TLBs Summary - SistOp

Lautaro Bachmann

Contents

Paging: Faster Translations (TLBs)

Introduction

translation-lookaside buffer, or TLB

To speed address translation, we are going to add what is called a translation-lookaside buffer, or TLB

A TLB is part of the chip's **memory-management unit (MMU),** and is simply a hardware **cache** of popular virtual-to-physical address translations;

Upon each virtual memory reference,

the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) **without** having to consult the page table

TLB Basic Algorithm

Assumptions

- We are using a simple linear page table
- We are using a hardware-managed TLB

first,

extract the virtual page number (VPN) from the virtual address and check if the TLB holds the translation for this VPN

TLB hit,

which means the TLB holds the translation.

We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory assuming protection checks do not fail

TLB miss

This happens when the CPU does not find the translation in the TLB

the hardware accesses the page table to find the translation and, updates the TLB with the translation

These set of actions are costly,

Finally,

once the TLB is updated, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB premise

The TLB is built on the premise that in the common case, translations are found in the cache

When a miss occurs,

the high cost of paging is incurred;

TIP: USE CACHING WHEN POSSIBLE

locality

temporal locality,

the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future.

spatial locality,

the idea is that if a program accesses memory at address x, it will likely soon access memory near x.

Hardware caches,

Hardware caches take advantage of locality by keeping copies of memory in small, fast on-chip memory.

If you want a fast cache,

it has to be small, Any large cache by definition is slow, and thus defeats the purpose.

Who Handles The TLB Miss?

Hardware-managed TLB

In the olden days, the hardware would handle the TLB miss entirely.

To do this, 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.

software-managed TLB.

the hardware simply raises an exception which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler.**

When run,

the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction

couple of important details.

the return-from-trap instruction

needs to be a little different than the return-from-trap when servicing a system call. when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that **caused** the trap;

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.

advantage of the software-managed approach

flexibility:

the OS can use any data structure it wants to implement the page table, without necessitating hardware change.

simplicity,

The hardware doesn't do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.

ASIDE: TLB VALID BIT $\neq$ PAGE TABLE VALID BIT

Page table valid bit

If the PTE is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a correctly-working program.

TLB valid bit,

refers to whether a TLB entry has a valid translation within it.

TLB Contents: What's In There?

Typical size

A typical TLB might have 32, 64, or 128 entries

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.

TLB entry

A TLB entry might look like this:

VPN | PFN | other bits

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

“other bits”.

valid bit,
says whether the entry has a valid translation or not.

protection bits,
determine how a page can be accessed

For example, code pages might be marked read and execute, whereas heap pages might be marked read and write.

other fields,
including an **address-space identifier,** a **dirty bit,** and so forth;

TLB Issue: Context Switches

issues when switching between processes

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.

possible solutions

One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process.

However, there is a cost:
each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead,
some systems add hardware support to enable sharing of the TLB across context switches.

In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a **process identi-**

fier (PID), only the ASID field is needed to differentiate otherwise identical translations.

Issue: Replacement Policy

Specifically, when we are installing a new entry in the TLB, we have to **replace** an old one, and thus the question: **which one to replace?**

a few typical policies.

least-recently-used

One common approach is to evict the least-recently-used entry

LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction.

random policy,

Another typical approach is to use a random policy which evicts a TLB mapping at random.

Summary

TLB coverage,

If the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of **TLB misses**, and thus run quite a bit more slowly. We refer to this phenomenon as exceeding the **TLB coverage**

other TLB issue

TLB access can easily become a bottleneck in the CPU pipeline, in particular with what is called a **physically-indexed cache.** With such a cache, address translation has to take place before the cache is accessed, which can slow things down quite a bit.