

CS 433 Final Exam – May 15, 2009
Professor Sarita Adve

Time: 3 Hours

Please clearly print your name and NetID and circle the appropriate category in the space provided below.

Name	Solutions	
NetID		
Category	3 Credit Hours	4 Credit Hours

Instructions

1. You may only use class handouts from this semester’s offering, the course text (*Computer Architecture: A Quantitative Approach – 4th Edition* – by Hennessy and Patterson), your own homework submissions for this course, papers indicated as reference material in class, and notes written or typed by yourself. You may also use homework solutions and sample midterms provided on the course website. No other materials are allowed, including other books, notes prepared by others, or materials from previous offerings of this course (except as noted here) or from other universities.
2. Calculators are allowed. You may not use any other electronic devices for any purpose during the exam.
3. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.
4. *In all cases, show your work. No credit will be given for numeric answers if there is no indication of how the answer was derived. Partial credit will be given even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.*
5. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.
6. This exam has **6 problems** and **12 pages** (including this one). **Problem 6 is only for graduate students.** All other problems are required for all students. Please budget your time appropriately. Good luck!

Problem	Maximum Points	Received Points
1	7	
2	16	
3	4	
4	15	
5	8	
6	10 (grads only)	
Total	50 for undergrads, 60 for grads	

Problem 1 [7 points]

Consider a virtual memory system with the following parameters:

- 32 KB direct mapped cache with 32 byte blocks
- 48 bit virtual addresses
- 32 bit physical addresses
- 4 KB page size
- A fully associative TLB with 256 entries
- Memory is byte addressable

Part A [3 points]

How many bits are needed for the page offset, virtual page number, and physical page number?

Solution:

2^{12} byte page size \Rightarrow 12 bits for the page offset

$48 - 12 = 36$ bits for virtual page number, $32 - 12 = 20$ bits for physical page number

Grading: 1 point for each set of bits.

Part B [4 points]

For the system described, what is the disadvantage of using a physically-indexed cache? Give two different changes to the cache design to enable physical indexing without this disadvantage and without changing the cache size. Please explain your answers and be specific to get partial credit.

Solution:

A 32KB direct mapped cache requires 15 bits for the index and block offset. Since 3 of these bits overlap with the page number bits, using a physically-indexed cache necessitates translating virtual addresses before accessing the cache. This can delay every memory access.

Two changes to the cache design to solve the above problem are:

- 1) We can increase the cache associativity to 8-way. Now we need three fewer bits to index the cache, thereby removing the overlap between the index and page number bits.
- 2) We could search 8 cache sets in parallel, corresponding to the 3 overlapping bits. This search can happen in parallel with the address translation. Once the translated address is available, the appropriate cache line is determined.

Grading: 1 point for describing the problem, 1.5 points for each of the two solutions.

Problem 2 [16 points]

Consider the following code, which sums the elements of a product of two matrices:

```
register int i, j, k; /* i, j, k are in the processor registers */
register float sum, a[8][8], b[8][8];

for(i = 0; i < 8; i++) { /* 1 */
    for(j = 0; j < 8; j++) { /* 2 */
        for(k = 0; k < 8; k++) { /* 3 */
            sum += a[i][k] * b[k][j]; /* 4 */
        }
    }
}
```

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.
- Assume that only the accesses to the arrays **a** and **b** generate accesses to the data cache. The rest of the variables are all allocated in registers.
- Assume a fully associative, LRU data cache with **8 lines**, where each line is **32 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. Lines (1), (2), and (3) take **10 cycles** for each invocation. Line (4) takes **10 cycles plus an additional 20 cycles per data cache miss** to wait for the data. That is, if both array accesses in line (4) miss, it takes a total of 50 cycles.
- Assume that the arrays **a** and **b** both start at cache line boundaries.

Part A [3 points]

How many accesses to arrays **a** and **b** will result in cache misses? Explain your answer.

Solution:

A full row of each array fits in one cache line. There are 8 total lines in the cache. Each access to **a** within the outermost loop is to the same cache line, so it is never evicted from the LRU cache until the next iteration of the outermost loop. For **b**, however, the accesses iterate through the 8 rows, and each line is evicted from the cache before it is accessed again. Thus there are:

a: 8 misses

b: $8 * 8 * 8 = 512$ misses

520 total misses

Grading: 1.5 points for calculating and explaining the misses for each array correctly.

Part B [8 points]

Now assume there is a data prefetch instruction with the format `prefetch(array[index1][index2])`. This prefetches the entire block containing the word `array[index1][index2]` 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 20 cycles for the data to get loaded into the cache.

Add prefetch instructions to the code so as to minimize the execution time. Do not transform the code in any other way. How many cache misses for each of arrays **a** and **b** will occur at line (4) in your modified code?

Solution:

```

for(i = 0; i < 8; i++) {                               /* 1 */
    prefetch(a[i][0]);                                 /* p1 */
    prefetch(b[0][0]);                                 /* p2 */
    for(j = 0; j < 8; j++) {                           /* 2 */
        for(k = 0; k < 8; k++) {                       /* 3 */
            prefetch(b[k+1][0]);                       /* p3 */
            sum += a[i][k] * b[k][j];                 /* 4 */
        }
        prefetch(b[0][0]);                             /* p4 */
    }
}

```

For each iteration of the outermost loop, we can insert one prefetch to avoid the compulsory miss for array **a** (line p1). Every access to **b** would be a miss, so we need to insert a prefetch for each of these accesses. In this solution, we prefetch the first row of **b** before each iteration of the innermost loop (lines p2 and p4), and then a prefetch in the innermost loop for the next iteration. (line p3, execution of line 4 and 3 provide the 20 necessary cycles to transfer the data to the cache).

For this modification, there are no cache misses at line 4.

Grading: 2 points for eliminating all misses for **a**. 2 points for eliminating the miss for `b[0][0]`. 2 points for eliminating the other misses for **b**. 2 points for giving the correct number of cache misses for the transformed code.

Part C [5 points]

Now, assume that the prefetch instruction is no longer available. Instead, you have access to a third matrix $c[8][8]$, which is not used anywhere else in the code. Assume that cache misses on a read to c incur the same 20 cycle penalty as a and b , but misses on a write are without penalty. Using matrix c , transform the code above so that it reduces the number of cache misses during data reads in part A by at least 75%. You may modify and add instructions, assume each one takes 10 cycles to execute plus appropriate data cache miss penalties. How many read misses (to arrays a , b , and c) does your new code have?

Solution:

Several correct solutions are possible, we list two possibilities:

1) By setting c to the transpose of b , we can reduce the number of cache misses:

```
for(i = 0; i < 8; i++) {                               /* 0.1 */
    for(j = 0; j < 8; j++) {                             /* 0.2 */
        c[j][i] = b[i][j];                               /* 0.3 */
    }
}
for(i = 0; i < 8; i++) {                               /* 1 */
    for(j = 0; j < 8; j++) {                             /* 2 */
        for(k = 0; k < 8; k++) {                         /* 3 */
            sum += a[i][k] * c[j][k];                   /* 4 */
        }
    }
}
```

Each time a new row is accessed for any of a , b , or c , it will miss in the cache. This happens 8 times for array b in line (0.3), 8 times for array a in line (4), and $8 * 8$ times for array c in line (4).

Total misses: $8 (0.3 b) + 8 (4 a) + 8 * 8 (4 c) = 80$

2) Alternatively, simply interchanging the order of the j and k for loops meets the goal, without using array c . For this approach, a still misses 8 times, while b now only misses 64 times, for a total of 72 misses.

Grading: 3 points for effectively transforming the code. 2 points for correctly calculating the misses.

Problem 3 [4 points]

Circle the most appropriate choice.

1. Which of the following is **NOT** a feature of *Intel Wide Dynamic Execution* in the *Intel Core 2 Duo* processor:
 - a) Four instruction decode units, decoding up to five instructions per cycle
 - b) Macro-fusion of instructions and micro-fusion of operations respectively
 - c) Out-of-order retirement (commit) unit
 - d) Advanced Branch and Loop prediction

2. The memory hierarchy for the *Quad-Core AMD Opteron* processor has:
 - a) Shared L1, L2, and L3 caches
 - b) Local L1 caches, shared L2 and L3 caches
 - c) Local L1 and L2 caches, shared L3 cache
 - d) Local L1, L2, and L3 caches

3. Which type of parallel architecture best describes the *Nvidia GPU* and *AMD Radeon* processors:
 - a) Single instruction stream, single data stream (SISD)
 - b) Single instruction stream, multiple data streams (SIMD)
 - c) Multiple instruction streams, single data stream (MISD)
 - d) Multiple instruction streams, multiple data streams (MIMD)

4. Which of the following is **NOT** true of the Cell processor:
 - a) Elements of the Cell processor are present in the Playstation 3 and Xbox 360 gaming systems.
 - b) The resident Power Processing Element (PPE) is similar to the PowerPC processors
 - c) Each Synergistic Processing Element (SPE) has a coherent cache using the MESI protocol.
 - d) The Cell's floating point computation power makes it an affordable choice for clusters for complex physics simulations.

Solution: 1)c 2)c 3)b 4)c

Grading: 1 point each

Problem 4 [15 points]

This problem concerns an invalidation based snooping cache coherence protocol for bus-based shared-memory multiprocessors. The protocol, called MOESI, has five states. A block in a cache C can be in one of the following states:

- **Modified:** The block is present only in a single cache and the data in the cache is dirty or modified (i.e., it reflects a more recent version than the copy in memory). This state is similar to the read-write state discussed in class and occurs because of a write.
- **Owned:** The block may be present in more than one cache. The memory may not have an up-to-date copy of this block. It is owned by cache C and C must service the requests of other caches to this block since memory may not have an up-to-date copy.
- **Exclusive:** The block is present in a single cache but is clean (i.e., memory has an up-to-date copy of the block).
- **Shared:** The block is possibly present in several caches. This state is similar to the read-only state discussed in class.
- **Invalid:** The block is not valid in this cache (space for the block may or may not be currently allocated in this cache). This is the same as the invalid state discussed in class.

If the cache has a block in Owned state, then it services any requests to that block from other processors. Assume that the memory does NOT update its copy even if the request is a read by some other cache and the Owner cache has put the block on the bus. Hence, the owner cache remains in Owned state and continues to service other requests, until the block is replaced from its cache. Also, assume that the only way to reach the Owned state is from the Modified or Exclusive state, when some other cache issues a read request for that block.

If a cache C has a block in exclusive or modified state, then it is responsible for servicing any requests to that block from other processors. If the request is a read, then the cache C transitions to the Owned state, and memory does NOT update its copy.

On replacement of a block in Owned or Modified state, the block is sent to memory, and memory resumes responsibility for servicing subsequent requests to that block. Replacement of a block in Exclusive state is similar, except that the block need not be sent to memory (since memory already has a copy).

Assume that after a cache performs a transaction on a bus, there is a mechanism for it to know whether other caches have a copy of the requested block or not at that time. This enables the cache to determine whether to transition to exclusive state.

Part A [3 points]

Consider a Block B in *Owned state* in the cache of processor P. Can B be in a non-invalid state in any other processor's cache. If yes, then what are the possible (non-invalid) states in which B could be in any of the other caches. If no, then explain why not.

Solution:

Yes. B can be in Shared state in other caches even when it is in Owned state in P1. This scenario results when P1 has the block in Modified state and another processor does a read on B. The only possible non-invalid state of B in other caches is S.

Grading: 2 points for recognizing that B may be non-invalid in another processor's cache, 1 point for the correct possible state. 3 points total.

Part B [6 points]

This part concerns the response of the cache of processor i to bus transactions initiated by the cache of processor j for block B. Fill out the following rows for the state transition table for the cache of processor i , showing the next state for block B in the cache and any action taken by the cache. Each entry should be filled out as: *Next State/Action* (e.g., S/Send block to memory) where *Next State* = **M, O, E, S, or I**

Action =

- **Send block to memory,**
- **Send block to cache,**
- **Send block to cache and memory, or**
- **No action**

Note: If an entry is not possible (i.e., the system cannot be in such a state), write “Not Possible” for that entry:

Current state of block B in cache of processor i	Read of block B by cache of processor j	Invalidate of block B by cache of processor j	Read+Invalidate of block B by cache of processor j
M			
O			

Solution:

Current state of block B in cache of processor i	Read of block B by cache of processor j	Invalidate of block B by cache of processor j	Read+Invalidate of block B by cache of processor j
M	O/Send block to cache	Not Possible	I/Send block to cache
O	O/Send block to cache	I/NA	I/Send block to cache

Grading: 1 point for each entry. Partial credit is awarded if at least the next state is computed correctly. 6 points total.

Part C [4 points]

Consider the following sequence of operations by two processors (P1.Read(B) means that processor P1 requests a read on block B). Determine the state of Block B in the caches of both the processors after each operation in the sequence. Both blocks are initially Invalid in both caches. The table below is provided to help organize your answer.

Sequence: P1.Read(B), P1.Write(B), P2.Read(B), P1.Read(B), P2.Write(B), P1.Read(B)

Operation	State in P1	State in P2
P1.Read(B)		

P1.Write(B)		
P2.Read(B)		
P1.Read(B)		
P2.Write(B)		
P1.Read(B)		

Solution:

Operation	State in P1	State in P2
P1.Read(B)	E	I
P1.Write(B)	M	I
P2.Read(B)	O	S
P1.Read(B)	O	S
P2.Write(B)	I	M
P1.Read(B)	S	O

Grading: 0.25 points for each part. 3 points total. 1 extra point if all are correct. Cascading errors get no credit.

Part D [2 points]

In what scenario would you see an advantage of using the Owned (O) state in the MOESI protocol (over protocols such as MESI and MSI discussed respectively in homework and class)?

Solution:

MOESI has an advantage if it is cheaper to transfer data between two caches than between memory and cache. For example, consider the following access stream to block B:

P1.Write(B), P2.Read(B), P3.Read(B), P4.Read(B)

In this case, P1's cache would send the data to P2, P3 and P4 for MOESI.

With MESI, P3 and P4 would get the data from memory and the read of P2 would result in a writeback to memory.

Grading: *2 points for the above answer. 2 points total.*

Problem 5 [8 Points]

This problem involves implementing a stack using an array in a multiprocessor system. The elements of the array can be accessed in parallel by multiple processors.

You have to write the following two functions:

- *Push*: This will add an element to the top of the stack.
- *Pop*: This will delete an element from the top of the stack.

Assume that *Push* is never called on a full stack and *Pop* is never called on an empty stack (i.e., you do not have to worry about overflow and underflow conditions).

Write the *Push* and *Pop* functions using an atomic *test&set* instruction to achieve synchronization. Add C-like pseudo code to the following stub (complete the incomplete statements as well):

```
int top; /* index for the top of the stack */
int index; /* current index for adding or deleting an element */
Lock lock_var; /* Lock variable for synchronization */
```

```
Push (item)
```

```
{
```

```
    index =
```

```
    stack[index] = item;
```

```
}
```

```
Pop (void)
```

```
{
```

```
    index =
```

```
    item = stack[index];
```

```
    return item;
```

```
}
```

Solution:

Push (item)

```
{
    while (test&set(lock_var) == 1); /* Spin until lock is obtained */
    index = top;
    top++;
    stack[index] = item;
    lock_var=0; /*Unlock*/
}
```

Pop (void)

```
{
    while (test&set(lock_var) == 1); /* Spin until lock is obtained */
    index = top-1;
    top--;
    item = stack[index];
    lock_var=0; /*Unlock*/
    return item;
}
```

Grading: *The key part to this problem is the correct identification and treatment of the critical section. 2 points are given for this and 2 points are given for the rest of the code. However, code with unreasonable semantics for Push and Pop will be marked down. 4 points for each function. 8 points total.*

ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM.

Problem 6 [10 points]

Assume the code below executes concurrently on two processors on data *f* in shared memory as shown. Assume *s*, *i*, and *j* are allocated in registers within each processor. Assume each processor has its own cache which is kept coherent through an invalidation based protocol. Assume a cache block size of 64 bytes with one valid bit per block and an int datatype size of 32 bytes.

```
struct foo {  
    int x;  
    int y;  
};
```

```
foo f;
```

Processor 1

```
for (i=0; i<100,000,000; i++)  
    s += f.x;
```

Processor 2

```
for(j=0; j<100,000,000; j++)  
    ++f.y;
```

Part A [4 points] What can you say about the possible cache performance of this code? Explain your answer.

Solution:

If *f* is aligned on a cache line boundary, then *f.x* and *f.y* will be in the same cache block. Processor 1 is only reading *f.x* and no other processor is writing to *f.x*. Ideally, processor 1 should not see any misses to *f.x* after the first load. However, because *f.x* and *f.y* are on the same cache line, each write of Processor 2 to *f.y* results in an invalidation of *f.x* and a subsequent read miss for Processor 1. This is called false sharing. It results in poor cache performance for this code.

Grading – One point for correctly identifying that the cache performance is not good for the case where the data structure is cache line boundary aligned. Three points for the explanation. It is OK if the students do not mention the term false sharing explicitly.

Part B [3 points] Explain how the code will perform on an update based cache coherent system?

Solution:

The cache performance of the given code will be better on an update based cache coherent system. Every update of Y on processor 2, will result in an update of the corresponding cache block in processor 1. Thus, processor 1 will not suffer any cache read misses for X unlike the previous part. That said, there will still be unnecessary and avoidable (given that X and Y are not truly shared) update traffic on the bus.

Grading: 1.5 points for stating that Processor 1 will not see any misses (after the first one) and 1.5 points for stating that Processor 2 incurs a lot of traffic.

Part C [3 points] Describe a software-only technique that could eliminate virtually all the misses in the invalidation protocol and perform better than the update protocol scenarios above. Please explain the technique and show the changed code.

Solution:

To reduce false-sharing, the goal should be to locate data objects so that only one truly (concurrently) shared object occurs per cache block frame in memory. If this is done, then even with just a single valid bit per cache block, false sharing is impossible. In this example, X and Y can be padded with an extra 32 bytes each to prevent false-sharing.

```
struct foo {  
    int x;  
    int pad1;  
    int y;  
    int pad 2;  
};
```

Alternate solutions:

1) Code restructuring to use a local copy of x and y in the loop
- three points will be awarded for this solution

2) Usage of locks.

```
Proc 1  
lock()  
for (i = 0; i < 100,000; i++)  
    s += f.x  
unlock()
```

```
Proc 2  
lock()  
for (i = 0; i < 100,000; i++)  
    ++f.y;  
unlock()
```

This solution will completely serialize the code and not improve performance over the update protocol. It will remove the misses, so one point will be awarded.

Grading: Three points for a solution to eliminate false-sharing from the code.