Name _____

Multiprocessors, Etc.
EE 7700-2
Take-Home Midterm Examination

Monday, 24 November 2003, 18:30 CST – Monday, 1 December 2003

Test Rules:

- Do not discuss the test with your classmates.
- You can use the class notes and papers to solve the problems.
- Several questions refer to papers used in the class. Those papers are linked to `http://www.ece.lsu.edu/tca/ref.html`.
- If a question is not clear, appears too hard, or too easy, ask me about it.
- Check the course home page daily for answers to questions about the exam, hints, etc.
- E-mail will be checked during the Thanksgiving break.

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: Diplomatically one might say that DGP and pre-execution each has its strengths and weaknesses. But partisans of each scheme by definition think differently.

Provide two partisan code examples, one for DGP, the other for Dynamic Speculative Pre-computation (DSP), the version of pre-execution described in Collins Micro 2001.

Being partisan, the DGP code example must work great on DGP and not do well with pre-execution. Similarly, the pre-execution code example must work well for pre-execution but not for DGP.

The code samples can be in MIPS or some other RISC ISA, and do not have to be syntactically perfect. It is not necessary to show every instruction but do show the load and the instructions in its dependence graph (backward slice). Use ellipses ($\cdots$) to show groups of uninteresting instructions.

Start by identifying a weakness of a scheme that the other does not have. Then construct an example targeting that weakness. [25 pts]

☐ Show the code examples.

☐ Show a diagram of their execution (like a pipeline execution diagram or PSE PED plot).

☐ Explain how each code sample targets a weakness of the "inferior" system.

Problem 2: A case could be made that a load that is pre-computed while it is in the IFQ could have been executed by the main processor if it had a larger reorder buffer. Make that case.

Compare three four-way superscalar dynamically scheduled systems. One has a 64-entry ROB (as the one simulated in the paper) and uses DGP with a 64-entry IFQ, one has a 64-entry ROB but does not use DGP, the third system also doesn't use DGP and has a 128-entry ROB; the systems are identical otherwise.

Suppose that in all systems instructions commit at a rate of 2 IPC (half the ideal rate) in regions of code without cache misses, the commit rate drops (of course) when a missing load is encountered. Also assume that all troublesome loads are properly identified. [25 pts]

(*a*) Using a simple code example explain why the system with the larger ROB might be at least as fast. *Hint: Performance is lost when a load is still waiting for the cache when it reaches the head of the ROB.*

(*b*) Because of the way DGP works, it might be slower than the system with the larger ROB, particularly for loads with more instructions in its dependence graph (backward slice). Explain why using an example.

(*c*) Even if the system with the larger ROB has a higher IPC the DGP system might still be better. Give a possible reason. *Hint: "higher IPC" is not quite the same thing as faster, especially in real life.*

Problem 3: For the trace cache described in Rotenberg 1999 an indirect jump (a jump in which the target address is a register value [or computed using register values]) must be the last instruction in a trace cache entry (or the penultimate instruction if there is a delay slot, as in MIPS and SPARC). The code below contains several indirect jumps, one of which is part of a switch statement, `jr s4`, and the others are used for procedure returns.

In this problem consider a system using the trace cache and next-trace predictor described in Jacobson 1997. In particular, trace cache entries are limited to sixteen instructions. The system runs the code fragment below. The problems begin on the next page. [25 pts]

```
# When loop starts:
#  $s1 has the address of a string.
#  $t0 has address at which to place alphanumeric characters from string.
#  $t1 has address at which to place punctuation characters from string.
#  $s3 is the address of a dispatch array.  The array has 256 elements, one
#      for each character.  An element can be ALPHA, PUNCT, WHITESPACE, or
#      OTHER (the line labels from the program), based on the corresponding
#      character.  For example, "A" is character 65 and so element 64 is ALPHA.
#      ASCII "&" is 38 and so the 38'th element is PUNCT.
#
# When code returns:
#  Alphanumeric characters written to $t0
#  Punctuation characters written to $t1

LOOP:
        lbu $s0, 0($s1)     # Load a character from the string.
        bne $s0, $0, SKIP   # Branch if $s0 not equal to $0 (constant zero).
        sll $s2, $s0, 2     # CTI delay slot.
        jr $ra              # Return
        add $v0, $t1, $t2   # CTI delay slot.
SKIP:
        add $s2, $s2, $s3
        lw $s4, 0($s2)      # Load an entry from the dispatch array.
        jr $s4              # And jump to the appropriate piece of code.
        nop

ALPHA:
        sb 0($t0), $s0      # Store alphanumeric character.
        j CONTINUE
        addi $t0, $t0, 1    # CTI delay slot.

PUNCT:
        sb 0($t1), $s0
        j CONTINUE
        addi $t1, $t1, 1    # CTI delay slot.
WHITESPACE:
        j CONTINUE
        addi $t2, $t2, 1    # CTI delay slot.
DEFAULT:
        jr $ra              # Return with an error code.
        addi $v0, $0, -1    # CTI delay slot.

CONTINUE:
        j LOOP
        addi $s1, $s1, 1    # CTI delay slot.
```

Problem 3, continued:

(*a*) For the code fragment above show a longest possible trace cache entry. Do not include code before or after the loop. Assume that the paths to the returns are not taken.

(*b*) Suppose the string that `s1` points to is of the form "`aa!  bb.  cc?  dd$ ee,` " (a repeating sequence of alpha, alpha, punctuation, single whitespace character) but much longer. After warmup how accurately would the correlated next trace predictor from Rotenberg 1999 (and Jacobson 1997) predict traces for this loop? Explain.

(*c*) How accurately would the "smaller" (uncorrelated) predictor predict traces for this loop? Explain.

Problem 3, continued:

(*d*) Explain how the trace cache, including the next trace predictor and other relevant hardware, would have to be modified so that an indirect jump (such as `jr s4`) need not be the last or penultimate instruction in a trace. (See the next part before completing this one.)

(*e*) Suppose all traces for the loop started with the first instruction (`lbu`) and held an entire iteration. If necessary, modify your next trace predictor so that it predicts such traces with close to 100% (after warmup) when processing the repeating string used above.

(*f*) Explain what it is about your next trace predictor that enables it to predict the traces that start with `lbu` with 100% accuracy.

Problem 4: Skipper (Cher 1999) does not skip indirect jumps, but for programs like gcc and TEX that would be useful because many indirect jumps are mispredicted. [10 pts]

(*a*) Explain how skipper would have to be modified to skip indirect jumps emitted for switch statements, like `jr s4` in the previous problem. In particular, discuss any changes needed to the mechanisms used for collecting information stored in the SCIT (skipped computation information table).

Most data in a SCIT entry for jumps can be collected by mechanisms similar to those for branches. There is one tricky item in a SCIT entry, concentrate on that one.

(*b*) Show a SCIT entry for the jump in the previous problem.

Problem 5: Consider once again the code from Problem 3. Suppose the first load, `lbu`, occasionally misses the L2 cache, requiring a painfully long wait for the data. Also suppose that thee indirect jump (`jr s4`) is not predicted accurately (perhaps because the input data does not have a short repeating sequence, as it did in Problem 3). [15 pts]

(*a*) Using anything covered in class (or something from outside class, or something you just made up) devise a way of either avoiding the `lbu`'s cache misses or the loss in performance due to those cache misses.

(*b*) Using anything covered in class (or something from outside class, or something you just made up), except skipper, devise a way of either avoiding the indirect jump's mispredictions, or the loss in performance due to those mispredictions. Remember that unlike Problem 3, the jump cannot be accurately predicted, perhaps because the input data is essentially random.