CS433: Computer Architecture – Fall 2019
Homework 5
Total Points: Undergraduates (44 points), Graduates (52 points)
Undergraduate students should only solve the first 4 problems. Graduate students should solve all problems.
Due Date: Nov 5, 2019 at 11:00am (See course information handout for more details)

Directions:

- **All students must write and sign the following statement at the end of their homework submission. "I have read the honor code for this class in the course information handout and have done this homework in conformance with that code. I understand fully the penalty for violating the honor code policies for this class." No credit will be given for a submission that does not contain this signed statement.**
- **On top of the first page of your homework solutions, please write your name and NETID, your partner's name and NETID, and whether you are an undergrad or grad student. Also, write your NETID on each successive page.**
- **Please show all work that you used to arrive at your answer. Answers without justification will not receive credit. Errors in numerical calculations will not be penalized. Cascading errors will usually be penalized only once.**

Problem 1 [5 Points]

A 4 entry victim cache for a 4KB direct mapped cache removes 80% of the conflict misses in a program. Without the victim cache, the miss rate is 0.064 (6.4%) and 67% of these misses are conflict misses. What is the percentage improvement in the AMAT (average memory access time) due to the victim cache?

Assume a hit in the main (4KB) cache takes 1 cycle. For a miss in the main cache that hits in the victim cache, assume an additional penalty of 1 cycle to access the victim cache. For a miss in both the main and victim caches, assume a further penalty of 48 cycles to get the data from memory. Assume a simple, single-issue, 5-stage pipeline, in-order processor that blocks on every read and write until it completes.

Solution: AMAT = Hit time + Miss Rate x Miss Penalty

Without the victim cache: AMAT = 1 + 0.064*48 = 4.072 cycles

With the victim cache: AMAT = 1 + 0.064*{((0.67*0.80)*1) + ((1-(0.67*0.80))*49)} = 2.489 cycles

Improvement: (4.072-2.489)/4.072 = 38.9%

Grading: 1 point for original AMAT, 3 points for victim cache AMAT, 1 point for percent improvement. For the victim cache AMAT, 1 point for determining the correct rate for victim cache hits and misses, 1 point for assigning the correct penalty to victim cache hits, and 1 point for assigning the correct penalty to victim cache misses.

Problem 2 [12 points]

You are building a computer system around a processor with in-order execution that runs at 1 GHz and has a CPI of 1, excluding memory accesses. The only instructions that read or write data from/to memory are loads (20% of all instructions) and stores (5% of all instructions).

The memory system for this computer has a split L1 cache. Both the I-cache and the D-cache hold 32 KB each. The I-cache has a 2% miss rate and 64 byte blocks, and the D-cache is a write-through, no-write-allocate cache with a 5% miss rate and 64 byte blocks. The hit time for both the I-cache and the D-cache is 1 ns. The L1 cache has a write buffer. 95% of writes to L1 find a free entry in the write buffer immediately. The other 5% of the writes have to wait until an entry frees up in the write buffer (assume that such writes arrive just as the write buffer initiates a request to L2 to free up its entry and the entry is not freed up until the L2 is done with the request). The processor is stalled on a write until a free write buffer entry is available.

The L2 cache is a unified write-back, write-allocate cache with a total size of 512 KB and a block size of 64-bytes. The hit time of the L2 cache is 15ns for both read hits and write hits. Tag comparison for hit/miss is included in the 15ns in all cases, do not add hit time to miss time on a miss. The local hit rate of the L2 cache is 80%. Also, 50% of all L2 cache blocks replaced are dirty. The 64-bit wide main memory has an access latency of 20ns (including the time for the request to reach from the L2 cache to the main memory), after which any number of bus words may be transferred at the rate of one bus word (64-bit) per bus cycle on the 64-bit wide 100 MHz main memory bus. Assume inclusion between the L1 and L2 caches and assume there is no write-back buffer at the L2 cache. Assume a write-back takes the same amount of time as an L2 read miss of the same size.

Assume all caches in the system are blocking; i.e., they can handle only one memory access (load, store, or writeback) at a time. When calculating the miss penalty for a load or store for a writeback cache, the time for any needed writebacks should be included in the miss penalty.

While calculating any time values (such as hit time, miss penalty, AMAT), please use ns (nanoseconds) as the unit of time. For miss rates below, give the **local miss rate** for that cache. By **miss penalty$_{L2}$,** we mean the time from the miss request issued by the L2 cache up to the time the data comes back to the L2 cache from main memory.

Part A [7 points]

Computing the AMAT (average memory access time) for *instruction accesses*.

i. Give the values of the following terms for instruction accesses: **hit time$_{L1}$, miss rate$_{L1}$, hit time$_{L2}$, miss rate$_{L2}$.** [1 point]

Solution:

hit time$_{L1}$ = 1 processor cycle = 1 ns
miss rate$_{L1}$= 0.02

hit time$_{L2}$ = 15 ns
miss rate$_{L2}$ = 1 – 0.8 = 0.2

Grading: 1 point for giving the correct values of all 4 terms, otherwise no points. This part is just meant to get everyone started on the right track.

ii. Give the formula for calculating **miss penalty$_{L2}$**, and compute the value of miss penalty L2. [4 points]

Solution: miss penalty$_{L2}$ = memory access latency + time to transfer one L2 cache block

Transfer rate of memory bus = 64 bits / bus cycle = 64 bits / 10 ns = 8 bytes / 10 ns = 0.8 bytes / ns

Time to transfer one L2 cache block = 64 bytes / 0.8 bytes = 80 ns.

So, miss penalty$_{L2}$ = 20 + 80 = 100 ns

However, 50% of all replaced blocks are dirty and so they need to be written back to main memory. This takes another 100 ns.

Therefore, miss penalty$_{L2}$ = 100 + 0.5 x 100 = 150 ns.

Grading: 1 point for the correct formula for miss penalty$_{L2}$. 1 point for correctly setting up the time to transfer one block. 1 point for the correct value of miss penalty$_{L2}$ prior to accounting for write backs. 1 point for noting that 50% of the time the replaced block will need to be written back, and for correctly setting up the value of miss penaltyL2 taking into account this write back time. No points to be taken off for calculation errors.

iii. Give the formula for calculating the AMAT for this system using the five terms whose values you computed above and any other values you need. [1 point]

Solution: AMAT = hit time$_{L1}$ + miss rate$_{L1}$ x (hit time$_{L2}$ + miss rate$_{L2}$ x miss penalty$_{L2}$)

Grading: 1 point for a completely correct formula, otherwise no points.

iv. Plug in the values into the AMAT formula above, and compute a numerical value for AMAT for instruction accesses. [1 point]

Solution: AMAT = 1 + 0.02 x (15 + 0.2 x 150) = 1.9 ns.

Grading: 1 point for setting up the correct values in AMAT formula. No points to be taken off for calculation errors.

Part B [2 points]

Computing the AMAT for *data reads*.

i. Give the value of **miss rate$_{L1}$** for data reads. [1 point]

Solution: miss $rate_{L1}$ = 0.05

Grading: 1 point for giving the correct value of miss $rate_{L1}$.

ii. Calculate the value of the AMAT for data reads using the above value, and other values you need. [1 point]

Solution: AMAT = hit $time_{L1}$ + miss $rate_{L1}$ x (hit $time_{L2}$ + miss $rate_{L2}$ x miss $penalty_{L2}$)

AMAT = 1 + 0.05 x (15 + 0.2 x 150) = 3.25 ns

Grading: 1 point for setting up the correct AMAT formula for data reads. No points to be taken off for calculation errors.

Part C [3 points]

Computing the AMAT for *data writes*. Assume miss $penalty_{L2}$ for a data write is the same as that computed previously for a data read.

i. Give the value of **write $time_{L2Buff}$**, the time for a write buffer entry to be written to the L2 cache. [2 points]

Solution: As the L2 cache hit rate is 80%, only 20% of the write buffer writes will miss in the L2 cache and incur the miss penaltyL2.

So, write $time_{L2Buff}$ = hit $time_{L2}$ + 0.2 x miss $penalty_{L2}$

Write $time_{L2Buff}$ = 15 + 0.2 x 150 = 45 ns

Grading: 1 point for setting up the correct formula for write $time_{L2Buff}$, and setting up the correct values in write $time_{L2Buff}$ formula. No points to be taken off for calculation errors.

ii. Calculate the value of the AMAT for data writes using the above information, and any other values that you need. Only include the time that the processor will be stalled. Hint: There are two cases to be considered here depending upon whether the write buffer is full or not. [1 point]

Solution: There are two cases to consider here. In 95% of the cases the write buffer will have empty space, so the processor will only need to wait 1 cycle. In the remaining 5% of the cases, the write buffer will be full, and the processor will have to wait for the additional time taken for a buffer entry to be written to the L2 cache, which is write timeL2Buff.

AMAT = hit $time_{L1}$ + 0.05 x write $time_{L2Buff}$ = 1 + 0.05 x (45) = 3.25 ns

Grading: 1 point for setting up the correct AMAT equation, and setting up the correct values in equation of AMAT. No points to be taken off for calculation errors.

Problem 3 [13 points]

Consider the following piece of code:

```
register int i, j;  /* i, j are in the processor registers */
register float sum1, sum2;
float a[64][64], b[64][64];

for (i = 0; i < 64; i++) {                  /* 1 */
        for (j = 0; j < 64; j++) {          /* 2 */
                sum1 += a[i][j];            /* 3 */
        }
        for (j = 0; j < 32; j++) {          /* 4 */
                sum2 += b[i][2*j];          /* 5 */
        }
}
```

Assume the following:

• There is a perfect instruction cache; i.e., do not worry about the time for any instruction accesses.

• Both *int* and *float* are of size 4 bytes.

• Only the accesses to the array locations a[i][j] and b[i][2*j] generate loads to the data cache. The rest of the variables are all allocated in registers.

• Assume a fully associative, LRU data cache with 32 lines, where each line has 16 bytes.

• Initially, the data cache is empty.

• To keep things simple, we will assume that statements in the above code are executed sequentially. The time to execute lines (1), (2), and (4) is 4 cycles for each invocation. Lines (3) and (5) take 10 cycles to execute and an additional 40 cycles to wait for the data if there is a data cache miss.

• There is a data prefetch instruction with the format prefetch(array[index]). This prefetches the entire block containing the word array[index] into the data cache. It takes 1 cycle for the processor to execute this instruction and send it to the data cache. The processor can then go ahead and execute subsequent instructions. If the prefetched data is not in the cache, it takes 40 cycles for the data to get loaded into the cache.

• The arrays **a** and **b** are stored in row major form.

• The arrays **a** and **b** both start at cache line boundaries.

Part A [2 points]

How many cycles does the above code fragment take to execute if we do NOT use prefetching?

Solution: Each line has 4 values, so every 4 accesses in line 3 will miss, and every 2 in line 5, for a total of 64*(16+16) = 2048 misses.

Line 1 executes 65 times, 65*4 = 260
Line 2 executes 64*65 times, 64*65*4 = 16640
Line 3 executes 64*64 times, 64*64*10 = 40960 (leaving misses for later)
Line 3 misses 64*64/4 times, 64*64/4*40 = 40960
Line 4 executes 64*33 times, 64*33*4 = 8448
Line 5 executes 64*32 times, 64*32*10 = 20480 (leaving misses for later)
Line 5 misses 64*32/2 times, 64*32/2*40 = 40960
Total cycles = 168708

Also calculate the average number of cycles per outer-loop iteration: 2636.0625.

Grading: 1 point for correct cycles taken by lines 3 and 5. 1 point for correct cycles taken by lines 1, 2, 4.

Part B [2 points]

Consider inserting prefetch instructions for the two inner loops for the arrays **a** and **b** respectively. Explain why we may want to unroll the loops to insert prefetches. What is the minimum number of times you would need to unroll for each of the two loops for this purpose?

Solution: There is one miss every four iterations of the first loop, and every two iterations of the second loop. The latency of this miss covers the prefetch time. We only need to issue a prefetch instruction once per cache line accessed. If code size is not a problem, unrolling the loop is the most efficient way to do this (it avoids branches that test for the correct iteration count). The first loop would need to be unrolled 4 times, and the second two times.

Grading: 1 point for unroll count for loop 1, 1 point for unroll count for loop 2.

Part C [4 points]

Unroll the inner loops for the number of times identified in part b, and insert the minimum number of software prefetches to minimize execution time. The technique to insert prefetches is analogous to software pipelining. You do not need to worry about startup and cleanup code and do not introduce any new loops.

Solution:

```
register int i, j; /* i, j are in the processor registers */
```

```
register float sum1, sum2, a[64][64], b[64][64];
for (i = 0; i < 64; i++) {                          /* 1 */
        for (j = 0; j < 64; j +=4) {        /* 2 */
                prefetch(a[i][j+4]);        /* P1 */
                sum1 += a[i][j];            /* 3a */
                sum1 += a[i][j+1];          /* 3b */
                sum1 += a[i][j+2];          /* 3c */
                sum1 += a[i][j+3];          /* 3d */
        }
        for (j = 0; j < 32; j += 2) {       /* 4 */
                prefetch(b[i][2*j+8];       /* P2 */
                sum2 += b[i][2*j];          /* 5a */
                sum2 += b[i][2*j+2];        /* 5b */
        }
}
```

Grading: 1 point for prefetch in first loop. 1 point for prefetch in second loop. 1 point in each loop for correct indices statements and loop header.

Part D [2 points]

How many cycles does the code in part (c) take to execute? Calculate the average speedup over the code without prefetching. Assume prefetches are not present in the startup code. Extra time needed by prefetches executing beyond the end of the loop execution time should not be counted.

Solution: Now the only misses are on the very first execution of line 3a (row major ordering means prefetching is effective even across outer iterations), and the first two executions of line 5a (the prefetch is preparing for the j+2 iteration). There are 3 misses total.

Line 1 executes 65 times, 65*4 = 260
Line 2 executes 64*17 times, 64*17*4 = 4352
Line P1 executes 64*16 times, 64*16*1 = 1024
Lines 3a-3d each execute 64*16 times, 64*16*4*10 = 40960
Line 3a misses only on its every first execution. 40*1 = 40
Line 4 executes 64*17 times, 64*17*4 = 4352
Line P2 executes 64*16 times, 64*16*1 = 1024
Lines 5a, 5b each execute 64*16 times. 64*16*2*10 = 20480
Line 5a misses on the first two executions. 40*2 = 80
Total cycles = 72572

The speedup over the code with no prefetching is 168708/72572, approximately 2.32.

Grading: 1 point for calculating correct number of misses. 1 point for the rest of the cycles.

Part E [3 points]

Is there another technique that can be used to achieve the same objective as loop unrolling in this example, but with fewer instructions? Explain this technique and illustrate its use for the code in part (c).

Solution: The simplest option is to issue excess prefetch requests. Costing only one cycle if the data has already been requested, that's probably cheaper than trying to use branches.

```
for (i = 0; i < 64; i++) {                  /* 1 */
        for (j = 0; j < 64; j++) {          /* 2 */
                prefetch(a[i][j+4]);
                sum1 += a[i][j];            /* 3 */
        }
        for (j = 0; j < 32; j++) {          /* 4 */
                prefetch(b[i][2j+8]);
                sum2 += b[i][2*j];          /* 5 */
        }
}
```

Grading: 1.5 points for prefetching in each loop. Code may also test j%4 (resp. j%2) to only issue the same number of prefetches as in the unrolled code.

Alternate solution with test j%4 (resp. j%2):

```
for (i = 0; i < 64; i++) {                          /* (1) */
        for (j = 0; j < 64; j++) {                  /* (2) */
                if (j%4 == 0)                       /* (3-0) */
                        prefetch(a[i][j+4]);        /* (3-1) */
                sum1 += a[i][j];                    /* (3-2) */
        }
        for (j = 0; j < 32; j++) {                  /* (4) */
                if (j%2 == 0)                       /* (5-0) */
                        prefetch(b[i][2*(j+2)]);    /* (5-1) */
                sum2 += b[i][2*j];                  /* (5-2) */
        }
}
```

Grading: 1.5 points each for inserting the correct if statement in each of the two inner loops.

Problem 4 [14 points]

Way prediction allows an associative cache to provide the hit time of a direct-mapped cache. The MIPS R10000 processor used way prediction to achieve a different goal: reduce the cost of the chip package. The R10000 hardware includes an on-chip L1 cache, on-chip L2 tag comparison circuitry, and an on-chip L2 way prediction table. L2 tag information is brought on chip to detect an L2 hit or miss. The way prediction table contains 8K 1-bit entries, each corresponding to two L2 cache blocks. L2 cache storage is built external to the processor package, is 2-way associative, and may have one of several block sizes.

Part A [2 points]

How can way prediction reduce the number of pins needed on the R10000 package to read L2 tags and data, and what is the impact on performance compared to a package with a full complement of pins to interface to the L2 cache?

Solution: With way prediction, only one way is read at a time rather than reading and comparing both. The package only needs enough pins to read the tag and data from a single line in a cycle instead of two, plus one extra bit to select the way. (Assuming the cache is just simple memory). A cache access takes an extra cycle whenever the way prediction is incorrect, and on every cache miss, which will slow performance.

Grading: 2 points for observing that we have to access both ways simultaneously if we don't have way prediction.

Part B [2 points]

How could a 2-associative cache be implemented with the same smaller number of pins but without the way prediction table? What is the performance drawback?

Solution: Without way prediction, the processor will access the ways sequentially. This will incur a delay whenever the data is found in the second way, and prediction would have been accurate.

Grading: 1 point for suggesting sequential access. 1 point for describing the performance loss.

Part C [4 points]

Assume that the R10000 uses most-recently used way prediction. What are reasonable design choices for the cache state update(s) to make when the desired data is in the predicted way, the desired data is in the non-predicted way, and the desired data is not in the L2 cache? Please fill in your answers in the following table.

Cache Access Case	Cache State Change Way Prediction Entry
Desired data is in the predicted way	No change
Desired data is in the non-predicted way	Flip state (to the way for this access)
Desired data is not in the L2 cache	Set to location of new data, or, Flip State (state will be flipped if we overwrite least recently used way)

Grading: 2 points per entry.

Part D [2 points]

For a 1024 KB L2 cache with 64-byte blocks and 8-way set associativity, how would the prediction table be organized for this new size? Give your answer in the form of "X entries by Y bits per entry."

Solution: (1024KB / 64 bytes per line) / 8 ways per set = 2048 sets. For 8-way associativity 3 bits are needed ($\log_2(8)$). So the table would be 2048 (2K) entries by 3 bits.

Grading: 2 points for correct organization. 1 point if only number of entries (sets) or entry width is correct.

Part E [2 points]

For an 8 MB L2 cache with 128-byte blocks and 2-way set associativity, what would the prediction table organization be? Again, give your answer as "X entries by Y bits per entry."

Solution: (8MB / 128 bytes per line) / 2 ways per set = 32K sets. Two-way associativity needs 1-bit entries, so the table is 32K entries by 1 bit.

Grading: 2 points for correct answer. 1 point if only number of entries or entry width is correct.

Part F [2 points]

What is the difference in the way that the R10000 with only 8K way prediction table entries will support the cache in part d) versus the cache in part e)? Hint: Think about the similarity between a way prediction table and a branch prediction table.

Solution: An 8Kb way prediction table is enough to support the cache in part D directly, or maybe even to treat the 8Kb as a smaller number of more sophisticated predictors. For the cache in part E, several ways will have to share a predictor, using some map from 32K sets onto 8K predictors. For best results, the mapping should aim to ensure that memory that will be used at similar times will be mapped to different ways, for example simply by dropping some high bits.

Grading: 1 point for saying there are enough entries for part D. 1 point for suggesting ways share predictors in part E.

NOTE: ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Problem 5 [8 points]

Consider a computer with an in-order CPU, and with a data cache block size of 64 bytes (16 words) and a 32-bit wide bus to the memory. The memory takes 10 cycles to supply the first word and 2 cycles per word to supply the rest of the block. The cache is non-blocking, and it can support any number of outstanding misses. The memory can service multiple requests simultaneously if required (techniques to achieve this will be discussed in class).

This cache and memory system implement a "**Requested Word First and Early Restart**" policy, and the bus delivers the block data in "**cyclic order"** starting with the requested word. Cyclic order means that if the requested word is the 5th in a block of size 16 words, then the order in which the words in the block are supplied is 5, 6, 7 … 16, 1, 2, 3, 4.

Part A [3 points]

Consider the following code fragment, which operates on an integer array A which is block-aligned (that is A[0] is located at the start of a cache block in memory):

```
for (i = 11; i < 100; i += 16) {/* 1 */
        A[i] *= 2;              /* 2 */
}
```

Suppose that the cache is big enough so that there are only compulsory misses. Further, statement 1 takes 4 cycles to execute, and statement 2 takes 4 cycles to execute in addition to any miss latency. Assume no overlap in the execution of these statements. Initially, the array A is not present in the cache, so any initial accesses to A cause misses in the cache.

What is the running time of this loop with the "*Requested Word First and Early Restart*" policy?

Solution: Statement 1 executes 7 times = 28 cycles

A[i] is supplied after 10 cycles. Thus, the time taken for statement 2 = (10 + 4)*6 = 84 cycles.

Total time = 112 cycles

Grading: 1 point for statement 1, 2 points for statement 2.

Part B [3 points]

How many cycles would the above loop take to run in a system with just "*Early Restart*" (i.e. the block is fetched in normal order, but the program is started early at arrival of requested word).

Solution: Statement 1 executes 7 times = 28 cycles

A[i] is supplied after (10 + 2*11) = 32 cycles (it is the 11th word in the block). Thus, the time taken for statement 2 = (32 + 4)*6 = 216 cycles.

Total time = 244 cycles

Grading: 2 points for data supply cycles, 1 point for statement execution time.

Part C [2 points]

How many cycles would the above loop take to run in a system with the base policy (i.e. normal fetch and restart)?

Solution: Statement 1 executes 7 times = 28 cycles

Block is supplied after (10 + 2*15) = 40 cycles. Thus, time taken for statement 2 = (44)*6 = 264 cycles.

Total time = 292 cycles

Grading: 1 point for data supply cycles, 1 point for statement execution time.