Section 7: Page Directories, Caches, and Demand Paging

CS 162

March 13, 2020

Contents

1 Vocabulary

- **Address Translation Structures** - There are two kinds you learned about in lecture: segmentation and page tables. Segments are linearly addressed chunks of memory that typically contain logically-related information, such as program code, data, stack of a single process. They are of the form (s,i) where memory addresses must be within an offset of i from base segment s. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware or more specifically to the RAM.
- **Cache** - A repository for copies that can be accessed more quickly than the original. Caches are good when the frequent case is frequent *enough* and the infrequent case is not too expensive. Caching ensures locality in two ways: temporal (time), keeping recently accessed data items 'saved', and spatial (space), since we often bring in contiguous blocks of data to the cache upon a cache miss.
- **AMAT** - Average Memory Access Time: a key measure for cache performance. The formula is (Hit Rate x Hit Time) + (Miss Rate x Miss Time) where Hit Rate + Miss Rate = 1.
- **Compulsory Miss** - The miss that occurs on the first reference to a block. This also happens with a 'cold' cache or a process migration. There's essentially nothing that you can do about this type of miss, but over the course of time, compulsory misses become insignificant compared to all the other memory accesses that occur.
- **Capacity Miss** - The miss that occurs when the cache can't contain all the blocks that the program accesses. One solution for capacity misses is increasing the cache size.
- **Conflict Miss** - The miss that occurs when multiple memory locations are mapped to the same cache location (i.e a collision occurs). In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache. These technically do not exist in virtual memory, since we use fully-associative caches.
- **Coherence Miss** - Coherence misses are caused by external processors or I/O devices that update what's in memory (i.e invalidates the previously cached data).
- **Tag** - Bits used to identify the block - should match the block's address. If no candidates match, cache reports a cache miss.
- **Index** - Bits used to find where to look up candidates in the cache. We must make sure the tag matches before reporting a cache hit.
- **Offset** - Bits used for data selection (the byte offset in the block). These are generally the lowest-order bits.
- **Direct Mapped Cache** - For a 2^N byte cache, the uppermost (32 - N) bits are the cache tag; the lowest M bits are the byte select (offset) bits where the block size is 2^M. In a direct mapped cache, there is only one entry in the cache that could possibly have a matching block.
- **N-way Set Associative Cache** - N directly mapped caches operate in parallel. The index is used to select a set from the cache, then N tags are checked against the input tag in parallel. Essentially, within each set there are N candidate cache blocks to be checked. The number of sets is X / N where X is the number of blocks held in the cache.

- **Fully Associative Cache** - N-way set associative cache, where N is the number of blocks held in the cache. Any entry can hold any block. An index is no longer needed. We compare cache tags from all cache entries against the input tag in parallel.
- **Translation Lookaside Buffer (TLB)** - A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them multiple times through page table accesses.
- **Demand Paging** - The process where the operating system only stores pages that are "in demand" in the main memory and stores the rest in persistent storage (disk). Accesses to pages not currently in memory page fault and the page fault handler will retrieve the request page from disk (paged in). When main memory is full, then as new pages are paged in old pages must be paged out through a process called eviction. Many cache eviction algorithms like least recently used can be applied to demand paging, the main memory is acting as the cache for pages which all start on disk.
- **Thrashing** - Phenomenon that occurs when a computer's virtual memory subsystem is constantly paging (exchanging data in memory for data on disk). This can lead to significant application slowdown.

2 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

Since each PTE is 4 bytes and each page contains 8KB, then a one-page page table would point to 2048 or 2^{11} pages, addressing a total of $2^{11} * 2^{13} = 2^{24}$ bytes.
Depth 1 = 2^{24} bytes
Depth 2 = 2^{35} bytes
Depth 3 = 2^{46} bytes
So in total, 3 levels of page tables are required.

List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, plus several bits – read, write, execute, and valid. This information can all fit into 4 bytes, since if physical memory is 2^{33} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation.

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

Best-case scenario: 1 memory lookup. Hit in TLB, once for actual memory operation.
Worst-case scenario: 4 memory lookups. Miss in TLB + 3 page table lookups in addition to the actual memory operation.

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:
- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61ns.

```
(10+50)*x+(1-x)*(50+10+50) = 61
solve for x gives x = .98 = 98% hit rate
```

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

```
With a TLB with a hit rate of 0.5:
x = 0.5
avg_time = (10+50)*x+(1-x)*(50+10+50)
avg_time = 85
Without a TLB:
time = 50 + 50
```

```
time = 100
```

3 Caches

3.1 Direct Mapped vs. Fully Associative Cache

An big data startup has just hired you to help design their new memory system for a byte-addressable system. Suppose the virtual and physical memory address space is 32 bits with a 4KB page size.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that uses an LRU replacement policy. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache does. What's a possible access pattern that could cause this to happen?

Let's say each cache held X amount of blocks. An access pattern would be to repeatedly iterate over $X + 1$ consecutive blocks, which would cause everything in the fully associated cache to miss every time.

3.2 Two-way Set Associative Cache

Instead, your boss tells you to build a 8KB 2-way set associative cache with 64 byte cache blocks.

How would you split a given virtual address into its tag, index, and offset numbers?

The number of offset bits is determined by the size of the cache blocks. Thus, the offset will take 6 bits, since $2^6 = 64$.

Recall that for a set associative cache, each 'set' holds N 'candidiate' blocks. Thus, to find the index we must find how many sets there are. We divide by N first to get total bytes per bank, then find how many blocks fit in each bank to get the number of blocks. Since it's two way set associative, the cache is split into two 4KB banks. Each bank can store 64 blocks, since total bytes per bank / block size = $2^{12}/2^6 = 2^6$, so there will be 6 index bits. This matches what we expect, which is that the whole cache can hold 128 blocks.

The remaining bits will be used as the tag (32-6-6 = 20).

It will look like this:

20 Bits	6 Bits	6 Bits
Tag	Index	Offset

3.3 Average Read Time

You finish building the cache, and you want to show your boss that there was a significant improvement in average read time.

Suppose your system uses a two level page table to translate virtual addresses, and your system uses the cache for the translation tables and data. Each memory access takes 50ns, the cache lookup time is 5ns, and your cache hit rate is 90%. What is the average time to read a location from memory?

Recall that page tables are held in memory as well. Since the page table has two levels, there are three reads for each access: read from first-level page table, read from second-level page table, and finally read from the physical page. Because the system also uses the cache for the translation tables, accessing a page table costs the same as going to memory. Thus, to get our final answer we should calculate the average access time, then multiply that by 3 to get the average read time.

The average access time is: $0.9 * 5 + 0.1 * (5 + 50) = 10$ns. The miss time includes the cache lookup time, as well as the time for a memory access. Since there are three accesses, we multiply this by 3 to get an average read time of 30ns.

3.4 Average Read Time with TLB

In addition to the cache, you add a TLB to aid you in memory accesses, with an access time of 10ns. Assuming the TLB hit rate is 95%, what is the average read time for a memory operation? You should use the answer from the previous question for your calculations.

If the TLB hits, we only need to read one page - the physical page mapped to in the TLB (we don't consider TLB accesses as physical memory accesses), with an access time of 10ns for a single read. Otherwise, we need to read the page table again; as in the previous part, the average read time for three accesses is 30ns. Thus, the average read time is $0.95 * (10 + 10) + 0.05 * (10 + 30) = 21$ns

4 Demand Paging

4.1 Page Replacement Algorithms

We will use this access pattern for the following section. Assume RAM has space to hold 3 pages maximum.

Page	A	B	C	D	A	B	D	C	B	A

4.1.1 FIFO

How many misses will you get with FIFO? 7 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A			D				C		
2		B			A					
3			C			B				

4.1.2 LRU

How many misses will you get with LRU? 8 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A			D						A
2		B			A			C		
3			C			B				

4.1.3 MIN

How many misses will you get with MIN? 5 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A									
2		B								
3			C	D				C		

4.1.4 FIFO vs. LRU

LRU is an approximation of MIN, which is provably optimal. Why does FIFO still do better in this case?

The LRU algorithm is based on a heuristic, trying to exploit temporal locality. It approximates MIN by assuming that the least recently used cache entry is the cache entry that will be needed at the furthest point in the future (i.e we can evict it now because 'the past is a good predictor of the future'). However, as seen in this access pattern, this is not always true.

4.2 Cached Paging

Consider a machine with a page size of 1024 bytes. There are 8KB of physical memory and 8KB of virtual memory. The TLB is a fully associative cache with space for 4 entries that is currently empty. Assume that the physical page number is always one more than the virtual page number. This is a sequence of memory address accesses for a program we are writing: 0x294, 0xA76, 0x5A4, 0x923, 0xCFF, 0xA12, 0xF9F, 0x392, 0x341.

Here is the current state of the page table.

Valid Bit	Physical Page Number
0	NULL
1	2
0	NULL
0	4
0	5
1	6
1	7
0	NULL

Explain what happens on a memory access.

First, we check the TLB. If the cached translation exists, we directly access the physical memory. If we get a TLB miss, then we must do a page walk in the page table to find an entry if it exists. If the entry is invalid or missing, we bring in the page, update our page table, and add the translation to our cache for future accesses.

How many TLB hits and page faults are there? What are the contents of the cache at the end of the sequence?

TLB hits: 5, Page Faults: 3

1. TLB miss (cold cache), PF
2. TLB miss (cold cache), PF
3. TLB miss (cold cache), hit
4. TLB hit
5. TLB miss (cold cache), PF
6. TLB hit
7. TLB hit
8. TLB hit
9. TLB hit

Valid Bit	Physical Page Number
1	1
1	2
1	3
1	4
0	5
1	6
1	7
0	NULL

Tag	Physical Page Number
0	1
2	3
1	2
3	4