Name Solution

Computer Architecture & Implementation

EE 7700-4

Midterm Examination

3 November 1998, 18:00-18:50 CST

Problem 1		(30 pts)
Problem 2		(20 pts)
Problem 3		(50 pts)
Exam Total		(100 pts)

Alias Solution

Problem 1: The code fragment below is to be run on a 3-PE trace processor with 16-instruction traces and two functional units per PE. The loop is held by exactly one trace, `Trace 1231: L0, L1, L2, L3, L4, L5, L6`; where 1231 is the trace number. Operation latencies are the same as the chapter 3 and 4 Hennessy and Patterson DLX implementations. Within a PE results are fully bypassed, between PEs there is an extra cycle delay before results can be used. The loop runs for many iterations, next-trace prediction is perfect. (30 pts)

```
LOOP:
L0:  add  r3, r1, r2
L1:  lw   r3, 0(r3)
L2:  add  r3, r3, r6
L3:  andi r6, r3, #0x39
L4:  addi r1, r1, #0x4
L5:  slt  r3, r1, r4
L6:  bneq r3, LOOP    ! For answers below branch always taken.
```

(*a*) Identify the live-in, live-out, and local registers in the trace. How much storage is needed for local register values?

Live-ins: r1 (L0), r2 (L0), r4 (L5), and r6 (L2).

Local: r3 (L0, L1, L2).

Live-outs: r1 (L4), r3 (L5), and r6 (L3).

Storage needed for the three local registers (r3 will be renamed to three different registers, storage is needed for each one).

(*b*) In the code above, which data values will the trace processor predict? Which values would it likely predict well? Justify your answer.

The trace processor will predict the live-ins and the load address.

Values of register r1 and the load address form stride sequences and so are easy to predict. Values of r2 and r4 do not change while the loop iterates and so are very easy to predict. The value of r6 depends upon loaded data, prediction ease depends on the data, varying from easy to impossible.

Problem 1, continued. The code from the previous page is repeated for convenience.

```
LOOP:
L0:  add  r3, r1, r2
L1:  lw   r3, 0(r3)
L2:  add  r3, r3, r6
L3:  andi r6, r3, #0x2AA
L4:  addi r1, r1, #0x4
L5:  slt  r3, r1, r4
L6:  bneq r3, LOOP   ! For answers below branch always taken.
```

(*c*) Show the execution of the code above on the trace processor using the diagram below, assuming the trace processor does *not* use data prediction.

On the diagram indicate when each instruction is dispatched (starts to execute) using the line numbers (*e.g.*, `L0, L1`). Using vertical lines or by boxing the line numbers belonging to a trace, show when each trace is issued to a PE and when it completes[1]. Initially all PEs are idle and all register values are available. Loads always hit the cache.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PE 0	L0	L1		L2	L3		L0	L1					L2	L3		L0	L1				
	L4	L5	L6				L4	L5	L6							L4	L5	L6			
PE 1			L0	L1			L2	L3		L0	L1					L2	L3		L0	L1	
			L4	L5	L6					L4	L5	L6							L4	L5	L6
PE 2					L0	L1				L2	L3		L0	L1					L2	L3	
					L4	L5	L6						L4	L5	L6						

Assumptions: an instruction completes in the cycle after it dispatches. For that reason traces don't end until one cycle after the last instruction.

(*d*) What is the issue rate (execution speed in instructions per cycle) of the trace processor running the loop above for a large number of iterations (on the trace processor without data prediction)?

Execution timing starts a repeating pattern in PE 0 at cycle 6; the other PEs execute with the same pattern. In 9 cycles 21 instructions execute for an issue rate of 2.333 IPC.

[1] There is only one trace, but it is executed for each iteration of the loop. Assume the processor re-issues a trace to a PE even if it is already present.

Problem 1, continued. The code from the previous page is repeated again for convenience.

```
LOOP:
L0:  add  r3, r1, r2
L1:  lw   r3, 0(r3)
L2:  add  r3, r3, r6
L3:  andi r6, r3, #0x39
L4:  addi r1, r1, #0x4
L5:  slt  r3, r1, r4
L6:  bneq r3, LOOP   ! For answers below branch always taken.
```

(*e*) Show the execution of the code on the diagram below on a trace processor that *does* use data prediction. Assume that data prediction is perfect.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PE 0	L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1
PE 1		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6	
PE 2			L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6		L0 L1	L4	L2 L5	L3 L6

(*f*) What is the issue rate of the trace processor running the loop above for a large number of iterations (on the trace processor with perfect data prediction)?

Because of perfect data prediction instructions do not have to wait for data from another PE, and so a repeating pattern of execution starts immediately. In 5 cycles 21 instructions are issued, for an issue rate of 4.2 IPC.

Problem 2: Write a single code fragment that will (*i*) run faster on a machine with a store barrier cache than one that assumes all unresolved store/load pairs are dependent; (*ii*) run faster on a machine using the store-set dependency scheme than on one using a store barrier cache; and (*iii*) run faster on a machine using memory renaming than on one using store-sets. Give register values and any other state needed so that the program has the necessary behavior.

Briefly explain how each of the load/store dependency schemes mentioned speeds execution of the program. (For reduced credit, two or more programs can be written.) (20 pts)

```
! Solution
     lw r2, 0(r1)    ! Cache miss, so r2 not available soon.
     lw r3, 4(r1)    ! Cache miss, so r3 not available soon.

S1:  sw 0(r2), r10
L1:  lw r20, 0(r4)   ! r2 != r4

S2:  sw 0(r3), r11
L2:  lw r21, 0(r5)   ! r5 == r3
L3:  lw r22, 0(r6)   ! r6 != r3 != r2

     add r15, r15, r20
     add r16, r16, r21
     add r17, r17, r22
```

In the code above, the addresses needed by the store instructions cannot be computed when the instructions issue because of the cache misses. (If the addresses were available there would be no reason to predict dependencies.)

The loads at L1 and L3 do not depend on the stores (see comments) so they can always be executed immediately.

In the conservative scheme both loads will wait for the store addresses.

If a store-barrier cache is used then the store at S2 will be part of a dependency violation the first time the code is executed. The second time, the following loads must wait for S2's address to be computed (even though L3 doesn't have to). Since only one non-dependent load waits this is faster.

If the store-set scheme is used then S2/L2 will be identified as a dependent pair and so neither of the non-dependent loads will wait. Execution is thus faster than with a store-barrier cache.

In the preceding schemes the adds would have to wait for the loads to complete, the second add had to wait in all three systems. With memory renaming the pair S2/L2 will be identified as dependent and the data to be written by S2 will be speculatively used as the result of the load L2, so the second add does not have to wait. Execution is therefore faster than using the store-set scheme.

Problem 3: Answer each question below.

(*a*) Two simulation techniques, dynamic compilation and augmentation, are being considered for evaluating two proposed architectural features, a new ALU instruction and additional integer registers. (The new ALU instruction computes the number of `1`'s in the operand's binary representation. The number of integer registers is increased from 32 to 64.) Compilers are available that can emit code for each feature. Accurate timing data is *not* needed.

If feasible, how would a dynamic compilation system that can simulate the unmodified ISA (for example Shade) be changed to simulate each of the two proposed features? If feasible, how would augmentation be used to simulate each feature? If it is infeasible for either simulation technique to simulate either feature, explain why. (10 pts)

Augmentation & new instruction: augmentation replaces the assembler opcode for the new instruction with a sequence of instructions that perform same operation.

Augmentation & extra registers: Every instruction that uses a register number higher than 31 is replaced by code that reads or writes the register values from a special area of the stack prepared by the augmentation system into host registers, then executes the instruction, stores the result if its register number is higher than 31, and restores the host registers to their previous values. This is barely feasible because almost every instruction would have to be replaced.

Dynamic compilation & new instruction: Add a new code sequence that simulates instruction.

Dynamic compilation & extra registers. Increase the size of the array used to hold register values. Modify the code that parses instructions since the register operand fields are each one bit larger.

(*b*) As described in class, a gshare branch predictor in which the size of the outcome history shift register is smaller than the address size of the BHT performs better when the outcome history is exclusive-ored with the higher-order bits of the branch address than if the lower-order bits were used. (10 pts)

Explain why, describing specific problems that would occur if the lower order bits were used.

Suppose the following alternative scheme were used. The branch predictor has a table of distinct random numbers (uniformly distributed over $[0, 2^n)$ where 2^n is the number of BHT entries, ignore the cost). The branch address is used to retrieve a number from this table, which is exclusive ored with the outcome history. Would it matter how the outcome history was aligned with the number? How would prediction accuracy compare to versions of gshare in which outcome history was exclusive ored with the higher-order and lower-order address bits? Explain.

Because the outcome history and branch address are exclusive-ored, different branch address/outcome history pairs can map to the same BHT entry, degrading performance. This won't happen if the portion of the branch address used is always the same, which is more likely with the higher-order bits.

The predictor using the table of random numbers would perform between gshare in which the outcomes were exclusive ored with the lower-order address bits and the higher-order address bits.

(*c*) When a processor discovers a load/store dependence misspeculation twelve instructions past the load have been issued and ten of them (none of which are branches) are data-dependent on the load. Explain why re-execution would be faster than squashing in this situation. Explain why re-execution would be faster even if all instructions following the load were data-dependent. (10 pts)

The two instructions don't have to be executed again and—more importantly—the other ten instructions do not have to be fetched, decoded, and issued again.

(*d*) Do the ILP limits obtained by Wall (described in Section 4.7 of Hennessy and Patterson) apply to processors (*i*) using multiple-path eager execution? (*ii*) trace processors without data prediction? (*iii*) trace processors using data prediction? Explain. (10 pts)

Short answer: They apply to all but trace processors using data prediction. With data prediction instructions can be executed earlier (thus more a time) than true dependencies would allow.

Long answer: The goal of Wall's study was to determine a bound (or limit) on how fast a processor could execute. The ultimate limit on execution speed assumed by the study was true data dependence. The execution speed (in instructions per cycle) was determined for an ideal processor that executes instructions as soon as their operands are computed or loaded. To achieve this the ideal processor would need perfect branch prediction, an unlimited number of functional units, unlimited fetch bandwidth and an unlimited number of memory ports, and perfect load/store dependence prediction. If implemented using reservation stations an unlimited supply would be needed, instead the study considered a separate instruction window and physical register store, both of unlimited size. Wall also looked at systems in which the branch prediction, load/store dependence prediction, and window size were less than ideal.

The three exotic systems in the problem differ from the ideal and other systems considered by Wall; the question is, do those differences give the exotic systems the potential to outperform the ideal system? Since only true data dependencies delay the execution of instructions in the ideal system to execute faster the exotic systems would have to allow instructions to execute sooner than true-data-dependency constraints would allow.

With eager execution a processor can get around the problem of inaccurate branch prediction by executing both sides of a branch. At best—with infinite resources—such a processor would execute as fast as one having perfect branch prediction, as does the ideal processor, and so the limits still apply.

A trace processor uses a trace cache and next-trace prediction to speed execution. The trace cache speeds execution over an ordinary processor by providing blocks of instructions (possibly containing control transfers) aligned and ready to execute. This might be faster than an ordinary processor in which only contiguous blocks of instructions can be fetched, but it does not exceed the ideal processor's unlimited fetch bandwidth capability. The partitioning of execution resources into processing elements (PEs) in a trace processor *reduces* the execution speed over a conventional processor with the same resources and so certainly does not allow it to outperform the ideal processor. For these reasons the limits apply to trace processors without data prediction.

A trace processor can use data prediction for live-in registers and addresses, allowing instructions to execute before their operands are available, (possibly before they are computed). The ideal processor would have to wait until a value was produced (the true dependency constraint), and so might be outperformed by a trace processor with data prediction. Therefore Wall's limits do not apply.

(*e*) What are the outcome patterns in a fixed-pattern confidence estimator, how are they used, and why were they chosen? (10 pts)

Would a similar technique for a gshare or gselect branch predictor (possibly with different fixed patterns) be effective? Explain.

The patterns used are all ones, all zeros, and alternating ($010101\cdots$ and $101010\cdots$). It is used in per-branch history schemes; if a branch's history matches one of the patterns the prediction is given high confidence. They were chosen because branches exhibiting such behavior in test programs were accurately predicted.

The fixed pattern scheme works because it identifies mostly taken and alternating branches. In a gshare or gselect predictor such branches could not be identified by looking at the outcome history (since it includes preceding branches). It is unlikely that a pattern of preceding branches (from multiple branch instructions) could be statically correlated with high prediction accuracy because they would only very rarely fit a neat pattern such as all taken, or all not taken.