18

Paging: Faster Translations (TLBs)

When we want to make things fast, the OS needs some help. And help usually comes from one place: the hardware. To speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or **TLB** [C68,C95]. 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; thus, a better name would be an **address-translation cache**. 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 (which has all translations). Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible [C95].

Figure 18.1 shows how hardware might handle a virtual address translation (assuming a simple linear page table and a hardware-managed TLB). In the common case (lines 3–9), we are hoping that a translation will be found in the TLB (a **TLB hit**) and thus the translation will be quite fast (done in hardware near the processing core). In the less common case (lines 10–19), the translation won't be in the cache (a **TLB miss**), and the system will have to consult the page table in main memory, update the TLB, and retry the memory reference.

DESIGN TIP: CACHING

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the "common-case fast" [HP06]. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With 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. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address x, it will likely soon access memory near x. Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don't we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

18.1 Who Handles the Miss?

One question that we must answer: who handles a TLB miss? Two answers are possible: the hardware, or the software (OS). In the olden days, the hardware had complex instruction sets (some-

```
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 18.1: TLB Control Flow Algorithm

times called **CISC**, for complex-instruction set computers) and the people who built the hardware didn't much trust those sneaky OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly *where* the page tables are located in memory (via a **page-table base register**, used in line 11 in Figure 18.1), 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. An example of an "older" architecture that has **hardware-managed TLBs** is the Intel x86 architecture, which uses a fixed **multi-level page table** (see the next chapter for details); the current page table is pointed to by the CR3 register [I09].

More modern architectures (e.g., MIPS R10k [H93] or Sun's SPARC v9 [WG00], both **RISC** or reduced-instruction set computers) have what is known as a **software-managed TLB**. On a TLB miss, the hardware simply raises an exception (line 11 in Figure 18.2), which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler**. As you might guess, this trap handler is code within the OS that is written with the express purpose of handling TLB misses. 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 (resulting in a TLB hit).

```
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)
```

Figure 18.2: TLB Control Flow Algorithm (OS Handled)

Let's discuss a couple of important details. 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. In the latter case, the return-from-trap should resume execution at the instruction *after* the trap into the OS, just as a return from a procedure call returns to the instruction immediately following the call into the procedure. In the former case, when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that *caused* the trap; this retry thus lets the instruction run again, this time resulting in a TLB hit. Thus, depending on how a trap or exception was caused, the hardware must save a different PC when trapping into the OS, in order to resume properly when the time to do so arrives.

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. Many solutions exist; for example, you could keep TLB miss handlers in physical memory (where they are **unmapped** and not subject to address translation), or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself; these **wired** translations always hit in the TLB.

The primary advantage of the software-managed approach is *flexibility*: the OS can use any data structure it wants to implement the page table, without necessitating hardware change. Another advantage is *simplicity*; as you can see in the TLB control flow (line 11 in Figure 18.2, in contrast to lines 11–19 in Figure 18.1), the hardware doesn't have to do much on a miss; it raises an exception, and the OS TLB miss handler does the rest.

ASIDE: RISC VS. CISC

In the 1980's, a great battle took place in the computer architecture community. On one side was the **CISC** camp, which stood for **Complex Instruction Set Computing**; on the other side was **RISC**, for **Reduced Instruction Set Computing** [PS81]. The RISC side was spear-headed by David Patterson at Berkeley and John Hennessy at Stanford (who are also co-authors of some famous books [HP06]), although later John Cocke was recognized with a Turing award for his earliest work on RISC [CM00].

CISC instruction sets tend to have a lot of instructions in them, and each instruction is relatively powerful. For example, you might see a string copy, which takes two pointers and a length and copies bytes from source to destination. The idea behind CISC was that instructions should be high-level primitives, to make the assembly language itself easier to use, and to make code more compact.

RISC instruction sets are exactly the opposite. A key observation behind RISC is that instruction sets are really compiler targets, and all compilers really want are a few simple primitives that they can use to generate high-performance code. Thus, RISC proponents argued, let's rip out as much from the hardware as possible (especially the microcode), and make what's left simple, uniform, and fast.

In the early days, RISC chips made a huge impact, as they were noticeably faster [BC91]; many papers were written; a few companies were formed (e.g., MIPS and Sun). However, as time progressed, CISC manufacturers such as Intel incorporated many RISC techniques into the core of their processors, for example by adding early pipeline stages that transformed complex instructions into micro-instructions which could then be processed in a RISC-like manner. These innovations, plus a growing number of transistors on each chip, allowed CISC to remain competitive. The end result is that the debate died down, and today both types of processors can be made to run fast.

18.2 TLB Contents: What's In There?

Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called

fully associative. Basically, 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 typical TLB entry might look like this:

VPN | PFN | other bits

Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations; the hardware searches the entries in parallel to see if there is a match.

More interesting are the "other bits". For example, the TLB commonly has a **valid** bit, which says whether the entry has a valid translation or not. Also common are **protection** bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked *read and execute*, whereas heap pages might be marked *read and write*. There may also be a few other fields, including an **address-space identifier**, a **dirty bit**, and so forth; see below for more information.

18.3 TLB Issue: Context Switches

With TLBs, a new issue arises when switching between processes (and hence address spaces). Specifically, the contents of the TLB contain virtual-to-physical translations that are only valid for the current running process; these translations are not meaningful for other processes. Thus, when switching to run another process, the hardware or OS or both must be careful.

To understand this better, let's look at an example. When one process (P1) is running, it accesses the TLB with translations that are valid for it. Assume here that the 0th virtual page of process P1 might be mapped to physical frame 10. Another process may also be ready in the system (P2), and the OS might be context-switching between it and P1; assume the 0th virtual page of P2 is mapped to physical frame 17. If entries for both processes were in the TLB, it might look like this:

VPN	PFN	valid	prot
0	10	1	rwx
—	—	0	—
0	17	1	rwx
—	—	0	—

In the TLB above, we clearly have a problem: VPN 0 translates to either PFN 10 (P1) or PFN 17 (P2), but the hardware can't distinguish which entry is meant for which process. Thus, we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes. And thus, a crux:

THE CRUX:
HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH
When context-switching between processes, the translations in the TLB for the last process are not meaningful to the about-to-be-run process. What should the hardware or OS do in order to solve this problem?

There are a number of possible solutions. One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this could be accomplished with an explicit (and privileged) hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed (the OS must change the PTBR on a context switch anyhow). In either case, the flush operation simply sets all valid bits to 0.

By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur a fair number of TLB misses as it touches data and code pages. If the OS is switching between processes frequently, this cost may be noticeable.

To overcome this cost, some systems add a little extra hardware support to enable sharing of the TLB across context switches. In particular, the hardware could 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 than that (say 8 for the ASID instead of the full 32 bits for a PID).

If we take our example TLB from above and add ASIDs (and a few other fields), we can observe that two identical VPNs for different processes can readily share the TLB; only the ASID field is needed to differentiate the two translations. With a few extra bits in each TLB entry, the OS and hardware can combine to enable entries from different processes's page tables share the TLB:

VPN	PFN	valid	prot	ASID
0	10	1	rwx	1
—	—	0	—	—
0	17	1	rwx	2
—	—	0	—	—

Thus, with address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion. Of course, the hardware also needs to know which process is currently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the currently-running process.

As an aside, you may also notice another case where two entries of the TLB are remarkably similar. In this example, there are two entries for two different processes at two different VPNs that 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 situation might arise, for example, when two processes *share* a page (a code page, for example). In the example above, process 1 is sharing physical page 101 with process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its AS. Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use.

18.4 Issue: Replacement Policy

As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. 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?

THE CRUX: TLB REPLACEMENT POLICY
Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

We will study such policies in some detail when we tackle the problem of swapping pages to disk in a virtual memory system. Here we'll just highlight a few of typical policies.

One common approach is to evict the **least-recently-used** or **LRU** entry. The idea here is to take advantage of locality in the memory-reference stream; thus, it is likely that an entry that has not recently been used is a good candidate for eviction as (perhaps) it won't soon be referenced again. Another typical approach is to use a **random** policy. Randomness sometimes makes a bad decision but has the nice property that there are not any weird corner case behaviors that can cause pessimal behavior, e.g., think of a loop accessing $n + 1$ pages, a TLB of size n, and an LRU replacement policy.

18.5 Real Code

To make sure you understand how the TLB works, let's look at some C code that accesses a large array named `a`. Figure 18.3 shows the code snippet of interest. Let's assume that integers in the array `a` are 4 bytes each, and that the page size is 4 KB. How many TLB misses does this code cause?

To answer this question, we have to first make some further assumptions. For now, let us assume that we ignore the instruction fetches that obviously must execute in order to run the code; these too could cause TLB misses (likely one, or perhaps two if the code straddle two consecutive virtual pages), but let's just focus on data accesses for now. Let's also ignore accesses to the loop variable `i`; after all, it will likely be stored in a register by a smart compiler during the looping. Thus, the only accesses we are interested in are the accesses to the array `a`.

Assuming the first integer of `a` is on the beginning of a page, we can assume this code will first generate a virtual address for the first integer, then the second, and so on. Let's assume the first access

```
// assume a[] is an array of 4-byte integers
int i;
for (i = 0; i < size; i++)
    a[i] = i;
```

Figure 18.3: TLB Hits/Misses During Array Access.

to the page causes a TLB miss, because that page has never been referenced before. The next references to integers on that same page cause TLB hits. In fact, from the loop above, we should expect the following repeating pattern: a TLB miss to the first integer in the array, followed by 1023 TLB hits for the remaining integers on the page. You could thus compute the TLB **hit rate** for this code snippet: $\frac{1023}{1024}$, or roughly 99.9%.

A quick question for you: how would you change the above code in order to generate a hit rate of 50%? How about 0%? Hint: it is easy; think about changing the amount you increment the variable `i` by, or the value of `size`.

18.6 An Example

Finally, let's briefly look at a real TLB and what is in it. This example is taken from the MIPS R4000 [H93], which is a great example of a modern system that uses software-managed TLBs. All 64 bits of this TLB entry can be seen in Figure 18.4.

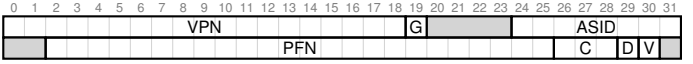

Figure 18.4: A MIPS TLB Entry.

The MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed. The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory (2^{24} 4KB pages).

There are a few other interesting bits in the MIPS TLB. We see a

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 (as described above). One question for you: what should the OS do if there are more than 256 (2^8) processes running at a time? Finally, we see 3 *Coherence* (C) bits, which determine how a page is cached by the hardware (a bit beyond the scope of these notes); a *dirty* bit which is marked when the page has been written to (we'll see the use of this later); 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; we'll see later why having larger pages might be useful. Finally, some of the 64 bits are unused (shaded gray in the diagram).

MIPS TLBs usually have 32 or 64 of these entries, most of which are used by user processes as they run. However, a few are reserved for the OS. A *wired* register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., while running the TLB miss handler).

Because the MIPS TLB is software managed, there needs to be instructions to update the TLB. The MIPS provides four such instructions: `TLBP`, which probes the TLB to see if a particular translation is in there; `TLBR`, which reads the contents of a TLB entry into registers; `TLBWI`, which replaces a specific TLB entry; and `TLBWR`, which replaces a random TLB entry. The OS uses these instructions to manage the TLB's contents. It is of course critical that these instructions are **privileged**; imagine what a user process could do if it could modify the contents of the TLB![1]

18.7 Summary

We have seen how hardware can help us make address translation faster. By providing a small, dedicated on-chip TLB as an address-translation cache, most memory references will hopefully be handled *without* having to access the page table in main memory. Thus, in the

[1] Answer: anything under the sun, including take over the machine, run arbitrary jobs or its own "OS", or make the Sun disappear.

DESIGN TIP: RAM ISN'T ALWAYS RAM
The term **random-access memory**, or **RAM**, implies that you can access any part of RAM just as quickly as another. While it is generally good to think of RAM in this way, you can probably see that because of hardware/OS features such as the TLB, accessing a particular page of memory may be more costly than you think, particularly if that page isn't currently mapped by your TLB. Thus, it is always good to remember the implementation tip: RAM isn't always RAM. Sometimes randomly accessing your address space, particular if the number of pages accessed exceeds the TLB coverage, can lead to severe performance penalties.

common case, the performance of the program will be almost as if memory isn't being virtualized at all, an excellent achievement for an operating system, and certainly essential to the use of paging in modern systems.

However, TLBs do not make the world rosy for every program that exists. In particular, 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**, and it can be quite a problem for certain systems. One solution, as we'll discuss in the next chapter, is to include support for larger page sizes; by mapping key data structures into regions of the program's address space that are mapped by larger pages, the effective coverage of the TLB can be increased. Support for large pages is often exploited by programs such as a **database management system** (a **DBMS**), which have certain data structures that are both large and randomly-accessed.

One other TLB issue worth mentioning: 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. Because of this potential problem, people have looked into all sorts of clever ways to access caches with *virtual* addresses, thus avoiding the expensive step of translation in the case of a cache hit. Such a **virtually-indexed cache** solves some performance problems, but introduces new issues into hardware design as well. See Wiggins's fine survey for more details [W03].

References

[BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization"
D. Bhandarkar and Douglas W. Clark
Communications of the ACM, September 1991
A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.

[CM00] "The evolution of RISC technology at IBM"
John Cocke and V. Markstein
IBM Journal of Research and Development, 44:1/2
A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.

[C95] "The Core of the Black Canyon Computer Corporation"
John Couleur
IEEE Annals of History of Computing, 17:4, 1995
In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.

[CG68] "Shared-access Data Processing System"
John F. Couleur and Edward L. Glaser
Patent 3412382, November 1968
The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.

[CP78] "The architecture of the IBM System/370"
R.P. Case and A. Padegs
Communications of the ACM. 21:1, 73-96, January 1978
*Perhaps the first paper to use the term **translation lookaside buffer**. The name arises from the historical name for a cache, which was a **lookaside buffer** as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a **translation lookaside buffer**. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*

[H93] "MIPS R4000 Microprocessor User's Manual".
Joe Heinrich, Prentice-Hall, June 1993
Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf

[HP06] "Computer Architecture: A Quantitative Approach"
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
A great book about computer architecture. Even better if you can find the classic first edition.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[PS81] "RISC-I: A Reduced Instruction Set VLSI Computer"
D.A. Patterson and C.H. Sequin
ISCA '81, Minneapolis, May 1981
The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.

[SB92] "CPU Performance Evaluation and Execution Time Prediction
Using Narrow Spectrum Benchmarking"
Rafael H. Saavedra-Barrera
EECS Department, University of California, Berkeley
Technical Report No. UCB/CSD-92-684, February 1992
www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf
A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.

[W03] "A Survey on the Interaction Between Caching, Translation and Protection"
Adam Wiggins
University of New South Wales TR UNSW-CSE-TR-0321, August, 2003
An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.

[WG00] "The SPARC Architecture Manual: Version 9"
David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California
Available: http://www.sparc.org/standards/SPARCV9.pdf

Homework

In this homework, you are to measure the size and cost of accessing a TLB. The idea is based on work by Saavedra-Barrera [SB92], who developed a simple but beautiful method to measure numerous aspects of cache hierarchies, all with a very simple user-level program. Read his work for more details.

The basic idea is to access some number of pages within large data structure (e.g., an array) and to time those accesses. For example, let's say the TLB size of a machine happens to be 4 (which would be very small, but useful for the purposes of this discussion). If you write a program that touches 4 or fewer pages, each access should be a TLB hit, and thus relatively fast. However, once you touch 5 pages or more, repeatedly in a loop, each access will suddenly jump in cost, to that of a TLB miss.

The basic code to loop through an array once should look like this:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

In this loop, one integer per page of the the array `a` is updated, up to the number of pages specified by `NUMPAGES`. By timing such a loop repeatedly (say, a few hundred million times in another loop around this one, or however many loops are needed to run for a few seconds), you can time how long each access takes (on average). By looking for jumps in cost as `NUMPAGES` increases, you can roughly determine how big the first-level TLB is, determine whether a second-level TLB exists (and how big it is if it does), and in general get a good sense of how TLB hits and misses can affect performance.

Here is an example graph:

As you can see in the graph, when just a few pages are accessed (8 or fewer), the average access time is roughly 5 nanoseconds. When 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. A final jump in cost occurs at around 1024 pages, at which point each access takes around 70 nanoseconds. From this data, we can conclude that there is a two-level TLB hierarchy; the first is quite small (probably holding between 8 and 16 entries); the second is larger but slower (holding roughly 512 entries). The over-

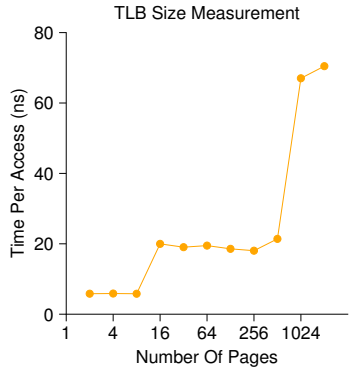

Figure 18.5: Discovering TLB Sizes and Miss Costs.

all difference between hits in the first-level TLB and misses is quite large, roughly a factor of fourteen. TLB performance matters!

Questions

- For timing, you'll need to use a timer such as that made available by `gettimeofday()`. How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)
- Write the program, called `tlb.c`, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.
- Now write a script in your favorite scripting language (csh, python, etc.) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?
- Next, graph the results, making a graph that looks similar to the one above. Use a good tool like `ploticus`. Visualization usually makes the data much easier to digest; why do you think that is?
- One thing to watch out for is compiler optimzation. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?
- Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up "pinning a thread" on Google for some clues) What will happen if you don't do this, and the code moves from one CPU to the other?
- Another issue that might arise relates to initialization. If you don't initialize the array `a` above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?