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

Compiling For Parallelism

Input: sequential code w/ loops
Output: parallel code

- vector machine
- multi-processor
- ...

Step 1: analyze loop nests to determine
interdependence of stmts

Step 2: transform loops to minimize dependences, overhead

Step 3: generate parallel code

- e.g. if loop iterations are independent, can run in parallel

Dependence

Flow/true dependence: read-after-write
s_1: `x := ...; ...` s_2: `... := x;`
$s_1 \ \delta \ s_2$
$s_1 \longrightarrow s_2$

Antidependence: write-after-read
s_1: `... := x; ...` s_2: `x := ...;`
$s_1 \ \delta \ s_2$
$s_1 \longrightarrow s_2$

Output dependence: write-after-write
s_1: `x := ...; ...` s_2: `x := ...;`
$s_1 \ \delta^o \ s_2$
$s_1 \longrightarrow s_2$

Control dependence: reference-after-test
s_1: `if (...) { ...` s_2: `...x... }`
$s_1 \ \delta^c \ s_2$
$s_1 \xrightarrow{c} s_2$

Easy to see in straight-line code
Harder to see in loops

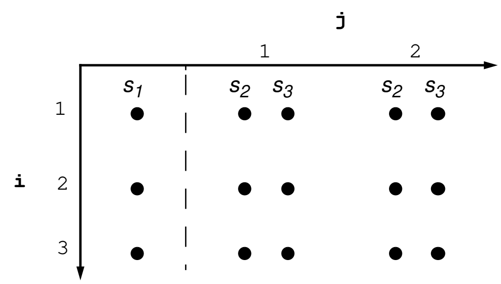
Iteration space

Think about loop dependences by "unrolling" loop

Represent each dynamic iteration in **iteration space**

- N+1-dimensional space if N-nested loops
- each point is an execution of a stmt in loop
- plot dependence between iteration space points

```
    for i := 1 to 3
s1:   a[i] := b[i]
      for j := 1 to 2
s2:     c[i,j] := a[i] + b[j]
s3:     d[i,j] := d[i,j+1] + d[i,j-1]
      end
    end
```



Summarizing dependences in iteration space

Summarize dependence between pts in iteration space

- $s_1\ \delta\ s_2$ iff some execution of s_1 δ some execution of s_2

Somewhat more precise summary: add direction to dependence

- $s_1\ \delta_=\ s_2$ iff each execution of s_1 δ an execution of s_2 *in the same iteration*
- $s_1\ \delta_<\ s_2$ iff each execution of s_1 δ an execution of s_2 *in a previous iteration*
  - **loop-carried dependence**
- $\delta_>$?

Even more precise summary: add distance to dependence

- $s_1\ \delta_i\ s_2$ iff each execution of s_1 δ an execution of s_2 *i iterations before*
- $\delta_= = \delta_0$
- $\delta_<$ summarizes all possible δ_n, $n>0$

Examples

```
for i := 2 to 4
  s1: a[i] := a[i-1] * 2
  s2: b[i-1] := a[i] + b[i]
end
```

```
for i := 2 to 4
  s1: a[i+1] := b[i]
  s2: x := a[i] * a[i-1]
  s3: b[i] := x
  s4: b[i+2] := x / 2
end
```


Loop nests

If multiple loops, use multiple direction subscripts

- e.g., $s_1\ \delta_{=,<}\ s_2$
- outermost iteration direction/distance first
- subscripts: **dependence vectors**

```
for i := 1 to 2
  for j := 2 to 5
    s1: a[i,j] := a[i,j-1] + b[i,j]
    s2: c[i,j] := a[i,j] + d[i+1,j-1]
    s3: d[i,j] := 1.0
  end
end
```

Legal dependence vectors are **lexicographically non-negative**

Testing array reference dependence

Simple example: GCD test

- many more sophisticated tests have been developed

Given two array references of the forms
`a[c*I+j]` and `a[d*I+k]`,
where `c`,`d`,`j`,`k` are constants and `I` is a loop index var

For dependence to exist then there must exist i_1, i_2 s.t.
$c*i_1+j = d*i_2+k$

Can only exist if GCD(`c`,`d`) divides `k-j` evenly

- if doesn't divide, then references independent

Can improve test by taking loop bounds into account

Compiling for a multiprocessor

Example target architecture:

- lots of powerful processors
- costly communication, synchronization between processors

Exploit hardware best by creating large independent tasks

For loops,
want loops with many independent iterations
with long-running bodies of equal duration

Example

```
for i := 1 to N
  s1: a[i,1] := 1.0
  for j := 2 to N
    s2: a[i,j] := a[i,j-1] + 1
    s3: b[i] := b[i] + a[i,j]
  end
end
```


Another example

```
for i := 2 to N
  for j := 1 to N
    s1: a[i-1,j] := a[i,j] + b[i,j]
    s2: b[i,j] := b[i-1,j] * 2
  end
end
```

Loop interchange

Sometimes can reorder loop nest to eliminate loop-carried dependences on outermost loop

When is this legal?
⇒ when data flow relationships are not changed

A test:

- reorder the entries in the dependences' distance vectors
- check that they remain lexicographically non-negative

Example, revisited

```
for i := 2 to N
  for j := 1 to N
    s1: a[i-1,j] := a[i,j] + b[i,j]
    s2: b[i,j] := b[i-1,j] * 2
  end
end
```


Another example

```
for i := 1 to 3
  for j := 2 to 5
    s1: a[i+1,j-1] := a[i,j] + b[i,j]
  end
end
```