Name Solution

Multiprocessors, Etc.

EE 7700-2

Final Examination

11 December 2003, 15:00–17:00 CST

Problem 1		(35 pts)
Problem 2		(15 pts)
Problem 3		(50 pts)
Exam Total		(100 pts)

Alias Ummmmm

Good Luck!

Problem 1: (35 pts)In the code fragment below the loop is executed for many iterations. The address used by the first load is never the same as the address used by the second load. Line labels are provided for convenience.

```
LOOP:
 A: lw r1, 0(r2)      # r1 = Mem[r2]  Loaded addresses unpredictable.
 B: lw r3, 0(r1)      # r3 = Mem[r1]
 C: addi r3, r3, 1    # r3 = r3 + 1
 D: sw 0(r1), r3      # Mem[r1] = r3
 E: bneq r2, r4, LOOP # if r2 != r4 then goto LOOP after next insn.
 F: addi r2, r2, 4    # r2 = r2 + 4  (Part of the loop body.)
```

(*a*) Draw a dataflow graph covering two iterations. Use ellipses ($\cdots$) to show how the dataflow graph would continue for additional iterations.

(*b*) Using the dataflow graph determine the ILP (ideal IPC) for a large number of iterations.

Assuming the address used by the store rarely is used again sooner than three iterations, the ILP will be six.

Problem 1, continued:

(*c*) Describe the potential for each of the techniques below to improve the performance of the loop over the base system described below.

The base system is 8-way superscalar and dynamically scheduled with a 128-entry reorder buffer and good branch prediction. There is a cache but performance is affected by cache misses. A load suffering an L2 cache miss must wait at least 100 cycles for the data.

If there is potential for improvement using the technique, **explain how that is achieved.** (Explain what is going faster, not details of the mechanism.) If there is little or no potential for performance improvement, **briefly explain why not.**

The loop has three performance limiters, each of which can be overcome by at least one of the techniques below.

The two loads, when they miss the cache, are performance limiters. The loads **are not** on the critical path, so their impact on performance is **not** due to lengthening the critical path (as determined by the dataflow graph analysis). Instead, they limit performance by stalling commit long enough so that the system cannot catch up, at least with a 128-element reorder buffer. See the solution to the 1000-entry ROB part for a detailed explanation.

The third performance limiter is the `F` instruction which limits execution to 6 IPC. That limiter is easy to remove since it just increments a register and its easy to determine the register value, say, at the 50th iteration.

☑ A system with a 1000-entry reorder buffer and other resources proportionately increased.

A larger reorder buffer would realize significant performance improvement.

Suppose the base system has been running for a while without suffering a cache miss and suppose the base system could sustain a fetch rate of eight instructions per cycle (perhaps using a MBP and multiported cache). Because instructions are fetched at a rate of eight per cycle but commit at a rate of six per cycle (due to the loop-carried dependence through the `F:addi`) the ROB will be full and the action will be taking place near the head of the reorder buffer. Part of that action is the execution of `A:lw`, suppose after a long string of hits it finally misses. Instructions B, C, and D will have to wait but the others go on executing. In particular the `F:addi`'s in this and later iterations continue to execute and so do the `A:lw`'s, which will probably miss the cache too since access is sequential. The 128-entry ROB can hold 21 iterations, and so by the time the load completes 100 cycles later execution will have reached the tail of the ROB. The data that arrives for the `A:lw` at the head will probably also be used by the `A:lw` in the next iteration, and so on, since access is sequential and a cache line can hold multiple words (on any reasonable system). Several of those `A:lw` will then complete execution simultaneously, followed by the B and C instructions. Thus within three cycles or so instructions making up the x iterations at the head will be finished and ready to commit, where x is the number of words on a cache line. Those instructions will commit at a rate of eight instructions per cycle. If this eight instruction per cycle commit keeps up long enough it will make up for the 100-cycle stall in commit and so the overall execution rate will still be six instructions per cycle. However, with a 128-entry ROB the 8 IPC commit will not be sustained long enough. To make up for a 100-cycle gap commit would have to be at 8 IPC for 300 cycles. At best, when the load completes the 127 other instructions in the ROB will be complete by the time they reach the head 16 cycles later and so 8 IPC can be sustained for them. In that time another 16 iterations could complete which would take another 12 cycles to commit. The problem could be solved with one of these $\sum$, but the easier way to solve it is to note that commit is advancing at 8 IPC while execution is advancing at 6 IPC. They start with a separation of 128 instructions and so they meet at the ROB head $\frac{128}{8-6} = 64$ cycles later. Therefore the 128-entry ROB is not large enough to hide the load miss latency. The system with the 1000-entry ROB would be able to sustain 8 IPC for $\frac{1000}{8-6} = 500$ cycles which would be enough.

Having a second `A:lw` miss would not be a problem because that miss would occur before it reached the ROB head. (When the first `A:lw` misses execution proceed from the head to the tail, and so the `A:lw` execute while the first `A:lw` is waiting for its data.) It's only not a problem because the `A:lw` address is determined from an add instruction which does not depend on the load that missed.

Even with a 1000-entry ROB some performance would be lost if there was a miss by `A:lw` followed by a miss by `B:lw`. This is a problem because B must wait for A, the two stall commit for 200 cycles and it would take 600 cycles of 8 IPC commit to catch up, and the ROB is just not big enough.

In all of the other performance improvement techniques the ROB size is limited to 128 entries per processor. ("Per processor" applying only to the multiprocessor.) Therefore the only way to avoid the performance loss from the loads is to one way or another prefetch them.

☑ Prefetch. (Indicate the simplest kind that would be effective, if any.)

Prefetch would have only minor improvement since the second load would not be helped by prefetch.

Because its addresses are sequential the first load is easy to prefetch. Prefetch instructions are the easiest solution. To prefetch a load 100 cycles before it is needed add instruction `pref 400(r2)` to the loop, where `pref` is the MIPS32 prefetch instruction. The offset of 400 is the product of 4 characters per load, one load (and iteration) per cycle, and 100 cycles. Because the system is 8-way superscalar the inclusion of the prefetch instruction will not slow down execution, it's still one cycle per loop iteration. (It would not be fair to call it faster because it's going at 7 IPC.) If the system were 6-way superscalar then the addition of a seventh instruction to the loop would slow down execution. Fortunately it's 8-way.

Of the hardware prefetch schemes, tagged prefetch would allow maximum execution rate only if the line size were at least 400 bytes (realistically, 512 bytes). Otherwise, the prefetched line would arrive late, though sooner than if it had not been prefetched at all. Adaptive sequential prefetch would be able to fetch enough lines to keep up, though it would waste cache space prefetching `B:lw`. A stream buffer would be best because of the sequential access. Stride prefetch would work but it's higher cost would not be necessary for the sequential pattern.

Unfortunately, `B:lw` cannot be prefetched because its addresses are unpredictable. The only technique for which correct prefetch can be seen is anthropic prefetch, in which a prefetch address is randomly generated. If the address turns out to be wrong the human observers' memory is erased.

☑ Pre-Execution

Pre-execution can potentially speed the loop above because the p-thread load initiating the fetch is executed well in advance and does not wait around for the data. To be completely effective the p-thread would have to prefetch a load 100 cycles in advance, and so would consist of a bunch of F's followed by `A:lw` and `B:lw`. If the p-thread were optimized the F's could be replaced by a single `addi r2, r2, 400` instruction or eliminated altogether and the A changed to `lw r1, 400(r2)`.

If the p-thread shared decode and rename logic with the main thread then it would have to consist of two instructions (the modified A and B), otherwise it would slow down the main-thread fetch rate to below 6 IPC. There would also have to be enough load/store unit ports to handle four accesses per cycle, the p-thread A and B instructions and their main-thread counterparts.

Problem 1, continued:

✓ Implicit Multithreading (IMT)

IMT would not realize much performance gains. A modified form of IMT might yield some gains.

If the IMT task corresponded exactly to the loop body shown above then IMT would not help because there would be no way to execute the loads early. If the increment of `r2` were moved to the beginning of the loop and if the IMT scheduler fetched only the first three instructions of each task until all PUs were full, and if there were lots of PUs then it might realize significant speedup. Moving the increment to the beginning of the loop is something the IMT (multiscalar) compiler might do. However, there is nothing in IMT that would limit fetch to the first three instructions until all PUs were full.

✓ Data Prediction

In the absence of cache misses it's possible that data prediction (coupled with a confidence estimator to avoid wasting resources on low-confidence predictions) could achieve an IPC of 8 by predicting the result of the easily predictable `F:addi`. With such data prediction loads would miss near the tail of the reorder buffer instead of the head and would have about 14 cycles to get data and so commit would be blocked for about 86 cycles. Because of the 8 IPC without cache misses, this system would be faster than the base system. It might be faster than the large reorder buffer system if cache misses were relatively rare. If cache misses were frequent then the steady 6 IPC achievable with a larger reorder buffer would be better than the sometimes-but-not-always 8 IPC with data prediction. Note that data prediction could not be used to predict the address of the `B:lw` because the problem stated the address is unpredictable.

✓ Dynamic Optimization (rePlay or optimization cache)

In the absence of cache misses rePlay could do several things to improve execution by combining several iterations (unrolling the loop). After rePlay detects that the loop iterates many times it could attempt to combine several, say 4, iterations into a frame. The combined iterations might look like:

```
# Before entering loop check that number of iterations is a multiple of 4.
LOOP:
 A: lw r1, 0(r2)      # r1 = Mem[r2]
 A1: lw r12, 4(r2)
 A2: lw r13, 8(r2)
 A3: lw r14, 0xc(r2)
 B: lw r3, 0(r1)      # r3 = Mem[r1]
 B1: lw r15, 0(r12)
 B2: lw r16, 0(r13)
 B3: lw r17, 0(r14)
 C: addi r3, r3, 1    # r3 = r3 + 1
 C1: addi r15, r15, 1
 C2: addi r16, r16, 1
 C3: addi r17, r17, 1
 D: sw 0(r1), r3      # Mem[r1] = r3
 D1: sw 0(r12), r15
 D2: sw 0(r13), r16
 D3: sw 0(r14), r17
 E: bneq r2, r4, LOOP
 F: addi r2, r2, 16
```

The original code used six instructions per iteration the new code uses $\frac{4\times4+2}{4} = 4.5$ instructions per iteration (where the new loop body does four iterations worth of work). The critical path through the loop body is still just instruction `F` and so the new code could execute at a rate of 8 IPC or $\frac{8}{6} = 1.33$ iterations per cycle on the base machine and 18 IPC or four iterations per cycle on an 18-way machine with sufficient resources. (If compiler or programmer had unrolled the loop 8 IPC would be achievable in the absence of cache misses on the base machine.)

Because the ROB is still 128 entries and a load does not start until the load instruction is in the ROB the system using rePlay would still suffer performance degradation due to cache misses.

Problem 1, continued:

☑ SMT (Simultaneous Multithreading), code can be modified.

Without code modification there could be no advantage at all on an SMT. To take advantage of SMT the code must be split into multiple threads. If each B address is used at most once then this is easy, divide the code into two threads. If the loop iterates 100 iterations one thread might do 0-49, the other 50-99. The modified loop body would be identical to the one given in the problem, the initialization code before the loop, which is not shown, would be different. In the modified code the initial value of `r2` and the value of `r4` for the two threads would be different.

If cache misses are far enough apart or if the SMT can support enough threads then when one thread suffers a cache miss other threads can compute for the 100 cycles it would take to get the data. A cache line that holds 16 words will provide enough data for 12 cycles ($\frac{16\times 6}{8}$) of full-speed fetch or commit. Each thread can generate at most one cache line miss per 16 cycles (for `A:lw`, we are assuming for now `B:lw` hits) and assuming each thread has a 128-entry ROB it can overlap two misses. Thus it would be committing or fetching full speed about $\frac{2\times 12}{100} = 24\%$ of the time. Four such threads, if they were scheduled properly would keep things busy nearly 100% of the time. Actual performance would be lower since misses would not be perfectly overlapped. For an individual thread there 10-cycles would elapse between data for load arriving and the next miss being generated.

With enough threads the code could execute at 8 IPC despite the A misses. The latency of the B misses would also be covered, though more threads might be needed.

The discussion above was based on the assumption that an address for the B load is never used twice. If that assumption does not hold then the load/increment/store might not work properly because the `B:lw` in one thread could load from the same address as the `B:lw` in another thread before the other thread does a store. If so, the stored value would only be incremented by 1, not 2 as it should be. The solution is to use an atomic increment instruction (not covered in the course) or some other kind of synchronization instruction, for example, one that would store the incremented value only if the previous value is the same as the one loaded. The overhead for such instructions might be low, more elaborate solutions such as semaphores would have higher overhead that might make up for any benefit.

☑ Multiprocessor, code can be modified.

The multiprocessor code modifications are similar to the SMT code modifications. Unlike the SMT, there is no upper bound on how fast the code could be executed as long as enough processors were available. Each processor would each suffer from load misses but because there could be many processors execution could outperform the base system or the SMT. Because each iteration is independent of the others (assuming B addresses not used twice) the code would enjoy linear speedup.

As with the SMT, some kind of synchronization would be needed to prevent the B load on two processors from simultaneously accessing the same location.

Problem 2: (15 pts)The code below runs on a multiprocessor using a write-invalidate, directory-based cache coherence protocol.

It takes more than 1 ns to send a message between any two processors, and any particular access can be completed in much less than an hour. The time needed to send a message from any processor to the home memory of address `0x1000` is identical. Initially the caches are empty.

☑ In time order, show the cache states, directory states, and messages that will be sent to execute the code below.

```
Proc 1, t = 0
lw 0x1000
```

```
Proc 2, t = 1 hour
lw 0x1000
```

```
Proc 3, t = 2 hours
sw 0x1000  <- STORE, NOT LOAD
```

```
Proc 1, t = 2 hours + 1 ns
lw 0x1000
```

```
Proc 4, t = 2 hours + 2 ns
lw 0x1000
```

Solution shown on the next page.

```
Solution
t=0
C1: Invalid
C2: Invalid
C3: Invalid
C4: Invalid
M: Not cached
P1: Miss
C1toM: GetShared
...
M: -> Shared 1
MtoC1: SharedData
...
C1: -> Shared

t = 1 hour
P2: Miss
C2toM: GetShared
...
M: -> Shared 1,2
MtoC2: SharedData
...
C2: -> Shared

t = 2 hours
P3: Miss
C3toM: GetExclusive

t = 2 hours + 1 ns
P3: Hit (There is nothing wrong with this.)

t = 2 hours + 2 ns
P4: Miss
C4toM: GetShared

...
(GetExclusive arrives first.)
MtoC1: Invalidate
MtoC2: Invalidate
(GetShared arrives.)
MtoC4: Retry. (Alternative is to queue message.)
...
C1: Shared -> Invalid
C1toM: Acknowledge
...
C2: Shared -> Invalid
C2toM: Acknowledge
...
(Retry received)
C3toM: GetShared
...
(Ack arrives)
M: Shared 1,2 -> Shared 2
...
(Ack arrives)
M: Shared 2 -> Exclusive 3
```

```
MtoC3: ExclusiveData
...
(GetShared Arrives)
MtoC3: Writeback
...
(ExclusiveData arrives)
C3: -> Exclusive
(write completes)
C3: -> Modified
(Writeback arrives.)
C3: -> Shared
C3toM: WritebackData
...
M: Exclusive 3 -> Shared 3,4
MtoC4: SharedData
...
C4: Invalid->Shared
```

Problem 3: Answer each question below.

(a) (7 pts)Explain how a stride prefetcher determine prefetch addresses. If a table is involved (*Hint: there is*) describe how the table is indexed, what is stored in each entry, and how it is used to predict addresses.

A PC-indexed table holds a tag, a state and the last effective address and stride used by the instruction (whose PC was used to index the table). The state indicates how many times the stored stride has been seen for this instruction. If the stored stride has been seen at least once then a prefetch address can be generated by adding the stride to the last effective address used. When the load completes the effective address, stride, and state are updated.

(b) (7 pts)Show how the multiple branch predictor (MBP) covered in class can predict three branches in one cycle without having to do three consecutive PHT lookups.

The PHT has three ports. One port is used for the counter for the next branch. The second port provides counters from two consecutive locations, corresponding to the next branch taken and the next branch not taken. A prediction using the counter from the first port is used to select one of the counters from the second port. A similar procedure is used for the third ports. The address for the first port is the GHR, the address for the second port is the GHR with a zero shifted in, the address for the third port is the GHR with two zeros shifted in.

(*c*) (7 pts)How might hierarchical sequencing of trace cache entries reduce the number of entries used in the trace cache? Show an example.

When there is a misprediction using hierarchical sequencing a new trace is constructed starting with the part of the current trace that has not been mispredicted. Without hierarchical sequencing the new trace would start following the branch. There will be more trace overlap without hierarchical sequencing since traces will start at each mispredicted branch. With more overlap more trace cache entries are needed.

(*d*) (7 pts)Why might it be harder to get speedup with computation re-use than with data prediction?

Computation re-use in many forms requires the values of the source operands; for many instructions it is just as fast to compute the result as to look it up in a re-use table. Data prediction can start when the address of the instruction is known, which is much earlier than when the source values are known.

(*e*) (7 pts)With confidence estimation data prediction accuracy can exceed 90% and result, by some measures, in good performance improvement. Then why is it not practical?

The problem is not the prediction accuracy but the size of the tables. In many studies the predictor tables are larger than the L1 cache.

(*f*) (8 pts)Pre-execution and dynamic optimization using rePlay have similarities.

☑ Describe a situation in which both pre-execution and dynamic optimization using rePlay could effectively pre-execute a load. Describe the steps involved.

Without rePlay or pre-execution a L1 load miss (l2 hit) would add to the critical path starting at the load, perhaps filling the reorder buffer. In the code below a p-thread could easily be constructed for the load and triggered far enough in advance to cover an L2 hit. A rePlay system could construct a frame which contains the instructions below but with the `lw` moved to the beginning while dependent instructions (the `add`) remaining at the end.

```
addi r2, r2, 4
# About 60 instructions, none write r2.
# All branches highly predictable, and all paths lead to insn below.
lw r1,0(r2)  # Frequently misses L1 cache
add r3,r1,r4 # Long chain of dependent instructions.
```

☑ Describe a situation in which pre-execution could effectively (though not perfectly) pre-execute a load but rePlay could not. Clearly indicate what rePlay's weakness is.

The same code as above, except the branches between the `addi` and `lw` are difficult to predict. A misprediction in one of those branches would not stop the p-thread. But rePlay would not even be able to construct a frame for those instructions because branches covered by a frame must be highly predictable.

(*g*) (7 pts)You are developing a branch predictor that will be particularly accurate for program X. Assume that Shade, SimpleScalar, and SimOS could all simulate the same ISA.

☑ Describe an advantage in using Shade over SimpleScalar.

It's fast since it only does functional simulation.

☑ Describe an advantage in using SimpleScalar over Shade.

It provides timing information so effects such as predictor table update times can be taken into account.

☑ Describe an advantage for SimOS.

It includes operating system code, so results would be more realistic.

☑ Would an instrumentation system be appropriate?

No, since the goal is to test a new predictor design. Instrumentation only works on existing systems, if an existing system had the predictor design it would not be new.