Full Name:_____

15-740/18-740, Spring 2017

Exam 1

March 9, 2017, 3:00pm-4:20pm

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer. A few pages of scratch paper are provided at the end of the text booklet, but your final answer should be written by the question.
- Show your work and discuss your answer. You will be graded more on your explanation than on your final answer.
- The exam has a maximum score of 20,000 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is CLOSED BOOK, CLOSED NOTES. You may use a calculator, but no networked devices (e.g., phones, laptops, etc.).

Do not write below this line

Problem	Your Score	Possible Points
1		8,000
2		5,000
3		7,000
Total		20,000

Trade-offs

Problem 1. (8000 points total):

In this problem, you will help Alice and Bob assess the performance of different architectural and programming optimizations. You may leave your answers as unreduced arithmetic if they are clearly explained. Hint: You may find it convenient to express numbers as powers of 2.

A. Alice is trying to optimize the performance of her program:

```
void func() {
    int array[1024];
    for (int i = 0; i < 32; i++) {
       for (int j = 0; j < 1024; j++) {
            array[j]++;
        }
     }
    return;
}
```

She is running this program on a large, powerful processor. It has a fully associative, LRU cache of 128B with a line size of 32B. The inner loop in the above code executes in exactly one millisecond when there is a cache hit, and in exactly 16 milliseconds when there is a cache miss. Loop overhead (incrementing and branching) takes zero milliseconds, and instructions are magically fetched for free.

How long does Alice's program take to complete? **2000 points**

The loop consists of $32 = 2^5$ iterations over $1024 = 2^{10}$ elements. Spatial locality within each cache line gives us one miss every 32B / 4B $= 8 = 2^3$ accesses. Each miss adds additional overhead of the miss penalty minus the hit time (which we'll count separately), or $16 - 1 = 2^4 - 2^0$ ms. Hence, the execution time is

$$
\begin{aligned}
&(2^{10} \cdot 2^5) \text{ iterations} \cdot 2^0 \text{ hit time} + (2^{10} \cdot 2^5 \cdot 2^{-3}) \text{ misses} \cdot (2^4 - 2^0) \text{ miss overhead} \\
&= 2^{15} + 2^{16} - 2^{12} \\
&\approx 3 \cdot 2^{15} \text{ ms}
\end{aligned}
$$

B. Bob suggests that Alice's program would run faster if she designed a new processor for it. Alice realizes that her program is memory-bound, so she decides to allocate more area to the cache and less to the core. Alice runs the same code as above on her new design, which has a larger cache of 4096B (same line size, etc), but takes two milliseconds to execute the inner loop on a cache hit and 16 milliseconds on a cache miss.

How long does Alice's program take to complete now? **1000 points**

The key difference here is that the entire array fits in the cache, so we only pay compulsory misses to bring it into the cache on the first iteration, and the rest of the iterations hit. The array takes 1024 ints / (8 ints / line) = 128 cache lines cache lines to fit,

$$2^{10} \cdot 2^5 \text{ iterations} \cdot 2^1 \text{ hit time} + 2^{10} \cdot 2^{-3} \text{ misses} \cdot (2^4 - 2^1) \text{ miss overhead}$$
$$= 2^{16} + 2^{11} - 2^8$$
$$\approx 2^{16} \text{ ms}$$

C. Bob notices that the above code runs much more quickly *on the original machine* if Alice is willing to modify her code. He suggests the following optimization:

```
void func() {
    int array[1024];
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
            for (int k = i * 32; k < (i+1) * 32; k++) {
                array[k]++;
            }
        }
    }
    return;
}
```

How long does Bob's program take to complete *on the original machine?* Do not make any further optimizations to the code.

This code blocks the array so that once again only the first iteration misses.

$$2^{10} \cdot 2^5 \text{ iterations} \cdot 2^0 \text{ hit time} + 2^{10} \cdot 2^{-3} \text{ misses} \cdot (2^4 - 2^0) \text{ miss overhead}$$
$$= 2^{15} + 2^{11} - 2^7$$
$$\approx 2^{15} \text{ ms}$$

D. Alice isn't satisfied with this performance, but she likes Bob's optimization. Alice designs a new processor that has 8 of the cores she designed for part (b) and a 128B cache from part (a). That is, the inner loop of the processor takes two milliseconds on a cache hit and sixteen milliseconds on a cache miss. To make this processor useful, Alice realizes she needs to edit the code. She modifies the code so that it is multithreaded, and each of 8 threads executes the following function:

```
void func(int threadid) { // threadid is a number from 0 to 7
    int array[1024];
    for (int i = threadid * 4; i < (threadid + 1) * 4; i++) {
        for (int j = 0; j < 32; j++) {
            for (int k = i * 32; k < (i+1) * 32; k++) {
                array[k]++;
            }
        }
    }
    return;
}
```

How long does Alice's multithreaded program take to complete on this multicore machine? **1000 points**

We split the iterations across the cores so that the execution time is given by the time taken for each core to finish its work. Each core executes one eighth (2^{-3}) of the iterations, only miss in the first iterations, but has a hit time of 2 ms.

$$
\begin{aligned}
&2^{10-3} \cdot 2^5 \text{ iterations} \cdot 2^1 \text{ hit time} + 2^{10-3-3} \text{ misses} \cdot (2^4 - 2^1) \text{ miss penalty} \\
&= 2^{13} + 2^8 - 2^5 \\
&\approx 2^{13} \text{ ms}
\end{aligned}
$$

E. Discuss the pros and cons of the above optimizations. When are architectural optimizations useful? When are code optimizations useful? How do they complement each other? **2000 points**

There are many acceptable answers to this question.

We were looking for answers that mentioned that: *(i)* code must optimize for the underlying architecture, and *(ii)* architecture must optimize for the applications its running, and *(iii)* best results are obtained when hardware and software are designed in tandem.

The above examples demonstrate this by showing that architecture (cache) and code (loop tiling) can both improve performance, but best results are obtained when a radical architectural change (multicore) with accompanying software changes (multithreading) enable a much better implementation.

F. Bob modifies the program so that it takes a checksum of the array:

```
void func() {
    int array[1024];
    int checksum = 0;
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
            for (int k = i * 32; k < (i+1) * 32; k++) {
                array[k]++;
            }
        }
    }

    for (int i = 0; i < 1024; i++) {
        checksum = magic_hash(checksum, array[i]);
    }
    return;
}
```

Alice rejects Bob's pull request, saying "My multithreaded code can't get more than 2X on this." Explain her reasoning. **1000 points**

The checksum is fully serialized, meaning that half the program can't be parallelized. By Amdahl's Law, the performance gains from parallelism are limited by the fraction of the program that runs in parallelism. Hence, on this example, Alice knows the max performance gain is $2\times$. .

Synchronization

Problem 2. (5000 points total):

The following questions are related to synchronization. You may assume sequential consistency for these questions, and the machine supports all atomic instructions presented in class.

A. Implement fetch-and-add with Load-Linked/Store-Conditional (LL/SC) **1000 points**

Assume the addends are given by address `addr` and register `R1`.

```
loop:
    LL   addr, R1
    ADD  R2, R1
    SC   R1, addr
    BEQZ loop
```

B. Consider the following Test and Set lock implementation: **2000 points**

```
void get_lock(int *lock) {
    while(test&set(lock) != 0) {}
    return;
}
```

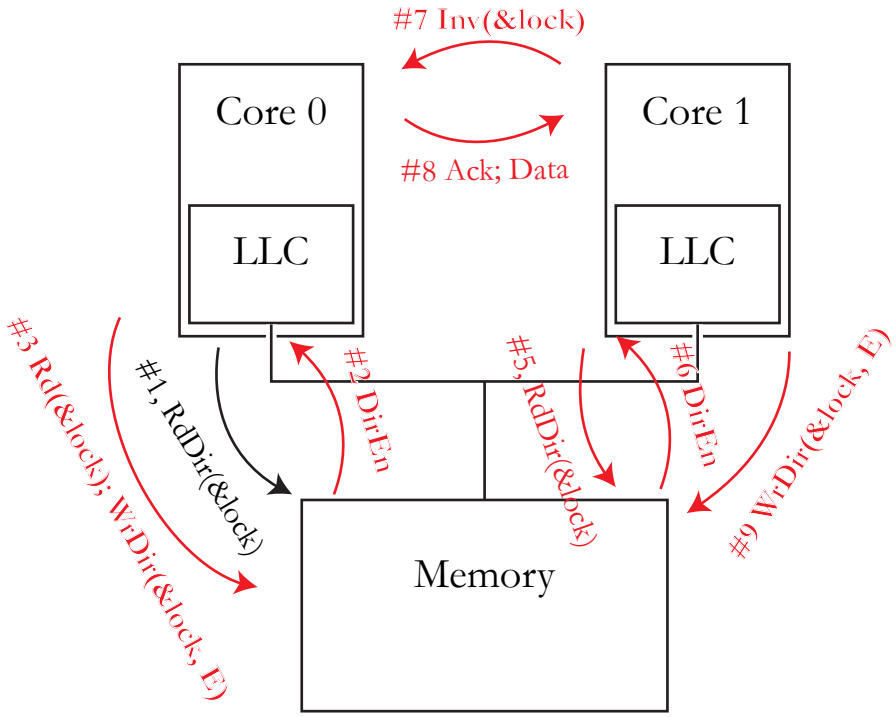
Suppose the following two threads are running on a dual-core processor with no SMT and directory-based coherence with a MESI protocol.

```
Thread 0                Thread 1
--------                --------
acquire_lock(lock)      ...

...                     ...

mat[2][67]++;           acquire_lock(lock);
mat[13][63]++;          mat[2][67]++
mat[86][32]++;          release_lock(lock);
release_lock(lock);
```

(a) This code sequence is relatively slow. Could you improve its performance by removing the locks? Why or why not?

Locks serialize execution and have high overhead. Locks are needed for correct execution, so you can't remove them.

(b) Indicate what coherence traffic will be generated by the locking through drawing arrows labeled with the message and order; the first has been done for you.



There is some variety in how directory protocols work; this solution is for a version of directory-based cache coherence where the cores send the invalidations. We were primarily looking for a demonstration of understanding that states are stored in the directory, which must be updated when data is moved.

C. John came up with the following "Test and Set and Test" lock:

```
void get_lock(int *lock) {
    while(lock != free) {
        while(int tmp = test&set(lock)) {
            if(tmp == free) {
                return;
            }
        }
    }
}
```

But when John ran benchmarks, it did not performance as well as he expected. When he showed it to Mary, she laughed. Why did she laugh, and how did she improve his implementation? **2000 points**

Test&set shouldn't be in the inner loop; the point of test&test&set is to use mostly reads to allow the local operation between releases. This code is just a slower version of test&set from (A).

Other bugs include skipping the inner while loop if the lock is initially free, and setting `tmp` in the while conditional.

Caching

Problem 3. (7000 points total):

In this question, you will demonstrate your mastery of caching. First, you will perform virtual memory lookups and cache accesses, as we have done in class. Then, you will analyze a novel caching optimization.

A. Consider the following byte-addressed, two-way set associative, 32B cache with a 4B line size, and an 8-bit address space. A page is 16 bytes (for 16 pages total). Furthermore, there is a two-way associative TLB with eight entries. The cache is now virtually indexed and physically tagged.

Cache

Set	Way 1						Way 2					
	Valid	Tag	Data				Valid	Tag	Data			
0	0x0	0xA	0x2	0xA9	0xA	0xCE	0x0	0x6	0x69	0x4A	0x44	0x52
1	0x1	0x6	0xDC	0x77	0x24	0xF9	0x1	0xA	0x79	0xD7	0x23	0xA3
2	0x1	0x1	0x4	0x8F	0xB4	0x39	0x1	0xA	0x13	0x86	0x36	0x81
3	0x0	0x1	0x64	0xF7	0xB2	0x1E	0x1	0x7	0xB2	0x84	0xD3	0x12

Page table

VPN	Valid	PPN
0x0	1	0x7
0x1	1	0x9
0x2	1	0xA
0x3	0	0xB
0x4	1	0x8
0x5	0	0xE
0x6	1	0x0
0x7	1	0x1
0x8	1	0xC
0x9	0	0x4
0xA	0	0x5
0xB	1	0xA
0xC	1	0xA
0xD	0	0xD
0xE	0	0x2
0xF	1	0x6

TLB

Set	Way 1			Way 2		
	Valid	Tag	PPN	Valid	Tag	PPN
0	0x0	0x1	0x4	0x0	0x3	0xF
1	0x0	0x3	0xE	0x0	0x1	0x9
2	0x0	0x2	0x1	0x1	0x1	0x0
3	0x0	0x0	0x5	0x1	0x2	0xA

What is the result of accesses to the following virtual addresses: 0x97, 0xB0, 0x79, 0xBB? Show all work. **3000 points**

(answer on next page)

(a) 0x97

address: 1001 0111
TLBT: 10, TLBI: 01
TLB miss
Page table entry is invalid; page fault

(b) 0xB0

address: 1011 0000
TLBT: 10, TLBI: 11
TLB hit
PPN: 1010
PA: 1010 0000
Tag: 1010
Idx: 00
Offset: 00
Invalid, cache miss

(c) 0x79

address: 0111 1010
TLBT: 01, TLBI: 11
TLB miss
PPN: 0001
PA: 0001 1001
Tag: 0001
Idx: 10
Offset: 01
Hit, data: 0x8F

(d) 0xBB

address: 1011 1011
TLBT: 10, TLBI: 11
TLB hit
PPN: 1010
PA: 1010 1011
Tag: 1010
Idx: 10
Offset: 11
Hit, data: 0x81

B. Column-Associative Caches **4000 points total**

Column associative caches[1] were a cache mechanism intended to provide the benefits of associativity without increasing the hit time. This is achieved by using a second hash function when the first results in a collision. One such alternate hashing mechanism—termed *bit flipping*—is to flip the highest bit in the set index[2]. In Figure 1, b is the first hash function (i.e., selecting the set index bits), and f is the second (bit-flipping). Initially, a_i was placed in location 010 after being hashed with function b. When a_j is hashed with b, it results in a collision, but hashing with f allows it to be placed in 110, avoiding a collision.

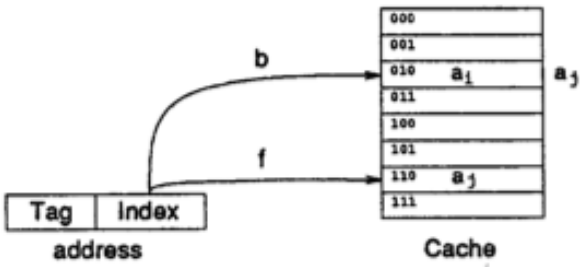


Figure 1: Column-Associative Cache with Bit Flipping

Operation

Upon accessing the cache, if a hit occurs in first location, the data is simply returned. If not, then we check the second location. If it hits, the data is returned, and the data in the first and second location are swapped. If both miss, the data is retrieved from memory, the data from the first location is moved to the second location, and the new data is put in the first location. Figure 2 shows the steps required to perform an access in a column associative cache, where $b[a]$ is the location of a when cached with function b, $f[a]$ is the location of a when cache with function f, and clobber2 is replacing the item in the second location after both locations miss.

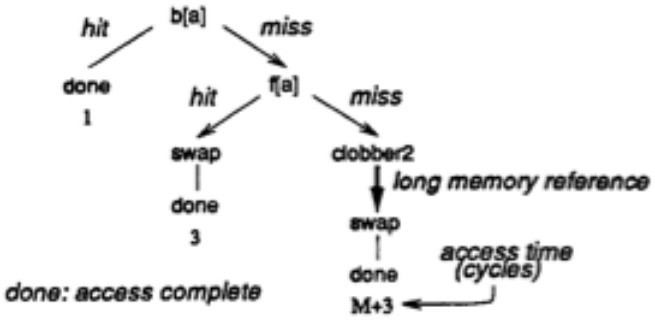


Figure 2: Steps to Access Column-Associative Cache

(questions on next page)

[1] Agarwal and Pudar, 1992

[2] To avoid aliasing with the line with the same tag that belongs in this set when *not flipped*, the highest bit in the set is appended to the tag.

(a) What kinds of misses does column associativity reduce, relative to a direct-mapped cache?

Conflict misses; column caching gives 2 locations for each datum, reducing conflicts and adding associativity.

(b) Compared to a direct mapped cache, what is the overall effect of column associativity on the AMAT.

It depends.

Column caching increases hit time in the second location, but also tends to lower miss ratio. The miss penalty is unaffected.

(c) Give a scenario where a column associative cache would outperform a direct mapped cache, and explain why.

A program where two "hot" lines happen to hash to the same set will perform better with column caching relative to a direct mapped cache, as column caching eliminates these misses.

(d) Give a scenario where column associative cache would outperform a 2-way associative cache and explain why.

A program with few conflict misses and a high hit ratio will outperform a 2-way associative cache. Column caching has lower hit time than 2-way caches because, like direct-mapped caches, it can drive the data from the data array without first checking the tag to see which way matched.

Similarly, a program with a very low hit ratio will perform better on a 2-way cache because column-associative caches make misses more expensive: two widely separated locations must be checked before checking the higher-level cache, whereas a 2-way cache checks both in parallel.

Scratch paper

Scratch paper

Scratch paper