

## Paging: Faster Translations (TLBs)

When we want to make things fast, the OS needs some help. And help usually comes from one place: the hardware. Here, to speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or **TLB**. A TLB (a part of the chip's **MMU**, or **memory-management unit**) is simply a hardware **cache** of popular virtual-to-physical address translations; thus, a better name would have simply been an **address-translation cache**. On any memory reference, the hardware will look first in the TLB to see if the desired translation is held therein; if it is, the translation is performed (quickly, in the processor) *without* having to consult the page table (which has all the translations for a given process).

Thus, the hardware (and the OS) follows the approach as seen in Figure 13.1 when servicing a memory reference by a process to a given virtual address. Note that the left column tells us whether it is the hardware (hw) or the OS (os) that performs the given action; in some cases (depending on the system), it could be one or the other, and hence we see hw/os.

In the common case, we are hoping that most translations will be found in the TLB (a **TLB hit**) and thus the translation will be quite fast (all in hardware, and all on chip near the processing core). In the less common case, the translation won't be in the cache (a **TLB miss**), and the system will have to perform extra

```

hw      1 - Extract VPN from address
hw      2 - Use the VPN to index the TLB
hw      if (the translation is in the cache) // a TLB hit
hw          3a - get PFN from TLB
hw          3b - use it to form physical address
hw          3c - issue request for physical address to the memory system
hw      else (translation is NOT in the cache) // a TLB miss
hw/os   4a - lookup translation in the page table (in memory)
hw/os   4b - if it is valid
hw/os           install the translation into the TLB
hw/os           retry the instruction (hopefully now a TLB hit)
hw      4c - if not valid, trap (invalid memory access)
os           terminate process (clean up)

```

Figure 13.1: TLB Control Flow

work to first consult the page table in main memory, update the TLB, and try the instruction again.

### 13.1 Who Handles the Miss?

One question that we must answer: who handles a TLB miss? Two answers here: the hardware, or the software (OS). In the olden days, the hardware had complex instruction sets (sometimes called **CISC**, for complex-instruction set computing) and the people who built the hardware didn't much trust those sneaky OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly where the page tables are located in memory (via a **page-table base register**, for example), as well as their exact format; on a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction (as in steps 4a, 4b, and 4c above). An example of an "older" architecture that has **hardware-managed TLBs** is the Intel x86 architecture, which uses a fixed **multi-level page table** (described in future notes); the current page table is pointed to by the CR3 register [I09].

More modern architectures (e.g., the MIPS R10k [H93] or SPARC

v9 [WG00], both **RISC** or reduced-instruction set computers) have what are known as **software-managed TLBs**. On a TLB miss, the hardware simply raises a trap, which stops the processor from doing what it has been doing and instead jumps to a **trap handler**. The trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special “privileged” instructions to update the TLB, and then return from the trap, which allows the hardware to try the instruction again (this time resulting in a TLB hit).

Note that when you are running the TLB miss-handling code, the OS needs to be extra careful not to cause more TLB misses to occur, otherwise it will induce an infinite loop of TLB misses. Many solutions are possible; one could keep TLB miss handlers in physical memory (and thus they **unmapped** and not subject to address translation), or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself; these **wired** translations will always hit in the TLB.

The big advantage of the software-managed approach is *flexibility*: the OS can use any data structure it wants to implement the page table, without any change in the hardware. Another advantage is *simplicity*; the hardware is greatly simplified by not having to worry about these intricate details.

## 13.2 TLB Contents: What’s In There?

Let’s look at the contents of the hardware TLB in more detail. A typical TLB might have 32 or 64 entries and be what is called **fully associative**. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A typical TLB entry might look like this:

VPN | PFN | other bits

Note that both the VPN and PFN are present in each entry,

as a translation could end up in any of these locations; the hardware searches the entries in parallel to see if there is a match.

More interesting are the “other bits”, which are used for a variety of reasons. The TLB commonly has a **valid** bit, which says whether the entry has a valid translation or not. Another common set of bits are **protection** bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked *read and execute*, whereas heap pages might be marked *read and write*. There may also be a few other fields, including an **address-space identifier**, a **dirty bit**, and so forth; see Section 13.5 below for more information.

### 13.3 TLB Issue: Context Switches

With TLBs, a new issue arises when switching between processes (and hence address spaces). Specifically, the contents of the TLB contain virtual-to-physical translations that are only valid for the current running process; these translations are not meaningful for other processes. Thus, when switching to run another process, the hardware or OS or both must be careful.

To understand this better, let’s look at an example. When one process (P1) is running, it accesses the TLB with translations that are valid for it. Assume here that the 0th virtual page of process P1 might be mapped to physical frame 10. Another process may also be ready in the system (P2), and the OS might be context-switching between it and P1; assume the 0th virtual page of P2 is mapped to physical frame 17. If entries for both processes were in the TLB, it might look like this:

VPN	PFN	valid	prot
0	10	1	rwX
—	—	0	—
0	17	1	rwX
—	—	0	—

In the TLB above, we clearly have a problem: VPN 0 translates to either PFN 10 (P1) or PFN 17 (P2), but the hardware can’t distinguish which entry is meant for which process. Thus,

we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes. And thus we arrive at a crux:

#### THE CRUX: TLB CONTENTS ON CONTEXT SWITCHES

When context-switching between processes, the translations in the TLB for the last process are not meaningful to the about-to-be-run process. What should the hardware or OS do in order to solve this problem?

There are a number of possible solutions. One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this could be accomplished with an explicit (and privileged) hardware instruction, whereas in a hardware-managed TLB, the flush could be enacted when the page-table base register is changed (the OS must change the contents of the page-table base register on such a switch anyhow). In either case, the flush operation simply sets all valid bits to 0.

By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur a fair number of TLB misses as it touches data and code pages. If the OS is switching between processes frequently, this cost may be noticeable.

To overcome this cost, some systems add a little extra hardware support to enable sharing of the TLB across context switches. In particular, the hardware could provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a **process identifier (PID)**, but usually it has fewer bits than that (say 8 for the ASID instead of the full 32 bits for a PID).

If we take our example TLB from above and add ASIDs (and a few other fields), we can observe that two identical VPNs for different processes can readily share the TLB; only the ASID field is needed to differentiate the two translations:

VPN	PFN	valid	prot	ASID
0	10	1	rwX	1
—	—	0	—	—
0	17	1	rwX	2
—	—	0	—	—

Thus, with address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion. Of course, the hardware also needs to know which process is currently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the currently-running process.

As an aside, you may also notice another case where two entries of the TLB are remarkably similar. In this example, there are two entries for two different processes at two different VPNs that point to the same *physical* page:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

This situation might arise, for example, when two processes *share* a page (a code page, for example). In the example above, process 1 is sharing physical page 101 with process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its AS. Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use.

### 13.4 Issue: Replacement Policy

As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to **re-place** an old one, and thus the question: which one to replace?

## THE CRUX: TLB REPLACEMENT POLICY

Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

We will study such policies in some detail when we tackle the problem of swapping pages to disk in a virtual memory system. Here we'll just highlight a few of typical policies.

One common approach is to evict the **least-recently-used** or **LRU** entry. The idea here is to take advantage of locality in the memory-reference stream; thus, it is likely that an entry that has not recently been used is a good candidate for eviction as (perhaps) it won't soon be referenced again. Another typical approach is to use a **random** policy. Randomness sometimes makes a bad decision but has the nice property that there are not any weird corner case behaviors that can cause pessimal behavior, e.g., think of a loop accessing  $n + 1$  pages, a TLB of size  $n$ , and an LRU replacement policy.

### 13.5 An Example

Before closing, let's briefly look at a real TLB and what is in it. This example is taken from the MIPS R4000 [3], which is a great example of a modern system that uses software-managed TLBs. All 64 bits of this TLB entry can be seen in Figure 13.2.

The MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed. The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory ( $2^{24}$  4KB pages).

There are a few other interesting bits in the MIPS TLB. We see a *global* bit (G), which is used for pages that are globally-shared among processes. Thus, if the global bit is set, the ASID is ignored. We also see the 8-bit *ASID*, which the OS can use to distinguish between address spaces (as described above). One question for you: what should the OS do if there are more than 256 ( $2^8$ ) processes running at a time? Finally, we see 3 *Coherence* (C) bits, which determine how a page is cached by the hardware (a bit beyond the scope of these notes); a *dirty* bit which is marked when the page has been written to (we'll see the use of this later); a *valid* bit which tells the hardware if there is a valid translation present in the entry. There is also a *page mask* field (not shown), which supports multiple page sizes; we'll see later why having larger pages might be useful. Finally, you can also see that some of the 64 bits are unused (and hence shaded gray in the diagram).

MIPS TLBs usually have 32 or 64 of these entries, most of which are used by user processes as they run. However, a few are reserved for the OS. A *wired* register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., while running the TLB miss handler).

Because the MIPS TLB is software managed, there needs to be instructions to update the TLB. The MIPS provides four such instructions: `TLBP`, which probes the TLB to see if a particular translation is in there; `TLBR`, which reads the contents of a TLB entry into registers; `TLBWI`, which replaces a specific TLB entry; and `TLBWR`, which replaces a random TLB entry. The OS uses these instructions to manage the TLB's contents. It is of course critical that these instructions are **privileged**; imagine what a user process could do if it could modify the contents of the TLB!

## 13.6 Summary

We have seen how hardware can help us make address trans-

lation faster. By providing a small, dedicated on-chip TLB as an address-translation cache, most memory references will hopefully be handled *without* having to access the page table in main memory. Thus, in the common case, the performance of the program will be almost as if memory isn't being virtualized at all, an excellent achievement for an operating system.

## References

[CP78] "The architecture of the IBM System/370"

R.P. Case and A. Padegs

Communications of the ACM. 21:1, 73-96, January 1978

*According to Hennessy and Patterson, this is the first paper to use the term "Translation Lookaside Buffer". Yes, we all wish it was called something that made more sense.*

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"

Intel, 2009

Available: <http://www.intel.com/products/processor/manuals>

*In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"*

[H93] "MIPS R4000 Microprocessor User's Manual".

Joe Heinrich, Prentice-Hall, June 1993

Available: [http://cag.csail.mit.edu/raw/documents/R4400\\_Uman\\_book.Ed2.pdf](http://cag.csail.mit.edu/raw/documents/R4400_Uman_book.Ed2.pdf)

[WG00] "The SPARC Architecture Manual: Version 9"

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

## DESIGN TIP: CACHING

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the “common-case fast”. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely again be accessed again soon in the future. Think of a loop variable or the instructions in a loop; over and over again they will be accessed. With spatial locality, the idea is that if a program accesses memory at address  $x$ , it will likely soon access memory near  $x$ . Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don't we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

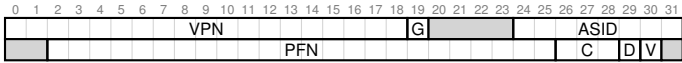


Figure 13.2: A MIPS TLB Entry.