Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes

12May06

ILP Overview, Branching

James Goodman

Department of Computer Science

Compile-time vs. Runtime

- Basic question: the trade-off between compiler work and runtime work
  - Compilers can do a better job because they can analyze, look into the future
  - Compilers don't have run-time information; must schedule code conservatively for correctness
- First experiments with out-of-order execution (CDC 6600, IBM S/360 Model 91) demonstrated the ability to speed up execution by reordering instructions.
- Cray then demonstrated with Cray-I that, if the timing of all instructions is known precisely, there is little or no benefit to out-of-order execution: instructions can be re-ordered at compile time.
  - Cray exploited the fact that his applications were highly structured programs (oblivious)
  - All loads could in theory be scheduled sufficiently far in advance so that latency didn't matter

Q: What can be done at compile time?

A: Lots, or little.

Three steps for capturing ILP

1. Check dependencies between instructions to determine which instructions can be grouped together for parallel execution
2. Assign instructions to the functional units on the hardware
3. Determine when instruction begins execution

Tasks for ILP Execution

Each of these tasks can be performed at least partially at compile time

1. Compiler indicates which instructions can be executed concurrently (or hardware infers it from the order).
2. Compiler designates a functional unit for each instruction (or the hardware dynamically assigns a free one).
3. Compiler indicates exactly which instructions should be initiated in each cycle (or hardware assures that resources are/will be free and issues when ready).

Instruction-Level Parallelism (ILP)

Overview of selected aspects

- Branch prediction
  - Cost of branches
  - Difficulty of prediction
  - Techniques for prediction
- Out-of-Order (OoO) execution
  - Dataflow
  - Hazard detection
  - Handling exceptions
  - Register renaming
- Speculative execution
  - Why speculate?
  - Benefits

Reducing Branch Costs

- Hennessy & Patterson, Section 3.4

Branch Prediction

When we decode a branch, we have already fetched future instructions

- To execute instructions after a branch we must know
  - whether it is taken (if it is conditional)
  - what the target address is
- In the best case, taken branches are problematic. If the branch is taken, or if the address has not yet been computed, we cannot proceed.
  - We can issue instructions speculatively
  - But which path?
- Issuing on both paths is always expensive and wasteful
  - wastes memory bandwidth, issue bandwidth, functional units, et
  - may do more harm than good

Ideas for Dealing with Branches

- Make branch decision early (no complex comparisons)
- Delayed branch: execute instructions regardless of branch decision
- Predict correct branch decision if possible
- Turn control dependence into data dependence: conditional instruction (predicated execution)

Importance of Good Prediction

- Predicting wrong path is no better than doing nothing (possibly worse)
  - multiple cycles lost
  - getting worse
    - If (expression) is not true don't execute the next X instructions
- Goal: predict with high probability but assess confidence
  - Low confidence prediction: be cautious (maybe do nothing)

Predicting Branches statically

- Assume always taken (or not taken)
- Branch backwards vs. branch forwards
- Based on instruction type (loop vs. if-then-else)
  - Give compiler the opportunity to tell what it knows