A quick summary about TLBs

M1 Mosig - Principles of Operating Systems and Concurrent Programming

Renaud Lachaize - November 2025

Important note: This document provides a quick summary/review regarding the notion of TLB. It does not provide a full coverage of the topic and is not meant as a replacement of the lectures and class discussion or the dedicated textbook chapter.

What is a TLB?

TLB stands for "translation lookaside buffer" but understanding this full name is not really important.

A TLB is a very small and fast hardware cache included in the MMU (memory management unit), which is itself integrated within each CPU core.

More precisely, a TLB is a specialized cache that is used to improve the performance of address translations, in the case of machines that use a paging-based approach for virtual memory support.

A TLB stores the result of the recently performed translations between virtual memory addresses and physical memory addresses (+ the corresponding permissions).

A TLB plays a crucial role for improving the performance of modern applications, which typically have a large memory footprint for their code and data.

At the very least, each TLB entry contains the following fields:

- A virtual page number (VPN)
- The corresponding physical page number (PPN, also known as PFN, page frame number)
- The access permissions associated with the page (present bit, user/kernel, read/write)
- A validity bit: boolean flag indicating whether this TLB entry is valid/meaningful or not.

(For simplicity, we ignore here the support required to deal with multiple page sizes.)

TLB designs

TLB lookups and TLB hits are always fully managed by the hardware.

However, there are two main types of TLB designs, which mainly differ in their way of handling TLB misses.

Design 1 - "Architected page table": This is the design adopted in (32-bit and 64-bit) mainstream processors, such as Intel/AMD x86 and Arm. In this design, every TLB miss is automatically handled by the MMU hardware, which is aware of the precise format/semantics of the paging structure (stored in main memory). Upon a TLB miss, the MMU hardware automatically walks trough the paging structure to find the translation information and inserts it into a TLB entry (note that this may require evicting/replacing another TLB entry if there is currently no available/unused entry). With this design, the TLB hardware operates almost (but not totally, as discussed in the next section) transparently from the point of view of the OS.

Design 2 - "Architected TLB": This is the design adopted by some processors, such as MIPS. In this approach, the MMU hardware is not aware of the paging structure format and location. Upon every TLB miss, a trap to the OS kernel is triggered, which results in the execution of a TLB miss (software) handler provided by the kernel code. The purpose of this handler is to look up the translation information in the paging structure (maintained by the kernel) and then (using specific/privileged CPU instructions) to insert this information into an entry of the TLB (note that this may require evicting/replacing another TLB entry if there is currently no available/unused entry). Finally, a special CPU instruction (used at the end of the TLB miss kernel handler) allows returning from the trap and retrying the instruction that initially caused the TLB miss.

Each of these two designs has some advantages and drawbacks, which we will not discuss here. (But it is a good exercise to try to think about them.)

Some processors (not very common nowadays) have a flexible design that allows choosing between the two operating modes described above (i.e., hardware-managed or software-managed TLB design).

TLB flushes and invalidations

When the kernel performs a context switch between two distinct processes on a CPU core, it is necessary to flush the contents of the TLB on this core (unless the TLB design has "tagged" entries, as discussed in the next section).

Flushing a TLB means invalidating all of its entries, that is, setting the validity bit to false in each entry. Performing the flush takes little time in itself but negatively impacts the performance of application afterwards, due to the additional TLB misses incurred.

Performing a TLB flush is necessary because, otherwise, the TLB could retain some translation results that are inadequate/incorrect in the context of the newly running process.

There are also other circumstances that require removing translation entries from the TLB. A typical example is when a process destroys a virtual memory mapping (via the `munmap()` system call). Another example is when the permissions of a page mapping are changed to be more restrictive (for instance, changing the permissions of a page/region from read-write to read-only). For such situations, some processors provide specific instructions allowing to invalidate only certain TLB entries (corresponding to a given VPN or a given virtual address range). It is also important to note that this kind of "mapping/permission invalidation" situation (unlike a simple context switch between process) is more difficult to handle in the case of a machine with multiple CPU cores. Indeed, an invalidation must be propagated to all the CPU cores (in case several/all of the CPU cores are running threads from the same process) and, furthermore, the hardware does not support automatic synchronization of changes between the TLBs of the different cores (unlike the general-purpose L1/L2/L3 CPU caches of the different cores). Therefore, whenever such an invalidation is required, the kernel must execute a complex and slow so-called "TLB shootdown" protocol, which involves sending interrupts to all the CPU cores (to notify them than they must also perform the invalidation on their local TLB) and awaiting their acknowledgments.

TLB tags

To avoid the necessity to systematically flush the TLB upon a context switch between distinct processes, some CPU designs leverage a technique named "TLB tagging". In such a design, the format of each TLB entry is extended with an additional field, storing the identifier of the process for which the translation applies. This identifier is named PID, or more often ASID (address space ID). In addition, this design also leverages a new CPU register (updated whenever necessary by the kernel upon a context switch) to store the ASID of the currently running process.

When the hardware MMU performs a lookup in the TLB, the lookup is based on the tuple `(requested VPN, current ASID)`. And, when a TLB entry is inserted (after a TLB miss), the ASID field of the entry is set to ASID of the currently running process.

In this way, there is no potential ambiguity about the translations stored in the TLB, and multiple translations for the same VPN (stored in different entries) can even coexist in the TLB.