CS 433 Homework 4

Assigned on 10/17/2017
Due in class on 11/7/2017

Instructions:

1. Please write your name and NetID clearly on the first page.
2. Refer to the course fact sheet for policies on collaboration.
3. Due **IN CLASS** on 11/7/2017.

Problem 1 [7 Points]

Consider a 4MB 8-way set-associative write-back cache with 64 byte block size for byte-addressable memory with 32-bit address. Assume a random replacement policy and a single core system.

Part A [2 points]

Which bits of the address counting from 0 as the LSB are used for the cache index?

Number of bytes per block = $64 = 2^6$
Number of cache blocks = $\frac{2^{22}}{2^6} = 2^{16}$
Number of blocks per way = $\frac{2^{16}}{8} = 2^{13}$

Number of bits for offset within a block = 6
Number of bits for index = 13

Bits 0-5 are used for the offset and bits 6-18 are used for indexing.

Part B [2 points]

Which bits of the address are used for the cache tag?

Bits 19-31 are used for the tag.

Part C [3 points]

How many bits of total storage does this cache need besides the 4MB for data? Remember to include any state bits needed.

13 bits for the tag, 1 bit for valid/invalid and 1 dirty bit for each cache line. This amounts to 15×2^{16} bits or 120 KB.

Problem 2 [16 points]

Consider a processor where each instruction takes, on average, 2 cycles and there are 1.5 references to memory per instruction. A program with 100,000 instructions is executed on this machine using a split cache of 32KB, obtaining a 95% hit rate, 2 ns hit time and an 18 ns miss penalty. Then, the same program is executed using a 64KB cache, resulting in a hit rate of 97%, a hit time of 3 ns and the same miss penalty that in the previous case. The cycle time of the processor is adjusted to match the cache hit latency.

Part A [1 point]

Explain why the larger cache has higher hit rate.

The larger cache can eliminate the capacity misses.

Part B [1 points]

Explain why the small cache has smaller access time (hit time).

The smaller cache requires lesser hardware and overheads, allowing a faster response.

Part C [4 points]

Calculate the AMAT for both cases. Which cache is better from this point of view?

AMAT = Hit time + Missrate * Miss penalty
For the smaller cache, AMAT $= 2 + 0.05 \times 18 = 2.9$ ns
For the larger cache, AMAT $= 3 + 0.03 \times 18 = 3.54$ ns

The 32 KB cache is better in terms of AMAT.

2 points for each AMAT.

Part D [4 points]

Calculate the CPI for both cases. Which cache is better from this point of view?

CPI = Nominal CPI + Miss penalty
For the smaller cache,
Cycle time = 2 ns
Miss penalty in cycle = 18 / 2 = 9 cycles
CPI $= 2 + 1.5 \times 0.05 \times 9 = 2.675$
For the larger cache,
Cycle time = 3 ns
Miss penalty in cycle = 18 / 3 = 6 cycles
CPI $= 2 + 1.5 \times 0.03 \times 6 = 2.27$

The 64 KB cache is better in terms of CPI.

2 points for each CPI.

Part E [6 points]

Calculate the execution time for both cases. Which cache is better? Please find the speedup.

Execution time = # of instructions * CPI * Cycle time
For the smaller cache $= 100000 \times 2.675 \times 2\,ns = 535\,\mu$s
For the larger cache $= 100000 \times 2.27 \times 3\,ns = 681\,\mu$s

The smaller cache is 22.5% faster.

3 points for each execution time.

Problem 3 [20 points]

Consider the following piece of code:

```
register int i, j;                /* times 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 4 bytes.
- Assume that 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.
- The arrays a and b are stored in row major form.
- 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.
- Assume that the arrays a and b both start at cache line boundaries.

Part A [4 points]

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

Each line has 4 values, so every fourth access in line 3 will miss, and every other in line 5, for a total of $64 \times (16 + 16) = 2048$ misses.

Line 1 executes 65 times, $65 \times 4 = 260$
Line 2 executes 64×65 times, $64 \times 65 \times 4 = 16640$
Line 3 executes 64×64 times, $64 \times 64 \times 10 = 40960$ (leaving misses for later)
Line 3 misses $64 \times \frac{64}{4}$ times, $64 \times \frac{64}{4} \times 40 = 40960$
Line 4 executes 64×33 times, $64 \times 33 \times 4 = 8448$
Line 5 executes 64×32 times, $64 \times 32 \times 10 = 20480$ (leaving misses for later).
Line 5 misses $64 \times \frac{32}{2}$ times, $64 \times \frac{32}{2} \times 40 = 40960$
Total cycles = 168708

Part B [4 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?

Ideally, prefetch instructions should be separated enough so that the previous prefetch receives the data before starting the next prefetch. However, the original loop body is too small. There is one miss every fourth iteration of the first loop, and every other iteration of the second loop. Assuming prefetching eliminates cache misses, thus, each loop body takes only 10 cycles which is not enough to hide prefetching latency of 40 cycles. We can increase the size of the loop body by applying loop unrolling. The first loop would need to be unrolled 4 times, and the second two times for this purpose.

2 points for the reason for loop unrolling; 1 point for the correct minimum number of unrolling for each loop;

Part C [8 points]

Unroll the inner loops for the number of times identified in part b, and insert the minimum number of prefetch instructions 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.

```
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) */
   }
}
```

2 points for correct index for each prefetch instruction; 3 points for correct loop unrolling of each loop;

Part D [4 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.

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

The speedup over the code with no prefetching is $\frac{168708}{72572}$, approximately 2.32.

2 points for correctly counting the number of misses; 1 point for correctly computing the overhead of each prefetch instruction;

Problem 4 [12 points]

Consider a virtual memory system where a TLB access takes 2 ns and there is a single level of a set-associative, write-back data cache with the following parameters:

- indexing the cache to access the data portion takes 6 ns
- indexing the tag array of the data cache takes 4 ns
- tag comparisons take 1.5 ns
- multiplexing the output data takes 1 ns

Assume these are the only parts that affect the cache access time. For each of the following configurations, calculate the amount of time it takes to get data from the cache on a hit, including any necessary TLB access time. Please explain your answers for full credit.

Part A [3 points]

The page size is 4KB and the data cache is 64 KB, 4-way set associative, with block size of 16 bytes. The cache is physically-indexed and physically-tagged.

Bits required for indexing = $\log \frac{2^{16}}{4 \times 16} = 10$
Bits required for block offset = 4
Total bits required to access the cache = 14
Total bits for the page offset = 12

So physical indexing cannot proceed before translation.

The virtual address must be translated before anything else can happen. So the 2 ns delay of the TLB access must be added into the total. Accessing the data and tag arrays may occur in parallel. The data values are available after $2 + 6 = 8\,ns$ and the tag hit signal appears at $2 + 4 + 1.5 = 7.5\,ns$. At this point, the correct way can be selected from the mux, which has an additional 1ns delay. Therefore, the total access time is $9\,ns$.

1 point for performing indexing after translation; 1 point for indexing in parallel with translation; 1 point for multiplexing and getting the correct answer;

Part B [3 points]

Same as the part A except that the cache is virtually-indexed and virtually-tagged.

In a virtually-indexed, virtually-tagged cache, the TLB is not used unless there is a cache miss, at which point translation would be required. Besides the TLB, the access is served in the same manner as above. Thus, the access time will be the part A's time minus the TLB access time, which is 9 - 2 = 7 ns.

1 point for noting that translation is unnecessary; 1 point for index and tag path times; 1 point for multiplexing and getting the correct answer;

Part C [3 points]

Same as the part A except that the cache is virtually-indexed and physically-tagged.

Physical tag comparison requires TLB translation. However, since the index is virtual, it can be sent to the cache to initiate the tag array and data array accesses while the TLB translation is happening. Since they take longer than the TLB access, the TLB access time will be hidden and not on the critical path. After $4\,ns$, the tags will be sent to the comparator with the translated tag from the TLB. After $5.5\,ns$ the tag hit signal will be generated. Accessing the data array occurs in parallel and will happen at $6\,ns$. After a 1 ns delay from multiplexing the data, the data will be available at $7\,ns$.

1 point for noting that translation is needed for tag but not on the critical path; 1 point for the intermediate calculations; 1 point for multiplexing and getting the correct answer;

Part D [3 points]

Same as the part A except that the cache is physically-indexed and virtually-tagged.

The index is physical and requires translation by the TLB. This results in a $2\,ns$ delay in translating the index. Although the virtual tag is available at time zero, nothing can be done with it until an index is available. Once the index is translated, it can be used to access the tag and data arrays. Data is available at $8\,ns$ while the tag hit signal comes at $7.5\,ns$. Adding the $1\,ns$ delay from multiplexing the data, the total access time is $9\,ns$.

1 point for noting that translation is needed for index; 1 point for the intermediate calculations; 1 pt for multiplexing and getting the correct answer;

Problem 5: Graduate students only [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 total instructions) and stores (5% of total instructions).

The memory system for this computer has a split L1 cache. Both the I-cache and the D-cache are direct-mapped and 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. 90% of writes to L1 find a free entry in the write buffer immediately. The other 10% of the writes have to wait until an entry frees up in the write buffer. An entry is held while the write buffer initiates a request to L2 and waits for L2. 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 12 ns for both read hits and write hits. L2 write for a miss takes 12 ns (after the allocate is done). Tag comparison for hit/miss is included in the 12 ns 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 20 ns (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 a read of the same size from memory.

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 penaltyL2, 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]

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

1. Give the values of the following terms for instruction accesses. L1 hit time, L1 miss rate, L2 hit time, and L2 miss rate. [1 point]

(L1 hit time) = 1 ns
(Cycle time) = 1 ns
(L1 miss rate) = 0.02
(L2 hit time) = 12 ns
(L2 miss rate) = $1 - 0.8 = 0.2$

2. Give the formula for calculating L2 miss penalty, and compute the value of L2 miss penalty. Don't forget to include the time to write back a dirty block. [4 points]

(L2 miss penalty) = (read a new block from memory) + 0.5 * (write back a dirty block to memory)

Memory transfer between L2 and memory takes the same amount of time regardless of read or write.

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

(Time to transfer a L2 cache block) = 64 / 0.8 = 80 ns

(read a new block from memory) = (write back a dirty block to memory) = 20 + 80 = 100 ns

(L2 miss penalty) = 100 + 0.5 * 100 = 150 ns

2 point for the correct formula for L2 miss penalty; 1 point for time to transfer a block; 1 pt for the remaining part;

3. 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]

> AMAT = (L1 hit time) + (L1 miss rate) x ((L2 hit time) + (L2 miss rate) x (L2 miss penalty))

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

> AMAT = 1 + 0.02 x (12 + 0.2 x 150) = 1.84 ns.

Part B [2 points]

Compute the AMAT for data reads.

1. Give the value of L1 miss rate for data reads. [1 point]

> (L1 miss rate) = 0.05

2. Calculate the value of the AMAT for data reads. [1 point]

> AMAT = (L1 hit time) + (L1 miss rate) x ((L2 hit time) + (L2 miss rate) x (L2 miss penalty))
> = 1 + 0.05 x (12 + 0.2 x 150)
> = 3.1 ns

Part C [3 points]

Compute the AMAT for data writes. Assume the miss penalty for a data write is the same as computed previously for a data read.

1. Give the value of time for a write buffer entry being written to the L2 cache. [2 points]

> As the L2 cache hit rate is 80%, only 20% of the write buffer writes will miss in the L2 cache and will thus incur the L2 miss penalty.
>
> (Write buffer time) = (L2 hit time) + 0.2 x (L2 miss penalty) = 1 x 12 + 0.2 x 150 = 42 ns

2. 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]

There are two cases to consider here. In 90% of the cases the write buffer will have empty space, so the processor will only need to wait 1 cycle. In the remaining 10% 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 computed above.

AMAT = (L1 hit time) + 0.1 x (write buffer time) = 1 + 0.1 x 42 = 5.2 ns