkadas, and Alan Cox. Shared Address Translation Revisited. *Eurosys*, 2016.

[34] Yu Du, Miao Zhou, Bruce Childers, Daniel Mose, and Rami Melhem. Supporting Superpages in Non-Contiguous Physical Memory. *International Symposium on High Performance Computer Architecture*, 2014.

[35] Babak Falsafi and Thomas Wenisch. A Primer on Hardware Prefetching. *Synthesis Lectures in Computer Architecture*, 2014.

[36] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *International Symposium on Microarchitecture*, 2014.

[37] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *International Symposium on Computer Architecture*, 2016.

[38] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *USENIX Annual Technical Conference*, 2014.

[39] Swapnil Haria, Mark Hill, and Michael Swift. Devirtualizing Memory in Heterogeneous Systems. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[40] Intel. Haswell microarchitecture. *www.7-cpu.com/cpu/Haswell.html*.

[41] Intel. Skylake microarchitecture. *www.7-cpu.com/cpu/Skylake.html*.

[42] Bruce Jacob and Trevor Mudge. A Look at Several Memory Management Units, TLB-Refull Mechanisms, and Page Table Organizations. *ASPLOS*, 1998.

[43] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon Steely, and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches Temporal Locality Aware (TLA) Cache Management Policies. *International Symposium on Microarchitecture*, 2010.

[44] Aamer Jaleel and Bruce Jacob. In-line Interrupt Handling for Sofware-Managed TLBs. *International Conference on Computer Design*, 2001.

[45] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon Steely, and Joel Emer. High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. *International Symposium on High Performance Computer Architecture*, 2015.

[46] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, , and A. Theodore Markettos. Efficient Tagged Memory. *International Conference on Computer Design*, 2017.

[47] Gokul Kandiraju and Anand Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *International Symposium on Computer Architecture*, 2002.

[48] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrian Cristal, Mark Hill, Kathryn McKinley, Mario Nemirovsky, Michael Swift, and Osman Unsal. Redundant Memory Mappings for Fast Access to Large Memories. *International Symposium on Computer Architecture*, 2015.

[49] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Energy-efficient address translation. In *International Symposium on High Performance Computer Architecture*, 2016.

[50] Richard Kessler and Mark Hill. Page Placement Algorithms for Large Real-Indexed Caches. *Transactions on Computer Systems*, 1992.

[51] Mohan Kumar, Steffen Maass, Sanidhya Kashyap, Jan Vesely, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. LATR: Lazy Translation Coherence. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[52] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *USENIX Conference on Operating Systems Design and Implementation*, 2016.

[53] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs. *ACM Transactions on Architecture and Code Optimization*, 10(1):2:1–2:38, April 2013.

[54] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-os interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[55] Daniel Lustig, Geeth Sethi, Abhishek Bhattacharjee, and Margaret Martonosi. Transistency Models: Memory Ordering at the Hardware-OS Interface. *IEEE Micro Top Picks*, 2017.

[56] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy John. CSALT: Context Switch Aware Large TLB. *International Symposium on Microarchitecture*, 2017.

[57] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *Operating Systems Review*, 2002.

[58] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *International Conference on Parallel Architecture and Compilation*, 2015.

[59] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. Shoot4u: Using vmm assists to optimize tlb operations on preempted vcpus. In *International Conference on Virtual Execution Environments*, 2016.

[60] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making Huge Pages Actually Useful. *ASPLOS*, 2018.

[61] Myrto Papadopoulou, Xin Tong, Andrew Seznec, and Andreas Moshovos. Prediction-Based Superpage-Friendly TLB Designs. *HPCA*, 2014.

[62] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using Superpages to Improve VIPT Caches. *International Symposium on Computer Architecture*, 2018.

[63] Chang Hyun Park, Taekyung Heo, Jungi Jeong, , and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. *International Symposium on Computer Architecture*, 2017.

[64] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *International Symposium on High Performance Computer Architecture*, 2014.

[65] Binh Pham, Derek Hower, Abhishek Bhattacharjee, and Trey Cain. TLB Shootdown Mitigation for Low-Power Many-Core Servers with L1 Virtual Caches. *Computer Architecture Letters*, 2018.

[66] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *International Symposium on Microarchitecture*, 2012.

[67] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. Using TLB Speculation to Overcome Page Splintering in Virtual Machines. Rutgers Technical Report DCS-TR-713, Department of Computer Science, Rutgers University, Pistcataway, NJ, 2015.

[68] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *International Symposium on Microarchitecture*, 2015.

[69] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[70] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Address Translation for Throughput Oriented Accelerators. *IEEE Micro Top Picks*, 2015.

[71] Javier Picorel, Djordje Jevdjiv, and Babak Falsafi. Near-Memory Address Translation. *International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[72] Jason Power, Mark Hill, and David Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. *International Symposium on High Performance Computer Architecture*, 2014.

[73] Bogdan Romanescu, Alvin Lebeck, and Daniel Sorin. Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[74] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *International Symposium on High-Performance Computer Architecture*, 2010.

[75] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, , and Lizy John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. *International Symposium on Computer Architecture*, 2017.

[76] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-Based TLB Preloading. *International Symposium on Computer Architecture*, 2000.

[77] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip Gibbons, Michael Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management. *International Symposium on Computer Architecture*, 2015.

[78] Andre Seznec. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers*, 2004.

[79] Andre Seznec. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers*, 53(7), 2004.

[80] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses Using Hardware and Software Page Placement. *International Conference on Supercomputing*, 1999.

[81] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. Scheduling Page Table Walks for Irregular GPU Applications. *International Symposium on Computer Architecture*, 2018.

[82] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[83] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. *International Symposium on Microarchitecture*, 2010.

[84] Madhusudan Talluri, Shin Kong, Mark Hill, and David Patterson. Tradeoffs in Supporting Two Page Sizes. *International Symposium on Computer Architecture*, 1992.

[85] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[86] Jan Vesely, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel Loh, Mark Oskin, and Steve Reinhardt. Generic System Calls for GPUs. *International Symposium on Computer Architecture*, 2018.

[87] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *International Symposium on Performance Analysis of Systems and Software*, 2016.

[88] Lluis Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-centric Memory Domains. *International Symposium on Computer Architecture*, 2014.

[89] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[90] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian Memory Protection. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[91] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. *International Symposium on Computer Architecture*, 2014.

[92] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alex Richardson, Simon W. Moore, and Robert N. M. Watson. CheriRTOS: A Capability Model for Embedded Devices. *International Conference on Computer Design*, 2018.

[93] Zi Yan, Jan Vesely, Guilherme Cox, and Abhishek Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. *International Symposium on Computer Architecture*, 2017.

[94] Hongil Yoon, Jason Lowe-Power, and Gurindar Sohi. Filtering Translation Bandwidth with Virtual Caching. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[95] Tianhao Zheng, Haishan Zhu, and Mattan Erez. SIPT: Speculatively Indexed, Physically Tagged Caches. *International Symposium on High Performance Computer Architecture*, 2018.