Name Solution

Computer Architecture and Implementation
EE 7700-4
Final Examination

7 December 1998, 15:00-17:00 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias Solution

Problem 1: The program below runs on a processor using PAp branch prediction and a JRS confidence estimator. The PAp branch predictor uses four branch outcomes (per branch history register) and the tables are large enough so there is no interference between branch instructions. The branch predictor uses two-bit saturating counters. The mispredict distance counter table has one entry per instruction (also no interference). The high confidence threshold is five. All table entries are zero when the code below starts executing.

```
/*             0  1  2  3  4  5  6  7   */
int *bt[] = { 0, 1, 1, 0, 1, 1, 0, 0 }; /* 8-element array. */

for(i=0; i<100; i++) for(j=0; j<8; j++) if( bt[j] ) x[i]++;
```

(*a*) Find the prediction accuracy of the branch in the `if` statement over the entire execution of the code (`i=0` through `99`). (7 pts)

← i=0 →						← i=1,2,3,... →					
j	Past 4 Outcomes	Next Outcome	Old Count	Correct	Old MD Counter	j	Past 4 Outcomes	Next Outcome	Old Count	Correct	Old MD Counter
0	0000	0	0	y	0	0	1100	0,0,0,...	0,0,0,...	y,y,y,...	2,2,3,...
1	0000	1	0	n	1	1	1000	1,1,1,...	0,1,2,...	n,n,y,...	3,3,4,...
2	0001	1	0	n	0	2	0001	1,1,1,...	1,2,3,...	n,y,y,...	0,0,**5**,...
3	0011	0	0	y	0	3	0011	0,0,0,...	0,0,0,...	y,y,y,...	0,1,**6**,...
4	0110	1	0	n	1	4	0110	1,1,1,...	0,0,0,...	n,n,n,...	1,2,**7**,...
5	1101	1	0	n	0	5	1101	1,1,1,...	1,2,3,...	n,y,y,...	0,0,0,...
6	1011	0	0	y	0	6	1011	0,0,0,...	0,0,0,...	y,y,y,...	0,1,1,...
7	0110	0	1	y	1	7	0110	0,0,0,...	1,1,1,...	y,y,y,...	1,2,2,...

Notes: For branch outcome, 1 indicates taken, 0 indicates not taken. Branch prediction counters saturate at 3. Same branch outcome table entry used at `j=4` and `j=7` (a collision), resulting in mispredictions whenever `j=4`. A single misprediction distance counter is used for the branch, its values are in the "Old MD Counter" column. Bold entries indicate high-confidence predictions.

As can be seen, predictions reach a stable pattern at `i=3`. Before `i=3` 24 predictions are made, 14 of them correct. At `i=3` 7 of 8 predictions are correct. For the entire execution the prediction accuracy is $\frac{14+7\times97}{800} = \frac{693}{800} = 0.86625$.

(*b*) What is the PVP and sensitivity of the JRS confidence estimator on the branch in the `if` statement from `i=50` through `i=99`? (7 pts)

Since predictions and confidence assignments are stable starting at `i=3` one only need consider a single outer iteration, say `i=3`. In such an iteration 3 branches are assigned high confidence and two are correct, for a PVP of $\frac{2}{3}$. Of the 7 correct branches only 2 are assigned high confidence for a sensitivity of $\frac{2}{7}$.

(*c*) Is the program above a good demonstration of the JRS confidence estimator? If it's not describe how the program could be modified to better demonstrate the confidence estimator. (6 pts)

This is not a good demonstration of a JRS confidence predictor because the repeated misprediction forces many accurate predictions to be assigned low confidence. In a better demonstration the table of numbers would not result in a collision in the branch predictor tables and so high confidence would be assigned to all branches after a short warmup.

Problem 2: The DLX assembler program below examines elements of an array. The program is to be run on a simultaneous multithreading processor based on a 4-way superscalar processor similar to the one described in Tulleson 96. The program runs as four threads, the value of `r1` when the code below starts is different in each thread.

The instruction fetch unit can fetch four consecutive and aligned instructions per cycle. Round-robin thread selection is used.

There are an unlimited supply of functional units. The results from ALU operations and loads that hit the cache are available in next cycle.

The memory system can support up to two outstanding cache misses. Cache miss delay is 50 cycles (regardless of address).

There are a total of 40 reservation stations. Reservation stations are not dedicated to a particular functional unit so an instruction can move from any reservation stations to any functional unit.

Branch prediction is perfect.

```
 ! r1: Base address.     (Each thread has own value.)
 ! r2: Stride, in bytes. (Same in all threads.)
 ! r3: Last value in array.  (Same in all threads.)
LOOP:
 lw  r7, 0(r1)        ! Load value.
 add r1, r1, r2
 beqz r7, SKIP        ! Goto SKIP if r7 zero
 addi r4, r4, #1
 add r5, r5, r7
SKIP:
 seq r6, r7, r3       ! Look for terminating value.
 beqz r6, LOOP        ! Branch taken if value not found.
```

The questions below ask for scenarios in which certain things become execution bottlenecks or are operating well. The answers might specify the state of the cache (which lines are present), the values in registers, values loaded from memory, etc. as long as they are not specified in the comments and consistent with the code.

Continued on next page.

(*a*) Under what conditions would instruction fetch bandwidth limit performance? What is that worst-case fetch bandwidth? Under what conditions would instruction fetch bandwidth be at a maximum? What is the maximum fetch bandwidth? (8 pts)

Fetch bandwidth is the average number of useful instructions fetched per cycle. To determine fetch bandwidth one must know how the instructions are aligned and whether the first branch is taken.

In a worst case `LOOP & 0x3 == 3`, that is, `lw` is the last instruction in a group and so each time it is fetched (after the first iteration) three useless instructions come with it. If `beqz r7, SKIP` is always taken then the `addi` and `add` following `beqz` will also be useless. In such an iteration five instructions are executed while 12 are fetched in three groups, for a fetch bandwidth of 5/3 or about 1.67 instructions per fetch. (Instructions from only one thread are fetched in a cycle, if all threads are active the per-thread fetch bandwidth is one quarter or about 0.417 instructions per cycle.)

Such worst case fetching will be *the bottleneck* if nothing else is. The one thing that could slow down the code above is data cache misses. So, for fetch bandwidth to limit performance one must have the conditions described above plus no cache misses.

(If there were lots of cache misses the reservation stations would fill, frequently stalling instruction fetch. In such a case a fetch bandwidth of 5/3 instructions per cycle would be more than enough.)

Fetch bandwidth is at a maximum when the `lw` is the first instruction in an aligned group and the first branch is never taken and there are no cache misses. In this case 7 instructions are fetched in two fetches for a bandwidth of 3.5 instructions per cycle.

(*b*) Under what conditions would the round-robin thread selection policy limit performance? Which thread selection policy would provide better performance? (6 pts)

One thread enjoys cache hits, while the other threads get only misses. The loop branch is always taken and the processor has the ability to execute past multiple predicted branches. Frequently all reservation stations will be full, most with instructions from the threads that missed, forcing stalls even when one thread could execute. Since only two outstanding cache misses can be serviced there is no benefit to having multiple load instructions from a single thread issued.

The icount policy would provide better performance since the fetch mechanism would not choose those threads having many instructions in reservation stations, as would threads waiting for the cache. Since reservation stations don't fill other threads are not slowed down.

(*c*) Consider the conditions under which the round-robin selection limits performance as asked for above. Would the performance limit be as severe if the memory system could simultaneously service more than two cache misses? Explain. (6 pts)

No, since time-consuming cache misses could be overlapped. Suppose the total number of cache misses needed to execute the code above was fixed. (This would be the case if the cache were sufficiently large and no other code was running.) A lower bound on the time to execute the code would be the time for the memory system to service the misses. Let M denote the number of misses. If only two simultaneous misses could be serviced the lower bound is $\frac{M}{2} t_{\text{missdelay}}$. The lower bound would be nearly achieved if each time the memory finishes servicing a miss a new load instruction is ready (has been issued and is waiting in a reservation station). If the memory system could handle 10 simultaneous misses the bound would be $\frac{M}{10} t_{\text{missdelay}}$. Since to achieve this bound a load instruction must be ready it makes sense to fetch instructions from a thread even though it has a load is waiting for a miss. Therefore, increasing the number of simultaneous misses in service would reduce the problems with round robin scheduling.

Problem 3: In a *forwarding* directory coherence protocol a miss by one processor to a block in an exclusive state in another processor is serviced by having a message sent directly from the cache holding the block to the one needing it. (Other messages are sent; since they are part of the solution they can't be described here.) Add forwarding to the directory cache coherence protocol presented in the text. The new protocol should do the following:

- Forward for both read and write misses to exclusive blocks. On a read miss the exclusive block should be changed to shared; of course for a write miss the exclusive block should be changed to invalid.
- It's possible that a block to be forwarded is evicted just before a forwarding message arrives. The protocol should work correctly in this case.
- Be sure that a second read miss (at some other processor) after a block enters the exclusive state is handled correctly.
- Be sure that it is not possible to have two caches simultaneously holding exclusive copies of the same block for an unlimited amount of time. (*Hint: This might occur in an incomplete design if there are write misses at two different processors at the same time.*)

The solution should show new states and other information needed at the caches and memories, new protocol messages, and new state transitions. State transitions should indicate how the directory is changed.

Provide time diagrams showing states and messages sent for each of the situations indicated in the bulleted items above. Show the messages and states, but do not show times (*e.g.*, 12 cycles). (20 pts)

New Protocol Messages

Message Name	Path	Cont.	Purpose
Forward read request	Memory to Cache	PA	Tells cache to forward.
Forward write request	Memory to Cache	PA	Tells cache to forward.
Forwarded data shared	Cache to Cache	PAD	Carries forwarded block.
Forwarded data exclusive	Cache to Cache	PAD	Carries forwarded block.
Read forward complete	Requester to Memory	PAD	Indicates forwarding completion.
Write forward complete	Requester to Memory	PA	Indicates forwarding completion.

The letters P, A, and D, in the contents column above are abbreviations for processor, address, and data.

New Memory States

State	Meaning
EtoE	Forwarding of block to service write miss in progress.
EtoS	Forwarding of block to service read miss in progress.

There are no new cache states but there are new cache state transitions.

New Memory State Transitions

Old State $\xrightarrow{\text{Event}}$ New State	Comments and actions.
Exclusive $\xrightarrow{\textit{Write miss}}$ **EtoE**	Send *forward write request* to remote cache (owner). Set owner to requester.
Exclusive $\xrightarrow{\textit{Read miss}}$ **EtoS**	Put owner and requester in directory (they should be the only two entries). Set owner to null. Send *forward read request* to remote cache (former owner).
EtoE $\xrightarrow{\textit{Write forward complete}}$ **Exclusive**	Forward complete.
EtoE $\xrightarrow{\textit{Data write back}}$ **Exclusive**	Eviction before forward message arrived. Write received data to memory. Send *data value reply* message to new owner (requester).
EtoS $\xrightarrow{\textit{Read forward complete}}$ **Shared**	Forward complete. Write received data to memory.
EtoS $\xrightarrow{\textit{Data write back}}$ **Shared**	Eviction occurred before forward message arrived. Write received data to memory. Remove owner from directory and send *data value reply* message to remaining directory entry (the requester).

New Cache State Transitions

Old State $\xrightarrow{\text{Event}}$ New State	Comments and actions.
Exclusive $\xrightarrow{\textit{Forward write request}}$ **Invalid**	Send *forwarded data exclusive* message to requester.
Exclusive $\xrightarrow{\textit{Forward read request}}$ **Shared**	Send *forwarded data shared* message to requester.
Not Pres. $\xrightarrow{\textit{Forwarded data exclusive}}$ **Exclusive**	Prepare new cache line and initialize with forwarded data. Send *write forward complete* message to memory.
Not Pres. $\xrightarrow{\textit{Forwarded data shared}}$ **Shared**	Prepare new cache line and initialize with forwarded data. Send *read forward complete* message to memory.
Not Pres. $\xrightarrow{\textit{Forward write request}}$ NA	Line was evicted, do nothing since memory will get *data write back* sent during the eviction.
Not Pres. $\xrightarrow{\textit{Forward read request}}$ NA	Line was evicted, do nothing since memory will get *data write back* sent during the eviction.

In the table above Not Pres. indicates that the address specified in the message is not present in the cache. In the first two rows where it is used a line is evicted, the state of the evicted line is not shown. In the second two rows the cache is not changed, so there isn't a current or next state.

Time Diagrams

To be added, eventually. If you would like to see it, ask.

Discussion

Under the forwarding protocol three messages bring data to a processor missing a block that is in an exclusive state elsewhere: First the existing *read miss* or *write miss* message is sent from the cache suffering the miss to the home memory. The home memory will respond with a new *forward read request* or *forward write request* message to the cache holding the exclusive block (the remote cache) which will respond with a *forwarded data shared* or *forwarded data exclusive* message to the original cache. Though the data has arrived, the job is not yet done.

An important question is what state to leave the memory in after it sends a forwarding message. To properly handle new requests and an eviction of the exclusive lines two transition states will be added to the memory controller. The **EtoE** state indicates that an exclusive block is being forwarded to another cache where it will be cached in the exclusive state. Similarly, the **EtoS** indicates forwarding on a read miss.

Normally, the new states will be exited when the requesting processor acknowledges receipt of the line, using two new messages: *read forward complete* and *write forward complete*, the former of which contains a copy of the block to update memory.

NAK messages will be sent to third processors attempting to access the memory location while in state **EtoE** or **EtoS**; they will have to retry. If the block is evicted from the remote cache before a forward request is received the memory will receive the *data write back* and complete the original request normally and the remote cache will ignore the forward request.

Problem 4: Answer each question below.

(*a*) Each line in the diagrams below shows, in program order, reads and writes issued by a processor. The letter in parenthesis indicates the address, the value written is shown with an arrow. Position indicates program order on one processor but not necessarily time or order between processors. Before the code is run address a holds 1, address b holds 2, address c holds 3, and address e holds 4. After this initialization the memory at these addresses is only changed by the writes shown below. (8 pts)

The solution is indicated in the diagrams below using this backhand font.

Indicate values returned by reads which could occur on a sequentially consistent memory system. Write the values next to the reads.

Proc. 1:	W(a)← 11	W(b)← 12	R(c)→3	W(e)← 14	
Proc. 2:	R(b)→12	R(a)→11	R(e)→14	W(c)← 13	R(b)→12

Indicate values returned by reads which could occur on a processor consistent (total store order) memory system but not a sequentially consistent memory system.

Proc. 1:	W(a)← 11	W(b)← 12	R(c)→13	W(e)← 14	
Proc. 2:	R(b)→2	R(a)→1	R(e)→4	W(c)← 13	R(b)→2

Indicate values returned by reads which could occur on a partial store order memory system but not a processor consistent system.

Proc. 1:	W(a)← 11	W(b)← 12	R(c)→3	W(e)← 14	
Proc. 2:	R(b)→12	R(a)→1	R(e)→4	W(c)← 13	R(b)→12

(*b*) Consider the following modification to the directory cache coherence protocol presented in the text. On a write to a block in the shared state, rather than invalidating other shared copies, the protocol will send the new value to each cache holding a shared copy. Is the memory system under this protocol coherent? If not, provide an example. (8 pts)

The protocol is not coherent since the order of writes made by two processors close in time may be seen differently by the two processors.

In the example below both processors write address a at the same time and then they execute two reads to that address. On the first read each processor reads the value it wrote. In both processors, after the first read but before the second the update messages arrives and so the second read returns the other processor's value. Since two write orders are seen the memory system is not coherent.

Proc. 1:	$W(a) \leftarrow 1$	$R(a) \rightarrow 1$	$R(a) \rightarrow 2$
Proc. 2:	$W(a) \leftarrow 2$	$R(a) \rightarrow 2$	$R(a) \rightarrow 1$

(*c*) A lock is implemented on a multiple-processor Tera MTA by a synchronizing store (using full/ empty bits) and on a multiprocessor using store-conditional and load linked instructions:

```
TEST:
 ll   r2, 0(r1)
 bnez r2, TEST     ! If locked, check again.
 addi r2, r0, #1   ! It's not locked! Prepare a "locked" value.
 sc   r2, 0(r1)    ! Try to write locked value.
 beqz r2, TEST     ! If store failed, try again.
```

Suppose when the lock is held multiple processors simultaneously attempt to obtain the lock (and so must wait). Compare the amount of memory traffic generated on the two machines over time.

After a long interval, the processor holding the lock releases it. Describe what happens, including the amount of memory traffic generated, on the two machines. (8 pts)

When multiple processors attempt to obtain the lock both machines will generate memory traffic. The MP will initially generate a small amount of traffic, one access per locker, but no traffic after that. The Tera will generate traffic for a longer period of time (several accesses per locker, the number depending on the retry limit), but no traffic after that.

(In the MP the block holding the lock value is loaded into the cache and repeatedly tested. In the Tera the word will be read from memory (along with the FE bit) and tested. Both the MP and Tera will spin (repeatedly recheck the lock). While the MP spins it just checks the cached value, generating no memory traffic. Each time the Tera checks the lock it will have re-load the value from memory, generating traffic (though the check is done by the hardware so CPU issue slots are not used). After a certain number of failed attempts to find the needed FE bit, the Tera thread waiting for the lock will set a trap bit for the word and wait, generating no more traffic.)

When the lock is relinquished the MP will generate a large amount of traffic, as cached values are invalidated from lockers' caches and later as the lockers attempt to write the lock.

If the memory accesses instructions on the Tera have not yet given up, one will find the relinquished lock. There will be no change in the rate of memory traffic. If, on the other hand, the threads have reached the retry limit then the thread releasing the lock will trigger a trap and the trap handler will select one of the waiting threads. The memory traffic here is generated by accesses to the waiting-locker list (the amount would be smaller than the MP traffic for a sufficiently large number of lockers).

(*d*) Describe possible execution speed advantages or disadvantages of having smaller traces and larger traces in a trace processor, assuming hardware cost is kept roughly constant (same number of PEs, functional units, etc.). (8 pts)

Smaller traces: there would be fewer branches per trace and so there would be fewer distinct traces and a greater chance of finding the needed trace in the trace cache (because of a shorter warm up time and because of fewer evictions).

Larger traces: Higher fetch bandwidth. More instructions would use local variables and so would benefit from the fastest result bypass (in systems without data prediction). With more instructions in a PE more functional units could be kept busy.

A disadvantage of larger traces (with the number of PEs held constant) is that, since a greater number of instructions are being fetched, there would be more frequent reexecution due to next trace and data mispredictions.

(*e*) Both a finite context method data predictor and a stride data predictor could predict the value of a loop index. Give two advantages of a stride data predictor for predicting a loop index. Be specific. (8 pts)

Much less memory would be needed per loop. The number of iterations that could be predicted using an FCM predictor is limited by its size. The FCM predictor would have to encounter a complete execution of the loop before it could start making predictions.