CIS501 – Computer Architecture — Midterm Exam Solutions

Prof. Martin — **Thursday, Oct. 27th, 2011**

1. **[13 Points] Short Answer**.

   (a) Caches are effective because real-world programs have two key properties. Name and describe these two properties:

   Answer: [4 points] (1) Temporal locality: *recently accessed memory locations* are more likely to accessed in the near future than other locations and (2) Spatial locality: location *nearby recently accessed memory* locations are more likely to accessed in the near future than other locations

   (b) In the past, processors had smaller caches than today. Give two reasons:

   Answer: [2 points]
   Any of: (1) Memory latency was shorter (caches didn't matter so much), (2) more transistors (cheaper to build large caches), and (3) programs and datasets are larger (thus benefit from larger cache).

   (c) Energy consumption is an important design constraint for portable devices. Why?

   Answer: [1 point] Primarily because of battery life concerns.

   (d) Energy consumption is an important design constraint for servers. Why?

   Answer: [1 point] The cost of energy into the servers and especially the cost of cooling.

   (e) What are the two types of energy (or power) consumption of a processor? What is the source of each type?

   Answer: [4 points] Dynamic power and static power. Dynamic energy is consumed when transistors switch. Static power is caused by "off" transistors leaking energy.

   (f) How can a compiler optimization increase performance yet hurt CPI?

   Answer: [1 point] It may have removed instruction (improving performance) but the instructions removed could have had lower than average CPI (thus increasing the average CPI of the remaining instructions).

2. **[11 Points] Performance Calculations**.

   (a) Consider a pipeline with a **five-cycle branch misprediction penalty** and a **single-cycle load-use delay penalty**. For a specific program, 20% of the instructions are loads, half of loads are followed immediately by a dependent instruction, 10% are store instructions, 15% are branches, the remaining 55% of instructions are simple single-cycle ALU operations. 80% of branches are predicted correctly. What is the average CPI of this program on this processor?

   Answer: [3 Points] The CPI would be one plus an extra cycle for loads followed by dependent operations (10% of instructions), an extra five cycles for each mis-predicted branch (20% of 15%). Thus, the CPI is $1 + (1 * 0.10) + (5 * 0.20 * 0.15) = 1.25$.

As a chip designer, you have been asked to determine the cache configuration of your company's next-generation chip. Two of your options are: (1) a direct-mapped cache with a **two-cycle hit latency** or (2) a set-associative cache with a **three-cycle hit latency**. In both cases, the miss latency is (an additional) 10 cycles.

   (b) If the miss rate of the direct-mapped cache is 15%, what is its average latency of a memory operation?

   Answer: [2 points]
   $2 + (15\% * 10) = 2 + 1.5 = 3.5$ cycles

   (c) What miss rate must the set-associative cache obtain to have a lower average latency than the direct-mapped cache with a 15% miss rate?

   Answer: [2 points]
   $3 + (x * 10) = 3.5$
   $(x * 10) = 0.5$
   $x = 0.05 = 5\%$

In a different project, you must choose between two different pipelines: (i) a relatively short pipeline with a 3-cycle mis-prediction penalty running at 1Ghz, or (ii) a longer pipeline with a 16-cycle mis-prediction penalty running at 2Ghz. In the target workload, 40% of the instructions are branches.

   (d) Calculate under what conditions is pipeline *(ii)* faster than pipeline *(i)*.

   Answer: [3 points] To determine the cross-over point, we find the point where the performance of each is equal, and then solve for the branch misprediction rate.
   $CPI_1 = 1 + (0.4 * 3 * x)$
   $CPI_2 = 1 + (0.4 * 16 * x)$
   However, pipeline 2 runs at twice the frequency, so we really need to divides its CPI by two. We can then set them equal to each other.
   $1 + (0.4 * 3 * x) = (1 + (0.4 * 16 * x))/2$
   $2 + (2.4 * x) = (1 + (6.4 * x))$
   $1 + (2.4 * x) = (6.4 * x))$
   $1 = 4 * x$
   $x = 0.25$
   Thus, if the misprediction rate is 25% or better (75% accuracy or higher), pipeline (ii) is faster.

   (e) Based on what you have learned about branches, does pipeline *(ii)* seem like a reasonable design point? Why or why not?

   Answer: [1 point] Yes, as prediction accuracies are well above 75%.

3. **[7 Points] Pipelining**.

(a) We discussed a "fast branch" optimization in which certain simple branches (such as compare to zero) can resolve during the "D" stage rather than the "X" stage. What was the advantage of this approach for a simple 5-stage pipeline?

Answer: A fast branch reduces the branch mis-prediction penalty from two cycles to one.

(b) What are two disadvantages and/or limitations of this fast branch optimization?

Answer: It only works for simple branches (equality or comparison to zero, otherwise cycle time would certainly be impacted) and it requires bypassing values into the D stage (which can also hurt the clock frequency). Alternatively, if bypassing is not added, logic would need to be introduced stall those branches that need a bypassed value, again limiting the overall beneficial impact of the optimization.

The standard pipeline discussed in class has five stages: fetch (F), decode (D), execute (X), memory (M), and writeback (W). Consider the following **four-stage** pipelines, formed by combining two stages of the classic five-stage pipeline.

(c) What specific **positive** impact would combining the "D" & "X" stages have on CPI?

Answer: Reduce the branch misprediction penalty from two to one.

(d) What specific **positive** impact would combining the "X" & "M" stages have on CPI?

Answer: Eliminate the load-to-use stall cycle.

(e) What specific **positive** impact might combining the "M" and "W" stages have on the clock frequency of the pipeline?

Answer: By combining these stages, it removes a bypassing path, reducing the size of the bypass muxes in the "X" stage. If the "X" stage was the slowest stage, this could improve the clock frequency of the pipeline.

(f) What specific impact would combining the "F" and "D" stages have on the implementation of branch prediction?

Answer: As decode is part of the fetch cycle, a branch target buffer (BTB) wouldn't be needed to determine which instructions are branches or what their target would be.

4. **[13 Points] Branch Prediction**

(a) What are the two purposes of a branch target buffer (BTB)?

Answer: [2 points] (1) determine if an instruction is a branch or not just based on the PC of the branch, and (2) if the branch is predicted taken, the addresses to which the branch jumps.

(b) What is the purpose of a *return address stack*? Assuming a processor with a BTB, why might it also have a *return address stack* in addition to the BTB?

Answer: [2 points] The return address stack predicts the target of function call return instructions. As the target of a return statement depends on when function called it (and the same function can be called from many places), a BTB is as effective at predicting return addresses as a return address stack.

(c) What is a *gshare* branch direction predictor and why does it generally increase prediction accuracy?

Answer: [2 points] The gshare predictor uses a branch pattern register that records the outcomes of the most recent n branches. By hashing the branch pattern register with the PC before making a prediction, the predictor then examines a different two-bit counter based on the context of the prediction, increase its accuracy.

(d) For fixed branch predictor size, sometimes a *bimodal* outperforms *gshare*. Give two distinct reasons this can occur.

Answer: [2 points] (1) each static branch updates multiple different counters, placing significant pressure on the capacity of the predictor. (2) the predictor takes longer to learn all the patterns, thus has a longer warmup period (that is, a gshare predictor could have a lower prediction accuracy than bimodal even for a predictor of infinite size).

(e) What technique is used to mitigate these tensions between bimodal and gshare?

Answer: [1 point] A hybrid predictor (which uses a "chooser" table to select between the bimodal or gshare predictor on each prediction).

(f) Consider a predictor with a single saturating counter which is either 1-bit or 2-bits. The 1-bit counter encodes either "Taken (T)" or "Not-taken (N)", and it is initialized to "Not-taken". The 2-bit counter encodes "Strongly Taken (T)", "Weakly Taken (t)", "Weakly Not-taken (n)" and "Strongly Not-taken (N)", and it is initialized to "Weakly Not-taken (n)".

i. Give a short sequence (4 or fewer) of branch directions (T or N) in which a 2-bit saturating counter is better than a 1-bit saturating counter. What is the prediction accuracy for *both* predictors?

Answer: [2 points] N T N — 33% vs 66% (answers may vary)

ii. Give a short sequence (4 or fewer) of branch directions (T or N) in which a 1-bit saturating counter is better than a 2-bit saturating counter. What is the prediction accuracy for *both* predictors?

Answer: [2 points] N T T — 66% vs 33% (answers may vary)

5. **[15 Points] Caches.**

(a) Consider system with 48-bit addresses and a 128KB direct-mapped data cache with 128-byte blocks. How many bits are in the offset, index, and tag for this cache?

Answer: [3 points] 7 offset bits, 10 index bits, and $48 - 7 - 10 = 31$ tag bits.

(b) What is the tag overhead percentage for the cache?

Answer: [2 points] Each tag is 32 bits (31 bits plus a valid bit) or 4 bytes. As the block size is 128-byte, the overhead is $4/128$ or approximately 3%.

(c) In an important program, two program variables continually conflict in this cache. For example, the variables might be at the following two addresses:

```
0000 0000 0010 0100 0100 1000 1001 0000 0100 1000 1101 1010
0000 0000 1000 0010 1000 0011 0001 0100 0100 1000 1010 0100
```

What are three general options for how the cache geometry might be changed so that these two addresses no longer conflict. For each option, also give the primary (or most likely) disadvantage and an approach than can help mitigate (or reduce) that disadvantage.

Answer: [3x3 = 9 points] Option #1: increase the associativity. Disadvantage: slower hit latency. Mitigation: way prediction.
Option #2: decrease block size. Disadvantage: may hurt hit rate by capturing less spatial locality. Mitigation: next-block sequential prefetching.
Option #3: increase cache size. Disadvantage: slower hit latency. Mitigation(s): pipeline the cache access over multiple cycles, allow independent instruction to not stall, and/or use a non-blocking/lockup-free cache.

(d) Beyond just changing the cache geometry, what else could be done that might help prevent these two variables from conflicting in the cache?

Answer: [1 point] Modify the program to change the data layout of these variables so they are at different addresses that no longer conflict. Another approach would be to adjust the virtual to physical page mappings.

6. **[7 Points] Pipeline Simulation.**

[Note: overall this was an extremely difficult question (median score was 2 out of 7 for the entire question). Many students got the correct answer for the first part (worth two points). Nobody really got the second part and only one student out of over 80 students really solved the last part of this question (although a few others received at least some partial credit).]

Consider using the simple "scoreboard" algorithm you used in the second homework assignment to simulate a pipelined processor with full bypassing, two read ports, a single write port, and a configurable load-use latency. When applied to a standard pipeline, there is an implicit assumption in the model that all instructions flow through all stages before entering the writeback stage to avoid a specific type of hazard.

What type of hazard is being avoided? What is the specific cause of the hazard?

Answer: Structural hazard on the single register write port.

With longer load latencies (say, three or more cycles) having all instructions travel through all stages has a significant disadvantage. What is that disadvantage?

Answer: Extra bypassing paths from each of those stages, which can hurt the clock frequency.

In lecture, we discussed a pipeline with a multi-cycle multiplier in which multiply and non-multiply instructions traveled through a different number of stages in the pipeline. Inspired by such a design, consider a modified pipeline in which non-memory instructions enter the writeback stage (W) directly after the execute stage (X). Memory instructions travel through additional stages based on the specific load latency being simulated.

— question continued on next page —

How would you model a pipeline that accounts for the hazard introduced by the change described above? Briefly describe your approach and modify the pseudo-code below to implement it. *Hint:* You may add additional tracking structures to the code, but doing so is not actually necessary to model the desired effect.

```
int scoreboard[MAX_REGS]
next_instruction = true
cycle_count = 0


while (true) {
  cycle_count = cycle_count + 1


  decrement all non-zero entries in the scoreboard



  if (next_instruction) {
      read next instruction from trace
      next_instruction = false
  }



  if (source1_register is valid and scoreboard[source1_register] > 0) or
     (source2_register is valid and scoreboard[source2_register] > 0) {


      continue        // Stall detected; go to next loop iteration

  }



  next_instruction = true



  if (instruction.destination_register is valid) {


      scoreboard[instruction.destination_register] = instruction.latency


  }

}
```

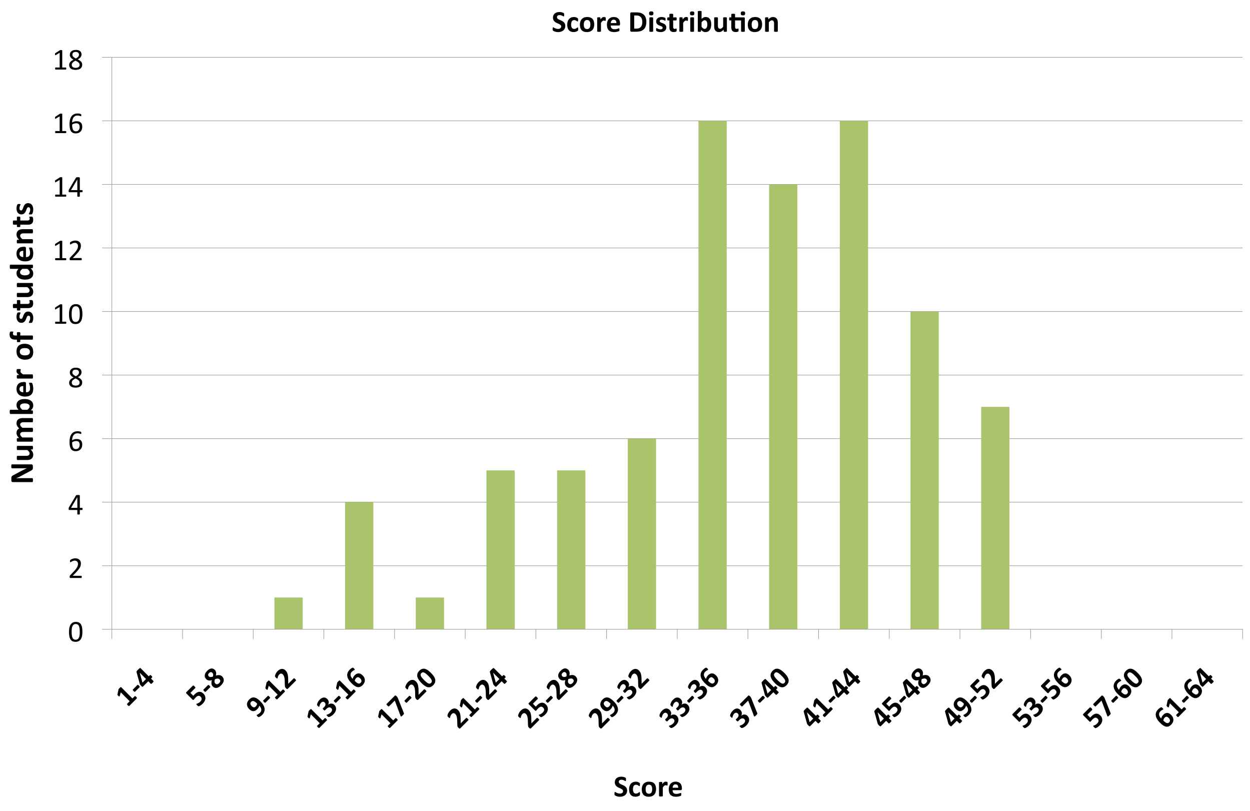
Answer: The algorithm can be adjusted to model the structural hazard on the write port in a few different ways. One simple way is to scan the scoreboard to see if any of the registers are already set to be ready in the same cycle. If so, that is a write port conflict. This can be done my looping over the scoreboard looking to see if there was already an instruction scheduled to write the register in the same cycle. That is, if scoreboard[i] == instruction.latency, there needs to be a stall.

```
int scoreboard[MAX_REGS]
next_instruction = true
cycle_count = 0

while (true) {
  cycle_count = cycle_count + 1
  decrement all non-zero entries in the scoreboard

  if (next_instruction) {
      read next instruction from trace
      next_instruction = false
  }

  if (source1_register is valid and scoreboard[source1_register] > 0) or
     (source2_register is valid and scoreboard[source2_register] > 0) {

      continue        // Stall detected; go to next loop iteration
  }

  if destination_register is valid:                          // **ADDED**
    for reg in 0...MAX_REGS-1 :                              // **ADDED**
      if (scoreboard[reg] == instruction.latency):           // **ADDED**
        continue                                             // **ADDED**

  next_instruction = true

  if (instruction.destination_register is valid) {
      scoreboard[instruction.destination_register] = instruction.latency
  }
}
```

A more efficient alternative would be to add another array of booleans to track which cycles the write port has already been allocated. At the start of each cycle, this array of booleans is then shifted down by one. This approach avoids iterating over the entire scoreboard each cycle.

— End of exam —

Distribution



Mean: 36.53 points — Median: 37 points — High: 52