Chapter 19 Paging: Faster Translation (TLBs)

Problem with paging: various overheads
- storage overhead for the page tables
- page tables stored in memory
- requires extra memory lookup for each virtual address
    - going to memory (for translation information)
      for every instruction fetch or load/store is prohibitively slow

Q: How can we speed up address translation?

- Can we avoid the extra memory reference?
- What HW is required?
- What must the OS do?

Def. Translation-Lookaside Buffer (TLB) — (should be called Address Translation Cache)
Cache of popular virtual-to-physical address translations.

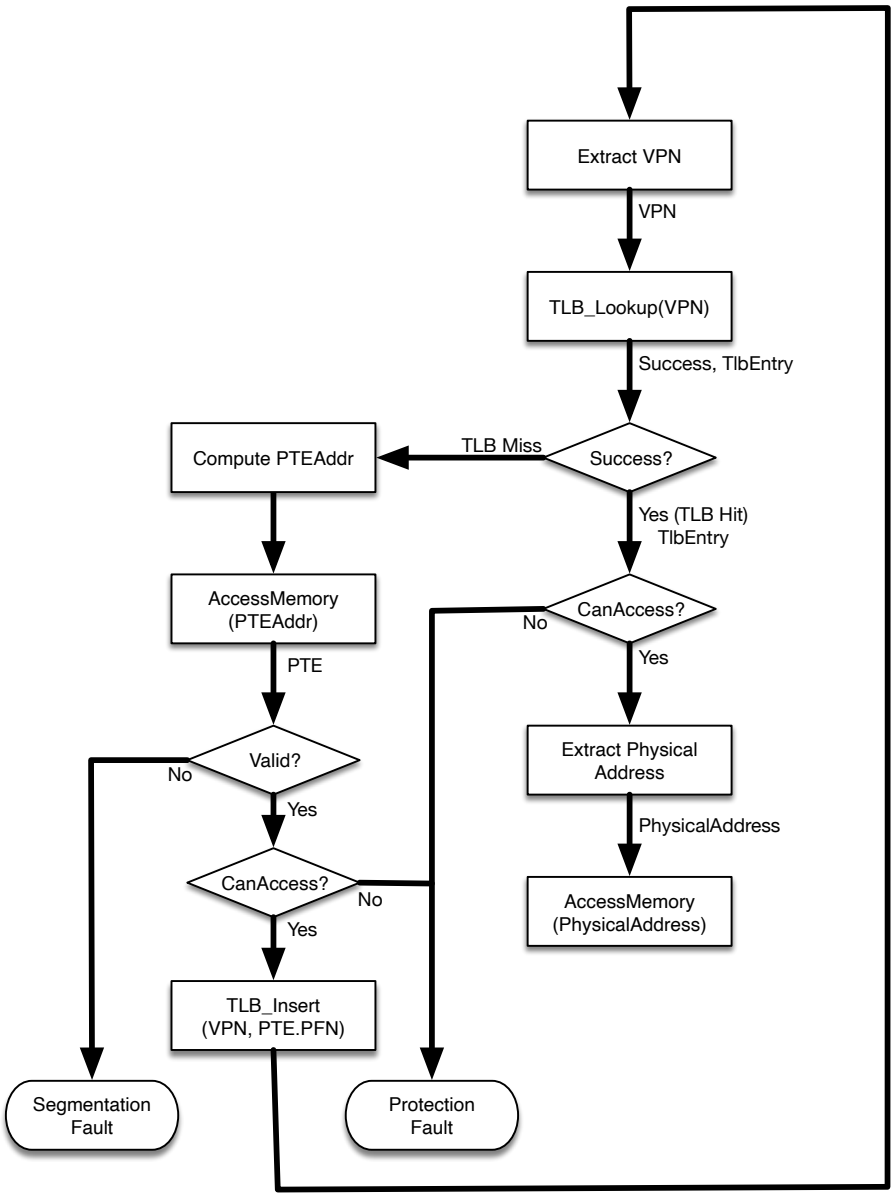
Idea:

For each virtual memory reference:

- HW checks if desired virtual address to physical address mapping is in TLB
    - If so, use TLB translation without consulting page table in memory
    - Otherwise, consult page table (which has all translations)

The TLB check must be really really fast — because DRAM is also quite fast!

19.1 TLB Basic Algorithm




```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

1. Extracting the virtual page number (VPN) from virtual address.
2. Check if VPN is in TLB
3. If it is: TLB hit (success)
    1. Check if can access relevant memory loc.
    2. If so, extract the offset
    3. Compute the physical address
    4. Do memory access (store in register)
4. Otherwise (TLB miss)
    1. Computer address of PTE for VPN; by using the Page Table Base Register (PTBR)
    2. Do memory access to fetch PTE from memory
    3. Check if PTE is valid
        1. Not valid: raise exception
    4. Check if can access:
        1. Not access: raise exception
    5. Otherwise: all okay
        1. TLB insert: update TLB with data from PTE
        2. Retry instruction now that the TLB has everything

Key idea with TLB

- should have low overhead (even if there is a miss)
- Should avoid misses as much as we can
    - Misses lead to more memory accesses (slower)

19.2 Example: Access An Array

Examine a simple virtual address trace and see how a TLB can improve performance…

Ex. Assume an array a with 10 4-byte integers starting at v.a. 100

Assume 8-bit virtual address space

- 4 bit VPN (16 virtual pages)
- 4 bit offset (16 bytes on each page)

Consider this C program:

```
int sum = 0;
for (i = 0; i < 10; i++) {
        sum += a[i];
}
```

For simplicity we ignore:

- memory access to fetch instructions, and
- accessing variables such *i* and *sum*. (probably in CPU registers anyway).

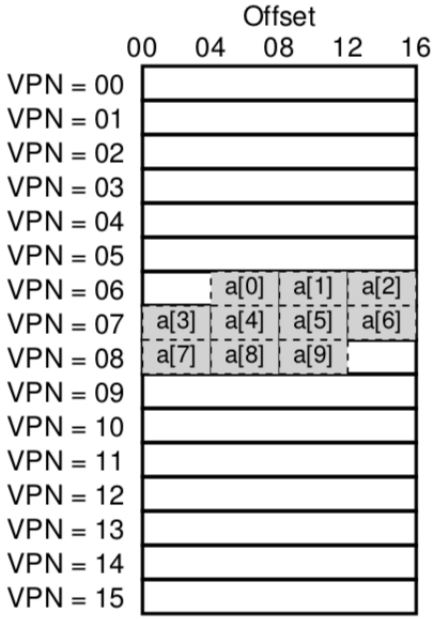


Figure 19.2: **Example: An Array In A Tiny Address Space**

a[0] = virtual address 100 —> **16*6 + 4**

1. Access a[0]: HW extract VPN = 06 from v.a. 100:
    1. TLB_Lookup(VPN=6): TLB miss
    2. Since a[0] has not been accessed before
    3. Access Page Table to get VPN = 6
        1. Gives us PTE, which contains PTE.PFN
    4. TLB_Insert(VPN=6, PFN)
2. Access a[1]: HW extract VPN = 06 from v.a. 104:
    1. TLB_Lookup(VPN=6): TLB hit
        1. Because VPN 6 is already in TLB
3. Access a[2]: TLB hit
4. Access a[3]: TLB miss
5. a[4]-a[6]: TLB hit
6. a[7]: TLB miss
7. a[8], a[9]: TLB hit

Summary: m, h, h, m, h, h, h, m, h, h

TLB hit rate = number of hits / total number of accesses

Hit rate = 7 / 10 = 70 % hit rate

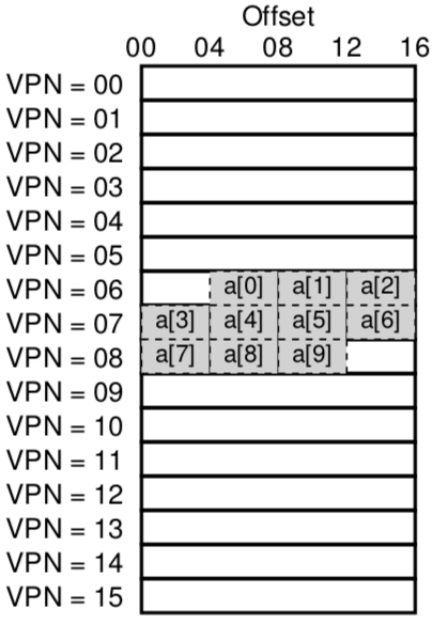


Figure 19.2: **Example: An Array In A Tiny Address Space**

Interesting: Even though we access different parts of a page we a TLB hit!

We say that the TLB improves performance due to **spatial locality.**

Def. Spatial Locality:
the elements we access are located close to one another in space (here: address space).

What if the page size were 32 bytes instead of 16 bytes? (5 bites instead of 4 bits)

- How many TLB misses?
- The array would be split over two pages instead of three
  - Answer: 2 TLB misses.
  - Each page could have 8 values, but in this example the first page will have 7 values.
  - And the second page will have the remaining 3 values.

Typical page size: 4 KB

- Gives excellent TLB performance!
- Only a single TLB miss per page of accesses
- Can loop over an array of size 4 KB with a single miss

If our program, soon after this loop, access the array again.

Likely to see no TLB misses if cache is big enough.

TLB hit rate is high due to **temporal locality.**

Def. Temporal Locality:
Quick re-referencing of memory items in time.

TLBs rely on both spatial and temporal locality for success!

Q: if caches are so great, why not make them bigger?

A: Fundamental laws physics: speed of light and space for transistors on chip.

By definition, large caches are slow!

Energy! Activate more transistors for comparisons!

19.3 Who Handles The TLB Miss?

OS or HW? It depends.

HW managed TLBs (Line 11 in Fig 19.1)

- **page table base register**: points to page table in memory
- HW takes care of walking the page table to
  - Find the correct PTE (page table entry)
  - Extract translation
  - Update TLB with translation
  - Retry instruction
- x86 uses a fixed multi-level page table (next chapter)

OS managed TLBs

- HW raise exception
  - Pause the instruction stream
  - (change to) kernel mode
  - Jump to OS trap handler for TLB misses
    - Look up in page table
    - Update TLB; privileged instruction
    - Return-from-trap
- HW retries the instruction ; resulting in TLB hit

Important details

- Return-from-trap
  - Now resume execution at the instruction that caused the trap
  - Different from the usual return-from-trap, which resumes at the next instruction
- Avoid infinite TLB misses (caused by the TLB handler itself)
  - Alt1: Keep TLB miss handles in physical memory
  - Alt2: Reserve some entries in the TLB for permanently-valid translations

OS approach is

- More flexible: can use any data structure it wants to implement page table
- Simple hardware: only raise exception on TLB miss…

19.4 TLB Contents: What's in There?

Typical sizes: 32, 64, 128 entries

Def. Fully Associative:

- A translation can be anywhere in the TLB
- HW will search the entire TLB in parallel to find the desired translation

TLB Entry: VPN | PFN | Other bits

Other bits:

- Valid: (set if) entry has a valid translation
  - At boot all entries are not-valid (until virtual memory is enabled)
  - Populated as more pages are accessed
- Protection
  - Code pages: mark as **read** and **execute**
  - Heap pages: mark as **read** and **write**
- Address-space identifier: see below
- Dirty: see below

19.5 TLB Issue: Context Switches

TLB contains translations only valid of the running process.

When switching between processes
HW/OS must ensure next process does not (accidentally)
use translations from previously running process!

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

Ex. Consider two procs P1 and P2

Assume P1: VPN=10 -> PFN=100

Assume P2: VPN=10 -> PFN=170

Problem: HW cannot distinguish which entry is meant for which process.

One solution:

- Naive solution: flush TLB on context switch: set all valid bits to 0.

Problem:

- Each time process runs, must incur TLB misses again and again for each context switch
  - Frequent switching, leads to high costs

To reduce this overhead:

- TLB HW support for context switching
- Address space identifiers (ASID) field in TLB
  - Similar to PID, but has fewer bits
  - Maybe 8 bits vs 32 bits for a PID
- Now we can distinguish between processes in TLB

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Two different processes with same VPN, pointing to **different** physical page.

ASID can also support **sharing code pages** (binaries or shared library)

Two different processes with different VPNs point to the **same** physical page.

Example:

- P1 and P2 share a code page a PFN 101
- But they have placed them at v.a. 10 and v.a. 50, respectively

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

19.6 Issue: Replacement Policy

Caches are small: can't fit all page table entries of a single process (let alone all processes) in cache

- So we need to replace an old page with a new page
- Q: Which one to replace?

Goal: *Minimize miss rate (maximize the hit rate)*

Two approaches:

- Evict the **least-recently-used (LRU)** entry
  - LRU assumes it is likely that an entry that hasn't been used recently:
  - Good candidate for eviction
- Evict pages at **random**
  - Can be good for cases where LRU observe (bad) corner-case behaviors
  - LRU bad: loop over n+1 pages, where TLB size is n.

19.7 A Real TLB Entry

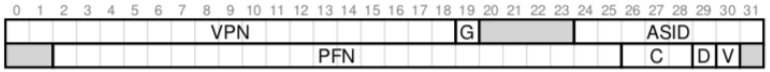


Figure 19.4: **A MIPS TLB Entry**

Here for the MIPS architecture:

- VPN 19 bits
- PFN 24 bits
- Offset 12 bits (4 KB pages)

Each virtual address space (per process):

$2^{19+12} = 2^{31} = 2$ GB

Total physical memory that MIPS can support:

$2^{24+12} = 2^{36} = 64$ GB

- ASID 8 bit
  - 256 processes an remain in TLB across context switches
- Other bits

  G - global bit - pages to be shared globally among all processes.

  C - coherence bits - how page cached by HW (out of scope)

  V - valid as above

  D - Dirty

19.8 Summary

If # pages a program access in a short period of time exceed TLB capacity

- Large number of TLB misses
- Program will run slowly
- Phenomenon: exceeding the TLB's coverage

Culler's law: RAM isn't always RAM

- RAM = Random Access
  - Accessing memory at random can cause problems for TLB coverage
  - TLB source of many performance problems
    (in reality we are accessing TLB rather than page tables,
    but when the TLB's coverage isn't met then things become slow…)

One solution (next chapter)

- Support larger page size
- Map key data structures into regions of a program's AS that are mapped by larger pages
  - Increase TLB coverage