Introduction

Using paging as the primary mechanism for supporting virtual memory can result in significant performance penalties. Paging necessitates a substantial quantity of mapping information since it divides the address space into small, fixed-sized units (i.e., pages). Because that mapping information is usually kept in physical memory, paging logically necessitates an additional memory lookup for each virtual address generated by the software. It is prohibitively slow to go to memory for translation information before every instruction fetch, explicit load, or store. As a result, this is our issue:

challenge

How can we reduce the time it takes to translate addresses and prevent the extra memory reference that paging appears to necessitate? What kind of hardware is required? What level of OS engagement is required?

When we want to do anything quickly, the OS frequently need assistance. And the OS's old friend, the hardware, often comes to the rescue. We're going to use a translation lookaside buffer, or TLB, to speed up address translation. A TLB is a hardware cache of popular virtual-to-physical address translations that is part of the chip's memory management unit (MMU). A better name for it would be an address-translation cache. When the hardware encounters a virtual memory reference, it first examines the TLB to see if the requested translation is stored there; if it is, the translation is done (fast) without the need to visit the page table (which has all translations). TLBs, in a sense, enable virtual memory because of their enormous performance impact.

The Basic Algorithm

Figure 19.1

```
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 depicts a rough sketch of how hardware might handle a virtual address translation, assuming a simple linear page table (an array) and a hardware-managed TLB (the hardware manages page table accesses; see below).

The hardware algorithm is as follows: to check if the TLB retains the translation for this VPN, first extract the VPN from the virtual address (Line 1 in **Figure 19.1**). (Line 2). If it does, the TLB has the translation. Success! We can now extract the PFN from the TLB entry, add it to the offset from the initial virtual address to get the appropriate physical address (PA), and access memory (Lines 5–7). (Line 4).

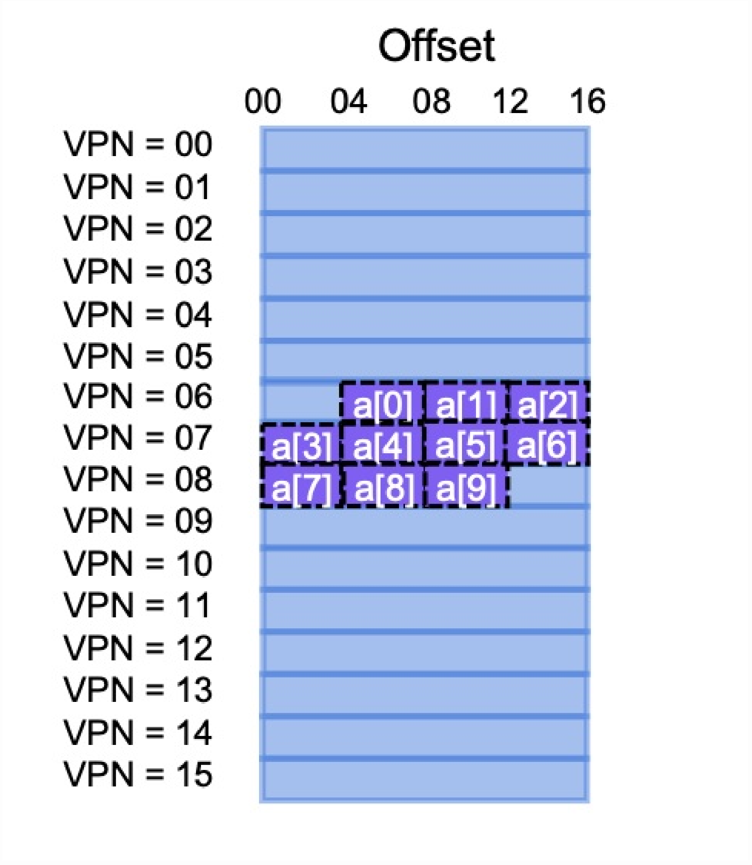
If the CPU cannot find the translation in the TLB, we have extra work. In this case, the hardware searches the page table for the translation (Lines 11–12) and updates the TLB with the translation (Lines 13, 15). (Line 18). The extra memory reference required to reach the page table makes this series of actions costly (Line 12). The hardware then retries the instruction, this time finding the translation in the TLB and processing the memory reference quickly.

The TLB, like all caches, assumes that translations are located in the cache (i.e., are hits). If so, the TLB is close to the processing core and is designed to be fast. When a miss occurs, the page table must be queried to find the translation, resulting in one additional memory reference (or more with more sophisticated page tables). TLB misses result in more memory accesses, which slows down the program. So we try to avoid TLB misses.

Array Access Example

Let’s look at a simple virtual address trace and see how a TLB can improve its speed. Assume we have a memory array of 10 4-byte integers starting at virtual address 100. Also assume an 8-bit virtual address space with 16-byte pages; a virtual address is composed of a 4-bit VPN (16 virtual pages) and a 4-bit offset (there are 16 bytes on each of those pages).

Figure 19.2



.guides/img/figure19-2

Figure 19.2 illustrates the array on the system’s 16 16-byte pages. As you can see, the array’s first entry (a[0]) starts on page (VPN=06, offset=04). The array continues on the next page (VPN=07) with `a[3]... a[6]` elements. Finally, the 10-entry array’s last three entries (a[7]… a[9]) appear on the next page (VPN=08).
Consider a basic loop that examines each array entry, like in C:

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

For simplicity, we'll imagine that the loop's sole memory accesses are to the array (ignoring the variables I and sum, as well as the instructions themselves). The first array element (a[0]) accesses virtual address 100. A correct translation is checked in the TLB by the hardware by extracting the VPN (VPN=06). If this is the first time the program accesses the array, a TLB miss occurs.

The next access is to a[1], and it's a TLB hit! The translation is already loaded into the TLB because the second element of the array is packed next to the first. And hence our success. Because a[2] is on the same page as a[0] and a[1,] it is also accessible.
Sadly, the program hits another TLB miss when it accesses a[3]. As they are on the same page in memory, the next entries (a[4]… a[6]) will also hit TLB.

Final TLB miss occurs when accessing a[7]. The hardware searches the page table again to find the virtual page in physical memory and updates the TLB. The final two accesses (a[8] and a[9]) benefit from the TLB update, resulting in two more hits.

For our 10 array accesses, we had the following TLB activity: 0 0 0 0 0 0 0 0 0 So our TLB hit rate (number of hits divided by total accesses) is 70%. It's not too high (we want near-perfect hit rates), but it's not zero. The TLB improves performance even if this is the first time the application accesses the array. Because the array's elements are closely packed into pages (in space), only the first access to an element on a page causes a TLB miss.

Notice how page size affects this example. The array access would have been better if the page size had been doubled (32 bytes instead of 16). Because pages are typically 4KB in size, dense array-based accesses have outstanding TLB performance, with only one miss per page of accesses.

To summarize TLB performance, we would expect even better results if we accessed the array again soon after the loop ended, assuming we had a large enough TLB to cache the required translations: hit, hit, hit, hit. Temporal locality, i.e., the rapid re-referencing of memory elements in time, would increase TLB hit rate. TLBs, like any cache, rely on program-defined spatial and temporal locality. If the program has such locality (and many do), the TLB hit rate will be high.

Who is responsible for TLB misses?

One question: who handles a TLB miss? The hardware or the software? (OS). CISC stands for complex instruction set computers, and the guys who created the hardware didn't trust those devious OS people. The hardware would then handle the TLB miss. To do this, the hardware must know the exact location of the page tables in memory (through a page-table base register, utilized in Line 11 of **Figure 19.1**), as well as their exact format. The Intel x86 architecture employs a fixed multi-level page table (see the following chapter for details); the current page table is pointed to by the CR3 register.

▼ **Figure 19.1**

```
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 19.3

```
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      RaiseException(TLB_MISS)
```

Modern architectures (like MIPS R10k or Sun's SPARC v9, both RISC processors) contain a software-managed TLB. The hardware simply raises an exception (line 11 in **Figure 19.3**), pausing the current instruction stream and jumping to a trap handler. As you might expect, this trap handler is OS code intended to handle TLB misses. When run, the code looks up the translation in the page table, updates the TLB, and returns from the trap; the hardware then retries the instruction (resulting in a TLB hit).

Let's clarify a few points. First, the return-from-trap command must be different than the one used to service a system call. In this situation, the return-from-trap should restart execution at the instruction following the OS trap, exactly as a return from a procedure call does. In the former situation, returning from a TLB miss-handling trap causes the hardware to retry the instruction, resulting in a TLB hit. So, depending on the trap or exception, the hardware must save a different PC when trapping into the OS to resume properly.

Second, when running the TLB miss-handling code, the OS must avoid causing an endless cycle of TLB misses. Keeping TLB miss handlers in physical memory (unmapped and not subject to address translation) or reserving some TLB entries for permanently valid translations and using some of those permanent translation slots for the handler code itself.

The main benefit of a software-managed page table is flexibility: the OS can use any data structure it wishes to implement the page table. The hardware doesn't do much on a miss: merely raise an exception and let the OS TLB miss handler do the rest, as seen in **Figure 19.3**.

Inside the TLB

Let’s take a closer look at the hardware TLB’s contents. A typical TLB could include 32, 64, or 128 entries and be fully associative. This simply means that any given translation can be found anywhere in the TLB, and the hardware will search the entire TLB in parallel to find it. The following is an example of a TLB entry:

VPN	PFN	other bits

Because a translation could wind up in any of these locations, both the VPN and PFN are contained in each entry (in hardware terms, the TLB is known as a fully-associative cache). To see if there is a match, the hardware searches the entries in parallel.

The “other pieces” are more intriguing. A valid bit in the TLB, for example, indicates whether the entry has a valid translation or not. Protection bits, which define how a page can be accessed, are also frequent (as in the page table). Code pages, for example, may be designated read and execute, but heap pages could be read and write. There may also be a few extra fields, such as an address-space identification, a dirty bit, and so on; for further information, see the section below.

TLB Issue: Context Switching

TLBs introduce new challenges while switching processes (and hence address spaces). The TLB contains virtual-to-physical translations that are only valid for the presently operating process. As a result, when switching between processes, the hardware or OS (or both) must verify that the new process does not employ translations from a previous process.

Let us use an example to better grasp the situation. When one process (P1) runs, it assumes the TLB is caching valid translations from P1's page table. Assume P1's 10th virtual page is assigned to frame 100.

Assume another process (P2) exists, and the OS decides to run it via a context switch. Assume P2's 10th virtual page is assigned to frame 170. The TLB would contain entries for both processes if they existed:

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

We have a problem in the TLB: VPN 10 is either PFN 100 (P1) or PFN 170 (P2), but the hardware can't tell the difference. So the TLB needs more development to appropriately and efficiently enable multi-process virtualization. So, a dilemma:

challenge

When context-switching between processes, the last process's TLB translations are useless to the next process. What should the hardware or OS do?

This problem has various solutions. Flushing the TLB on context transitions might empty it before launching the following process. With a software-based TLB, the flush can be performed when the page-table base register is modified (note the OS must change the PTBR on a context switch anyhow). In any scenario, flushing sets all valid bits to 0, thus clearing the TLB.

We now have a workable solution by flushing the TLB on each context switch. But there's a catch: every time a process touches its data or code pages, it causes TLB misses. This cost may be high if the OS switches processes often.

Some systems provide hardware support for TLB sharing across context transitions to decrease this overhead. Some hardware systems provide an ASID field in the TLB. The ASID is a process identifier (PID), although it has less bits (e.g., 8 bits for the ASID versus 32 bits for a PID).

Adding ASIDs to our previous TLB shows that processes can easily share TLBs: only the ASID field distinguishes otherwise identical translations. Here is an example of a TLB containing an ASID field:

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

With address-space IDs, the TLB can hold translations from many processes simultaneously. To conduct translations, the hardware has to know which process is presently running, thus the OS must set a privileged register to the current process's ASID.

You may have also thought of another situation where two TLB entries are strikingly similar. In this case, two entries for two processes with two VPNs 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 can happen when two processes share a page (a code page, for example). In the aforementioned example, Process 1 and Process 2 share physical page 101; P1 maps it to the 10th page of its address space, while P2 maps it to the 50th page. Sharing code pages (in binaries or shared libraries) decreases memory overheads by lowering the number of physical pages used.

The Replacement Policy Issue

Another issue to address with any cache, and thus also with the TLB, is cache replacement. In particular, when we install a new entry in the TLB, we must replace an existing one, which raises the question of which one to replace.

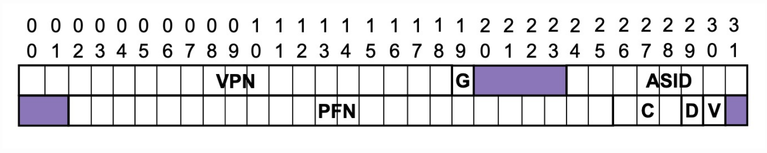
challenge

When we add a new TLB entry, which one should be replaced? Of course, the idea is to reduce misses (or increase hits) and therefore improve performance.

When we handle the topic of switching pages to disk, we'll go through a few common rules in greater depth; for now, we'll just highlight a few examples. Evicting the least-recently-used or LRU entry is a popular strategy. In the memory-reference stream, LRU seeks to take advantage of locality, presuming that an entry that hasn't been utilized in a while is a good candidate for eviction. A random policy, which evicts a TLB mapping at random, is another common option. Such a policy is useful because of its simplicity and ability to avoid corner-case behaviors; for example, when a program loops over n + 1 pages with a TLB of size n, a "reasonable" policy like LRU behaves quite unreasonably; in this case, LRU misses on every access, whereas random performs much better.

A Real TLB Entry

Figure 19.4



.guides/img/figure19-4

Finally, consider a real TLB. **Figure 19.4** shows a somewhat simplified MIPS TLB entry from the MIPS R4000, a modern system that uses software-managed TLBs.

32-bit address space with 4KB pages on MIPS R4000. Our typical virtual address would have a 20-bit VPN and a 12-bit offset. However, as seen in the TLB, the VPN requires just 19 bits because user addresses only come from half of the address space (the rest is reserved for the kernel). The VPN can support systems with up to 64GB (physical) main memory because it converts to a 24-bit PFN (224 4KB pages).

The MIPS TLB has a few extra noteworthy pieces. We notice a global bit (G) for shared pages between processes. The ASID is disregarded if the global bit is set. The OS can use the ASID to distinguish across address spaces (as described above). What should the OS do if there are more than 256 (28) processes running? On the other hand, a dirty bit indicates when a page has been written to (we'll see how this is used later) and a valid bit indicates if the entry contains a valid translation. We'll see why having larger pages might be advantageous later. Finally, about 64 bits are wasted (which are shaded in the diagram).

Most MIPS TLBs feature 32 or 64 of these entries, which are used by user programs. But the OS gets a few. When a TLB miss occurs, the OS can set a wired register to instruct the hardware how many TLB slots to reserve for the OS (e.g., in the TLB miss handler).

A software upgrade is required since the MIPS TLB is software managed. The MIPS provides four such instructions: TLBP probes the TLB for a specific translation; TLBR reads a TLB entry's contents into registers; TLBWI re-places a specific TLB entry; and TLBWR replaces a random TLB entry. So the OS can handle the TLB's content. Imagine what a user process could do if it could manipulate the contents of the TLB (hint: take over the machine, run its own malicious "OS", or even make the Sun disappear).

Summary

We’ve seen how hardware can help with address translation. Most memory references should be handled without touching the main memory’s page table, thanks to an on-chip TLB acting as an address-translation cache. It works almost as if memory isn’t virtualized at all, which is amazing for an operating system and required by current systems to use paging.

TLBs, on the other hand, do not benefit every program. This causes a significant number of TLB misses, slowing down the program. This is known as TLB coverage surpassing, and it might be problematic for some programs. The next chapter will show how to add support for larger page sizes. The TLB’s effective coverage can be increased by mapping critical data structures into larger pages’ address space. Massive page support is widely used in programs like database management systems (DBMS) that have large and unpredictable data structures.

TLB access, especially with a physically indexed cache, can rapidly become a CPU pipeline bottleneck. Using such a cache requires address translation, which can considerably slow things down. Due to this potential difficulty, people have come up with clever ways to access caches using virtual addresses, skipping the costly step of translation. A virtual indexed cache improves performance but introduces new hardware design difficulties.