Section 9: Caches & Page Replacement Algorithms

CS162

October 30th - November 1st, 2019

Contents

1 Vocabulary

1.1 Caches

Covered in Lecture 13.

- **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.

1.2 Basic Page Replacement Algorithms

Covered in Lecture 14.

- **Policy Misses** The miss that occurs when pages were previously in memory but were selected to be paged out because of the replacement policy.
- **Random** Random: Pick a random page for every replacement. Unpredictable and hard to make any guarantees. TLBs are typically implemented with this policy.
- **FIFO** First In, First Out: Selects the oldest page to be replaced. It is fair, but suboptimal because it throws out heavily used pages instead of infrequently used pages.
- **MIN** Minimum: Replace the page that won't be used for the longest time. Provably optimal. To approximate MIN, take advantage of the fact that the past is a good predictor of the future (see **LRU**).
- **LRU** Least Recently Used: Replace the page which hasn't been used for the longest time. An approximation of MIN. Not actually implemented in reality because it's expensive; see **Clock**

1.3 Advanced Page Replacement Algorithms

Covered in Lecture 14.

- **Clock** Clock Algorithm: An approximation of LRU. Main idea: replace *an* old page, not the *oldest* page. On a page fault, check the page currently pointed to by the 'clock hand. Checks a use bit which indicates whether a page has been used recently; clears it if it is set and advances the clock hand. Otherwise, if the use bit is 0, selects this candidate for replacement.

  Other bits used for Clock: "modified"/"dirty" indicates whether page must be written back to disk upon pageout; "valid" indicates whether the program is allowed to reference this page; "read-only"/"writable" indicates whether the program is allowed to modify this page.
- **Nth Chance** Nth Chance Algorithm: An approximation of LRU. A version of Clock Algorithm where each page gets N chances before being selected for replacement. The clock hand must sweep by N times without the page being used before the page is replaced. For a large N, this is a very good approximation of LRU.
- **Second Chance List** Second-Chance List Algorithm: An approximation of LRU. Divides pages into two - an active list and a second-chance list. The active list uses a replacement policy of FIFO, while the second-chance list uses a replacement policy of LRU. Not required reading, but if you're interested in the details, this algorithm is covered in detail in this paper: https://users.soe.ucsc.edu/~sbrandt/221/Papers/Memory/levy-computer82.pdf. The version presented in lecture and for the purposes of this course includes some significant simplifications.

1.4 Belady's Anomaly

The phenomenon in which increasing the number of page frames results in an increase in the number of page frames for a given memory access pattern. This is common for FIFO, Second Chance, and the random page replacement algorithm. For more information, check out http://nob.cs.ucdavis.edu/classes/ecs150-2008-02/handouts/memory/mm-belady.pdf

2 Caching

2.1 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.

2.2 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 sets. 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

2.3 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.

2.4 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

3 Page Replacement Algorithms

We will use this access pattern for the following section.

Page	A	B	C	D	A	B	D	C	B	A

3.1 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				

3.2 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				

3.3 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		

3.4 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.

3.5 Improving Cache Performance

If we increase the cache size, are we always guaranteed to get better cache performance?

3.5.1 FIFO

No; this phenomemon is known as Belady's anomaly. Increasing the cache size may actually worsen performance. The FIFO algorithm is a good example of this. See the **Belady's Algorithm** definition in the **Vocabulary** section for more details.

3.5.2 LRU

Yes. Given the same access pattern, the contents of a cache of size S is always a subset of the contents of a cache with size S + 1. This holds true for any stack algorithm; LRU is one.

3.5.3 MIN

Yes. Given the same access pattern, the contents of a cache of size S is always a subset of the contents of a cache with size S + 1. This holds true for any stack algorithm; MIN is one.

4 Clock Algorithm

4.1 Clock Page Table Entry

Suppose that we have a 32-bit virtual address split as follows:

10 Bits Table ID	10 Bits Page ID	12 Bits Offset

Assume that the physical address is 32-bit as well. Show the format of a page table entry (PTE) complete with bits required to support the clock algorithm.

20 Bits PPN	8 Bits Other	1 Bit Dirty	1 Bit Use	1 Bit Writable	1 Bit Valid

4.2 Clock Algorithm Step-through

For this problem, assume that physical memory can hold at most four pages. What pages remain in memory at the end of the following sequence of page table operations and what are the use bits set to for each of these pages?

Page	A	B	C	A	C	D	B	D	A	E	F

E: 1, F: 1, C: 0, D: 0
Recall that the clock hand only advances on page faults. No page replacement occurs until $t = 10$, when all pages are full. At $t = 10$, all pages have the use bit set. The clock hand does a full sweep, setting all use bits to 0, and selects page 1 (currently holding A) to be paged out. The clock hand advances and now points to page 2 (currently holding B). At $t = 11$, we check page 2's use bit, and since it is not set, select page 2 to be paged out. F is brought in to page 2. The clock hand advances and now points to page 3. We reach the end of the input and end.
Note: The table shows the clock hand position before page faults occur.

Page	A	B	C	A	C	D	B	D	A	E	F
1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	E: 1	E: 1
2		B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 0	F: 1
3			C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 0	C: 0
4						D: 1	D: 1	D: 1	D: 1	D: 0	D: 0
Clock	1	2	3	4	4	4	1	1	1	1	2

5 Second Chance List Algorithm

Suppose you have four pages of physical memory: 00, 01, 10, 11 and five pages of virtual memory: 000, 001, 010, 011, 101. Run the second chance list algorithm, with two physical pages delegated to the active list and two physical pages delegated to the second chance list.

The access pattern (assume we write to these pages) for virtual pages is as follows:

Page	000	001	010	011	000	101

The page table looks like this at the start of the algorithm.

Virtual Page	Physical Page	Extra
000		PAGEOUT
001		PAGEOUT
010		PAGEOUT
011		PAGEOUT
101		PAGEOUT

The 'extra' bits should read 'RW', 'INVALID', 'FIFO: 0', 'LRU: 0' where the bit for FIFO / LRU is 0 for more recent and 1 for less recent.

5.1 After Writes to '000', '001'

What does the table look like after these first two writes?

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 1
001	01	RW, FIFO: 0
010		PAGEOUT
011		PAGEOUT
101		PAGEOUT

5.2 After Writes to '010', '011'

What does the table look like after the next two writes?

Virtual Page	Physical Page	Extra
000	00	INVALID, LRU: 1
001	01	INVALID, LRU: 0
010	10	RW, FIFO: 1
011	11	RW, FIFO: 0
101		PAGEOUT

5.3 After Write to '000'

What happens when you write to virtual page 000? What does the table look like after this write?

Virtual page 000 is translated to physical page 00. However, this page is marked as invalid, so the page fault handler will be invoked. The handler will use the FIFO replacement algorithm on the active list to find a physical page to evict, and insert that page into the second-chance list. Then physical page 00 is inserted into the active list.

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 0
001	01	INVALID, LRU: 1
010	10	INVALID, LRU: 0
011	11	RW, FIFO: 1
101		PAGEOUT

5.4 After Write to '101'

What happens when you write to virtual page 101? What does the table look like after this write?

Virtual page 101 does not have a physical page assigned, so this will be a page fault. The page fault handler will use the LRU replacement algorithm on the second chance list to find a physical page to evict, then use this physical page for 101.

Next, it will use the FIFO replacement algorithm on the active list to find a physical page to evict, and insert that page into the second-chance list.

In this case, virtual page 001 is paged out, and virtual page 101 now maps to physical page 01. Finally, physical page 01 is inserted into the newly empty slot in the active list.

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 1
001		PAGEOUT
010	10	INVALID, LRU: 1
011	11	INVALID, LRU: 0
101	01	RW, FIFO: 0

6 Memory Mapped Files

Most sections did not cover this question - it may show up in a future section. Recommended reading for more context on `mmap`: https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html

Memory-mapped files, either as the fundamental way to access files or as an additional feature, are supported by most OS-s. Consider the Unix `mmap` system call that has the form (somewhat simplified):

```
void * mmap(void * addr, sizet_len, int prot, int flags, int filedes, off_t offset);
```

where data is being mapped from the currently open file *filedes* starting at the position in the file described by *offset*; *len* is the size of the part of the file that is being mapped into the process' address space; and *prot* describes whether the mapped part of the is readable or writable. The return value, *addr*, is the address in the process' address space of the mapped file region.

When the process subsequently references a memory location that has been mapped from the file for the first time, what operations happen in the operating system? What happens upon subsequent accesses?

> On the first access - page fault. The OS will commit the page, i.e map the page to a physical page. On subsequent accesses, the page of the file is in memory.

Assume that you have a program that will read sequentially through a very large file, computing some summary operation on all the bytes in the file. Compare the efficiency of performing this task when you are using conventional read system calls versus using `mmap`.

> Sequential reading with normal read library may be worse because it may read the file in larger number of chunks, and incurs multiple copy overheads for all reads. Actually, the answer is more complex and depends heavily on implementation details. You can think of various ways to tune either `mmap()` or `read()` performance, depending on what your code uses. To get a sense of the complexities involved, check out this Stack Overflow answer: http://stackoverflow.com/questions/9817233/why-mmap-is-faster-than-sequential-io

Assume that you have a program that will read randomly from a very large file. Compare the efficiency of performing this task using conventional `read` and `lseek` system calls versus using `mmap`.

> `mmap` is generally better. For truly random access patterns, they have similar performance.