CS 433 Homework 5

Assigned on 11/7/2017
Due in class on 11/30/2017

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 11/30/2017.

Problem 1 [4 Points]

Consider a system with the following processor components and policies:

- A direct-mapped L1 data cache of size 4KB (4×2^{10} bytes) and block size of 16 bytes, indexed and tagged using physical addresses, and using a write-allocate, write-back policy
- A fully-associative data TLB with 4 entries and an LRU replacement policy
- Physical addresses of 32 bits, and virtual addresses of 40 bits
- Byte addressable memory
- Page size of 1MB

Part A [2 points]

Which bits (counting from 0 at the LSB) of the virtual address are used to obtain a virtual to physical translation from the TLB? Explain exactly how these bits are used to make the translation, assuming there is a TLB hit.

The virtual address is 40 bits long. Because the virtual page size is 2^{20} bytes, and memory is byte addressable, the virtual page offset is 20 bits. Thus, the first $40-20=20$ bits (20-39 bits) are used for address translation at the TLB. Since the TLB is fully associative, all of these bits are used for the tag; i.e., there are no index bits. When a virtual address is presented for translation, the hardware first checks to see if the 20 bit tag is present in the TLB by comparing it to all other entries simultaneously. If a valid match is found (i.e., a TLB hit) and no protection violation occurs, the page frame number is read directly from the TLB.

1 point for explanation; 1 point for correct numbers

Part B [2 points]

Which bits of the virtual or physical address are used as the tag, index, and block offset bits for accessing the L1 data cache? Explicitly specify which of these bits can be used directly from the virtual address without any translation.

Since the cache is physically indexed and physically tagged, all of the bits for accessing the cache must come from the physical address. However, since the lowest 20 bits of the virtual address form the page offset and are therefore not translated, these 20 bits can be used directly from the virtual address. The remaining 12 bits (of the total of 32 bits in the physical address) must be used after translation. Since the block size is 16 bytes $(=2^4)$ bytes, and memory is byte addressable, the lowest 4 bits are used as block offset. Since the cache is direct mapped, the number of sets is $\frac{4KB}{16B}=2^8$. Therefore, 8 bits are needed for the index. The remaining $32-8-4=20$ bits are needed for the tag. As mentioned above, the index and offset bits can be used before translation while the tag bits must await the translation for the 12 uppermost bits.

1 point for explanation; 1 point for correct numbers

Problem 2 [12 points]

Consider a tiny system with virtual memory. Physical addresses are 8 bits long, but only $2^7 = 128$ bytes of physical memory is installed, at physical addresses 0 up to 127. Pages are $2^4 = 16$ bytes long. Virtual addresses are 10 bits long. An exception is raised if a program accesses a virtual address whose virtual page has no mapping in the page table, or is mapped to a physical page outside of installed physical memory.

The contents of main memory are shown in Table . To find the physical address of a byte, read the least significant digit from the column label and the most significant digit from the row label. For example, the shaded byte in the second row is at physical address 0x12. All entries are in hexadecimal.

Table 1: Main Memory

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0_	4e	65	76	65	72	20	67	6f	6e	6e	61	20	67	69	76	65
0x1_	20	79	**6f**	75	20	75	70	0a	4e	65	76	65	72	20	67	6f
0x2_	6e	6e	61	20	**6c**	**65**	**74**	**20**	79	6f	75	20	64	6f	77	6e
0x3_	0a	4e	65	76	65	72	20	67	6f	6e	6e	61	20	72	75	6e
0x4_	20	61	72	6f	75	6e	64	20	**61**	**6e**	**64**	**20**	64	65	73	65
0x5_	72	74	20	79	6f	75	0a	4e	65	76	65	72	20	67	6f	6e
0x6_	6e	61	20	6d	61	6b	65	20	79	6f	75	20	**63**	**72**	**79**	**0a**
0x7_	4e	65	76	65	72	20	67	6f	6e	6e	61	20	73	61	79	20

The page table is shown in Table 2. The virtual page number in the left column is mapped to the physical page number in the second column. Virtual page numbers are listed in binary.

Table 2: Page Table

Virtual page	Physical page
0	0x2
1	0x4
10	0x1
11	0x5
100	0x4
101	0x7
110	0x9

Part A[2 points]

List the four bytes in the word beginning at physical address 0x34.

0x65,0x72,0x20,0x67

Part B[2 points]

How many virtual addresses refer to the first byte of the shaded word in row 0x2_? List them.

Only virtual page 000 maps to physical page 2, so there is only 1 address, 0x04

Part C[2 points]

How many virtual addresses refer to the first byte of the shaded word in row 0x4_? List them.

Virtual pages 1 and 4 map to physical page 4, so there are 2 addresses, 0x18 and 0x48

Part D[2 points]

How many virtual addresses refer to the first byte of the shaded word in row 0x6_? List them.

No virtual pages map to physical page 6, so there are no addresses that refer to those bytes.

Part E[2 points]

What data is returned if the program loads a word from virtual address 0x5C (01011100)?

Data is returned from the word at physical address 0x7C : 0x73, 0x61, 0x79, 0x20 (bytes or address sufficient)

Part F[2 points]

What is the result if the program loads a word from virtual address 0x64 (01100100)?

An exception, because virtual page 6 is mapped to physical page 9, which is outside the range of installed memory.

Problem 3 [11 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. Assume *f* is aligned on a cache block boundary; i.e., *f.x* is the first word in a cache block and *f.y* is the second word in the cache block.

```
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 [3 points]

What can you say about the possible cache performance of this code? Explain your answer.

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.

Part B [4 points]

Explain how the code will perform on an update based cache coherent system?

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.

2 pts for identifying that Processor 1 does not see any misses except the first miss. 2 pts for noting the high traffic for Processor 2

Part C [4 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. Your solution should not add synchronization.

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 solution: Use a local variable to store the value of f.x.

```
Processor 1:
int temp = f.x;
for (i=0; i<100,000,000; i++)
   s += temp;
Processor 2:
for(j=0; j<100,000,000; j++)
   ++f.y;
```

Problem 4 [21 points]

This problem concerns MOESI, an invalidation based snooping cache coherence protocol, for bus-based shared-memory multiprocessors with a single level of cache per processor. The MOESI protocol has five states. A block starting at address Addr can be in one of the following states in cache C:

- Modified: The block is present only in cache C and the data in the cache is dirty or modified (i.e., it reflects a more recent version than the copy in memory).
- Owned: The block is present in cache C and may also be present in other caches. The memory may not have an up-to-date copy of this block. The block is said to be 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 (C) but is clean (i.e., memory has an up-to-date copy of the block).
- Shared: The block is present in cache C and possibly present in other caches.
- Invalid: The block is not valid in cache C (space for the block may or may not be currently allocated in this cache).

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 the 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 [2 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 processors 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.

Yes. B can be in Shared state in other caches even when it is in Owned state in P. This scenario results when P 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.

Part B [9 points]

This part concerns the response of the cache of processor i to bus transactions initiated by the cache of processor j for a block that starts at address B (referred to as block B below). 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:

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 (with no read request for the block)	Read +Invalidate of block B by cache of processor j
M	O/Send block to j	Not Possible	I/Send Block to j
O	O/Send block to j	I/No Action	I/Send Block to j
E	O/Send block to j	Not Possible	I/Send Block to j

1 pt per correct entry

Part C [10 points]

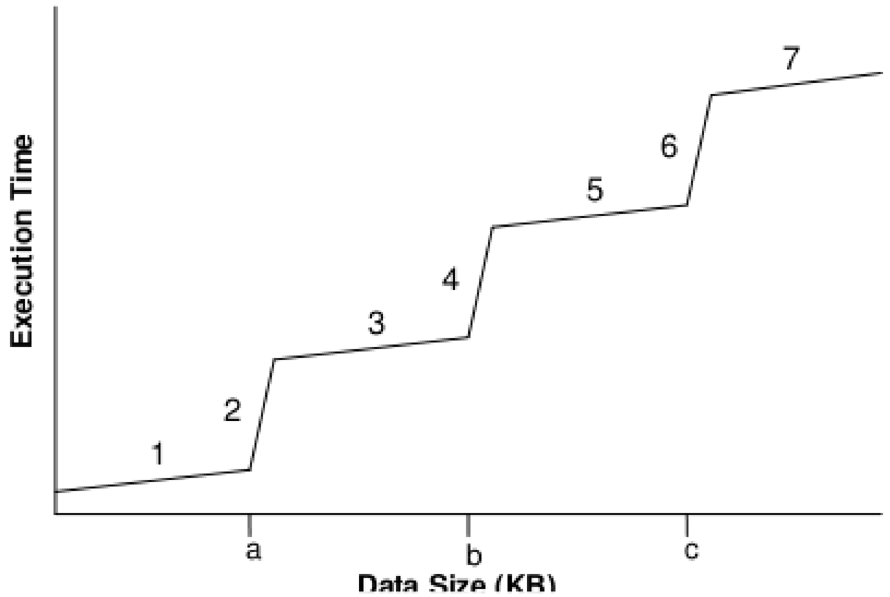
Consider the following sequence of operations by two processors for a block that starts at address B. Determine the state of that block in the caches of both the processors after each operation in the sequence for the MOESI protocol. Both caches are initially empty. The table below is provided to help organize your answer.

No.	Operation	MOESI	
		P1	P2
1	P1 reads B	E	I
2	P1 writes B	M	I
3	P2 writes B	I	M
4	P1 reads B	S	O
5	P1 writes B	M	I
6	P2 reads B	O	S

1 pt per correct entry

Problem 5: Graduate students only [12 points]

Consider the following graph obtained by varying the amount of data accessed by a certain benchmark. The only thing you know of the experiments is that the system uses virtual memory, a data TLB, only one level of data cache, and the data TLB maps a much smaller amount of data than can be contained in the data cache. You may assume that there are no conflict misses in the caches and TLB. Further assume that instructions always fit in the instruction TLB and an L1 instruction cache.



Part A [7 points]

Give an explanation for the shape of the curve in each of the regions numbered 1 through 7.

Region on graph	Explanation
1	Execution time slowly increases (performance decreases) due to increasing data size but remains at a roughly similar level
2	At this point, the TLB overflows and execution time sharply increases to handle the increased TLB misses
3	Execution time again slowly increases due to increasing data size and plateaus at a higher level than before due to overhead from TLB misses
4	At this point, the data cache overflows, causing a high frequency of cache misses and execution time again sharply increases
5	Execution time again slowly increases due to increasing data size and plateaus at a high level due to overhead from retrieving data directly from main memory due to cache misses
6	Execution time again sharply increases due to physical memory filling up and thrashing occurring between disk and physical memory
7	Execution time is very high due to overhead from TLB misses, cache misses and virtual memory thrashing. It is slowly increasing due to increasing data size

1 pt for a resonable explanation of each region

Part B [5 points]

From the graph, can you make a reasonable guess at any of the following system properties? If so, what are they? If not, why not? Explain your answers. (Note: your answers can be in terms of a, b, and c).

1. Number of TLB entries
2. Page size
3. Physical memory size
4. Virtual memory size
5. Cache size

There is no reasonable guess for page size and virtual memory size. There is also no reasonable guess for the number of TLB entries since it depends on the page size. It is acceptable if you guess that the cache size is b KB and the physical memory size is c KB, since these are the points at which the execution time shows significant degradations. However, these quantities are actually only upper bounds, since the actual size of these structures depends on the temporal and spatial reuse in the access stream. (The actual size depends on a property known as the working set of the application.)

1 pt for each item in the list