

CS 573

Midterm Exam

Solutions

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your social security number, but not your name, on each sheet of paper you turn in
- show your work whenever appropriate. There can be no partial credit unless I see how answers were arrived
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 10:20 to finish the exam. The questions are equally weighted.

Here is a little bit of code for a generic three-operand register-based computer, implementing a bubble sort. The questions on this exam are all based on this code. (Note: assume \$0 always contains 0).

```
        addi $1, $0, 40      ; $1 is used to hold a constant 40 to mark end of loop
        addi $2, $0, 0      ; set outer loop counter to 0
outer   addi $3, $0, 0      ; set inner loop counter to 0
inner   ld   $4, awry($3)   ; read awry[i] into $4
        ld   $5, awry+4($3) ; read awry[i+1] into $5 (integers are 4 bytes)
        ble  $4, $5, noswap ; if awry[i] > awry[i+1], swap
        addi $6, $4, 0      ; tmp = awry[i]
        addi $4, $5, 0      ; awry[i] = awry[j]
        addi $5, $6, 0      ; awry[j] = tmp
        st   $4, awry($3)   ; write swapped values back to memory
        st   $5, awry+4($3)
noswap  addi $3, $3, 4      ; increment inner loop counter, see if we're done
        bne  $1, $3, inner
indone  addi $2, $2, 4      ; increment outer loop counter, see if we're done
        bne  $1, $3, outer
```

1. Suppose this code is executed on a computer with an 8K direct-mapped L1 data cache with a 16-byte cache line. Estimate the hit rate in the L1 data cache, assuming the data turns out to have already been sorted when the program is executed. Would a victim cache be likely to improve it?

Since the line size is four times the size of an element in the array, there will be a cache miss on every fourth iteration of the inner loop, on the first iteration of the outer loop only. Consequently, there will be three cache misses. These are all compulsory misses.

On the other hand, the inner loop is executed a total of 100 times (ten times for each iteration of the outer loop, and the outer loop is executed ten times). There are two memory accesses in each iteration of the inner loop (making the assumption that the array is already sorted, so there are no write-backs), so there are a total of 200 data memory accesses.

The hit rate is 197/200, or 98.5%

There are no conflict misses, so a victim cache won't help the situation at all.

2. Suppose this code is executed on a processor using a one-bit branch predictor. Estimate the prediction accuracy, assuming all of the elements of the array are in fact already sorted when you begin execution.

This question required a clarification during the exam: assume the predictor is initialized to "not taken".

There are a total of three branch instructions in the code: one at the end of each loop, and one to determine whether to swap two elements. These are all initially predicting "do not branch."

- (a) *The branch at the end of the inner loop mispredicts on the first iteration (it predicts “no branch” and the branch is taken). After this it is predicting “branch taken” for the remaining nine iterations of the loop; it will be correct for eight of these and incorrect for the last one. So, there will be eight correct predictions on the ten iterations. It will now be predicting “branch not taken” the next time it enters the inner loop, and the same result will occur. So, it will correctly predict 80 out of 100 branches.*
- (b) *The branch at the end of the outer loop exhibits exactly the same behavior; it correctly predicts eight out of ten iterations.*
- (c) *Finally, the `ble` is taken on every iteration (for a total of 100). Its first execution will predict “not taken”, but the remaining 99 will all be correct.*

*Putting it all together, we have 187 correct predictions out of a total of 210 branches, for a correct prediction rate of **approximately 90%**.*

3. Qualitatively discuss the extent to which a superscalar, out-of-order implementation is likely to be able to execute this code effectively both with and without register renaming (*i.e.*, how important would register renaming be to taking advantage of the instruction level parallelism in this code?).

*The code reuses its result registers on each loop iteration. Consequently, it’s only possible to reorder code within a single iteration. With register renaming, it becomes possible to allocate new physical registers on each iteration; **this makes a lot more reordering possible.***

4. Consider modifying this code for an EPIC-style processor.

- (a) The three lines of code immediately following the `ble` instruction swap two elements of the array, prior to writing the data back out to memory. Show how you can use IA-64 parallel execution semantics (all instructions that execute in parallel read their operands simultaneously, perform the operation, and write their results simultaneously) to do this operation in only two instructions.

If we have EPIC-style semantics, we can replace those instructions with

```
addi $5, $4, 0
addi $4, $5, 0;;
```

(the intent here is that we are executing these two instructions in parallel). Note that answers that took the existing three instructions and grouped them into two groups (but still with a total of three instructions) weren’t answering the question.

- (b) Suppose we add predicate registers, and an IA-64 style compare instruction

```
cmp.gt p1 = $10, $11
```

which would compare register \$10 to \$11, and set the predicate bit `p1` to 1 if \$10 is greater than \$11, and 0 otherwise. Use this instruction, and predication, to disable the swap and writeback code instead of using the `ble` from the original code. Of course, you won’t be using registers \$10 and \$11, you’ll be using other registers.

```
cmp.gt p1 = $4, $5
(p1) addi $6, $4, 0
(p1) addi $4, $5, 0
(p1) addi $5, $6, 0
(p1) st $4, awry($3)
(p1) st $5, awry+4($3)
```

This solution assumes no change to the instruction semantics other than those specified in Part (b) of the problem. Using the two-instruction swap code from Part (a) would also be OK. Thinking a little harder about the code and grouping the instructions so that the `cmp` and the swap code are in a single group, while the writeback code is predicated and in a second group, would be better. Like this:

```
cmp.gt p1 = $4, $5
addi $5, $4, 0
addi $4, $5, 0;;

(p1) st $4, awry($3)
(p1) st $5, awry+4($3);;
```

It’s probably worth pointing out that the swap code isn’t actually needed at all. You could just write the values back into the array swapped (whether using `ble` or predication). But I needed it to make a worthwhile question...