Lecture 7: Speculative Execution and Recovery using Reorder Buffer

Branch prediction and speculative execution, precise interrupt, reorder buffer, Tomasulo improvements

Control Dependencies

- Every instruction is control dependent on some set of branches
  - if p1
    - S1;
  - if p2
    - S2;
- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
  - control dependencies must be preserved to preserve program order

Performance Impact

If CPU stalls on branches, how much would CPI increase?

- Control dependence need not be preserved in the whole execution
  - willing to execute instructions that should not have been executed, thereby violating the control dependencies, if can do so without affecting correctness of the program
- Two properties critical to program correctness are data flow and exception behavior

Branch Prediction and Speculative Execution

- Speculation is to run instructions on prediction - predictions could be wrong
- Branch prediction: crucial to performance, could be very accurate
- Mis-prediction is less frequent event - but can we simply ignore?

Example:

```plaintext
for (i=0; i<1000; i++)
    C[i] = A[i] + B[i];
```

Branch prediction: predict the execution as accurate as possible (frequent cases)
Speculative execution recovery: if prediction is wrong, roll the execution back

Exception Behavior

- Preserving exception behavior -- exceptions must be raised exactly as in sequential execution
  - Same sequences
  - No "extra" exceptions
- Example:
  - DADDU R2, R3, R4
  - BEQZ R2, L1
  - LW R1, 0(R2)
  - L1:
    - Problem with moving LW before BEQZ?
- Again, a dynamic execution must produce the same register/memory contents as a sequential execution, any time it is stopped

Precise Interrupts

- Tomasulo had:
  - In-order issue, out-of-order execution, and out-of-order completion

- Need to "fix" the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.
Branch Prediction vs. Precise Interrupt

- Mis-prediction is "exception" on the branch inst
- Continuation "branches out" on exceptions
  - Every instruction is "predicted" not to take the "branch" to interrupt handler
- Some technique for handling both issue: in-order completion or commit: change register/memory only in program order
- How does it ensure the correctness?

The Hardware: Reorder Buffer

- If last write result is in program order, reg/memory always get the correct values
- Reorder buffer (ROB) - reorder out-of-order inst to program order at the time of writing reg/memory (commit)
- If some inst goes wrong, handle it at the time of commit - just flush inst afterwards
- Inst cannot write reg/memory immediately after execution, so ROB also buffer the results
- No result in Tomasulo original

Reorder Buffer Details

- Holds branch valid and exception bits
  - Flush pipeline when any bit is set
  - How do the architectural states look like after the flushing?
- Holds dest, result and PC
  - Write results to dest at the time of commit
  - Which PC to hold?
  - A ready bit (not shown) indicates if the
- Supplies operands between execution complete and commit

ROB: Circular Buffer

- head
- tail

Speculative Tomasulo Algorithm

1. Issue - get instruction from FP Op Queue
   - Condition: a free RS at required FU
   - Actions: (1) decode the instruction; (2) allocate a RS and RDB entry; (3) do source register renaming; (4) do dest register renaming; (5) read Register File; (6) dispatch the decoded and renamed instruction to the RS and RDB

2. Execution - operate on operands (EX)
   - Condition: At a given FU, At least one instruction is ready
   - Action: select a ready instruction and send it to the FU

3. Write result - finish execution (WB)
   - Condition: At a given FU, some instruction finishes FU execution
   - Actions: (1) FU writes to CDB, broadcast to all RS and to the RDB (2) FU broadcast tag (RDB index) to all RSs; (3) deallocate the RS. Note no register status update at this time
Speculative Tomasulo Algorithm

4. Commit—update register with reorder result
   - Condition: ROB is not empty and ROB head inst has finished execution
   - Actions if no mis-prediction/exception: (1) write result to register/memory, (2) update register status, (3) de-allocate the ROB entry
   - Actions if with mis-prediction/exception: flush the pipeline, e.g. (1) flush IFQ; (2) clear register status; (3) flush all R3 and reset FU; (4) reset ROB

Speculative Execution Correctness

For any committed inst i in E(Sp, P), i receives the outputs in E(Sp, P) of its parents in E(Sp, P)

Assume i has a source Rx produced by j. Three possibilities at i rename:
1. Rx is not renamed
   => i receives j's output from the register
2. Rx is renamed and j WB has finished (or finishing)
   => i receives j's output from ROB
3. Rx is renamed, and j EXE has not finished
   => i will receive j's value from CDB broadcasting

And i is reading operands is not affected by later mis-speculatively instructions

Tomasulo Summary

- Reservations stations: implicit register renaming to larger set of registers+ buffering source operands
- Prevents registers as bottleneck
- Avoids WAR, WAW hazards of scoreboard
- Not limited to basic blocks when compared to static scheduling (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
- Don’t stall for L1 data cache miss (enough TLP for L2 miss?)
- Can support memory-level parallelism
- Lasting contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation (discusses later)
- 360/91 descendants are Pentium III; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21064

Speculative Execution Correctness

- E(Sp, P) commits the same set of instructions as E(Sp, P) executes
- For any committed inst i in E(Sp, P), i receives the outputs in E(Sp, P) of its parents in E(Sp, P)
- In E(Sp, P) any register or memory word receives the output of a committed inst j, where j is the last inst that writes to the register or memory word in E(Sp, P)

Code Example

Loop:
  LW R2, 0(R1)
  DADDIU R2, R2, #1
  SW R2, 0(R1)
  DADDIU R1, R1, #4
  BNE R2, R3, Loop
  LW R3, 0(R1)

... How would this code be executed? What if the BNE is incorrect predicted?

Tomasulo Complexity and Efficiency

Can dependent instructions be scheduled back-to-back?

Modern processors employ deep pipeline

=> Can the rename stage be finished in one fast cycle?
Review Tomasulo Inst Scheduling
Both in RS, no contention on CDB or FU
ADD R2, R2, 45 # R2->tag p, result = A
SUB R6, R2, R4 # R4 is ready, = B
Cycle 1: ADD starts at FU, producing A
Cycle 2: ADD broadcast p + A
SUB matches on p and accepts A
Cycle 3: SUB starts execution, FU calc A-B
A is produced at cycle 1, but consumed at cycle 3 -- unavoidable?

Review Data Forwarding
MIPS pipeline data forwarding:
FU/MEM => FU
Why not in Tomasulo?
REG/ROB
FU
bypass
ROB
Cycle 1: If tag is broadcast one-cycle earlier...
Cycle 2: forward A from FU output to FU input...

Review Scheduling
RS1: ADD R6, R2, R4
RS2: SUB R12, R0, R6
RS3: ADD R12, R11, R6
ADD(0) has been ready and selected
1. ADD(0)'s tag is broadcast, and operation sent to FU.
2. SUB is woken up and selected.
3. SUB's tag is broadcast, operation sent to FU.
   Forwarding logic replace 2nd FU opened with FU output. ADD(0) is waken up and accepts FU output. FU is selected.
4. So on and so forth...
RS can be centralized or distributed
One cycle earlier
How to address CDB contention?

How to Handle Variable Latency?
Tag broadcast Cycle n+1
Cycle n
Control data bus
One method: Use result shift register to track latency and control tag/data bus

Revised Pipeline Stages

- As efficient as MIPS pipeline (instruction throughput)
- With data forwarding and bypassing