Instruction Scheduling

Reorder instructions to better fit target machine's pipeline

- fill control transfer delay slots
- avoid using result of multi-cycle operations too early
  - loads, floating point operations, ...
- schedule code for VLIW, superscalar machines
  - coordinate multiple instructions to fit available machine resources

Techniques:

- **list scheduling**, in a basic block
- **trace scheduling** across conditional branches
- **software pipelining**, across loop iterations

Loop unrolling often can help scheduling

Register allocation can hurt scheduling

List scheduling

[Gibbons & Muchnick 86]

Schedule a basic block...

- obeying data dependencies
- avoiding interlocks

Previous work: exponential, $O(n^4)$ algorithms

This work: $O(n^2)$ algorithm, simple

Pipeline model

Hazards considered:

- load followed by use of target of load
- store followed by a load
- load followed by
  ALU op or load/store with address calculation

```
r2 := r1 + 1
sp := sp - 12
*A := r0
r3 := *(sp+4)
r4 := *(sp+8)
sp := sp - 8
*sp := r2
r5 := *A
r4 := r0 + 1
```


Step 1: construct data dependence graph

Convert linear basic block into a DAG representing data dependencies

Loads & stores assumed to alias,
except that different offsets from common base reg (e.g. `sp`) do not alias

```
① r2 := r1 + 1
② sp := sp - 12
③ *A := r0
④ r3 := *(sp+4)
⑤ r4 := *(sp+8)
⑥ sp := sp - 8
⑦ *sp := r2
⑧ r5 := *A
⑨ r4 := r0 + 1
```

Step 2: traverse dependency graph, emitting code

Maintain set of **candidate nodes**
whose data dependence predecessors have been emitted

```
candidates := roots of DAG
while |candidates| > 0 do
    select best available candidate node
    emit it
    remove it from the DAG
    add any new root nodes to candidate set
```

Best node:

1. doesn't interlock with previous instruction, or
2. does interlock with an immediate successor node, or
3. has the most immediate successor nodes, or
4. is along the longest path to the leaves of the DAG

[Previous work used lookahead in DAG to guide choice
⇒ complex and slow (worst case)]

Results

[“Effectiveness of a Machine-Level Global Optimizer”,
Johnson & Miller, PLDI ’86]

Compiling small benchmark programs: 7% improvement