Chapter 19: Faster Translations (TLBs)

How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require? What hardware support is required? What OS involvement is needed?

To speed up the address translation, we use **translation-lookaside buffer**, or **TLB**. TLB is part of the chip’s **memory-management unit** and it is simply a hardware cache of popular virtual-to-physical address translations (**address-translation cache**).

**19.1 TLB Basic Algorithm**

Text

Description automatically generated

The above picture shows how hardware might handle virtual address translation, assuming a simple linear page table and a hardware-managed TLB.

The algorithm first extracts the VPN from the virtual address and check if the TLB holds the translation for this VPN. If it does, we have a TLB hit, meaning that TLB holds the translation. We can now extract the PFN from the relevant TLB entry, concatenate that onto the offset from the original virtual address and form the desired physical address, and access memory, if it does not fail.

If we have a TLB miss, in this example, the hardware accesses the page table to find the translation and updates the TLB with the translation assuming that the memory is valid and accessible. Once the TLB is updated, the hardware retries the instruction and the memory reference is processed quickly.

If a hit happens, little overhead is added. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference results.

**19.2 Example: Accessing An Array**

Assume that we have an array of 10 4-byte integers in memory, starting at virtual address 100. We also have a 8-bit virtual address space with 16-byte page.

Consider the loop to access each element in the list, assuming that accessing the first element will be a TLB miss, then the next two elements will be a hit because they live on the same page. Then, when we try to access a[3], we encounter a miss because it now lives on different page. The next three elements will now then be a hit. The same things happen to the next pages.

Table

Description automatically generated

In this scenario, the hit rate will be 70%. This is due to use of **spatial locality** (memory near x is likely to be accessed) of the TLB cache. Notice that the size of the pages plays a very important role as if it is bigger, then the hit rate would be increased.

If the program ever accesses the array again, we would encounter better result of maybe all hits. This is because the TLB cache uses **temporal locality** (memory recently accessed will likely to be accessed again).

**19.3 Who Handles The TLB Miss?**

The hardware or the software? We can use the hardware as it needs to know exactly where the page tables are located in the memory (via a page-table base register, used in Line 11 in Figure 19.1) 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.

Modern architecture have what is known as the **software-managed TLB**. On a TLB miss, the hardware simply raises an exception, pausing the current instruction stream, raising the privilege level of the kernel mode and jump to a trap handler.

There are some details we need to pay attention to. First, the return-from-trap instruction needs to be a little different than the return-from-trap we saw before when servicing a system call. Second, when running the TLB miss-handling code, the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur. We can solve this by keeping TLB miss handlers in physical memory or reserve some entries in the TLB for permanent-valid translations and use some of those permanent translation slots for the handler code itself.

The primary advantage of the software-managed approach is **flexibility** as the OS can use any data structure it wants to implement the page table without necessitating hardware change. Another advantage is **simplicity**.

**19.4 TLB Contents: What’s In There?**

A typical TLB might have 32, 64, or 128 entries and be what is called **fully associative**. 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 TLB entry might look like this:

VPN | PFN | other bits

Since both VPN and PFN are in each entry, a translation could end up in any of these locations. The other bits include valid bit, protection bits, dirty bit, etc.

**19.5 TLB Issue: Context Switches**

The TLB contains virtual-to-physical translations that are only valid for the currently running process. As a result, when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process. For example:

A picture containing text, clock

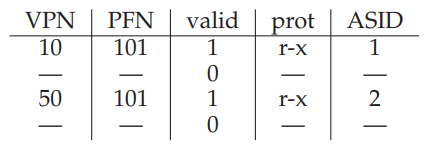
Description automatically generated

A solution to this is to flush the TLB on context switches. This creates a cost that each time a process runs, it must incur TLB misses. To solve this issue, some hardware systems 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. If we take the above example and add ASID, we will have

Table

Description automatically generated

OS must set some privileges to the ASID of the current process. Another case is that two entries of the TLB are remarkably similar. For example, two entries for two different processes with two different VPNs that point to the same physical page:



Sharing of code pages is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

**19.6 Issue: Replacement Policy**

Another issue is **cache replacement**, i.e., when we installing a new entry in the TLB, which one shall we replace?

One common approach is to evict the **least-recently-used** or **LRU** entry. Another typical approach is to use a **random policy**, which evicts a TLB mapping at random.

**19.7 A Real TLB Entry**

Table

Description automatically generated

MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would need a 20-bit VPN and 12-bit offset. However, there are only 19 bits for the VPN. However, user addresses will only come from half the address space, so we only need 19 bits.

We can see the 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.

The coherence bits (C) determine how a page is cached by the hardware. A dirty bit which is marked when the page has been written to. 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.