Instruction Scheduling

CSE 501
Lecture 13
May 13, 2013
Questions?

Last lecture?

Reading?

Programming assignment?
Code generation

Recall our separation of concerns. Code generation is:

- Instruction selection,
- Register allocation, and
- Instruction scheduling.

This time, we'll talk about instruction scheduling.
Introduction

Instruction scheduling is a way for the compiler to exploit the instruction-level parallelism provided by the architecture.

Depending on the application and the architecture, we might expect benefits ranging from a few percent to factors of 20 or more.

ILP can be exposed in different ways:
• long-latency instructions via pipelining, explicit lookaheads, and branch-delay slots,
• issuing multiple instructions per cycle with VLIW or superscalar

VLIW, explicit lookaheads, and branch-delay slots are exposed in the ISA; pipelining and superscalar are a product of particular implementations.
Implications

If the ILP is an explicit part of the ISA, then we're obliged to pay attention to the problem. But any solutions we come up with will be useful for the lifetime of the architecture.

If the ILP is a part of the particular implementation, we may be able to ignore the problem, or at least treat it pretty abstractly. We'll probably have to tune, or perhaps redesign, for different implementations.
Instruction scheduling

- Driven by processor design and implementation
- Reorder instructions to reduce execution time
- Must produce correct code
- Avoid spilling registers
- Operate efficiently

Sample latencies

- 1 cycle - load immediate, integer add, shift
- 3 cycles - fp add, multiply
- 10 cycles - division
- 0 to 8 cycles - branch
- 2 to 300 cycles - load
Causes of instruction latency

Data hazards
• instructions depend on result of previous instruction

Structural hazards
• limited hardware resources (ALU, FPU, Mem)

Control hazards
• conditional branch depends on result of previous instruction
• instructions guarded by conditional branch depend on direction taken

Hazards causes *bubbles* (or *stalls* or *interlocks*) in the pipeline, increasing execution time
Data hazards

There are several sorts of data hazard:
- read-after-write, one instruction writes a register and a later instruction reads it
- write-after-read, one instruction reads a register and a later instruction overwrites it
- write-after-write, one instruction writes a register and a later instruction overwrites it

Usually the hardware will employ a trick called register renaming to avoid write-after-read and write-after-write hazards.

 Allows the implementation to use many more registers without changing the ISA.
Register renaming

There are a lot of ways to accomplish renaming; I'll try to give you the basic idea.

The register numbers we use in instructions are architectural names. Typically, a processor will have many more physical registers that are not exposed in the ISA.

Instructions read and write registers. For example, the instruction \( r1 = r2 + r3 \) reads \( r2 \) and \( r3 \), but writes \( r1 \).

Wherever we fetch an instruction, we look up the architectural names of the registers it reads (e.g., \( r2 \) and \( r3 \)) to find the physical names of the registers that will hold the actual values (where they are ready). We allocate a new physical name for the architectural register being written (e.g., \( r1 \)).

After all the reads of a physical register are complete, we free it for later use.
Register renaming example

Consider the sequence
\[
\begin{align*}
  r1 &= \text{load}(\text{foo}) \\
  r1 &= r1 + 2 \\
  \text{store } r1 \text{ somewhere} \\
  r1 &= \text{load}(\text{bar}) \\
  r1 &= r1 + 4 \\
  \text{store } r1 \text{ somewhere else}
\end{align*}
\]

If we rewrite it by hand (or by compiler), as
\[
\begin{align*}
  r1 &= \text{load}(\text{foo}) \\
  r1 &= r1 + 2 \\
  \text{store } r1 \text{ somewhere} \\
  r2 &= \text{load}(\text{bar}) \\
  r2 &= r2 + 4 \\
  \text{store } r2 \text{ somewhere else}
\end{align*}
\]

then we could potentially run it twice as fast.
Unfortunately requires twice as many registers in the ISA.
Register renaming example, ...

If we attack it, at runtime, via renaming, we see something like this

\[
\begin{align*}
  r1 &= \text{load}(\text{foo}) \\
  r1 &= r1 + 2 \\
  \text{store } r1 \text{ somewhere} \\
  r1 &= \text{load}(\text{bar}) \\
  r1 &= r1 + 4 \\
  \text{store } r1 \text{ somewhere else}
\end{align*}
\]

\[
\begin{align*}
  r101 &= \text{load}(\text{foo}) \\
  r102 &= r101 + 2 \\
  \text{store } r102 \text{ somewhere} \\
  r103 &= \text{load}(\text{bar}) \\
  r104 &= r103 + 4 \\
  \text{store } r104 \text{ somewhere else}
\end{align*}
\]

Allows the same freedom of execution order without changing the ISA.
Data dependences

Even with renaming, data dependences can restrict our parallelism

\[ r_{1} = \text{load}(\text{foo}) \]
\[ r_{2} = r_{1} + 2 \]
\[ \text{store } r_{1} \text{ somewhere} \]
\[ r_{1} = \text{load}(\text{bar}) \]
\[ r_{2} = r_{1} + 4 \]
\[ \text{store } r_{1} \text{ somewhere else} \]

\[ r_{101} = \text{load}(\text{foo}) \]
\[ r_{102} = r_{101} + 2 \]
\[ \text{store } r_{102} \text{ somewhere} \]
\[ r_{103} = \text{load}(\text{bar}) \]
\[ r_{104} = r_{103} + 4 \]
\[ \text{store } r_{104} \text{ somewhere else} \]

Can we start the \text{load(bar)} before finishing the first store?
Data dependences

Besides the constraints caused by registers, memory references may depend on each other

A reference $Y$ depends on an earlier reference $X$ if there exists:
- flow dependence, $X$ writes a location that $Y$ later reads, or read-after-write
- anti dependence, $X$ reads a location that $Y$ later writes, or write-after-read
- output dependence, $X$ writes a location that $Y$ later writes, or write-after-write

There's a 4th case that doesn't constrain ordering
- input dependence, $X$ reads a location that $Y$ later reads, or read-after-read
Approaches

There's been a lot of research in the area:

- list scheduling, for basic blocks and extended basic blocks
- trace scheduling, traces through routine
- superblobk scheduling, trace through routine
- unrolling + list scheduling, inner loops
- software pipelining, inner loops
- percolation scheduling, complete routine

List scheduling is pretty common for all sorts of machines; software pipelining is less common, but still used for high-performance machines (it's awfully expensive to implement and maintain).
List scheduling

A simple approach:
• maintain a queue of ready instructions by cycle
• at each step:
  a) choose an instruction and schedule it
  b) update all ready queues

Two flavors of list scheduling
• Forward list scheduling - start with available ops working forward. Ready if all operands available.
• Backward list scheduling - start with ops that have no successors and work backwards. Ready if latency covers all uses.

Problems
• how to tell when an instruction is ready?
• how to choose between ready instructions?

Scheduling even straight-line code is NP Complete.
Gibbons & Muchnik

A form of forward list scheduling
- uses a dependence DAG to model constraints, where nodes are instructions and edges represent flow/anti/output dependences
- several heuristics to select among set of ready instructions

Algorithm
1) construct DAG in backwards prepass, ignoring transitive dependences
2) candidates = roots of DAG
3) while candidates remain
   a) pick instruction from candidates
   b) schedule instruction & update DAG
   c) add newly exposes instructions to candidates
Selecting the next instruction

Principle
• Select instruction most likely to cause interlocks with later instructions

Heuristics, in priority order
• will not interlock with prior instructions
• degree of interlock with successors in the DAG (select early, so more potential candidates)
• most successors (create most new candidates)
• longest path to leaves of DAG (critical path, aiming to balance progress on all paths)

\(O(n^2)\) in the worst case, for both building the DAG and scheduling instructions.
Tends to run in linear time.

Gives reasonable results for pipelined processors.
Less-reasonable processors

Pursuing performance, many modern processors issue multiple instructions per cycle.

(Very) Long Instruction Word (LIW or VLIW)
- Multiflow Trace, Intel i860, Tera, IA-64
- extend instruction set to pack multiple operations in an instruction
- may use NOPs if no instruction ready
- statically scheduled by compiler
- simplifies hardware, complicates compiler

Superscalar
- UltraSparc, R10K, Opteron
- retain instruction set
- window of $k$ instructions
- within window, hardware dynamically executes instructions in parallel where legal
- complicates hardware, but scheduling can still help (maybe)
Dealing with advanced processors

Within basic blocks
• minor extension to list scheduling
• schedule multiple instructions per cycle

Beyond basic blocks
• more candidates for schedule
• hide latency of branch instructions

Dealing with complex control flow
• trace scheduling
• percolation scheduling

Dealing with loops
• unrolling + list scheduling
• software pipelining
Instruction scheduling versus register allocation

There's a phase ordering problem between register allocation and instruction scheduling.

- If we do scheduling first, we might increase register pressure so much that we provoke spilling. And how do we schedule the spill code?
- If we do allocation first, the choice of register numbers can severely restrict the scheduler's ability to re-order instructions.

An early approach was to run scheduling twice, before & after, called prepass and postpass scheduling (what bad names!)
Combining scheduling and allocation

Assigning registers
- first fit - always choose lowest available register number, reduces total number of registers, increases dependences
- round-robin - cycle through all registers, typically uses more registers and reduces the number of dependences

Integrated prepass scheduling
- by Goodman & Hsu
- make a guess at global register pressure
- schedule basic block, working with register limits
- like list scheduling, but paying attention to register pressure

But how do they manage when we're going to have to spill?
Branch prediction

Will a conditional branch be taken?
• affects instruction scheduling
• execution penalty for incorrect guess

Prediction approaches
• hardware predictors
• profiling (feedback to compiler)
• compile-time heuristics

Static branch prediction
• no run-time information
• single prediction for all executions
• perfect prediction = 50 to 100% correct

Simple (target) heuristic
• predict conditional branch taken
• catches loop back edges
Branch prediction

Ball & Larus, "Branch Prediction for Free"

Loop branch heuristic, find forward, back, and exit edges, predict back edge, non-exit edge (covers about 55% of branches)

About 80% of remaining branches covered by
• Pointer heuristic, predict pointer not Null, pointers differ
• Call heuristic, predict branch not leading to call
• Opcode heuristic, predict > 0, floating-point value differ
• Return heuristic, predict branch not leading to return
• Store heuristic, predict branch leading to store of variable
• Loop heuristic, predict branch leading to loop header

Experimentally, they achieve a miss rate of less than 30% for non-loop branches, less than 20% overall.
Instruction scheduling for (inner) loops

There are a couple of approaches:
- loop unrolling, followed by list scheduling
- software pipelining

Loop unrolling
- create multiple copies of loop body
- more candidates for scheduling

Problems
- choosing degree of unrolling $k$
- pipeline hiccup every $k$ iterations
- increased compilation time
- hard on the instruction cache
Example

Here's a simple example, scheduling for a VLIW machine

```
//do i = 1, N
1   load
2   mult
3   mult
4   mult
5   mult
6   mult
    load
    load
    load
    load
    load
    load
```

// do i = 1, N, 3

Software pipelining

A technique applied manually or automatically
• overlaps iterations of loop
• select schedule for loop body
• initiate new iteration after every \( k \) cycles, before previous iterations are complete
• initiation interval is constant

Example

\[
\begin{array}{cccc}
i=0 & i=1 & i=2 & i=3 \\
\hline
1 & \text{load} & & \\
2 & \text{mult} & \text{load} & \\
3 & \text{mult} & \text{load} & \\
4 & \text{L: store mult load br L} & & \\
5 & \text{store mult load} & & \\
6 & \text{store mult} & & \\
7 & \text{store} & & \\
\end{array}
\]
Details

Variable renaming
  • need to renumber registers to avoid anti & output dependences
  • problem for both unrolling and software pipelining
  • need a copy for each unrolled/overlapped iteration

Example

```
a = a = a =
  = a = a = a
  a = b =
    = a = b
```
Details

Loop-carried dependences

- dependences across loop iterations
- can create cycles in dependence graph
- recurrence = cycle of flow dependences

Examples

```
  do i = 1, N
    a = a + 1
  enddo

  do i = 1, N
    a[i] = a[i - 2]
  enddo
```
Pipeline initiation interval

Determines the rate that iterations are executed.
- smaller interval implies higher throughput
- constrained by dependences and resources

Lower bound on interval
- \( k \geq \) cycles needed for recurrences
- \( k \geq \) ceiling(iteration resources / total resources)

Have to explore to find minimum initiation interval
Conclusion

Work on scheduling pushes us to learn implementation details; can't simply know the ISA.

List scheduling is reasonably simple and effective.
- Gibbons & Muchnick is a good starting point

Software pipelining can be really effective, either by hand or as a compiler transformation.
- There are many techniques for compiler use; modulo scheduling is an industrial-strength approach.

Interactions with register allocation are annoying. Usually software pipelining arranges it's own registers.