15740-18740 Computer Architecture, Fall 2010
Midterm Exam II

Instructor: Onur Mutlu
Teaching Assistants: Evangelos Vlachos, Lavanya Subramanian, Vivek Seshadri
Date: November 22, 2010

Name:

Problem 1 (60 points)	:	
Problem 2 (30 points)	:	
Problem 3 (18 points)	:	
Problem 4 (20 points)	:	
Problem 5 (24 points)	:	
Problem 6 (24 points)	:	
Bonus Problem 7 (24 points)	:	
Legibility and Name (5 points)	:	
Total (181 + 24 points)	:	

Instructions:

1. This is a closed book exam. You are allowed to have two letter-sized cheat sheets.
2. No electronic devices may be used
3. This exam lasts 1 hour 50 minutes.
4. Clearly indicate your final answer.
5. Please show your work when needed.
6. Please write your name on every sheet.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Problem 1: Potpourri (60 points)

1) Always Not Taken (6 points)

A processor employs a static branch prediction technique which always predicts a conditional branch to be not taken. Vivek runs a set of applications on this processor and determines the conditional branch prediction accuracy to be 30%.

a) What can be done to improve conditional branch prediction accuracy for this set of applications without changing the processor?

Reorder the basic blocks such that all the taken branches are converted to not-taken branches and vice versa.

b) What other benefits does your scheme provide (if anything else)? Be precise.

Fewer fetch breaks.

2) Cache Logic (6 points)

Assume a 32-bit address space, a cache size of 64KB and a cache block size of 64 bytes.

a) How many tag comparators do you need to build this cache, if the cache is direct-mapped?

One (number of comparators = number of ways)

b) How many tag comparators do you need to build this cache, if this cache is 4-way set-associative?

4

c) How many tag comparators do you need to build this cache, if this cache is fully associative?

Number of tag comparators = associativity of the cache = $\frac{64\ KB}{64\ bytes} = 1k$

3) DRAM Refresh (3 points)

Why is there a need to perform "refresh" operations in DRAM?

Because the capacitor that stores a value loses charge over time.

4) DRAM Scheduling (6 points)

What is the purpose of request batching in DRAM scheduling (think about parallelism aware batch scheduling)?

To avoid starvation of threads

Where else can the idea of request batching be applied? Be brief, but specific.

Any place where there could be prioritization of threads that could lead to starvation. For example, thread-aware packet schedulers in on-chip networks or thread-aware disk schedulers.

5) Execution-based Prefetching (9 points)

In lecture, we talked about the idea of creating a "thread" for pre-executing a piece of the program solely for prefetching purposes. Assume we have a mechanism that creates such threads, which we called "speculative threads" in lecture, as opposed to the "main program thread" which is the original application. These "speculative threads" can be executed

1. on a core separate from the core where the main program thread is running.
2. on a separate hardware thread context in the same core where the main program thread is running.
3. or on the same hardware context in the same core where the main program thread is running, when the main program thread is idle due to long-latency cache misses.

Option 1 (Separate core)

Advantage:

The main program thread runs without sharing any computing resource with the speculative thread. This reduces the interference between the two threads.

Disadvantage:

Requires another idle core to be present in the system. Also, blocks can be prefetched only into the first shared level of cache.

Option 2 (Separate hardware thread context on the same core)

Advantage:

Running on the same core can ensure that blocks are prefetched directly into the core's L1 cache, increasing the benefit of prefetching.

Disadvantage:

The speculative thread competes with the main thread for compute resources in the pipeline and can significantly slow down the main thread's execution.

Option 3 (On the same hardware thread context during idle cycles)

Advantage:

Can prefetch data into the L1 cache without slowing down the main thread.

Disadvantage:

Cannot always ensure timely prefetches as the speculative thread runs only when the main thread stalls ⇒ Prefetching lookahead/distance is limited

6) Superscalar Decode (10 points)

Suppose you are building a superscalar processor that can issue 4 x86 instructions in parallel. You find that decoding takes a significant latency. Explain what characteristic of the x86 ISA causes this. Why?

x86 has variable length instructions. Multiple instruction decode happens serially because a younger instruction can be decoded only after the length of the previous one is known.

Briefly describe one technique that reduces the latency of the x86 multiple instruction decoder.

1) Decoded I-Cache: We can store decoded instructions in the instruction cache. This will allow us to bypass the decoder on a hit in the I-cache.
2) Other possible answers: Pre-decoding, wimpy decoders, trace cache.

What are the disadvantages of the technique?

1) Decoded I-Cache: Reduces i-cache utilization. Reduces decode latency only on an I-cache hit; I-cache miss instructions still need to be decoded.
2) Each other answer has its respective disadvantages.

7) Trace Caches (10 points)

What does branch promotion allow that enhances the performance of a trace cache?

Branch promotion allows more basic blocks to be grouped into a single trace, thereby improving the trace cache utilization.

How is the promoted branch handled at fetch and execute stages of the processor?

When a promoted branch is fetched, it is tagged with the static prediction (it is *not* predicted; an implicit prediction is made by the existence of the next basic block in the same trace cache line). When it is executed, the correct direction is compared to the implicit static prediction; pipeline is flushed on a misprediction.

A curious scientist discovers that the branch prediction accuracy on a processor with a trace cache that employs branch promotion is higher than that on a processor with a trace cache that does not employ branch promotion. Explain why this is the case.

Promoted branches do not interfere with other branches in the BTB and in the direction predictor. Hence, branch promotion can increase the prediction accuracy of other branches.

8) Predicated Execution (5 points)

Assume the following branch is predicted perfectly (100% accuracy) when it is executed on a 8-wide superscalar, out-of-order processor.

```
if (a == b) {
   x = 1;
} else {
   x = 3;
}
```

Would you ever want to convert this code into predicated code? Circle one: YES NO
Why, why not? Explain clearly.

Yes. 100% prediction accuracy does not imply that the branch will either be taken 100% of the time or not taken 100% of the time. So we might still want to predicate the code if the branch is sometimes taken and sometimes not taken. This ensures that there are no fetch breaks when the branch is taken.

9) Prefetching Algorithms (5 points)

Recall that content directed prefetching identifies pointers in a fetched cache line and generates prefetch requests for those pointer addresses. What are two advantages this prefetching mechanism has that Markov and Stream prefetchers do not have?

It does not require any training (both Markov and Stream do).

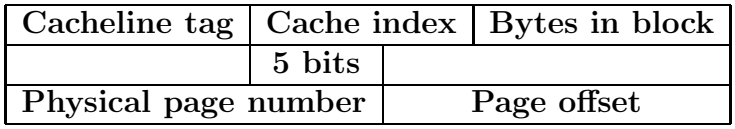
It does not require any storage (both Markov and Stream do).

Problem 2: Multi-core Cache Partitioning (30 points)

Suppose we have a system with 32 cores that are sharing a physical second-level cache. Assume each core is running a single single-threaded application, and all 32 cores are concurrently running applications. Assume that the page size of the architecture is 8KB, the block size of the cache is 128 bytes, and the cache uses LRU replacement. We would like to ensure each application gets a dedicated amount of cache space in this shared cache without interference from other cores. We would like to enforce this using the OS-based page coloring mechanism to partition the cache, which we discussed in lecture. (Hint: the operating system ensures, using virtual memory mechanisms, that the applications do not contend for the same space in the cache)

a) What is the minimum size the L2 cache needs to be such that each application is allocated its dedicated space in the cache via page coloring? Show your work. (4 points)

For OS based page coloring to work in this case, we need at least 32 colors. This means we need at least 5 bits of the cache index to intersect with the physical page number.



So, with associativity A, page size P, the minimum cache size is given by,

$$\begin{aligned} C &\geq A * 2^5 * P = A * 32 * P \\ \Rightarrow C &\geq A * 32 * 8KB = A * 256KB \end{aligned}$$

Minimum cache size (associativity = 1) is 256KB

b) Assume the cache is 4MB, 32-way associative. Can the operating system ensure that the cache is partitioned such that no two applications interfere for cache space? Show your work. (4 Points)

For a given associativity, minimum cache size = A * 256 KB (from part a). Therefore, for a 32-way associative cache, minimum cache size required for the OS to ensure partitioning without interference is 32 * 256 KB = 8 MB. Since the cache size is only 4 MB, the OS, in this case, cannot ensure partitioning without interference.

c) Assume you would like to design a 32MB shared cache such that the operating system has the ability to ensure that the cache is partitioned such that no two applications interfere for cache space. What is the minimum associativity of the cache such that this is possible? Show your work. (4 points)

Minimum associativity = 1

d) What is the maximum associativity of the cache such that this is possible? Show your work. (4 points)

From part a),

$$\begin{aligned} C &\geq A * 256KB \\ \Rightarrow A &\leq C/256KB \\ \Rightarrow A &\leq 32MB/256KB \\ \Rightarrow A &\leq 128 \end{aligned}$$

Therefore, maximum associativity is 128.

e) Suppose we decide to change the cache design and use utility based cache partitioning (UCP) to partition the cache, instead of OS-based page coloring. Assume we would like to design a 4MB cache with a 128-byte block size. What is the minimum associativity of the cache such that each application is guaranteed a minimum amount of space without interference? (3 points)

Utility based cache partitioning needs to give at least one way for each application. Otherwise, the application will receive no cache space. Hence, the minimum associativity is 32.

f) Is it desirable to implement UCP on a cache with this minimum associativity? Why, why not? Explain. (3 points)

No, it is not desirable to implement UCP. There will be no benefit gained from UCP since UCP guarantees at least one way per application. This means all applications will be allocated exactly one way of the cache, i.e. the cache is equally and statically partitioned regardless of applications' utility for caching.

g) What is the maximum associativity of a cache that uses UCP such that each application is guaranteed a minimum amount of space without (4 points) interference?

The maximum associativity corresponds to a fully associative design. For the given configuration, it is $\frac{4\ MB}{128\ bytes} = \frac{2^{22}}{2^7} = 2^{15} = 32k$ **ways.**

h) Is it desirable to implement UCP on a cache with this maximum associativity? Why, why not? Explain. (4 points)

It is not desirable to implement UCP with this maximum associativity because the overhead of UCP for 32 applications on this cache will likely outweigh its benefits. UCP will only work with LRU replacement policy. But implementing LRU on top of a 32k-way cache is impractical. Also the number of counters needed by UCP and the partitioning solution space for UCP are very large for such a cache.

Problem 3: Where to Prefetch into? (18 points)

Lavanya, Vivek, and Evangelos are evaluating the design decisions for a new prefetcher to be implemented on their new single-core system with a two-level cache hierarchy. Lavanya designs a prefetcher that places the prefetched cache blocks into the L1 cache. Vivek modifies the design such that the prefetcher places the prefetched blocks into the L2 cache instead of the L1 (all else remains the same).

a) On workload A, Lavanya's design performs better than Vivek's design. Is this possible? Circle one: YES NO
Explain why. (6 points)

Yes. For workload A, the prefetcher could be both accurate and timely, and may not cause much pollution with the demand-fetched blocks in the L1 data cache. Therefore, prefetching into L1 reduces the latency of demands that access the prefetched blocks.

b) On workload B, Vivek's design performs better than Lavanya's. Is this possible? Circle one: YES NO
Explain why. (6 points)

Yes. For workload B, the prefetcher may be accurate but not timely (i.e. too early). Prefetching blocks into the L1 cache evicts other useful cache blocks that are needed soon again, but the prefetched blocks do not get used until after they are evicted from L1. Prefetching into L2 reduces this pollution caused by untimely prefetches.

c) Evangelos modifies the design such that the prefetcher places the prefetched blocks into a separate prefetch buffer accessed in parallel with the L2 cache. To his surprise, he finds that this design degrades performance compared to both Lavanya and Vivek's designs on both workloads A and B. Is this possible? Circle one: YES NO
Explain clearly why this could happen, if it is possible. (6 points)

Yes. The prefetch buffer may be too small (compared to the L1 cache and L2 cache) to store all the prefetched cache blocks, Prefetched blocks can evict older prefetched blocks from the buffer even before they are used.

Problem 4: Register File (20 points)

a) You are asked to design a 16-wide-issue superscalar machine. Assuming each instruction sources two register operands and writes to one destination register, how many read and write ports are needed in the register file? (State your assumptions, if needed, clearly) (2 points)

Number of read ports = 32
Number of write ports = 16

b) Why is having so many ports to the register file undesirable? (3 points)

Increases latency, area, and power consumption of the register file.

c) In class, we discussed several techniques to reduce the port requirements into the register file. Describe one such technique. (5 points)

Clustering: Partition the backend (scheduler, register file, execution units) into clusters. Each cluster either has a part of the register file or has a replicated copy of it. Each cluster schedules and executes a fraction of the issue width number of instructions.

d) Describe how and why it reduces number of ports into the register file? (5 points)

Each cluster schedules and executes a fraction of the issue width number of instructions, which reduces the number of read and write ports.

e) What additional complexities or overheads does the technique introduce that were not present in the baseline processor? (5 points)

When different source registers of an instruction are in different clusters, we need additional copy operations to move the values from one cluster to another. Also, when instructions write to registers in another cluster, the value cannot be sent to dependent instructions immediately (next cycle).

Problem 5: DRAM Read-Write Switching Penalty (24 points)

The DRAM controller needs to service both read and write requests to DRAM. The read requests are load misses in the last level cache and the write requests are writebacks from the last level cache. At any point in time, the controller will likely have a set of read and write requests to be serviced. Whenever a read request is followed by a write request or vice versa, the controller has to stall for some number of DRAM cycles during which it cannot issue any requests to DRAM.

In a modern DRAM, switching from a read to write takes 2 DRAM cycles and switching from a write to read takes 6 DRAM cycles. Assume that there is only one bank in the system and a row-buffer hit takes 14 DRAM cycles and a row-buffer conflict takes 34 DRAM cycles. At some point in time, the following requests are in the DRAM controller queue. A request is represented as (Operation row-number, column-number). The leftmost request is the oldest request and the rightmost is the youngest.

a) A memory controller does not distinguish between reads and writes and simply uses FR-FCFS (row-hit first scheduling policy). Determine the total number of DRAM cycles taken to service the above requests in the request queue. Assume that the first scheduled request is a row conflict and has no switching penalty. Show your work for partial credit. (4 points)

No.	Request	Access Cost	Switch Penalty
1	Rd 1,1	34	0
2	Wr 2,3	34	2
3	Rd 3,1	34	6
4	Wr 4,7	34	2
5	Rd 5,9	34	6
6	Wr 5,5	14	2

202 cycles.

b) Assume you modified the FR-FCFS scheduler such that it always prioritizes write requests (and everything else remains the same). Determine the total number of DRAM cycles taken to service the above requests. Show your work clearly. (4 points)

No.	Request	Access Cost	Switch Penalty
1	Wr 2,3	34	0
2	Wr 4,7	34	0
3	Wr 5,5	34	0
4	Rd 5,9	14	6
5	Rd 1,1	34	0
6	Rd 3,1	34	0

190 cycles.

c) Is prioritizing write requests (as in (b)) a good idea? Why, why not? (4 points)

No. Not a good idea because writes are not critical to system performance since a processor does not stall on a writeback request (unless the write buffer is full, which should not be the common case).

d) You are now given the task of improving the scheduler. Describe how you would design the controller such that the system performance is maximized in the presence of read-write penalty. Describe all modifications you would make (to the scheduling mechanism, buffering, etc.) to make your solution viable and implementable. (8 points)

The controller should service read requests while buffering writes in a write buffer. When the write buffer gets full, it can flush all (or some of) the writes in the buffer to the main memory. This can ensure fewer switches between reads and writes while also ensuring high performance by prioritizing reads.

e) Determine the total number of DRAM cycles taken to service the above requests in the request queue, with your new design. Show your work clearly. (4 points)

No.	Request	Access Cost	Switch Penalty
1	Rd 1,1	34	0
2	Rd 3,1	34	0
3	Rd 5,9	34	0
4	Wr 5,5	14	2
5	Wr 2,3	34	0
6	Wr 4,7	34	0

So the controller takes 186 cycles to finish all the requests.

Problem 6: Hardware/Software Cooperation (24 points)

In class, we have talked about several hardware/software cooperative techniques, solving different problems. Pick your favorite (if you do not have a favorite, pick one you feel the most comfortable explaining to us – we will call it your favorite). Explain how it works by answering the following questions, clearly focusing on the technique's advantages and disadvantages (make sure your comparison points are clear).

a) The name of the technique: (2 points)

Agree predictor
(Many other possible answers: block-structured ISA, virtual memory, efficient content directed prefetching)

b) The problem the technique solves (be specific): (4 points)

Negative interference between branches in the pattern history table(s) of the branch predictor.

c) Basic idea of the technique (be clear – we are looking for the high level idea and insight, not implementation details): (5 points)

A branch instruction is tagged with a static bias for the direction (Taken or Not Taken). The PHT entry records whether or not the prediction agrees or disagrees with the bias bit instead of whether or not the branch is taken. Since most branches are biased, they will tend to agree with the bias bit even if they interfere in the PHT → increases positive interference in PHTs.

d) What is an alternative technique to solve the same problem. Briefly explain the key idea of the technique (be clear) (5 points)

Randomizing the index into the pattern history tables. Use a hash function to generate the index into the pattern history table, which reduces the probability of branches aliasing in the table.

e) Advantages of your favorite technique over the alternative: (4 points)

1) Increases the likelihood of positive interference between branches (assuming branches are biased). Randomizing the index reduces the probability of interference, but does not inherently make it more positive.
2) Works when there are many branches competing for PHTs. Randomizing the index cannot significantly reduce interference if there are already too many branches mapped to the PHTs.

f) Disadvantages of your favorite technique over the alternative: (4 points)

1) Requires compiler support based on profiling.
2) Profile-time bias can be opposite of run-time bias.
3) Does not work for branches that are not biased (Randomizing the index can).

Bonus (Predicated Execution)

Predicated execution converts control dependencies into data dependencies, thereby eliminating branches. The following is a piece of code with a single branch in it.

```
if (a == c) {
   x = a;
}
else {
   y = b;
}
z = x + y;
```

Assume the following mnemonics:

```
1.  MOV R1, R2          // Copy the value of R2 to R1
2.  ADD R1, R2, R3      // R1 = R2 + R3
3.  BEQ label, R1, R2   // If (R1 == R2), jump to label
4.  JMP label           // Jump to label unconditionally
5.  SETEQ p1, R1, R2    // If R1 == R2, set p1 to true; else, set p1 to false
6.  p1 INSTRUCTION      // If p1 is true, execute INSTRUCTION; else, NOP
7.  !p1 INSTRUCTION     // If p1 is false, execute INSTRUCTION; else, NOP
                        // INSTRUCTION can be either instruction 1 or 2
```

Assume the following mapping between variables and registers:
a → R1, b → R2, c → R3
x → R4, y → R5, z → R6

a) Write an assembly code equivalent to the above program using conditional branches (you can use only instructions 1, 2, 3 and 4). (3 points)

```
BEQ label, R1, R3
MOV R5, R2
JMP exit
label:  MOV R4, R1
exit:  ADD R6, R4, R5
```

b) Write an assembly code equivalent to the above program using predicated instructions (you can use all instructions except 3 and 4). (5 points)

```
SETEQ p1, R1, R3
p1 MOV R4, R1
!p1 MOV R5, R2
ADD R6, R4, R5
```

c) Briefly explain how predicated execution affects each of the following stages of the superscalar pipeline.

Fetch (4 points)

Predicated execution helps avoid fetch breaks by removing control dependencies.

Dependency Check (as part of Decode) (4 points)

Since predicated instructions source from the predicate register, the dependency check logic should also check for dependencies on these registers.

Rename (4 points)

Renaming becomes more complex because a predicated instruction may or may not write its result to the destination. For example, in the predicated code in part b), only one of the instructions 2 & 3 will write to the destination. So the registers R4 and R5 in instruction 4 cannot be renamed using conventional methods before the predicate is computed.

Commit (4 points)

A predicated instruction writes its result into the architectural state only if the predicate is true. Otherwise the result should simply be discarded.