CMPE 110 Computer Architecture
Fall 2015, Homework #3 Solution

Computer Engineering
UC Santa Cruz

December 1, 2015

Question 1. Instruction Cache (16 points)

Solution.

Question 1.A Catagorizing Cache Misses (4 points)

See the table below. The first instruction fetch causes a compulsory miss, but then we hit in that cache line when fetching the second addiu instruction. The jump instruction causes a conflict miss. The target of the jump instruction maps to the same set as the second cache line, so this is a conflict miss.

The first instruction fetch of the second iteration hits in the cache since that cache line was brought in on the first iteration. The cache lines at addresses 0x110 and 0x210 will continue to conflict in the cache every iteration.

Address	Instruction	Iteration 1	Iteration 2
	`loop:`		
0x108	`addiu r1, r1, -1`	*compulsory*	
0x10c	`addiu r2, r2, 1`		
0x110	`j foo`	*compulsory*	*conflict*
	`...`		
	`foo:`		
0x218	`addiu r6, r6, 1`	*conflict*	*conflict*
0x21c	`bne r1, r0, loop`		

Question 1.B Average Memory Access Latency (4 points)

Each iteration performs five instruction cache accesses. There are three misses on the first iteration, and two misses on each remaining iteration for a total of $3 + 2 \times 63 = 129$ misses. So the total miss rate is $129/320 = 0.40$.

Avg Mem Access Latency = Hit Time + (Miss Rate $\times$ Miss Penalty) = 1 + (0.40 $\times$ 5) = 3 cycles

Clearly the fixed overhead of the extra compulsory cache miss in the first iteration is largely irrelevant compared to the conflict misses in the remaining iterations of the loop.

Question 1.C Set-Associativity (6 points)

Since the majority of the instruction cache misses are conflict misses, it is possible that increasing the associativity will reduce the number of instruction cache misses. More specifically, two lines are conflicting in the same set of the direct-mapped cache, so a two-way set-associative cache will be able to keep both lines present in the cache at the same time. Of course, we need to also ensure that the third cache line doesn't map to the same set, which in this case a quick look at the addresses verifies that it does not. So a two-way set-associative instruction cache should have a miss rate close to zero on this loop. The only misses would be two compulsory misses and one conflict miss on the first iteration of the loop.

Question 2. Cache Mapping and Access (16 points)

Solution.

Question 2.A Direct-Mapped, Cache Fields (2 points)

Field	Size (bits)
Cacheline Offset	8
Cacheline Index	11
Tag	13

Offset = $\lceil \log_2 (64 \times 4) \rceil = 8$ bits. Index = $\lceil \log_2 \frac{512\ KB}{64 \times 4} \rceil = 11$ bits.

Tag = $32 - 8 - 11 = 13$ bits.

Question 2.B Fully-Associative, Cache Fields (2 points)

Field	Size (bits)
Cacheline Offset	8
Cacheline Index	0
Tag	24

Offset = $\lceil \log_2 (64 \times 4) \rceil = 8$ bits. Same as in part A. Fully-associative caches do not use the index field.

Tag = $32 - 8 = 24$ bits.

Question 2.C 16-Way Set-Associative, Cache Fields (2 point)

Field	Size (bits)
Cacheline Offset	8
Cacheline Index	7
Tag	17

Offset = $\lceil \log_2 (64 \times 4) \rceil = 8$ bits. Same as in part A. Index = $\lceil \log_2 \frac{512\ KB}{16 \times 64 \times 4} \rceil = 7$ bits.

Tag = $32 - 8 - 7 = 17$ bits.

Question 2.D Direct-Mapped, Cache Transactions (4 points)

Address	Request Type	Cachline Index	Hit or Miss?	Modified	Tag	Data	Caused Replace?	Write-back to Memory?
0x128	read	0x1	miss	0	0x0	M[0x100]	no	no
0xF40	write	0xf	miss	1	0x0	D[0xF00]	no	no
0xA00051	read	0x0	miss	0	0x14	M[0xA00000]	no	no
0x093	write	0x0	miss	1	0x0	D[0x000]	yes	no
0x4000B44	read	0xb	miss	0	0x80	M[0x4000B00]	no	no

Question 2.E 16-Way Set-Associative, Cache Transactions (4 points)

Address	Request Type	Cachline Index	Hit or Miss?	Modified	Tag	Data	Caused Replace?	Write-back to Memory?
0x128	read	0x1	miss	0	0x0	M[0x100]	no	no
0xF40	write	0xf	miss	1	0x0	D[0xF00]	no	no
0xA00051	read	0x0	miss	0	0x140	M[0xA00000]	no	no
0x093	write	0x0	miss	1	0x0	D[0x000]	no	no
0x4000B44	read	0xb	miss	0	0x800	M[0x4000B00]	no	no

Question 2.F Overhead (2 points)

Direct-mapped:

Cacheline size: no. of words × size of word × 8 bits/byte = 2048 bits

Overhead: (Tag + Valid + Modified) × no. of cache lines = $13 + 1 + 1 = 15$ bits per cacheline ×2048 cache lines.

Actual Size = (no. of cachelines × size of cacheline) + (no. of cachelines × overhead fields size) = $(2048 \times 2048) + (2048 \times 15)$

Final Answer in terms of bits: $4194304 + 30720 = 4225024$ bits

Final Answer in terms of Bytes: $512\ KB + 3.75\ KB = 515.75\ KB$

16-way Set-Associative:

Cacheline size: no. of words × size of word × 8 bits/byte = 2048 bits

Overhead: (Tag + Valid + Modified + Lowest LRU bits for 16-way replacement policy) $= 13 + 1 + 1 + 4 = 19$ bits per cacheline ×2048 cache lines.

Actual Size = (no. of cachelines × size of cacheline) + (no. of cachelines × other fields size) = $(2048 \times 2048) + (2048 \times 19)$

Final Answer in terms of bits: $4194304 + 38912 = 4233216$ bits

Final Answer in terms of Bytes: $512\ KB + 4.75\ KB = 516.75\ KB$

Yes, the structure of the cache changes the overhead because the size of the tags are different and set-associative cache needs state elements to keep track of replacement policy, whereas direct-mapped does not.

Question 3. Average Memory Access Time (12 points)

Solution.

Question 3.A Ideal System (4 points)

In an idea system with no cache misses, systems behave according to base CPI.

(Execution Time)$_B$ = IC $\times$ CPI $\times$ clock cycle time = $IC_B \times 2 \times 333\ ps$

(Execution Time)$_A$ = IC $\times$ CPI $\times$ clock cycle time = $IC_A \times 1 \times 500\ ps = (1.25 \times IC_B) \times 1 \times 500\ ps$

Speedup = (Execution Time)$_B$ / (Execution Time)$_A$

Speedup = $\frac{2 \times 333}{1.25 \times 1 \times 500} = 1.0656$

Computer A is faster by 6.56%

Question 3.B No L2 Cache (4 points)

$CPI =$ Base CPI $+ stall_{IL1} + stall_{DL1} = (missrate_{IL1} \times misspenalty_{IL1}) + (30\% \times (missrate_{DL1} \times misspenalty_{DL1})$

The miss penalty for IL1 and IL2 is to access main memory.

$CPI_A = 1 + 0.02 \times 250 + 0.3 \times 0.08 \times 250 = 12$ cycles/instruction

$CPI_B = 2 + 0.02 \times 300 + 0.3 \times 0.05 \times 300 = 12.5$ cycles/instruction

(Execution Time)$_B$ = IC $\times$ CPI $\times$ clock cycle time = $IC_B \times 12.5 \times 333\ ps$

(Execution Time)$_A$ = IC $\times$ CPI $\times$ clock cycle time = $IC_A \times 12 \times 500\ ps = (1.25 \times IC_B) \times 12 \times 500\ ps$

Speedup = (Execution Time)$_A$ / (Execution Time)$_B$

Speedup = $\frac{1.25 \times 12 \times 500}{12.5 \times 333} = 1.8018$

Computer B is faster by 80.18%

Question 3.C Full System (4 points)

CPI = Base CPI $+ missrate_{L1} \times (hittime_{L2} + missrate_{L2local} + misspenalty_{L2})$

$missrate_{L1}$ should include both IL1 and DL1 accesses.

$CPI_A = 1 + 0.02 \times (15 + 0.03 \times 250) + 0.3 \times 0.08 \times (15 + 0.03 \times 250) = 1.99$

$CPI_B = 2 + 0.02 \times (12 + 0.04 \times 300) + 0.3 \times 0.05 \times (12 + 0.04 \times 300) = 2.84$

(Execution Time)$_B$ = IC $\times$ CPI $\times$ clock cycle time = $IC_B \times 2.84 \times 333\ ps$

(Execution Time)$_A$ = IC $\times$ CPI $\times$ clock cycle time = $IC_A \times 1.99 \times 500\ ps = (1.25 \times IC_B) \times 1.99 \times 500\ ps$

Speedup = (Execution Time)$_A$ / (Execution Time)$_B$

Speedup = $\frac{1.25 \times 1.99 \times 500}{2.84 \times 333} = 1.315$

Computer B is faster by 31.5%

Question 4. Array vs. List Cache Behavior (20 points)

Solution.

				CPI Breakdown			
Question	**Number of Instructions**	**CPI**	**Execution Time (cyc)**	**Useful Work**	**RAW Stalls**	**Control Squashes**	**Memory Stalls**
4.A	256	1.50	384	1.0	0.0	0.25	0.25
4.B	256	2.25	576	1.0	0.0	0.25	1.00

Question 4.A Analyzing Performance of an Array Data Structure (7 points)

See the pipeline diagrams on the next page.

Table 1 shows the pipeline diagram for the first iteration of the loop. Notice how the very first load instruction misses in the data cache due to a compulsory miss. Since the hit latency is one cycle and the miss penalty is four cycles, the first load instruction will stall in the M stage for a total of five cycles. The second load instruction also misses in the data cache. Both store instructions hit in the data cache because they are essentially writing the same locations that we just brought into the data cache with the two load instructions. These hits illustrate how the data cache can exploit temporal locality.

To calculate how many cycles it takes to execute the first iteration we want to ignore the pipeline startup overhead, but we need to be careful how we account for the fact that the first load misses in the data cache. We want to capture the extra stall cycles due to a miss otherwise our estimate of the total execution time will be incorrect. To figure out how to do this accounting, we just need to think critically about how the pipeline diagram will look over many iterations (at least five). Students should be able to convince themselves that we need to start counting in between cycles 3 and 4 and stop counting in between cycles 21 and 22. The total number of cycles to execute the first iteration is thus 18 cycles. Of the 18 cycles, eight are due to useful work, eight are due to data cache misses, and two are due to squashing instructions after a branch misprediction. We can use these numbers to calculate the CPI breakdown.

Table 2 shows the pipeline diagram for the second iteration of the loop. Notice how now all of the load and store instructions hit in the data cache because they are accessing the two cache lines brought into the data cache in the first iteration. Since we are assuming we have a very large, fully-associative data cache there are no capacity or conflict misses. These hits illustrate how the data cache can exploit spatial locality. The total number of cycles to execute the second iteration is 10. Of the 10 cycles, eight are due to useful work, and two are due to squashing instructions after a branch misprediction.

There are 64 4-Byte elements in the array and each cache line holds four 4B elements. The arrays are cache-line aligned, so the loads in the first iteration will miss, the loads will hit in the next three iterations, and then the loads in the fifth iteration will miss again as we move onto the next cache lines. There will be 32 iteration of the loop since we stop when the pointers get to the middle of the array. Of those 32 iterations, eight will look like the first iteration and 24 will look like the second iteration. The table is filled out based on this observation and the pipeline diagrams.

The total execution time in cycles is $8 \times 18 + 24 \times 10 = 384$. Of these, $8 \times 8 = 64$ are dueto cache misses, $32 \times 2 = 64$ are due to squashing instructions after a branch misprediction. Based on these counts we can calculate the CPI breakdown.

	Dynamic Instruction	Cycle 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	`lw r12, 0(r4)`	F	D	X	M	M	M	M	M	W														
2	`lw r13, 0(r5)`		F	D	X	X	X	X	X	M	M	M	M	M	W									
3	`sw r12, 0(r5)`			F	D	D	D	D	D	X	X	X	X	X	M	W								
4	`sw r13, 0(r4)`				F	F	F	F	F	D	D	D	D	D	X	M	W							
5	`addiu r14, r5, 0`									F	F	F	F	F	D	X	M	W						
6	`addiu r4, r4, 4`														F	D	X	M	W					
7	`addiu r5, r5, -4`															F	D	X	M	W				
8	`bne r4, r14, loop`																F	D	X	M	W			
9	`opA`																	F	D	-	-	-		
10	`opB`																		F	-	-	-	-	
11	`lw r12, 0(r4)`																			F	D	X	M	W

Table 1: Pipeline Diagram For First Iteration Using Array Data Structure

	Dynamic Instruction	Cycle 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	`lw r12, 0(r4)`	F	D	X	M	W										
2	`lw r13, 0(r5)`		F	D	X	M	W									
3	`sw r12, 0(r5)`			F	D	X	M	W								
4	`sw r13, 0(r4)`				F	D	X	M	W							
5	`addiu r14, r5, 0`					F	D	X	M	W						
6	`addiu r4, r4, 4`						F	D	X	M	W					
7	`addiu r5, r5, -4`							F	D	X	M	W				
8	`bne r4, r14, loop`								F	D	X	M	W			
9	`opA`									F	D	-	-	-		
10	`opB`										F	-	-	-	-	
11	`lw r12, 0(r4)`											F	D	X	M	W

Table 2: Pipeline Diagram For Second Iteration Using Array Data Structure

Question 4.B Analyzing Performance of a Linked-List Data Structure (7 points)

	Dynamic Instruction	Cycle 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	`lw r12, 0(r4)`	F	D	X	M	M	M	M	M	W														
2	`lw r13, 0(r5)`		F	D	X	X	X	X	X	M	M	M	M	M	W									
3	`sw r12, 0(r5)`			F	D	D	D	D	D	X	X	X	X	X	M	W								
4	`sw r13, 0(r4)`				F	F	F	F	F	D	D	D	D	D	X	M	W							
5	`addiu r14, r5, 0`									F	F	F	F	F	D	X	M	W						
6	`lw r4, 4(r4)`														F	D	X	M	W					
7	`lw r5, 8(r5)`															F	D	X	M	W				
8	`bne r4, r14, loop`																F	D	X	M	W			
9	`opA`																	F	D	-	-	-		
10	`opB`																		F	-	-	-	-	
11	`lw r12, 0(r4)`																			F	D	X	M	W

Table 3: Pipeline Diagram For First Iteration Using Linked-List Data Structure

Table 3 shows the pipeline diagram for the first iteration of the loop. This pipeline diagram looks very similar to the pipeline diagram from 4A. The first and second load instructions miss, the two store instructions hit (due to temporal locality), and the third and fourth load instructions hit (due to spatial locality). We calculate the number of cycles to execute the first iteration as in the previous part. The only difference from the previous part is that all of the 32 iterations look like the first iteration because there is no opportunity to exploit spatial locality between linked-list nodes.

The total execution time in cycles is $32 \times 18 = 576$. Of these, $8 \times 32 = 256$ are due to cache misses, $32 \times 2 = 64$ are due to squashing instructions after a branch misprediction. Based on these counts we can calculate the CPI breakdown.

Question 4.C Comparison of Data Structures (6 points)

- Which data structure performs better in this specific example and why?

  The implementation of the reverse operation requires the same number of instructions (8) for both data structures, and both data structures are able to exploit temporal locality to produce cache hits for the store instructions. The key difference is that the reverse operation on the array data structure is able to exploit inter-element spatial locality since four elements are all stored on the same cache line. This results in additional cache hits for 75% of the loop iterations. This is in contrast to the reverse operation on the linked-list data structure; since linked-list nodes are all on different cache lines every iteration experiences compulsory misses when accessing the elements. The better cache behavior of the reverse operation on the array data structure results in a lower CPI and a speedup of 1.5×.

- How would the execution time change as a function of cache line size assuming we always only allocate one linked-list node per cache line?

  If we increase the cache-line size from 16B to 32B then the reverse operation on the array data structure will only experience compulsory misses every eighth iteration (instead of every fourth iteration). The overall execution time will decrease to 352 cycles. Longer cache lines facilitate exploiting greater amounts of spatial locality and will further reduce the execution time for the reverse operation on the array data structure. The reverse operation on the linked-list data structure is a different story. If we assume that there is only one linked-list node stored on each cache line (even for longer cache lines) then the performance of the reverse operation on the linked-list data structure will not change. Assuming that there is one linked-list node per cache line may or may not be a reasonable assumption; it really depends on the exact implementation of the dynamic memory allocator and how many elements we have inserted and deleted, and in what order we have done these insertions and deletions. If we insert and delete many elements then the memory can easily become fragmented such that there really is a single linked-list node per cache line. So in a realistic context, I would imagine increasing the cache-line size might reduce the number of misses a bit for the highly irregular linked-list data structure, but not nearly as much as for the highly regular array data structure.

- How would the execution time change if we assumed larger cache lines and that the memory allocator organizes multiple linked-list nodes on the same cache line?

  An aggressive dynamic memory allocator could work very hard to ensure that multiple linked-list nodes are packed into a single cache line. For example, the allocator could repack nodes after insertions and deletions, potentially even reorganizing nodes so that neighboring nodes are on the same cache line. In this case, increasing cache-line size could cause a more significant increase in performance for the linked-list data structure. However, even if all nodes are organized in sequence and packed on cache-lines as much as possible, the linked list will still have more cache misses simply because each element requires 96B instead of just 32B. In other words, the linked-list data structure will

always be spread across more cache lines than the array data structure due to the overhead of storing pointers.

- How does the execution time change as the number of elements in the data structure grows asymptotically large? How does this relate to the theoretical asymptotic behavior of $O(n)$?

  If we keep the assumptions given in the problem, then scaling to larger data structures will not change the overall comparison. Let.s calculate the execution time in cycles as a function of n for the array data structure: $\frac{n/2}{4} \times 18 + (\frac{n}{2} - \frac{n/2}{4}) \times 10 = \frac{n}{8} \times 18 - \frac{n}{8} \times 10 + \frac{n}{2} \times 10 = 6n$. Now let's do the same thing for linked-list data structure: $\frac{n}{2} \times 18 = 9n$. So for all n, the reverse operation on the array data structure will always be 50% faster than the linked-list data structure. So why does the reverse operation have the same asymptotic behavior on both data structures? This is because the overhead due to cache misses in the linked-list data structure is a constant factor overhead which is independent of n. Big-O notation captures the growth rate of a function but explicitly neglects constant factors. So while big-O notation is useful for general first-order analysis, it does not tell the whole story when we are worried about the actual performance of different algorithms and data structures on real architectures.

- How would the execution time change if both data structures were already present in the cache such that there were no cache misses?

  If all elements were already in the cache, then there would only be cache hits and the reverse operation would have the exact same performance on both data structures.

- Can we draw any broad conclusions about the cache behavior of more regular array- or matrix-based data structures vs. more irregular list-, tree-, or graph-based data-structures that make extensive use of dynamic memory allocation and pointers?

  Regular data structures usually have better cache behavior because these data structures often have significant inherent spatial locality, so it is easier for the cache microarchitecture (through choice of cache-line size, replacement policy, etc) to exploit this spatial locality for improved performance. Irregular data structures usually have worse cache behavior because they lack inherent spatial locality. Of course, these are significant generalizations. As we have already seen, dynamic memory location can play a large role and it is also possible to transform the layout of some irregular data structures to specifically improve their spatial locality.