CMPE 110 Homework #3

John Allard
February 23nd, 2015
jhallard@ucsc.edu

1. **Question #1 - Basic Cache**

Consider a 512-KByte cache with 32 word cache lines.This cache uses write-back scheme, and the address is 32 bits wide. (The three tables for parts A, B, and C have been condensed into a single table found below part C).

Note - Since it is not stated, I am assuming that the lower 2 bits are hard-coded as `00`, which means all accesses are word-aligned. I am also including both the word offset and byte offset in my calculation of cacheline offset, as we are supposed to do as stated in question 436 on Piazza.

(a) **Question 1.A Direct-Mapped, Cache fields** Assume the cache is direct mapped. Fill in the table below to specify the size of each address field.

Explanation - For a direct-mapped cache, we need to have a 5-bit word offset because each cache-line is 32 words wide, we then need the 2 hard-coded bits to account for byte offset within each word, for a total of 7 bits for cacheline offset. Since our cache is 52-KByte large, and each line is 32 words, we have 4,096 cache lines. The amount of bits needed to address this amount of cache-lines is 12 ($2^{12} = 4,096$). This leaves 13 bits for the tag.

(b) **Question 1.B Fully Associative Cache Fields**
Assume the cache is fully associative, fill in the table below to specify the size of each address field.

Explanation - For a fully-associative cache, we still need 7 bits for the offset, which determines which of the 32 words in a cache-line our data is stored at. Since fully-associative caches do not use an index, this leaves 25 bits left for the tag.

(c) **Question 1.C 8-Way Set-Associative, Cache Fields**
Assume the cache is 8-way associative, fill in the table below to specify the size of each address field.

Explanation - Once again, we need 7 bits for the offset because each cache-line is 32-words wide. With 512Kbyte of memory and an 8-way set-associative cache, we will need 11 bits to state which set we are looking for ($2^9 = 4,096/8$). Finally we have 16 bits left to form the tag.

Table 1: Q 1A, 1B, and 1C Calculations

Field	Size (1A)	Size (1B)	Size (1C)
Cache line offset	7	7	7
Cache line index	12	0	9
Tag	13	25	16

(d) **Question 1.D Direct Mapped Cache Transactions**

Assume the cache is direct-mapped. Fill in the table on the next page to identify the content of the cache after each of the following memory accesses. Assume the cache is initially empty (aka cold). Specify if an entry causes another line to be replaced from the cache, and if an entry has to write its data back to memory. For the data column, specify the data in the block by referring to its address like M[address]

Table 2: Q1D Calculations

Address	Request	Cacheline Ind	Valid	Modified	Tag	Data	Caused Replace	Write-back?
0x128	read	2	1	0	0	M[0x100]	0	0
0xF40	write	30	1	1	0	D[0xF00]	0	0
0xC00024	read	0	1	0	24	M[0xC00000]	0	0
0x014	write	0	1	1	0	D[0x000]	1	0
0x100F44	read	30	1	1	2	D[0x100F00]	1	1

Explanation - The system starts with a read call to a cold cache, so the Valid bit is originally zero and is 1 after the read, which has to go to memory to fetch the appropriate data. The cacheline index is 2 (decimal), representing bits 19:7 in the address. The tag is zero, which consists of bits 31:20. Next a write is called, with a cacheline index of 30 and a tag of 0 once again. This cacheline is invalid before the operation but it now valid. A writeback does not occur since another piece of data has not attempted to write to this spot yet. We then read from cacheline index 0, which is a miss (still cold). The appropriate data is pulled from memory (the 32 words starting at `M[0xC00000]` and stored in the cache, changing the valid bit to high for this line. Next we try and write to that cachline, which does cause a replace. The data is modified and still valid after the operation. Finally, we read from cacheline 30, a spot we have written to earlier, but the tags are different. This forces a write back of the data from the first read operation and causes a replace to occur.

(e) **Question 1.E 8-Way Set Associative, Cache Transactions**
Assume the cache is 8-way set-associative. Fill the table below to identify the content of the cache after each of the following memory accesses. Assume the cache is empty in the beginning (also known as cold cache). Specify if an entry causes another line to be replaced from the cache, and if an entry has to write its data back to memory. For the data column, specify the data in the block by referring to its address like M[address]. Write accesses will modify the data, so lets indicate the data after a write access with D[address].

Table 3: Q1E Calculations

Address	Request	Cacheline Ind	Valid	Modified	Tag	Data	Caused Replace	Write-back?
0x128	read	2	0	0	0	M[0x100]	0	0
0xF40	write	30	1	1	0	D[0xF00]	0	0
0xC00024	read	0	1	0	96	M[0xC00000]	0	0
0x014	write	0	1	1	0	D[0x000]	0	0
0x100F44	read	30	1	0	16	D[0x100F00]	0	0

Explanation - This sequence is very much like the one it the last problem, except that the 8-way associativity allows us to not have to replace or write back like was required in the direct-mapped cache. This is because when two addresses map to the same index, we can now put it in any of the open 8 spots per set.

(f) **Question 1.F Overhead**
What is the overhead and actual size of the direct-mapped cache? What is the overhead and actual size of the 8-way set-associative cache? Does the structure change the overhead in terms of number of memory bits?
Answer :

2. **Question #2 - Impact of Cache Access Time**
In this problem, we will be comparing various microarchitectures for a simple data cache. For all

parts, the memory requests use a 32-bit address, although you should assume that all addresses are word aligned. Since words are four bytes, this means the bottom two bits of the address will always be zero. All caches contain exactly eight cache lines, and each cache line contains four words (i.e., each cache line is 16 bytes long). Thus the total cache capacity is $8 \times 16B = 128B$

This problem will require you to identify the critical path of a specific microarchitecture. The table below lists simplified delay equations for the cache hardware components. These delay equations are parameterized by the size of each component. Delay is measured in normalized gate delays, where 1τ is the delay of a single inverter driving four identical inverters. To simplify things, assume that the delay of a component is always the same regardless of the order in which different inputs arrive at a component. More specifically, the delay of a write access is the same regardless of whether the address arrives before the write data or vice versa. Also assume that we are using combinational memories (i.e., the address is set and the data is returned on the same cycle).

(a) **Question 2.A - Sequential Tag Check then Memory Access**

The diagram on the previous page illustrates two cache microarchitectures that serialize the tag check before data access. This means that for both reads and writes, the cache completely finishes the tag check and accesses the data memory only on a cache hit. figure (a) is for a directed-mapped cache, while figure (b) is for a two-way set-associative cache. We now want to determine the critical path and cycle time in units of τ for each cache microarchitecture. As an example, the table below shows the critical path and cycle time for the directed-mapped cache from (a). Note that because we are serializing the tag check before data access and because the delay equations are the same for both read and write accesses, the critical path is the same regardless of whether we are doing a read or a write. This is a simplification, but it will do for the purposes of this part. Note that the tag is 25 bits, but each row of the tag memory requires 26 bits since it must also include a valid bit. Create a table similar to the one in the example which identifies the critical path and cycle time in units of τ for the two-way set-associative cache in (b). Compare the cycle times of the two cache microarchitectures. What is the primary reason one microarchitecture is slower than the other microarchitecture?

Table 4: Q2A Calculations

Component	Delay Eq.	Delay(τ)
addr_reg_M0	2	2
tag_decoder	$2 + 2 * 2$	6
tag_mem	$12 + (4 + 26)/32$	13
tag_cmp	$2 + 3\log_2(26)$	17
tag_and	2	2
tag_or	2	2
data_mem	$12 + (8 + 128)/32$	17
data_mux	$1 + 2\log_2(4)$	5
rdata_reg_M1	1	1
Total		65

Compare the Cycle times between the two cache microarchitectures, why is one slower than the other.

Answer : Well, one is slower than the other, but by my calculation this is only by a single cycle. The two-way set associative is one cycle faster, (clocking in at 65τ vs 66τ for the direct-mapped), and this is mostly due to the fact that it can decode half the tags in parallel, saving it 2 cycles compared to the direct-mapped, which must compare all 8 indexes with the same comparator. Accessing the tag memory saves another cycle, since we are once again searching through two halves in parallel. However, 2 of these saved cycles are un-done by the added OR-gate that compares the hit bits from both tag memories. I have a feeling that I made a mistake somewhere and the difference was supposed to be more pronounced, but in general an n-way associative

cache will have a faster look-up time than an m-way ($m < n$) associative cache because of parallelization. This will come at the expense of more complicated hardware though.

(b) **Question 2.B - Parallel Read Hit Path**

The diagram on the next page illustrates two cache microarchitectures with parallel read hit paths and pipelined write hit paths. This means that for a single read request, the tag check is done in parallel with the data memory read access, while for for a single write request the tag check is done in stage M0 and the data memory write access is done in stage M1. Figure (a) is for a directed-mapped cache, while figure (b) is for a two-way set-associative cache. For this part we will focus just on the parallel read hit path for both the direct-mapped and set-associative caches. Create two tables similar to the one from the previous part which identifies the critical path and cycle time in units of for just the parallel read hit paths. Note that since the tag check and the data memory read access are done in parallel, you will need to examine both of these paths to determine which one is in fact the critical path. Compare the cycle times of the two cache microarchitectures. What is the primary reason one microarchitecture is slower than the other microarchitecture?

Answer : The following is for the direct-mapped cache with parallel read and pipelined writes. To find the critical path, I actually had to calculate the times for 2 different paths, then compare those to find the longest. The first path I look at was the one from the `idx` line, through the `add_reg_M1`, through the `idk mux`, through the `data_decoder`, and finally through the `data_memory` and `rdata_reg_M1`. This is the path that the circuit uses to forward the index to the data memory before the tag has even been compared for correctness. I found that this path takes 39τ to finish. The second path I checked was the path that compared the tags, this path starts like the one in 2.A but doesn't end up going into the data memory (on reads). This path was the critical path and is listed below.

Table 5: Q2B Calculations, sub-part A

Component	Delay Eq.	Delay(τ)
`addr_reg_M0`	2	2
`tag_decoder`	$2 + 2 * 3$	8
`tag_mem`	$12 + (8 + 26)/32$	14
`tag_cmp`	$2 + 3\log_2(26)$	17
`tag_and`	2	2
`hit_reg_M1`	1	1
`Total`		44

The above table shows that by using a parallel structure to grab the data while also verifying it is valid reduces the time needed to grab a set of data from the cache from 66τ to 44τ, a significant improvement, at the cost of increased power draw (we read from the data memory even when we don't need to).

I then did the same calculations on the Two-Way Set Associative Cache, finding the 2 longest paths, calculating the time is takes to traverse both of them, and declaring the longest one as the critical path. For the 2-way set assoicative, I once again compared the path that is used to send the index to the data memory to the path that is used to compare the tags. The path that is used to forward the index to the data memory took 40τ units of time. The path that is used to find and compare the tags to know if the read is valid took slightly longer, at 43τ time units, so this is the critical path. This path is enumerated below :

Table 6: Q2D Calculations, sub-part B

Component	Delay Eq.	Delay(τ)
addr_reg_M0	2	2
tag_decoder	$2 + 2 * 3$	6
tag_mem	$12 + (8 + 26)/32$	13
tag_cmp	$2 + 3\log_2(26)$	17
tag_and	2	2
tag_or	2	2
hit_reg_M1	1	1
Total		43

The above table shows that once again, introducing a parellel structure for reads can significantly improve the performance of the cache, with the time needed to traverse the critical path droppign from 65τ to 43τ for this two-way set associative architecture.

Compare the Cycle times between the two cache microarchitectures, why is one slower than the other.

Answer : Once again, the two-way set associative architecture slightly edges out the direct mapped architecture in terms of time for the critical path, with a time of 43τ for the two-way and a time of 44τ for the direct mapped. This is expected, because the two-way parallelizes some of its computation it is expected that there would be some assoicated speed-up (or else why would do it), at the cost of a slightly more complicated architecture.

(c) **Question 2.C - Pipelined Write Hit Path**
For this part we will focus just on the pipelined write hit path for both the direct-mapped and set-associative caches shown on the next page. Create two tables similar to the one in the previous parts which identifies the critical path and cycle time in units of for just the pipelined write hit path. Note that since the tag check and the data memory write access happen in two different stages, you will need to examine both of these paths to determine which one is in fact the critical path. Compare the cycle times of the two cache microarchitectures. What is the primary reason one microarchitecture is slower than the other microarchitecture?

3. **Question #3 : Cache Hierarchies**
Assume a processor uses a dedicated L1 cache for instructions (IL1) and a dedicated L1 cache for data (DL1). The processor also uses a shared L2 cache that serves as an intermediate level between each of the L1 caches and memory.

The processor has a cycle time of 1 ns (i.e., operates at 1 GHz frequency). A hit to either IL1 or DL1 takes 1 cycle (time to look up the cache and return the data to processor). IL1 has 8% miss rate. DL1 has 15% miss rate. Assume load/store instructions comprise 25% of the total instructions. Hit access to L2 from either of the IL1 or DL1 caches takes 6 cycles (time to lookup the cache and return the data to either of higher level caches). L2 has total miss rate of 30%. From L2, it takes 50 cycles to access memory

(a) **Question 3.A AMAT**
What is the average memory access time?

Answer : AMAT = hit time + (miss %)*(mis penalty)
AMAT = 1 cycle + $\frac{1}{2} * [0.08 + 0.15]$*(miss penalty)
miss penalty is another layered memory access, this time between the L2 cache and main memory, so I will once again calculate the average memory access time after a miss on the L1 cache.
miss penalty = 6 + 0.3*50 (6 cycles to reach L2, 30% of the time we have to go MM which takes 50 cycles).

AMAT = 1 cycle + $\frac{1}{2} * [[.08 + .15] * [6 + 0.3 * 50]]$
AMAT = 3.415

(b) **Question 3.B No Caches**

Assume CPI = 1 if the processor has no memory stalls. Without the caches, each memory access would take 52 cycles. What is the CPI of the processor without any caches?

Answer : The CPI would be equal to $1 \times 0.75 + 50 \times 0.25 = 13.75$. This is because 75% of the time an instruction occurs which has a CPI of 1, during the 25% of the time that a load or store occurs, the CPI drops to 52. Taking a weighted average gives 13.75 as the CPI.

(c) **Question 3.C All Caches**

Assume CPI = 1 if the processor has no memory stalls. Without the caches, each memory access would take 52 cycles. What is the CPI of the processor with all of the caches? Remember it takes 50 cycles to access memory from

Answer : The CPI would be equal to $1 \times 0.75 + 0.25 \times 3.415 = 1.603$. I got this by using the AMAT calculated in part 3.A as the number of cycles it would take for each memory access, which occurs 25% of the time. The other 75% of the time the CPI remains 1. Contrasting this result with the one from problem 3.B highlights the importance of caches.

4. **Question #4 :**
Assume we have a processor that support virtual memory with 4KB page size. The processor uses one page table per program to track the mapping of virtual addresses to physical addresses, and has 24-bit virtual address. Assume we have 4-entry fully associative TLB, with the content specified in table below

Table 7:

Valid	Tag	Phys. Address
0	`0xB`	`0x1B`
1	`0x5`	`0x15`
1	`0x3`	`0x13`
1	`0x6`	`0x16`

(a) **Question 4.A - Program Table Entries**
Assume each entry in page table is 3-bytes. How many entries does the processor need to have in the page table for each program, and what is the total page table size? How about if the processor supports 16KB page size?

Answer : - Each virtual address is 3 bytes, which is 24 bits, which means it can map to 2^{24} unique addresses. Each page is 4KB, which is 2^{12} unqique addresses. This means that our page table must support $\frac{2^{24}}{2^{12}} = 2^{12} = 4096$ entries in the page table. Having this many entries gives the user access to the entire 24 bits of address space they are allocated. The total page table size would then be 4096 entries multiples by the 3 bytes per entry, so $12,288$ bytes total.

If the page size was 16KB instead of 4KB, we would then have 4 times as few entries, or 1024 entries in the page table. This is because each entry in the page table points to 4 times as many addresses, so we only need a quarter of the entries to address the entire 24 bits of address space. The page table in this case would be $3 \times 1024 = 3072$ bytes.

(b) **Question 4.B - TLB and Page Table States**
Given the address sequence and the initial state of TLB and page table, what would be the final state of the TLB and page table? Assume ideal LRU replacement for the TLB. Specify what accesses hit on TLB, what addresses hit on page table, and what addresses are page fault.

Table 8: Question 4.B - TLB Page Table

Valid	Tag	Phys. Address
1	0x0	0x1B
1	0x5	0x15
1	0x3	0x13
1	0x6	0x16

Explanation - The page table doesn't seem much different in the end then in the beginning, but it actually changed quite a bit. To start, the 0x5D10 request hit on the TLB, so it didn't need to go to the page table. The next request, B000 was in the page table but the valid bit was not set, so we needed to go to the page table. The page table also was invalid for tag B, so we had a page miss and had to go to disk. This replaced the previous first for in the TLB, setting the valid bit to high. tHe page table now also will not hit a page fault for tag B. The next call was 0x7200, which was a miss in the TLB but a hit in the page table. This value was loaded into the TLB in the 3rd row. Technically the 3rd and 4th row were equally unused, so I chose the first one in the table. The next was 0x6800, which was a hit in the TLB (4th row). The next call, 5800 was also a hit in the TLB. the 0x0000 call is a TLB miss and a miss in the page table, resulting in a page fault. After being loaded from disk, this new value replaces the first row (containing the 0xB tag), because it is the least recently used. Finally the 0x3100 request is a TLB miss, since it had been replaced earlier. We go to the page table and get a hit, replacing the 3rd row in the TLB (that contained the 0x7 tag).

(c) **Question 4.C - Two-Way Set Associative TLB**

Assume the TLB is 2-way set-associative instead of fully-associative. Given the address sequence and the initial state of TLB and page table, what would be the final state of the TLB and page table? Assume ideal LRU replacement for the TLB. Specify what accesses hit on TLB, what addresses hit on page table, and what addresses are page fault.

Table 9: Question 4.C - 2-Way Associative TLB Page Table

Valid	Tag	Phys. Address
1	0x0	0x1B
1	0x5	0x15
1	0x3	0x13
1	0x6	0x16

(d) **Question 4.D - 16KB Page Size**

(e) **Question 4.E - Trade-Offs**
Discuss the advantages and disadvantages of a larger page size.

Answer : Having a larger page size can have some positive effects on the performance for your system. For example, having a page size that is twice as large means that a program can store twice the memory in a single page, which means that a single row in the TLB and Page Table refer to twice as many addresses. This increases the likelyhood of getting a hit, which increases the overall performance of a cache. This also has tradeoffs, for instance, if we make a page size too large then we end up wasting memory for small processes. If our page size if 64KB and many programs only need 2KB of memory, we will run out of room much quicker then is needed. Also, if our page size is too large, we then have less pages to offer to different programs. This normally isn't a problem, but for instance, if the page size was 1/10th the size of the total memory, we

could only have 10 processes running simultaniously. This all implies that there is no thing as a set proper page size, this value changes as the average behavior of the user changes. If the user is constantly running very large programs, it makes sense to increase the page size at the expense of wasting memory when smaller programs are run.